MapReduce

Dr. H. EL BOUHISSI BRAHAMI

M1 SIA

Hadoop -Rappel

Le projet Hadoop consiste en deux grandes parties:

- Stockage des données : HDFS (Hadoop Distributed File System)
- Traitement des données : MapReduce / Yarn

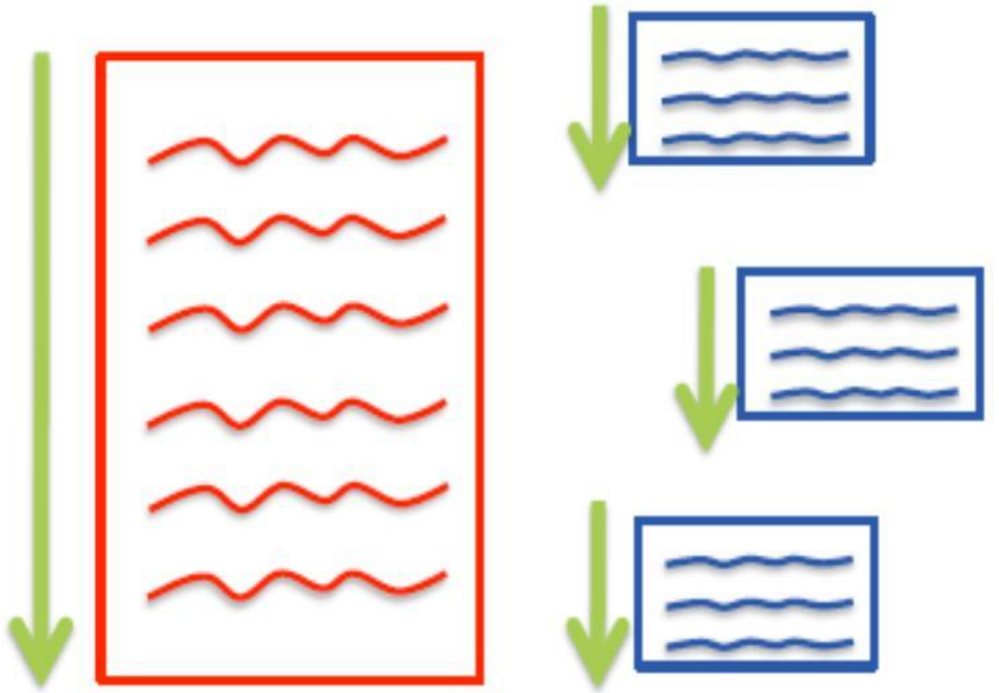
Principe :

- Diviser les données
- Les sauvegarder sur une collection de machines, appelées cluster
- Traiter les données directement là où elles sont stockées, plutôt que de les copier à partir d'un serveur distribué
- Il est possible d'ajouter des machines à votre cluster, au fur et à mesure que les données augmentent

Map-Reduce -

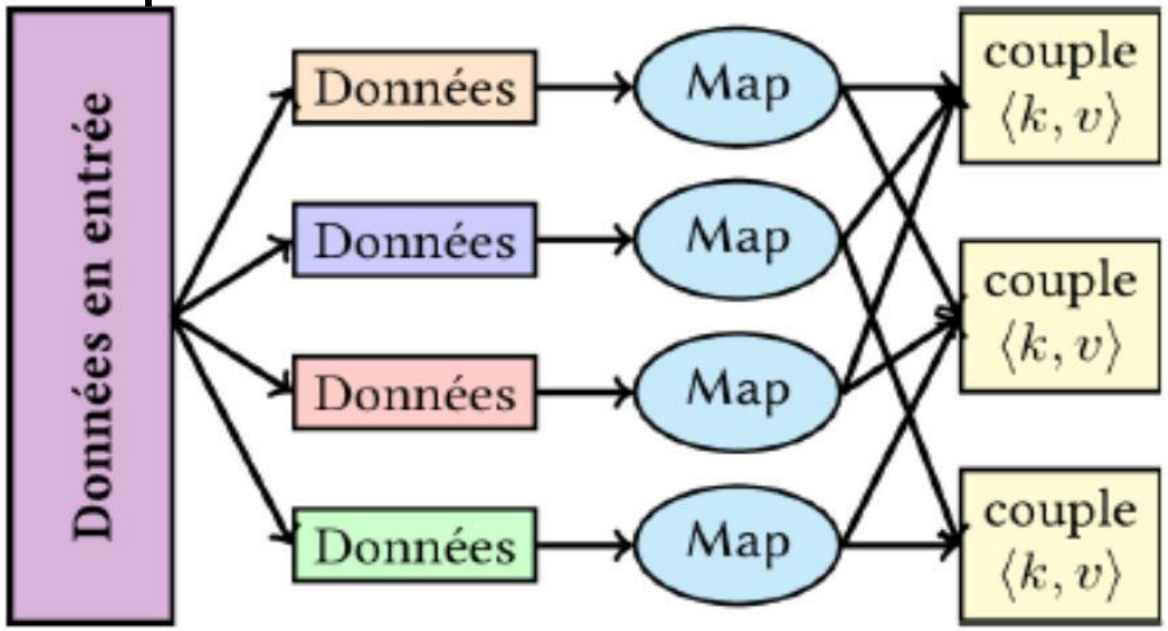
Patron d'architecture de développement permettant de traiter des données volumineuses de manière parallèle et distribuée.

- A la base, le langage Java est utilisé, mais grâce à une caractéristique de Hadoop appelée Hadoop Steaming, il est possible d'utiliser d'autres langages comme Python ou Ruby.
- Au lieu de parcourir le fichier séquentiellement (bcp de temps), il est divisé en morceaux qui sont parcourus en parallèle.



Opération Map

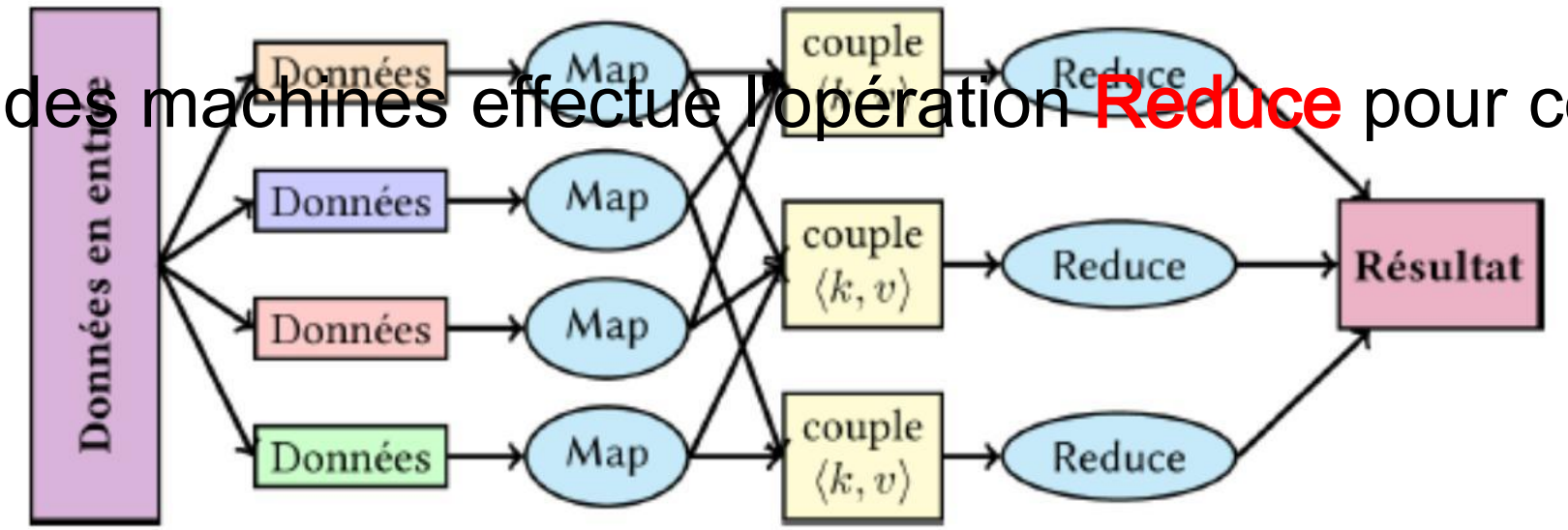
Transforme les données d'entrée en série de couples clé/valeur. Les couples
clés/valeurs doivent avoir un sens par rapport au problème à résoudre.
Les données doivent être découpées en plusieurs fragments et l'opération **Map**
est exécutée sur chaque machine du cluster sur un fragment distinct.



Opération Reduce

Applique un traitement à toutes les valeurs de chacune des clés distinctes
produites par l'opération Map.

- Pour chaque machine du cluster est attribué une des clés uniques produites
  par l'opération Map, en lui donnant la liste des valeurs associées à cette clé.
- Chacune des machines effectue l'opération **Reduce** pour cette clé.



Map-Reduce -

- Choisir une manière de découper les données d'entrée pour pouvoir paralléliser l'opération Map.

- Définir quelle clé utiliser pour le problème à résoudre.

- Écrire le programme pour l'opération Map.

- Écrire le programme pour l'opération Reduce.

Map-Reduce - Étapes de traitements

Un traitement MapReduce est caractérisé par 4 étapes distinctes :

Split : Découper les données d'entrée en plusieurs fragments.

Map : Mapper chacun de ces fragments pour obtenir des couples (clé/valeur).

Shuffle : Grouper ces couples par clé.

Reduce : Réduire les groupes indexés par clé en une forme finale, avec une
valeur pour chacune des clés distinctes.

Chacune des tâches (à l'exception de la première) seront effectuées de manière distribuée.

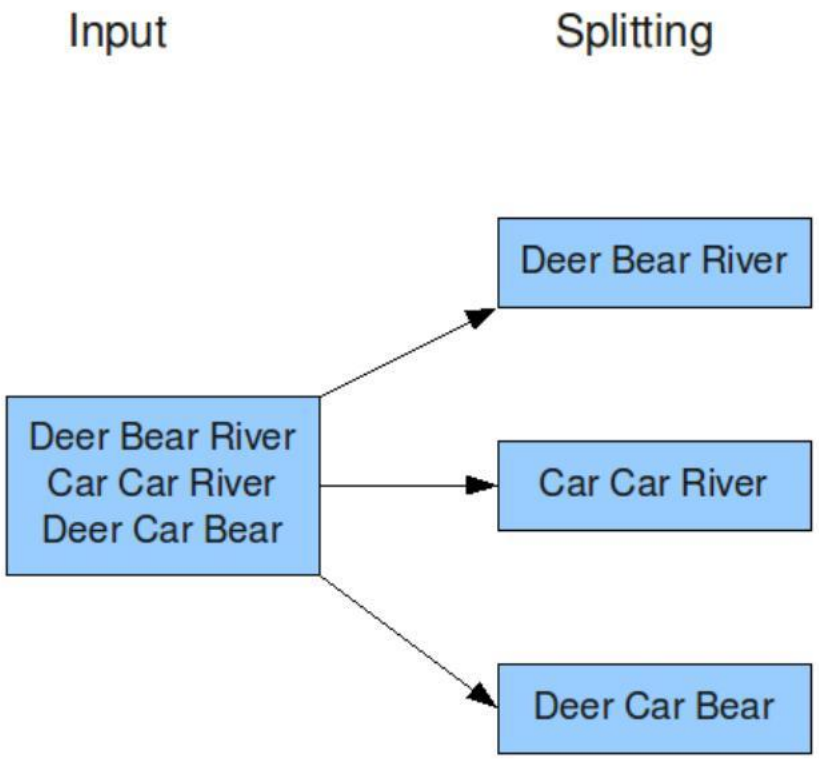
Le calcul distribué, groupement par clé distincte... se font par Hadoop de manière transparente.

Exemple de WordCount

Compter le nombre d'occurrence des mots dans un document

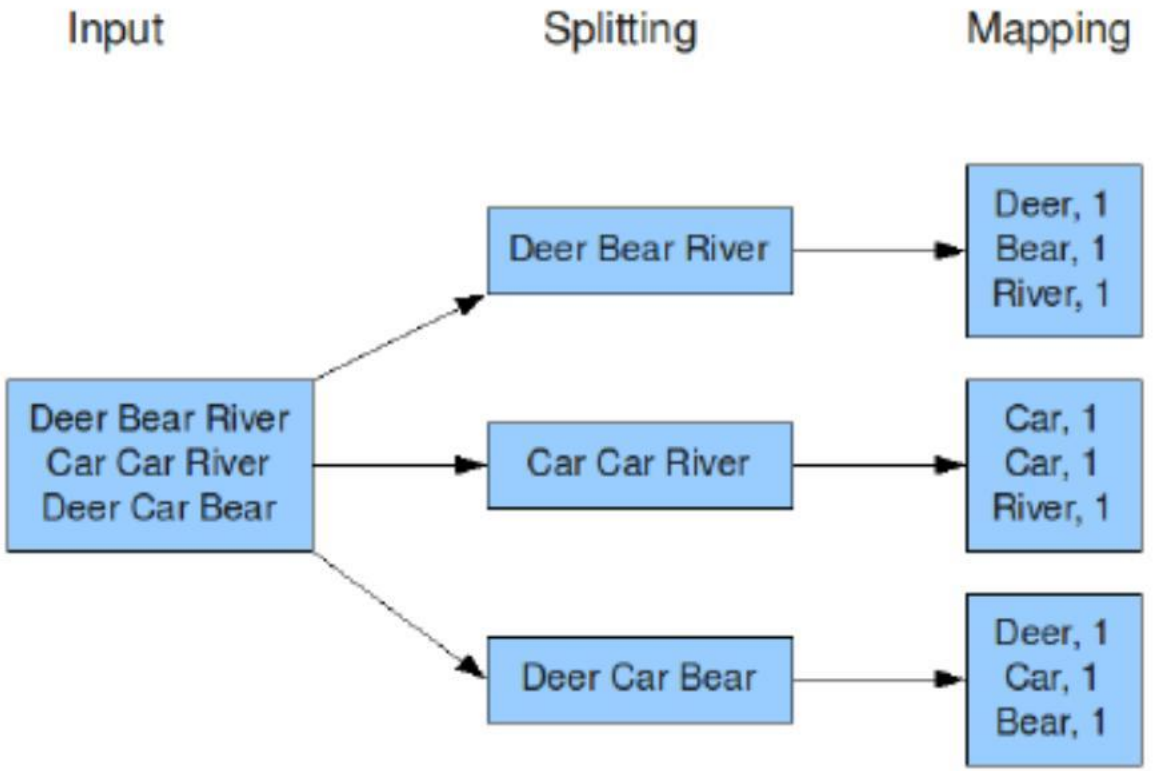
Deer Bear River
Car Car River
Deer Car Bear

Étape 1 : Comment découper les données ?
Chacune des machines doit travailler sur une partie du texte
Exemple de découpage : ligne par ligne.



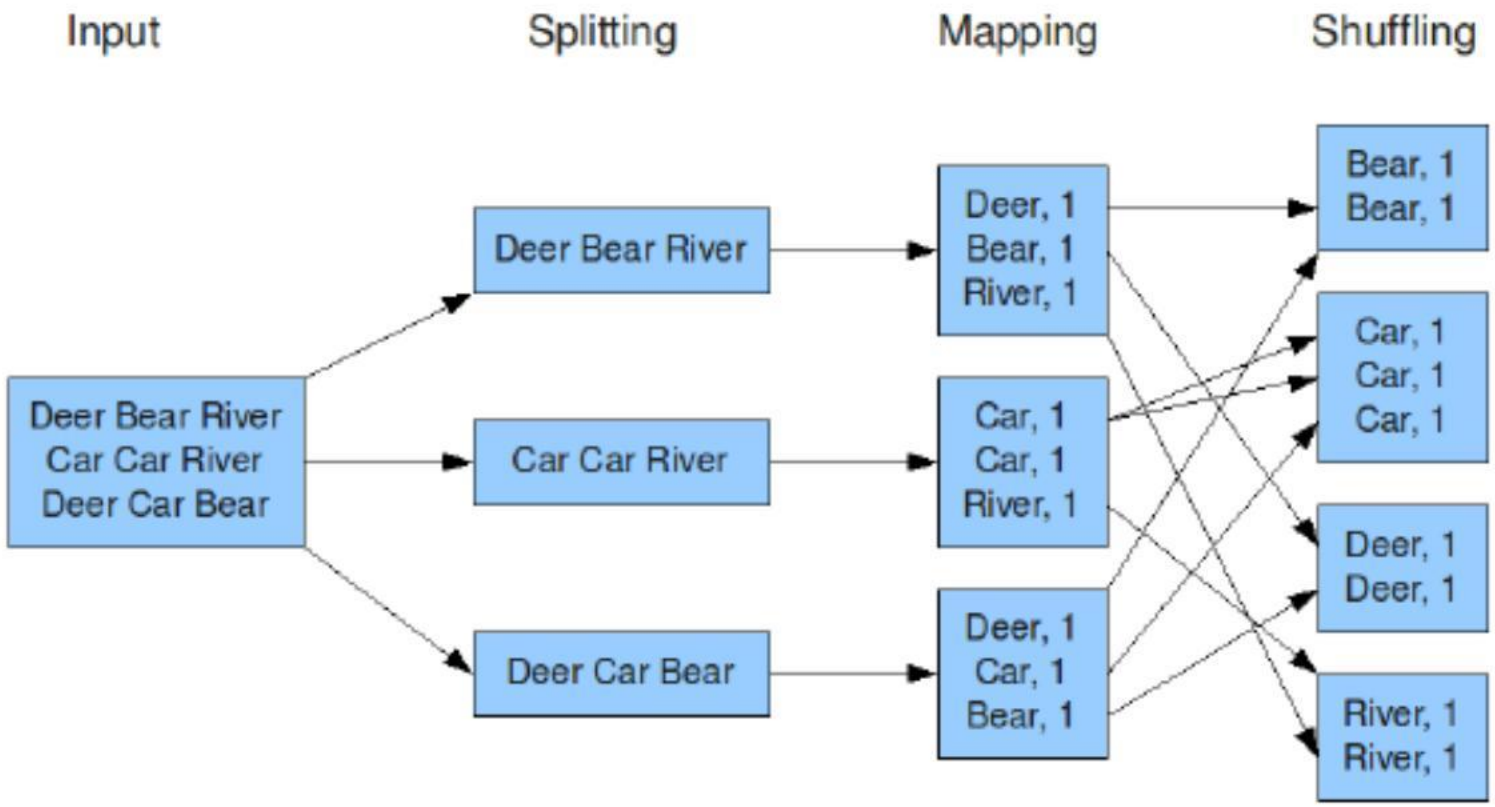
Exemple de WordCount

Étape 2 : Déterminer la clé et définir l'opération Map
Clé : le mot lui-même.
Opération Map : Générer le couple clé/valeur : (mot, 1)



Exemple de

Étape 3 : Grouper tous les couples par clé commune (**Shuffle**)
Cette opération est effectuée automatiquement et de manière distribuée par hadoop

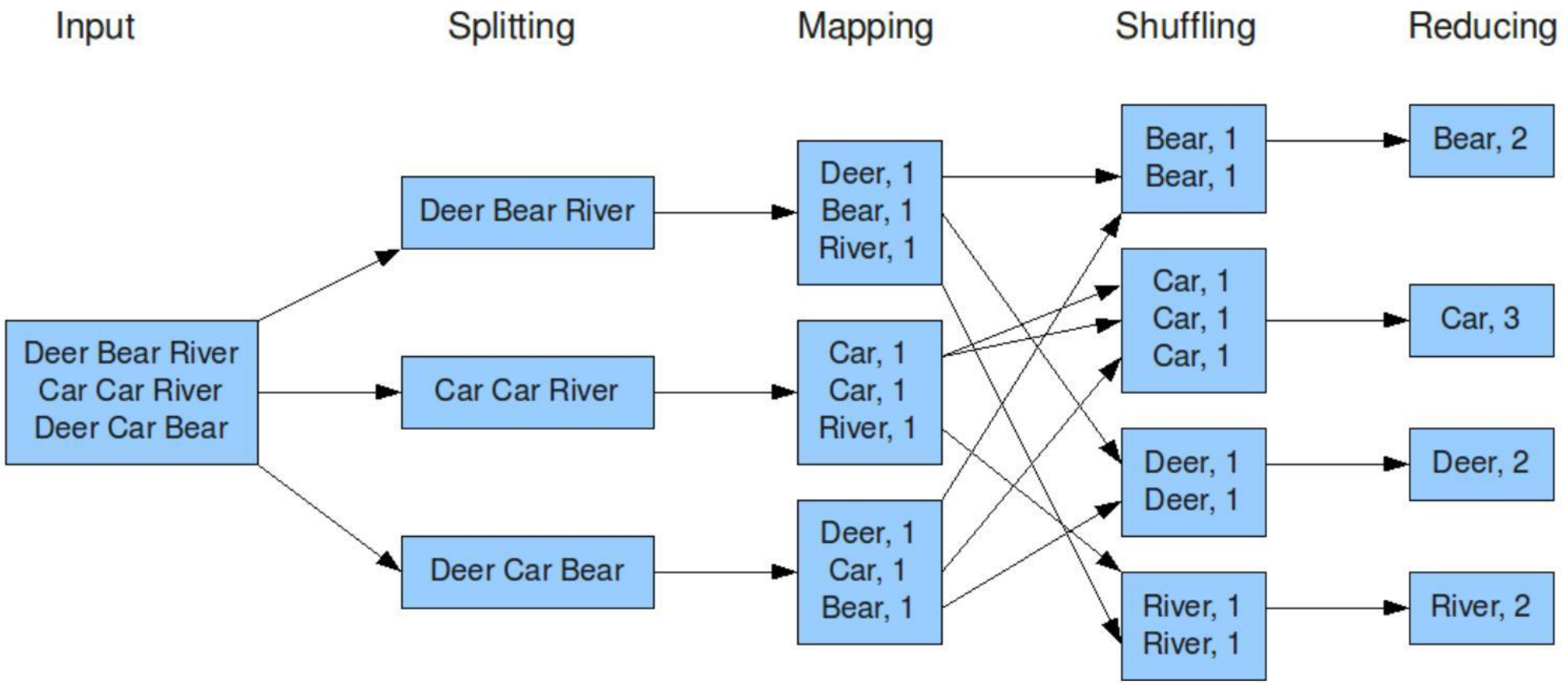


Exemple de WordCount

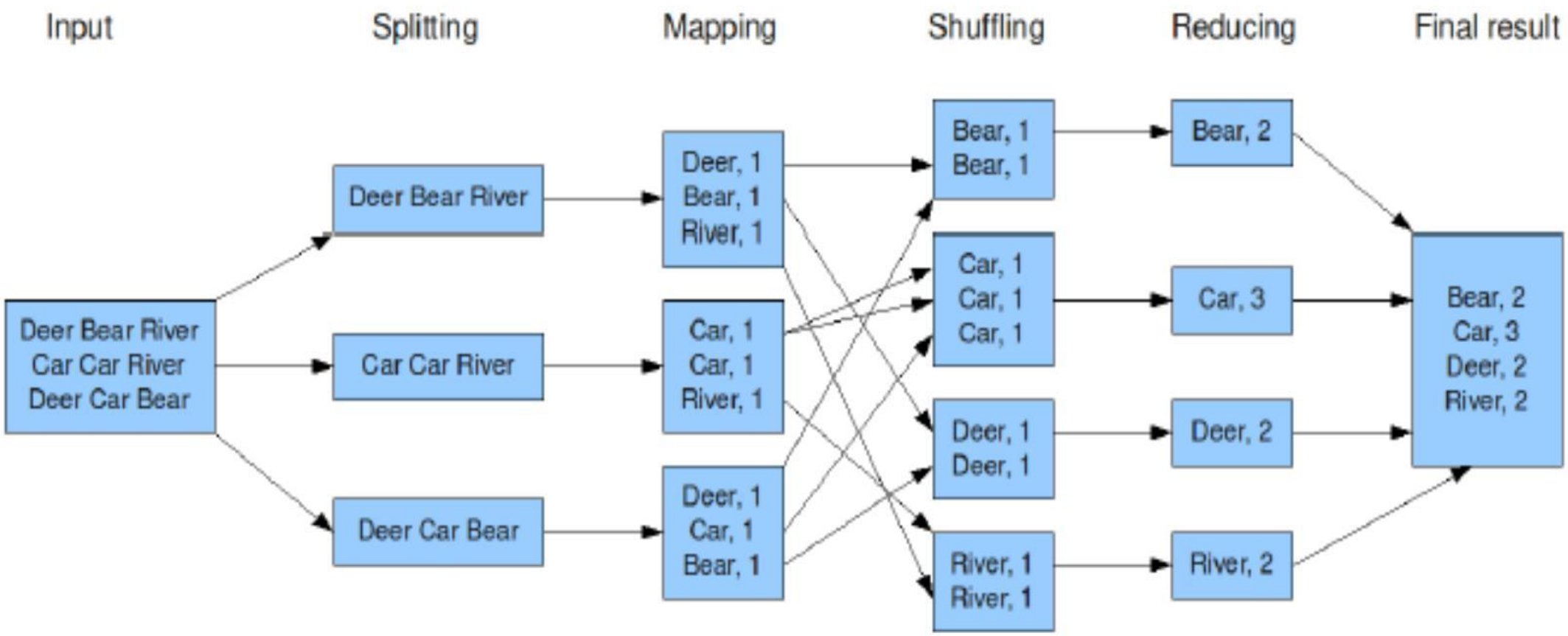
Étape 4 : Définir l'opération Reduce

Elle sera appliquée sur chacun des groupes par clé distincte.

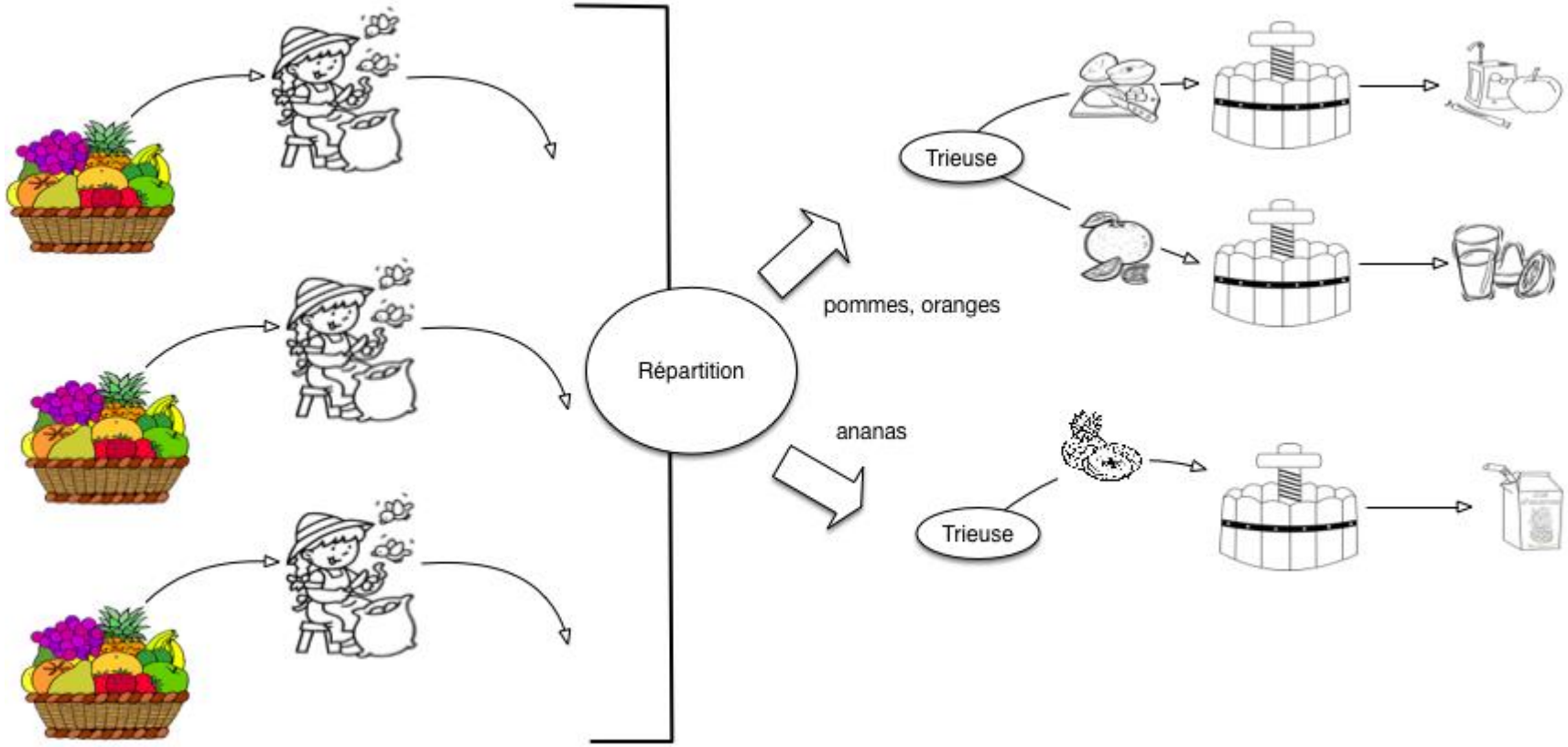
Additionner toutes les valeurs liées à la clé spécifiée.



Exemple de WordCount – Schéma



MapReduce : Rappel



Nous avons deux ateliers d'assemblage, le premier prenant en charge les pommes et les oranges, et le second les ananas.

MapReduce : Exemple

Multiplication d'une matrice et d'un vecteur

Cette exemple est légèrement plus complexe car il faut savoir comment multiplier une matrice à un vecteur.

Si on considère une matrice de taille 3×3:

$$\begin{pmatrix} 2 & 8 & 0 \\ 0 & 4 & 1 \\ 5 & 0 & 3 \end{pmatrix}$$

Et un vecteur de taille 3:

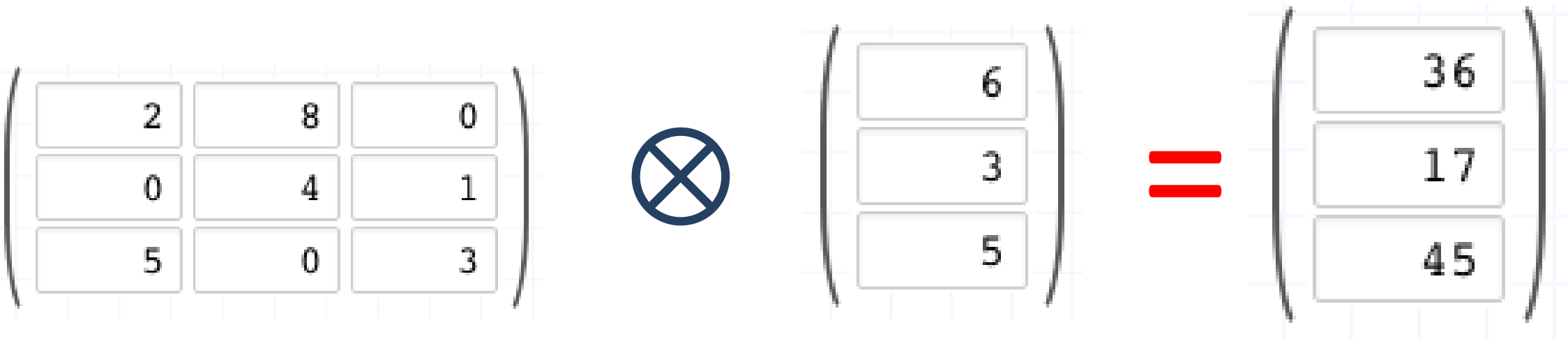
$$\begin{pmatrix} 6 \\ 3 \\ 5 \end{pmatrix}$$

MapReduce : Exemple

L'opération de multiplication d'une matrice et d'un vecteur consiste à additionner chaque ligne de la matrice aux valeurs du vecteur ce qui donne trois opérations.

Dans notre cas:

- **1ère opération:** 2*6 + 8*3 + 0*5 = 36
- **2è opération:** 0*6 + 4*3 + 1*5 = 17
- **3e opération:** 5*6 + 0*3 + 3*5 = 45



MapReduce : Exemple

Pour effectuer ce calcul avec un *job MapReduce*, on va considérer un triplet :
(i, j, a_{ij}) avec :

- i : **numéro de ligne**
- j : **numéro de colonne**
- a_{ij} : **la valeur**

Si la valeur dans la matrice est 0, elle ne sera pas représentée.

Dans cet exemple, on effectue le calcul pour une matrice de petite taille toutefois, pour justifier l'utilisation d'un *job MapReduce* et pour que ce calcul soit pertinent, il faudrait une opération dont la matrice est de taille très grande qui nécessiterait plusieurs minutes de calcul.

MapReduce : Exemple

Map: dans un premier temps, on considère que les valeurs du vecteur sont représentables sous la forme (i, v_i) et que les valeurs de la matrice sont a_{ij} (pour la valeur située à la i-ème ligne et j-ème colonne).

L'opération *Map* consiste à extraire une paire $(i, a_{ij} * v_i)$ qui consiste à effectuer la multiplication de la valeur a_{ij} (située à la ligne i et à la colonne j de la matrice) avec la valeur du vecteur v_i (située à la ligne i du vecteur).

Sachant que le résultat est un vecteur, on peut représenter le résultat de l'opération sous la forme de la paire indiquée précédemment avec i étant le numéro de ligne comme clé de la paire. La valeur étant le résultat de la multiplication $a_{ij} * v_i$.
Il y a donc, autant de tâche *Map* que de multiplication à effectuer.

MapReduce : Exemple

Dans notre cas, les paires obtenues sont:
Pour la 1ère ligne (0 est l'indice de la 1ère ligne):

•1ère opération *Map*: **(0, 2 * 6) soit (0,12)**
- 2e opération *Map*: **(0, 8 * 3) soit (0, 24)**

•Il n'y a pas de 3è opération pour la 1ère ligne car la valeur du vecteur est 0.
Pour la 2e ligne (1 est l'indice de la 2e ligne):

•3e opération *Map*: (1, 4 * 3) soit (1, 12). Comme précédemment à cause du 0, on passe directement au calcul de la 2è valeur de la ligne.

•4e opération *Map*: **(1, 1 * 5) soit (1, 5)**
Pour la 3e ligne (2 est l'indice de la 2e ligne):
- 5e opération *Map*: **(2, 5 * 6) soit (2, 30)**
- 6e opération *Map*: **(2, 3 * 5) soit (2, 15)**

MapReduce : Exemple

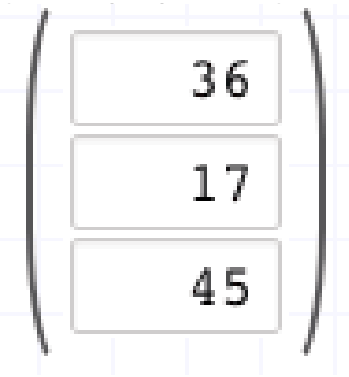
•A l'issue de toutes les opérations *Map*, on obtient la liste de paires: (0,12), (0, 24), (1, 12), (1, 5), (2, 30) et (2, 15).
Cette liste est indiquée de façon ordonnée toutefois dans la réalité elle n'est pas ordonnée car les opérations *Map* se font de façon parallèle et les résultats ne sont pas forcément ordonnancés de cette façon.

•***Shuffle* et *Sort***: cette étape va ordonner les paires suivant leur clé et répartir équitablement les paires sur les nœuds. On pourrait avoir une répartition de cette façon:

- Node 1: (0,12), (0, 24), (1, 12), (1, 5)
- Node 2: (2, 30) et (2, 15)

•***Reduce***: cette étape va permettre d'agréger toutes les valeurs suivant leur clé. On aura c
valeurs:

- (0, [12, 24])
- (1, [12, 5])
- (2, [30, 15])



•En additionnant les valeurs agrégées, on obtient la liste: (0, 36), (1, 17) et (2, 45).
•Sachant que les clés correspondent à l'indice de ligne du vecteur, on a bien le résultat de la multiplication:

Comment fonctionne MapReduce

Hadoop divise le travail en tâches. Il existe deux types de tâches :

- Les tâches de Map (Splits & Mapping)
- Les tâches de réduction (Shuffling, Reducing)

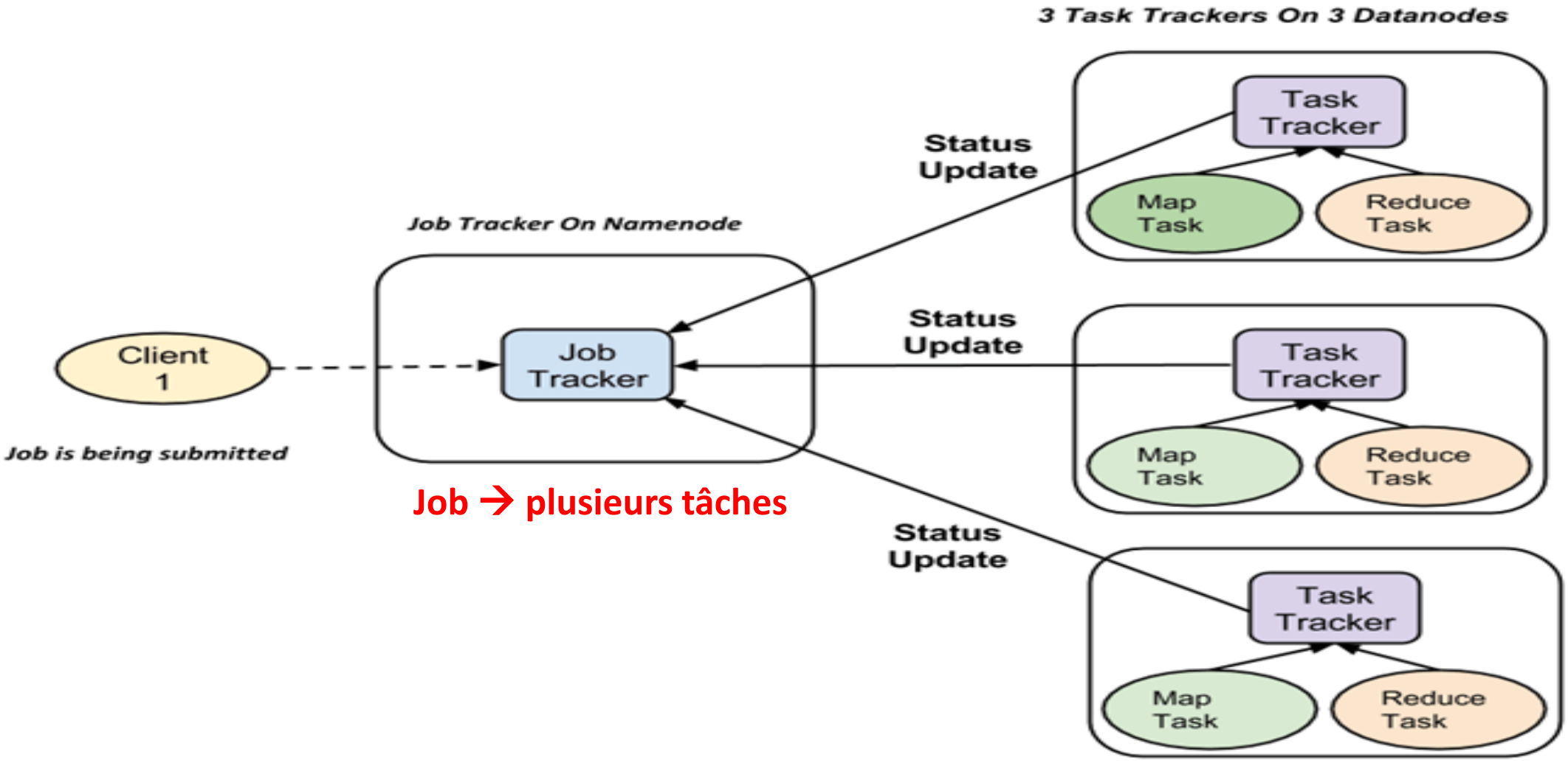
Le processus d'exécution complet (exécution des tâches Map et Reduce, ou les deux) est contrôlé par deux types d'entités appelées : **Jobtracker et TaskTrackers**

Jobtracker : Il agit comme un maître (responsable de l'exécution complète de la tâche soumise).

Plusieurs **Task Trackers** : Agissent comme des esclaves, chacun d'entre eux exécutant la tâche.

Pour chaque tâche soumise pour exécution dans le système, il y a un **Jobtracker** qui réside sur Namenode et il y a plusieurs **TaskTrackers** qui résident sur Datanode.

Comment fonctionne MapReduce



MapReduce en pseudo-code

```
1 // En pseudo code cela donnerait
2 Map(void * document) {
3     int cles = 1;
4     foreach mot in document
5         calculIntermediaire(mot, cles);
6 }
```

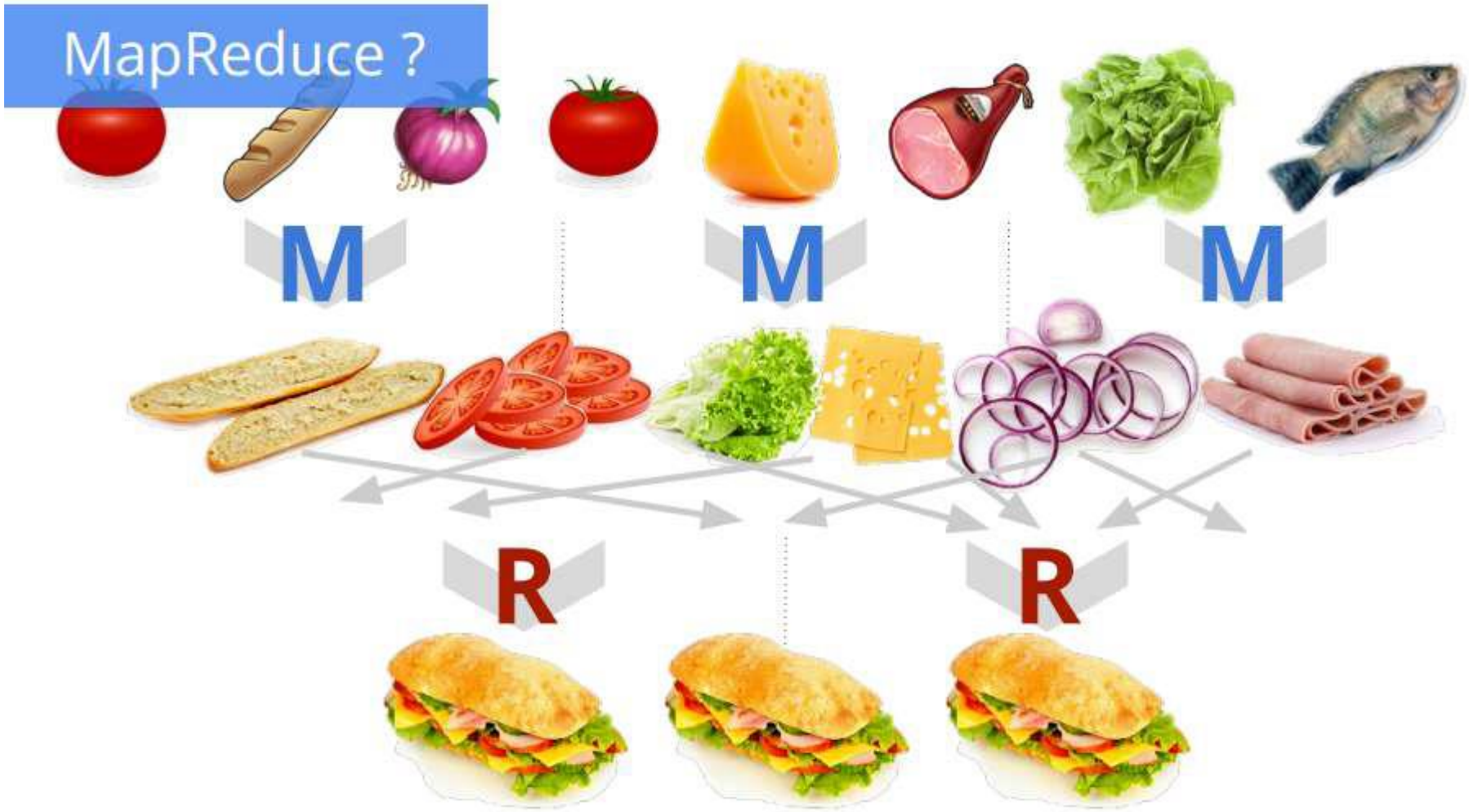
Le nombre de tâches Map ne dépend pas du nombre de nœuds, mais du nombre de blocs de données en entrée. Chaque bloc se fait assigner une seule tâche Map. De plus, toutes les tâches Map n'ont pas besoin d'être exécutées en même temps en parallèle

MapReduce en pseudo-code

```
1 // En pseudo code cela donnerait
2 Reduce(int cles, Iterator values) {
3     int result = 0;
4     foreach v in values
5         result += v;
6 }
```

Pour le traitement, les taches Reduce suivent le même schéma que les taches Map. Elles n'ont pas à s'exécuter parallèlement et dès qu'un nœud fini son traitement un autre lui est aussitôt assigné.

Hadoop : MapReduce



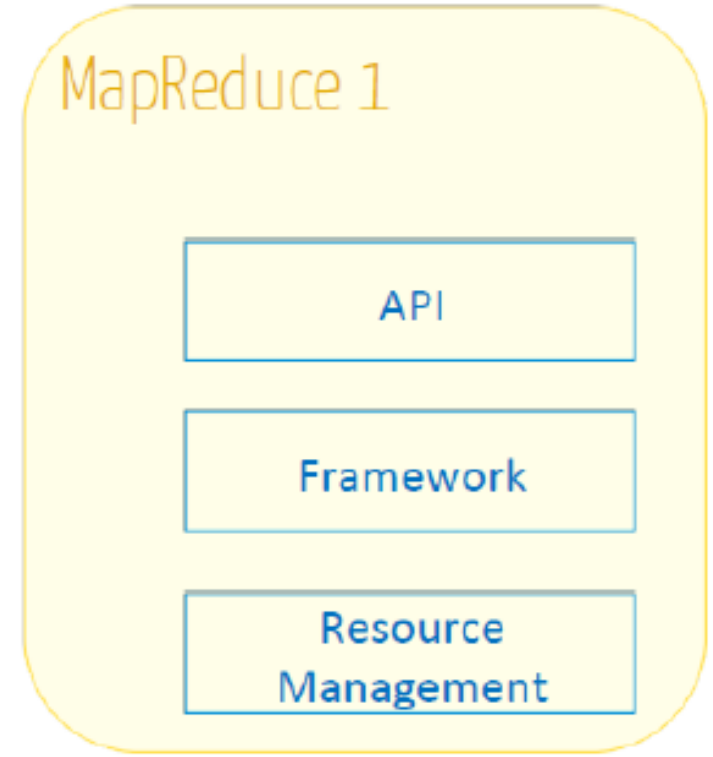
Hadoop : MapReduce V1 (MRv1)

MapReduce v1 intègre trois composants :

API : Permettre au programmeur l'écriture d'applications MapReduce.

Framework : Services permettant l'exécution des Jobs MapReduce, le Shuffle/Sort…

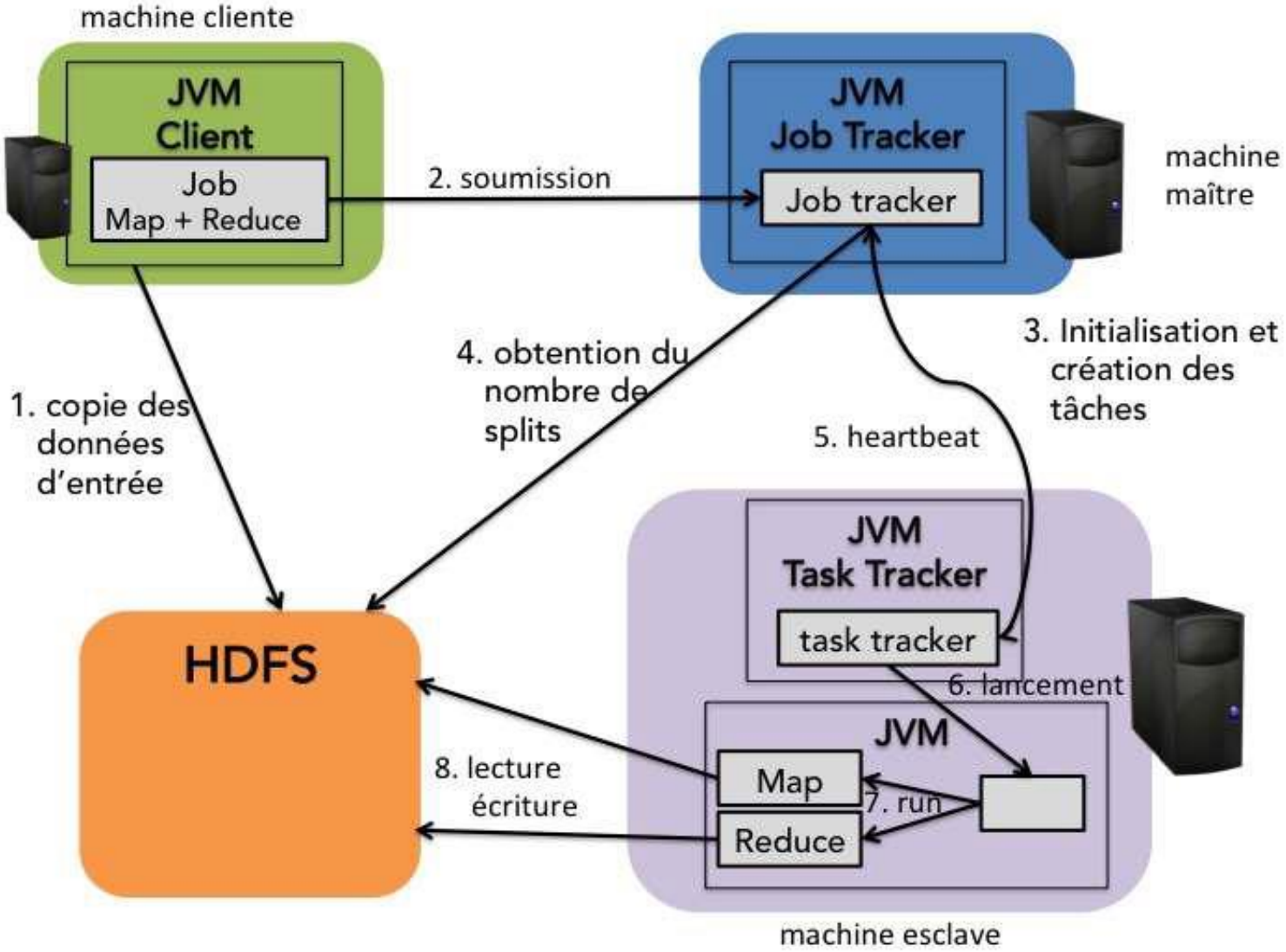
Resource Management : Infrastructure pour gérer les nœuds du cluster, allouer des ressources et ordonnancer les jobs.



Hadoop : MapReduce V1 (MRv1)

Un job MapReduce (ou une Application MapReduce) est divisé sur plusieurs tâches appelées mappers et reducers.
Chaque tâche est exécutée sur un noeud du cluster
Chaque nœud a un certain nombre de slots prédéfinis:
– Map Slots
– Reduce Slots
Un slot est une unité d'exécution qui représente la capacité du task tracker à exécuter une tâche (map ou reduce) individuellement, à un moment donné.
Le Job Tracker se charge à la fois:
– D'allouer les ressources (mémoire, CPU…) aux différentes tâches
– De coordonner l'exécution des jobs MapReduce
– De réserver et ordonnancer les slots, et de gérer les fautes en réallouant les slots au besoin

Soumission d'un Job MapReduce



Soumission d’un Job MapReduce

1. Un client hadoop copie ses données sur HDFS

2. Le client soumet le travail à effectuer au job tracker sous la forme d'une **archive.jar** et des noms des fichiers d'entrée et de sortie.

3. Le **job tracker** demande au **Namenode** où se trouvent les blocs correspondants aux données d'entrée.

4. Il détermine alors quels sont les **noeuds Task Tracker** les plus appropriés pour exécuter les traitements (colocalisation ou proximité des nœuds). Il envoie alors au **task tracker** sélectionné et pour chaque bloc de données, le travail à effectuer (Map, Reduce ou Shuffle, fichier .jar).

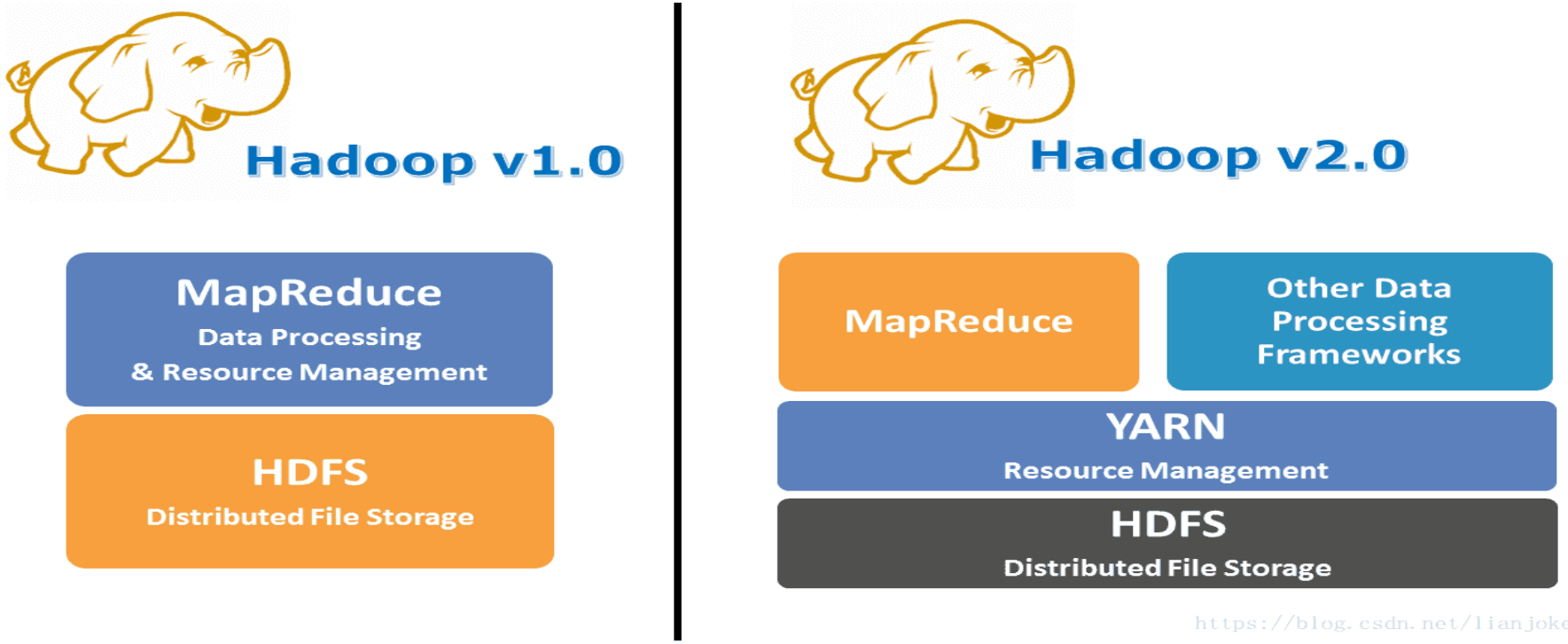
5. Les **task trackers** envoient régulièrement un message (**hearbeat**) au **job tracker** pour l'informer de l'avancement de la tâche et de leur nombre de slots disponibles.

6. Quand toutes les opérations envoyées aux **task trackers** sont confirmées comme étant effectuées, la tâche est considérée comme effectuée.

MapReduce v1 : Problèmes

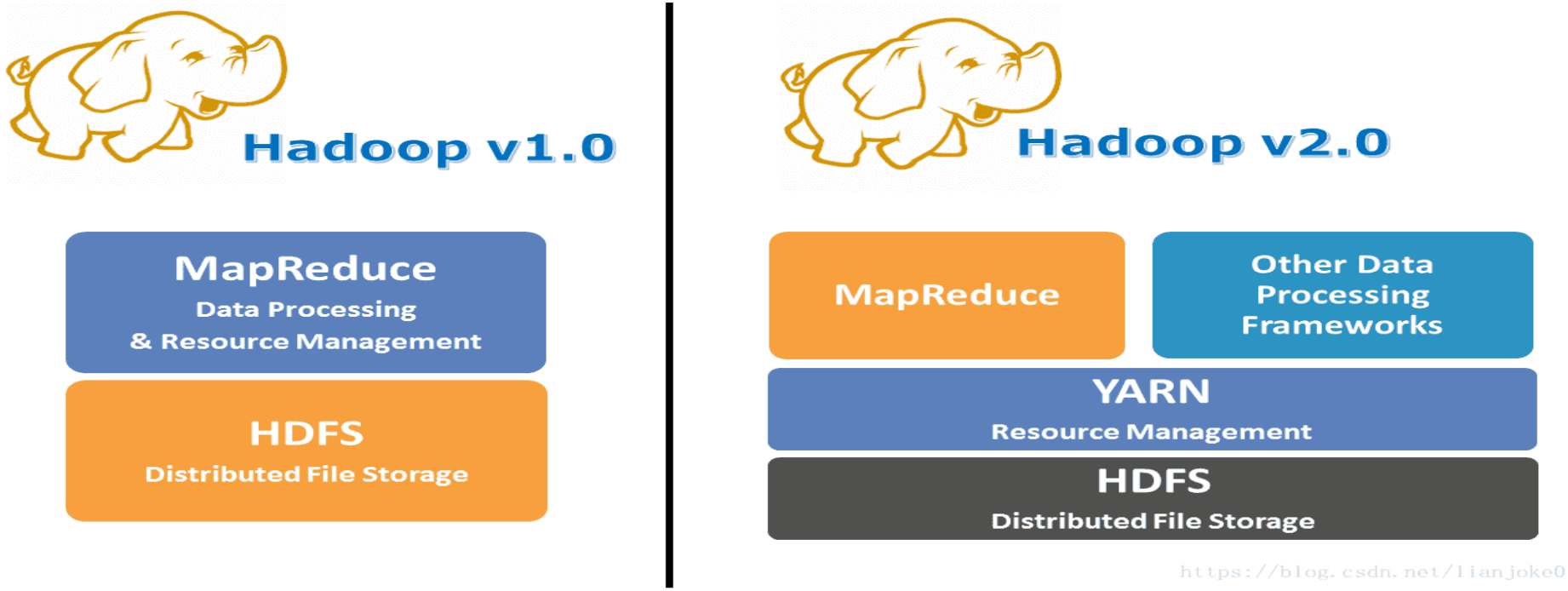
- Le Job Tracker s'exécute sur une seule machine, et fait plusieurs tâches (gestion de ressources, ordonnancement et monitoring des tâches…).
- Problème de scalabilité: les nombreux Datanodes existants ne sont pas exploités, et le nombre de nœuds par cluster limité à 4000.
- Si le Job Tracker tombe en panne, tous les jobs doivent redémarrer.
- Problème de disponibilité: Le nombre de slots map et de slots reduce est prédéfini
- Problème d'exploitation: si on a plusieurs jobs map à exécuter, et que les slots map sont pleins, les slots reduce ne peuvent pas être utilisés, et vice-versa.
- Le Job Tracker est fortement intégré à Map Reduce.
- Problème d'interopérabilité: impossible d'exécuter des applications non MapReduce sur HDFS

Hadoop : Versions



Pour répondre à ces différents problèmes, plusieurs améliorations ont été apportées à Hadoop (version 2.x). Notamment, l'architecture d'Hadoop a été modifiée pour introduire YARN : *Yet Another Ressource Negociator*, un Framework permettant d'exécuter n'importe quel type d'application distribuée sur un cluster Hadoop, pas uniquement les applications MapReduce.

MapReduce v2



- MapReduce v2 sépare la gestion des ressources de celle des tâches MR.
- Définition de nouveaux démons.
- Supporte les applications MR et non-MR.
- Pas de notion de slots: les nœuds ont des ressources (CPU, mémoire..) allouées aux applications à la demande

MapReduce v2 : Nouveaux démons

Resource Manager (RM) :
– Tourne sur le nœud master
– Ordonnanceur de ressources global
– Permet l'arbitrage des ressources entre plusieurs applications.

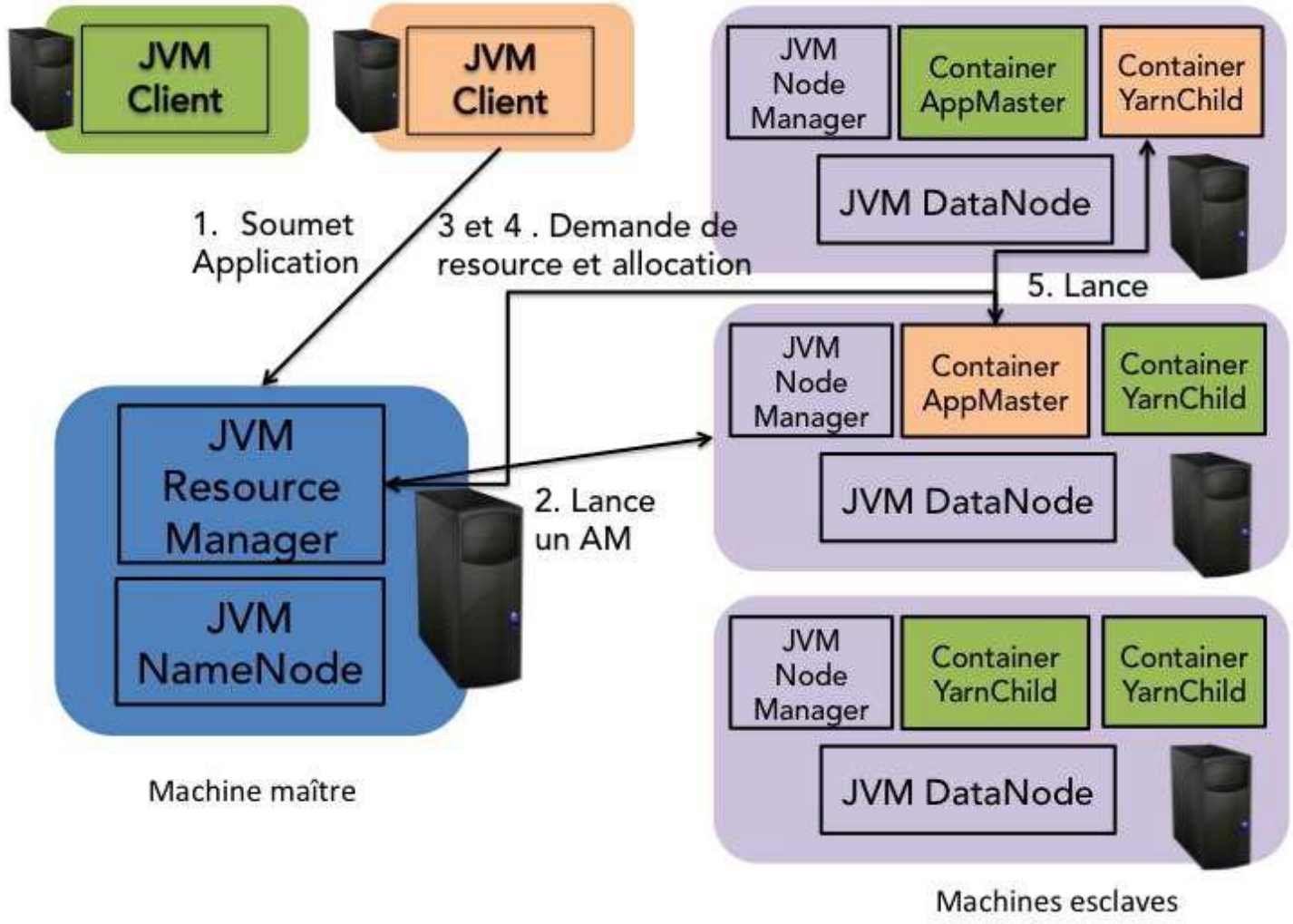
Node Manager (NM) :
– S'exécute sur les nœuds esclaves
– Communique avec RM.

Containers :
– Créés par RM à la demande
– Se voit allouer des ressources sur le nœud esclave.

Application Master (AM) :
– Un seul par application
– S'exécute sur un container
– Demande plusieurs containers pour exécuter les tâches de l'application.

Soumission d’un Job MapReduce v2

Le schéma ci-dessous illustre le schéma simplifié de soumission et d'exécution d'un travail dans Hadoop 2.0 avec YARN



Soumission d’un Job MapReduce v2

Le schéma de soumission et d'exécution d'un job dans cette nouvelle architecture est donc le suivant :

1. Un client hadoop copie ses données sur HDFS.

2. Le client soumet le travail à effectuer au **resource manager** sous la forme d'une

archive.jar et des noms des fichiers d'entrée et de sortie.

3. Le **resource manager** alloue alors un **container** pour **l'application master** sur un **node manager.**

4. L'application master demande au **resource manager** un ou plusieurs **containers**

avec des préférences de localisation dépendant de la localité des données d'entrée du travail.

5. Le **resource manager** alloue alors un ou plusieurs **containers** (child) à **l'application master**.

6. **L'application master** choisit parmi la liste des tâches (par exemple Map et Reduce) et demarre une instance de la tâche choisie dans un des **containers** qui lui a été alloué. Il collabore alors avec le **node manager** pour utiliser les ressources acquises. Il communique aussi souvent avec le **resource manager** (**message heartbeat**) pour la tolérance aux pannes.