« TECHNO DU BIG-DATA »

Stéphane Derrode, Dpt MI -
Stephane.derrode@ec-lyon.fr

Sommaire

1. Big Data et parallélisme
2. Modèle MapReduce
3. HDFS
4. MapReduce sur cluster Hadoop
5. Générateurs et Itérateurs en Python
6. Ecosystème Hadoop

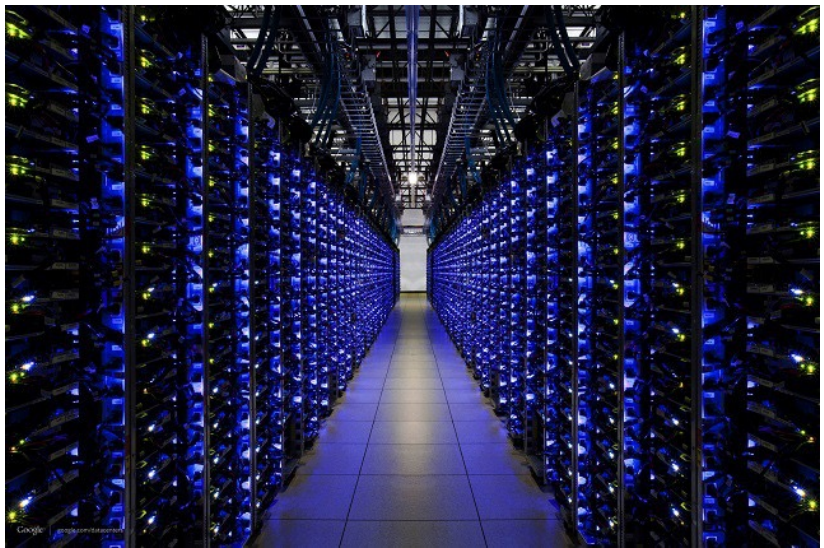
BIG DATA ET PARALLÉLISME

« Big Data »

- Exemples:
  - Google, 2008: 20 PB / jour, 180 GB / job
  - Web index : 50 Milliard de pages, 15 PB
  - Large Hadron Collider (LHC)@CERN : 15PB / année

- Capacité d'un (gros) serveur
  - RAM : 256 GB
  - DD : 24 TB
  - Vitesse de transfert du DD : 100 MB/s

- Solution : parallélisme
  - 1 serveur : 8 DD, lire le web en 230 jours
  - Hadoop cluster @ Yahoo : 4000 serveurs, lire le web en // = 1h20



Google data center

Tolérance aux erreurs

- Le problème du parallélisme
  - 1 serveurs bug tous les quelques mois
  - 1000 serveurs -> temps moyen avant bug < 1 jour

- Un « gros » job peut prendre plusieurs jours
  - Une panne matériel : c'est donc la normalité!
  - Parallélisme : impossible de relancer partiellement en cas de panne
  - Point de contrôle, réplication : difficile à implémenter correctement

- Plateformes Big data : tout le monde doit pouvoir écrire des programmes
  - Encapsule le parallélisme
  - Encapsule la tolérances aux pannes
  - Codé une fois par des experts, profitables à tous (non experts)

MODÈLE MAP REDUCE

Inspiré de la programmation fonctionnelle

Deux fonctions très simples inspirées de la programmation fonctionnelle:

- **Transformation : map**
  - map(f, [x_1, …, x_n]) = [f(x_1), …, f(x_n)]
  - Exemple : map(2*, [1, 2, 3]) = [(2*, 1), (2*, 2), (2*, 3)] = [2, 4, 6]
- **Agrégation : reduce**
  - reduce(f, [x_1, …, x_n]) = f(x_1, f(x_2, …, f(x_{n-1}, x_n))
  - Exemple : reduce(+, [2, 4, 6]) = (+2 (+4 6)) = 12

Ces fonctions sont génériques car elles prennent en paramètre une fonction : le développeur fournit les fonctions.

- map(toUpperCase, [‘hello’, ‘data’]) = [‘HELLO’, ‘DATA’]
- reduce(max, [3, 45, 27]) = 45

Les données sont toujours représentées par des paires (clé, valeur)

Une clé peut être de n'importe quel type

- **('Hello', 17)**
  - 'Hello' est la clé (text)
  - 17 est la valeur (int)

Lorsque les données ne sont pas des couples (clé, valeur)

- Un texte est représenté par (numéro de ligne, contenu de la ligne)

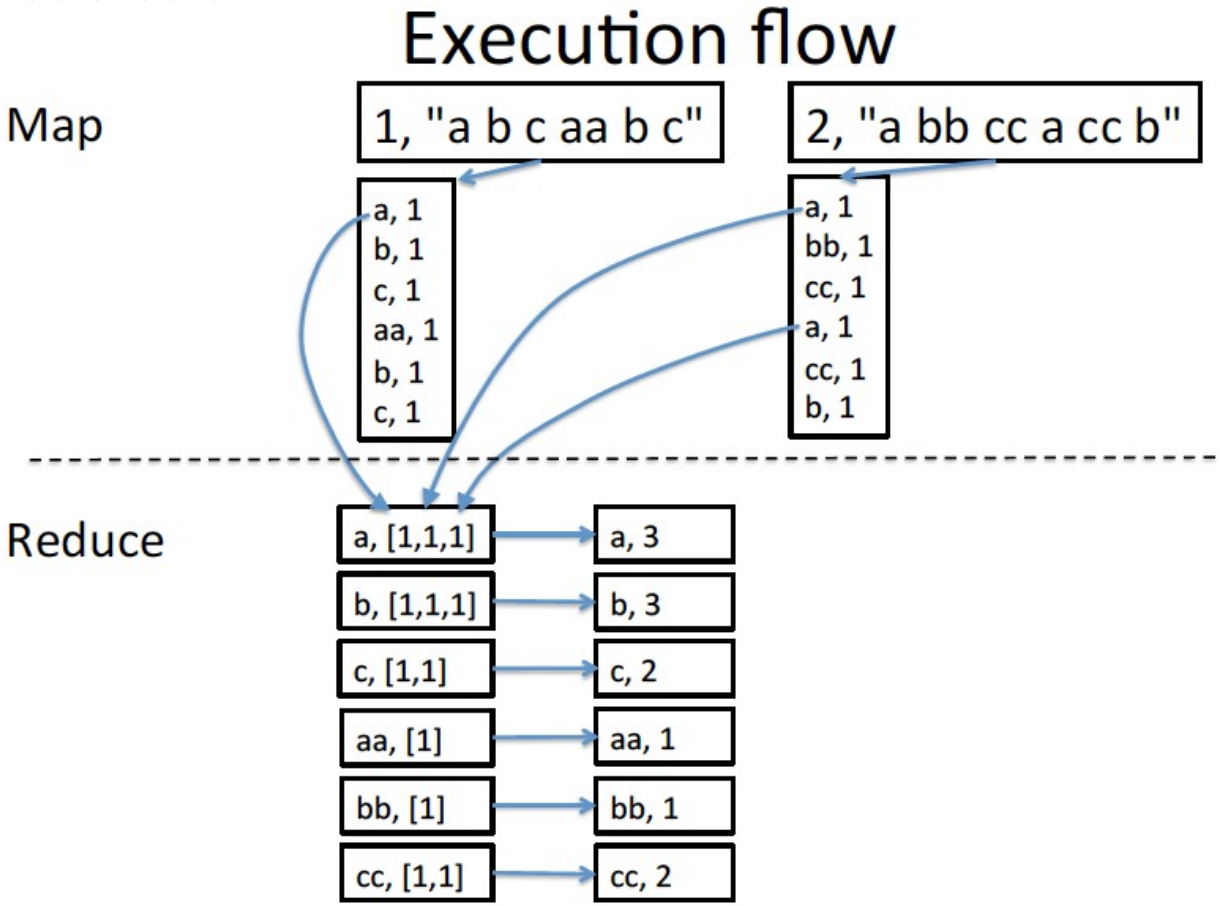
Map-Reduce appliqué à des couples (clé, valeur)

- **Map**, *f* est appliquée sur chaque couple **indépendamment**
  f(clé, valeur)→ list(clé, valeur)
- **Reduce**, f est appliquée sur **toute les valeurs** de même clé
  f(clé, liste(valeur)) → list(clé, valeur)

Les types des clés et des valeurs n'ont pas besoin d'être les mêmes en entrée et en sortie.

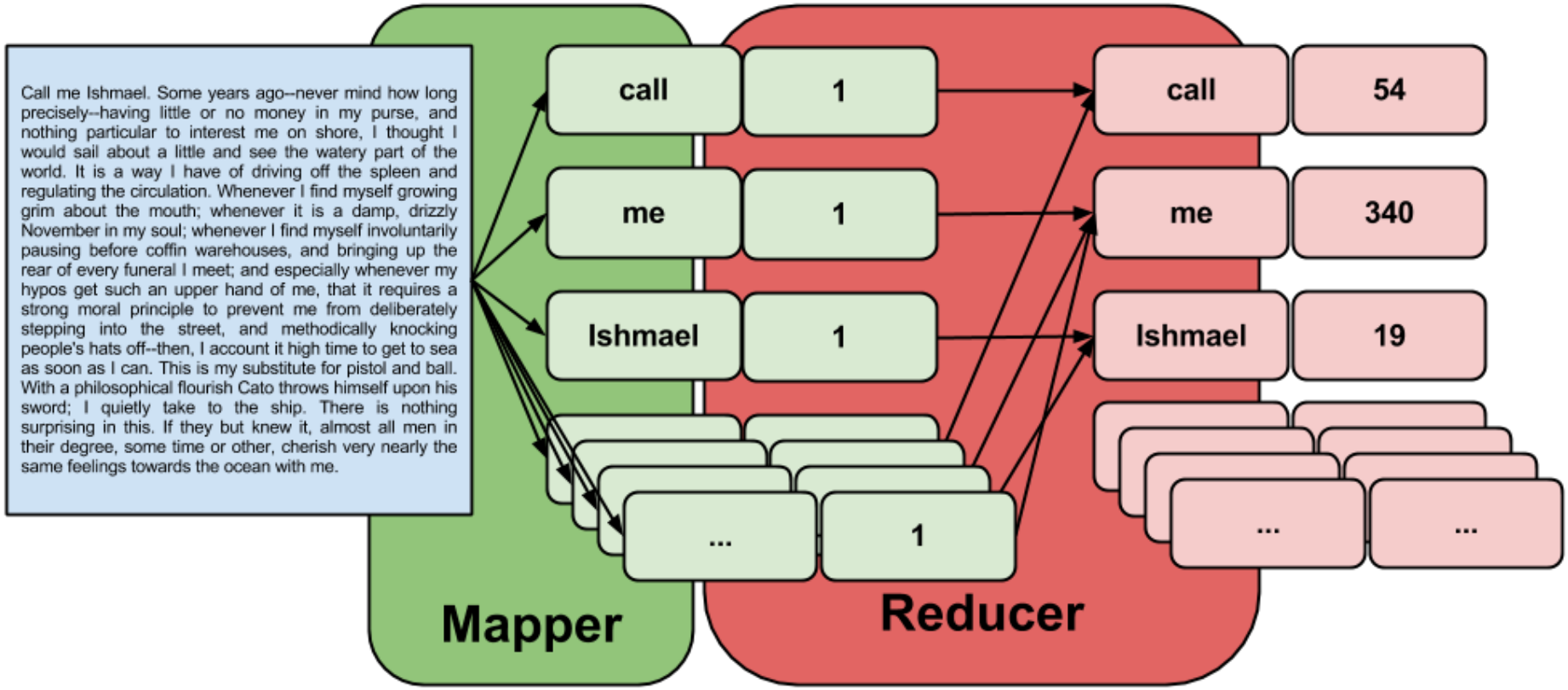
Exemple : comptage de la fréquence d'un mot

- Données d'entrée: un fichier de 2 lignes
  - 1, 'a b c aa b c'
  - 2, 'a bb cc a cc b'



Map-Reduce famous Word Count

Le ‘Hello world’ du map-reduce



Hadoop map-reduce : natif en Java mais connecteur Python

Word Count in python (map)

```
#!/usr/bin/env python3
# fichier wc_mapper.py

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line=line.strip()
    # split the line into words
    words=line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        # tab-delimited; the trivial word count is 1
        print(word,'\t1')
```

Word Count in python (reduce)

```
#!/usr/bin/env python3
# fichier wc_reducer.py
import sys

current_word=None
current_count=0
word=None
for line in sys.stdin:
    line=line.strip()
    word, count=line.split('\t', 1)
    try:
        count=int(count)
    except ValueError:
        continue

    if current_word==word:
        current_count+=count
    else:
        print(current_word, '\t', current_count)
        current_count=count
        current_word=word

print(current_word, '\t', current_count)
```

Démonstration : en local (sur sa machine)

```
>> echo "foo foo quux labs quux" | ./wc_mapper.py
>> echo "foo foo quux labs quux" | ./wc_mapper.py | sort
>> echo "foo foo quux labs quux" | ./wc_mapper.py | sort |
./wc_reducer.py

>> wget http://www.textfiles.com/etext/FICTION/dracula
>> more dracula

>> cat dracula | ./wc_mapper.py
>> head -n 20 dracula | ./wc_mapper.py| sort | ./wc_reducer.py
```

HDFS

Hadoop Distributed File System

Système de fichiers distribués (HDFS)

- Objectifs:
  - Tolérant aux erreurs (redondance)
  - Performant (accès parallèle)
- Fichiers volumineux
  - Lecture et écriture séquentielle
- Traitement de données « au plus près »

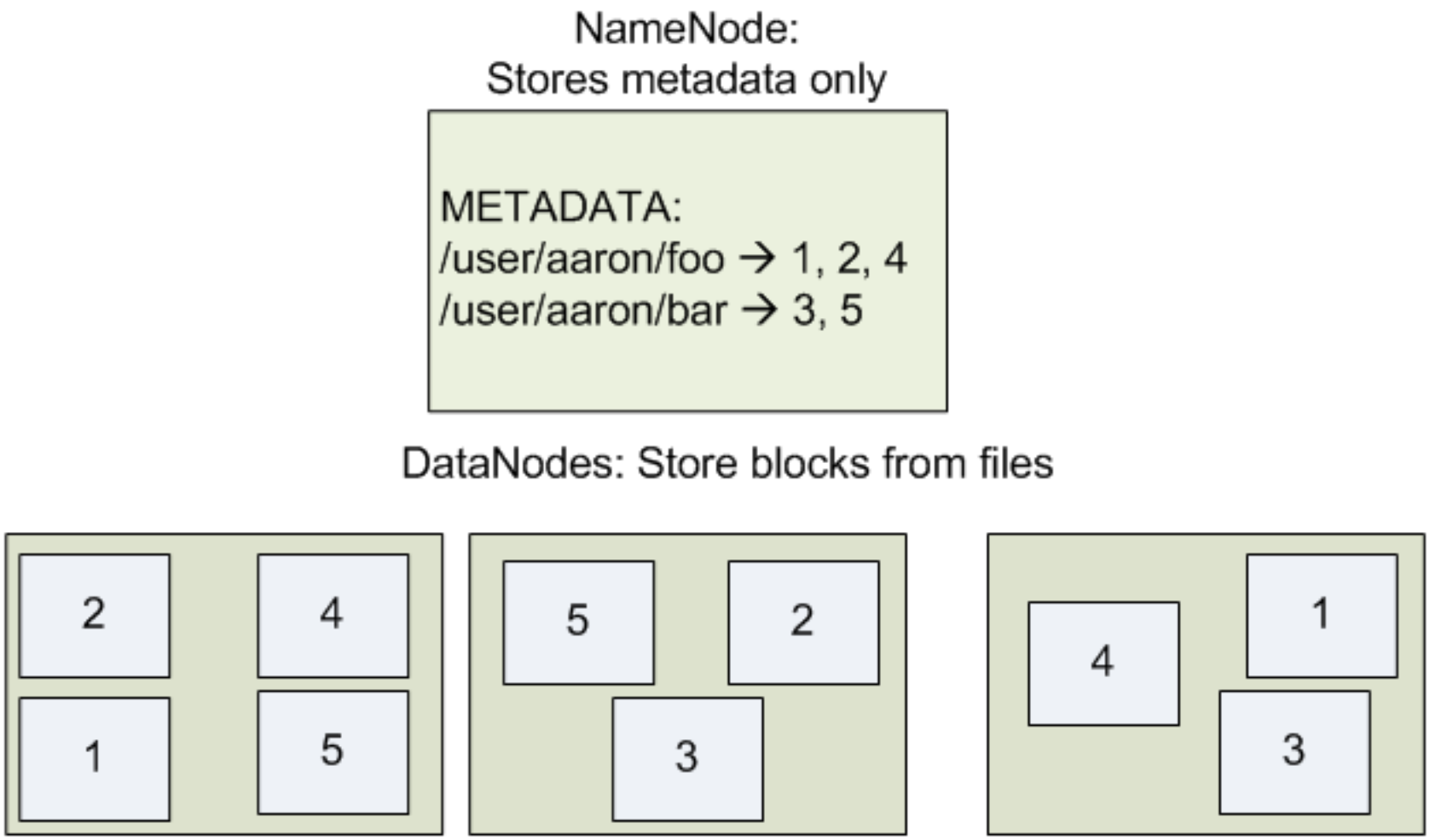
  Les données sont stockées sur les machines qui les traitent
  - Pour un meilleur usage des machines
  - Pour éviter les transferts réseaux (lag)
- Les données sont organisées en fichiers et répertoires
  - Mime les systèmes de gestion de fichiers standards
  - Les fichiers sont découpés en blocks (64MB) et éparpillés sur les serveurs avec réplication (3 fois par défaut)
  - Si possible, traite les données sur les machines où elles sont stockées.

Architecture « maitre / esclave »

- Un maitre : le « NameNode »
  - Gère les noms de fichiers, les droits d'accès…
    Stocke les metadata associées aux fichiers
    Garde tout en mémoire RAM (maximum : 60M objets and 16 GO)
  - Supervise les opérations sur les fichiers et les blocks
  - Supervise la santé du système (échecs, crash), et équilibre les charges

- Des milliers d'esclaves : les « DataNode »
  - Stocke les données (blocks).
    - Les données ne transitent jamais par le « NameNode ».
  - Réalise les opérations de lecture et d'écriture.
  - Réalise les copies (replications) ordonnées par le « NameNode »
  - Vérifie régulièrement la santé du « NameNode »
  - Rapporte au « NameNode » si des blocks sont corrompus (checksum)

Architecture « maitre / esclave », « master / slave »



https://developer.yahoo.com/hadoop/tutorial/module2.html

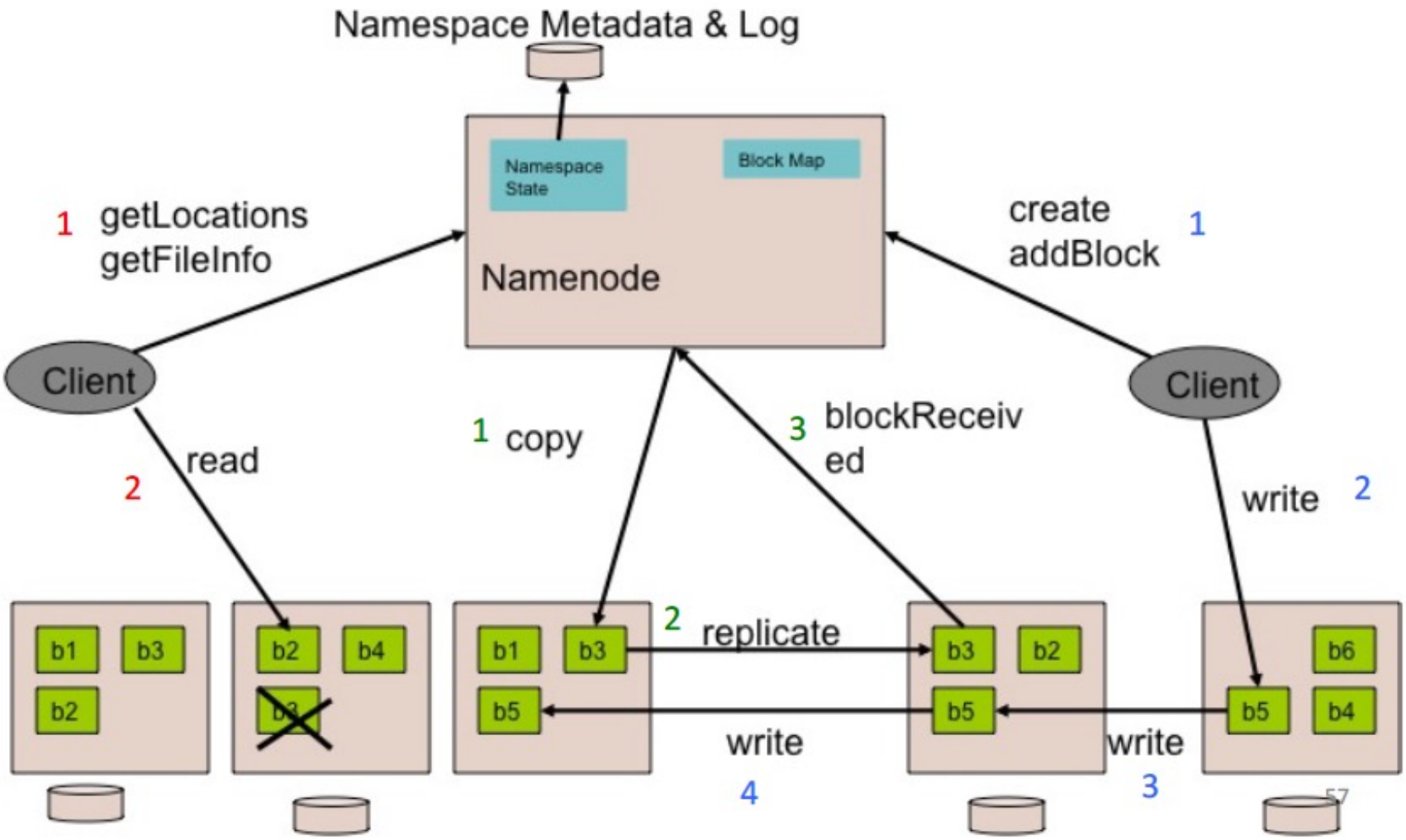
Ecrire un fichier sur HDFS

1. Le client envoie une demande au *NameNode* pour créer un nouveau fichier.
2. Le *NameNode* vérifie les autorisations du client et les conflits tels que existence du fichier.
3. Le *NameNode* choisit les *DataNodes* pour stocker les fichiers et leurs réplicas. Il constitue une pipeline de *DataNodes*.
4. Les blocks sont alloués sur ces *DataNodes*.
5. Le flux de donné est envoyé depuis le client vers le premier *DataNode* du pipeline.
6. Chaque *DataNode* transfert les données reçues au *DataNode* suivant dans le pipeline.

Lire un fichier sur HDFS

1. Le client envoie une demande au *NameNode* pour lire un fichier
2. Le *NameNode* vérifie que le fichier existe et construit une liste de *DataNodes* contenant les premiers blocks.
3. Pour chaque block, le *NameNode* envoie l'adresse des *DataNodes* les hébergeant (la liste est ordonnée en fonction de la proximité au client).
4. Le client se connecte au *Datanode* le plus près contenant le premier block du fichier
5. La fin de lecture du block
   - Ferme la connexion au *Datanode*.
   - Ouvre la connexion au *Datanode* contenant le block suivant.
6. Lorsque tous les blocks sont lus, le client questionne le *NameNode* sur les blocks suivants.

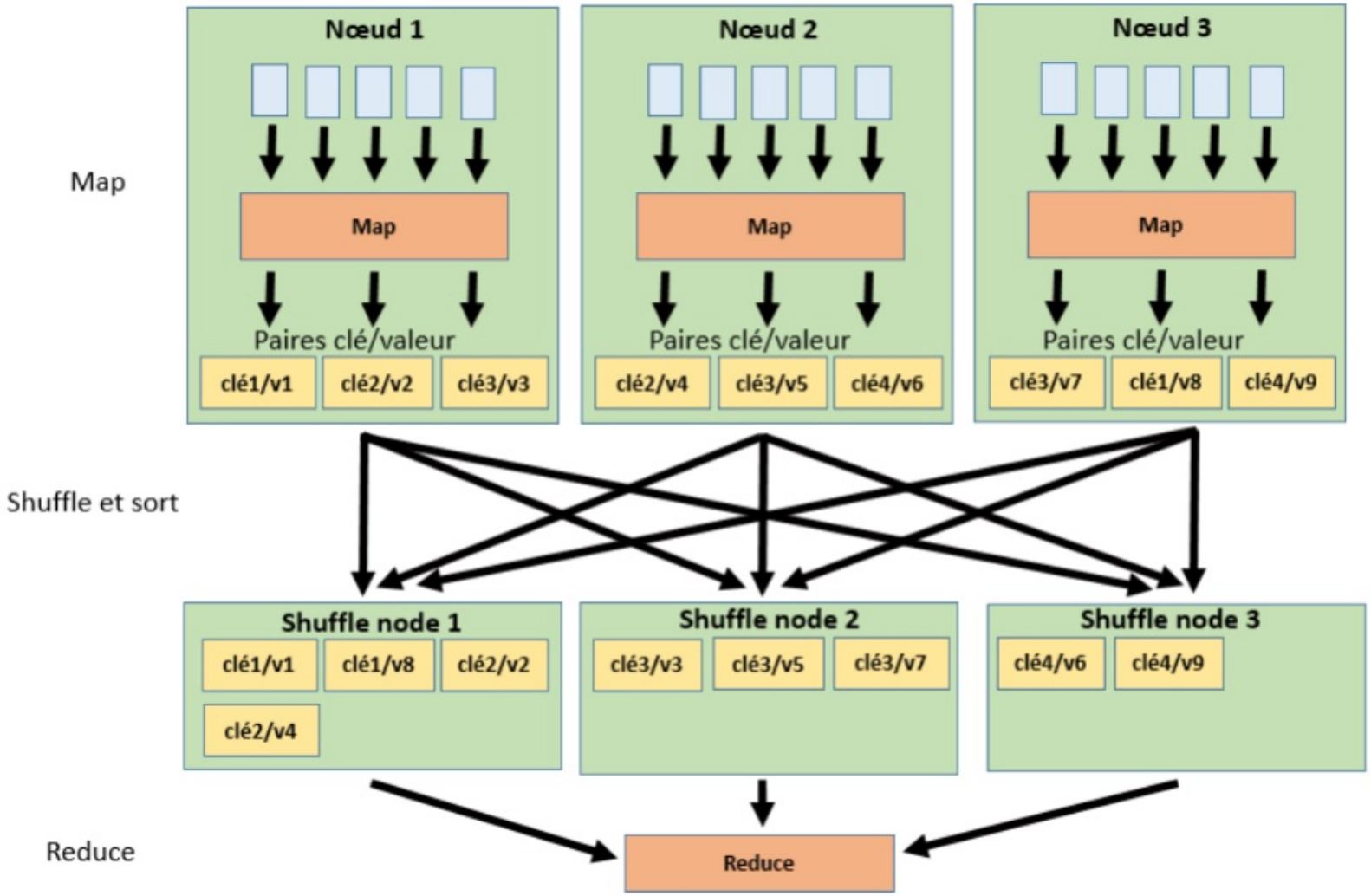
Protocol de lecture/écriture/réplication sur HDFS



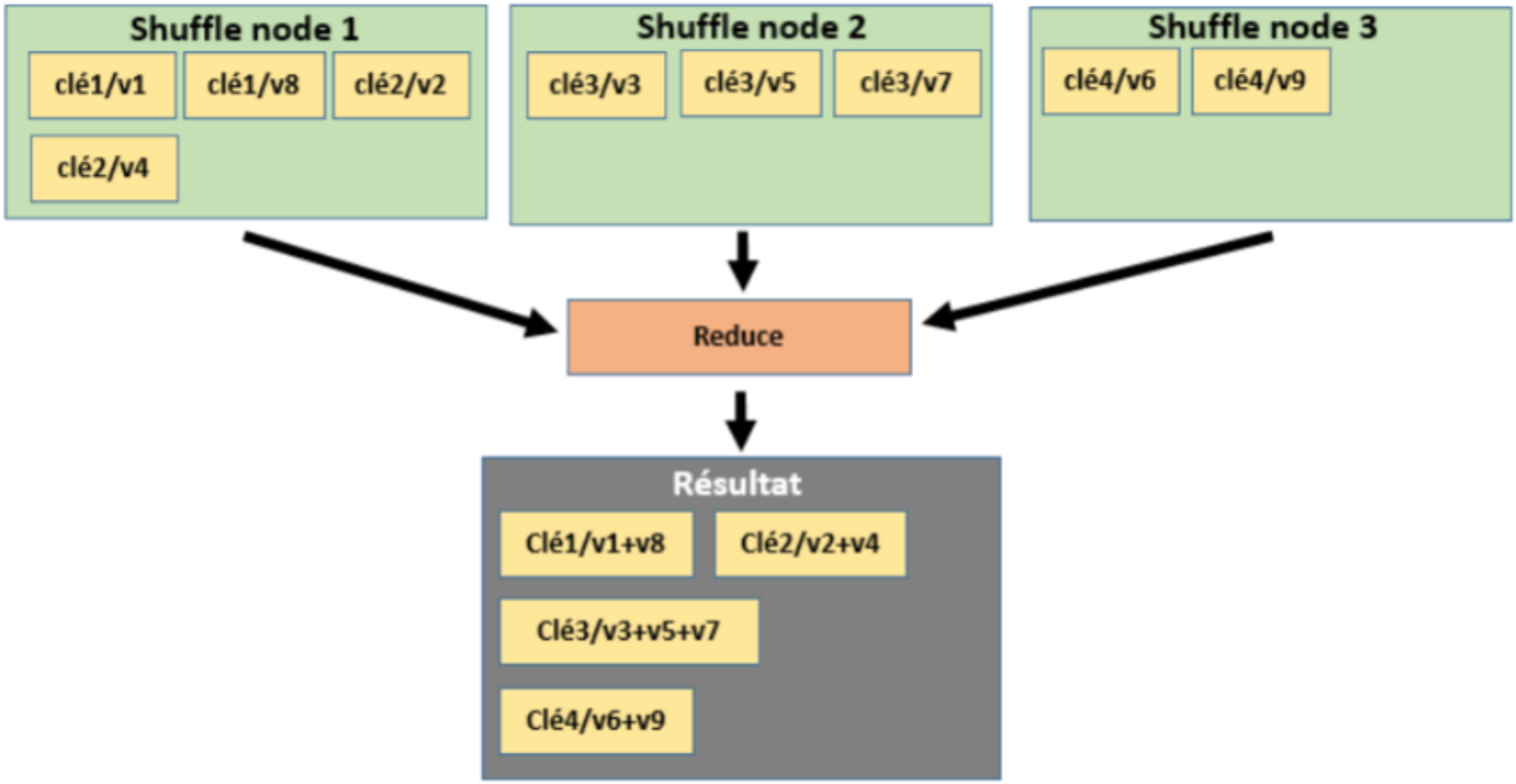
MAP-REDUCE

Map Reduce sur cluster Hadoop

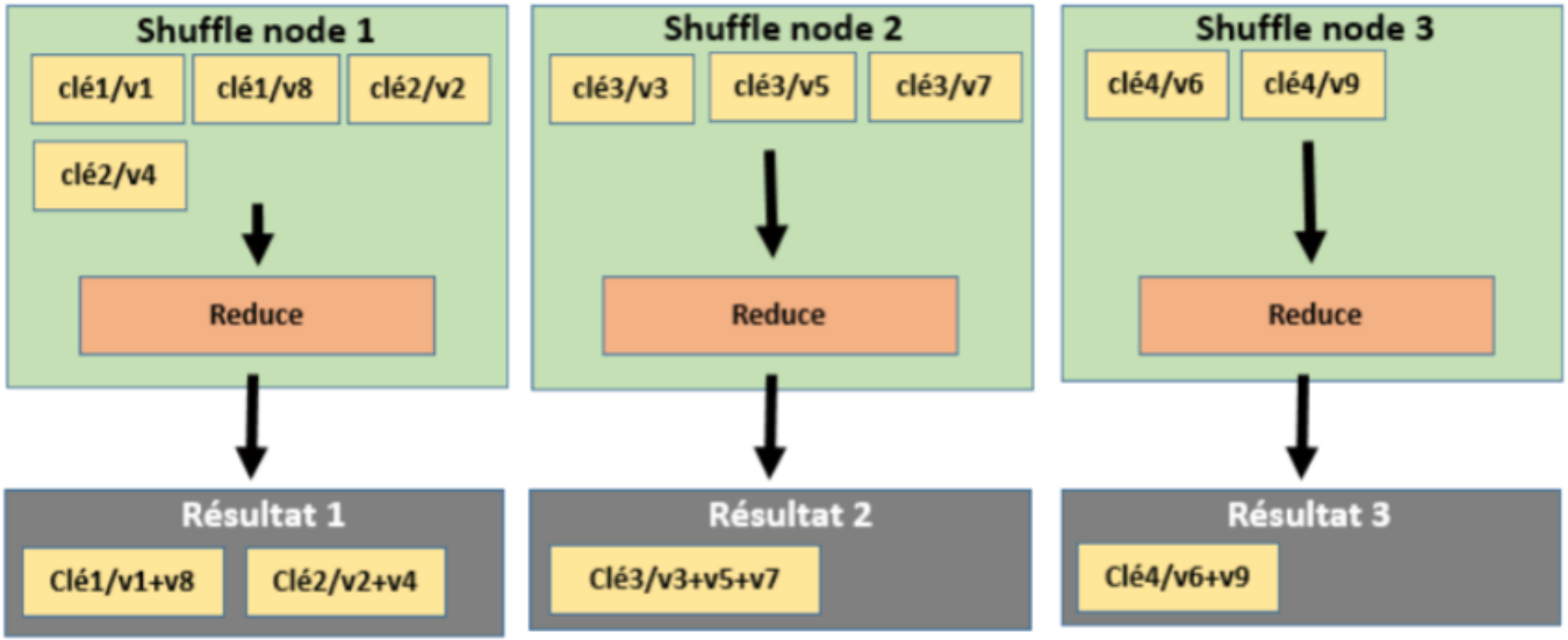
Map Reduce : shuffling and sorting



Map Reduce : reducing



Map Reduce : multiple reducing



WordCount sur un cluster HDFS

```
>> export STREAMINGJAR='/…/hadoop-streaming-2.7.2.jar'

>> hadoop jar $STREAMINGJAR -files wc_mapper.py,wc_reducer.py -mapper
wc_mapper.py -reducer wc_reducer.py -input livres/dracula -output sortie
```

hadoop-streaming.jar : librairie Java utilisée lors de l'exécution d'opérations MapReduce. Elle établit un lien entre Hadoop et le code externe MapReduce que vous fournissez.

-files : indique à Hadoop que les fichiers spécifiés sont nécessaires pour effectuer cette tâche MapReduce, et qu'ils doivent être copiés sur tous les nœuds de travail.

-mapper : indique à Hadoop quel fichier doit être utilisé comme *mappeur*.

-reducer : indique à Hadoop quel fichier doit être utilisé comme *reduceur*.

-input : indique le nom du fichier d'entrée sur lequel s'applique la tache (si vous spécifiez un répertoire, le Job s'appliquera à tous les fichiers du répertoire).

-output : le répertoire sur lequel la sortie sera écrite. Ce répertoire sera créé par la tâche. Il doit être détruit entre deux exécutions, sous peine d'un message d'erreur lors de l'exécution (>> hdfs dfs -rm -r -f sortie)

Nombre de taches en parallèle :

-D mapred.reduce.tasks=3

-D mapred.map.tasks=5

Hadoop n'honore pas nécessairement mapred.map.tasks (cela est considéré comme une éventuelle aide pour choisir automatiquement le nombre adéquat).

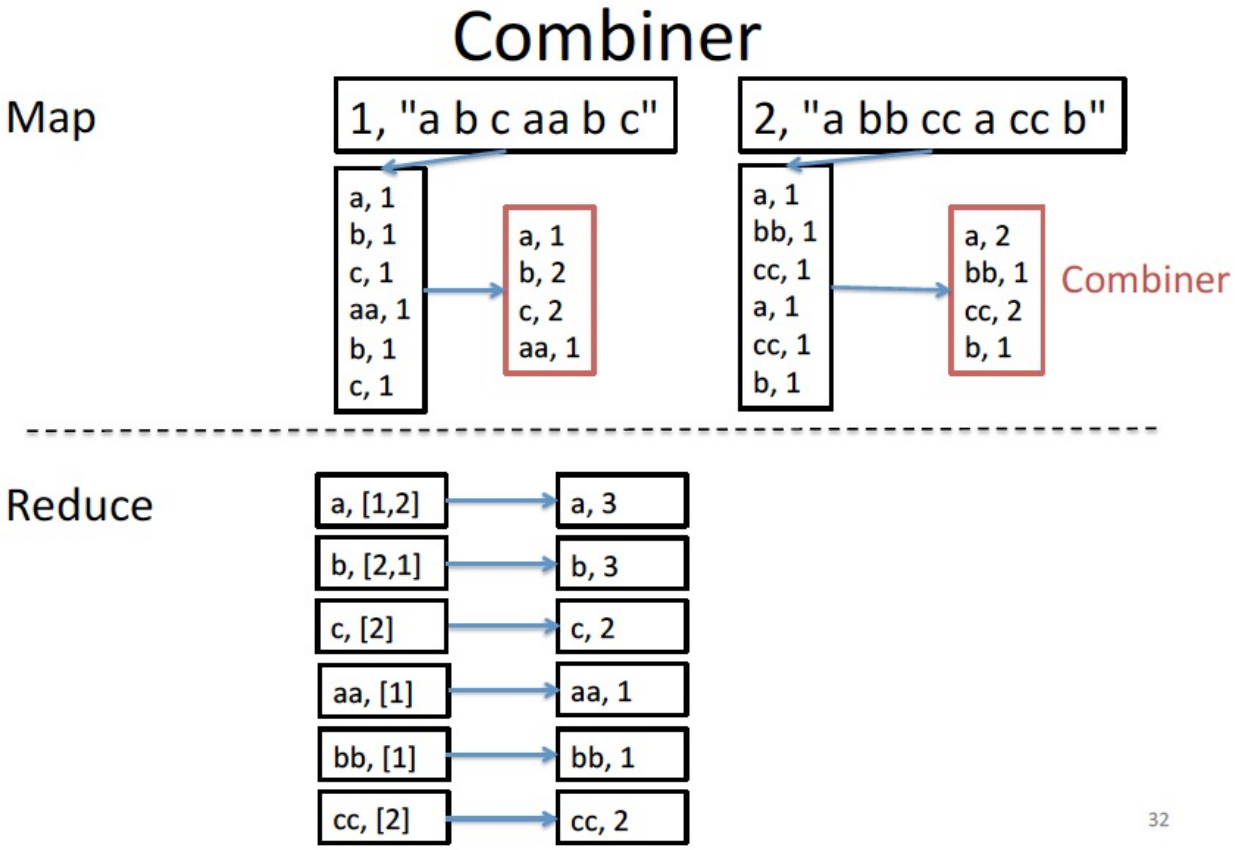
-D mapred.reduce.tasks=0 pour un job map-only

Le combiner

- Potentiel problème du mapper : de nombreuses paires (clé, valeur) à la sortie:
  - Envoyés au reducer à travers le réseau (cf shuffling)
  - Etape très couteuse en terme de temps d'exécution

- Ajout d'une opération : le combiner
  - Peut être vu comme un mini-reducer, qui travaille au niveau de chaque mapper, pour commencer l'agrégation.
  - C'est une étape Hadoop optionnelle : le résultat ne doit pas dépendre d'elle.

Le combiner




```
>> hadoop jar $STREAMINGJAR -files wc_mapper.py,wc_reducer.py -mapper
wc_mapper.py -combiner wc_reducer.py -reducer wc_reducer.py -input
livres/dracula -output sortie
```

Si la fonction « reduce » est à la fois commutative et associative, alors on peut utiliser le reducer comme combiner !

GÉNÉRATEUR ET ITÉRATEUR EN PYTHON

Map-Reduce improved!

Iterators en python

Un **itérateur** est une sorte de curseur qui a pour mission de se déplacer dans une séquence d'objets. L'itérateur permet de parcourir chaque objet d'une séquence sans se préoccuper de la structure sous-jacente.

Une liste et une liste en compréhension sont des itérateurs

```
# liste : iterator
liste=[1,2,3,4,5,6,7,8,9,10]
for x in liste:
        print(x)


# liste en comprehension : iterator
a_list=[1,9,8,4]
A=[elem*2 for elem in a_list]
print(A)
```

Iterators en python

```
Class Fib:
        def __init__(self, max):
                self.max=max

        def __iter__(self):
                self.a=0
                self.b=1
                return self

        def __next__(self):
                fib=self.a
                if fib>self.max:
                        raise StopIteration
                self.a, self.b = self.b, self.a+self.b
                return fib

if __name__=="__main__":

        fib=Fib(100)
        for n in fib:
                print(n, end=' ')
        print(list(Fib(200)))
```

Generators en python

Un **générateur** permet de simplifier la création d'itérateurs.

Le mot clé **yield** est un peu similaire au **return** des fonctions sauf qu'il ne signifie pas la fin de l'exécution de la fonction mais une mise en pause et à la prochaine itération la fonction recherchera le prochain **yield**.

```
def generateur1():
        yield "a"
        yield "b"
        yield "c"

g1=generateur1()
for v in g1:
        print(v)
```

```
def generateur2(n):
        for i in range(n):
                if i==5:
                        print("Ceci est le 5eme tour")
                yield i+1

g2=generateur2(10)
for v in g2:
        print(v)
```

WordCount amélioré avec itérateurs et générateurs

```
import sys

def read_input(file):
    for line in file:
        # split the line into words
        yield line.split()

def main(separator='\t'):
    # input comes from STDIN (standard input)
    data=read_input(sys.stdin)
    for words in data:
        # tab-delimited; the trivial word count is 1
        for word in words:
            print(word, separator, '1')

if __name__=="__main__":
    main()
```

Amélioration avec itérateurs et générateurs

```
# fichier wc_mapper_improved
from itertools import groupby
from operator import itemgetter
import sys

def read_mapper_output(file, separator='\t'):
    for line in file:
        yield line.rstrip().split(separator,1)

def main(separator='\t'):
    data=read_mapper_output(sys.stdin, separator=separator)

    # groupby groups multiple word-count pairs by word,
    # and creates an iterator that returns consecutive keys and their group:
    # current_word - string containing a word (the key)
    # group - iterator yielding all ["current_word", "count"] items
    for current_word, group in groupby(data, itemgetter(0)):
        try:
            total_count=sum(int(count) for current_word, count in group)
            print(current_word, separator, total_count)
        except ValueError:
            # count was not a number, so silently discard this item
            pass

if __name__=="__main__":
    main()
```

CHAINAGE MAP-REDUCE

Librairies mrjob

Chainage de mapper et reducer

Exemple : Compter le nombre de lettres qui apparaissent dans un texte, plutôt que le nombre de mots, peut être accompli avec 2 *mappers* :

- un premier mapper qui génère des paires (mots, 1) à partir des lignes du fichier
- un second mapper qui génère des paires (lettre, 1) à partir des paires (mots, 1) du premier mapper.

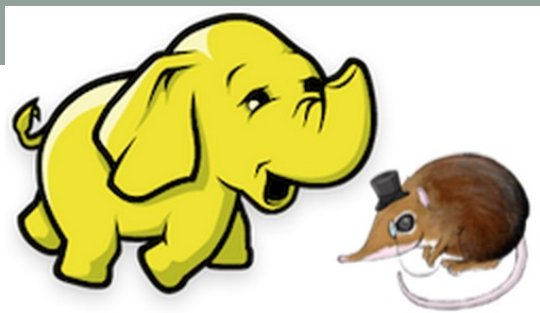
(ce n'est que pour l'exemple, on pourrait tout faire avec un seul mapper)

Remarque : On pourrait sans doute optimiser l'algorithme en lançant le second mapper après un combiner sur le premier mapper.

Très compliqué en natif -> utilisation de librairies python qui simplifient le développement d'algorithmes chainés:

- **dumbo** : https://github.com/klbostee/dumbo/.
  Pas mis à jour depuis 5 ans!
- **mrjob** : https://pythonhosted.org/mrjob/
- **pydoop** : https://crs4.github.io/pydoop/
  Uniquement sur Python 2.x
- **luigi** : https://github.com/spotify/luigi
  Aide pour construire des pipelines complexes de traitement par lots

MR Job

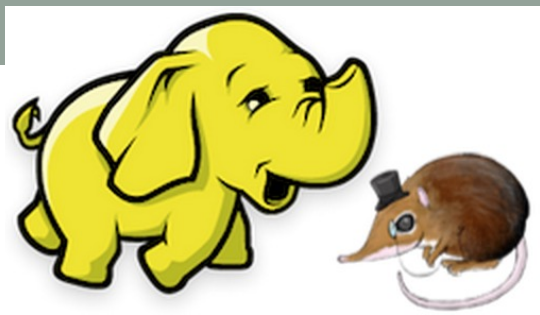


- Ecrire des jobs MapReduce en Python
- Librairie open-source (pip install), maintenu par Yelp
- Enveloppe du « Hadoop streaming » en cpython 2.5+
- Peut s'exécuter localement, ou sur hadoop (et sur Amazon EMR)

```
>> python3 wc1_mrjob.py dracula > resultLocal.txt
>> python3 wc1_mrjob.py dracula -r local --mapper
>> python3 wc1_mrjob.py dracula -r hadoop > resultHadoop.txt
```

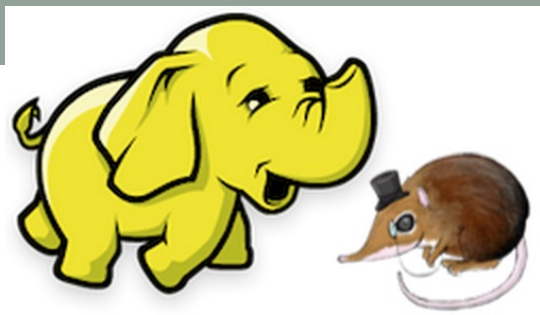
```
# fichier wc1_mrjob
from mrjob.job import MRJob
import re
WORD_RE=re.compile(r"[\w]+")

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()
```

MR Job



```
# fichier wc2_mrjob
from mrjob.job import MRJob

class MRWordCountUtility(MRJob):
    def __init__(self,*args, **kwargs):
        super(MRWordCountUtility,self).__init__(*args, **kwargs)
        self.chars=0
        self.words=0
        self.lines=0

    def mapper(self, _, line):
        # Don't actually yield anything for each line. Instead, collect them
        # and yield the sums when all lines have been processed. The results
        # will be collected by the reducer.
        self.chars+=len(line)+1 # +1 for newline
        self.words+=sum(1 for word in line.split() if word.strip())
        self.lines+=1

    def mapper_final(self):
        yield('chars', self.chars)
        yield('words', self.words)
        yield('lines', self.lines)

    def reducer(self, key, values):
        yield(key, sum(values))

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

MR Job



```
# fichier muw_mrjob
from mrjob.job import MRJob
from mrjob.protocol import JSONValueProtocol
from mrjob.step import MRStep
import re
WORD_RE=re.compile(r"[\w']+")
class MRMostUsedWord(MRJob):
    OUTPUT_PROTOCOL=JSONValueProtocol
    def mapper_get_words(self, _, line):
        # yield each word in the line
    def combiner_count_words(self, word, counts):
        # sum the words we've seen so far
    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
    def steps(self):
        return[
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]
if__name__=='__main__':
    MRMostUsedWord.run()
```

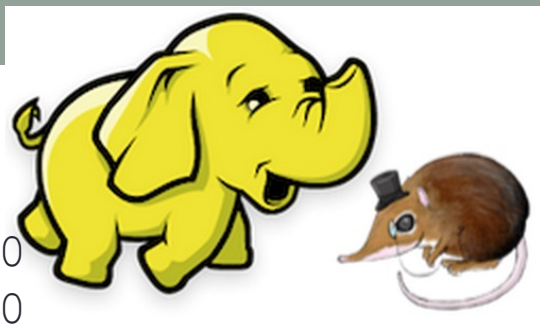
Tous les détails : https://media.readthedocs.org/pdf/mrjob/latest/mrjob.pdf

SYSTÈME DE RECOMMANDATION VIDEO

Exemple largement inspiré de

http://aimotion.blogspot.com/2012/08/introduction-to-recommendations-with.html

Historique de données



```
Jack Matthews|Lady in the Water|3.000000
Jack Matthews|Snakes on a Plane|4.000000
Jack Matthews|The Night Listener|3.000000
Mick LaSalle|Lady in the Water|3.000000
Mick LaSalle|Snakes on a Plane|4.000000
Mick LaSalle|You, Me and Dupree|2.000000
Mick LaSalle|The Night Listener|3.000000
Claudia Puig|Snakes on a Plane|3.500000
Claudia Puig|You, Me and Dupree|2.500000
Claudia Puig|Superman Returns|4.000000
Claudia Puig|The Night Listener|4.500000
Toby|Snakes on a Plane|4.500000
Toby|Superman Returns|4.000000
Toby|You, Me and Dupree|1.000000
Gene Seymour|Lady in the Water|3.000000
Gene Seymour|Snakes on a Plane|3.500000
```

Objectif : proposer une recommandation sur l'analyse de l'historique de notations des films par les clients. Plutôt que les noms des clients et des films, nous afficherons des identifiants (imaginaires).

Historique de données

Notre objectif est de calculer jusqu'à quel point 2 films sont similaires, et donc de proposer un film aux clients par analyse de l'historique des autres clients.

Pour cela, 3 étapes (c'est à dire 3 jobs consécutifs):

1. Pour chaque pair de films A et B, trouver tous les clients qui ont noté A et B.
2. Construire un vecteur de notes pour le film A et un vecteur de notes pour le film B, et calculer la corrélation entre ces deux vecteurs.
3. Pour chaque film, trier les films les plus similaires.

Lorsqu'un client regarde un film, vous pouvez lui conseiller de regarder les films qui lui sont le plus corrélés.

Les 3 jobs consécutifs

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from mrjob.protocol import JSONValueProtocol
…

class MoviesSimilarities(MRJob):
    OUTPUT_PROTOCOL = JSONValueProtocol
    def steps(self):
        return [
            MRStep(mapper=self.group_by_user_rating,
                   reducer=self.count_ratings_users_freq),
            MRStep(mapper=self.pairwise_items,
                   reducer=self.calculate_similarity),
            MRStep(mapper=self.calculate_ranking,
                   reducer=self.top_similar_items)
        ]
    …

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

Job 1. Trouver tous les clients qui ont noté A et B

Le mapper émet « *user_id, (item_id, rating)* »

```
def group_by_user_rating(self, key, line):
    """
    Emit the user_id and group by their ratings
    (item and rating)
    17   70,3
    35   21,1
    49   19,2
    49   21,1
    49   70,4
    87   19,1
    87   21,2
    98   19,2
    """
    user_id, item_id, rating = line.split('|')

    yield  user_id, (item_id, float(rating))
```

Job 1. Trouver tous les clients qui ont noté A et B

Le *reducer* émet « *user_id, (item_count, item_sum, final)* »

```
def count_ratings_users_freq(self, user_id, values):
        """
        For each user, also emit user rating sum and count for
           use later steps.
        17    1,3,(70,3)
        35    1,1,(21,1)
        49    3,7,(19,2 21,1 70,4)
        87    2,3,(19,1 21,2)
        98    1,2,(19,2)
        """
        item_count = 0
        item_sum = 0
        final = []
        for item_id, rating in values:
            item_count += 1
            item_sum += rating
            final.append((item_id, rating))

        yield user_id, (item_count, item_sum, final)
```

Job 2. Construire des vecteurs de notes pour le films

Le *mapper* émet « *(item1[0], item2[0]), (item1[1], item2[1])* »

```
def pairwise_items(self, user_id, values):
    '''
    The output drops the user from the key entirely, instead
        it emits the pair of items as the key:
    19,21  2,1
    19,70  2,4
    21,70  1,4
    19,21  1,2
     '''
    item_count, item_sum, ratings = values

    #bottleneck at combinations
    for item1, item2 in combinations(ratings, 2):
        yield (item1[0], item2[0]), (item1[1], item2[1])
```

Job 2. Construire des vecteurs de notes pour les films

Le *reducer* émet « *(item_xname, item_yname), (similarity, n)* »

```
def calculate_similarity(self, pair_key, lines):
    '''
    Sum components of each corating pair across all users
      who rated both item x and item y, then calculate
      pairwise pearson similarity and corating counts.
    19,21   0.4,2
    21,19   0.4,2
    19,70   0.6,1
    70,19   0.6,1
    21,70   0.1,1
    70,21   0.1,1
    '''
```

Job 2. Construire des vecteurs de notes pour les films

```
def calculate_similarity(self, pair_key, lines):

    sum_xx, sum_xy, sum_yy, sum_x, sum_y, n = \
            (0.0, 0.0, 0.0, 0.0, 0.0, 0)
    item_pair, co_ratings = pair_key, lines
    item_xname, item_yname = item_pair
    for item_x, item_y in lines:
        sum_xx += item_x * item_x
        sum_yy += item_y * item_y
        sum_xy += item_x * item_y
        sum_y += item_y
        sum_x += item_x
        n += 1

     corr_sim = correlation(n, sum_xy, sum_x, \
                                    sum_y, sum_xx, sum_yy)

    yield (item_xname, item_yname), (corr_sim, n)
```

Job 3. Pour chaque film, trier les films les plus similaires

Le *mapper* émet « *(item_x, corr_sim), (item_y, n)* »

```
def calculate_ranking(self, item_keys, values):
    '''
    Emit items with similarity in key for ranking:
    19,0.4    70,1
    19,0.6    21,2
    21,0.6    19,2
    21,0.9    70,1
    70,0.4    19,1
    70,0.9    21,1
    '''
    corr_sim, n = values
    item_x, item_y = item_keys
    if int(n) > 0:
        yield (item_x, corr_sim), (item_y, n)
```

Job 3. Pour chaque film, trier les films les plus similaires

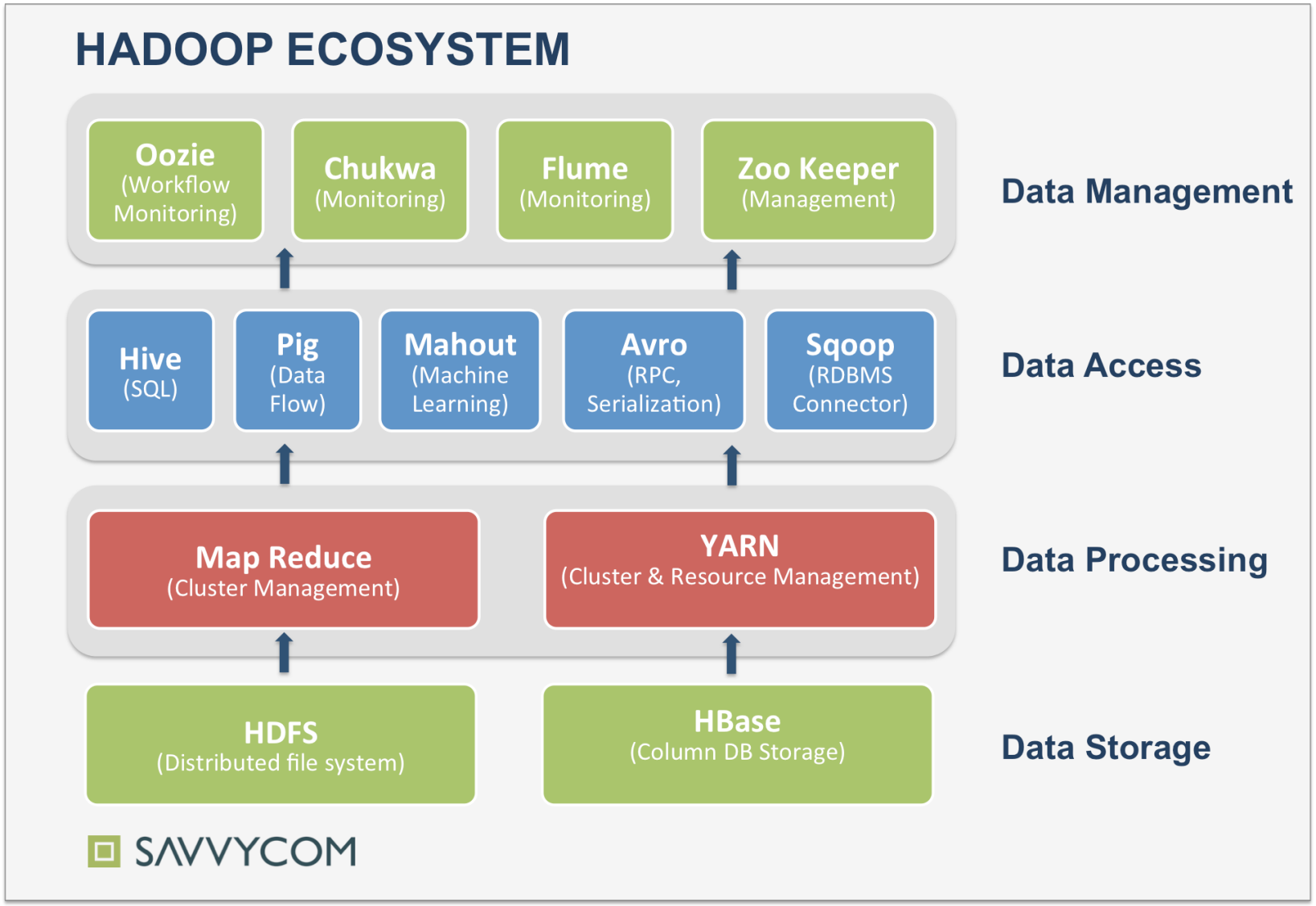
Le reducer émet « *None, (item_x, item_y, corr_sim, n)* »

```
def top_similar_items(self, key_sim, similar_ns):
    '''
    For each item emit closest items:
      [De La Soul, A Tribe Called Quest, 0.6, 1]
      [De La Soul, 2Pac, 0.4, 2]
    '''
    item_x, corr_sim = key_sim
    for item_y, n in similar_ns:
        yield None, (item_x, item_y, corr_sim, n)
```

ECOSYSTÈME HADOOP

Qui utilise Hadoop?

Ecosystème Hadoop



Description complète : https://hadoopecosystemtable.github.io

Quelques outils 1/3

HIVE et **PIG** sont deux outils qui permettent d'interagir avec les données contenues dans HDFS en exécutant des jobs MapReduce de façon transparente.

Hive est un outil développé par Facebook qui permet d'utiliser HADOOP par le biais de requêtes HiveQL. Ce langage est proche du SQL. Il permet de construire un modèle de données relationnel basé sur les données contenues dans HDFS.
Ce modèle de données est stocké dans un *métastore* qui gère la définition des tables ainsi que les métadonnées.

Pig est un outil développé par Yahoo qui permet d'exécuter des jobs mapreduce en utilisant un langage de scripting (PigLatin). Les scripts PigLatin permettent de travailler sur des données contenues dans HDFS. Ces données sont traitées par MapReduce qui retourne le résultat de ses calculs à Pig.

Quelques outils 2/3

Flume: Il s'agit d'un outil permettant d'injecter de gros volumes de données en temps réel dans HDFS. Cet outil est capable de « streamer » des données depuis n'importe quelle source pour les ajouter dans HADOOP. Flume est extensible : il intègre des données en provenance de sources variées.

HBASE: C'est une base de données non relationnelle distribuée. Elle a la particularité d'utiliser les données directement dans HDFS sans passer par des tâches. MapReduce. HBASE présente donc des caractéristiques assez similaires à celles de HDFS (capacité à gérer des volumes de données de plusieurs Po, forte tolérance de panne…). HBASE est bien adaptée pour gérer des données parsemées comme par exemple une table de plusieurs milliers de colonnes avec une majorité de cellules vides.

APACHE HBASE

Quelques outils 3/3

Oozie est un outil d'ordonnancement de jobs HADOOP. En effet, de nombreux traitements ne peuvent être réalisés par un seul job. Il faut alors créer et gérer une chaîne de traitement. C'est ce que permet de faire Oozie. Il est également capable de gérer des dépendances temporelles, d'exécuter une tâche plusieurs fois de suite ou encore de faire remonter d'éventuelles erreurs.

HUE est un outil permettant d'obtenir une interface graphique de HADOOP. Cette interface graphique comprend un navigateur de fichiers permettant d'accéder à HDFS, un navigateur de job MapReduce, un navigateur HBASE ainsi qu'un éditeur de requête pour Hive et Pig.