Outils Hadoop pour le BigData

Pierre Nerzic - pierre.nerzic@univ-rennes1.fr

février-mars 2018

Abstract

Il s'agit des transparents du cours mis sous une forme plus facilement imprimable et lisible.

Ces documents ne sont pas totalement libres de droits. Ce sont des supports de cours mis à votre disposition pour vos études sous la licence *Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International*.

Version du 24/05/2018 à 09:44

Table des matières

Semaine 1

Principes du « Map-Reduce »

Le cours de cette semaine présente les concepts suivants :

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

Figure 1: Logo Hadoop

1.1. Introduction

1.1.1. 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 ces transparents pour la liste en France, qui est très similaire et inclut la connaissance de Python en 13^{e} position.

1.1.2. 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

1.1.3. 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 2014 : 300 Po de données (300 millions de Go), 4 Po de nouvelles données par jour, Amazon : 1 Eo.
- **BigScience** : télescopes (1 Po/jour), CERN (500 To/jour, 140 Po de stockage), génome, environnement...

Les 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.

1.1.4. 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.

1.1.5. Un *Data Center*

Imaginez 5000 ordinateurs connectés entre eux ; c'est un *cluster* :

Voir la figure 2, page 14.

1.1.6. 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) :

Voir la figure 3, page 14.

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

1.1.7. 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.

Figure 2: Datacenter Google

Figure 3: Blade server

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*.

1.1.8. Hadoop ?

Figure 4: Logo 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.

1.2. Hadoop File System (HDFS)

1.2.1. 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.

1.2.2. 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 ».

1.2.3. 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.

1.2.4. 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
```

1.2.5. Comment fonctionne HDFS ?

Comme avec de nombreux systèmes de fichiers, 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 sont numérotés et chaque fichier sait quels blocs il occupe.

Les blocs d'un même fichier ne sont pas forcément tous sur la même machine. Les répartir sur plusieurs machines permet d'y accéder simultanément par plusieurs processus.

En plus, dans HDFS, il y a réplication des blocs sur plusieurs machines pour se prémunir contre les pannes. Chaque fichier se trouve donc en plusieurs exemplaires à différents endroits.

1.2.6. 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.

1.2.7. Un schéma des nodes HDFS

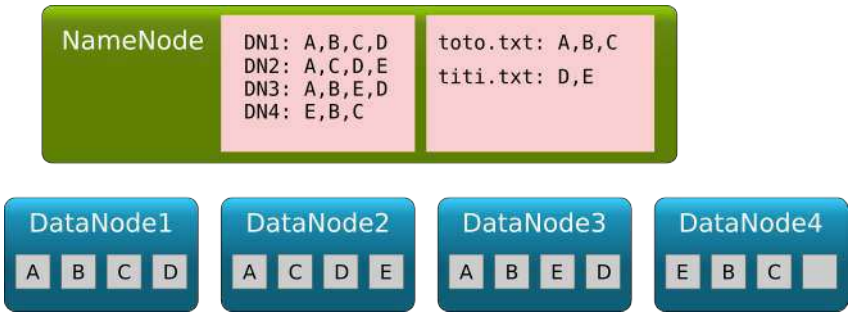

Figure 5: Maitre et esclaves HDFS

Les `datanodes` contiennent des blocs, le `namenode` sait où sont les fichiers : quels blocs et quels `datanodes`.

Consulter cette page pour des explications complètes.

1.2.8. Explications

Les `datanodes` contiennent des blocs (notés A,B,C...). 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.

1.2.9. 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.

1.2.10. API Java pour HDFS

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`

1.2.11. 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.

1.2.12. 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());
    }
  }
}
```

1.2.13. 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();
  }
}
```

1.2.14. 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 outStream = fs.create(nomcomplet);
      outStream.writeUTF("Bonjour tout le monde !");
      outStream.close();
    }
    fs.close();
  }
}
```

1.2.15. 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

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

Taper `make HDFSwrite` par exemple.

1.3. Algorithmes « Map-Reduce »

1.3.1. Principes

Le but est de recueillir une information synthétique à partir d'un jeu de données.

Exemples sur une liste d'articles définis par leur prix :

- calculer le montant total des ventes d'un article,
- calculer 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.

1.3.2. 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)
```

1.3.3. 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 :

```
pour chaque voiture, faire :
    prix = getPrix(voiture courante)
retourner max(prix rencontrés)
```

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.

1.3.4. Exemple en python

```python
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) )
```

1.3.5. 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 final[1].

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

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.

1.3.6. 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.

1.3.7. Parallélisation de Reduce

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

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

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.

1.3.8. Un schéma

Voir la figure 6, page 23.

1.4. YARN et MapReduce

1.4.1. Qu'est-ce que YARN ?

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

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

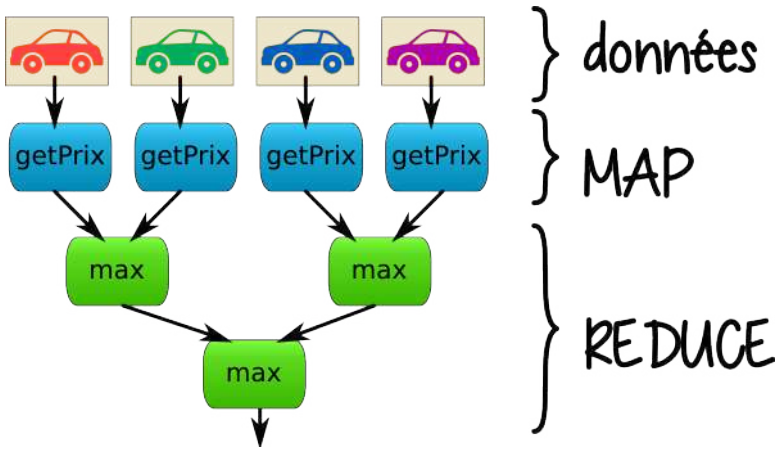

Figure 6: Arbre Map-Reduce

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.

1.4.2. 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.

1.4.3. 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.

1.4.4. 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.

1.4.5. Schéma de Map

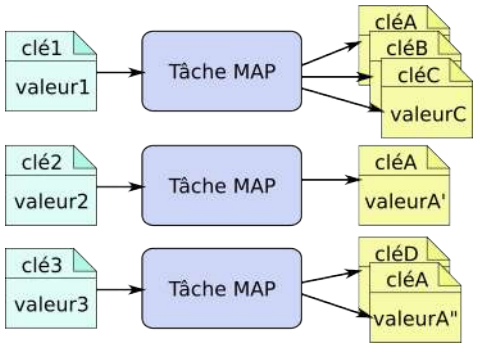

Figure 7: Tâches MAP et paires (clé, valeur)

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

1.4.6. 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.

1.4.7. Schéma de Reduce

Voir la figure 8, page 25.

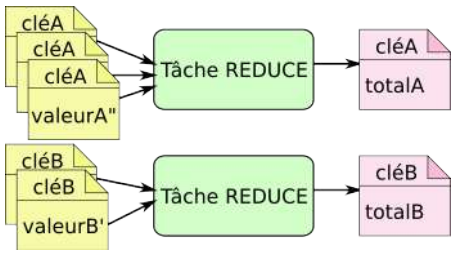

Figure 8: Tâches Reduce et paires (clé, valeur)

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.

1.4.8. 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)

1.4.9. 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.

1.4.10. 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.

1.4.11. Un schéma

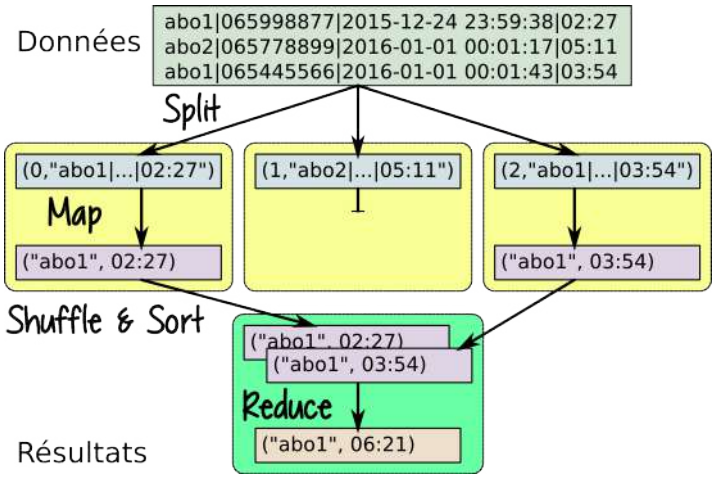

Figure 9: Étapes MapReduce

1.4.12. 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.

1.4.13. 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.

1.5. Mise en œuvre dans Hadoop

1.5.1. 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 patrons (*templates*) paramétrées par les types des clés et des valeurs.

1.5.2. 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);
    }
}
```

1.5.3. 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

1.5.4. 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();
```

1.5.5. 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.

1.5.6. 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.

1.5.7. 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);
    }
}
```

1.5.8. Explications

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

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

Une chose cruciale n'est pas du tout vérifiée par Java : il est obligatoire que les types des clés `TypCleI` et valeurs d'entrée `TypValI` du réducteur soient exactement les mêmes que les types des clés et valeurs de sortie du *mapper*. Si vous mettez des types différents, ça passera à la compilation mais plantera à l'exécution.

1.5.9. Squelette de `Traitement`

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

```
public class Traitement extends Configured implements Tool
{
    public int run(String[] args) throws Exception
    {
        /* voir transparent suivant */
    }

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

1.5.10. Squelette de `Traitement` (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(Traitement.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;
}
```

1.5.11. 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.

1.5.12. 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 Traitement 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
   ```

Semaine 2

Approfondissement sur MapReduce

Le cours de cette semaine présente davantage de détails sur les jobs MapReduce dans YARN :

- spécification des entrées
- spécification des paires (clé, valeurs)
- spécification des sorties
- traitement de certains fichiers
- MapReduce dans d'autres langages sur YARN

2.1. Jobs MapReduce

2.1.1. Création et lancement d'un `Job`

Revenons sur le lancement d'un job MapReduce :

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

2.1.2. Configuration d'un `Job`

Pour commencer avec des choses simples, il est possible de définir quelques options concernant un Job :

- `void setNumReduceTasks(int tasks)` définit le nombre de tâches de type *Reduce*. Par contre, il n'est pas possible de modifier le nombre de tâches *Map* parce que ça dépend uniquement des données à traiter.
- `void setPriority(JobPriority priorité)` fournir `JobPriority.HIGH`, `JobPriority.LOW`...

Exemple :

```
job.setNumReduceTasks(1);
job.setPriority(JobPriority.LOW);
```

2.1.3. Spécification des entrées

Les lignes suivantes spécifient ce qu'on veut traiter :

```
FileInputFormat.addInputPath(job, new Path(args[0]));
job.setInputFormatClass(TextInputFormat.class);
```

- La première ligne indique quels sont les fichiers HDFS à traiter,
- La seconde ligne indique le type de contenu de ces fichiers.

Voici davantage d'informations sur ces instructions.

2.1.4. Fichiers d'entrée

Cette instruction indique où prendre les fichiers à traiter :

```
FileInputFormat.addInputPath(job, new Path("NOMCOMPLET"));
```

C'est un appel à une méthode statique dans la classe `FileInputFormat`.

- Si le chemin fourni est un dossier, alors tous ses fichiers sont employés,
- Si les fichiers trouvés sont compressés (extensions `.gz`, `.bz2`, `.lzo`...), ils sont automatiquement décompressés.

Les sous-classes de `FileInputFormat` telles que `TextInputFormat` et `KeyValueTextInputFormat` sont également responsables de la séparation (*split*) des données en paires (clé,valeur).

2.1.5. Format des données d'entrée

Cette instruction spécifie le type des fichiers à lire et implicitement, les clés et les valeurs rencontrées :

```
job.setInputFormatClass(TextInputFormat.class);
```

Important: les types des clés et valeurs du `Mapper` doivent coïncider avec la classe indiquée pour le fichier.

Ici, la classe `TextInputFormat` est une sous-classe de `FileInputFormat<LongWritable,Text>`. Donc il faut écrire :

```
public class TraitementMapper
        extends Mapper<LongWritable,Text, TypCleI,TypValI>
{
    @Override
    public void map(LongWritable cleE, Text valE, ...
```

2.1.6. Autres formats d'entrée

Il existe d'autres formats d'entrée, comme `KeyValueTextInputFormat` qui est capable de lire des fichiers déjà au format (clé, valeur) :

- les lignes se finissent par un `'\n'` ou un `'\r'` (cause sûrement un pb avec des fichiers Windows qui ont les deux à la fois)
- chaque ligne est un couple (clé, valeur)
- c'est une tabulation `'\t'` qui sépare la clé de la valeur
- ces deux informations sont des `Text`

```
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

```
public class TraitementMapper
        extends Mapper<Text,Text, TypCleI,TypValI>
```

2.1.7. Changement du séparateur de `KeyValueTextInputFormat`

On peut changer le séparateur, par exemple une virgule :

```
Configuration conf = new Configuration();
conf.set(
    "mapreduce.input.keyvaluelinerecordreader.key.value.separator",
    ",");

Job job = new Job(conf);
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

2.1.8. Format des données intermédiaires

Les types des clés et valeurs sortant du *mapper* et allant au *reducer*, notés `TypCleI` et `TypValI` dans ce qui précède, sont définis par les instructions suivantes :

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
```

Elles forcent la définition du *mapper* et du *reducer* ainsi :

```
class TraitementMapper extends Mapper<..., Text, IntWritable>
class TraitementReducer extends Reducer<Text, IntWritable, ...>
```

Elles sont absolument obligatoires quand ce ne sont pas les types par défaut, `ClassCastException` lors du lancement du *reducer* sinon.

2.1.9. Format des données de sortie

Voici les instructions qui spécifient le format du fichier de sortie :

```
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);
```

Ce doivent être les types de sortie du *Reducer* :

```
class TraitementReducer
        extends Reducer<..., Text, DoubleWritable>
```

La classe `TextOutputFormat<K,V>` est paramétrée par les types des clés et des valeurs. Par défaut, ce sont tous deux des `Text`.

Il existe d'autres classes pour produire les données de sortie (voir plus loin), dont des écrivains sur mesure (voir en TP).

2.1.10. Fichiers de sortie

Les résultats du job sont enregistrés dans des fichiers situés dans le dossier indiqué par :

```
FileOutputFormat.setOutputPath(job, new Path("DOSSIER"));
```

YARN enregistre un fichier par Reducteur final. Leurs noms sont `part-r-00000`, `part-r-00001`,...

2.1.11. Post-traitement des résultats

Au lieu de récupérer un simple fichier, on peut afficher proprement le résultat final :

```
job.setOutputFormatClass(SequenceFileOutputFormat.class);
if (job.waitForCompletion(true)) {
    SequenceFile.Reader.Option fichier =
        SequenceFile.Reader.file(new Path(args[1],"part-r-00000"));
    SequenceFile.Reader reader =
        new SequenceFile.Reader(conf, fichier);
    IntWritable annee = new IntWritable();
    FloatWritable temperature = new FloatWritable();
    while (reader.next(annee, temperature)) {
        System.out.println(annee + " : " + temperature);
    }
    reader.close();
}
```

2.2. Types des clés et valeurs

2.2.1. Type `Writable`

Nous avons vu la semaine dernière qu'il fallait employer des `Writable` : `Text`, `IntWritable`, `FloatWritable`. L'interface `Writable` est une optimisation/simplification de l'interface `Serializable`

de Java. Celle de Java construit des structures plus lourdes que celle de Hadoop, parce qu'elle contiennent les noms des types des données, tandis que les `Writable` ne contiennent que les octets des données, et d'autre part les `Writable` sont modifiables (*mutables*).

Il existe différents types de `Writable` pour des collections. On va donner l'exemple d'un `Writable` spécifique dérivant d'un type tableau.

2.2.2. Classe `ArrayWritable`

Le type `ArrayWritable` représente des tableaux de `Writable` quelconques. Il est préférable de la sous-classer pour qu'elle contienne les données voulues :

```
public class IntArrayWritable extends ArrayWritable {
    public IntArrayWritable() { super(IntWritable.class); }
    public IntArrayWritable(int size) {
        super(IntWritable.class);
        IntWritable[] values = new IntWritable[size];
        for (int i=0; i<size; i++) values[i] = new IntWritable();
        set(values);
    }
    public IntWritable itemAt(int index) {
        Writable[] values = get();
        return (IntWritable)values[index];
    }
}
```

2.2.3. Emploi de cette classe

Voici un exemple presque fonctionnel qui montre comment créer et utiliser une telle structure :

```
public class TraitementMapper
    extends Mapper<LongWritable, Text, Text, IntArrayWritable>
{
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException
    {
        Text cle = new Text("ok");
        IntArrayWritable valeur = new IntArrayWritable(2);
        valeur.itemAt(0).set(123);
        valeur.itemAt(1).set(value.getLength());
        context.write(cle, valeur);
    }
}
```

2.2.4. Méthodes supplémentaires

Vous pouvez rajouter vos propres méthodes à cette classe. Par exemple pour additionner un autre `IntArrayWritable` à `this` :

```
public void add(IntArrayWritable autre)
{
    // récupérer les valeurs
    Writable[] values = this.get();
    Writable[] autres = autre.get();

    // this = this + autre
    for (int i=0; i<values.length; i++) {
        IntWritable val = (IntWritable)values[i];
        IntWritable aut = (IntWritable)autres[i];
        val.set(val.get() + aut.get());
    }
}
```

2.2.5. Interface `Writable`

L'interface `Writable` gère des contenus transmis entre un *mapper* et un *reducer*. Pour l'implémenter, il suffit de deux méthodes :

- `public void write(DataOutput sortie)` : elle écrit des données sur `sortie`,
- `public void readFields(DataInput entree)` : vous devez extraire les mêmes données et dans le même ordre.

Les deux classes `DataInput` et `DataOutput` sont des sortes de flots binaires (*binary stream*), comme des fichiers. Ils possèdent toutes les méthodes pour lire/écrire les types courants :

- `DataInput` : `readBoolean()`, `readInt()`, `readFloat()`, `readLine()`, etc.
- `DataOutput` : `writeBoolean(b)`, `writeInt(i)`, `writeFloat(f)`, `writeLine(s)`...

2.2.6. Exemple d'un `Writable`

Voici un exemple pour calculer une moyenne dans le *reducer* :

```
public class Moyenne implements Writable
{
    private double total = 0.0;
    private long nombre = 0L;

    public void write(DataOutput sortie) throws IOException {
        sortie.writeDouble(total);
        sortie.writeLong(nombre);
    }

    public void readFields(DataInput entree) throws IOException {
        total  = entree.readDouble();
        nombre = entree.readLong();
    }
}
```

2.2.7. Méthodes supplémentaires

On peut lui ajouter des méthodes pour faciliter la programmation du *mapper* et du *reducer* :

```
    public void set(double valeur) {
        total = valeur;
        nombre = 1L;
    }

    public void add(Moyenne autre) {
        total  += autre.total;
        nombre += autre.nombre;
    }

    public double getMoyenne() {
        return total/nombre;
    }
```

2.2.8. Méthodes diverses mais nécessaires

Pour que ça fonctionne bien, il faut rajouter un constructeur sans paramètre ainsi qu'une méthode `toString()`.

```
    public Moyenne()
    {
        total = 0.0;
        nombre = 0L;
    }

    public String toString() {
        return "Moyenne(total="+total+", nombre="+nombre+")";
        // OU return "Moyenne("+getMoyenne()+")";
    }
```

2.2.9. Utilisation dans un *Mapper*

Voici comment on peut l'employer côté *mapper* :

```
public class MoyenneHauteurArbresMapper
        extends Mapper<LongWritable, Text, Text, Moyenne>
{
    @Override
    public void map(LongWritable cleE, Text valeurE, Context context) throws Exception
    {
        Arbre.fromLine(valeurE.toString());
        Text cleI = new Text(Arbre.getGenre());
        Moyenne valeurI = new Moyenne();
        valeurI.set(Arbre.getHauteur());
```

```
        context.write(cleI, valeurI);
    }
}
```

NB: il manque tout ce qui est exception et filtrage des lignes.

2.2.10. Utilisation dans un *Reducer*

Voici comment on peut l'employer côté *reducer* :

```
public class MoyenneHauteurArbresReducer
        extends Reducer<Text, Moyenne, Text, DoubleWritable>
{
    @Override
    public void reduce(Text cleI, Iterable<Moyenne> valeursI, Context context) throws Exce
    {
        // cumuler les totaux et nombres de valeursI
        Moyenne moyenne = new Moyenne();
        for (Moyenne moy : valeursI) {
            moyenne.add(moy);
        }
        valeurS = new DoubleWritable(moyenne.getMoyenne());
        context.write(cleI, valeurS);
    }
}
```

2.2.11. Configuration du *Driver*

Pour finir, voici le cœur du *driver* :

```
Configuration conf = this.getConf();
Job job = Job.getInstance(conf, "MoyenneHauteurArbres Job");
job.setJarByClass(MoyenneHauteurArbresDriver.class);
job.setMapperClass(MoyenneHauteurArbresMapper.class);
job.setReducerClass(MoyenneHauteurArbresReducer.class);
FileInputFormat.addInputPath(job, new Path("arbres.csv"));
job.setInputFormatClass(TextInputFormat.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Moyenne.class);
FileOutputFormat.setOutputPath(job, new Path("resultats"));
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(DoubleWritable.class);
boolean success = job.waitForCompletion(true);
```

2.3. Efficacité

2.3.1. Remarque importante sur l'efficacité

Il faut éviter toute allocation mémoire répétée comme :

```
for (int i=0; i<10000; i++) {
    IntWritable valeur = new IntWritable(i);
    ...
}
```

Il vaut mieux créer les objets hors de la boucle et utiliser leur modificateur ainsi :

```
IntWritable valeur = new IntWritable();
for (int i=0; i<10000; i++) {
    valeur.set(i);
    ...
}
```

C'est possible parce que les `Writable` sont réaffectables (*mutables*).

2.3.2. Allocation en dehors des méthodes

En poursuivant de la même manière, on enlève les allocations des méthodes :

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

2.3.3. Piège à éviter

N'oubliez pas que le *reducer* peut être relancé plusieurs fois :

```
public class TraitementReducer extends Reducer<...>
{
    private TypValI valS = new TypValS(...);
    private Moyenne moyenne = new Moyenne();
    @Override
    public void reduce(Text cleI, Iterable<Moyenne> valeursI, ...
    {
        // BUG : moyenne n'a pas été remise à zéro !!!
        for (Moyenne valeurI : valeursI) {
            moyenne.add(valeurI);
```

```
        }
        valeurS = new DoubleWritable(moyenne.getMoyenne());
        context.write(cleI, valeurS);
    }
}
```

2.4. Entre Map et Reduce

2.4.1. *Combiner*

Pour l'instant, nous avons vu deux sortes de tâches : *map* et *reduce*. Nous avons vu que les paires (clé,valeur) produites par les tâches *map* sont envoyées par une étape appelée *shuffle and sort* à travers le réseau vers les tâches *réduce* de manière à regrouper toutes les clés identiques sur la même machine.

Quand on traite des données volumineuses, ça peut devenir trop lent. Hadoop propose un troisième intervenant, entre *map* et *reduce* qui effectue un traitement local des paires produites par *map*. C'est le « *Combiner* ». Son travail est de faire une première étape de réduction de tout ce qui est produit sur une machine.

2.4.2. Schéma du *Combiner*

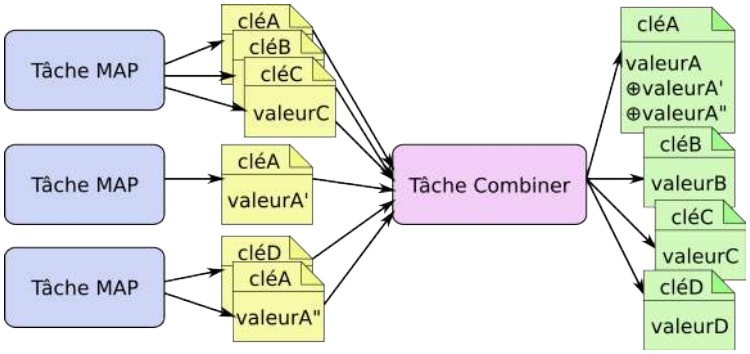

Figure 10: Cominer

Il traite des paires ayant la même clé sur la même machine que les tâches *Map*. Les paires qu'il émet sont envoyées aux *reducers*.

2.4.3. Cas d'emploi d'un *Combiner*

Le *combiner* permet de gagner du temps, non seulement en regroupant les valeurs présentes sur la même machine, mais en faisant un petit calcul au passage.

Par exemple, pour calculer la plus grande valeur d'un ensemble de données, il peut calculer le maximum de ce qu'il reçoit localement, et au lieu de produire une liste de valeurs à destination des réducteurs, il n'en produit qu'une seule.

De fait, dans certains cas, le *combiner* est identique au *reducer*. On peut utiliser la même classe pour les deux.

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

2.4.4. Cas de non-emploi d'un *Combiner*

Que pensez-vous de ceci ? On veut calculer la température moyenne par station météo depuis 1901. Les tâches *map* parcourent les relevés, extraient l'identifiant de la station météo et la température relevée, ceci pour chaque mesure. Les paires sont (idstation, temperature). Les tâches *reduce* calculent la moyenne par station météo.

Peut-on utiliser un *combiner* chargé de calculer la moyenne des température par station sur chaque machine contenant un *map* ?

On ne peut pas employer de *combiner* quand l'opérateur d'agrégation n'est pas commutatif ou pas associatif. Les opérateurs *somme*, *min* et *max* sont commutatifs et associatifs, mais pas le calcul d'une moyenne.

2.4.5. Différences entre un *Combiner* et un *Reducer*

1. Les paramètres d'entrée et de sortie du *Combiner* doivent être identiques à ceux de sortie du *Mapper*, tandis que les types des paramètres de sortie du *Reducer* peuvent être différents de ceux de son entrée.
2. On ne peut pas employer un *Combiner* quand la fonction n'est pas commutative et associative.
3. Les *Combiners* reçoivent leurs paires d'un seul *Mapper*, tandis que les *Reducers* reçoivent les paires de tous les *Combiners* et/ou tous les *Mappers*. Les *Combiners* ont une vue restreinte des données.
4. Hadoop n'est pas du tout obligé de lancer un *Combiner*, c'est seulement une optimisation locale. Il ne faut donc pas concevoir un algorithme « map-combine-reduce » dans lequel le *Combiner* jouerait un rôle spécifique.

2.4.6. Squelette de *Combiner*

Les *combiners* reprennent la même interface que les *reducers* sauf que les paires de sortie doivent être du même type que les paires d'entrée :

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

2.5. MapReduce dans d'autres langages

2.5.1. Présentation

Hadoop permet de programmer un Job MapReduce dans d'autres langages que Java : Ruby, Python, C++... En fait, il suffit que le langage permette de lire `stdin` et écrive ses résultats sur `stdout`.

- Le *Mapper* est un programme qui lit des lignes sur stdin, calcule ce qu'il veut, puis écrit des lignes au format `"%s\t%s\n"` (clé, valeur) sur la sortie.
- Entretemps, les lignes sont triées selon la clé, exactement comme le ferait la commande Unix `sort`
- Le *Reducer* doit lire des lignes au format `"%s\t%s\n"` (clé, valeur). Il doit d'abord séparer ces deux informations, puis traiter les lignes successives ayant la même clé. Ça vous rappellera la commande Unix `uniq -c`.

2.5.2. Exemple de *Mapper* en Python

Voici le « compteur de mots » programmé en python :

```
##!/usr/bin/python
## -*- coding: utf-8 -*-
import sys
## traiter chaque ligne de l'entrée standard
for ligne in sys.stdin:
    # couper en mots et traiter chacun d'eux
    for mot in ligne.split():
        # trivialement: ce mot est en 1 exemplaire
        paire = (mot, 1)
        # écrire la paire
        print '%s\t%s' % paire
```

Chaque ligne est découpée en mots ; chacun est écrit en tant que clé, avec la valeur 1 sur stdout.

2.5.3. Algorithme du réducteur

C'est un peu plus compliqué. C'est de l'algorithmique. Vous allez recevoir N lignes composées de paires (clé,valeur). Vous devez accumuler (somme, moyenne, min, max...) les valeurs correspondant à des clés identiques. Vous savez que les clés sont triées dans l'ordre, donc des lignes successives auront la même clé, sauf quand on change de clé.

Le parcours se fait ligne par ligne. Il faut donc mémoriser la clé de la ligne précédente ainsi que l'accumulation des valeurs de cette clé. Quand la ligne courante a la même clé que la précédente, on met à jour le cumul. Sinon, on affiche le cumul (et sa clé) sur la sortie et on le remet à zéro.

À la fin, ne pas oublier d'afficher la dernière clé et le cumul.

2.5.4. Exemple de *Reducer* en Python

Voici le cœur du compteur de mots sans les premières lignes :

```
cle_prec,nombre_total = None,0
for ligne in sys.stdin:
    cle, valeur = ligne.split('\t', 1)
    if cle == cle_prec:
        nombre_total += int(valeur)
    else:
        if cle_prec != None:
            paire = (cle_prec, nombre_total)
            print '%s\t%s' % paire
        cle_prec = cle
        nombre_total = int(valeur)
if cle_prec != None:
    paire = (cle_prec, nombre_total)
    print '%s\t%s' % paire
```

2.5.5. Lancement de ce Job

Pour lancer un tel job, il faut

- placer les deux scripts `mapper.py` et `reducer.py` sur HDFS
- taper la commande complexe suivante :

```
yarn jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
    -files mapper.py,reducer.py \
    -mapper mapper.py  -reducer reducer.py \
    -input livres  -output sortie
```

Il est souhaitable d'en faire un script ou un `Makefile`.

Il est intéressant de voir qu'on peut le lancer dans un tube Unix, mais en mode séquentiel :

```
cat data | mapper.py | sort | reducer.py
```

Semaine 3

Étude de cas MapReduce

Le cours de cette semaine explique comment créer une application MapReduce pour YARN complète. Deux projets vont être expliqués. Ils concernent des statistiques sur des documents.

- calcul de la variance
- calcul de la valeur médiane

On va appliquer ça à la longueur des lignes de textes, mais ça pourrait être la durée de séjour sur une place de parking payante, la température mesurée à midi, le nombre de sangliers dans les forêts...

3.0.1. Application

Soient des données : des romans sous forme de fichiers texte. On s'intéresse à la longueur des lignes et on veut la variance...

```
 1897
DRACULA
by Bram Stoker
CHAPTER I.
JONATHAN HARKER'S JOURNAL.
(Kept in shorthand.)

3 May.  Bistriz.- Left Munich at 8:35 P.M., on 1st May, arriving at
Vienna early next morning; should have arrived at 6:46, but train
was an hour late.  Buda-Pesth seems a wonderful place, from the glimpse
which I got of it from the train and the little I could walk through
the streets.  I feared to go very far from the station, as we had
arrived late and would start as near the correct time as possible.  The
impression I had was that we were leaving the West and entering the
East; the most western of splendid bridges over the Danube, which is
here of noble width and depth, took us among the traditions of Turkish
rule.
```

3.1. Calcul de la variance

3.1.1. Définition

La *variance* d'un ensemble de données permet de caractériser la dispersion des valeurs autour de la moyenne. La variance est le carré de l'*écart type*.

La variance V_x d'une population x_i est la moyenne des carrés des écarts des x_i à la moyenne m :

$$V_x = \frac{1}{n}\sum_i (x_i - m)^2$$

Vous pourrez trouver la notation σ^2 pour la variance et μ ou $\bar{x}$ pour la moyenne, selon la discipline (proba ou stats).

Le problème de cette équation, c'est qu'il faut d'abord parcourir les données pour extraire la moyenne.

3.1.2. Autre écriture

Il existe un autre algorithme, en 6 étapes :

1. Nombre de valeurs : $n = \sum_i 1$
2. Somme des valeurs : $S_x = \sum_i x_i$
3. Somme des carrés : $S_{x2} = \sum_i x_i^2$
4. Moyenne des valeurs : $M_x = S_x/n$
5. Moyenne des carrés : $M_{x2} = S_{x2}/n$
6. Variance : $V_x = M_{x2} - M_x * M_x$

Cet algorithme ne demande qu'un seul passage dans les données et il est parallélisable avec MapReduce.

3.1.3. Programmation séquentielle

On écrit l'algorithme ainsi :

1. Initialisation :

   ```
   n = Sx = Sx2 = 0
   ```

2. Pour chaque donnée x :

   ```
   n += 1 ; Sx += x ; Sx2 += x*x
   ```

3. Terminaison, calculer la variance par :

   ```
   Mx = Sx/n ; Mx2 = Sx2/n ; Vx = Mx2 - Mx*Mx
   ```

Il reste à transformer cette écriture en MapReduce.

3.1.4. Remarques

L'algorithme précédent est peu précis lorsque les nombres sont petits en valeur absolue. Il est alors préférable d'utiliser une variante dans laquelle on décale toutes les valeurs d'une même constante.

En effet, $V_{x+K} = V_x$.

On choisit alors $K = x_1$, c'est à dire le premier x des données. Cela donne quelque chose comme ça pour le 2e point page précédente :

```
si n==0 alors K = x
n += 1 ; Sx += (x-K) ; Sx2 += (x-K)*(x-K)
```

mais c'est difficilement applicable dans le cadre MapReduce, car il faut traiter à part l'une des données et transmettre la constante K à toutes les autres, ou alors choisir K arbitrairement.

3.1.5. Écriture MapReduce

Le principe est de ne faire qu'un seul passage à travers les données, lors de la phase *Map*. Cette étape doit collecter les trois informations n, S_x et S_{x2}. C'est l'étape *Reduce* qui calcule les moyennes et la variance.

- Map
  - extrait la valeur x_i à partir de la donnée courante
  - émet un triplet $(1,\ x_i,\ x_i^2)$ en tant que valeur, associé à une clé identique pour tous les triplets
- Combine
  - reçoit une liste de triplets associés à la même clé
  - additionne tous ces triplets pour obtenir $(n,\ S_x,\ S_{x2})$
- Reduce
  - même calculs que Combine, on obtient les sommes finales
  - calcule M_x, M_{x2} et V_x

3.1.6. Classe `Variance`

Comment transmettre les triplets `(n, Sx, Sx2)` entre les trois processus ?

- dériver la classe `ArrayWritable` : pas simple, et le pb, c'est que n est un entier, les autres sont des réels.
- définir notre classe, appelée `Variance` qui implémente `Writable` :
  - variables membres `n`, `Sx` et `Sx2`,
  - méthodes des `Writable` pour lire et écrire les variables,
  - constructeur : initialise les trois variables à zéro,
  - affectation de x pour faciliter *map*,
  - addition de deux `Variance` pour faciliter *combine* et *reduce*,
  - calcul des moyennes et de la variance pour faciliter *reduce*.

3.1.7. Classe `Variance` (entête)

Voici le début de la classe : 

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.Writable;

public class Variance implements Writable
{
    private long n;
    private double Sx;
    private double Sx2;
```

Notez les types : c'est du « Big Data », donc potentiellement, les données peuvent être énormes. Un `int` est limité à ±2 milliards.

3.1.8. Classe `Variance` (constructeur)

Voici le constructeur de la classe :

```
public Variance()
{
    clear();
}

public void clear()
{
    n = 0L;
    Sx = 0.0;
    Sx2 = 0.0;
}
```

Le suffixe `L` indique que c'est une constante `long`.

3.1.9. Classe `Variance` (méthodes `Writable`)

Un `Writable` doit implémenter ces deux méthodes :

```
public void write(DataOutput sortie) throws IOException
{
    sortie.writeLong(n);
    sortie.writeDouble(Sx);
    sortie.writeDouble(Sx2);
}

public void readFields(DataInput entree) throws IOException
{
    n = entree.readLong();
    Sx = entree.readDouble();
    Sx2 = entree.readDouble();
}
```

Il faut lire et écrire exactement la même chose dans le même ordre.

3.1.10. Classe `Variance` (méthode pour *map*)

Chaque *mapper* va avoir besoin de produire un triplet initialisé à partir de la valeur courante. La méthode suivante va lui faciliter la vie :

```
public void set(double valeur)
{
    n = 1L;
    Sx = valeur;
    Sx2 = valeur*valeur;
}
```

Il suffira de ceci dans le *mapper* pour chaque valeur x à traiter :

```
Variance valeurI = new Variance();
valeurI.set(x);
context.write(cleI, valeurI);
```

3.1.11. Classe `Variance` (méthode pour *combine*)

Les *combiners* et le *reducer* vont devoir additionner de nombreux triplets afin d'obtenir les totaux sur les trois champs. Voici une méthode pratique :

```
public void add(Variance autre)
{
    n += autre.n;
    Sx += autre.Sx;
    Sx2 += autre.Sx2;
}
```

Il leur suffira de faire ceci pour chaque liste de valeurs reçues :

```
Variance valeurS = new Variance();
for (Variance valeur: valeursI) {
    valeurS.add(valeur);
}
```

3.1.12. Classe `Variance` (méthode pour *reduce*)

Il est pratique de rajouter la méthode de calcul de la variance, plutôt que la coder dans le *reducer*, car toutes les variables sont sur place :

```
public double getVariance()
{
    double Mx = Sx / n;
    double Mx2 = Sx2 / n;
    return Mx2 - Mx*Mx;
}
```

Avec ça, cette classe peut facilement être intégrée dans un autre projet ou modifiée pour un autre calcul mathématique du même genre (cumul de valeurs).

3.1.13. Classe `Variance` (affichage)

Pour finir, on rajoute la méthode d'affichage :

```
public String toString()
{
    return "Variance(n="+n+", Sx="+Sx+", Sx²="+Sx2+")";
    // OU return "Variance("+getVariance()+")";
}
```

3.1.14. *Mapper* pour la variance

La classe *Mapper* reçoit un texte à traiter. Chaque *thread* va s'occuper d'une seule ligne. Le but est de calculer la variance des longueurs de ligne.

- En entrée du *Mapper*, il y a un texte, donc des paires (LongWritable, Text), parce que le *driver* configure le *job* par :

  `job.setInputFormatClass(TextInputFormat.class);`

- En sortie du *Mapper*, on aura des paires (Text, Variance), la clé étant identique pour toutes les paires.

On a donc l'entête suivant :

```
public class VarianceLongueurLignesMapper
        extends Mapper<LongWritable, Text, Text, Variance>
{
```

3.1.15. Classe `VarianceLongueurLignesMapper`

Voici le corps du *mapper*. Les allocations sont faites en dehors. La clé étant constante, elle est allouée et affectée une seule fois.

```
private Text cleI = new Text("Vx");
private Variance valeurI = new Variance();

@Override
public void map(LongWritable cleE, Text valeurE, Context context)
        throws IOException, InterruptedException
{
    valeurI.set( valeurE.getLength() );
    context.write(cleI, valeurI);
}
```

`valeurE` contient l'une des lignes du texte à traiter. `cleE` contient son offset, c'est à dire le numéro du premier octet de la ligne.

3.1.16. *Combiner*

Ce processus est optionnel de deux manières : d'une part on peut s'en passer, et d'autre part, même programmé, il n'est pas forcément lancé par YARN. S'il est lancé, il est associé à un *Mapper* et il tourne pour réduire les données venant d'une seule machine.

Son rôle est d'avancer le travail du *Reducer*. On va lui demander de calculer les sommes partielles. Comme il se trouve entre le *Mapper* et le *Reducer*, son entête est :

```
public class VarianceLongueurLignesCombiner
        extends Reducer<Text, Variance, Text, Variance>
{
```

On doit remettre les mêmes types en entrée et en sortie pour les clés et valeurs.

3.1.17. Classe `VarianceLongueurLignesCombiner`

Voici le corps du *combiner*. Les clés et valeurs de sortie ne sont pas allouées en dehors.

```
private Variance valeurS = new Variance();
public void reduce(Text cleI, Iterable<Variance> valeursI, Context context)
        throws IOException, InterruptedException
{
    Text cleS = cleI;
    valeurS.clear();
    for (Variance valeur : valeursI) {
        valeurS.add(valeur);
    }
    context.write(cleS, valeurS);
}
```

La clé d'entrée est recopiée en sortie et les valeurs d'entrée sont additionnées. On va retrouver ce même schéma dans le *reducer*.

3.1.18. Classe `VarianceLongueurLignesReducer`

Le *reducer* reçoit toutes les sommes partielles, des *mappers* et éventuellement des *combiners*. Il calcule les sommes finales et la valeur de la variance.

En entrée, il va recevoir des paires `(Text, Variance)` et en sortie, il ne produit qu'un nombre, la variance des longueurs de ligne, donc il produit une paire `(Text, DoubleWritable)`.

Voici la définition de la classe :

```
public class VarianceLongueurLignesReducer
        extends Reducer<Text, Variance, Text, DoubleWritable>
{
```

La clé de sortie sera encore la même clé. C'est comme ça quand on calcule une information synthétique sur la totalité des données. On aurait des clés différentes s'il fallait distinguer différentes variances.

3.1.19. `VarianceLongueurLignesReducer` (méthode `reduce`)

Voici le source du *reducer* :

```
private Variance total = new Variance();
private DoubleWritable valeurS = new DoubleWritable();

@Override
public void reduce(Text cleI, Iterable<Variance> valeursI, Context context)
        throws IOException, InterruptedException
{
    Text cleS = cleI;
    total.clear();
```

```
    for (Variance valeur : valeursI) {
        total.add(valeur);
    }
    valeurS.set( total.getVariance() );
    context.write(cleS, valeurS);
}
```

3.1.20. Programme principal

Le programme principal, appelé *Driver* crée un job YARN. Il définit les classes, les types des données et les fichiers concernés. Voici son point d'entrée avec la méthode `main` :

```
public class VarianceLongueurLignesDriver
    extends Configured implements Tool
{
    public static void main(String[] args) throws Exception
    {
        // préparer et lancer un job
        VarianceLongueurLignesDriver driver =
                new VarianceLongueurLignesDriver();
        int exitCode = ToolRunner.run(driver, args);
        System.exit(exitCode);
    }
```

Tout est dans la méthode `run` surchargée de la classe `Configured`.

3.1.21. *Driver*

Il a plusieurs choses à faire :

- Vérifier les paramètres. Ils sont dans le tableau `String[] args` passé en paramètre de la méthode.
- Créer le job YARN
- Définir les classes des *Mapper*, *Combiner* et *Reducer* afin que YARN sache quoi lancer
- Définir les types des clés et valeurs sortant du *Mapper*
- Définir les types des clés et valeurs sortant du *Reducer*
- Définir les fichiers ou dossiers à traiter : ce sont les paramètres du programme.
- Lancer le job.

C'est la fonction la plus longue de tout le logiciel.

3.1.22. Classe `VarianceLongueurLignesDriver` (méthode `run`)

Voici le début avec la vérification des paramètres et la création du job YARN :

```
@Override
public int run(String[] args) throws Exception
{
    // vérifier les paramètres
```

```
    if (args.length != 2) {
        System.err.println("fournir les dossiers d'entrée et de sortie");
        System.exit(-1);
    }

    // créer le job map-reduce
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "VarianceLongueurLignes");
    job.setJarByClass(VarianceLongueurLignesDriver.class);
```

3.1.23. Classe `VarianceLongueurLignesDriver` (classes)

Voici comment on définit les classes des traitements. On nomme les classes correspondant aux traitements :

```
    // définir les classes Mapper, Combiner et Reducer
    job.setMapperClass(VarianceLongueurLignesMapper.class);
    job.setCombinerClass(VarianceLongueurLignesCombiner.class);
    job.setReducerClass(VarianceLongueurLignesReducer.class);
```

C'est nécessaire car une fois compilé, rien ne dit à YARN quelle classe fait quoi. Le projet suivant montrera une situation où il y a deux MapReduce successifs, donc plusieurs classes pour le même type de travail.

3.1.24. Classe `VarianceLongueurLignesDriver` (entrée)

Voici comment on spécifie les données à traiter. La première instruction dit qu'on va traiter des textes (un fichier CSV, même compressé, est un fichier texte, un .mp3 est un fichier binaire).

La deuxième instruction indique où est le fichier à traiter. Si on fournit un nom de dossier, alors tous les fichiers de ce dossier seront traités. Et avec la troisième ligne, on indique d'aller aussi traiter tous les fichiers des sous-...dossiers.

```
    // définir les données d'entrée
    job.setInputFormatClass(TextInputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileInputFormat.setInputDirRecursive(job, true);
```

La classe `TextInputFormat` fait que les paires fournies au *mapper* seront des `(LongWritable, Text)`.

3.1.25. Classe `VarianceLongueurLignesDriver` (sorties)

Ensuite, on configure les types des clés intermédiaires et finales ainsi que le dossier de sortie :

```
    // sorties du mapper = entrées du reducer et du combiner
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Variance.class);
```

```
    // définir les données de sortie : dossier et types des paires
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);
```

Le *reducer* écrira le résultat, une paire `(Text, DoubleWritable)` dans un fichier appelé `part-r-00000` dans le dossier de sortie.

3.1.26. Classe `VarianceLongueurLignesDriver` (lancement)

Pour finir, on lance le job :

```
    // lancer le job et attendre sa fin
    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
}
```

La méthode `run` doit retourner un code d'erreur : 0=ok, 1=erreur. Or la méthode `waitForCompletion` retourne un booléen valant `true` si c'est ok, `false` si ça a planté.

Télécharger le projet complet VarianceLongueurLignes.tar.gz.

3.1.27. Commandes de compilation de cet ensemble

Il reste à voir comment on lance tout ça :

1. **Compilation des sources**. La variable `CLASSPATH` contient les archives jar nécessaires à la compilation, par exemple `~/lib/hadoop/hadoop-common-2.7.1.jar:~/lib/hadoop/hadoop-mapreduce-c`
   Les chemins sont séparés par un `:`

   ```
   javac -cp $CLASSPATH *.java
   ```

2. **Création d'une archive jar**. On doit dire quelle est la classe principale, celle qui contient `main()`.

   ```
   jar cfe projet.jar VarianceLongueurLignesDriver *.class
   ```

C'est plus compliqué si les fichiers source sont placés dans un dossier `src` et qu'il faut un dossier `bin` pour mettre les binaires, et aussi s'il y a des paquetages. Voir le `Makefile` des TP.

3.1.28. Commandes de lancement de cet ensemble

Voici maintenant l'exécution, une fois qu'on a l'archive `projet.jar`, à faire à chaque lancement :

1. **Suppression du dossier des résultats**

   ```
   hdfs dfs -rm -r -f resultats
   ```

2. **Lancement du job**. On donne les noms des dossiers.

   ```
   yarn jar projet.jar /share/livres resultats
   ```

3. **Affichage des messages**. Il faut connaître l'identifiant de l'application. Il est affiché dans les premières lignes suivant le lancement. Il est utile de rajouter `| more` à la commande.

   ```
   yarn logs -applicationId application_14579178155_003
   ```

4. **Affichage des résultats**

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

3.1.29. Bilan

- Écrire le calcul à la manière MapReduce
  L'écriture initiale ne le permet pas. Il faut trouver un moyen de ré-écrire la formule afin de paralléliser les calculs. Ici, ce sont les sommes partielles sur plusieurs données (1, x et x^2) qui peuvent être faites simultanément par les *combiners*. Par contre, cela demande des connaissances en mathématiques (Théorème de König-Huygens, wikipedia) qui sont hors compétence.
  Si on ne trouve pas les bonnes formules, on ne peut pas faire le travail en MapReduce.
- Définir la structure de données échangée entre les *mappers* et le *reducer*
  Si on ne peut pas faire avec les types prédéfinis simples, on doit implémenter la classe `Writable` et rajouter les méthodes utiles.

3.2. Calcul d'une médiane

3.2.1. Principe

Le programme précédent peut facilement être adapté au calcul de différentes moyennes : arithmétique, géométrique (exponentielle de la moyenne des logarithmes), harmonique (inverse de la moyenne des inverses), quadratique (racine de la moyenne des carrés).

La *médiane* est une autre sorte d'information statistique. C'est une valeur qui caractérise un ensemble de données. Elle est telle qu'il y a autant de données meilleures qu'elle, que de pires.

Par exemple, dans le sac (*bag* ou multiensemble) {58, 81, 36, 93, 3, 8, 64, 43, 3}, la médiane est 43. Il y a autant de valeurs plus petites qu'elle {36, 3, 8, 3} que de plus grandes {58, 81, 93, 64}.

La médiane n'est pas forcément au milieu des données. Par contre, elle l'est quand les données sont triées et en nombre impair.

3.2.2. Principe du calcul en MapReduce

On se propose de calculer la médiane des longueurs des lignes d'un ensemble de textes.

Dans un très grand corpus, de très nombreuses lignes ont la même longueur. Ainsi, ça conduit à un multi-ensemble où le même élément est présent de nombreuses fois. Par exemple, une ligne de longueur 67 est très commune.

Le calcul de la médiane reste le même : il faut trouver la longueur telle qu'il y a autant de lignes plus courtes que de lignes plus longues.

Pour éviter un tri de toutes les longueurs de lignes qui serait très coûteux, le principe est de travailler avec un histogramme des longueurs des lignes. Cet histogramme peut être calculé avec un MapReduce.

3.2.3. Histogramme des longueurs des lignes

Cette image montre le nombre de lignes de chaque longueur. Elle permet de visualiser la position de la médiane, qui est 64 (ici).

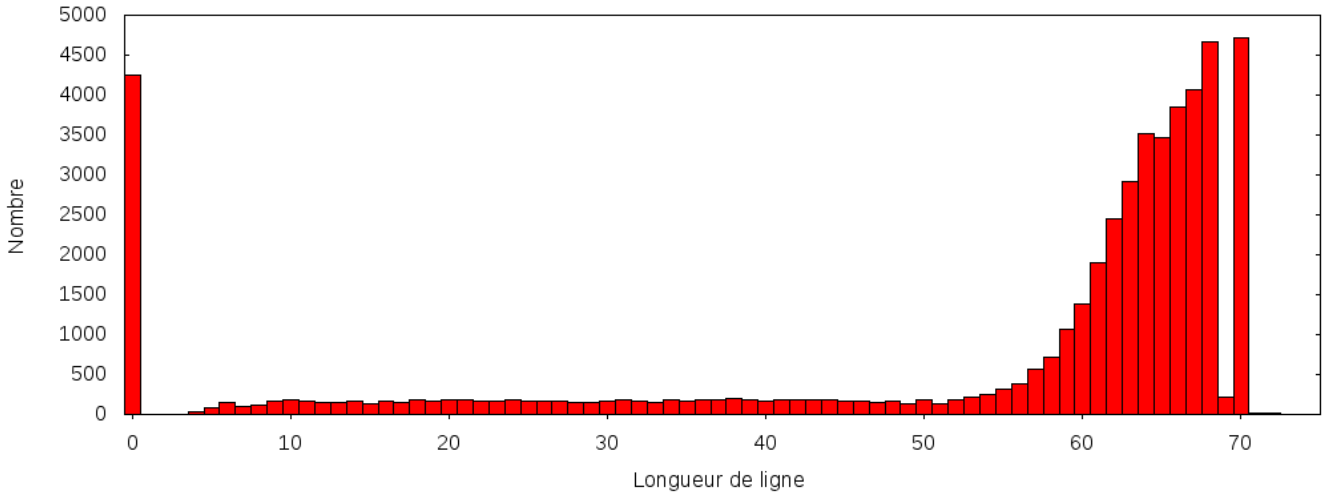

Figure 11: Histogramme des longueurs des ligne

La somme des nombres d'un côté, par exemple à gauche entre 0 et 63 est juste inférieure à la moitié du nombre total de lignes.

3.2.4. Données à calculer

En partant des textes organisés en lignes, on va calculer le nombre de lignes de chaque longueur, ainsi que le nombre de lignes total.

- Le nombre de lignes de chaque longueur par un premier MapReduce. Son travail consiste à produire l'histogramme. On va récupérer un tableau (longueur de ligne, nombre de lignes de cette longueur).
- Le nombre de lignes total par un second MapReduce appliqué à l'histogramme (c'est un choix purement pédagogique).

Ensuite, on va calculer la position de la médiane par un parcours de l'histogramme de gauche à droite en accumulant les nombres de lignes. Dès qu'on dépasse la moitié du nombre total, c'est qu'on est sur la médiane.

3.2.5. Premier MapReduce (le *mapper*)

Map1 reçoit des textes ligne par ligne. Chaque thread génère une paire `(length(ligne), 1)` :

```
public class MedianeLongueurLignesEtape1Mapper
    extends Mapper<LongWritable, Text, IntWritable, LongWritable>
{
    private final LongWritable valeurI = new LongWritable(1L);
    private IntWritable cleI = new IntWritable();
    @Override
    public void map(LongWritable cleE, Text valeurE, Context context) throws IOException,
```

```
    {
        cleI.set(valeurE.getLength());
        context.write(cleI, valeurI);
    }
}
```

3.2.6. Premier MapReduce (le *reducer*)

Reduce1 additionne les paires reçues de Map1 :

```
public class MedianeLongueurLignesEtape1Reducer
    extends Reducer<IntWritable, LongWritable, IntWritable, LongWritable>
{
    @Override
    public void reduce(IntWritable cleI, Iterable<LongWritable> valeursI, Context context)
    {
        IntWritable cleS = cleI;
        long nombre = 0L;
        for (LongWritable valeurI : valeursI) {
            nombre += valeurI.get();
        }
        LongWritable valeurS = new LongWritable(nombre);
        context.write(cleS, valeurS);
    }
}
```

3.2.7. Premier MapReduce (le *driver*)

Le driver construit un job MapReduce afin de traiter les textes. Il indique un dossier temporaire pour enregistrer les résultats.

Plusieurs MapReduce peuvent être construits et lancés successivement :

```
    Configuration conf = this.getConf();

    Job etape1 = Job.getInstance(conf, "MedianeLongueurLignes1");
    etape1.setJarByClass(MedianeLongueurLignesDriver.class);
    ...
    Job etape2 = Job.getInstance(conf, "MedianeLongueurLignes2");
    etape2.setJarByClass(MedianeLongueurLignesDriver.class);
    ...
    if (!etape1.waitForCompletion(true)) return 1;
    if (!etape2.waitForCompletion(true)) return 1;
```

3.2.8. Format des fichiers

La particularité, c'est que la sortie du premier MapReduce est mise dans un `SequenceFile` afin d'être facilement relue. C'est un format binaire contenant des paires (clé, valeur).

Voici la configuration de la sortie de l'étape 1 :

```
    etape1.setOutputFormatClass(SequenceFileOutputFormat.class);
    FileOutputFormat.setOutputPath(etape1, PathTMP1);
    etape1.setOutputKeyClass(IntWritable.class);
    etape1.setOutputValueClass(LongWritable.class);
```

Voici la configuration de l'entrée de l'étape 2 :

```
    etape2.setInputFormatClass(SequenceFileInputFormat.class);
    FileInputFormat.addInputPath(etape2, PathTMP1);
```

3.2.9. Second MapReduce (le *mapper*)

Map2 reçoit une paire `(longueur, nombre)` venant du `SequenceFile`. Il produit une paire `(0, nombre)` :

```
public class MedianeLongueurLignesEtape2Mapper
    extends Mapper<IntWritable, LongWritable, IntWritable, LongWritable>
{
    // même clé pour tout le monde
    private IntWritable cleI = new IntWritable(0);

    @Override
    public void map(IntWritable cleE, LongWritable valeurE, Context context)
            throws IOException, InterruptedException
    {
        context.write(cleI, valeurE);
    }
}
```

3.2.10. Second MapReduce (le *reducer*)

En fait, le second *reducer* est exactement le même que le premier : il calcule le total des valeurs par clé. Dans le second *map*, il n'y a qu'une seule clé.

On peut également l'utiliser directement comme *combiner* dans les deux jobs MapReduce.

3.2.11. Le post-traitement

- Le premier MapReduce produit un fichier (temporaire) contenant des paires (longueur de ligne, nombre de lignes de cette longueur). On a vu que c'était un `SequenceFile`.
- Le second MapReduce produit un fichier (temporaire) contenant une seule paire (0, nombre de lignes total). C'est un simple fichier texte sur HDFS.

Il y a maintenant deux choses à faire :

1. Récupérer le nombre de lignes total issu du second MapReduce
2. Récupérer et traiter l'histogramme issu du premier MapReduce

Ce qui est important, c'est que YARN fait trier les lignes sortant du *reducer* dans l'ordre croissant des clés. C'est à dire l'histogramme sera automatiquement dans l'ordre.

3.2.12. Récupérer le nombre de lignes total

Il faut relire le fichier `part-r-00000` issu du second MapReduce. On veut le 2e mot de la première ligne :

```
private long ReadTotal() throws IOException
{
    FSDataInputStream inStream =
            fs.open(new Path(TMP2, "part-r-00000"));
    try {
        InputStreamReader isr = new InputStreamReader(inStream);
        BufferedReader br = new BufferedReader(isr);
        String line = br.readLine();
        String[] mots = line.split("\t");
        return Integer.parseInt(mots[1]);
    } finally {
        inStream.close();
    }
}
```

3.2.13. Parcourir l'histogramme

Il faut relire le `SequenceFile` issu du premier MapReduce. C'est un peu plus complexe :

```
SequenceFile.Reader.Option fichier =
    SequenceFile.Reader.file(new Path(TMP1,"part-r-00000"));
SequenceFile.Reader reader = new SequenceFile.Reader(conf, fichier);
try {
    IntWritable longueur = new IntWritable();
    LongWritable nombre = new LongWritable();
    while (reader.next(longueur, nombre)) {
        // traiter le couple (longueur, nombre)
        ...
    }
    System.out.println("resultat : "+resultat);
} finally {
    reader.close();
}
```

3.2.14. Calculer la médiane

C'est de l'algorithmique standard. Dans la boucle précédente, on parcourt les couples (longueur, nombre) dans l'ordre croissant des longueurs. Il faut juste arrêter la boucle quand la somme des nombres vus jusque là dépasse la moitié du nombre total.

```
    long limite = ReadTotal() / 2;
    long cumul = 0L;
```

```
    while (reader.next(longueur, nombre)) {
        cumul += nombre.get();
        if (cumul >= limite) {
            break; // arrêter la boucle tout de suite
        }
    }
    return longueur.get();
```

On peut améliorer pour retourner une interpolation entre la longueur actuelle et sa précédente, selon le dépassement de la limite.

3.2.15. Bilan

Cet exemple a montré comment exploiter les résultats d'un ou plusieurs MapReduce dans un même programme. Plusieurs formats de fichiers peuvent être employés.

Télécharger le projet complet MedianeLongueurLignes.tar.gz.

Semaine 4

Spark

Le cours de cette semaine présente le système de programmation Spark, un autre mécanisme de Hadoop pour écrire des programmes de type MapReduce, nettement plus performants et plus polyvalents que ceux basés sur YARN. Nous programmerons sur Spark en Python, mais d'autres langages comme Scala sont possibles.

4.1. Introduction

4.1.1. Présentation de Spark

Spark est une API de programmation parallèle sur des données.

L'objet principal de Spark est le RDD : *Resilient Distributed Dataset*. C'est un dispositif pour traiter une collection de données par des algorithmes parallèles robustes. Un RDD ne contient pas vraiment de données, mais seulement un traitement.

Ce traitement n'est effectué que lorsque cela apparaît nécessaire. On appelle cela l'évaluation paresseuse. D'autre part, Spark fait en sorte que le traitement soit distribué sur le cluster, donc calculé rapidement, et n'échoue pas même si des machines tombent en panne.

4.1.2. Avantages de Spark

Spark permet d'écrire des traitements complexes composés de plusieurs phases *map-reduce*. On peut le faire également avec YARN, mais les données issues de chaque phase doivent être stockées sur HDFS, pour être réutilisées immédiatement après dans la phase suivante. Cela prend beaucoup de temps et d'espace.

Les jobs YARN sont assez longs à lancer et exécuter. Il y a des temps de latence considérables.

Au contraire, Spark utilise beaucoup mieux la mémoire centrale des machines du cluster et gère lui-même l'enchaînement des tâches.

Les traitements peuvent être écrits dans plusieurs langages : Scala, Java et Python. On utilisera ce dernier pour sa simplicité pédagogique et le fait que vous l'apprenez dans d'autres cours.

4.1.3. Premier exemple Spark

Soit un fichier de données de type CSV provenant de http://opendata.paris.fr décrivant des arbres remarquables à Paris. Chaque ligne décrit un arbre : position GPS, arrondissement, genre, espèce, famille, année de plantation, hauteur, circonférence, etc. Le séparateur est `';'`. La première ligne contient les titres.

On souhaite afficher l'année de plantation (champ n°6) de l'arbre le plus grand (champ n°7).

Avec des commandes Unix, ce traitement s'écrirait :

```
cat arbres.csv | cut -d';' -f6,7 | egrep -v 'HAUTEUR|;$' |\
    sort -t';' -k2 -n -r | head -n 1
```

Par contre, j'aurais apprécié que cut permette de changer l'ordre des champs, ça aurait facilité le classement.

4.1.4. Principe du traitement

Voyons comment faire la même chose avec Spark. Une fois que le fichier `arbres.csv` est placé sur HDFS, il faut :

1. séparer les champs de ce fichier.
2. extraire le 7e et 6e champs dans cet ordre – ce sont la hauteur de l'arbre et son année de plantation. On en fait une paire (clé, valeur). La clé est la hauteur de l'arbre, la valeur est son année de plantation.
3. éliminer la clé correspondant à la ligne de titre du fichier et les clés vides (hauteur inconnue).
4. convertir les clés en `float`
5. classer les paires selon la clé dans l'ordre décroissant.
6. afficher la première des paires. C'est le résultat voulu.

4.1.5. Programme pySpark

Voici le programme « pySpark » `arbres.py` :

```
##!/usr/bin/python
from pyspark import SparkConf, SparkContext
sc = SparkContext(conf=SparkConf().setAppName("arbres"))

arbres = sc.textFile("hdfs:/share/paris/arbres.csv")
tableau = arbres.map(lambda ligne: ligne.split(';'))
paires = tableau.map(lambda champs: (champs[6],champs[5]))
pairesok1 = paires.filter(
    lambda (hauteur,annee): hauteur!='' and hauteur!='HAUTEUR')
pairesok2 = pairesok1.map(
    lambda (hauteur,annee): (float(hauteur), annee))
classement = pairesok2.sortByKey(ascending=False)
print classement.first()
```

4.1.6. Remarques

Les deux premières instructions consistent à extraire les données du fichier. C'est d'assez bas niveau puisqu'on travaille au niveau des lignes et des caractères.

Dans *MapReduce* sur YARN, ces aspects avaient été isolés dans une classe `Arbres` qui masquait les détails et fournissait des méthodes pratiques, comme `getHauteur` et `getAnnee`.

Comparé aux programmes *MapReduce* en Java, Spark paraît plus rustique. Mais c'est sa rapidité, entre 10 et 100 fois supérieure à YARN qui le rend extrêmement intéressant.

On programme en Spark un peu comme dans le TP2 : la problématique est 1) d'arriver à construire des RDD contenant ce dont on a besoin et 2) d'écrire des fonctions de traitement.

4.1.7. Lancement

Spark offre plusieurs manières de lancer le programme, dont :

- Lancement sur un cluster de *Spark Workers*

  ```
  spark-submit --master spark://$(hostname -f):7077 \
  arbres.py
  ```

  L'option `--master` de cette commande indique à Spark qu'on doit faire appel au cluster de machines sur lesquelles tournent des *Spark Workers*. Ce sont des processus clients chargés de faire les calculs distribués pour Spark.
- Spark permet aussi de lancer l'exécution sur YARN :

  ```
  spark-submit --master yarn-cluster arbres.py
  ```

Ce sont les esclaves YARN qui exécutent le programme Spark.

4.1.8. Commentaires

Le programme précédent fait appel à des *lambda*. Ce sont des fonctions sans nom (anonymes).

Voici une fonction avec un nom employée dans un *map* Python :

```
def double(nombre):
    return nombre * 2

map(double, [1,2,3,4])
```

Cela peut s'écrire également avec une *lambda* :

```
map(lambda nombre: nombre * 2, [1,2,3,4])
```

La syntaxe est `lambda paramètres: expression`. Ça crée une fonction qui n'a pas de nom mais qu'on peut placer dans un *map*.

4.1.9. Fonction *lambda* ou fonction nommée ?

Faut-il employer une *lambda* ou bien une fonction nommée ?

- Complexité
  - Une *lambda* ne peut pas contenir un algorithme complexe. Elles sont limitées à une expression seulement.
  - Au contraire, une fonction peut contenir des boucles, des tests, des affectations à volonté
- Lisibilité
  - Les *lambda* sont beaucoup moins lisibles que les fonctions, impossibles à commenter, parfois cryptiques...
- Praticité
  - Les *lambda* sont très courtes et plus pratiques à écrire sur place, tandis que les fonctions doivent être définies ailleurs que là où on les emploie.

4.1.10. Fonction *lambda* ou fonction nommée ?

En conclusion :

- Les *lambda* sont intéressantes à connaître, plutôt pratiques
- On ne les emploiera que pour des expressions très simples, immédiates à comprendre.

Exemples pySpark (extrait de l'exemple initial) :

```
## chaque ligne est découpée en liste de mots
...map(lambda ligne: ligne.split(';'))
## on retourne un tuple composé des champs 6 et 5
...map(lambda champs: (champs[6],champs[5]))
## on ne garde que si clé n'est ni vide ni HAUTEUR
...filter(lambda (cle,val): cle!='' and cle!='HAUTEUR')
## on convertit la clé en float
...map(lambda (cle,val): (float(cle), val))
```

4.1.11. Fonction *lambda* ou fonction nommée ?

Le même exemple complet avec des fonctions nommées :

```
def separ(ligne):
    return ligne.split(';')
def hauteurannee(champs):
    return (champs[6],champs[5])
def garderok( (cle,val) ):
    return cle!='' and cle!='HAUTEUR'
def convfloat( (cle, val) ):
    return (float(cle), val)

tableau = arbres.map(separ)
paires = tableau.map(hauteurannee)
pairesok1 = paires.filter(garderok)
pairesok2 = pairesok1.map(convfloat)
```

Les traitements sont éloignés de leur définition.

4.1.12. Dernière remarque sur les fonctions

Spark fait tourner les fonctions sur des machines différentes afin d'accélérer le traitement global. Il ne faut donc surtout pas affecter des variables globales dans les fonctions — elles ne pourront pas être transmises d'une machine à l'autre.

Chaque fonction doit être autonome, isolée. Donc ne pas faire :

```
total = 0
def cumuler(champs):
    global total
```

```
    total += float(champ[6])
    return champ[5]

annees = tableau.map(cumuler)
```

Il y a quand même des variables globales dans Spark, mais on n'en parlera pas dans ce cours.

4.2. Éléments de l'API Spark

4.2.1. Principes

Spark est puissant car il repose sur des principes peu nombreux et simples. Consulter la documentation.

- Données :
  - **RDD** : ils représentent des données distribuées modifiées par une *transformation*, par exemple un *map* ou un *filter*.
  - Variables partagées entre des traitements et distribuées sur le cluster de machines.
- Calculs :
  - **Transformations** : ce sont des fonctions (au sens mathématique) du type : RDD ← *transformation*(RDD). Elles créent un nouveau RDD à partir d'un existant.
  - **Actions** : ce sont des fonctions qui permettent d'extraire des informations des RDD, par exemple les afficher sur l'écran ou les enregistrer dans un fichier.

4.2.2. Début d'un programme

Un programme pySpark doit commencer par ceci :

```
##!/usr/bin/python
from pyspark import SparkConf, SparkContext

nomappli = "essai1"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)
```

`sc` représente le contexte Spark. C'est un objet qui possède plusieurs méthodes dont celles qui créent des RDD.

Pour lancer le programme, faire : `spark-submit essai1.py`

4.2.3. RDD

Un RDD est une collection de données abstraite, résultant de la transformation d'un autre RDD ou d'une création à partir de données existantes. Un RDD est distribué, c'est à dire réparti sur plusieurs machines afin de paralléliser les traitements.

On peut créer un RDD de deux manières :

- Paralléliser une collection

Si votre programme contient des données itérables (tableau, liste...), elles peuvent devenir un RDD.

```
donnees = ['veau', 'vache', 'cochon', 'couvée']
RDD = sc.parallelize(donnees)
```

On le nomme « collection parallélisée ».

4.2.4. RDD (suite)

- Jeux de données externes

Spark peut utiliser de nombreuses sources de données Hadoop : HDFS, HBase... et il connait de nombreux types de fichiers : texte et les formats Hadoop tels que SequenceFile vu en semaine 2. Il y a d'autres formats de lecture. Consulter la documentation.

Voici comment lire un simple fichier texte ou CSV :

```
RDD = sc.textFile("hdfs:/share/data.txt")
```

Comme avec MapReduce, chaque ligne du fichier constitue un enregistrement. Les transformations appliquées sur le RDD traiteront chaque ligne séparément. Les lignes du fichier sont distribuées sur différentes machines pour un traitement parallèle.

4.2.5. Lire et écrire des *SequenceFile*

Certains traitements Spark font appel à la notion de paires (clé,valeur). C'est le cas de l'exemple initial. Les clés permettent par exemple de classer des valeurs dans un certain ordre.

Pour stocker efficacement ce genre de RDD, on emploie un *SequenceFile*. Voir la documentation Hadoop.

- Lecture d'un `SequenceFile` dans un RDD

Cette fonction lit les paires du fichier et crée un RDD :

```
RDD = sc.sequenceFile("hdfs:/share/data1.seq")
```

- Écriture d'un RDD dans un `SequenceFile`

Cette méthode enregistre les paires (clé, valeur) du RDD :

```
RDD.saveAsSequenceFile("hdfs:/share/data2.seq")
```

4.2.6. Actions

Avant de voir les transformations, voyons les actions. Ce sont des méthodes qui s'appliquent à un RDD pour retourner une valeur ou une collection.

- `liste = RDD.collect()` retourne le RDD sous forme d'une liste Python. Attention à la taille si c'est du BigData.
- `nombre = RDD.count()` retourne le nombre d'éléments

- `premier = RDD.first()` retourne le premier élément
- `premiers = RDD.take(n)` retourne les n premiers éléments.
  Note: il n'y a pas de méthode `last` pour retourner le ou les derniers éléments.
- `resultat = RDD.reduce(fonction)` applique une fonction d'agrégation (associative) du type `fn(a,b)→c`

```
grand = RDD.reduce(lambda a,b: max(a,b))
```

4.2.7. Transformations

Les RDD possèdent plusieurs méthodes qui ressemblent aux fonctions `map`, `filter`, *etc.* de Python.

En Python ordinaire, `map` est une fonction dont le premier paramètre est une *lambda* ou le nom d'une fonction, le second paramètre est la collection à traiter :

```
liste = [1,2,3,4]
doubles = map(lambda n: n*2, liste)
```

En pySpark, `map` est une méthode de la classe RDD, son seul paramètre est une *lambda* ou le nom d'une fonction :

```
liste = sc.parallelize([1,2,3,4])
doubles = liste.map(lambda n: n*2)
```

Les deux retournent les résultats, liste ou RDD.

4.2.8. Transformations de type *map*

Chacune de ces méthodes retourne un nouveau RDD à partir de celui qui est concerné (appelé `self` en Python).

- `RDD.map(fonction)` : chaque appel à la fonction doit retourner une valeur qui est mise dans le RDD sortant.

```
RDD = sc.parallelize([1,2,3,4])
print RDD.map(lambda n: n+1).collect()
```

- `RDD.filter(fonction)` : la fonction retourne un booléen. Il ne reste du RDD que les éléments pour lesquels la fonction retourne `True`.

```
RDD = sc.parallelize([1,2,3,4])
print RDD.filter(lambda n: (n%2)==0).collect()
```

4.2.9. Transformations de type *map* (suite)

- `RDD.flatMap(fonction)` : chaque appel à la fonction doit retourner une liste (vide ou pas) et toutes ces listes sont concaténées dans le RDD sortant.

```
RDD = sc.parallelize([0,1,2,3])
print RDD.flatMap(lambda n: [n]*n).collect()
```

En Python, la notation [*chose*]**n* produit la liste [*chose chose chose...*] contenant *n* éléments. Si n=0, ça crée une liste vide. Exemple `['ok']*3` vaut `['ok', 'ok', 'ok']`.

Donc, ici, les résultats de la lambda sont `[]`, `[1]`, `[2, 2]`, `[3, 3, 3]` et au retour du `flatMap` on aura `[1, 2, 2, 3, 3, 3]`

4.2.10. Transformations ensemblistes

Ces transformations regroupent deux RDD, `self` et celui passé en paramètre.

- `RDD.distinct()` : retourne un seul exemplaire de chaque élément.

```
RDD = sc.parallelize([1, 2, 3, 4, 6, 5, 4, 3])
print RDD.distinct().collect()
```

- `RDD1.union(RDD2)` : contrairement à son nom, ça retourne la concaténation et non pas l'union des deux RDD. Rajouter `distinct()` pour faire une vraie union.

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])
print RDD1.union(RDD2).collect()
print RDD1.union(RDD2).distinct().collect()
```

4.2.11. Transformations ensemblistes (suite)

- `RDD1.intersection(RDD2)` : retourne l'intersection des deux RDD.

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])
print RDD1.intersection(RDD2).collect()
```

4.2.12. Transformations sur des paires (clé, valeur)

Les transformations suivantes manipulent des RDD dont les éléments sont des paires (clé, valeur) (*(K,V)* en anglais)

- `RDD.groupByKey()` : retourne un RDD dont les éléments sont des paires (clé, liste des valeurs ayant cette clé dans le RDD concerné).
- `RDD.sortByKey(ascending)` : retourne un RDD dont les clés sont triées. Mettre `True` ou `False`.
- `RDD.reduceByKey(fonction)` : regroupe les valeurs ayant la même clé et leur applique la fonction `(a,b)→c`.

```
RDD = sc.parallelize([ (1,"paul"),(2,"anne"),
    (1,"emile"),(2,"marie"),(1,"victor") ])
print RDD.reduceByKey(lambda a,b: a+"-"+b).collect()
```

retourne `[(1, "paul-emile-victor"), (2, "anne-marie")]`

4.2.13. Transformations de type jointure

Spark permet de calculer des jointures entre RDD1={(K1,V1)...} et RDD2={(K2,V2)...} et partageant des clés K identiques.

- `RDD1.join(RDD2)` : retourne toutes les paires (K, (V1, V2)) lorsque V1 et V2 ont la même clé.
- `RDD1.leftOuterJoin(RDD2)` : retourne les paires (K, (V1, V2)) ou (K, (V1, None)) si (K,V2) manque dans RDD2
- `RDD1.rightOuterJoin(RDD2)` : retourne les paires (K, (V1, V2)) ou (K, (None, V2)) si (K,V1) manque dans RDD1
- `RDD1.fullOuterJoin(RDD2)` : retourne toutes les paires (K, (V1, V2)), (K, (V1, None)) ou (K, (None, V2))

```
RDD1 = sc.parallelize([ (1,"tintin"),(2,"asterix"),(3,"spirou") ])
RDD2 = sc.parallelize([ (1,1930),(2,1961),(1,1931),(4,1974) ])
print RDD1.join(RDD2).collect()
```

4.3. SparkSQL

4.3.1. Présentation

SparkSQL rajoute une couche simili-SQL au dessus des RDD de Spark. Ça s'appuie sur deux concepts :

DataFrames Ce sont des tables SparkSQL : des données sous forme de colonnes nommées. On peut les construire à partir de fichiers JSON, de RDD ou de tables Hive (voir le dernier cours).

RDDSchema C'est la définition de la structure d'un DataFrame. C'est la liste des colonnes et de leurs types. Un RDDSchema peut être défini à l'aide d'un fichier JSON.

Il y a des liens entre DataFrame et RDD. Les RDD ne sont que des données, des n-uplets bruts. Les DataFrames sont accompagnées d'un schéma.

4.3.2. Début d'un programme

Un programme pySparkSQL doit commencer par ceci :

```
##!/usr/bin/python
from pyspark import SparkConf, SparkContext, SQLContext
from pyspark.sql.functions import *

nomappli = "essai1"
config = SparkConf().setAppName(nomappli)
sc = SparkContext(conf=config)
sqlContext = SQLContext(sc)
```

`sqlContext` représente le contexte SparkSQL. C'est un objet qui possède plusieurs méthodes dont celles qui créent des DataFrames et celles qui permettent de lancer des requêtes SQL.

4.3.3. Créer un DataFrame

Il y a plusieurs manières de créer un DataFrame. Il faut à la fois fournir le schéma (noms et types des colonnes) et les données. L'une des méthodes simples consiste à utiliser un fichier JSON.

Un fichier JSON est pratique car il contient à la fois les données et le schéma, mais ça ne convient que pour de petites données.

Un fichier JSON contient la sérialisation d'une structure de données JavaScript. Pour les données qui nous intéressent, c'est simple. Chaque n-uplet est englobé par `{...}` ; les champs sont écrits `"nom":"valeur"`. Voici un exemple de trois n-uplets :

```
{"nom":"Paul"}
{"nom":"Émile", "age":30}
{"nom":"Victor", "age":19}
```

4.3.4. Créer un DataFrame à partir d'un fichier JSON

Voici comment créer un DataFrame :

```
df = sqlContext.read.json("fichier.json")
```

Elles retournent un DataFrame `df` contenant les données. Voir plus loin ce qu'on peut en faire.

A savoir qu'un DataFrame ainsi créé ne connaît pas les types des colonnes, seulement leurs noms.

4.3.5. Créer un DataFrame à partir d'un RDD

C'est plus compliqué car il faut indiquer le schéma. Un schéma est une liste de `StructField`. Chacun est un couple (nom, type).

```
## création d'un RDD sur le fichier personnes.csv
fichier = sc.textFile("hdfs:/tmp/personnes.csv")
tableau = fichier.map(lambda ligne: ligne.split(";"))
## définition du schéma
champ1 = StructField("nom", StringType)
champ2 = StructField("age", IntType)
schema = [champ1, champ2]
## création d'un DataFrame sur le RDD
personnes = sqlContext.createDataFrame(tableau, schema)
```

`personnes` est un DataFrame contenant les données et le schéma.

4.3.6. Extraction d'informations d'un DataFrame

Il est facile d'extraire une colonne d'un DataFrame :

```
colonneAge = personnes.age
```

Note: si une propriété est vide ou vaut `null`, python voit `None`.

La propriété `columns` retourne la liste des noms des colonnes :

```
print personnes.columns
```

La classe DataFrame possède de nombreuses méthodes qui seront présentées plus loin, page 71.

4.3.7. Donner un nom de table SQL à un DataFrame

Cela consiste à donner un nom désignant la future table SQL contenue dans le DataFrame. C'est une opération nécessaire pour exécuter des requêtes SQL. En effet, la variable `personnes` contenant le DataFrame ne connaît pas son propre nom.

```
personnes.registerTempTable("personnes")
```

Le DataFrame pourra être utilisé dans une requête SQL sous le nom `personnes`. Il est donc commode de remettre le même nom que le DataFrame.

NB: ce n'est qu'une table temporaire, elle disparaît à la fin du programme.

4.3.8. Exemple de requête SQL

Une fois que le DataFrame est rempli et nommé, on peut l'interroger. Il y a plusieurs moyens. Le premier est d'écrire directement une requête SQL.

```
resultat = sqlContext.sql("SELECT nom FROM personnes")

for nuplet in resultat.collect():
  print nuplet.nom
```

Le résultat de la méthode `sql` est un nouveau DataFrame contenant les n-uplets demandés (documentation). On les affiche à l'aide d'une simple boucle et d'un appel à `collect()` comme en pySpark.

Un autre moyen pour écrire des requêtes est d'appeler les méthodes de l'API.

4.4. API SparkSQL

4.4.1. Aperçu

L'API SparkSQL pour Python est très complète. Elle comprend plusieurs classes ayant chacune de nombreuses méthodes :

`DataFrame` représente une table de données relationnelles
`Column` représente une colonne d'un DataFrame
`Row` représente l'un des n-uplets d'un DataFrame

Ces classes permettent d'écrire une requête SQL autrement qu'en SQL, à l'aide d'appels de méthodes enchaînés.

4.4.2. Exemple de requête par l'API

Soit une table de clients (idclient, nom) et une table d'achats (idachat, idclient, montant). On veut afficher les noms des clients ayant fait au moins un achat d'un montant supérieur à 30.

En SQL, on l'écrirait :

```
SELECT DISTINCT nom FROM achats, clients
    WHERE achats.idclient = clients.idclient
    AND achats.montant > 30.0;
```

En pySparkSQL :

```
resultat = achats.filter(achats.montant > 30.0)          \
    .join(clients, clients.idclient == achats.idclient) \
    .select("nom")                                       \
    .distinct()
```

4.4.3. Classe DataFrame

C'est la classe principale. Elle définit des méthodes à appliquer aux tables. Il y en a quelques unes à connaître :

- `filter(condition)` retourne un nouveau DataFrame qui ne contient que les n-uplets qui satisfont la condition. Cette condition peut être écrite dans une chaîne SQL ou sous forme d'une condition Python.

```
resultat = achats.filter("montant > 30.0")
resultat = achats.filter(achats.montant > 30.0)
```

`Remarquer la différence de nommage des champs.`

4.4.4. Méthodes de DataFrame

- `count()` retourne le nombre de n-uplets du DataFrame concerné.
- `distinct()` retourne un nouveau DataFrame ne contenant que les n-uplets distincts
- `limit(n)` retourne un nouveau DataFrame ne contenant que les n premiers n-uplets
- `join(autre, condition, type)` fait une jointure entre `self` et autre sur la condition. Le type de jointure est une chaîne parmi `"inner"` (défaut), `"outer"`, `"left_outer"`, `"right_outer"` et `"semijoin"`
- `collect()` retourne le contenu de `self` sous forme d'une liste de `Row`. On peut l'utiliser pour un affichage final :

```
print achats.filter(achats.idclient == 1).collect()
```

4.4.5. Agrégation

- `groupBy(colonnes)` regroupe les n-uplets qui ont la même valeur pour les colonnes qui sont désignées par une chaîne SQL. Cette méthode retourne un objet appelé `GroupedData` sur lequel on peut appliquer les méthodes suivantes :
- `count()` : nombre d'éléments par groupe
- `avg(colonnes)` : moyenne des colonnes par groupe
- `max(colonnes)`, `min(colonnes)` : max et min des colonnes par groupe
- `sum(colonnes)` : addition des colonnes par groupe

```
tapc = achats.groupBy("idclient").sum("montant")
napc = achats.groupBy("idclient").count()
```

L'agrégation crée des colonnes appelées d'après la fonction : `"AVG(montant)"`, `"MAX(montant)"`, etc.

4.4.6. Classement

- `sort(colonnes)` classe les n-uplets de `self` selon les colonnes, dans l'ordre croissant. Si on spécifie la colonne par un nom pyspark (*table.champ*, on peut lui appliquer la *méthode* `desc()` pour classer dans l'ordre décroissant ; sinon, il faut employer la *fonction* `desc(colonnes)` pour classer dans l'ordre décroissant.

```
topa = achats.groupBy("idclient").sum("montant")    \
        .sort(desc("SUM(montant)")).first()
```

La méthode `first` retourne le premier n-uplet de `self`.

Semaine 5

Cassandra et SparkSQL

Le cours de cette semaine présente le SGBD Cassandra, conçu pour le stockage de mégadonnées sous forme de tables ressemblant à celle de SQL.

Figure 12: Cassandra

Ce qui caractérise Cassandra, c'est la distribution des n-uplets sur des machines organisées en anneau et un langage ressemblant à SQL pour les interroger.

Un plugin permet d'utiliser pySpark pour traiter les données Cassandra.

5.1. Cassandra

5.1.1. Présentation rapide

Cassandra est totalement indépendant de Hadoop. En général, ces deux-là s'excluent car chacun réquisitionne toute la mémoire et la capacité de calcul.

Cassandra gère lui-même la distribution et la réplication des données. Les requêtes distribuées sont extrêmement rapides.

Cassandra offre un langage d'interrogation appelé CQL très similaire à SQL, mais beaucoup plus limité et certains aspects sont très spécifiques.

Cassandra est issu d'un projet FaceBook, rendu libre en 2008 sous licence Apache. Une version professionnelle est développée par DataStax.

NB: tout ne sera pas expliqué dans ce cours, il faudrait cinq fois plus de temps.

5.1.2. Modèle de fonctionnement

Cassandra est **distribué**, c'est à dire que :

1. Les données sont disposées sur plusieurs machines, avec ou sans réplication (certaines machines ont des données en commun).

2. Les traitements sont effectués simultanément sur ces machines, selon les données qu'elles ont, par rapport à ce qu'il faut faire.

Cassandra est également **décentralisé**, c'est à dire qu'aucune machine n'a un rôle particulier.

- Dans Hadoop, les machines n'ont pas les mêmes rôles : namenode, datanode, nodemanager...
- Au contraire, dans Cassandra, les machines ont toutes le même rôle : stocker les données et calculer les requêtes. On peut contacter n'importe laquelle pour toute requête.

5.1.3. Structure du cluster et données

Les machines, appelées *nodes*, sont organisées en un « anneau » (*ring*) : chacune est reliée à une précédente et une suivante, le tout formant une boucle.

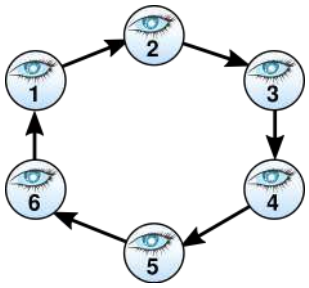

Figure 13: Structure d'anneau

L'anneau est construit automatiquement, par découverte de proche en proche à partir de quelques machines initiales.

5.1.4. Communication entre machines

Les communications sont gérées d'un mode appelé *Gossip* (rumeur) : les informations vont d'un nœud à l'autre.

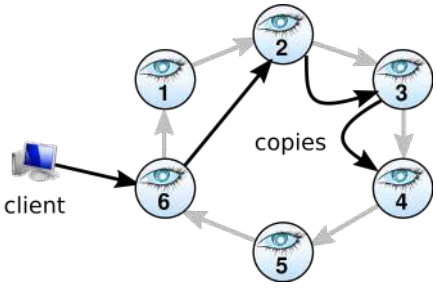

Figure 14: Structure d'anneau

Un client dépose des données sur l'un des nœuds, elles sont dupliquées et envoyées aux nœuds concernés (voir plus loin).

5.1.5. Cohérence des données

Les mises à jour des données sont donc effectuées de proche en proche, et de manière non synchronisée.

Ce modèle sans arbitre central pose un problème pour définir la cohérence des données (*consistency*). À un moment donné, il est possible que les machines n'aient pas toutes les mêmes valeurs dans les tables, le temps que les mises à jour se propagent.

5.1.6. Théorème CAP

En fait, c'est un problème de fond. Dans tout système de données, il est impossible d'avoir simultanément :

- la cohérence (**C**onsistency) : les clients lisent tous la même valeur au même moment,
- la disponibilité (**A**vailability) : les données peuvent être lues et écrites à tout moment,
- la résistance aux **P**artitions : les données peuvent être distribuées et résister à des pannes de communication entre machines.

Chaque système privilégie deux de ces critères. Par exemple les SGBD SQL ne garantissent que le couple CA, Cassandra privilégie AP, et HBase uniquement CP.

5.1.7. Théorème CAP

Concernant la consistance, Cassandra propose trois niveaux :

Consistance stricte Celle des SGBD relationnels. Toute lecture de données doit retourner la dernière valeurs écrite.

Consistance causale Si une donnée est modifiée plusieurs fois en séquence par le même agent, il faut obliger les lectures à retourner la dernière valeur. Par contre si elle est modifiée simultanément par des agents différents, alors il se peut que certains ne relisent pas la même valeur que les autres.

Consistance finale Les mises à jour sont faites progressivement, en arrière-plan. Les données ne sont cohérentes qu'au bout d'un certain temps.

Cela se choisit lors de la création des tables.

5.1.8. Modèle de données

Vocabulaire de Cassandra concernant les données :

Espace de clés C'est l'équivalent d'une base de données dans le monde SQL. Un *keyspace* peut contenir plusieurs tables.

Table Une table regroupe des colonnes. Comme en SQL, il y a une clé primaire et les colonnes peuvent être indexées pour des recherches plus rapides.

Colonne Elle contiennent les attributs des n-uplets. Ce sont des paires (clé, valeur), la clé est l'identifiant d'un n-uplet.

Partition C'est l'équivalent d'un n-uplet, une liste de (nom de colonne, valeur). Les n-uplets sont identifiés par la clé primaire.

5.1.9. Stockage des données

Chaque nœud Cassandra stocke une partie des n-uplets d'une table. Ce n'est pas fait n'importe comment :

- La clé primaire des n-uplet est transformée en un nombre appelé *token* par un hachage. Plusieurs algorithmes de hachage sont disponibles : murmur, md5, lexicographique.
- Chaque machine stocke un intervalle de ces hachages (de tel token à tel token) et les intervalles pris en charge se suivent en croissant, selon l'ordre de l'anneau. Ça forme une partition mathématique régulière des tokens.

Ainsi tous les tokens possibles sont quelque part dans l'anneau, et on sait très rapidement à quelle machine s'adresser quand on demande un n-uplet.

5.1.10. Réplication et redistribution des données

En général, pour la fiabilité, les données sont dupliquées N fois, par exemple 3 fois.

Les données d'une machine sont automatiquement recopiées sur les $N-1$ suivantes dans l'ordre de l'anneau, par le *gossip*.

Si une machine devient inaccessible, ou si on rajoute une nouvelle machine dans l'anneau, c'est assez compliqué. La partition des tokens est bouleversée. Normalement, il doit y avoir le même nombre de tokens sur chaque machine. Il faut donc redistribuer les tokens sur les machines voisines.

Pour faire tout cela plus facilement et ne pas surcharger le réseau, Cassandra définit des nœuds virtuels : une machine physique contient plusieurs machines virtuelles, en fonction de leur puissance de calcul, chacune stockant un intervalle de tokens.

5.1.11. Stockage des données

Initialement, les nouveaux n-uplets sont stockés en mémoire, dans une sorte de *cache*. Comme avec un système de fichiers, Cassandra enregistre un journal pour la fiabilité. Ça permet de modifier les données récentes très rapidement.

Les données stables sont placées dans une structure appelée SSTable *sorted string table*. C'est là qu'elles résident, une fois consolidées.

Lorsqu'on supprime un n-uplet, il est simplement marqué comme supprimé (*tombstone*). La suppression effective se fera ultérieurement lors d'un processus appelé compactage (*compaction*). Ce processus reconstruit de nouvelles SSTables à partir des anciennes.

5.1.12. Informations sur le cluster

La commande `nodetool` permet de gérer le cluster. On lui fournit un argument qui indique quoi faire :

- Pour afficher des informations générales sur le cluster :

```
nodetool info
nodetool describecluster
```

- Pour afficher l'état de chaque machine dans le cluster :

```
nodetool status
```

- Pour afficher la liste (énorme et incompréhensible) des intervalles de tokens machine par machine :

```
nodetool ring
```

5.1.13. Connexion au shell Cassandra CQL

Ce shell permet de manipuler et interroger la base de données dans un langage ressemblant à SQL.

Il faut fournir le nom ou le n°IP d'une des machines du cluster. Par exemple, à l'IUT, il suffira de taper :

```
prompt$ cqlsh master
```

Cela ouvre un shell CQL dans lequel on tape les requêtes (`^D` ou `exit` pour quitter).

La première à connaître est `HELP`. On peut lui ajouter un paramètre, par exemple `HELP DESCRIBE`.

5.1.14. Premières commandes

- Création d'un espace de clés (ensemble de tables)

```
CREATE KEYSPACE [IF NOT EXISTS] nomkeyspace
    WITH REPLICATION = {
        'class': 'SimpleStrategy',
        'replication_factor': 2
    };
```

La stratégie `SimpleStrategy` convient pour les clusters locaux. Ici, les données seront répliquées en 2 exemplaires.

- Suppression d'un keyspace et de tout son contenu

```
DROP KEYSPACE nomkeyspace;
```

5.1.15. Affichage d'informations

- Liste des keyspaces existants

```
DESCRIBE KEYSPACES;
```

- Structure d'un keyspace : cela affiche toutes les commandes servant à le reconstruire ainsi que ses tables

```
DESCRIBE KEYSPACE nomkeyspace;
```

- Sélection d'un keyspace pour travailler

```
USE nomkeyspace;
```

Au lieu de changer de keyspace, on peut aussi préfixer toutes les tables par `nomkeyspace.nomtable`

5.1.16. Premières commandes, suite

- Création d'une table

```
CREATE TABLE [IF NOT EXISTS] nomtable ( def colonnes );
```

On peut préfixer le nom de la table par `nomkeyspace.` si on est hors keyspace ou qu'on veut en désigner un autre.

Les définitions de colonnes sont comme en SQL : *nom type*. Les types sont `boolean`, `int`, `float`, `varchar`, `text`, `blob`, `timestamp`, etc.

Il y a des types spéciaux, comme `counter`, `list`, `set`, `map`, etc.

Voir la documentation.

5.1.17. Identification des n-uplets

Soit une table représentant des clients :

```
CREATE TABLE clients (
    idclient INT,        -- n° du client
    departement INT,     -- n° du département, ex: 22, 29, 35...
    nom TEXT, ...        -- coordonnées du client...
    PRIMARY KEY (...)
```

On a plusieurs possibilités pour la contrainte `PRIMARY KEY` :

- `PRIMARY KEY (idclient)` : les n-uplets sont identifiés par le n° client, c'est la « *row key* »
- `PRIMARY KEY (departement, idclient)` : la clé est *composite*, `departement` sert de « *clé de partition* ». Tous les clients du même département seront sur la même machine et ils seront classés par `idclient`.

5.1.18. Création d'un index secondaire

La clé est un index primaire. On rajoute un index secondaire par :

```
CREATE INDEX ON table ( nomcolonne );
```

L'index s'appelle généralement `table_nomcolonne_idx`.

Pour supprimer un index :

```
DROP INDEX table_nomcolonne_idx
```

Il n'est pas du tout recommandé de construire un index lorsque :

- les données sont extrêmement différentes (ex: une adresse mail)
- les données sont très peu différentes (ex: une année)

Il vaut mieux dénormaliser le schéma, construire une autre table ayant une clé primaire adaptée.

5.1.19. Insertion de données

C'est un peu comme en SQL et avec d'autres possibilités :

```
INSERT INTO nomtable (nomscolonnes...) VALUES (valeurs...);
INSERT INTO nomtable JSON 'données json';
```

Contrairement à SQL, les noms des colonnes concernées sont obligatoires, mais toutes les colonnes n'ont pas obligation d'y être, les absentes seront affectées avec `null`.

Exemples :

```
INSERT INTO clients
    (idclient, departement, nom) VALUES (1, 22, 'pierre');
INSERT INTO clients
    (idclient, nom) VALUES (2, 'paul');
INSERT INTO clients JSON '{"id":3, "nom":"jacques"}';
```

5.1.20. Insertion par fichier CSV

Il est possible de stocker les données dans un fichier CSV et de les injecter dans une table par :

```
COPY nomtable(nomscolonnes...) FROM 'fichier.csv'
    WITH DELIMITER=';' AND HEADER=TRUE;
```

On doit mettre les noms des colonnes dans le même ordre que le fichier CSV.

Une table peut être enregistrée dans un fichier CSV par :

```
COPY nomtable(nomscolonnes...) TO 'fichier.csv';
```

Le fichier sera créé/écrasé avec les données indiquées.

5.1.21. Sélection de données

C'est comme en SQL :

```
SELECT nomscolonnes... FROM table
    [WHERE condition]
    [LIMIT nombre]
    [ALLOW FILTERING];
```

Les colonnes peuvent utiliser la notation `*`.

Il y a une très forte limite sur la clause `WHERE` : elle doit sélectionner des n-uplets contigus dans un index. Donc c'est limité aux conditions sur les clés primaires ou secondaires. On peut ajouter les mots-clés `ALLOW FILTERING` pour rompre cette contrainte mais ce n'est pas recommandé car ça oblige à traiter tous les n-uplets.

5.1.22. Agrégation

On peut faire ceci comme en SQL :

```
SELECT nomscolonnes... FROM table
    ...
    GROUP BY clé de partition;
```

Les colonnes peuvent faire appel aux fonctions d'agrégation `COUNT`, `MIN`, `MAX`, `AVG`, `SUM` ainsi que des clauses `GROUP BY`. Mais dans ce cas, il est impératif que la colonne groupée soit une clé de partition (la première dans la clé primaire).

Vous voyez que le schéma des tables doit être conçu en fonction des requêtes et non pas en fonction des dépendances fonctionnelles entre les données. Cela implique de *dénormaliser* le schéma.

5.1.23. Autres requêtes

Oubliez les jointures, ça n'est pas possible. Cassandra, comme les autres systèmes, est destiné à stocker d'énormes volumes de données et à y accéder rapidement. Les possibilités de traitement sont très limitées. Pour en faire davantage, il faut programmer avec les API de Cassandra, ou avec SparkSQL, voir plus loin.

CQL offre d'autres possibilités : création de *triggers* et de fonctions, mais ça nous entraînerait trop loin.

5.1.24. Mise à jour de n-uplets

Comme avec SQL, on peut mettre à jour ou supprimer des valeurs :

```
UPDATE nomtable SET nomcolonnes=valeur WHERE condition;
DELETE FROM nomtable WHERE condition;
```

Il faut savoir que les valeurs supprimées sont marquées *mortes*, elles créent seulement une *tombstone*, et seront réellement supprimées un peu plus tard.

Il en va de même avec les mises à jour, elles sont associées à un *timestamp* qui permet de savoir laquelle est la plus récente sur un nœud.

5.2. Injection de données

5.2.1. Présentation

On s'intéresse au remplissage de tables Cassandra par des fichiers extérieurs (HDFS ou autres).

Ce n'est pas un problème avec une application neuve, remplissant ses tables au fur et à mesure. C'est un problème si on dispose déjà des données sous une autre forme et qu'on veut les placer dans Cassandra. Les requêtes `COPY FROM` sont très lentes.

La technique proposée consiste à créer directement des structures de données internes de Cassandra, des `SSTables`, puis d'utiliser un outil du SDK, `sstableloader`, voir cette page pour déplacer ces tables dans les dossiers internes. On appelle cela du *bulk loading*.

5.2.2. Étapes

On suppose que la table et son keyspace sont déjà créés.

1. Dans un premier temps, il faut programmer en Java un lecteur pour les données dont on dispose.
   a. D'abord, il faut préparer deux chaînes : le schéma de la table et la requête CQL d'insertion.
   b. On doit créer un écrivain de `SSTable`. C'est une instance de `CQLSSTableWriter` prenant le schéma et la requête.
   c. Ensuite, on lit chaque n-uplet des données et on le fournit à l'écrivain.

   Il y a une limite RAM au nombre de n-uplets pouvant être écrit ensemble, donc il faut périodiquement changer d'écrivain.
2. Ensuite, on lance `sstableloader` sur ces tables.

Consulter ce blog pour un exemple complet.

5.2.3. Définition du schéma et de la requête d'insertion

Voici comment définir les deux chaînes :

```
String schema = "CREATE TABLE ks.table (...)");
String insert = "INSERT INTO ks.table (...) VALUES (?,?,?...)";
```

Constatez que la requête d'insertion est une requête préparée. Chaque `?` sera remplacé par une valeur lors de la lecture des données, mais ça sera fait automatiquement par l'écrivain de `SSTable`.

5.2.4. Création de l'écrivain de `SSTable`

Ensuite, on initialise l'écrivain à l'aide d'un *builder* :

```
int num = 0;
String outDir = String.format("/tmp/cass/%03d/ks/table",num++);
CQLSSTableWriter.Builder builder = CQLSSTableWriter.builder();
builder.inDirectory(new File(outDir))
       .forTable(schema)
       .using(insert)
       .withPartitioner(new Murmur3Partitioner());
CQLSSTableWriter writer = builder.build();
```

Le dossier destination des `SSTable`, `outDir` est à définir là il y a beaucoup de place libre. D'autre part, son chemin est structuré ainsi : `dossier/n°/ks/table`. Cet écrivain devra être recréé en incrémentant le numéro tous les quelques dizaines de milliers de n-uplets.

5.2.5. Écriture de n-uplets

Voici le principe, extraire les colonnes puis les écrire :

```
// extraire les colonnes des données
String[] champs = ligne.split(";");
Integer col1 = Integer.parseInt(champs[0]);
Float   col2 = Float.parseFloat(champs[1]);
String  col3 = champs[2];

// écrire un n-uplet
writer.addRow(col1, col2, col3);
```

NB: les colonnes doivent être des objets correspondant au schéma. La méthode `addRow` remplit les paramètres de la requête préparée.

Il existe des variantes de `addRow`, consulter les sources, mais le problème principal est le nombre des allocations mémoire qu'elle fait à chaque appel.

5.2.6. Algorithme général

Voici le programme général :

```
import org.apache.cassandra.config.Config;

public static void main(String[] args)
{
    Config.setClientMode(true);
    // ouvrir le fichier de données
    BufferedReader br = new BufferedReader(new InputStreamReader(...));
    String ligne;  long numligne = 0;
    while ((ligne = br.readLine()) != null) {
        // créer l'écrivain si besoin
        if (numligne++ % 50000 == 0) {...}
        // extraire les colonnes des données...
        // écrire le n-uplet...
    }
}
```

5.2.7. Envoi des tables à Cassandra

Pour finir, il reste à placer les SSTables obtenues dans Cassandra. On peut utiliser ce petit script bash :

```
for d in /tmp/cass/*
do
    sstableloader -d master $d/ks/table
done
```

`master` étant le nom de l'une des machines du cluster. On doit retrouver le *keyspace* et le nom de la table dans le chemin fourni à `sstableloader`.

On va maintenant voir comment utiliser une table Cassandra avec Spark.

5.3. SparkSQL sur Cassandra

5.3.1. Présentation

Le cours précédent avait présenté les concepts de *DataFrame*. C'est à dire l'association entre un RDD et un schéma. Cette association est automatique quand on utilise Cassandra en tant que source de données.

Pour cela, il suffit d'importer un *plugin* établissant le lien entre pySpark et Cassandra. Il s'appelle `pyspark-cassandra`. Il fonctionne actuellement très bien, mais hélas, il n'est pas « officiel » et donc pourrait devenir obsolète.

Ensuite, on ouvre une table Cassandra et on obtient un DataFrame sur lequel on peut faire tout calcul Spark souhaité.

5.3.2. Début d'un script

Un script pySpark doit commencer par ces lignes :

```
##!/usr/bin/python
## -*- coding: utf-8 -*-

from pyspark import SparkConf
from pyspark_cassandra import CassandraSparkContext

## contexte d'exécution pour spark-submit
appName = "MonApplicationSparkCassandra"
conf = SparkConf()  \
       .setAppName(appName) \
       .setMaster("spark://master:7077") \
       .set("spark.cassandra.connection.host", "master")
csc = CassandraSparkContext(conf=conf)
```

5.3.3. Ouverture d'une table Cassandra

Comment faire plus simple que ceci pour ouvrir une table appelée `clients` dans un keyspace `ks` ?

```
clients = csc.cassandraTable("ks", "clients")
```

Toutes les requêtes Spark sont possibles, voici quelques exemples :

```
print clients.count()
print client.map(lambda client: client.age).filter(None).mean()
```

Les n-uplets sont également vus comme des dictionnaires Python, on peut accèder aux colonnes par la notation `nuplet['nomcol']`

5.3.4. Lancement d'un script

Le lancement est un peu plus compliqué, mais il suffit de faire un script shell :

```
spark-submit \
  --py-files /usr/lib/spark/jars/pyspark-cassandra-0.7.0.jar \
  script.py
```

Si on doit ajouter d'autres scripts Python, tels qu'une classe pour traiter les données, il faut les ajouter après le jar, avec une virgule pour séparer :

```
spark-submit \
  --py-files \
    /usr/lib/spark/jars/pyspark-cassandra-0.7.0.jar,client.py \
  script.py
```

Semaine 6

ElasticSearch et Kibana

Nous allons étudier une autre sorte de base de données, tournée vers l'indexation et la recherche rapide d'informations parmi des méga-données, ElasticSearch. Nous allons aussi voir son interface graphique Kibana.

PAS FINI, c'est pour début juin 2018

Semaine 7

Pig

Le cours de cette semaine présente le système Pig et son langage Pig Latin. Ce dernier est un langage de programmation de requêtes sur des fichiers HDFS qui se veut plus simple que Java pour écrire des jobs MapReduce. Pig sert à lancer les programmes Pig Latin dans l'environnement Hadoop.

7.1. Introduction

7.1.1. Présentation de Pig

Apache Pig est un logiciel initialement créé par Yahoo!. Il permet d'écrire des traitement utiles sur des données, sans subir la complexité de Java. Le but est de rendre Hadoop accessible à des non-informaticiens scientifiques : physiciens, statisticiens, mathématiciens...

Pig propose un langage de scripts appelé « Pig Latin ». Ce langage est qualifié de « Data Flow Language ». Ses instructions décrivent des traitements sur un flot de données. Conceptuellement, ça ressemble à un tube Unix ; chaque commande modifie le flot de données qui la traverse. Pig Latin permet également de construire des traitements beaucoup plus variés et non-linéaires.

Pig traduit les programmes Pig Latin en jobs MapReduce et intègre les résultats dans le flot.

7.1.2. Exemple de programme Pig

Ce programme affiche les 10 adultes les plus jeunes extraits d'un fichier csv contenant 3 colonnes : identifiant, nom et age.

```
personnes = LOAD 'personnes.csv' USING PigStorage(';')
        AS (userid:int, nom:chararray, age:int);
jeunesadultes = FILTER personnes BY age >= 18 AND age < 24;
classement = ORDER jeunesadultes BY age;
resultat = LIMIT classement 10;
DUMP resultat;
```

Pour l'exécuter : `pig programme.pig`. Ça lance un job MapReduce dans Hadoop. On peut aussi taper les instructions une par une dans le shell de Pig.

Le but de ce cours : comprendre ce script et en écrire d'autres.

7.1.3. Comparaison entre SQL et Pig Latin

Il y a quelques ressemblances apparentes entre SQL et Pig Latin. Il y a plusieurs mots clés en commun (`JOIN`, `ORDER`, `LIMIT`...) mais leur principe est différent :

- En SQL, on construit des requêtes qui décrivent les données à obtenir. On ne sait pas comment le moteur SQL va faire pour calculer le résultat. On sait seulement qu'en interne, la requête va être décomposée en boucles et en comparaisons sur les données et en utilisant au mieux les index.

- En Pig Latin, on construit des programmes qui contiennent des instructions. On décrit exactement comment le résultat doit être obtenu, quels calculs doivent être faits et dans quel ordre.

Également, Pig a été conçu pour les données incertaines de Hadoop, tandis que SQL tourne sur des SGBD parfaitement sains.

7.2. Langage Pig Latin

7.2.1. Structure d'un programme

Les commentaires sont placés entre `/*...*/` ou à partir de `--` et la fin de ligne.

Un programme Pig Latin est une succession d'instructions. Toutes doivent être terminées par un `;`

Comme dans SQL, il n'y a pas de notion de variables, ni de fonctions/procédures.

Le résultat de chaque instruction Pig est une collection de n-uplets. On l'appelle *relation*. On peut la voir comme une table de base de données.

Chaque instruction Pig prend une relation en entrée et produit une nouvelle relation en sortie.

```
sortie = INSTRUCTION entree PARAMETRES ...;
```

7.2.2. Exécution d'un programme

Lorsque vous lancez l'exécution d'un programme, Pig commence par l'analyser. Chaque instruction, si elle est syntaxiquement correcte, est rajoutée à une sorte de plan d'action, une succession de MapReduce, et c'est seulement à la fin du programme que ce plan d'action est exécuté en fonction de ce que vous demandez à la fin.

L'instruction `EXPLAIN relation` affiche le plan d'action prévu pour calculer la relation. C'est assez indigeste quand on n'est pas spécialiste.

7.2.3. Relations et alias

La syntaxe `nom = INSTRUCTION ... ;` définit un *alias*, c'est à dire un nom pour la relation créée par l'instruction. Ce nom étant généralement employé dans les instructions suivantes, c'est ça qui construit un flot de traitement.

```
nom1 = LOAD ... ;
nom2 = FILTER nom1 ... ;
nom3 = ORDER nom2 ... ;
nom4 = LIMIT nom3 ... ;
```

Le même alias peut être réutilisé dans des instructions différentes, ce qui crée des bifurcations dans le flot de traitement : séparations ou regroupements.

Il n'est pas recommandé de réaffecter le même alias.

7.2.4. Enchaînement des instructions

Pig permet soit d'enchaîner les instructions par le mécanisme des alias, soit par un appel imbriqué.

```
nom4 = LIMIT (ORDER (FILTER (LOAD ...) ...) ...) ... ;
```

Vous choisirez celui que vous trouvez le plus lisible.

Toutefois, les appels imbriqués ne permettent pas de faire facilement des séparations de traitement, au contraire des alias :

```
nom1 = LOAD ... ;
nom2 = FILTER nom1 ... ;
nom3 = FILTER nom1 ... ;
nom4 = JOIN nom2 ..., nom3 ;
```

7.2.5. Relations et types

Une *relation* est une collection ordonnée de n-uplets qui possèdent tous les mêmes champs. Voici les types possibles.

Les *types scalaires* sont :

- `int` et `long` pour les entiers, `float` et `double` pour les réels
- `chararray` pour des chaînes quelconques.
- `bytearray` pour des objets binaires quelconques

Il y a aussi trois *types complexes* :

- dictionnaires (*maps*) : `[nom#mickey, age#87]`
- n-uplets (*tuples*) de taille fixe : `(mickey, 87, hergé)`
- sacs (*bags*) = ensembles sans ordre de tuples : `{(mickey, 87), (asterix, 56), (tintin, 86)}`

7.2.6. Schéma d'une relation

La liste des champs d'une relation est appelé *schéma*. C'est un n-uplet. On l'écrit `(nom1:type1, nom2:type2, ...)`

Par exemple, une relation contenant des employés aura le schéma suivant :

```
(id:long, nom:chararray, prenom:chararray, photo:bytearray,
    ancienneté:int, salaire:float)
```

L'instruction `LOAD 'fichier.csv' AS schéma;` permet de lire un fichier CSV et d'en faire une relation d'après le schéma indiqué.

7.2.7. Schémas complexes (tuples)

Pig permet la création d'une relation basée sur un schéma incluant des données complexes. Soit un fichier contenant des segments 3D :

```
S1 ➪ (3,8,9) ➪ (4,5,6)
S2 ➪ (1,4,7) ➪ (3,7,5)
S3 ➪ (2,5,8) ➪ (9,5,8)
```

J'utilise le caractère '➪' pour représenter une tabulation.

Voici comment lire ce fichier :

```
segments = LOAD 'segments.csv' AS (
    nom:chararray,
    P1:tuple(x1:int, y1:int, z1:int),
    P2:tuple(x2:int, y2:int, z2:int));
DUMP segments;
```

7.2.8. Schémas complexes (bags)

On peut également lire des sacs, c'est à dire des ensembles de données de mêmes types mais en nombre quelconque :

```
L1 ➪ {(3,8,9),(4,5,6)}
L2 ➪ {(4,8,1),(6,3,7),(7,4,5),(5,2,9),(2,7,1)}
L3 ➪ {(4,3,5),(6,7,1),(3,1,7)}
```

Le schéma de ce fichier est :

```
(nom:chararray, Points:{tuple(x:int, y:int, z:int)})
```

Explications :

- Le deuxième champ du schéma est spécifié ainsi :
  « nom du champ »:{« type du contenu du sac »}
- Les données de ce champ doivent être au format
  {« liste de valeurs correspondant au type »}

7.2.9. Schémas complexes (maps)

Pour finir, avec les dictionnaires, voici le contenu du fichier `heros.csv` :

```
1 ➪ [nom#asterix,metier#guerrier]
2 ➪ [nom#tintin,metier#journaliste]
3 ➪ [nom#spirou,metier#groom]
```

On en fait une relation par :

```
heros = LOAD 'heros.csv' AS (id:int, infos:map[chararray])
```

Remarque : toutes ces constructions, tuple, map et bags peuvent être imbriquées, mais certaines combinaisons sont difficiles à spécifier.

7.2.10. Nommage des champs

Il y a deux syntaxes pour nommer les champs d'une relation. Soit on emploie leur nom en clair, soit on les désigne par leur position `$0` désignant le premier champ, `$1` le deuxième et ainsi de suite.

On emploie la seconde syntaxe quand les noms des champs ne sont pas connus ou qu'ils ont été générés dynamiquement.

Quand il y a ambiguïté sur la relation concernée, on préfixe le nom du champ par le nom de la relation : `relation.champ`

- Lorsqu'un champ est un *tuple*, ses éléments sont nommés `relation.champ.element`. Par exemple `segments.P1.z1`
- Pour un champ de type *map*, ses éléments sont nommés `relation.champ#element`. Par exemple `heros.infos#metier`
- Il n'y a pas de syntaxe pour l'accès aux champs de type *bag*.

7.3. Instructions Pig

7.3.1. Introduction

Il y a plusieurs catégories d'instructions : interaction avec les fichiers, filtrage, jointures...

Pour commencer, il y a également des instructions d'accès aux fichiers HDFS à l'intérieur de Pig. Ces commandes fonctionnent comme dans Unix ou comme le suggère leur nom et elles sont plutôt destinées à être tapées dans le shell de Pig.

- dossiers : `cd`, `ls`, `mkdir`, `rmf` (`rmf` = `rm -f -r`)
- fichiers : `cat`, `cp`, `copyFromLocal`, `copyToLocal`, `mv`, `rm`
- divers : `help`, `quit`, `clear`

7.3.2. Chargement et enregistrement de fichiers

- `LOAD 'fichier' USING PigStorage('sep') AS schema;`
  Charge le fichier qui doit être au format CSV avec des champs séparés par `sep` et en leur attribuant les noms et types du schéma. Des données qui ne correspondent pas restent vides.
  - Il n'est pas nécessaire de mettre la clause `USING` quand le séparateur est la tabulation.
  - NB: le fichier doit être présent dans HDFS ; rien ne signale l'erreur autrement que l'échec du programme entier.
- `STORE relation INTO 'fichier' USING PigStorage('sep');`
  Enregistre la relation dans le fichier, sous forme d'un fichier CSV séparé par `sep`, ex: `';'`, `':'`...

Dans les deux cas, si le fichier porte une extension `.gz`, `.bz2`, il est (dé)compressé automatiquement.

7.3.3. Affichage de relations

- `DUMP relation;`
  Lance le calcul MapReduce de la relation et affiche les résultats à l'écran. C'est seulement à ce stade que la relation est calculée.
- `SAMPLE relation;`
  Affiche quelques n-uplets choisis au hasard, une sorte d'échantillon de la relation.
- `DESCRIBE relation;`
  Affiche le schéma, c'est à dire les champs, noms et types, de la relation. C'est à utiliser dès qu'on a un doute sur le programme.

7.3.4. Instruction `ORDER`

Elle classe les n-uplets dans l'ordre croissant (ou décroissant si DESC) des champs indiqués

```
ORDER relation BY champ [ASC|DESC], ...
```

Ça ressemble totalement à ce qu'on fait avec SQL. Voir l'exemple du transparent suivant.

```
RANK relation BY champ [ASC|DESC], ...
```

Retourne une relation ayant un premier champ supplémentaire, le rang des n-uplets par rapport au critère indiqué.

7.3.5. Instruction `LIMIT`

Elle ne conserve de la relation que les N premiers n-uplets.

```
LIMIT relation N
```

On l'utilise en général avec `ORDER`. Par exemple, cette instruction affiche les 10 plus gros achats :

```
triparmontant = ORDER achats BY montant DESC;
meilleurs = LIMIT triparmontant 10;
DUMP meilleurs;
```

On peut aussi l'écrire de manière imbriquée :

```
DUMP LIMIT (ORDER achats BY montant DESC) 10;
```

7.3.6. Instruction FILTER

Elle sert à créer une relation ne contenant que les n-uplets qui vérifient une condition.

```
FILTER relation BY condition;
```

La condition :

- comparaisons : mêmes opérateurs qu'en C et en Java
- nullité (vide) d'un champ : `IS NULL`, `IS NOT NULL`
- connecteurs logiques : (mêmes opérateurs qu'en SQL) `AND`, `OR` et `NOT`

```
clients = LOAD 'clients.csv'
        AS (idclient:int, age:int, adresse:chararray);
tresvieuxclients = FILTER clients
        BY age > 1720 AND adresse IS NOT NULL;
```

7.3.7. Instruction DISTINCT

Elle sert à supprimer les n-uplets en double. Cela ressemble à la commande Unix `uniq` (sauf qu'ils n'ont pas besoin d'être dans l'ordre).

```
DISTINCT relation;
```

Note: la totalité des champs des tuples sont pris en compte pour éliminer les doublons.

Si on veut éliminer les doublons en se basant sur une partie des champs, alors il faut employer un `FOREACH`, voir plus loin.

7.3.8. Instruction FOREACH GENERATE

C'est une instruction qui peut être très complexe. Dans sa forme la plus simple, elle sert à générer une relation à partir d'une autre, par exemple faire une projection.

```
FOREACH relation GENERATE expr1 AS champ1, ...;
```

Crée une nouvelle relation contenant les champs indiqués. Ça peut être des champs de la relation fournie ou des valeurs calculées ; la clause `AS` permet de leur donner un nom.

Exemple, on génère des bons d'achats égaux à 5% du total des achats des clients :

```
bonsachat = FOREACH totalachatsparclient
    GENERATE idclient, montant*0.05 AS bon;
```

7.3.9. Énumération de champs

Lorsqu'il y a de nombreux champs dans la relation d'entrée et aussi dans celle qu'il faut générer, il serait pénible de tous les écrire. Pig propose une notation `champA .. champB` pour énumérer tous les champs compris entre les deux. Si champA est omis, alors ça part du premier ; si champB est omis, alors l'énumération va jusqu'au dernier champ.

Par exemple, une relation comprend 20 champs, on veut seulement retirer le 8^{e} :

```
relation_sans_champ8 = FOREACH relation_complexe
    GENERATE .. champ7 champ9 ..;
```

Attention, il faut écrire deux points bien espacés.

7.3.10. Instruction GROUP BY

L'instruction `GROUP relation BY champ` rassemble tous les tuples de la relation qui ont la même valeur pour le champ. Elle construit une nouvelle relation contenant des couples (champ, {tuples pour lesquels *champ* est le même}).

Soit une relation appelée `achats` (idachat, idclient, montant) :

```
1, 1, 12.50
2, 2, 21.75
3, 3, 56.25
4, 1, 34.00
5, 3,  3.30
```

`GROUP achats BY idclient` produit ceci :

```
(1, {(4,1,34.0), (1,1,12.5)})
(2, {(2,2,21.75)})
(3, {(3,3,56.25), (5,3,3.30)})
```

7.3.11. Remarque sur GROUP BY

L'instruction `GROUP BY` crée une nouvelle relation composée de couples dont les champs sont nommés :

- `group` : c'est le nom donné au champ qui a servi à construire les groupements, on ne peut pas changer le nom
- *relation* : le nom de la relation groupée est donnée au *bag*. Il contient tous les n-uplets dont le champ `BY` a la même valeur.

Il aurait été souhaitable que `GROUP achats BY idclient` produise des couples (idclient, achats). Mais non, ils sont nommés (group, achats). Donc si on fait

```
achatsparclient = GROUP achats BY idclient;
```

on devra mentionner `achatsparclient.group` et `achatsparclient.achats`.

7.3.12. Instruction GROUP ALL

L'instruction `GROUP relation ALL` regroupe tous les n-uplets de la relation dans un seul n-uplet composé d'un champ appelé `group` et valant `all`, et d'un second champ appelé comme la relation à grouper.

```
montants = FOREACH achats GENERATE montant;
montants_tous = GROUP montants ALL;
```

Crée ce seul n-uplet : `(all, {(12.50),(21.75),(56.25),(34.00),(3.30)})`

7.3.13. Utilisation de GROUP BY et FOREACH

On utilise en général FOREACH pour traiter le résultat d'un GROUP. Ce sont des couples (rappel) :

- Le premier champ venant du GROUP BY s'appelle group,
- Le second champ est un sac (*bag*) contenant tous les n-uplets liés à group. Ce sac porte le nom de la relation qui a servi à le créer.

On utilise FOREACH pour agréger le sac en une seule valeur, par exemple la somme de l'un des champs.

```
achats = LOAD 'achats.csv' AS (idachat, idclient, montant);
achatsparclient = GROUP achats BY idclient;
totalachatsparclient = FOREACH achatsparclient
    GENERATE group AS idclient, SUM(achats.montant) AS total;
```

7.3.14. Opérateurs

Pig propose plusieurs opérateurs permettant d'agréger les valeurs d'un sac. Le sac est construit par un GROUP BY ou GROUP ALL.

- SUM,AVG calcule la somme/moyenne des valeurs numériques.
- MAX,MIN retournent la plus grande/petite valeur
- COUNT calcule le nombre d'éléments du sac sans les null
- COUNT_STAR calcule le nombre d'éléments avec les null

Il y a d'autres opérateurs :

- CONCAT(v1, v2, ...) concatène les valeurs fournies.
- DIFF(sac1, sac2) compare les deux sacs et retourne un sac contenant les éléments qui ne sont pas en commun.
- SIZE retourne le nombre d'éléments du champ fourni.

7.3.15. Utilisation de GROUP et FOREACH

Dans certains cas, on souhaite aplatir le résultat d'un GROUP, c'est à dire au lieu d'avoir des sacs contenant tous les n-uplets regroupés, on les veut tous à part. Ça va donc créer autant de n-uplets séparés qu'il y avait d'éléments dans les sacs.

```
achatsparclient = GROUP achats BY idclient;
plusieurs = FILTER achatsparclient BY COUNT(achats)>1;
clientsmultiples = FOREACH plusieurs
    GENERATE group AS idclient, FLATTEN(achats.montant);
```

produit ceci, les achats des clients qui en ont plusieurs :

```
(1, 1, 34.0)
(4, 1, 12.5)
(3, 3, 56.25)
(5, 3, 3.30)
```

7.3.16. Instruction FOREACH GENERATE complexe

FOREACH relation { traitements... ; GENERATE ... } permet d'insérer des traitements avant le GENERATE.

- sousrelation = FILTER relation BY condition;

Exemple, on veut offrir un bon d'achat seulement aux clients qui ont fait de gros achats, le bon d'achat étant égal à 15% du montant total de ces gros achats :

```
achatsparclient = GROUP achats BY idclient;
grosbonsachat = FOREACH achatsparclient {
    grosachats = FILTER achats BY montant>=30.0;
    GENERATE group, SUM(grosachats.montant)*0.15 AS grosbon;
};
```

La relation `achat` du `FILTER` désigne le second champ du `GROUP achatsparclient` traité par le `FOREACH`.

7.3.17. Instruction `FOREACH GENERATE` complexe (suite)

Il est possible d'imbriquer d'autres instructions dans un `FOREACH` :

- `sousrelation = FOREACH relation GENERATE...;`
- `sousrelation = LIMIT relation N;`
- `sousrelation = DISTINCT relation;`

Exemple, on cherche les clients ayant acheté des produits différents :

```
achats = LOAD 'achats.csv'
    AS (idachat:int, idclient:int, idproduit:int, montant:float);
achatsparclient = GROUP achats BY idclient;
nombreparclient = FOREACH achatsparclient {
    produits = FOREACH achats GENERATE idproduit;
    differents = DISTINCT produits;
    GENERATE group AS idclient, COUNT(differents) AS nombre;
}
resultat = FILTER nombreparclient BY nombre>1;
```

7.3.18. `DISTINCT` sur certaines propriétés

On a vu que l'instruction `DISTINCT` filtre seulement les n-uplets exactement identiques. Voici comment supprimer les n-uplets en double sur certains champs seulement. Ca veut dire qu'on garde seulement l'un des n-uplets au hasard quand il y en a plusieurs qui ont les mêmes valeurs sur les champs considérés.

Soit une relation `A` de schéma (a1,a2,a3,a4,a5). On veut que les triplets (a1,a2,a3) soient uniques.

```
A = LOAD 'data.csv' AS (a1,a2,a3,a4,a5);
A_unique = FOREACH (GROUP A BY (a1,a2,a3)) {
    seul = LIMIT A 1;
    GENERATE group.a1,group.a2,group.a3,
        FLATTEN(seul.a4),FLATTEN(seul.a5);
}
```

7.3.19. Instruction `JOIN`

Les jointures permettent, comme en SQL de créer une troisième table à partir de plusieurs tables qui ont un champ en commun.

```
JOIN relation1 BY champ1a, relation2 BY champ2b, ...
```

Cela crée une relation ayant autant de champs que toutes les relations mentionnées. Les n-uplets qu'on y trouve sont ceux du produit cartésien entre toutes ces relations, pour lesquels le `champ1a` de la `relation1` est égal au `champ2b` de la `relation2`.

Dans la nouvelle relation, les champs sont nommés `relation1::champ1a`, `relation1::champ1b`, ...

7.3.20. Exemple de jointure

Soit une relation `clients` (idclient, nom) :

```
1 lucien
2 andré
3 marcel
```

Et une relation `achats` (idachat, idclient, montant) :

```
1, 1, 12.50
2, 2, 21.75
3, 3, 56.25
4, 1, 34.00
5, 3,  3.30
```

7.3.21. Exemple de jointure (suite)

L'instruction `JOIN clients BY idclient, achats BY idclient` crée (idclient, nom, idachat, idclient, montant) :

```
(1,lucien,4,1,34.25)
(1,lucien,1,1,12.5)
(2,andré,2,2,21.75)
(3,marcel,3,3,56.25)
(3,marcel,5,3,3.30)
```

Il y a d'autres types de jointure :

- `JOIN relationG BY champG LEFT, relationD BY champD` pour une jointure à gauche
- `JOIN relationG BY champG RIGHT, relationD BY champD` pour une jointure à droite
- `CROSS relation1, relation2, ...` produit cartésien

7.3.22. Instruction `UNION`

Cette instruction regroupe les n-uplets des relations indiquées.

```
UNION ONSCHEMA relation1, relation2, ...
```

Il est très préférable que les relations aient les mêmes schémas. Chaque champ crée sa propre colonne.

7.4. Conclusion

7.4.1. Comparaison entre SQL et Pig (le retour)

Revenons sur une comparaison entre SQL et Pig. Soit une petite base de données de clients et d'achats. La voici en SQL ; en Pig, ça sera deux fichiers CSV.

```
CREATE TABLE clients (
    idclient INTEGER PRIMARY KEY,
    nom VARCHAR(255));
CREATE TABLE achats (
    idachat INTEGER PRIMARY KEY,
    idclient INTEGER,
    FOREIGN KEY (idclient) REFERENCES clients(idclient),
    montant NUMERIC(7,2));
```

On veut calculer le montant total des achats par client.

7.4.2. Affichage nom et total des achats

Voici la requête SQL :

```
SELECT nom, SUM(montant) FROM clients JOIN achats
    ON clients.idclient = achats.idclient
    GROUP BY clients.idclient;
```

C'est très différent en Pig Latin, à méditer :

```
achats = LOAD 'achats.csv' AS (idachat, idclient, montant);
achatsparclients = GROUP achats BY idclient;
totaux = FOREACH achatsparclients
            GENERATE group, SUM(achats.montant) AS total;
clients = LOAD 'clients.csv' AS (idclient, nom);
jointure = JOIN clients BY idclient, totaux BY group;
resultat = FOREACH jointure GENERATE nom,total;
DUMP resultat;
```

Semaine 8

HBase et Hive

Le cours de cette semaine présente HBase et Hive.

HBase est un SGBD non relationnel, orienté colonne adapté au stockage et à l'accès rapide à des mégadonnées.

Hive est une surcouche de HBase afin d'offrir des fonctionnalités similaires à SQL.

8.1. Introduction

8.1.1. Présentation de HBase

HBase est un système de stockage efficace pour des données très volumineuses. Il permet d'accéder aux données très rapidement même quand elles sont gigantesques. Une variante de HBase est notamment utilisée par FaceBook pour stocker tous les messages SMS, email et chat, voir cette page.

HBase mémorise des n-uplets constitués de colonnes (champs). Les n-uplets sont identifiés par une **clé**. À l'affichage, les colonnes d'un même n-uplet sont affichées successivement :

```
Clés       Colonnes et Valeurs
isbn7615   colonne=auteur valeur="Jules Verne"
isbn7615   colonne=titre  valeur="De la Terre à la Lune"
isbn7892   colonne=auteur valeur="Jules Verne"
isbn7892   colonne=titre  valeur="Autour de la Lune"
```

8.1.2. Structure interne

Pour obtenir une grande efficacité, les données des tables HBase sont séparées en *régions*. Une région contient un certain nombre de n-uplets contigus (un intervalle de clés successives).

Une nouvelle table est mise initialement dans une seule région. Lorsqu'elle dépasse une certaine limite, elle se fait couper en deux régions au milieu de ses clés. Et ainsi de suite si les régions deviennent trop grosses.

Chaque région est gérée par un Serveur de Région (*Region Server*). Ces serveurs sont distribués sur le cluster, ex: un par machine. Un même serveur de région peut s'occuper de plusieurs régions de la même table.

Au final, les données sont stockées sur HDFS.

8.1.3. Tables et régions

Une table est découpée en régions faisant à peu près la même taille. Le découpage est basé sur les clés. Chaque région est gérée par un Serveur de région. Un même serveur peut gérer plusieurs régions.

Voir la figure 15, page 96.

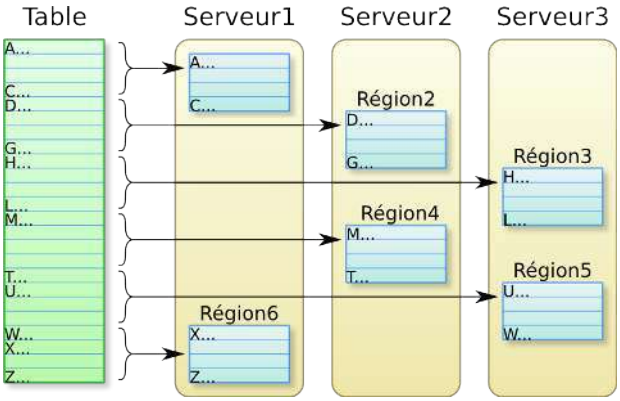

Figure 15: Régions HBase

8.1.4. Différences entre HBase et SQL

Voici quelques caractéristiques de HBase :

- Les n-uplets sont classés selon leur clé, dans l'ordre alphabétique. Cette particularité est extrêmement importante pour la recherche d'informations. On est amené à définir les clés de façon à rapprocher les données connexes.
- Les n-uplets de HBase peuvent être incomplets. Les colonnes ne sont pas forcément remplies pour chaque n-uplet, au point qu'on peut même avoir des colonnes différentes pour les n-uplets. Ce ne sont pas des valeurs `null`, mais des colonnes carrément absentes. On qualifie ça de « matrice creuse » (*sparse data*).
- Les colonnes appelées *qualifiers* sont groupées en *familles*.
- Les valeurs, appelées *cellules* sont enregistrées en un certain nombre de versions, avec une date appelée *timestamp*.

8.1.5. Structure des données

- Au plus haut niveau, une table HBase est un dictionnaire <clé, n-uplet> trié sur les **clés**,
- Chaque **n-uplet** est une liste de *familles*,
- Une **famille** est un dictionnaire <nomcolonne, cellule> trié sur les noms de colonnes (aussi appelées *qualifier*),
- Une **cellule** est une liste de (quelques) paires <valeur, date>. La date, un *timestamp* permet d'identifier la version de la valeur.

Donc finalement, pour obtenir une valeur isolée, il faut fournir un quadruplet :

```
(clé, nomfamille, nomcolonne, date)
```

Si la date est omise, HBase retourne la valeur la plus récente.

8.1.6. Exemple

On veut enregistrer les coordonnées et les achats de clients. On va construire une table contenant trois familles :

- La famille `personne` contiendra les informations de base :
  - colonnes `personne:nom` et `personne:prenom`
- La famille `coords` contiendra l'adresse :
  - colonnes `coords:rue`, `coords:ville`, `coords:cp`, `coords:pays`
- La famille `achats` contiendra les achats effectués :
  - colonnes `achats:date`, `achats:montant`, `achats:idfacture`

HBase autorise à dé-normaliser un schéma (redondance dans les informations) afin d'accéder aux données plus rapidement.

8.1.7. Nature des clés

Les familles et colonnes constituent un n-uplet. Chacun est identifié par une clé.

Les clés HBase sont constituées de n'importe quel tableau d'octets : chaîne, nombre... En fait, c'est un point assez gênant quand on programme en Java avec HBase, on doit tout transformer en tableaux d'octets : clés et valeurs. En Java, ça se fait par :

```
final byte[] octets = Bytes.toBytes(donnée);
```

Voir page 102 pour tous les détails.

Si on utilise le shell de HBase, alors la conversion des chaînes en octets et inversement est faite implicitement, il n'y a pas à s'en soucier.

8.1.8. Ordre des clés

Les n-uplets sont classés par ordre des clés et cet ordre est celui des octets. C'est donc l'ordre lexicographique pour des chaînes et l'ordre des octets internes pour les nombres. Ces derniers sont donc mal classés à cause de la représentation interne car le bit de poids fort vaut 1 pour les nombres négatifs ; -10 est rangé après 3.

Par exemple, si les clés sont composées de `"client"` concaténée à un numéro, le classement sera :

```
client1
client10
client11
client2
client3
```

Il faudrait écrire tous les numéros sur le même nombre de chiffres.

8.1.9. Choix des clés

Pour retrouver rapidement une valeur, il faut bien choisir les clés. Il est important que des données connexes aient une clé très similaire.

Par exemple, on veut stocker des pages web. Si on indexe sur leur domaine, les pages vont être rangées n'importe comment. La technique consiste à inverser le domaine, comme un package Java.

URL	URL inversé
info.iut-lannion.fr	com.alien.monster
monster.alien.com	com.alien.www
mp.iut-lannion.fr	fr.iut-lannion.info
www.alien.com	fr.iut-lannion.mp
www.iut-lannion.fr	fr.iut-lannion.www

Même chose pour les dates : AAAAMMJJ

8.1.10. Éviter le hotspotting

Il faut également concevoir les clés de manière à éviter l'accumulation de trafic sur une seule région. On appelle ça un point chaud (*hotspotting*). Ça arrive si les clients ont tendance à manipuler souvent les mêmes n-uplets.

Paradoxalement, ça peut être provoqué par le classement des clés pour l'efficacité comme dans le transparent précédent.

Ça vient aussi du fait qu'il n'y a qu'un seul serveur par région[2]. Il n'y a donc pas de parallélisme possible.

Pour résoudre ce problème, on peut disperser les clés en rajoutant du « sel », c'est à dire un bidule plus ou moins aléatoire placé au début de la clé, de manière à écarter celles qui sont trop fréquemment demandées : un timestamp, un hash du début de la clé...

8.2. Travail avec HBase

8.2.1. Shell de HBase

HBase offre plusieurs mécanismes pour travailler, dont :

- Un shell où on tape des commandes,
- Une API à utiliser dans des programmes Java, voir plus loin,
- Une API Python appelée HappyBase.

Il y a aussi une page Web dynamique qui affiche l'état du service et permet de voir les tables.

On va d'abord voir le shell HBase. On le lance en tapant :

```
hbase shell
```

Il faut savoir que c'est le langage Ruby qui sert de shell. Les commandes sont écrites dans la syntaxe de ce langage.

8.2.2. Commandes HBase de base

Voici les premières commandes :

- `status` affiche l'état du service HBase
- `version` affiche la version de HBase
- `list` affiche les noms des tables existantes
- `describe 'table'` décrit la table dont on donne le nom.

NB: ne pas mettre de `;` à la fin des commandes.

Attention à bien mettre les noms de tables, de familles et de colonnes entre `'...'`

Les commandes suivantes sont les opérations CRUD : créer, lire, modifier, supprimer.

8.2.3. Création d'une table

Il y a deux syntaxes :

- `create 'NOMTABLE', 'FAMILLE1', 'FAMILLE2'...`
- `create 'NOMTABLE', {NAME=>'FAMILLE1'}, {NAME=>'FAMILLE2'}...`

La seconde syntaxe est celle de Ruby. On spécifie les familles par un dictionnaire {propriété=>valeur}. D'autres propriétés sont possibles, par exemple `VERSIONS` pour indiquer le nombre de versions à garder.

Remarques :

- Les familles doivent être définies lors de la création. C'est coûteux de créer une famille ultérieurement.
- On ne définit que les noms des familles, pas les colonnes. Les colonnes sont créées dynamiquement.

8.2.4. Destruction d'une table

C'est en deux temps, il faut d'abord désactiver la table, puis la supprimer :

1. `disable 'NOMTABLE'`
2. `drop 'NOMTABLE'`

Désactiver la table permet de bloquer toutes les requêtes.

[2]Une évolution de HBase est demandée pour permettre plusieurs serveurs sur une même région.

8.2.5. Ajout et suppression de n-uplets

- **Ajout de cellules**

Un n-uplet est composé de plusieurs colonnes. L'insertion d'un n-uplet se fait colonne par colonne. On indique la famille de la colonne. Les colonnes peuvent être créées à volonté.

```
put 'NOMTABLE', 'CLE', 'FAM:COLONNE', 'VALEUR'
```

- **Suppression de cellules**

Il y a plusieurs variantes selon ce qu'on veut supprimer, seulement une valeur, une cellule, ou tout un n-uplet :

```
deleteall 'NOMTABLE', 'CLE', 'FAM:COLONNE', TIMESTAMP
deleteall 'NOMTABLE', 'CLE', 'FAM:COLONNE'
deleteall 'NOMTABLE', 'CLE'
```

8.2.6. Affichage de n-uplets

La commande `get` affiche les valeurs désignées par une seule clé. On peut spécifier le nom de la colonne avec sa famille et éventuellement le timestamp.

```
get 'NOMTABLE', 'CLE'
get 'NOMTABLE', 'CLE', 'FAM:COLONNE'
get 'NOMTABLE', 'CLE', 'FAM:COLONNE', TIMESTAMP
```

La première variante affiche toutes les colonnes ayant cette clé. La deuxième affiche toutes les valeurs avec leur timestamp.

8.2.7. Recherche de n-uplets

La commande `scan` affiche les n-uplets sélectionnés par les conditions. La difficulté, c'est d'écrire les conditions en Ruby.

```
scan 'NOMTABLE', {CONDITIONS}
```

Parmi les conditions possibles :

- `COLUMNS=>['FAM:COLONNE',...]` pour sélectionner certaines colonnes.
- `STARTROW=>'CLE1', STOPROW=>'CLE2'` pour sélectionner les n-uplets de [CLE1, CLE2[.

Ou alors (exclusif), une condition basée sur un filtre :

- `FILTER=>"PrefixFilter('binary:client')"`

Il existe de nombreux filtres, voir la doc.

8.2.8. Filtres

L'ensemble des filtres d'un scan doit être placé entre `"..."`.

Plusieurs filtres peuvent être combinés avec `AND`, `OR` et les parenthèses.

Exemple :

```
{ FILTER =>
    "PrefixFilter('client') AND ColumnPrefixFilter('achat')" }
```

- `PrefixFilter('chaîne')` : accepte les valeurs dont la clé commence par la chaîne
- `ColumnPrefixFilter('chaîne')` : accepte les valeurs dont la colonne commence par la chaîne.

8.2.9. Filtres, suite

Ensuite, on a plusieurs filtres qui comparent quelque chose à une valeur constante. La syntaxe générale est du type :

`MachinFiter(OPCMP, VAL)`

... avec *OPCMP VAL* définies ainsi :

- `OPCMP` doit être l'un des opérateurs `<`, `<=`, `=`, `!=`, `>` ou `>=` (sans mettre de quotes autour)
- `VAL` est une constante qui doit valoir :
  - `'binary:chaine'` pour une chaîne telle quelle
  - `'substring:chaine'` pour une sous-chaîne
  - `'regexstring:motif'` pour un motif egrep, voir la doc.

Exemple :

```
{ FILTER => "ValueFilter(=,'substring:iut-lannion')"}
```

8.2.10. Filtres, suite

Plusieurs filtres questionnent la clé, famille ou colonne d'une valeur :

- `RowFilter(OPCMP, VAL)`
- `FamilyFilter(OPCMP, VAL)`
- `QualifierFilter(OPCMP, VAL)`
  accepte les n-uplet dont la clé, famille, colonne correspond à la constante
- `SingleColumnValueFilter('fam','col',OPCMP,VAL)`
  garde les n-uplets dont la colonne `'fam:col'` correspond à la constante. Ce filtre est utile pour garder des n-uplets dont l'un des champs possède une valeur qui nous intéresse.
- `ValueFilter(OPCMP, VAL)`
  accepte les valeurs qui correspondent à la constante

8.2.11. Comptage de n-uplets

Voici comment compter les n-uplets d'une table, en configurant le cache pour en prendre 1000 à la fois

`count 'NOMTABLE', CACHE => 1000`

C'est tout ?

Oui, HBase n'est qu'un stockage de mégadonnées. Il n'a pas de dispositif d'interrogations sophistiqué (pas de requêtes imbriquées, d'agrégation, etc.)

Pour des requêtes SQL sophistiquées, il faut faire appel à Hive. Hive est un SGBD qui offre un langage ressemblant à SQL et qui s'appuie sur HBase.

8.3. API Java de HBASE

8.3.1. Introduction

On peut écrire des programmes Java qui accèdent aux tables HBase. Il y a un petit nombre de classes et de méthodes à connaître pour démarrer.

Nous allons voir comment :

- créer une table
- ajouter des cellules
- récupérer une cellule
- parcourir les cellules

Noter qu'on se bat contre le temps, HBase évolue très vite et de nombreux aspects deviennent rapidement obsolètes (*deprecated*). L'API était en 0.98.12 cette année (hadoop 2.7.1), maintenant c'est la 2.0 et la 3.0 est déjà annoncée.

8.3.2. Imports communs

Pour commencer, les classes à importer :

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.*;
```

Ensuite, chaque programme contient ces lignes (API 0.98) qui établissent une connexion avec le serveur HBase :

```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(config);
try {
    ...
} finally {
    admin.close();
}
```

8.3.3. Création d'une table

Voici une fonction qui crée une table :

```
static void CreerTable(HBaseAdmin admin,
            String nomtable, String... familles)
            throws IOException
{
    TableName tn = TableName.valueOf(nomtable);
    HTableDescriptor htd = new HTableDescriptor(tn);
    for (String famille: familles) {
        htd.addFamily(new HColumnDescriptor(famille));
    }
    admin.createTable(htd);
}
```

Notez que j'ai fourni le `HBaseAdmin` en paramètre et que les familles sont sous forme de *varargs*.

8.3.4. Suppression d'une table

Voici comment on supprime une table, en vérifiant au préalable si elle existe :

```
static void SupprimerTable(HBaseAdmin admin, String nomtable)
{
    TableName tn = TableName.valueOf(nomtable);
    if (admin.tableExists(tn)) {
        admin.disableTable(tn);
        admin.deleteTable(tn);
    }
}
```

8.3.5. Manipulation d'une table

Dans les fonctions suivantes, on va modifier les données d'une table existante. Pour cela, il faut récupérer un objet `HTable` représentant la table. Il est important de libérer cet objet dès qu'il ne sert plus. Voici comment faire en API 0.98 :

```
static void OperationSurTable(Configuration config,
                String nomtable, ...)
{
    HTable table = new HTable(config, nomtable);
    try {
        ... opérations sur le contenu de la table ...
    } finally {
        table.close();
    }
}
```

Prévoir aussi l'arrivée de `IOException` à tout moment.

8.3.6. Insertion d'une valeur

L'insertion d'une valeur consiste à créer une instance de la classe `Put`. Cet objet spécifie la valeur à insérer :

- identifiant du n-uplet auquel elle appartient
- nom de la famille
- nom de la colonne
- valeur
- en option, le timestamp à lui affecter.

Toutes les données concernées doivent être converties en tableaux d'octets.

8.3.7. Transformation en tableaux d'octets

HBase stocke des données binaires quelconques : chaînes, nombres, images jpg, etc. Il faut seulement les convertir en `byte[]`.

Convertir une donnée en octets se fait quelque soit son type par :

```
final byte[] octets = Bytes.toBytes(donnée);
```

Dans le cas de clés de n-uplets de type nombre (int, long, float et double), le classement des clés sera fantaisiste à cause de la représentation interne, voir cette page. Du fait que le signe du nombre soit en tête et vaut 1 pour les nombres négatifs, 0 pour les nombres positifs, un nombre négatif sera considéré comme plus grand qu'un positif.

Il est possible d'y remédier en trafiquant le tableau d'octets afin d'inverser le signe mais c'est hors sujet.

8.3.8. Transformation inverse

La récupération des données à partir des octets n'est pas uniforme. Il faut impérativement connaître le type de la donnée pour la récupérer correctement. Il existe plusieurs fonctions, voir la doc :

```
String chaine = Bytes.toString(octets);
Double nombre = Bytes.toDouble(octets);
Long entier = Bytes.toLong(octets);
```

Dans certains cas, HBase nous retourne un grand tableau d'octets dans lequel nous devons piocher ceux qui nous intéressent. Nous avons donc trois informations : le tableau, l'offset du premier octet utile et le nombre d'octets. Il faut alors faire ainsi :

```
Double nombre = Bytes.toDouble(octets, debut, taille);
```

8.3.9. Insertion d'une valeur, fonction

```
static void AjouterValeur(Configuration config,
        String nomtable, String id,
        String fam, String col, String val)
{
    HTable table = new HTable(config, nomtable);
    try {
        // construire un Put
        final byte[] rawid = Bytes.toBytes(id);
        Put action = new Put(rawid);
        final byte[] rawfam = Bytes.toBytes(fam);
        final byte[] rawcol = Bytes.toBytes(col);
        final byte[] rawval = Bytes.toBytes(val);
        action.add(rawfam, rawcol, rawval);
        // effectuer l'ajout dans la table
        table.put(action);
    } finally { table.close(); }
}
```

8.3.10. Insertion d'une valeur, critique

- Le problème de la fonction précédente, c'est qu'on a ajouté une valeur de type chaîne. Il faut écrire une autre fonction pour ajouter un entier, un réel, etc.
  Il faudrait réfléchir à une fonction un peu plus générale, à laquelle on peut fournir une donnée quelconque et qui l'ajoute correctement en binaire. C'est pas tout à fait trivial, car la méthode `Bytes.toBytes` n'accepte pas le type `Object` en paramètre.
- Il ne faut pas insérer de nombreuses valeurs une par une avec la méthode `table.put`. Utiliser la surcharge `table.put(ArrayList<Put> liste)`.

8.3.11. Extraire une valeur

La récupération d'une cellule fait appel à un `Get`. Il se construit avec l'identifiant du n-uplet voulu. Ensuite, on applique ce `Get` à la table. Elle retourne un `Result` contenant les cellules du n-uplet.

```
static void AfficherNuplet(Configuration config,
            String nomtable, String id)
{
    final byte[] rawid = Bytes.toBytes(id);
    Get action = new Get(rawid);
    // appliquer le get à la table
    HTable table = new HTable(config, nomtable);
    try {
        Result result = table.get(action);
        AfficherResult(result);
    } finally { table.close(); }
}
```

8.3.12. Résultat d'un `Get`

Un `Result` est une sorte de dictionnaire (famille,colonne)→valeur

- Sa méthode `getValue(famille, colonne)` retourne les octets de la valeur désignée, s'il y en a une :

```
byte[] octets = result.getValue(rawfam, rawcol);
```

- On peut parcourir toutes les cellules par une boucle :

```
void AfficherResult(Result result)
{
    for (Cell cell: result.listCells()) {
        AfficherCell(cell);
    }
}
```

8.3.13. Affichage d'une cellule

C'est un peu lourdingue car il faut extraire les données de tableaux d'octets avec offset et taille, et attention si le type n'est pas une chaîne.

```
static void AfficherCell(Cell cell)
{
    // extraire la famille
    String fam = Bytes.toString(cell.getFamilyArray(),
            cell.getFamilyOffset(), cell.getFamilyLength());
    // extraire la colonne (qualifier)
    String col = Bytes.toString(cell.getQualifierArray(),
            cell.getQualifierOffset(), cell.getQualifierLength());
    // extraire la valeur (PB si c'est pas un String)
    String val = Bytes.toString(cell.getValueArray(),
            cell.getValueOffset(), cell.getValueLength());
    System.out.println(fam+":"+col+" = "+val);
}
```

8.3.14. Parcours des n-uplets d'une table

Réaliser un `Scan` par programme n'est pas très compliqué. Il faut fournir la clé de départ et celle d'arrêt (ou alors le scan se fait sur toute la table). On reçoit une énumération de `Result`.

```
static void ParcourirTable(Configuration config,
            String nomtable, String start, String stop)
{
    final byte[] rawstart = Bytes.toBytes(start);
    final byte[] rawstop = Bytes.toBytes(stop);
    Scan action = new Scan(rawstart, rawstop);

    HTable table = new HTable(config, nomtable);
    ResultScanner results = table.getScanner(action);
    for (Result result: results) {
        AfficherResult(result);
    }
}
```

8.3.15. Paramétrage d'un Scan

Il est possible de filtrer le scan pour se limiter à :

- certaines familles et colonnes (on peut en demander plusieurs à la fois), sinon par défaut, c'est toutes les colonnes.

```
action.add(rawfam, rawcol);
```

- rajouter des filtres sur les valeurs, par exemple ci-dessous, on cherche les colonnes supérieures ou égales à une limite.
  NB: Il faut utiliser la classe `CompareOp` et `BinaryComparator`

```
BinaryComparator binlimite = new BinaryComparator(rawlimite);
SingleColumnValueFilter filtre = new SingleColumnValueFilter(
        rawfam, rawcol, CompareOp.GREATER_OR_EQUAL, binlimite);
action.setFilter(filtre);
```

8.3.16. Filtrage d'un Scan

Pour définir une condition complexe, il faut construire un `FilterList` qu'on attribue au `Scan`. Un `FilterList` représente soit un « et » (`MUST_PASS_ALL`), soit un « ou » (`MUST_PASS_ONE`). Les `FilterList` peuvent être imbriqués pour faire des combinaisons complexes.

```
SingleColumnValueFilter filtre1 = ...
QualifierFilter filtre2 = ...

FilterList conjonction = new FilterList(
    FilterList.Operator.MUST_PASS_ALL, filtre1, filtre2);
action.setFilter(conjonction);
```

Attention aux comparaisons de nombres. Elles sont basées sur la comparaison des octets internes, or généralement, les nombres négatifs ont un poids fort plus grand que celui des nombres positifs.

8.4. Hive

8.4.1. Présentation rapide

Hive simplifie le travail avec une base de données comme HBase ou des fichiers CSV. Hive permet d'écrire des requêtes dans un langage inspiré de SQL et appelé HiveQL. Ces requêtes sont transformées en jobs MapReduce.

Pour travailler, il suffit définir un schéma qui est associé aux données. Ce schéma donne les noms et types des colonnes, et structure les informations en tables exploitables par HiveQL.

8.4.2. Définition d'un schéma

Le schéma d'une table est également appelé méta-données (c'est à dire informations sur les données). Les métadonnées sont stockées dans une base de données MySQL, appelée *metastore*.

Voici la définition d'une table avec son schéma :

```
CREATE TABLE releves (
        idreleve STRING,
        annee INT, ...
        temperature FLOAT, quality BYTE,
        ...)
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

Le début est classique, sauf les contraintes d'intégrité : il n'y en a pas. La fin de la requête indique que les données sont dans un fichier CSV. Voyons d'abord les types des colonnes.

8.4.3. Types HiveQL

Hive définit les types suivants :

- `BIGINT` (8 octets), `INT` (4), `SMALLINT` (2), `BYTE` (1 octet)
- `FLOAT` et `DOUBLE`
- `BOOLEAN` valant `TRUE` ou `FALSE`
- `STRING`, on peut spécifier le codage (UTF8 ou autre)
- `TIMESTAMP` exprimé en nombre de secondes.nanosecondes depuis le 01/01/1970 UTC
- données structurées comme avec Pig :
  - `ARRAY<type>` indique qu'il y a une liste de *type*
  - `STRUCT<nom1:type1, nom2:type2...>` pour une structure regroupant plusieurs valeurs
  - `MAP<typecle, typeval>` pour une suite de paires clé,valeur

8.4.4. Séparations des champs pour la lecture

La création d'une table se fait ainsi :

```
CREATE TABLE nom (schéma) ROW FORMAT DELIMITED descr du format
```

Les directives situées après le schéma indiquent la manière dont les données sont stockées dans le fichier CSV. Ce sont :

- `FIELDS TERMINATED BY ';'` : il y a un `;` pour séparer les champs
- `COLLECTION ITEMS TERMINATED BY ','` : il y a un `,` entre les éléments d'un ARRAY
- `MAP KEYS TERMINATED BY ':'` : il y a un `:` entre les clés et les valeurs d'un MAP
- `LINES TERMINATED BY '\n'` : il y a un `\n` en fin de ligne
- `STORED AS TEXTFILE` : c'est un CSV.

8.4.5. Chargement des données

Voici comment charger un fichier CSV qui se trouve sur HDFS, dans la table :

```
LOAD DATA INPATH '/share/noaa/data/186293'
    OVERWRITE INTO TABLE releves;
```

NB: le problème est que Hive **déplace** le fichier CSV dans ses propres dossiers, afin de ne pas dupliquer les données. Sinon, on peut écrire `CREATE EXTERNAL TABLE ...` pour empêcher Hive de capturer le fichier.

On peut aussi charger un fichier local (pas HDFS) :

```
LOAD DATA LOCAL INPATH 'stations.csv'
    OVERWRITE INTO TABLE stations;
```

Le fichier est alors copié sur HDFS dans les dossiers de Hive.

8.4.6. Liens entre HBase et Hive

Il est également possible d'employer une table HBase.

Cela fait appel à la notion de gestionnaire de stockage (*Storage Handler*). C'est simplement une classe générale qui gère la lecture des données. Elle a des sous-classes pour différents systèmes, dont `HBaseStorageHandler` pour HBase.

```
CREATE TABLE stations(idst INT, name STRING, ..., lat FLOAT,...)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH
SERDEPROPERTIES("hbase.columns.mapping" = ":idst,data:name,...")
TBLPROPERTIES("hbase.table.name" = "stations");
```

La clause SERDEPROPERTIES (serialisation/désérialisation) associe les noms des colonnes HBase à ceux de la table Hive.

8.4.7. Requêtes HiveQL

Comme avec les SGBD conventionnels, il y a un shell lancé par la commande `hive`. C'est là qu'on tape les requêtes SQL. Ce sont principalement des `SELECT`. Toutes les clauses que vous connaissez sont disponibles : `FROM`, `JOIN`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`.

Il y en a d'autres pour optimiser le travail MapReduce sous-jacent, par exemple quand vous voulez classer sur une colonne, il faut écrire :

```
SELECT... DISTRIBUTE BY colonne SORT BY colonne;
```

La directive envoie les n-uplets concernés sur une seule machine afin de les comparer plus rapidement pour établir le classement.

8.4.8. Autres directives

Il est également possible d'exporter des résultats dans un dossier :

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/meteo/chaud'
SELECT annee,mois,jour,temperature
FROM releves
WHERE temperature > 40.0;
```

Parmi les quelques autres commandes, il y a :

- `SHOW TABLES;` pour afficher la liste des tables (elles sont dans le metastore).
- `DESCRIBE EXTENDED table;` affiche le schéma de la table