Le paradigme Map-Reduce

Les fonctions `map`, `reduce` et `filter` de la programmation fonctionnelle

Le principe de la programmation fonctionnelle est de n'écrire que des fonctions, et d'enchainer ces fonctions. On évite ainsi de faire appel à des variables globales, ce qui minimise les problèmes d'*effet de bord* (une variable est modifiée par un programme A, et cela influe sur le comportement d'un autre programme B, sans que l'auteur de B en soit conscient).

PYTHON

```python
def avec_effet_bord(A) :
    # A est une liste
    A[-1] = "*"
    return(A)

def sans_effet_bord(A) :
    # A est une liste
    B = [a for a in A]
    B[-1] = "*"
    return(B)

L1 = [1,2,3]

L1_bis = avec_effet_bord(L1)
print(L1)
print(L1_bis)

L2 = [1,2,3]

L2_bis = sans_effet_bord(L2)
print(L2)
print(L2_bis)
```

La programmation fonctionnelle est une idée ancienne en informatique qui a été à la naissance de langages tels que LISP. Voici un exemple de code LISP (moyennement lisible !) :

LISP

```lisp
(defun factorial (n &optional (acc 1))
  "Calcule la factorielle de l'entier n."
  (if (<= n 1)
      acc
    (factorial (- n 1) (* acc n))))
```

En `Python`, la programmation fonctionnelle s'appuie essentiellement sur les itérateurs et la fonction `map`.

La fonction `map`

La fonction `map` applique une fonction à un objet itérable et rend un itérateur.

PYTHON

```python
carre = lambda t:t*t

n = 15
x = map(carre,range(n))
```

Vérifier que l'objet `x` ainsi construit est bien un itérateur

PYTHON

```python
x = map(carre,range(n))
l = [carre(i) for i in range(n)]
```

Comparer les empreintes mémoires des objets `x` et `l` en utilisant la fonction `getsizeof` du module `sys`

La fonction `filter`

La fonction `filter` filtre un objet itérable suivant le résultat d'une fonction à valeurs booléenne.

```python
f1 = lambda t:t%2==0

x = map(carre,range(n))
y = filter(f1,x)
```

Que fait la fonction `f1` ?
Que va contenir l'objet `y` ?

Bien sûr, on peut tout grouper en une commande, plus ou moins lisible :

```python
y = filter(lambda t:t%2==0, map(lambda t:t*t , range(15)))
```

La fonction `reduce`

La fonction `reduce` s'applique à un objet itérable pour le réduire en appliquant successivement aux termes de cet objet une fonction binaire (qui prend deux paramètres en entrée). Par exemple, soit `f` une fonction binaire renvoyant un seul résultat, `reduce(f,[a,b,c,d])` va calculer `f(f(f(a,b),c),d)`.

```python
from functools import reduce
f2 = lambda t,u: t+u

x = map(carre,range(n))
y = filter(f1,x)
z = reduce(f2,y)
print(z)
```

Que fait la fonction `f2` ?
Que va contenir l'objet `z` ?

Exercice 1

Soit un échantillon $X = (x_1, ..., x_n)$, on veut calculer la variance

$$Var(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 = \left(\frac{1}{n}\sum_{i=1}^{n}x_i^2\right) - \mu^2$$

où μ désigne la moyenne de l'échantillon

$$\mu = \frac{1}{n}\sum_{i=1}^{n}x_i$$

Utiliser une programmation fonctionnelle pour écrire des fonctions prenant X en argument et rendant la moyenne et la variance.

```python
```

```
from random import uniform
from statistics import mean, variance

n = 1000
X = [uniform(2.5, 10.0) for i in range(n)]
print("Résultats attendus")
print(f"Moyenne : {mean(X):.3f}, Variance : {variance(X)*(n-1)/(n):.3f}")

print("\nRésultats obtenus")
res = calcul_moy_var(X)
print(f"Moyenne : {res[0]:.3f} - Variance : {res[1]:.3f}")
```

- Fonction 1 : la variable `n` peut être calculée au sein de la fonction, la moyenne et la variance peuvent être calculées en deux temps
- Fonction 2 : la variable `n` est calculée via la programmation fonctionnelle, la moyenne et la variance peuvent être calculées en deux temps
- Fonction 2bis : utilisation d'un seul mapper

Solution

Fonction 1

PYTHON

```
def calcul_moy_var_1(X) :
    n = len(X)

    func_map = lambda t:t
    mapper1 = map(func_map,X)

    somme = lambda t,u : (t+u)
    moy = reduce(somme,mapper1)/n

    carre = lambda t:t*t
    mapper2 = map(carre,X)

    var = reduce(somme,mapper2)/n - moy**2

    return(moy,var)
```

Fonction 2

PYTHON

```
def calcul_moy_var_2(X) :
    func_map = lambda t:(t,1)
    mapper1 = map(func_map,X)

    somme = lambda t,u : (t[0]+u[0],t[1]+u[1])
    div = lambda t,u : t/u

    moy = reduce(div,reduce(somme,mapper1))

    carre = lambda t:(t*t,1)
    mapper2 = map(carre,X)

    var = reduce(div,reduce(somme,mapper2)) - moy**2

    return(moy,var)
```

Fonction 2bis

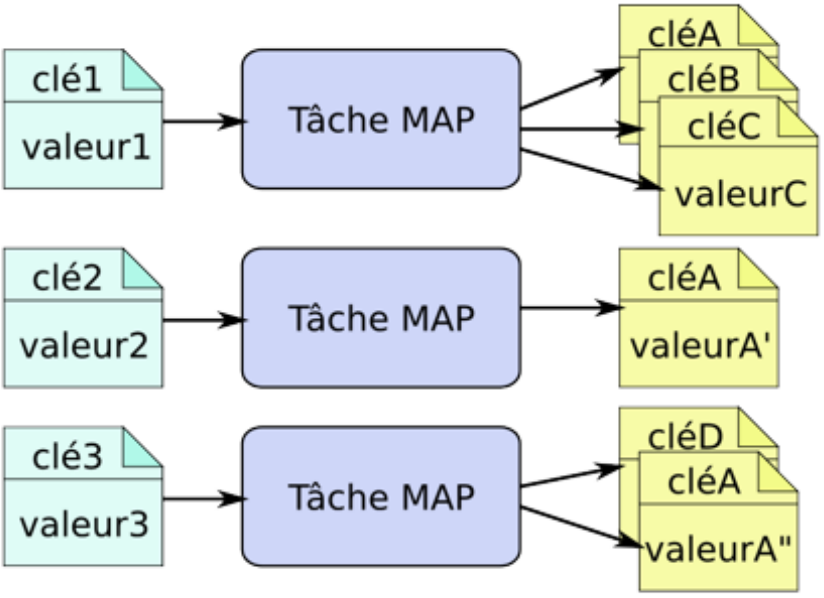
PYTHON

```
def calcul_moy_var_2b(X) :
    func_map = lambda t:(t*t,t,1)
    mapper = map(func_map,X)

    somme = lambda t,u : [a + b for a,b in zip(t,u)]

    tmp = reduce(somme,mapper)

    moy = tmp[1]/tmp[2]
    var = tmp[0]/tmp[2] - moy**2

    return(moy,var)
```

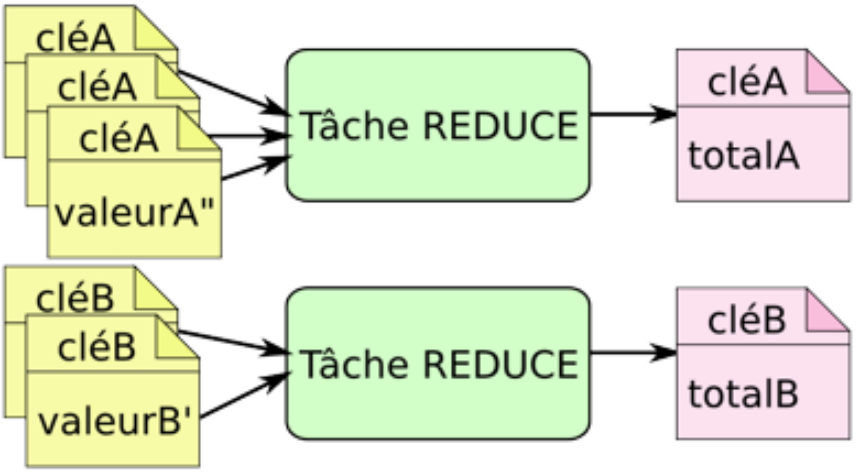
Le principe général de Map-Reduce

Le paradigme map-reduce a été introduit en 2004 par des ingénieurs de Google dans l'article : https://research.google.com/archive/mapreduce-osdi04.pdf. Il reprend des idées bien connues de la programmation fonctionnelle en les appliquant à du calcul massivement parallèle.

- Données d'entrée sous forme d'une collection clé-valeur, qui seront éventuellement réparties sur plusieurs machines.
- Mapper : programme qui prend un couple clé-valeur et rend une liste de couples clé-valeur



- Shuffle and Sort : étape gérée entièrement par le système hadoop ; les couples clé-valeur produits par les différentes instances du mapper et qui correspondent à une même clé sont regroupés et envoyés à une même machine sous la forme `clé:[valeur_1,…,valeur_n]` .
- Reducer : les données regroupées par l'étape shuffle sont traitées et rendent un couple clé-valeur qui est écrit dans un fichier.



- Plusieurs phases map-reduce peuvent être enchaînées, et il peut y avoir des reducer partiels (combiner) qui font la tâche du reducer sur la partie des données auxquelles ils ont accès.

Une implémentation du wordcount

On va expérimenter une version de l'algorithme de wordcount en utilisant des tuyaux unix.

- Ecrire un court fichier texte

texte

```txt
tata toto tutu titi tata
toto tata tititi
titi
```

- Ecrire un premier fichier python :

map

```python
#! /usr/bin/env python3

import sys
import re
motif=re.compile("\w+")

for ligne in sys.stdin:
    liste_mots = motif.findall(ligne.strip().lower())
    for mot in set(liste_mots) :
        print(f"{mot}\t{liste_mots.count(mot)}")
```

- Rendre le fichier `map` exécutable puis effectuer la commande unix :

```shell
cat texte | ./map
```

On a ainsi appliqué la fonction `map` à notre fichier texte.

- La partie `shuffle and sort` consiste simplement à appliquer la fonction unix `sort` à la sortie de la commande précédente :

```shell
cat texte | ./map | sort
```

- Partie `reduce` : Ecrire en python un programme `reduce` qui rend la liste des mots avec leur nombre d'apparition.
  La commande unix suivante :

```shell
cat texte | ./map | sort | ./reduce
```

doit fournir les résultats ci-dessous :

```shell
tata    3
titi    2
tititi  1
toto    2
tutu    1
```

▾ Solution

reduce

```python
#! /usr/bin/env python3

import sys

cle_prec,nb_tot = None,0

for ligne in sys.stdin:
    cle,valeur = ligne.split('\t')
    if cle == cle_prec :
        nb_tot += int(valeur)
    else :
        if cle_prec != None:
            print(f"{cle_prec}\t{nb_tot}")
        cle_prec = cle
        nb_tot = int(valeur)
        cle_prec = cle
if cle_prec != None:
    print(f"{cle_prec}\t{nb_tot}")
```

Dans la solution proposée, on notera qu'on utilise fortement le fait que l'entrée est déjà triée (étape `shuffle and sort`), et que sans ce tri préalable le résultat sera beaucoup moins intéressant.

Ce que nous venons de faire est juste une démonstration de la faisabilité d'un algorithme basé sur des fonctions `map` et `reduce` . Elle n'est pas du tout utilisable dans la pratique car sous cette forme, tout tourne dans un unique processus, et n'est pas parallélisable.

Exercice 2

Ecrire des fonctions `map` et `reduce` , de telle façon que l'on prenne en entrée un texte, comme on l'a fait précédemment, et que cela rende les nombres de lettres, de mots, et de lignes que contient ce texte.

▾ Pour tester

Le fichier texte `hadoop_wikipedia` (https://math.univ-angers.fr/~badreau/data/hadoop_wikipedia) (tiré de la page Wikipédia sur Hadoop) associé à la commande unix :

```shell
cat hadoop_wikipedia | ./map | sort | ./reduce
```

doit renvoyer les résultats suivants :

```shell
```

```
lettres 2335
lignes  34
mots    437
```

▼ Solution

Il suffit de modifier le fichier `map` :

map

PYTHON

```python
#! /usr/bin/env python3

import sys
import re

mot = re.compile("\w+")

for ligne in sys.stdin:
    liste_mots = mot.findall(ligne.strip().lower())
    print("lignes\t1")
    print(f"mots\t{len(liste_mots)}")
    print(f"lettres\t{sum([len(mot) for mot in liste_mots])}")
```

La bibliothèque `mrjob` de `python`

`mrjob` a été développé par la société Yelp, qui publie des avis participatifs sur les commerces locaux ; pour ce travail, elle doit gérer une très grosse quantité d'avis et en extraire des statistiques. La bibliotèque `mrjob` est prévue pour lancer des commandes map-reduce sur un cluster `hadoop` local, ou sur un cloud comme AWS de Amazon. Elle permet aussi un traitement en local, uniquement en python, afin de permettre la mise au point des algorithmes. C'est ce mode que nous allons utiliser ici.

On peut trouver ici https://www.youtube.com/watch?v=txT_sA1malk une vidéo d'une conférence donnée au PyCon 2014, par Jim Blomo, un des créateurs de `mrjob` , où il explique un certain nombre de choses sur ce package.

Une documentation sur `mrjob` est disponible ici (https://mrjob.readthedocs.io/en/latest).

Par défaut, un programme `mrjob` consiste en une classe, qui hérite de la la classe `MRJob` et qui contient au moins deux méthodes `mapper` et `reducer` . Le programme contient aussi une partie `if __name__ == '__main__':` pour exécuter la classe.

Une implémentation du wordcount avec `mrjob`

Voici un exemple de classe `MRJob` , comptant la fréquence d'apparition de chaque mot d'un texte :

wordcount_mrjob.py

PYTHON

```python
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w]+")

class MRWordFreqCount(MRJob):
    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def reducer(self, word, counts):
        yield word, sum(counts)

if __name__ == '__main__':
    MRWordFreqCount.run()
```

On peut tester ce programme en l'associant au fichier `texte` précédent par la commande unix :

SHELL
```
python wordcount_mrjob.py texte
```

`texte` peut aussi être remplacé par un répertoire contenant plusieurs fichiers textes. Par défaut, le résultat sort sur la sortie standard, mais on peut bien sûr rediriger cette sortie standard vers un fichier :

SHELL
```
python wordcount_mrjob.py texte > wordcount_mrjob.out
```

On peut améliorer le programme en rajoutant dans la classe une méthode `combiner`, qui permet d'effectuer une partie des opérations de réduction de manière locale sur la même machine qui aura effectué les `mapper` ; autrement dit, la somme globale est effectuée à partir de sommes partielles et non directement. Cette opération permet d'optimiser l'exécution de l'algorithme en réduisant le volume de données échangées.

wordcount_mrjob_bis.py

PYTHON
```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w]+")

class MRWordFreqCount(MRJob):
    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

     def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)


if __name__ == '__main__' :
    MRWordFreqCount.run()
```

Dans l'exemple indiqué ici, le `combiner` est exactement le même que le `reducer`, mais cela n'a aucune raison d'être le cas en général.

Le combiner doit être utilisé uniquement avec des fonctions commutatives et associatives.

Soit un (petit) échantillon `X` que nous allons insérer dans le fichier texte suivant :

sample_X

TEXTE
```
0
1
2
3
4
```

Le programme suivant calcule la moyenne d'un échantillon par deux algorithmes de type map-reduce : l'un utilise un `combiner`, l'autre non.

exemples_combiner_mrjob.py

PYTHON

```python
#! /usr/bin/env python3

from mrjob.job import MRJob

class moyenne_sansCombiner(MRJob):
    def mapper(self, _, x):
        yield _, float(x)

    def reducer(self, _, valeurs):
        L_valeurs = list(valeurs)
        yield "Moyenne (sans combiner) : ", f"{sum(L_valeurs)/len(L_valeurs):.3f}"

class moyenne_avecCombiner(MRJob):
    def mapper(self, _, x):
        yield _, float(x)

    def combiner(self, _, valeurs):
        L_valeurs = list(valeurs)
        yield _, sum(L_valeurs)/len(L_valeurs)

    def reducer(self, _, valeurs):
        L_valeurs = list(valeurs)
        yield "Moyenne (avec combiner) : ", f"{sum(L_valeurs)/len(L_valeurs):.3f}"

if __name__ == '__main__':
    moyenne_sansCombiner.run()
    moyenne_avecCombiner.run()
```

On peut enregistrer les résultats de ces fonctions dans un fichier texte :

SHELL

```
python exemples_combiner_mrjob.py sample_X > test_combiner.out
```

Pouvez-vous prédire avec certitude les résultats obtenus par ces deux fonctions ? Pourquoi ?

Un `mapreduce` avec plusieurs étapes

Pour des applications moins élémentaires que le simple wordcout, il faudra envisager un programme combinant plusieurs étapes de mapreduce. Dans ce cas, il faudra définir plusieurs opérateurs `map` et `reduce` et décrire un schéma d'opération indiquant l'ordre des commandes. Voici l'allure d'un tel programme ayant deux étapes, l'une comprenant un `mapper` , un `combiner` et un `reducer` , et l'autre ayant seulement un `reducer` :

PYTHON

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class MaClasse(MRJob):
    def mapper1(self, cle, valeur):
        # traitement mapper
        yield cle_m1, valeur_m1

    def combiner(self, cle, valeurs):
        # traitement combiner
        yield cle_c, valeur_c

    def reducer1(self, cle, valeurs):
        # traitement reducer1
        yield cle_r1, valeur_r1

    def reducer2(self, cle, valeurs):
        # traitement reducer2
        yield cle_r2, valeur_r2

    def steps(self):
        return [
           MRStep(mapper=self.mapper1,
                  combiner=self.combiner1,
                  reducer=self.reducer1),
           MRStep(reducer=self.reducer2)
        ]

if __name__ == '__main__'   :
    MaClasse.run()
```

Exercice 3

Ecrire un programme mrjob prenant en entrée un texte, et sortant le mot le plus utilisé dans ce texte parmi les mots de plus de 5 lettres.

▾ Pour tester

Le fichier texte `hadoop_wikipedia` (https://math.univ-angers.fr/~badreau/data/hadoop_wikipedia) associé au programme `plus_frequent.py` par la commande unix :

```shell
python plus_frequent.py hadoop_wikipedia
```

SHELL

doit renvoyer le résultat :

```shell
"hadoop"    20
```

SHELL

▾ Solution

plus_frequent.py

PYTHON

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

motif = re.compile("\w{5,}")

class MRWordFreqCount(MRJob):
    def mapper(self, _, line):
        for word in motif.findall(line):
            yield word.lower(), 1

    def reducer1(self, word, counts):
        yield None, (sum(counts), word)

    def reducer2(self, _, reduc1):
        count,word = max(reduc1)
        yield word, count

    def steps(self):
        return [
           MRStep(mapper=self.mapper,
                  reducer=self.reducer1),
           MRStep(reducer=self.reducer2)
        ]

if __name__ == '__main__':
    MRWordFreqCount.run()
```

Exercice 4

Un exercice de traitement de données avec `mrjob`*. Cet exercice est extrait de la conférence citée au début de ce paragraphe.*

On considère les données anonymisées des visites du site de Microsoft (https://math.univ-angers.fr/~badreau/data/anonymous-msweb.zip) ainsi que le fichier (https://math.univ-angers.fr/~badreau/data/anonymous-msweb.info) décrivant ces données.

Les fichiers `anonymous-msweb.data` et `anonymous-msweb.test` contiennent plusieurs types de lignes :

- des lignes commençant par un "A" (attribut line):

```txt
A,1226,1,"MS Schedule+ Support","/msschedplussupport"
```

Une telle ligne dit que la page dont l'ID est 1226 a pour titre "MS Schedule+ Support" et se trouve à l'emplacement "/msschedplussupport" sur le serveur web.

- des groupes de lignes :

```txt
C,"10062",10062
V,1008,1
V,1007,1
V,1034,1
```

Un tel groupe dit que l'utilisateur 10062 (qui a été anonymisé) a visité les pages d'ID : 1008, 1007, 1034.

- Enregistrer et extraire les documents dans un dossier `anonymous-msweb` .
- Question 1 : Afficher les ID et le nombre de visites des pages qui ont été visitées plus de 600 fois.
- Question 2 : Afficher les titres des 5 pages les plus visitées.

- Question 3 : Afficher les 15 utilisateurs qui ont fait le plus de visites.
- Question 4 : Afficher les 5 utilisateurs qui ont fait le plus de visites, avec pour chacun la liste des noms des pages visitées.

Le but du TP est de répondre à ces différentes questions de deux manières différentes :

- Version 1 : Les données n'étant pas gigantesques ici (car correspondant à une très courte durée d'enregistrement), vous pouvez utiliser des techniques standard (python simple, ou pandas).
- Version 2 : Il serait aussi intéressant que vous proposiez une technique utilisant `map-reduce` avec `mrjob`, afin de permettre un passage à l'échelle.

La réponse à chaque question devra pouvoir être obtenue par la commande unix suivante :

```shell
python QuestionX.py anonymous-msweb
```

> Pour les questions 3 et 4, utiliser d'abord un algorithme de pré-traitement permettant au programme `mrjob` de traiter les informations d'un utilisateur `C` sur une même ligne (par exemple, ramener les lignes suivantes commençant par `V` sur la même ligne) :

```txt
C,"10062",10062,V,1008,1,V,1007,1,V,1034,1
```

Solution (https://math.univ-angers.fr/~badreau/data/Exercice4.zip)

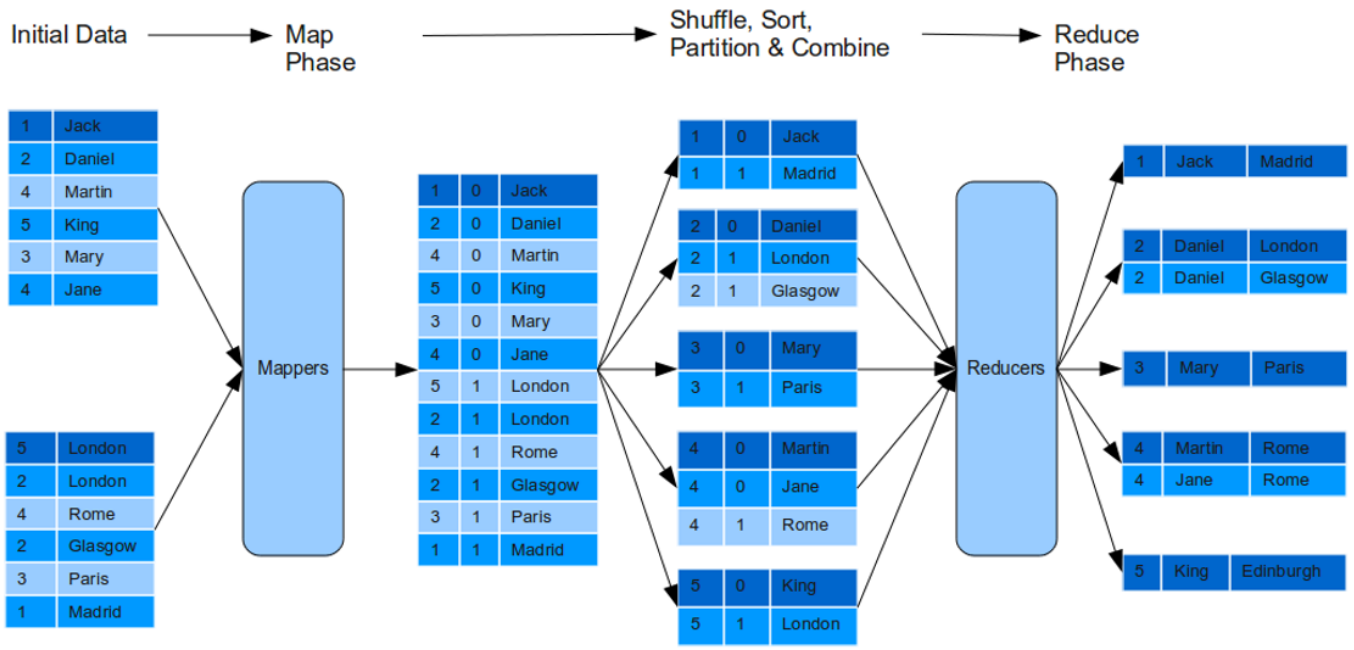
Compléments sur MapReduce

Les jointures avec MapReduce

Les jointures sont à la base du travail avec des BDD relationnelles. Le schéma général est :

```sql
SELECT * FROM table1, table2 WHERE table1.attribut1 = table2.attribut2
```

Il s'agit d'une opération très couteuse en mémoire, mais qui reste indispensable, même dans un cadre NoSQL. On peut l'éffectuer par une opération de map-reduce, suivant le schéma :



Index (https://math.univ-angers.fr/~badreau/)

Last updated 2023-10-11 13:16:24 +0200