Réalisez des calculs distribués sur des données massives

20 heures Moyenne

Mis à jour le 08/04/2020

Divisez (et distribuez) pour régner

Dans cette partie, nous allons étudier MapReduce, un modèle de programmation qui fournit un cadre pour **automatiser le calcul parallèle sur des données massives**. Pour la petite histoire, ce modèle a

été proposé dans les années 2000, par deux ingénieurs de chez Google, qui ont observé qu'un grand nombre des traitements massivement parallèles, mis en place pour les besoins de leur moteur de recherche, suivaient une stratégie de parallélisation identique. De ces observations est né le modèle de programmation MapReduce, décrit pour la première fois en 2004 dans un article de recherche. Son principe général est que toute parallélisation de traitement sur des données massives peut s'effectuer uniquement à l'aide de deux types d'opérations : une opération `map` et une opération `reduce` .

Diviser pour régner

Quittons pour un petit moment le monde du calcul distribué et des données massives et rappelons nous les *bons vieux* paradigmes de conception d'algorithmes et notamment le très célèbre **Divisez pour régner** et ses trois étapes consistant, pour un problème initial donné, à:

1. **Diviser** : découper le problème initial en sous-problèmes;
2. **Régner** : résoudre les sous-problèmes indépendamment soit de manière récursive, soit directement s'ils sont de petite taille;
3. **Combiner** : construire la solution du problème initial en combinant les solutions des différents sous-problémes.

MapReduce c'est **Divisez pour distribuer pour régner** en ce sens que la stratégie mise en place pour exécuter un calcul sur des données massives consiste à découper les données en sous-ensembles de plus petite taille, que nous appelerons des **lots** ou des **fragments** dans la suite, et à affecter chaque lot à une machine du cluster permettant ainsi leur traitement en parallèle. Il suffira ensuite d'agréger l'ensemble des résultats intermédiaires obtenus pour chaque lot pour construire le résultat final.

Jusque là, c'est plutôt simple et plein de bon sens. Mais pourquoi parle t'on alors de MapReduce plutôt que de Divide, Distribute and Conquer?

Inspiration fonctionnelle

Pour répondre à cela, faisons maintenant un petit détour du côté des paradigmes de programmation ! Si vous êtes adepte de la **programmation fonctionnelle**, qui donne un rôle central aux fonctions, alors vous avez déjà très certainement compris pourquoi "Map" et "Reduce". Si ce n'est pas le cas, sachez que MapReduce s'inspire très largement de ce paradigme de programmation et plus particulièrement des opérateurs de listes `map` et `reduce` . En programmation fonctionnelle,

- **`map`** consiste à appliquer une même fonction à tous les éléments de la liste;

text

```
map(f)[x0, ..., xn] = [f(x0, ...,f(xn)]
map(*4)[2, 3, 6] = [8, 12, 24]
```

- **`reduce`** applique une fonction récursivement à une liste et retourne un seul résultat;

text

```
reduce(f)[x0, ..., xn] = f(x0, f(x1, f(x2, ...)))
reduce(+)[2, 3, 6] = (2 + (3 + 6)) = 11
```

`map` et `reduce` sont des opérateurs génériques et leur combinaison permet donc de modéliser énormément de problèmes.

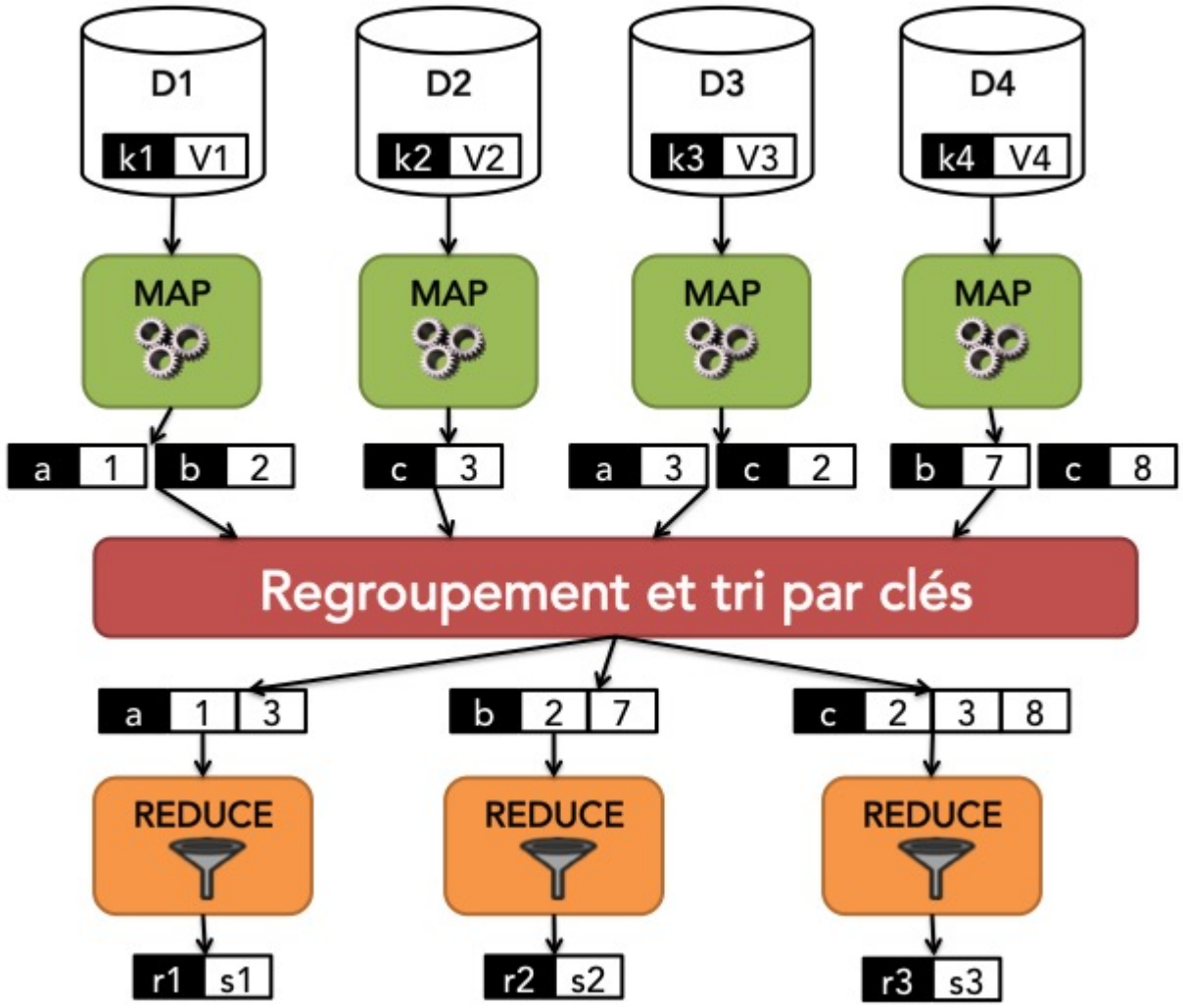
Nous avons maintenant presque tous les ingrédients pour expliquer le modèle de programmation MapReduce.

MapReduce sur des paires `(clé, valeur)`

Le dernier ingrédient, fondamental pour l'automatisation de la parallélisation, est que **l'ensemble des données est représentée sous la forme de paires** `(clé, valeur)` , à l'instar des tables d'association.

Nous avons maintenant tous les ingrédients pour expliquer le fonctionnement général de MapReduce.

1. L'ensemble des données à traiter est découpé en plusieurs lots ou sous-ensembles.
2. Dans une première étape, l'étape MAP, l'opération `map` , spécifiée pour notre problème, est appliquée à chaque lot. Cette opération transforme la paire `(clé, valeur)` représentant le lot en une liste de nouvelles paires `(clé, valeur)` constituant ainsi des résultats intermédiaires du traitement à effectuer sur les données complètes.
3. Avant d'être envoyés à l'étape REDUCE, les résultats intermédiaires sont regroupés et triés par clé. C'est l'étape de `SHUFFLE and SORT` .
4. Enfin, l'étape REDUCE consiste à appliquer l'opération `reduce` , spécifiée pour notre problème, à chaque clé. Elle agrège tous les résultats intermédiaires associés à une même clé et renvoie donc pour chaque clé une valeur unique.



Wordcount, le "Hello World!" de MapReduce !

Pour rendre le fonctionnement de MapReduce plus concret, nous allons l'illustrer avec "WordCount!" l'exemple typique de MapReduce, si typique qu'il en est devenu le "Hello World!" de MapReduce et du calcul distribué.

Rien de très compliqué. Prenons en entrée une collection de documents textuels : l'objectif de Wordcount est de calculer le nombre d'occurrences de chaque mot dans la collection.

Supposons que vous pouvez stocker l'ensemble de votre collection dans un seul fichier ; alors, ce n'est vraiment pas un problème difficile et les quelques lignes de code ci-dessous peuvent répondre à ce problème.

python

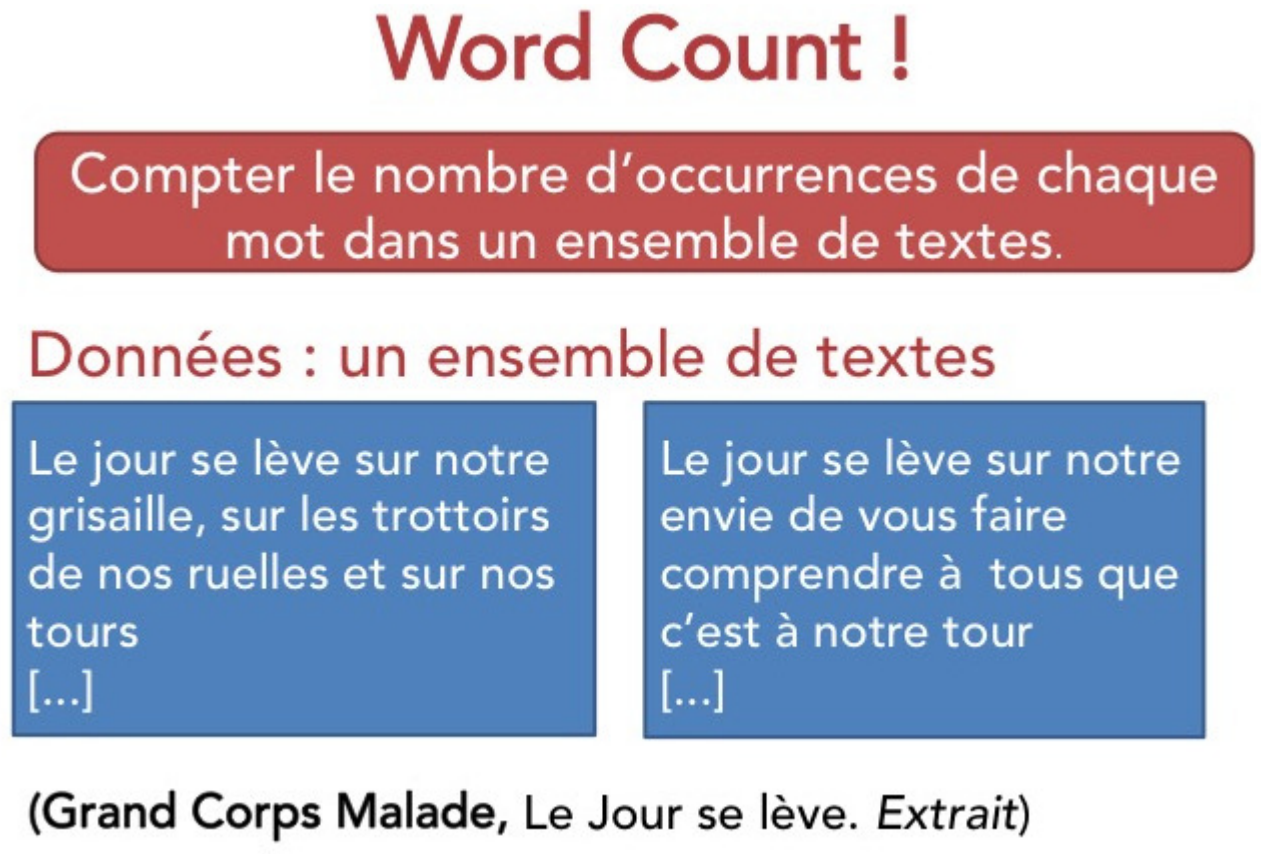
```python
from collections import defaultdict

def wordCount(text):
    counts=defaultdict(int)
    for word in text.split():
        counts[word.lower()] +=1
```

```
7     return counts
```

Notez que l'on utilise ici un dictionnaire qui est une collection d'éléments de type `(clé, valeur)` . La clé correspond au mot et la valeur un entier correspondant à son nombre d'occurrences.

Bien evidemment, si votre texte est grand, voire très, très grand, à l'image de la collection Wikipedia qui contient environ 27 milliards de mots (source : Wikipedia), cette solution séquentielle ne suffira pas et il est nécessaire de réaliser ce comptage de manière distribuée. C'est là qu'intervient MapReduce.

Nous allons travailler sur deux extraits du texte de la chanson "Le Jour se lève" de Grand Corps Malade. Nous sommes bien loin des 27 milliards de mots de Wikipédia mais l'objectif est d'illustrer le principe de MapReduce.



Nous allons donc supposer que nos données d'entrée ont été découpées en différents fragments et qu'une opération de simplification a été appliquée sur chaque fragment pour supprimer les caractères de ponctuation, transformer chaque mot en son singulier ("nos" devient "notre") et ne garder que les mots de plus de 3 caractères. Nous pouvons représenter très facilement ces fragments sous la forme de paires `(clé, valeur)` , en prenant comme clé le nom du fichier et comme valeur la chaîne de caractères correspondant au contenu textuel du fichier.

python

```python
D1 = {"./lot1.txt" : "jour lève notre grisaille"}
D2 = {"./lot2.txt" : "trottoir notre ruelle notre tour"}
D3 = {"./lot3.txt" : "jour lève notre envie vous"}
D4 = {"./lot4.txt" : "faire comprendre tous notre tour"}
```

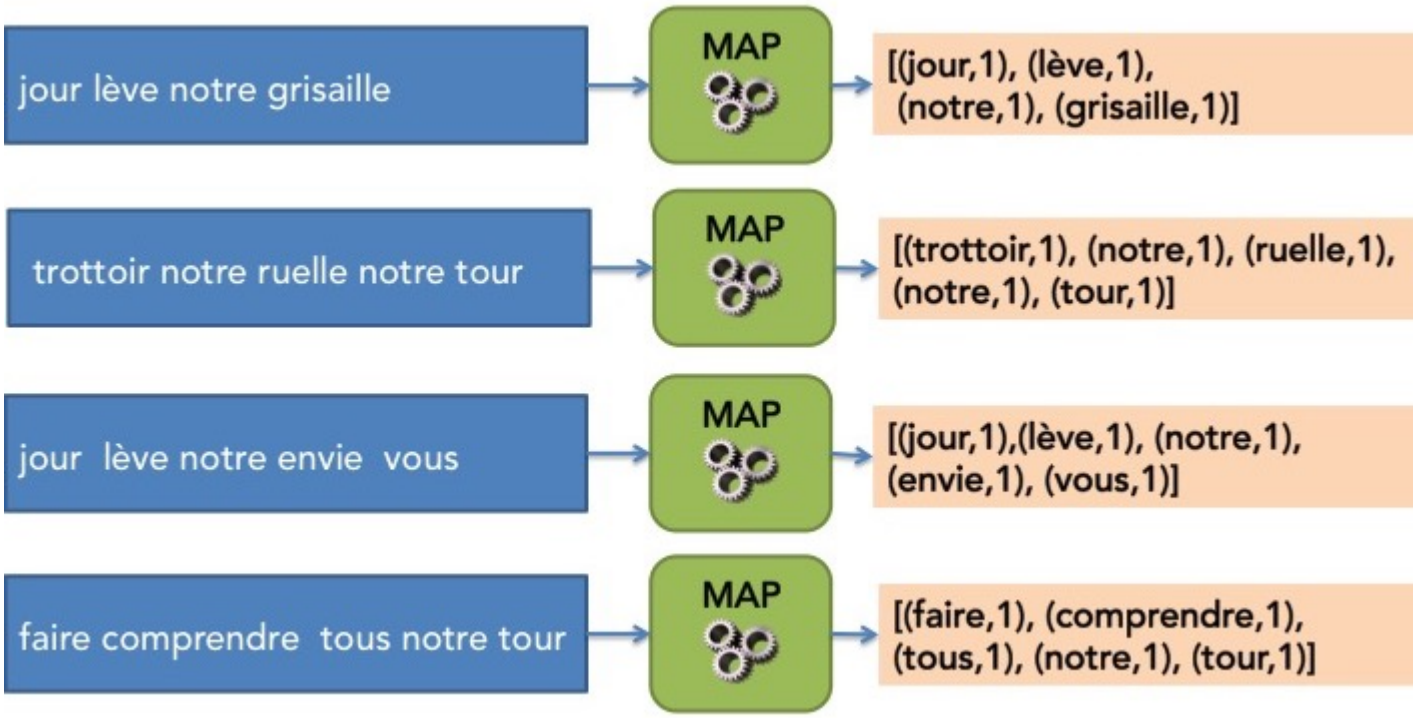
Il nous faut maintenant déterminer la clé à utiliser pour l'opération `map` . La manière dont nous avons répondu au problème en séquentiel nous oriente tout naturellement vers le choix de prendre comme clés les mots du texte.

L'étape suivante est d'écrire le code de l'opération `map` selon le schéma imposé par MapReduce, c'est-à-dire qu'elle doit retourner une liste de paires `(clé, valeur)` . Dans le cas de WordCount, l'opération `map` va donc décomposer le texte du fragment fourni en entrée et elle va générer pour chaque mot une paire `(mot, 1)` . Nous pouvons écrire tout cela très simplement en python.

python

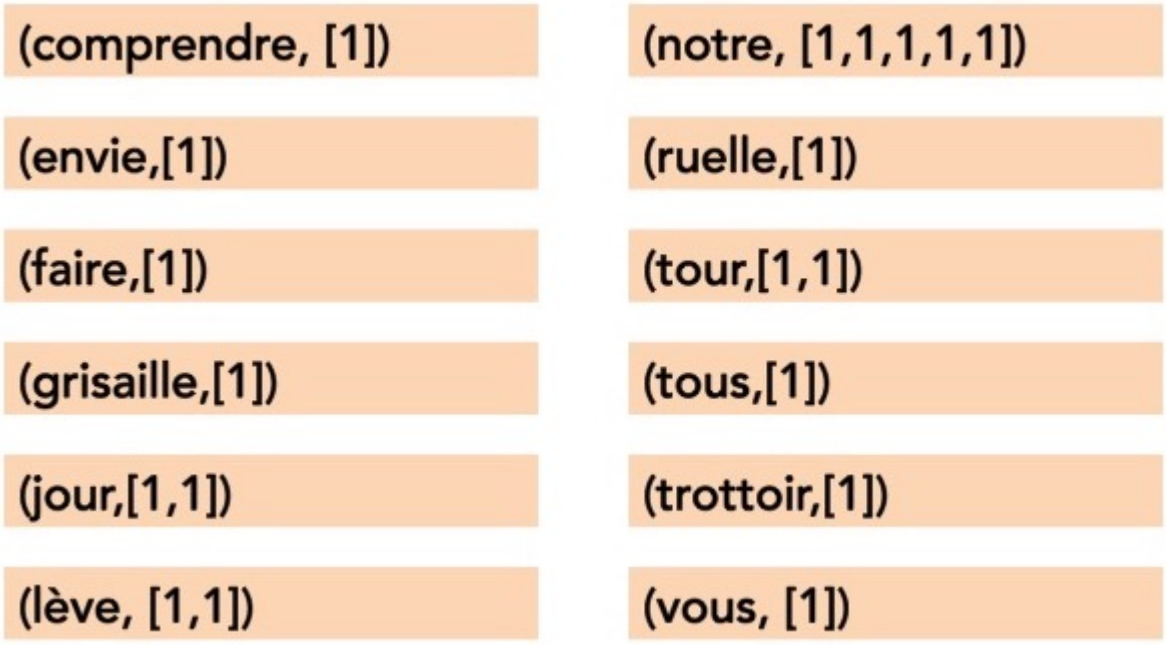
```python
def map(key,value):
    intermediate=[]
    for word in value.split():
        intermediate.append((word, 1))
    return intermediate
```

Nous avons donc maintenant tout ce qu'il faut pour l'étape MAP de MapReduce qui consiste à appliquer l'opération `map` à chaque fragment en parallèle comme l'illustre la figure ci-dessous.



A la fin de l'étape MAP, nous avons donc plusieurs listes de paires `(clé, valeur)` .

A ce stade du cours, nous allons considérer que nous sommes un peu magicien et que d'un petit coup de baguette nous sommes capables de regrouper et de trier, par clé commune, les résultats intermédiaires fournis par l'étape MAP. Cela correspond à l'étape `SHUFFLE and SORT` . Nous verrons plus tard que, dans la pratique, cette étape est entièrement gérée par le framework d'exécution de MapReduce, de manière distribuée, et qu'il est donc légitime de ne pas la détailler plus ici.



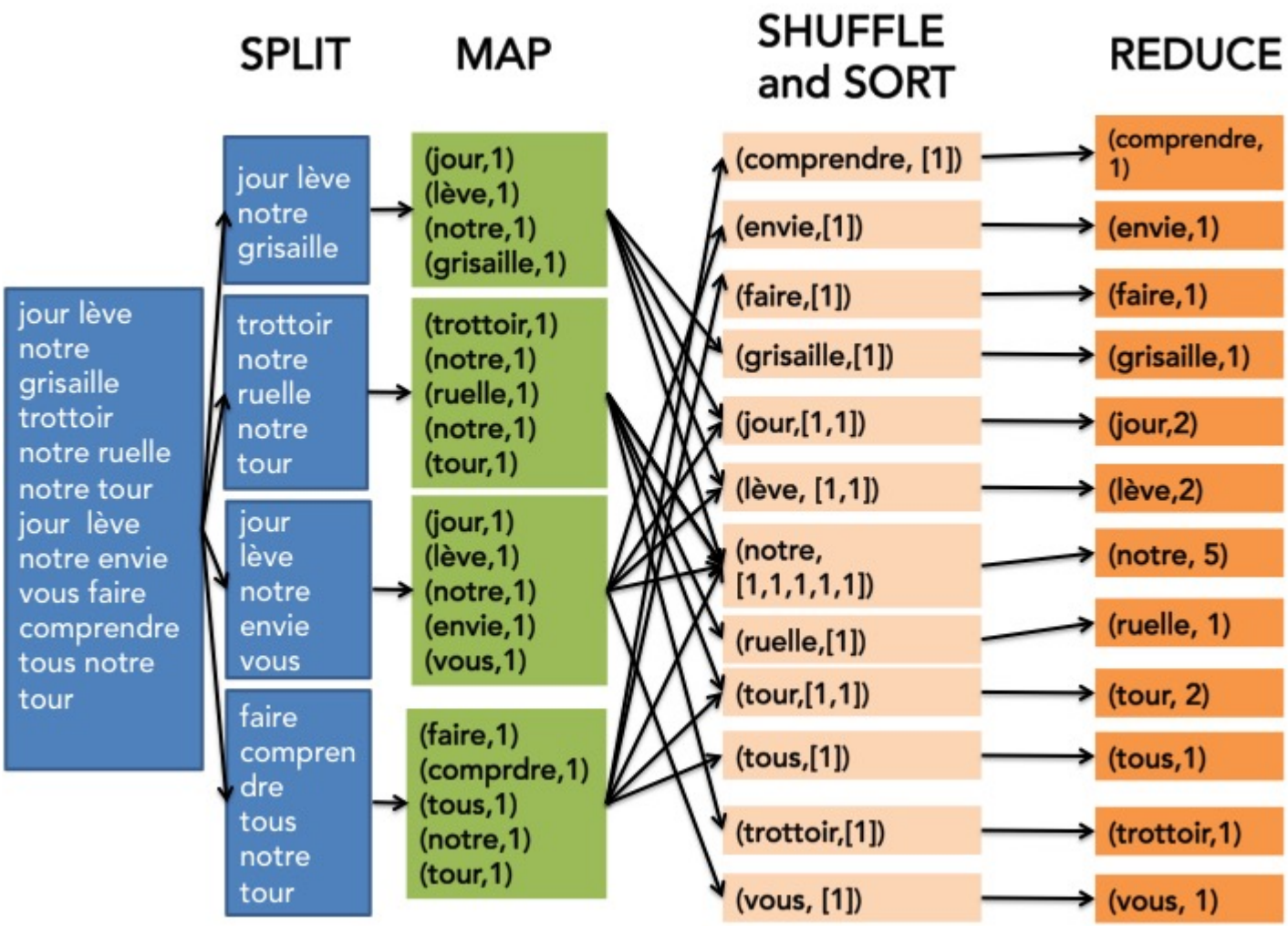
Nous avons donc maintenant à notre disposition un ensemble de paires `(clé, liste_de_valeurs)` . Il nous reste maintenant à écrire le code de l'opération `reduce` , selon le schéma imposé par MapReduce. Pour WordCount, l'opération `reduce` va donc juste consister à sommer toutes les valeurs de la liste associée à une clé. Nous pouvons à nouveau écrire cela très simplement en python.

python

```python
def reduce(key, values):
    result = 0
    for c in values:
        result = result +c
    return (key, result)
```

L'étape REDUCE de MapReduce peut donc être appliquée. Elle consiste à appliquer l'opération `reduce` à chaque paire `(clé, liste_de_valeurs)` en parallèle.

Le schéma ci-dessous illustre l'application des différentes étapes de MapReduce à notre exemple.



À ce stade du cours, vous devez avoir compris que l'approche MapReduce pour le calcul distribué consiste à reformuler son problème en fonctions parallélisables avec un schéma relativement contraint.

C'est très beau sur le papier tout cela mais quid de l'implémentation concrète dans un contexte Big Data, c'est-à-dire sur un cluster de machines ? Comment se fait l'ordonnancement et la distribution des calculs ? Comment MapReduce permet-il de répondre aux principales problèmatiques du calcul distribué abordées dans la partie précédence à savoir :

- l'accès et le partage de données ;
- la gestion des erreurs et la tolérance aux pannes ;
- la localisation des données ?

Ce sont de très bonnes questions et nous verrons dans un prochain chapitre que ces différentes questions sont prises en charge par un framework d'exécution distribuée de MapReduce. Nous présenterons notamment les grands principes de Hadoop MapReduce, l'implémentation libre de référence de MapReduce.

Il est bien évidemment nécessaire de comprendre et de maîtriser les rouages de ces frameworks d'exécution distribuée pour la mise en oeuvre d'algorithmes MapReduce, mais finalement la première grande difficulté reste la (re)formulation de votre problème en MapReduce. Pour vous entraîner à penser en MapReduce, je vous propose de l'illustrer à nouveaux sur deux exemples différents dans le chapitre suivant.

En résumé, MapReduce c'est :

- La généralisation du paradigme de conception d'algorithmes *diviser pour régner* au cadre distribué.
- Un modèle de programmation reposant sur la combinaison de deux fonctions simples, `map` et `reduce`, inspirées de la programmation fonctionnelle.
- Un framework d'exécution prenant en charge le deploiement et la distribution des calculs sur un cluster.
- Le rôle des developpeurs d'applications distribuées, c'est donc de penser en MapReduce :
    1. Choisir une manière de découper les données afin que l'opération MAP soit parallélisable.
    2. Choisir la clé à utiliser pour le problème ciblé.
    3. Écrire le code de la fonction pour l'opération MAP.
    4. Écrire le code de la fonction pour l'opération REDUCE.

QUIZ : À LA DÉCOUVERTE DES MÉGADONNÉES

PARCOUREZ LES PRINCIPAUX ALGORITHMES MAPREDUCE

Les professeurs

Céline Hudelot

Professeur des Universités en Informatique à CentraleSupélec.

Régis Behmo

Expert en machine learning, développeur fullstack, grimpeur invétéré et gros, très gros amateur de nouilles chinoises.