# TD2 - Fouille de textes
# MapReduce avec Python

## Introduction

MapReduce est un modèle de programmation parallèle défini par Google et inspiré de concepts issus de la programmation fonctionnelle. Il est conçu pour manipuler de grandes quantités de données via une architecture distribuée. L'implémentation la plus connue de ce modèle est probablement celle d'Apache, appelée Hadoop.

Le but de ce TD est de comprendre le principe de parallélisation effectué par MapReduce, et de l'appliquer sur des exemples simples en utilisant une implémentation minimaliste en Python.

*Remarque* : une grande partie de ce TD a été réalisée par Vincent Labatut au sein d'Avignon Université pour l'UE Application Business Intelligence, et dispensé par R. Dufour et V. Labatut.

## Principe de MapReduce

Le principe est le suivant : on découpe d'abord les données en plusieurs parties traitables indépendamment par des processeurs distincts, puis on combine les résultats obtenus pour chaque partie de manière à construire le résultat global.

*Décomposition du traitement*

Le traitement est distribué sur un ensemble de nœuds de calcul. Il se compose des étapes suivantes, illustrées par la Figure 1 :

1. **Découpage des données** : les données d'entrée sont lues par le nœud principal, et découpées en n parties. Chaque partie Pi ($1 \leq i \leq n$) est associée à une clé unique Ki permettant de l'identifier.
2. **Fonction Map** : une fonction Map est invoquée séparément pour chaque partie, généralement en parallèle sur plusieurs nœuds. Elle est donc invoquée n fois au total. Cette fonction prend en entrée une paire (Ki,Pi) issue du découpage des données, et renvoie une liste de paires $Li = \langle(Kj', Vij), ...\rangle$. Chaque paire contient une clé $Kj'$ et une valeur Vij ($1 \leq j \leq m$) résultant du traitement réalisé par la fonction. Une même clé peut apparaître plusieurs fois dans la liste. L'ensemble des m clés possibles est le même pour toutes les exécutions de Map.
3. **Répartition des paires** : chaque clé $Kj'$ est associée à un nœud de traitement (plusieurs clés peuvent être associées à un même nœud). Ce nœud reçoit toutes les paires ($Kj'$,Vij) produites par l'ensemble des nœuds ayant exécuté la fonction Map. Une paire ($Kj'$,$L'j$) est construite, dont le deuxième terme est une liste contenant toutes les valeurs associées à la clé considérée : $L'j = \langle Vij, \ldots\rangle$.
4. **Fonction Reduce** : chaque nœud sélectionné lors de la répartition exécute une fonction Reduce pour chacune des clés $Kj'$ qui lui sont associées. Autrement dit, la fonction est exécutée un total de m fois (en considérant tous les nœuds). La fonction prend en entrée la paire ($Kj'$,$L'j$) issue de la répartition. À l'issue de son traitement, elle renvoie une valeur $Vj'$ (ou parfois plusieurs).
5. **Résultat final** : le résultat final est une liste de paires $L'' = \langle(Kj', Vj'), ..., (Kj', Vj')\rangle$.

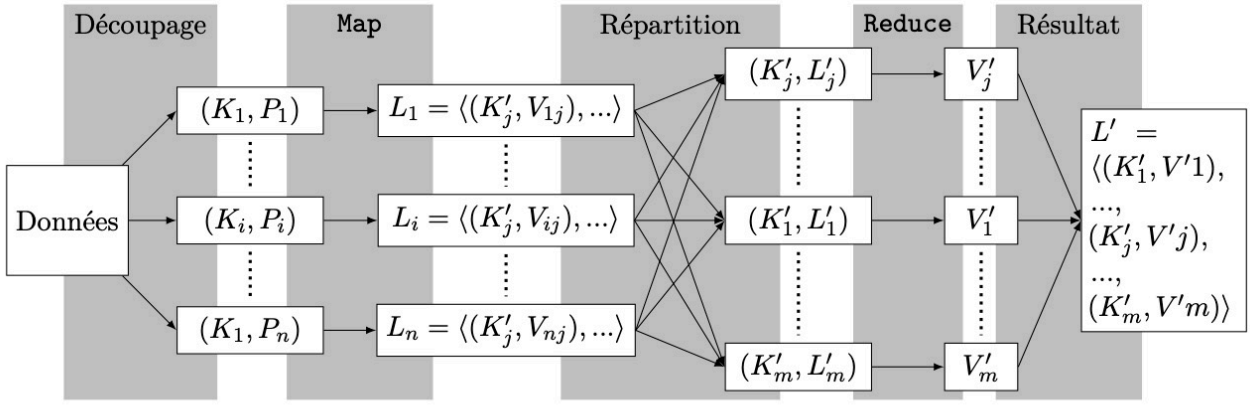


**Figure 1.** Décomposition d'un traitement de type *MapReduce*.

*Exemple*

L'exemple typique utilisé pour illustrer simplement cette approche est celle du décompte de mots dans un texte. Supposons qu'on a le texte suivant :

```
Voici la première ligne,
et voici une autre ligne : la seconde ligne.
La troisième ligne.
Et enfin la dernière ligne.
```

**Découpage** : Dans cet exemple, on pourrait envisager que le découpage des données se fasse par phrases. À l'issue de cette étape, on aurait donc les paires (l1,"Voici la première ligne,"), (l2,"et voici une autre ligne : la seconde ligne."), (l3,"La troisième ligne."), et (l4,"Et enfin la dernière ligne.") (où li est la ligne numéro i, avec $1 \leq i \leq 4$).

**Map** : Supposons qu'on ait 4 nœuds de calcul disponibles, exécutant chacun la fonction Map pour une clé différente (donc pour une ligne). Notez qu'il peut y avoir moins de 4 nœuds, auquel cas certains d'entre-eux devront traiter plusieurs clés (indépendamment). Dans cet exemple, le rôle de Map est de décomposer la phrase reçue en entrée, et de renvoyer les mots obtenus, en leur associant la valeur 1 (chaque mot apparaissant une fois dans la phrase). Ainsi, pour la ligne l2, on a : L2 = ⟨("et", 1), ("voici", 1), ("une", 1), ("autre", 1), ("ligne", 1), ("la", 1), ("seconde", 1), ("ligne", 1)⟩.
Les clés Kj′ obtenues à l'issue de cette étape correspondent aux mots uniques contenus dans le texte traité, à savoir : "autre", "dernière", "enfin", "et", "la", "ligne", "première", "seconde", "troisième", "une", "voici". On a donc m = 11. De plus, dans cet exemple, toutes les valeurs Vij sont égales à 1.

**Répartition** : Pour chaque clé Kj′ détectée lors du Map, on construit la paire (Kj′ , L′j ) associée. Par exemple, pour la clé K6′ ="ligne", on a L′6 = ⟨1, 1, 1, 1, 1⟩ (i.e. un 1 pour chaque occurrence du mot).

**Reduce** : Supposons qu'au moment de la répartition, on ait seulement 3 noeuds disponibles : chacun va traiter indépendamment 3 ou 4 des 11 clés Kj′. Dans cet exemple, Reduce consiste simplement à additionner les valeurs trouvées dans la paire de l'étape précédente. Ainsi, pour K5′ ="la", la fonction renverra V5′ = 4, alors que pour K6′ ="ligne" on aura V6′ = 5.

**Résultat** : Le résultat final associe à chaque clé Kj′ le nombre d'occurrence du mot correspondant. On peut ici le représenter sous la forme d'une liste de paires : L′′ = ⟨ ("autre", 1), ("dernière", 1), ("enfin", 1), ("et", 2), ("la", 4), ("ligne", 5), ("première", 1), ("seconde", 1), ("troisième", 1), ("une", 1), ("voici", 2) ⟩.

## Utilisation depuis Python

Pour des raisons pratiques, nous allons utiliser la bibliothèque Mrs-MapReduce, une version très légère de MapReduce implémentée en Python. Dans le cadre d'une exploitation plus professionnelle, nous nous tournerions plutôt vers la plate-forme Apache Hadoop, qui possède de bien meilleures performances (mais dont le processus d'installation et de configuration est bien plus ardu). Notez que bien qu'Hadoop soit implémentée en Java, plusieurs bibliothèques ont été publiées pour permettre de l'invoquer depuis Python (ou d'autres langages répandus).

*Installation*

$ pip install mrs-mapreduce —user

Le module doit être installé sur chaque machine destinée à faire partie du cluster.

*Programmation*

À titre d'exemple, nous allons reprendre le problème de décompte de mots décrit précédemment.
Le module nécessite de définir trois opérations :
1. Chargement des données (une partie de l'étape 1 dans la Section *Décomposition du traitement*) ;
2. Fonction Map (étape 2) ;
3. Fonction Reduce (une partie de l'étape 4).

Le reste du traitement est réalisé automatiquement par le module. Notez qu'il est possible de surcharger ce comportement par défaut pour l'adapter à des problèmes spécifiques.
Le plus simple pour définir les trois opérations mentionnées ci-dessus est de créer une classe héritant de mrs.MapReduce, et de surcharger les méthodes suivantes, chacune correspondant à l'une des opérations :
1. input_data(self, job);
2. map(self, key, value);
3. reduce(self, key, values).

*Préparation des données*

**Exercice 1**

Décompressez-la de manière à obtenir trois dossiers wp_20, wp_200 et wp_2000 situés à la racine de votre projet.

Le corpus complet (wp_2000) contient 2 871 textes tirés de la version française de Wikipedia, obtenus en crawlant automatiquement le site à partir de la page Recherche d'information (avec une limite de 2 hops et en restant dans Wikipedia). Les articles sont nettoyés pour ne contenir que lettres, chiffres, et ponctuation. Les dossiers wp_20, wp_200 sont des versions réduites du corpus complet, contenant respectivement 20 et 200 articles. On considère chaque article comme une partie des données (notation Pi dans la Section 1.1), identifiée de façon unique par le nom du fichier qui le contient (Ki).

**Exercice 2**

Créez une classe WordCount héritant de mrs.MapReduce (n'oubliez pas d'importer le module mrs). Dans cette classe, écrivez une fonction input_data(self, job) qui initialise l'objet job. Celui-ci est une instance d'une classe interne au module, chargée de représenter les données à traiter.
Vous devez d'abord accéder au dossier contenant les données, afin de lister les fichiers texte qu'il contient. On suppose que ce dossier est passé en paramètre de la ligne de commande ayant servi à lancer le programme depuis un terminal, donc il est accessible via self.args[0]. Pour lister les fichiers qu'il contient, le plus simple est d'utiliser le module glob.
Il faut ensuite placer la liste de fichiers ainsi obtenue dans le paramètre job, en utilisant sa méthode file_data(lst) (où lst est la liste des fichiers à traiter), et renvoyer le tout pour terminer la méthode.

**Exercice 3**

À la fin de votre classe, ajoutez les lignes suivantes :

```
if __name__ == '__main__':
    mrs.main(WordCount)
```

Elles permettent de passer le programme à traiter au module.

Remarque : il est obligatoire de surcharger map et reduce pour pouvoir exécuter votre classe. Lorsque vous testez vos méthodes, il est donc nécessaire de définir des versions minimales de map et de reduce, quitte à renvoyer des données bidons, afin d'éviter de provoquer des exceptions. Notez que les données bidons renvoyées doivent toutefois être compatibles avec les structures de données attendues de la part de map et reduce.

## Fonctions Map et Reduce

**Exercice 4**

Toujours dans la même classe, surchargez la méthode map(self, key, value), qui reçoit en paramètres une clé key (correspondant à Ki) et une valeur value (correspondant à Pi, ici : du texte).
Cette méthode doit décomposer value en mots, afin de construire la liste Li de paires (Kj′ , Vij ) décrite en Section 1.1. Comme expliqué en Section 1.2, pour l'application considérée ici : les Kj′ sont les mots, et les Vij sont tous égaux à 1. On considèrera que tout ce qui n'est pas une lettre est un séparateur de mots (y compris les chiffres). De plus, on ne veut pas tenir compte de la casse.
Le module *re* vous permet de traiter les expressions régulières en Python. Ici, en particulier, sa fonction split vous sera utile. Notez aussi que l'expression régulière \W représente n'importe quel

caractère utilisé pour séparer des mots (en tenant compte des signes diacritiques définis pour le locale courant). Attention à bien ignorer les chaînes vides ou ne contenant que des espaces.
La méthode map est une fonction génératrice, ce qui signifie qu'en réalité, elle ne renvoie pas une liste complète. Au lieu de cela, elle permet de parcourir la liste itérativement, en renvoyant à chaque appel successif l'élément suivant dans cette liste. Exprimé différemment : chaque élément est produit à la demande. Pour obtenir ce comportement, vous devez utilisez le mot-clé *yield*.

**Exercice 5**

Surchargez la méthode reduce(self, key, values), qui reçoit en paramètres une clé key (correspondant à Kj′) et une liste values de valeurs associées à cette clé (correspondant à L′j).
Dans notre cas, cette clé est un mot, et les valeurs sont numériques.
Dans l'application considérée ici, cette méthode doit calculer la somme des valeurs associées à la clé reçue. Notez que faire cette somme revient à compter le nombre de valeurs présentes dans la liste, puisque celles-ci sont toutes égales à 1. La fonction doit simplement renvoyer Vj′, la valeur calculée pour la clé Kj′ (le module se charge de reconstituer automatiquement la paire (Kj′ , Vj′) décrite à l'étape 4 dans la Section 1.1).
Remarquez que dans le cas général (i.e. hors de l'application considérée ici), il est possible d'avoir à renvoyer plusieurs valeurs pour une clé donnée (et non pas une seule valeur comme ici). Pour cette raison, à l'instar de map, reduce doit elle aussi être une fonction génératrice. Il est donc là aussi nécessaire d'utiliser yield (même pour une seule valeur).

*Exécution*

Le module distingue deux types de machines : un maître et un ou plusieurs esclaves (une variante du paradigme client/serveur). Le maître se charge de l'initialisation, d'indiquer aux esclaves ce qu'ils ont à faire, de collecter les résultats qu'ils produisent, de finaliser le traitement. Les esclaves ne sont concernés que par les fonctions map et reduce.
Il est nécessaire que le module soit installé sur toutes les machines constituant le cluster (qu'elles soient le maître ou l'un des esclaves). De la même façon, votre propre code source doit lui aussi être dupliqué sur chacune de ces machines.

*Test*

Avant d'exécuter le programme en mode MapReduce, il est possible de le tester localement en utilisant la commande suivante :

**$ python WordCount.py input output --mrs-verbose**

Où input est le dossier contenant les textes, et output est celui destiné à recevoir les résultats du traitement.
Si le traitement se déroule correctement, un fichier texte devrait apparaître dans le dossier output : il contient le résultat du traitement, c'est-à-dire ici une liste de mots avec pour chacun sa fréquence dans le corpus traité.

*Lancement du maître*

La commande suivante permet de lancer le maître, en supposant que votre classe se nomme WordCount.py :

**$ python WordCount.py -I Master -P yyyyy --mrs-verbose input output**

La valeur yyyyy correspond au port que vous désirez utiliser, tandis qu'input et output ont la même signification que précédemment.
Une fois que le maître est initialisé, il se met en attente des esclaves :

**2018-10-01 11:57:45,148: INFO: Listening on port yyyyy.**
**Map: 0.0% complete. Reduce: 0.0% complete.**

Remarque. il est possible d'utiliser un script pour lancer le maître et tous ses esclaves en une seule fois. Cependant, cela nécessite l'utilisation de librairies additionnelles, et une configuration spécifique. Aussi, par souci de simplicité, nous resterons en mode manuel pour ce TP.

*Lancement d'un esclave*

La commande suivante est à exécuter sur chaque esclave, en indiquant à la place de x.x.x.x l'adresse IP du maître :

**$ python WordCount.py -I Slave -M x.x.x.x:yyyyy --mrs-verbose**

Notez qu'il est possible d'exécuter l'esclave sur la même machine, voire plusieurs esclaves sur la même machine, en désignant le maître par localhost.
L'esclave va se mettre aux ordres du maître :

**2018-10-01 12:00:00,431: INFO: Reported ready to master.**

Le maître devrait alors détecter l'esclave et lui assigner du travail :

**2018-10-01 12:00:00,429: INFO: New slave 0 on host z.z.z.z**
**2018-10-01 12:00:00,431: INFO: Starting work on dataset: map_2W9ni0fW**

Où z.z.z.z est l'adresse IP de l'esclave.
Pour vous aider à déboguer votre code, voici (une partie de) la sortie que vous devriez obtenir pour le plus gros fichier :

```
a 39276
aa 126
aaa 61
aaaa 15
aaaaaaaaa 1
aaaaaaaazzeeeeeer 1
aaai 1
aaas 1
aab 1
aabab 1
...
```

```
zx 15
zy 6
zygalski 1
zygmunt 1
zymotique 1
zynga 4
zyriab 1
zytglogge 1
zytkow 1
zyxel 1
zz 5
```

Notez que le fait de passer les mots en minuscules avec la fonction lower() supprime les accents, ce qui explique qu'ils soient absents de la liste précédente.

*Temps d'exécution*

Vous pouvez mesurer le temps d'exécution côté maître en utilisant la commande time d'Ubuntu :

**$ time python WordCount.py ...**

Cette commande affiche les valeurs suivantes :
• real : temps réel écoulé entre le début et la fin de la commande (qui peut comprendre du temps processeur utilisé par d'autres processus).
• user : temps processeur passé par la commande en mode utilisateur (i.e. hors noyau, attente, etc.)
• sys : temps processeur passé par la commande en mode noyau.

**Exercice 6**

Expérimentez de manière à étudier l'évolution du temps de calcul en fonction du nombre d'esclaves, de la taille des données, etc. Résumez vos résultats dans une table, et discutez-les.

## Autre application

Dans le cadre de l'informatique décisionnelle et de la fouille de données, MapReduce peut être utile pour effectuer du pré-traitement sur les données brutes, c'est-à-dire pour les préparer à être manipulées par des algorithmes tels que des classificateurs, par exemple. On peut en particulier filtrer ces données, c'est à dire écarter les instances qui ne respectent pas certains critères, ou bien transformer ces données (par exemple : normaliser un champ numérique).
Le dossier socio fourni avec le sujet contient 4 versions d'une table qui décrit une population d'individus grâce aux 5 champs suivants :

• **id** : identifiant unique (entier),
• **age** : âge en année (entier),
• **height** : taille en cm (entier),
• **weight** : masse en kg (entier),
• **sex** : sexe (catégorie : female vs. male).

La table complète contient 1 000 000 d'individus, et chaque autre table est constituée seulement d'une partie de ces individus.

**Exercice 7**

Écrivez un programme MapReduce permettant de calculer l'indice de masse corporelle (IMC) moyen, son minimum et son maximum, seulement pour les individus adultes, en les distinguant en fonction de leur sexe. Autrement dit, on veut obtenir en fin de traitement un triplet de valeurs (moyenne, minimum, maximum) pour les femmes, et un autre triplet pour les hommes. Le traitement doit être réalisé en une seule passe.

Notez qu'à la différence de l'exemple du décompte de mots, les données sont ici toutes contenues dans le même fichier (et non pas dans une multitude de fichiers). Il n'est donc pas nécessaire de surcharger input_data... mais cela reste une possibilité : tout dépend du reste du traitement.
Exemple de sortie obtenue pour le plus gros fichier :

*female 23.9335607785*
*female 0.734618916437*
*female 51.8082591419*
*male 25.0423629084*
*male 5.88235294118*
*male 53.2185663381*

La forme de ce fichier peut varier, en fonction de ce que renvoie reduce exactement : liste de trois éléments vs. triplet.

**Exercice 8**

Comme précédemment, expérimentez avec plusieurs configurations, en faisant varier la taille des données et le nombre d'esclaves. Donnez un tableau récapitulant les différents temps obtenus, et discutez-les.

## À rendre

Vous devez me rendre à la fois sur votre code source et votre rapport, qui explicitera à l'algorithme MapReduce par rapport au code réalisé. Votre rendu doit également être accompagné des sorties de vos algorithmes (sous forme de fichiers texte ou de captures d'écran).

Votre rendu se fera par courriel sous la forme d'une archive au **format .zip** à l'adresse suivante : richard.dufour@univ-nantes.fr