Institut Supérieur d'Informatique et Mathématiques
Monastir

Cours : Big DATA

--Chapitre 2 : Traitement des données massives avec Hadoop--

Dernière MAJ:
Novembre 2024

Asma KERKENI asma.kerkeni@gmail.com

Objectifs

- Au terme de ce chapitre, vous serez capable de:
  - Expliquer le principe de localité de données
  - Décrire la vue d'ensemble et architecture(s) d'Hadoop
  - Décrire le fonctionnement de HDFS
  - Ecrire des algorithmes Map-Reduce
  - Comprendre la gestion de ressource avec YARN

Motivation

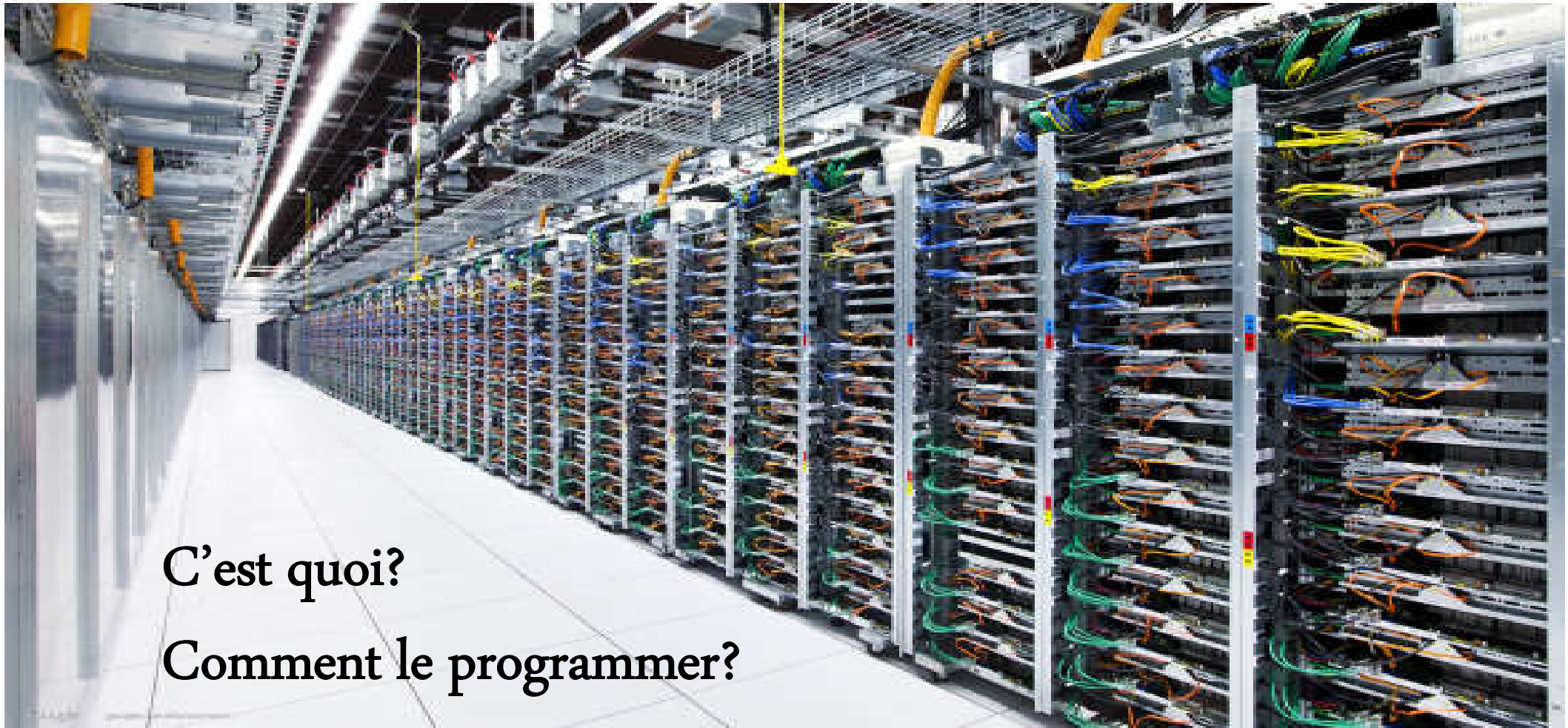
Too much data

Not enough compute power, storage or infrastructure

Plan

- Contexte d'apparition
- Présentation d'Hadoop
- Ecosystème d'Hadoop
- Système de fichiers distribué d'Hadoop : HDFS
- Hadoop Map-Reduce
- Allocation et gestion de ressources avec Yarn
- API JAVA d'Hadoop

Contexte



C’est quoi?

Comment le programmer?

Distribution des données et traitements

- Centre de données -

- Dans un centre de données, les serveurs sont empilés dans des baies (Rack) équipées pour leur fournir l'alimentation, la connexion réseau vers la grappe de serveur, la ventilation.
- Une baie contient environ 40 serveurs. Un centre de données contient quelques centaines de baies. Ordre de grandeur : quelques milliers de serveurs par centre.
- Combien des centres chez Google, Facebook, Amazon ... ? Des millions de serveurs.

Un serveur

Une baie (*rack*) de serveurs

BigData

Data Center

Distribution des données et traitements

- Principe du *Data Locality* -

- C'est toujours l'accès aux données qui coute cher, pas le calcul lui-même une fois les données dans le cœur de calcul.
- Principe de *data locality* ou proximité des données.

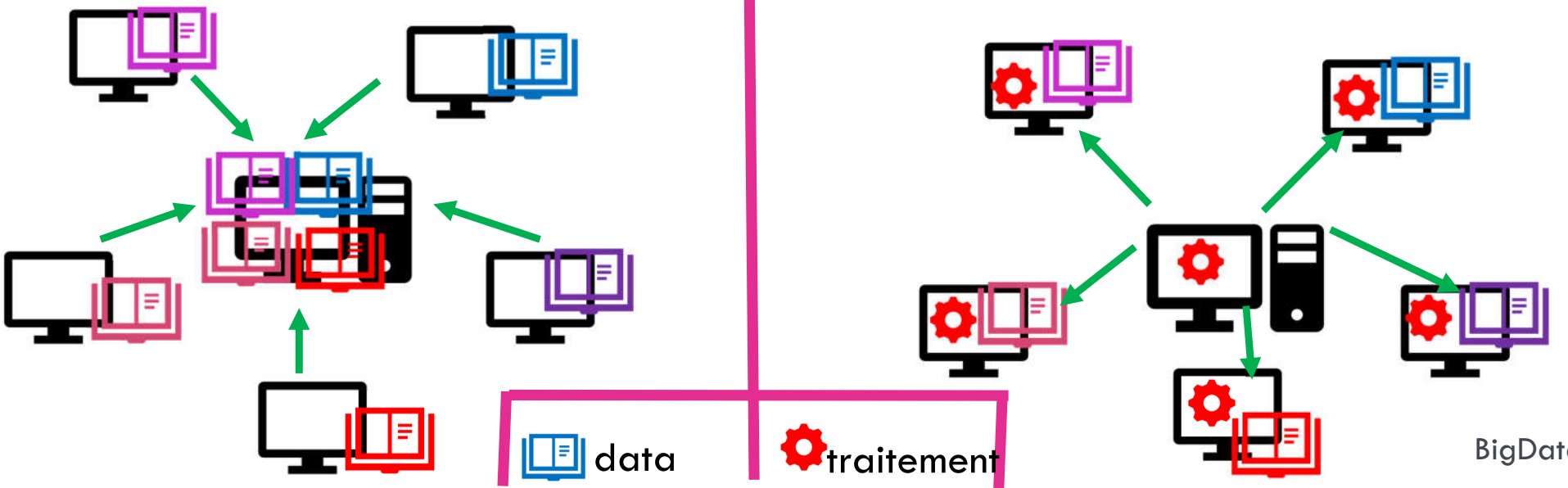
Classique:

Ramener les données au serveur pour les traiter

Big Data:

Amener les codes de traitements aux données

data locality



Distribution des données et traitements

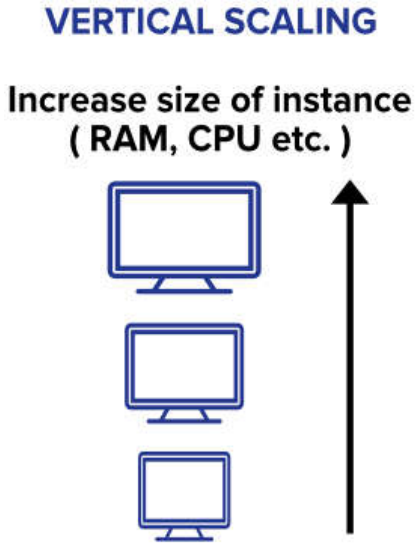
-Besoins du Big Data-

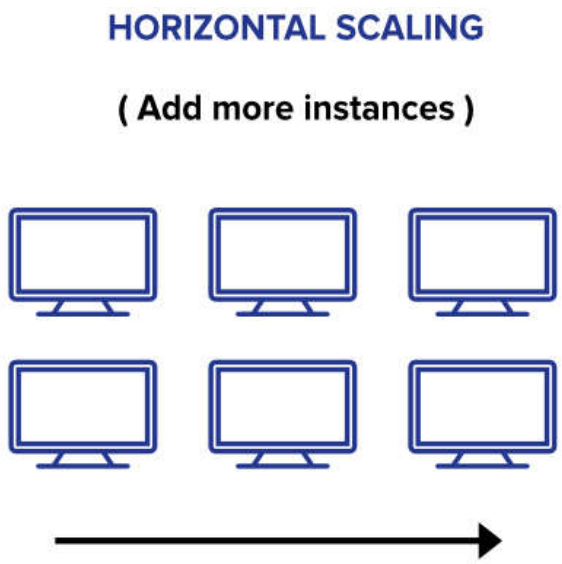
- **Besoins:** Frameworks dédiés prenant en charge les enjeux du calcul distribué :
  - Optimisation des transferts disques et réseau en limitant les déplacements de données
  - Tolérance aux pannes *(fault tolerance).*
  - Scalabilité pour permettre d'adapter la puissance au besoin (*scalability*)

Distribution des données et traitements

-Scalabilité-

☐ **Scalabilité:** C'est la capacité d'un système à gérer des quantités croissantes de données, sans diminution significative de ses performances





- Plus facile de maintenir une seule machine.
- Control centralisé sur les données et les calculs.

- Mise à niveau illimitée de la puissance de calcul d'un système
- Tolérance aux pannes

10

Hadoop: Présentation et écosystème

Présentation du Framework

- Hadoop est un framework **open source, écrit en Java** et géré par la fondation Apache.
- **Java est le langage de préférence** pour écrire des programme Hadoop natifs. Néanmoins, il est possible d'utiliser Python, Bash, Ruby, Perl ...
- Hadoop offre un **rapport performance-prix très compétitif** (Amazon EMS, réutilisation de PC existants, aucun coûts de licences ni de matériel spécialisé HPC).
- Le nom "Hadoop" était initialement celui d'un éléphant en peluche, jouet préféré du fils de **Doug Cutting.**

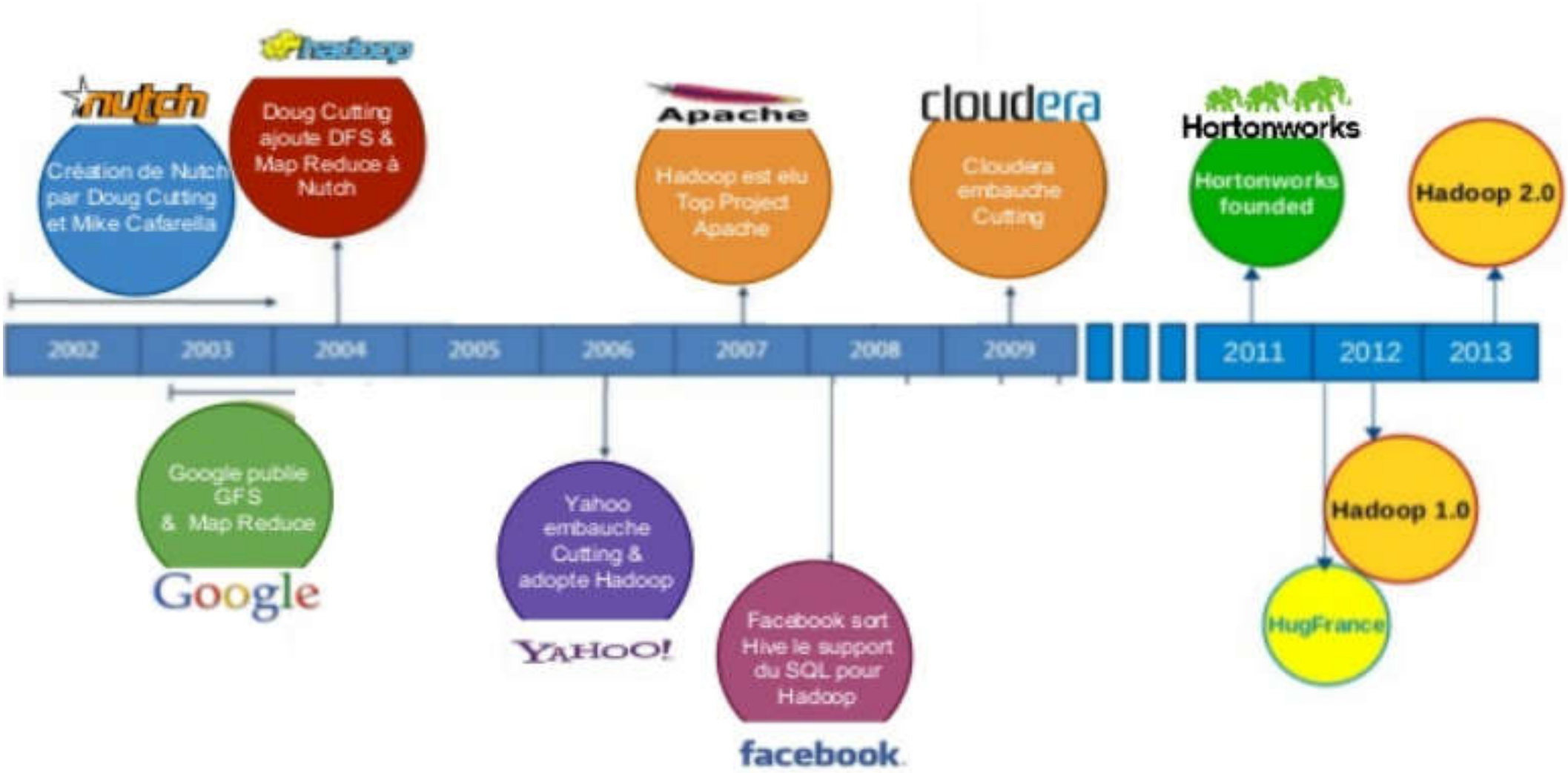
Doug Cutting créateur de Hadoop.

Présentation du Framwork

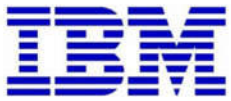
- Le projet Hadoop consiste en deux grandes parties:
  - Stockage des données : HDFS (Hadoop Distributed File System)
  - Traitement des données : MapReduce / Yarn

- **Principe:**
  - Diviser les données
  - Les sauvegarder sur une collection de machines, appelée 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. Les machines peuvent être hétérogènes.

Historique de Hadoop

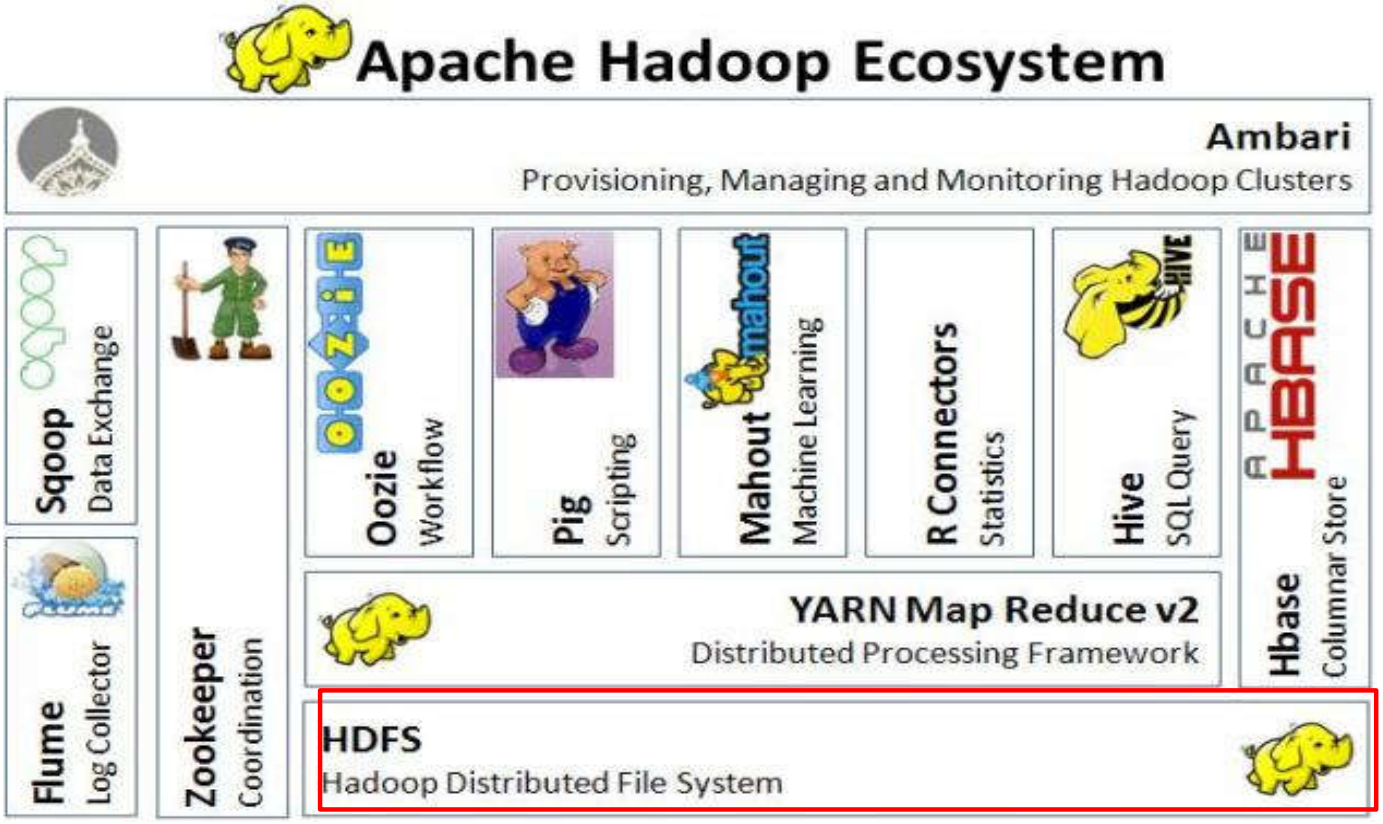


Qui utilise Hadoop ?



Ecosystème de Hadoop

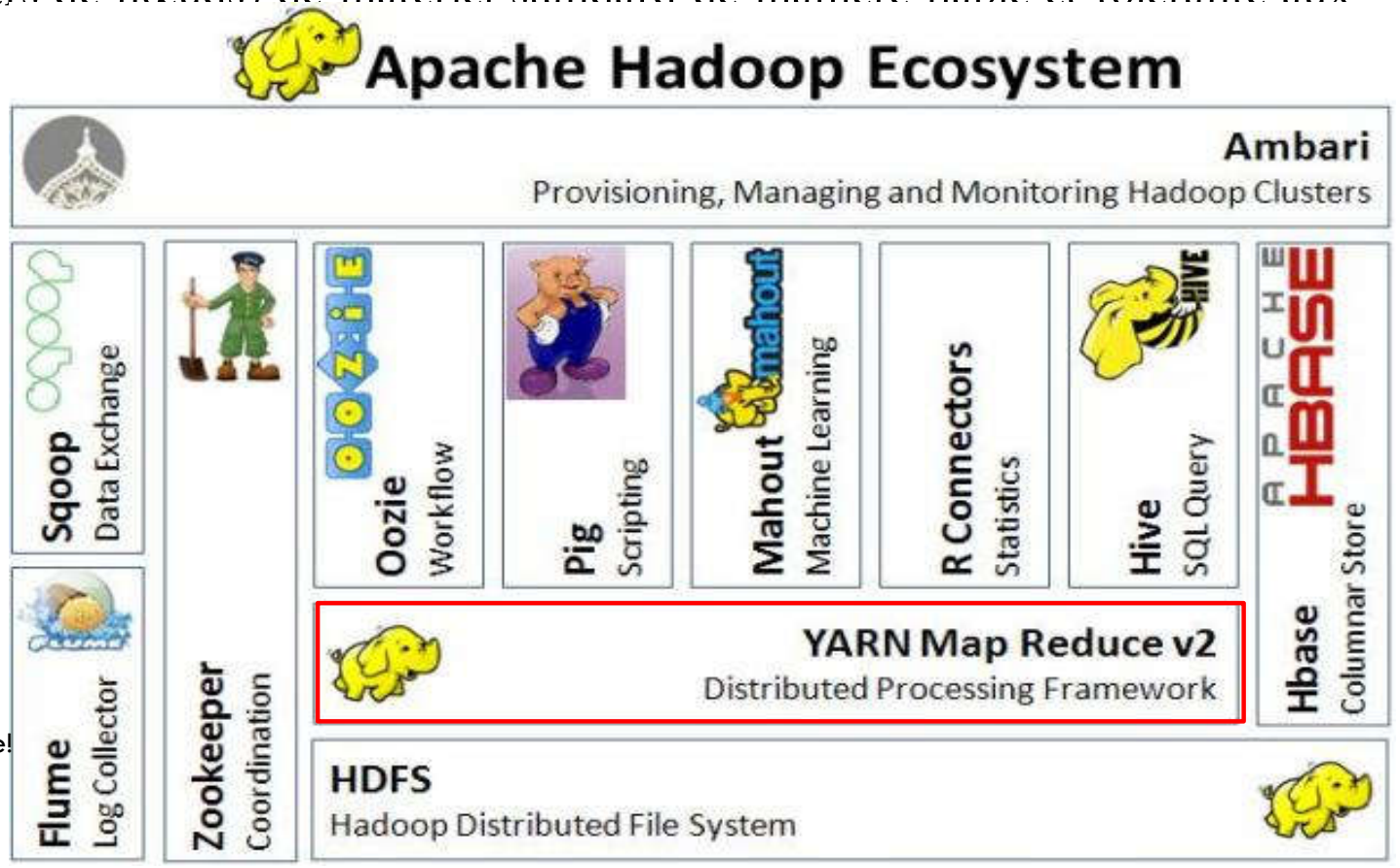
- Le système de fichiers distribués Hadoop (HDFS) est basé sur le système de fichiers Google (GFS).
- Il fournit un système de fichiers distribué conçu pour fonctionner sur du matériel standard et présente de nombreuses similitudes avec les systèmes de fichiers distribués existants avec certaines différences.
- Il est hautement tolérant aux pannes et est conçu pour être déployé sur du matériel à faible coût.



NB: La liste présentée ci-dessus n'est pas exhaustive!

Ecosystème de Hadoop

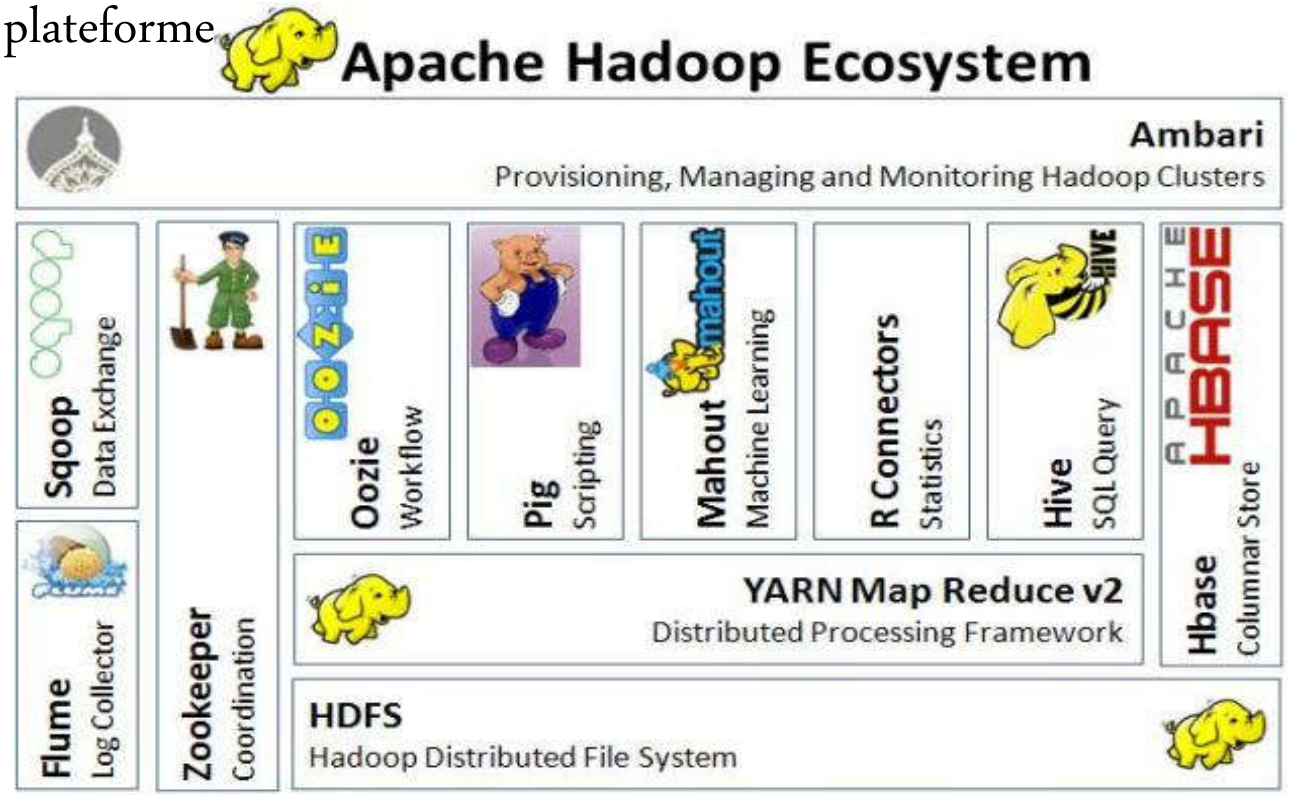
- MapReduce est un modèle de programmation parallèle pour l'écriture d'applications distribuées conçu par Google.
- Permet un traitement efficace de grandes quantités de données (plusieurs téraoctets) sur des grappes étendues (des milliers de nœuds) de matériel standard de manière fiable et tolérante aux pannes.



NB: La liste présentée ci-dessus n'est pas exhaustive!

Ecosystème de Hadoop

- Plusieurs outils existent pour permettre:
  - L'extraction et le stockage des données de/sur HDFS
  - La simplification des opérations de traitement sur ces données
  - La gestion et coordination de la plateforme
  - Le monitoring du cluster



NB: La liste présentée ci-dessus n'est pas exhaustive!

HDFS: Hadoop Distributed File System

HDFS : Hadoop Distributed File System

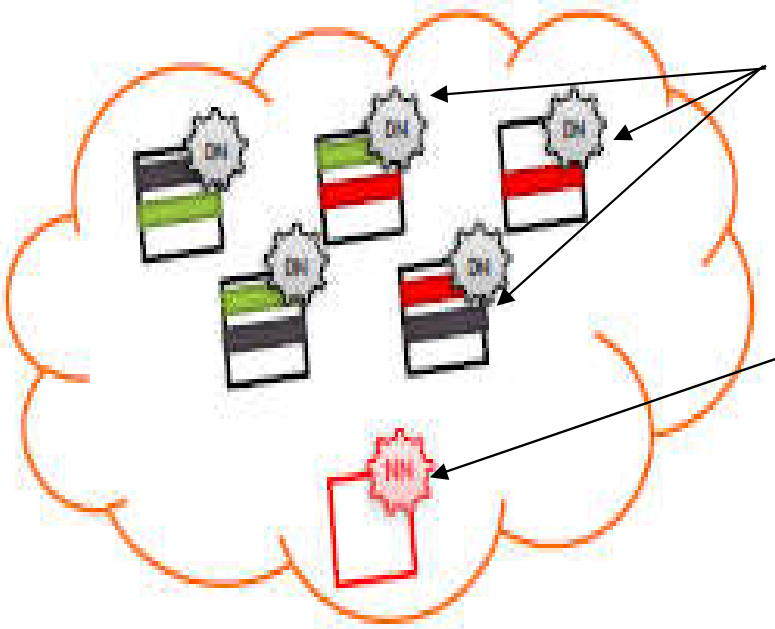
- Définition-

- HDFS est **un système de fichiers distribué** et la couche native **de stockage et d'accès** à des données d'Hadoop.
- Conçu pour stocker et gérer des fichiers de **très grande taille** dans un cadre distribué de manière **transparente** comme s'ils étaient sur le disque dur local.
- HDFS est **hautement tolérant aux pannes** et conçu avec un **matériel à faible coût**.
- Pour garantir une tolérance aux pannes, les blocs de chaque fichier sont **répliqués,** de manière intelligente, sur plusieurs machines.

HDFS : Hadoop Distributed File System

- Types des noeuds-

- 2 types de noeuds dans HDFS:
  - 1 seul noeud pour les méta-données : les noms et blocs des fichiers ainsi que leur localisation dans le cluster (un gros annuaire).
  - plusieurs noeuds pour sauvegarder les données réelles



Data Node:

Exemple de configuration du Name Node
Processors: 2 Quad Core CPUs running @ 2 GHZ
RAM: 128 GB
Disk: 6 x 1TB SATA
Network: 10 Gigabit Ethernet

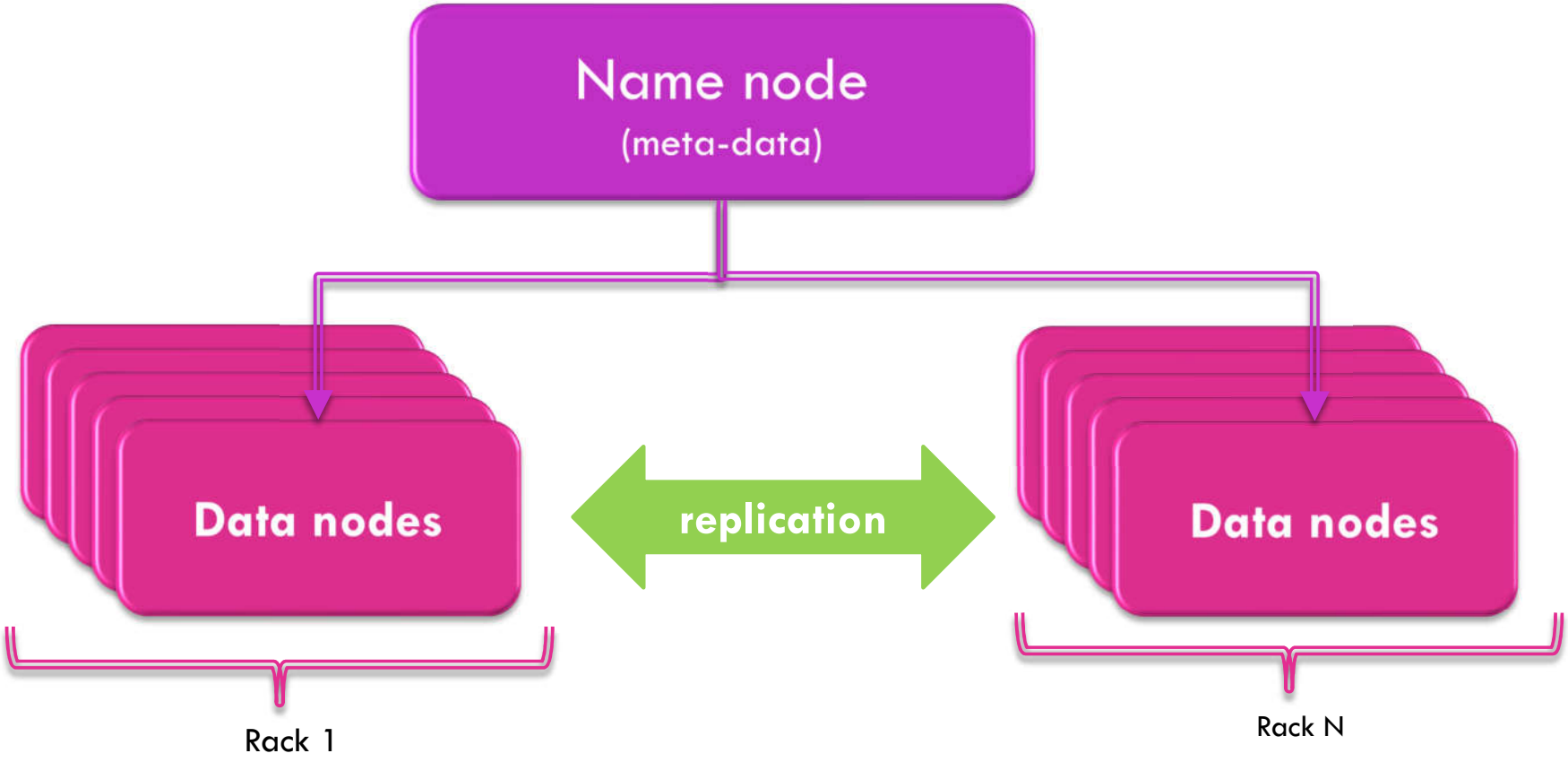
Name Node

Exemple de Configuration du Data Node
Processors: 2 Quad Core CPUs running 4 2 GHz
RAM: 64 GB
Disk: 12-24 x 1TB SATA Network: 10 Gigabit Ethernet

HDFS: Hadoop Distributed File System

- Architecture Hadoop 1.0 -

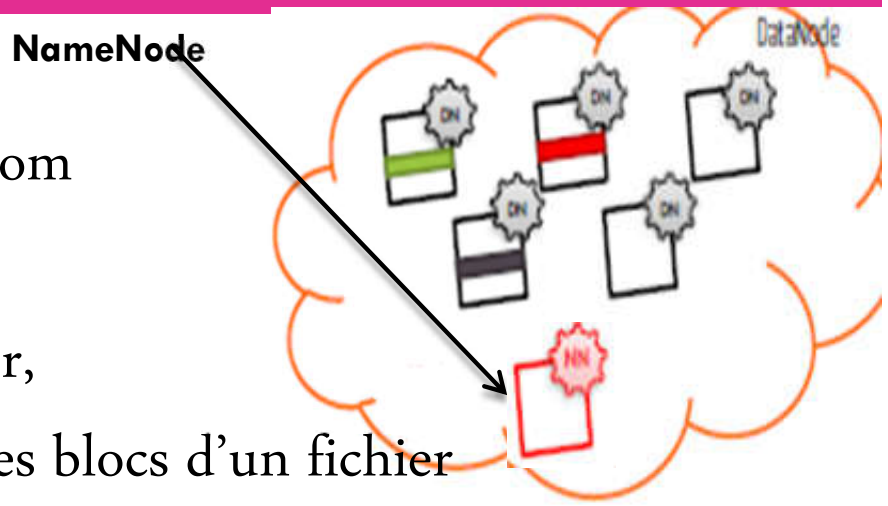
☐ Topologie Maitre-esclave



HDFS : Hadoop Distributed File System

- Types des noeuds-

☐ NameNode

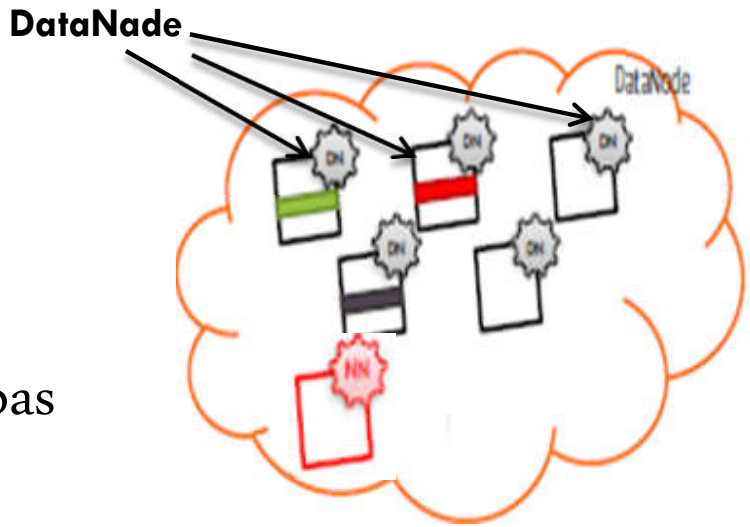


- ❑ Une machine séparée sur lequel tourne un dénom appelé namenode
- ❑ Contient des métadonnées, pas de données user,
- ❑ Permet de retrouver les nœuds qui hébergent les blocs d'un fichier
- ❑ Cordonne l'accès aux données dans le système de fichiers.
- ❑ La responsabilité principale d'un Namenode est de stocker le namespace de HDFS :
  - L' arborescence des répertoires
  - Les permissions des fichiers
  - Le mapping des fichiers aux blocs
- ❑ Ces données sont stockées dans deux structures : FsImage et EditLog

HDFS : Hadoop Distributed File System

- Types des noeuds-

☐ DataNode (nœud de données)



- Uen machine sur laquelle tourne un démon esclave
- Il stocke les données réelles.
- Exécute les demandes de lecture et d'écriture de bas niveau des clients du système de fichiers.

- Il y a un DataNode pour chaque machine au sein du cluster.

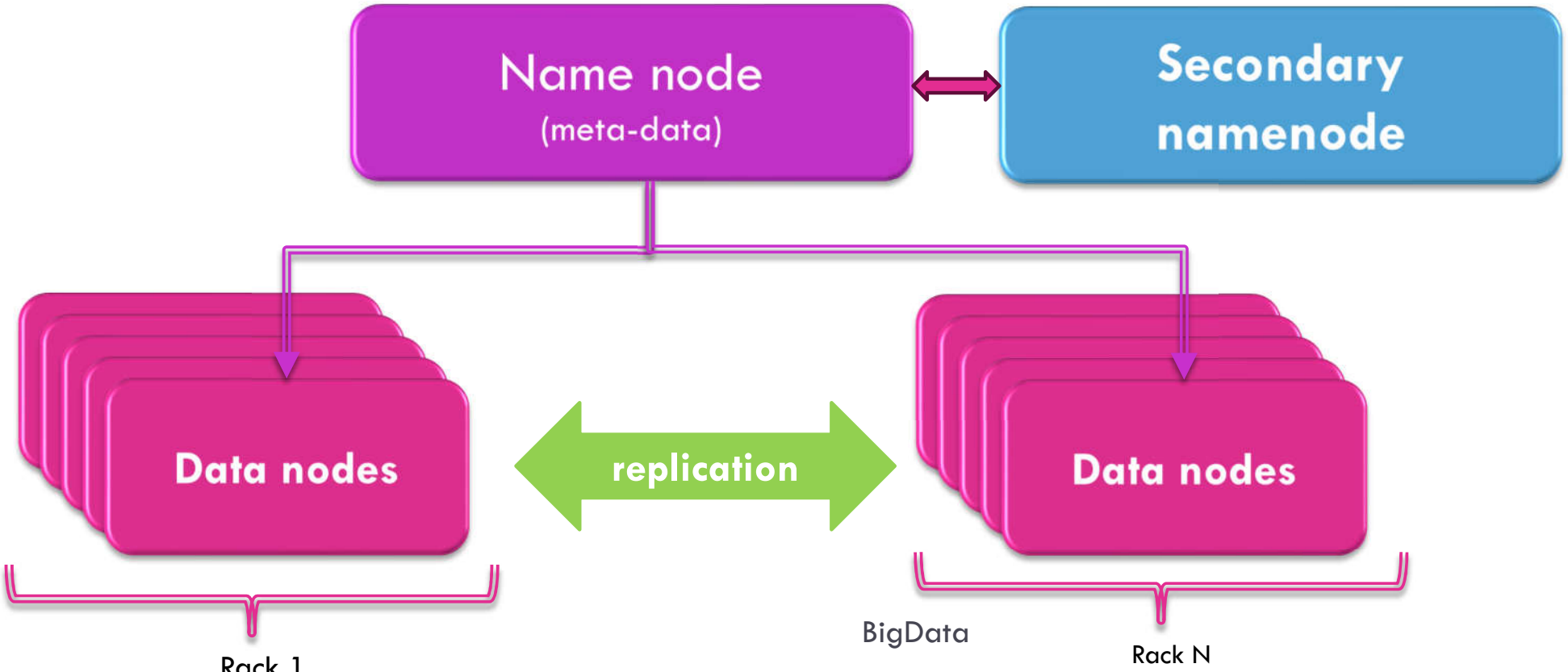
- Les Datanodes sont sous les ordres du Namenode et sont surnommés les Workers (ou esclaves).

→ Ils sont donc sollicités par les Namenodes lors des opérations de lecture et d'écriture!

HDFS: Hadoop Distributed File System

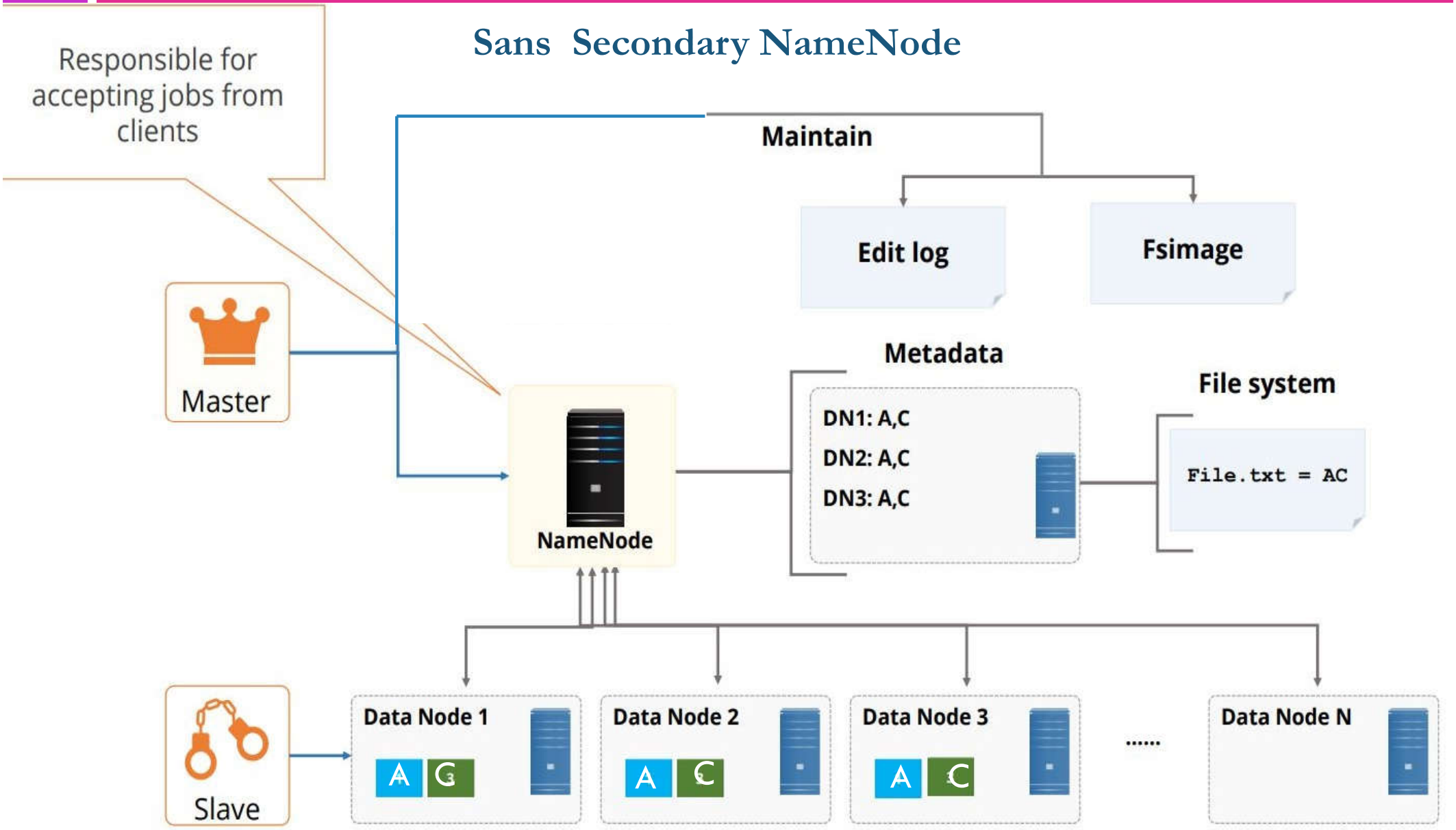
- Architecture Hadoop 2.0 -

- **Problème** : Trop de charge sur le NameNode
- **Solution**: Ajout du **Secondary NameNode**: un assistant du NameNode pour réduire sa charge (Ne le remplace pas)



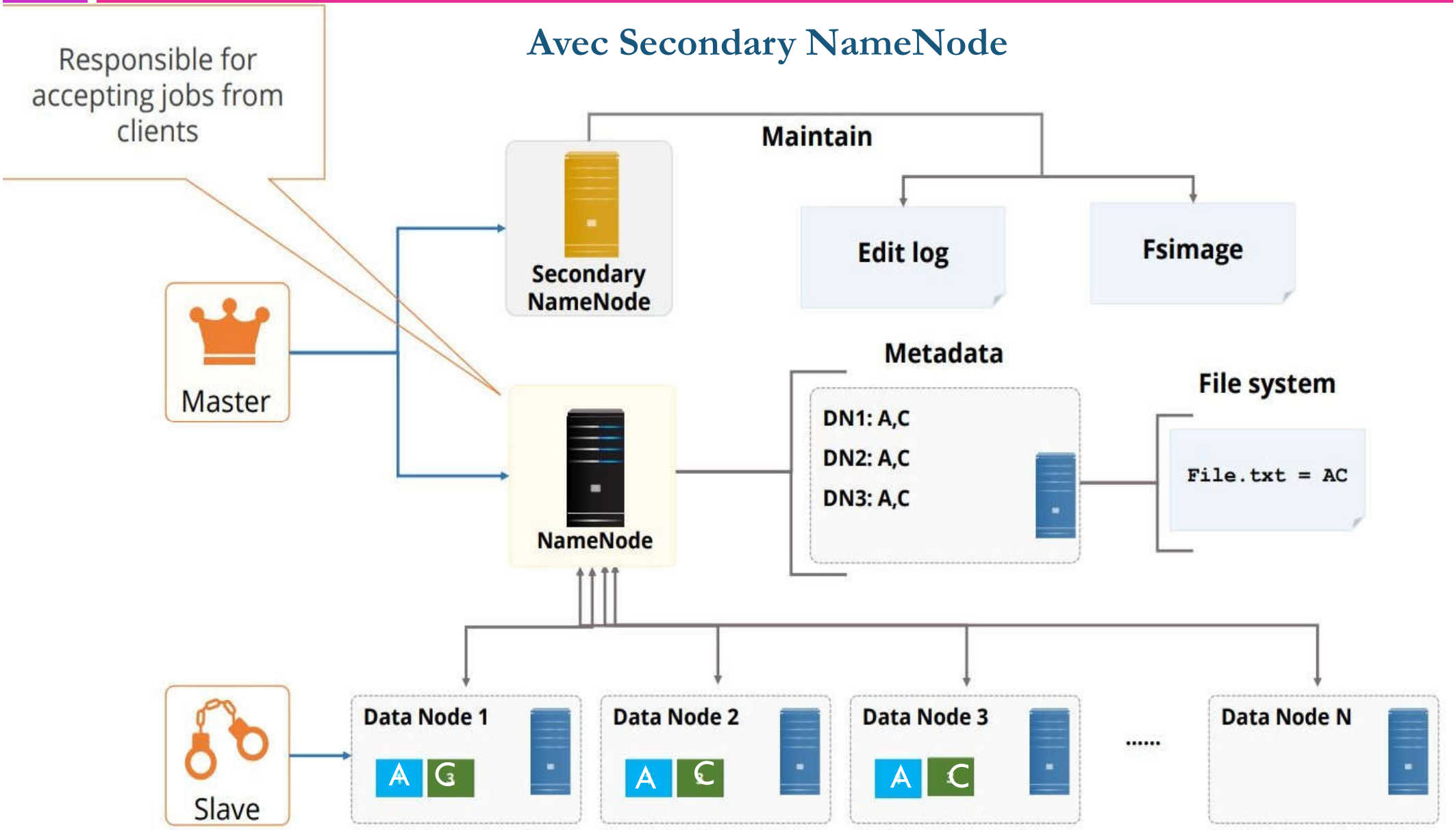
HDFS: Hadoop Distributed File System

- Rôle du Secondary Namenode-



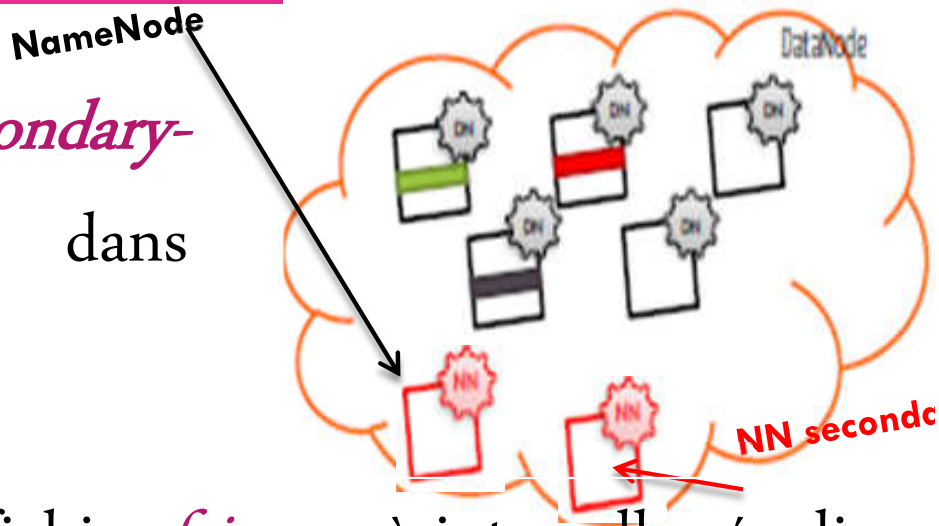
HDFS: Hadoop Distributed File System

- Rôle du Secondary Namenode-



HDFS: Hadoop Distributed File System

- Rôle du Secondary Namenode-

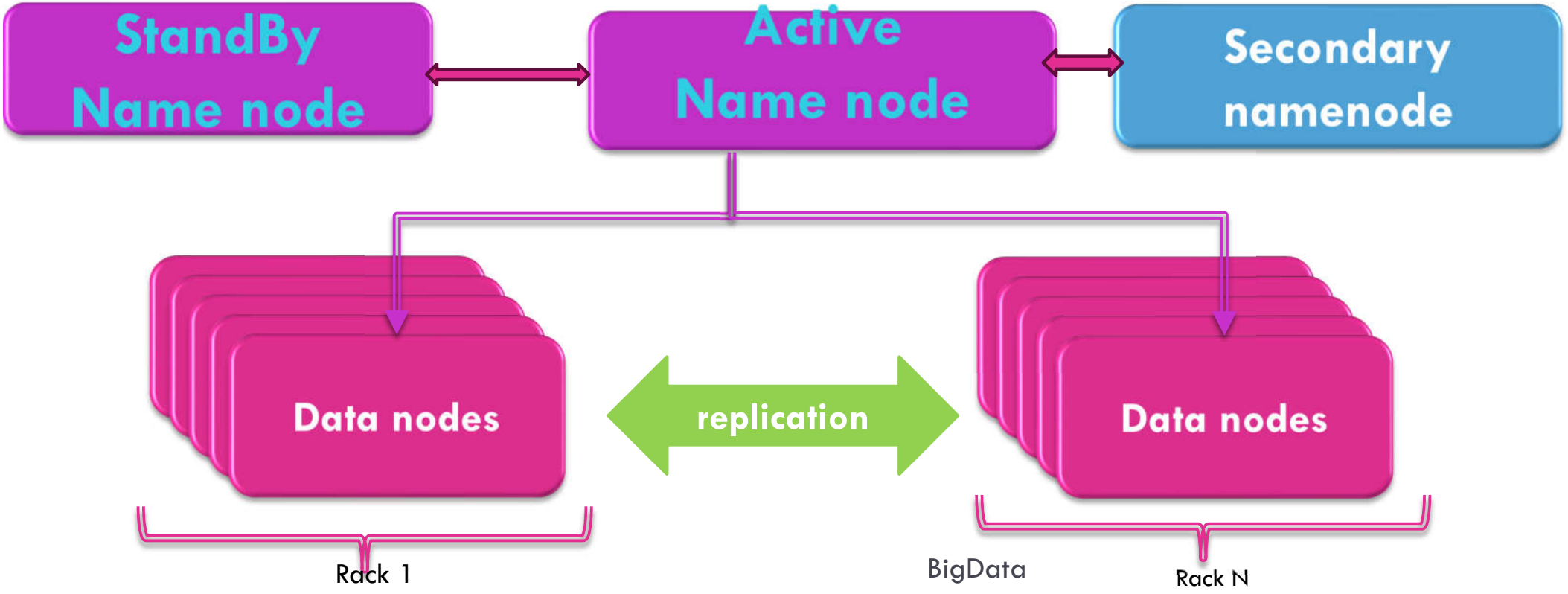


- Ce Nœud supplémentaire appelé ***Secondary-Namenode*** a été mis en place dans l'architecture Hadoop version2.
- le *Secondary NameNode* met à jour le fichier *fsimage* à intervalle régulier en y intégrant le contenu des *editlogs*.
- Le *SecondaryNameNode* intervient soit :
  - lorsque le fichier *editlogs* atteint une taille prédéfinie.
  - à intervalle régulier (par exemple une fois par heure).

HDFS: Hadoop Distributed File System

- Architecture Hadoop 2.0 -

- **Problème** : Le Namenode est un **SPOF** (Single Point Of Failure): Si ce service est arrêté, le cluster sera nom disponible!
- **Solution**: Possibilité de mettre plusieurs instances du NameNode (un actif et un ou plusieurs StandBy : solution High Availability (HA) mais plus côuteuse).

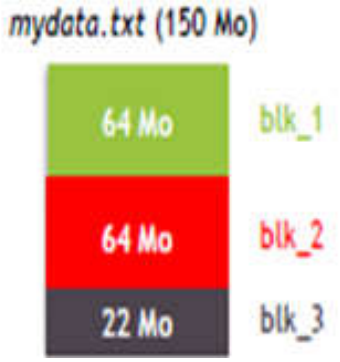


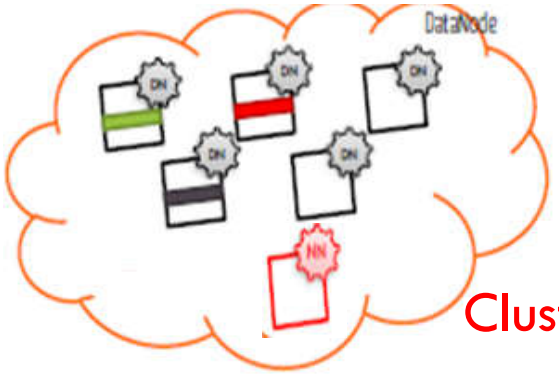
HDFS: Hadoop Distributed File System

-Décomposition en blocs-

- Les fichiers sont physiquement découpés en blocs d'octets de grande taille (64 Mo: Hadoop 1.x et 128 Mo (Hadoop 2.x)) pour optimiser les temps de transfert et d'accès

  - Quand un fichier mydata.txt est enregistré dans HDFS, il est décomposé en grands blocs chaque bloc ayant un nom unique: blk_1, blk_2…





Cluster Hadoop

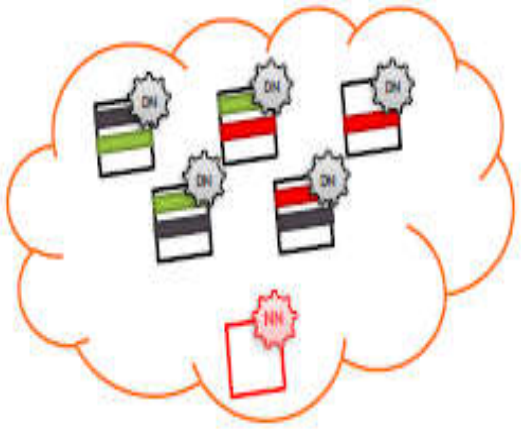
- Ces blocs sont ensuite répartis sur plusieurs machines, permettant ainsi de traiter un même fichier en parallèle.
- Cela permet aussi de ne pas être limité par la capacité de stockage d'une seule machine pour au contraire tirer parti de tout l'espace disponible du cluster de machines ;

HDFS: Hadoop Distributed File System

-Réplication des données -

Si l'un des nœuds a un problème, les données seront perdues ?

- Hadoop réplique chaque bloc 3 fois (par défaut)
- Le facteur de réplication par défaut peut être ajusté en modifiant la propriété ***dfs.replication*** dans le fichier de configuration ***hdfs-site.xml***.



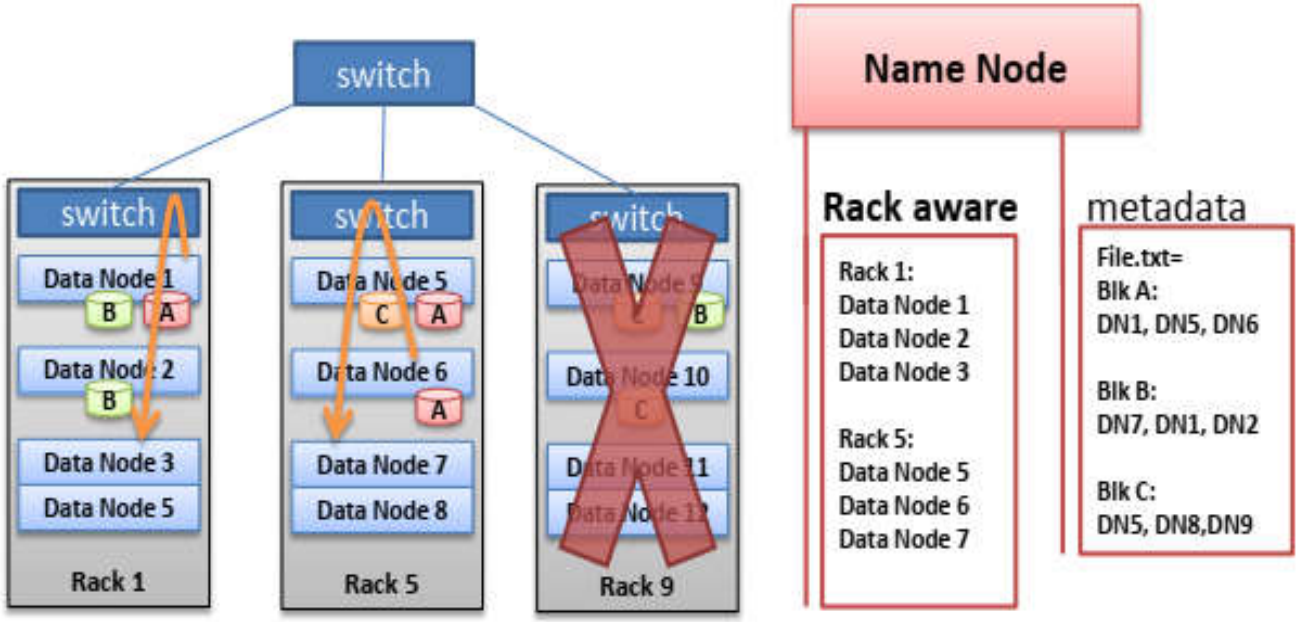
Cluster Hadoop

- Si un nœud tombe en panne, les blocs qui y étaient hébergés seront répliqués pour avoir toujours 3 copies stockées.
- Application du concept de Rack-Awareness dans la réplication

HDFS: Hadoop Distributed File System

- Principe du Rack Awarness -

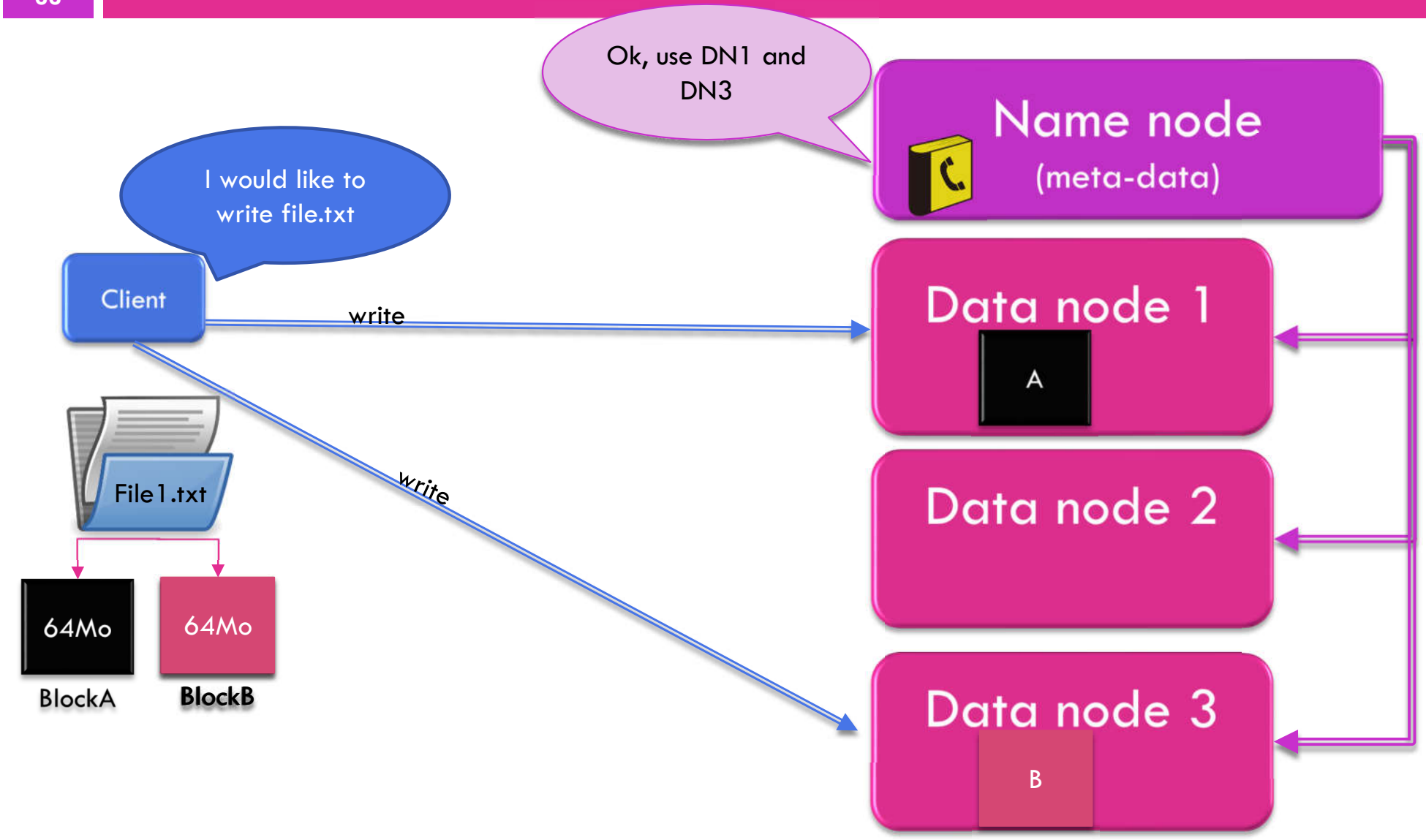
- 2 copies de chaque bloc se trouvent dans deux nœuds de données distincts du même rack afin de réduire la latence,
- La troisième copie est placée sur un autre rack pour améliorer la redondance et la disponibilité.



- Ne jamais perdre toutes les données si tout le rack tombe en panne.
- Gardez les flux volumineux dans le rack lorsque cela est possible.
- Répartition basée sur l'hypothèse que dans le rack la bande passante est meilleure et la latence est faible.

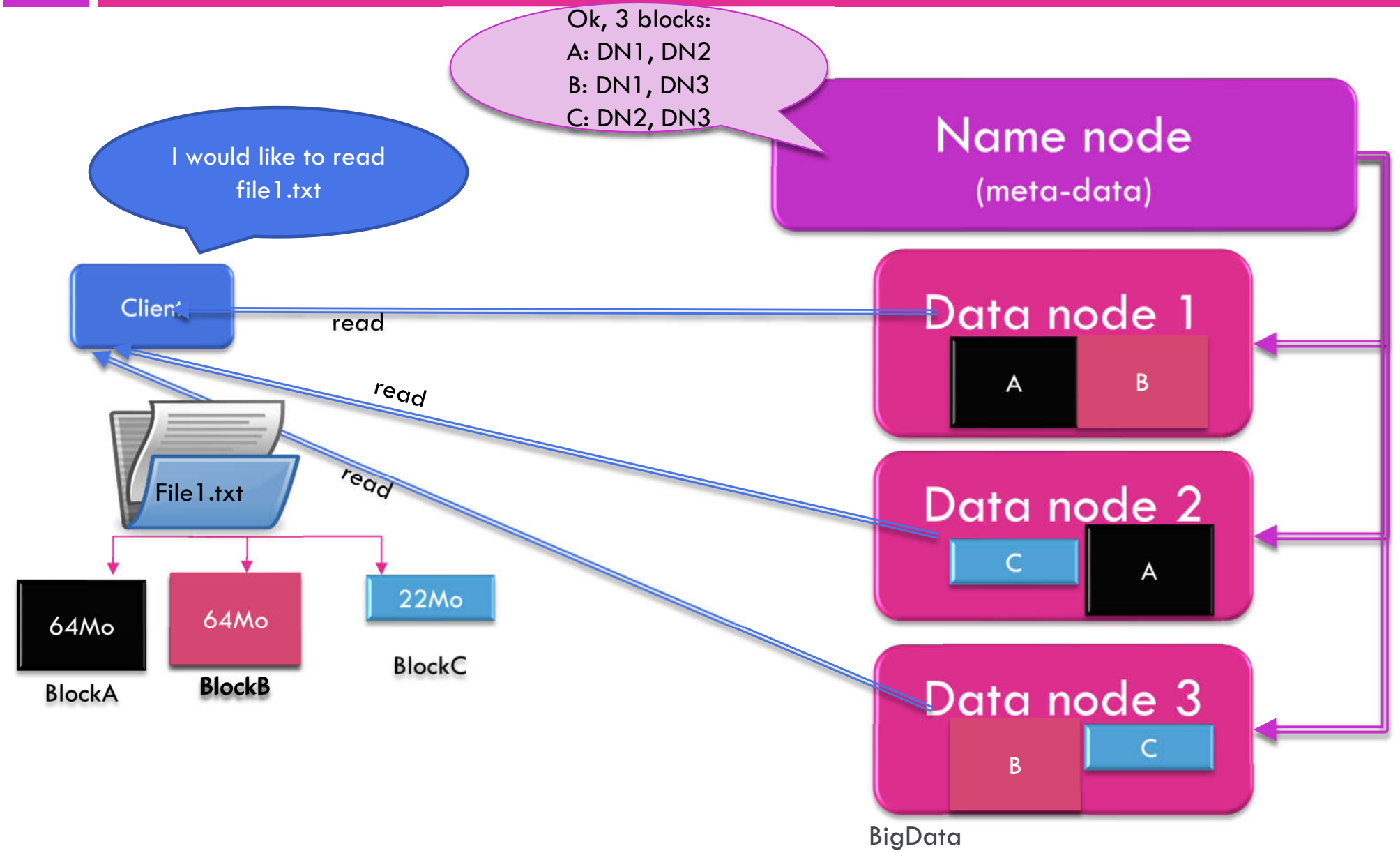
HDFS: Hadoop Distributed File System

Ecriture/Lecture d'un fichier



HDFS: Hadoop Distributed File System

Ecriture/Lecture d'un fichier



HDFS: Hadoop Distributed File System

Ecriture/Lecture d'un fichier

- HDFS suit la philosophie **Write Once – Read Many**.
- Il n'est pas possible d'éditer un fichier existant déjà dans HDFS.
- Il est possible d'ajouter en fin de fichier (append)
- HDFS s'appuie sur le système de gestion de fichier natif. Sous linux sont ext3 ou ext4 le plus souvent.

HDFS: Quelques commandes

- Il y a deux possibilités pour manipuler HDFS :
  - Soit via l'API Java
  - Soit directement depuis un terminal via les commandes

```
$ hdfs dfs <commande hdfs="" />
```

```
$ hadoop fs <commande HDFS>
```

En particulier, les commandes principales sont :

```
hdfs dfs -ls                  #listing home dir
hdfs dfs -ls /user            #listing user dir…
hdfs dfs -mv <src><dst>       #Moves files from source to destination.
hdfs dfs -du -h /user         #space used
hdfs dfs -mkdir newdir        #creating dir
hdfs dfs -put myfile.csv .    #storing a file on HDFS
hdfs dfs -get myfile.csv .    #getting a file from HDFS
hdfs dfs -cat myfile.csv .    #edit a file from HDFS
```

Documentation officielle : https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/FileSystemShell.html

Programmation MAP REDUCE

Map-Reduce et programmation fonctionnelle

Map

- Applique une fonction sur chaque élément d'une liste
- Retourne une liste de résultats
- Map(f(x),X[1:n]) à [f(X[1]),...,f(X[n])]

Exemple :

Map (X^2,[0,1,2,3,4,5])=[0,1,4,9,16,25]

Reduce/fold

- Fait l'itération d'une fonction sur une liste d'éléments
- Applique la fonction sur les résultats précédents et l'élément courant
- Retourne un et un seul résultat,

Exemple :

Reduce (**x+y**,[0,1,2,3,4,5])=(((((0+1)+2)+3)+4)+5)=15

Map-Reduce: Exemple 1

-Calcul des ventes-

Imaginons que vous ayez plusieurs magasins que vous gérez à travers le monde

Objectif

Calculer le total des ventes par magasin pour l'année 2018

Clef	Valeur
Miami	88.14
NYC	3.10

2018-01-01	Miami	Clothes	25.99
2018-01-01	Miami	Music	12.15
2018-01-02	NYC	Toys	3.10
2018-01-02	Miami	Clothes	50.00
2017-05-16	London	Toys	25.19

Map-Reduce: Exemple 1

-Calcul des ventes-

❑ Approche traditionnelle

- Si on utilise les hashtables sur 1To, Problèmes ?
  - ❑ Ça ne marchera pas ?
  - ☑ Problème de mémoire ?
  - ☑ Temps de traitement long ?
  - ❑ Réponses erronées ?
- Le traitement séquentiel de toutes les données peut s'avérer très long
- Plus on a de magasins, plus l'ajout des valeurs à la table est long
- Il est possible de tomber à court de mémoire pour enregistrer cette table
- Mais cela peut marcher, et le résultat sera correct

2018-01-01	Miami	Clothes	25.99
2018-01-01	Miami	Music	12.15
2018-01-02	NYC	Toys	3.10
2018-01-02	Miami	Clothes	50.00
2017-05-16	London	Toys	25.19

Map-Reduce: Exemple 1

-Calcul des ventes-

❑ Approche Big Data

On suppose que le livre sur HDFS est découpé en deux parties:

2018-01-01	Miami	Clothes	25.99
2018-01-01	Miami	Music	12.15
2018-01-02	NYC	Toys	3.10
2018-01-02	Miami	Clothes	50.00
2017-05-16	London	Toys	25.19

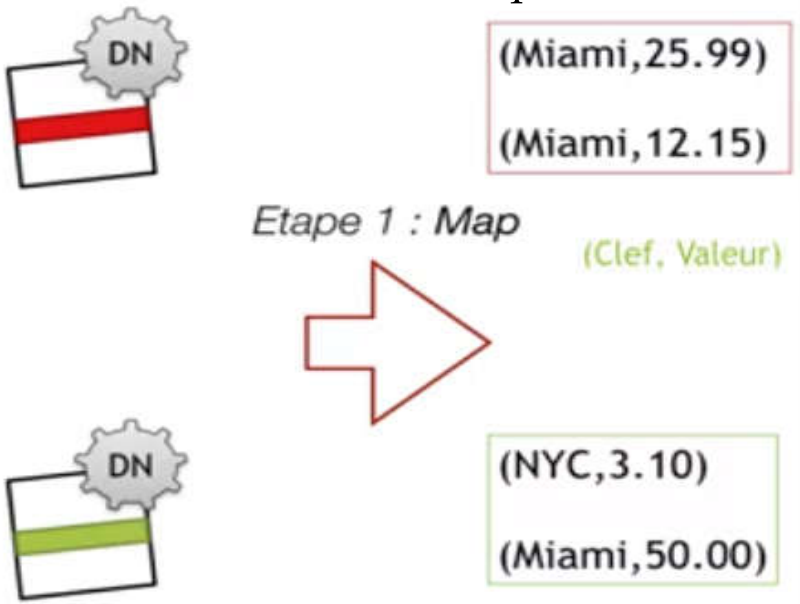
Map-Reduce: Exemple 1

-Calcul des ventes-

❑ Etape 1: MAP

- ❑ Faire des petits traitements en parallèle ligne par ligne: Pas d'opérations de calculs entre plusieurs lignes.
- ❑ 2 Types de filtrage dans cet exemple:
  - **Filtrage vertical :** diminuer le nombre de colonnes: nom magasin, valeur des ventes .
  - **Filtrage horizontal**: supprimer les lignes non concernées (année autre que 2018).

2018-01-01	Miami	Clothes	25.99
2018-01-01	Miami	Music	12.15
2018-01-02	NYC	Toys	3.10
2018-01-02	Miami	Clothes	50.00
2017-05-16	London	Toys	25.19



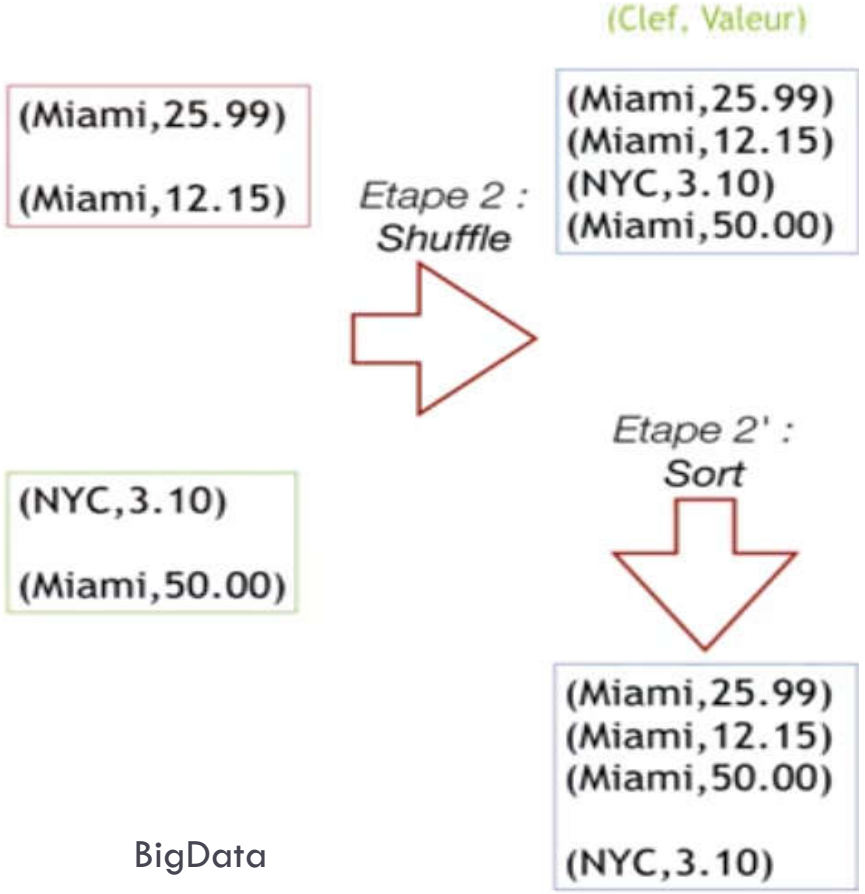
Map-Reduce: Exemple 1

-Calcul des ventes-

❑ Etape 2: Sort and Shuffle

- ❑ Etape faite automatiquement par Hadoop, composée de deux sous étapes:
  - **Shuffle:** Rassemblement des données dans la même machine
  - **Sort:** Tri par clé pour regrouper les ventes de même magasin successivement

2018-01-01	Miami	Clothes	25.99
2018-01-01	Miami	Music	12.15
2018-01-02	NYC	Toys	3.10
2018-01-02	Miami	Clothes	50.00
2017-05-16	London	Toys	25.19



Map-Reduce: Exemple 1

-Calcul des ventes-

❑ Etape 3: Reduce

- ❑ Comme le mapper, le code de reducer doit être écrit par l'utilisateur
- ❑ Ici; faire la somme des ventes par magasin.

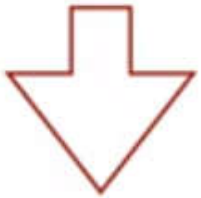
2018-01-01	Miami	Clothes	25.99
2018-01-01	Miami	Music	12.15
2018-01-02	NYC	Toys	3.10
2018-01-02	Miami	Clothes	50.00
2017-05-16	London	Toys	25.19

(Clef, Valeur)

(Miami,25.99)
(Miami,12.15)
(Miami,50.00)

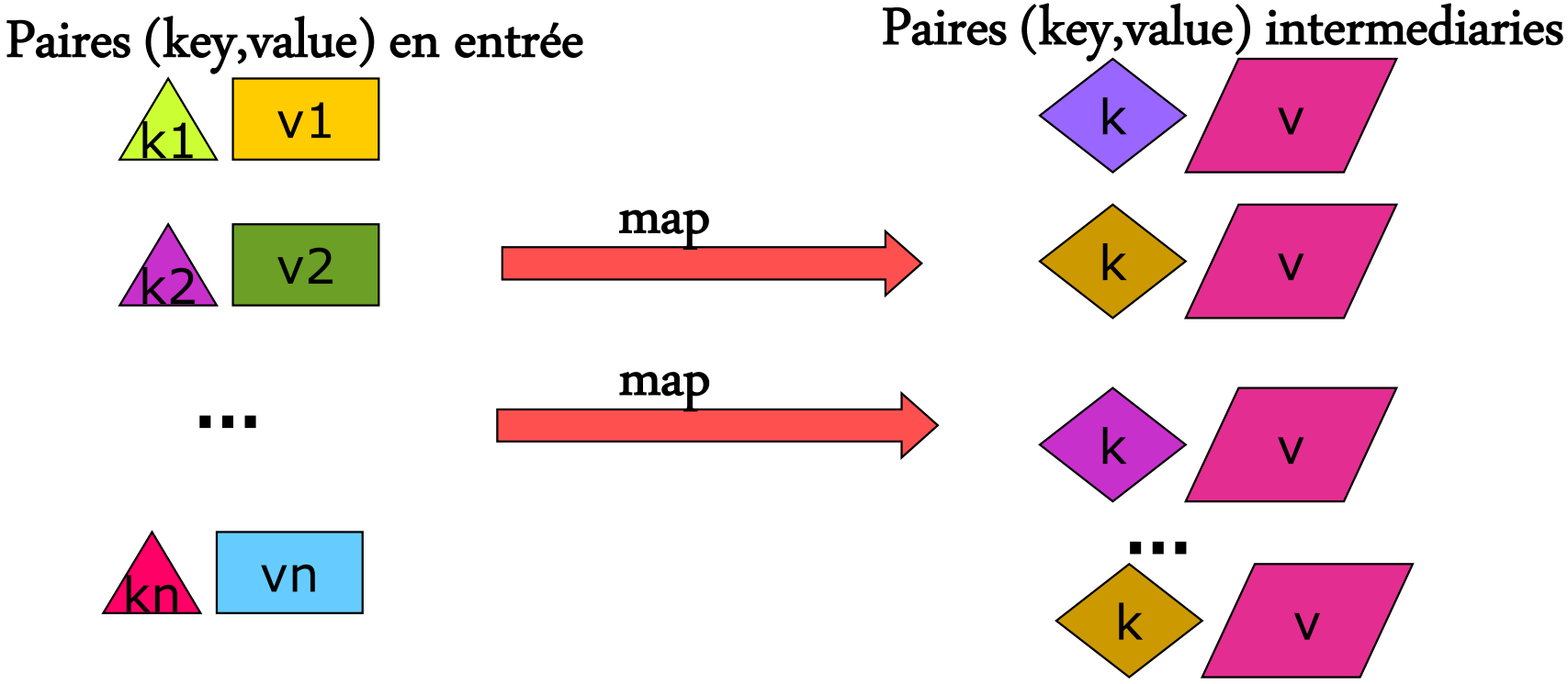
(NYC,3.10)

Etape 3 : Reduce



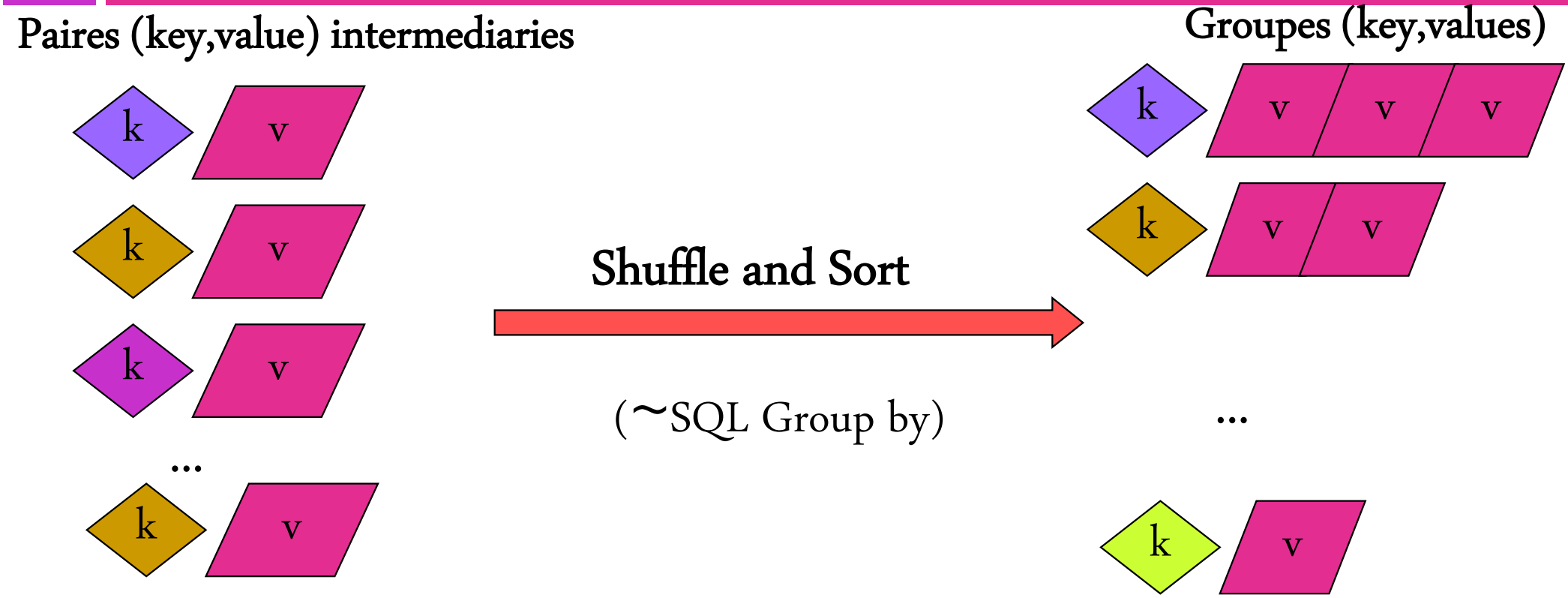
(Miami , 88.14)
(NYC , 3.10)

A retenir: L'étape MAP



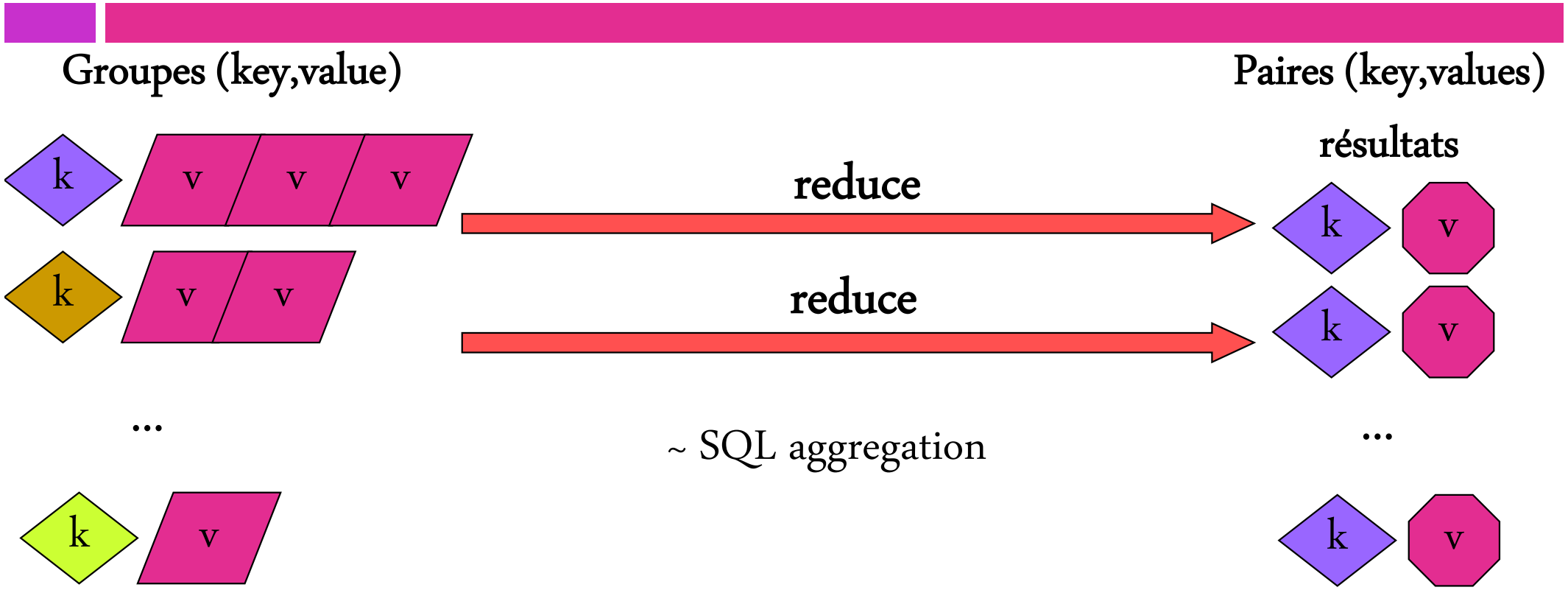
- Dans l'étape **Map,** le but est de partir d'un couple <clé, valeur> et d'y associé de nouveaux couples <clé, valeur>. Les clés en entrée sont différentes des clés produites par le Map.
- Le nombre de tâches Map ne dépend pas du nombre de nœuds, mais du nombre de blocs de données en entrée. Chaque bloc se fait assigner une seule tâche Map.

A retenir : L'étape Shuffle and Sort



- C'est un étape **automatique de regroupement et tri** s'intègre entre MAP et REDUCE pour la redistribution des données afin que les paires produites par Map ayant les mêmes clés soient sur les mêmes machines.

A retenir : L'étape REDUCE

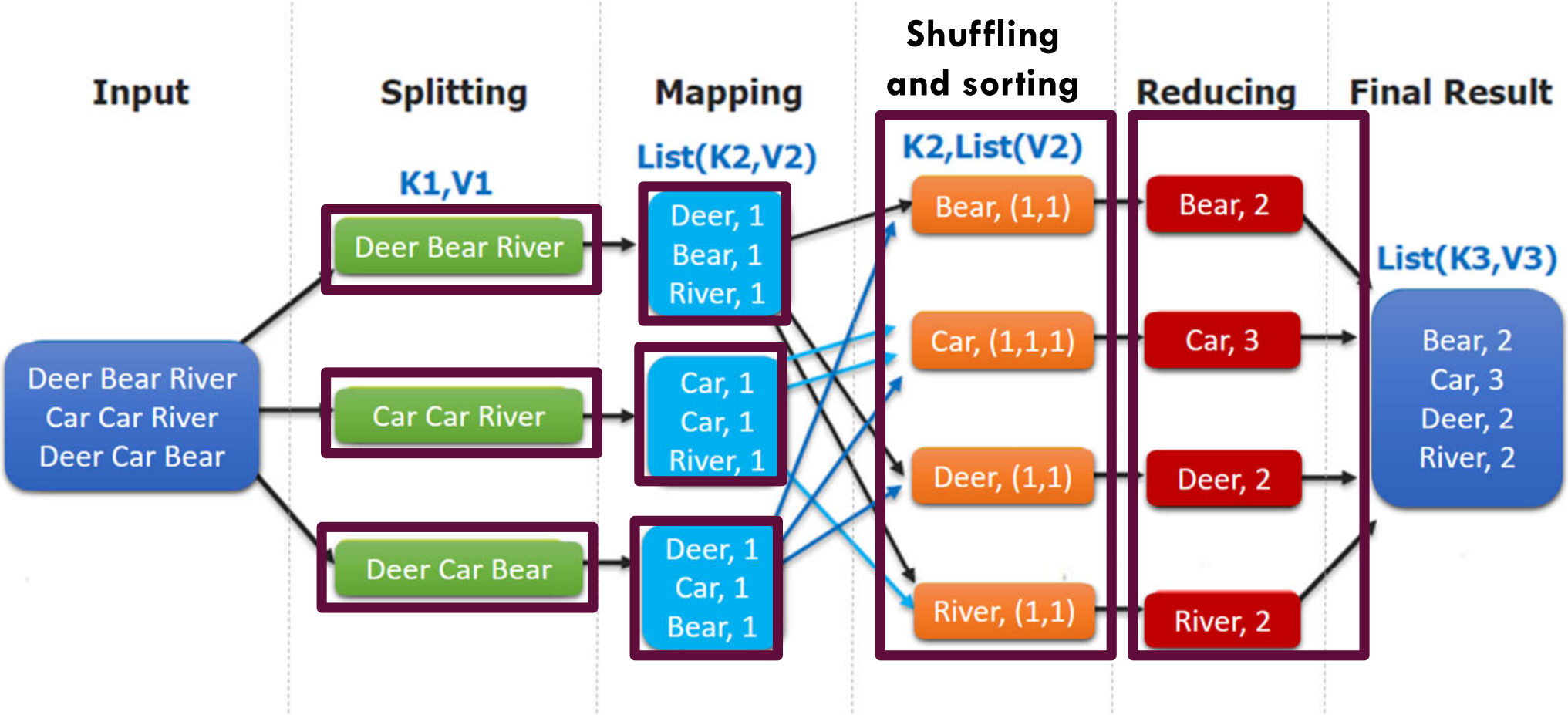


- Dans l'étape Reduce le but est d'associer toutes les valeurs correspondantes à la même clé. On souhaite donc rassembler tous les couples <clé, valeur>.
- Pour le traitement, les taches Reduce suivent le même schéma que les taches Map.
- Elles n'ont pas à s'exécuter parallèlement et dès qu'un nœud fini son traitement un autre lui est aussitôt assigné.

Map-Reduce: Exemple 2

-Word Count-

- Word Count: Comptage des occurrences des mots



Map-Reduce: Exemple 2

-Word Count-

☐ Pseudocodes du map et du reduce

```
map(key, text): # input: key=position, text=line
  for each word in text:
    Emit(word,1) # outputs: key/value

reduce(key, list of values): # input: key == word, our mapper output
  count = 0
  for each v in values:
    count += v
  Emit(key, count) # it is possible to emit multiple (key, value) pairs here
```

Map-Reduce: Exemple 3

-Graphe Social-

- On administre un réseau social comportant des millions d'utilisateurs.
- Pour chaque utilisateur, on a dans notre BD la liste des utilisateurs qui sont ses amis sur le réseau (via une requête SQL).
- On souhaite afficher quand un utilisateur va sur la page d'un autre utilisateur une indication *Vous avez N amis en commun* ;
- On ne peut pas se permettre d'effectuer une série de requêtes SQL à chaque fois que la page est accédée (trop lourd en traitement).

⇒ On va donc développer des programmes Map et Reduce pour cette opération et exécuter le traitement toutes les nuits sur notre BD, en stockant le résultat dans une nouvelle table.

Map-Reduce: Exemple 3

-Graphe Social-

Ici, nos données d'entrée sont sous la forme Utilisateur ⇒ Amis :

$A \Rightarrow B, C, D$
$B \Rightarrow A, C, D, E$
$C \Rightarrow A, B, D, E$
$D \Rightarrow A, B, C, E$
$E \Rightarrow B, C, D$

- Puisqu'on est intéressé par l'information *amis en commun entre deux utilisateurs* et qu'on aura à terme une valeur par clé, on va choisir pour clé la concaténation entre deux utilisateurs.
- Par exemple, la clé $A-B$ désignera *les amis en communs des utilisateurs A et B*.
- On peut segmenter les données d'entrée là aussi par ligne.

Map-Reduce: Exemple 3

-Graphe Social-

La phase MAP

- Notre opération Map va se contenter de prendre la liste des amis fournie en entrée, et va générer toutes les clés distinctes possibles à partir de cette liste.
- La valeur sera simplement la liste d'amis, telle quelle.
- On fait également en sorte que la clé soit toujours triée par ordre alphabétique (clé $B - A$ sera exprimée sous la forme $A - B$).
- Ce traitement peut paraître contre-intuitif, mais il va à terme nous permettre d'obtenir, pour chaque clé distincte, deux couples (key, value) : les deux listes d'amis de chacun des utilisateurs qui composent la clé.

Map-Reduce: Exemple 3

-Graphe Social-

La phase MAP

Le pseudo code de notre opération Map est le suivant :

```
UTILISATEUR = [PREMIERE PARTIE DE LA LIGNE]
POUR AMI dans [RESTE DE LA LIGNE], FAIRE:
  SI UTILISATEUR < AMI:
    CLEF = UTILISATEUR+    +AMI
  SINON:
    CLEF = AMI+    +UTILISATEUR
  GENERER COUPLE (CLEF; [RESTE DE LA LIGNE])
```

Par exemple, pour la première ligne :

```
A => B, C, D
```

On obtiendra les couples (key, value) :

```
("A-B"; "B C D")   ("A-C"; "B C D")   ("A-D"; "B C D")
```

Map-Reduce: Exemple 3

-Graphe Social-

La phase MAP

Pour la seconde ligne :

```
B => A, C, D, E
```

On obtiendra ainsi :

```
("A-B"; "A C D E")  ("B-C"; "A C D E")  ("B-D"; "A C D E")
("B-E"; "A C D E")
```

Pour la troisième ligne :

```
C => A, B, D, E
```

On aura :

```
("A-C"; "A B D E")  ("B-C"; "A B D E")  ("C-D"; "A B D E")
("C-E"; "A B D E")
```

...et ainsi de suite pour nos 5 lignes d'entrée

Map-Reduce: Exemple 3

-Graphe Social-

Entre la phase MAP et la phase REDUCE

Une fois l'opération Map effectuée, Hadoop va récupérer les couples (key, valeur) de tous les fragments et les grouper par clé distincte. Le résultat sur la base de nos données d'entrée :

```
Pour la clef "A-B": valeurs "A C D E" et "B C D"
Pour la clef "A-C": valeurs "A B D E" et "B C D"
Pour la clef "A-D": valeurs "A B C E" et "B C D"
Pour la clef "B-C": valeurs "A B D E" et "A C D E"
Pour la clef "B-D": valeurs "A B C E" et "A C D E"
Pour la clef "B-E": valeurs "A C D E" et "B C D"
Pour la clef "C-D": valeurs "A B C E" et "A B D E"
Pour la clef "C-E": valeurs "A B D E" et "B C D"
Pour la clef "D-E": valeurs "A B C E" et "B C D"
```

...on obtient bien, pour chaque clé *USER*1 – *USER*2, deux listes d'amis : les amis de *USER*1 et ceux de *USER*2.

Map-Reduce: Exemple 3

-Graphe Social-

La phase REDUCE

Il nous faut enfin écrire notre programme Reduce. Il va recevoir en entrée toutes les valeurs associées à une clé. Son rôle va être très simple : déterminer quels sont les amis qui apparaissent dans les listes (les valeurs) qui nous sont fournies.

```
LISTE_AMIS_COMMUNS=[]    // Liste vide au départ.
SI LONGUEUR(VALEURS)!=2, ALORS:  // Ne devrait pas se produire.
  RENVOYER ERREUR
SINON:
  POUR AMI DANS VALEURS[0], FAIRE:
    SI AMI EGALEMENT PRESENT DANS VALEURS[1], ALORS:
      // Présent dans les deux listes d'amis, on l'ajoute.
      LISTE_AMIS_COMMUNS+=AMI
RENVOYER LISTE_AMIS_COMMUNS
```

Map-Reduce: Exemple 3

-Graphe Social-

La phase REDUCE

Après exécution de l'opération Reduce pour les valeurs de chaque clé unique, on obtiendra donc, pour une clé $A - B$, les utilisateurs qui apparaissent dans la liste des amis de A et dans la liste des amis de B. Autrement dit, on obtiendra la liste des amis en commun des utilisateurs A et B. Le résultat est :

```
"A-B": "C, D"
"A-C": "B, D"
"A-D": "B, C"
"B-C": "A, D, E"
"B-D": "A, C, E"
"B-E": "C, D"
"C-D": "A, B, E"
"C-E": "B, D"
"D-E": "B, C"
```

On sait ainsi que A et B ont pour amis communs les utilisateurs C et D, ou encore que B et C ont pour amis communs les utilisateurs A, D et E.

Map-Reduce: Exemple 3

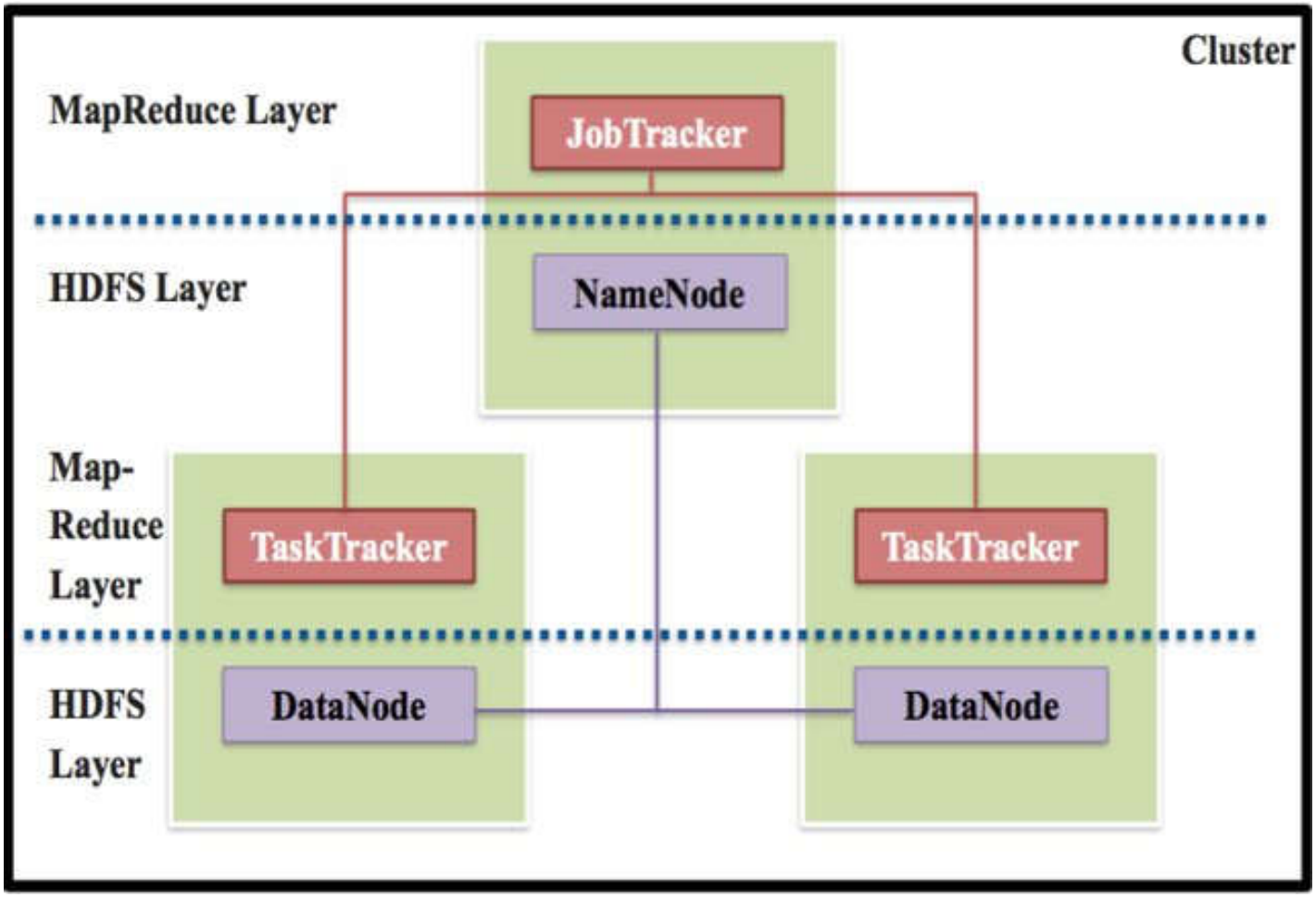
-Graphe Social-

Synthèse

- En utilisant le modèle MapReduce, on a ainsi pu créer deux programmes très simples (nos programmes Map et Reduce) de quelques lignes de code seulement, qui permettent d'effectuer un traitement somme toute assez complexe.
- Mieux encore, notre traitement est parallélisable : même avec des dizaines de millions d'utilisateurs, du moment qu'on a assez de machines au sein du cluster Hadoop, le traitement sera effectué rapidement. Pour aller plus vite, il nous suffit de rajouter plus de machines.
- Pour notre réseau social, il suffira d'effectuer ce traitement toutes les nuits à heure fixe, et de stocker les résultats dans une table.
  Ainsi, lorsqu'un utilisateur visitera la page d'un autre utilisateur, un seul SELECT dans la BD suffira pour obtenir la liste des amis en commun - avec un poids en traitement très faible pour le serveur.

Fonctionnement de Map Reduce

Hadoop MapReduce: Fonctionnement



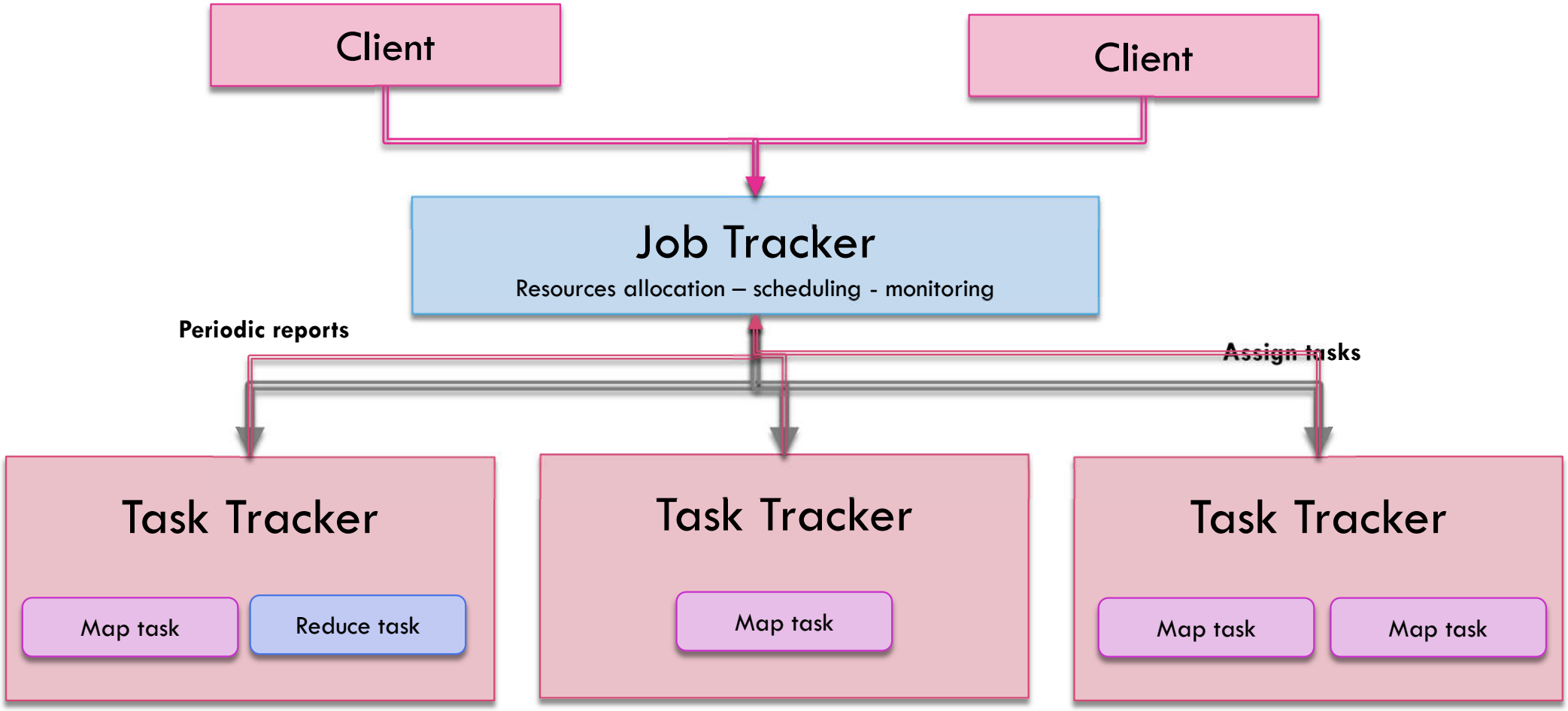
Hadoop MapReduce: Fonctionnement

- Deux démons: Job Tracker et Task Tacker
- Job Tracker
  - S'exécute sur la même machine que le Namenode (machine master)
  - Divise le travail sur les Mappers et Reducers s'exécutant sur les différents nœuds.
- Task Tacker
  - S'exécute sur chacun des nœuds pour exécuter les vraies tâches de Map-Reduce.
  - Choisit de traiter (map et reduce) un bloc sur la même machine que lui affecté
  - S'il est déjà occupé, la tâche revient à un autre tracker qui utilisera le réseau (rare).

Hadoop MapReduce: Fonctionnement

- Un job Map-Reduce est divisé en plusieurs tâches **mappers** et **reducers**
- Chaque tâche est exécutée sur un nœud 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 Map-Reduce
  - De réserver et ordonnancer les slots, et de gérer les fautes en réallouant les slots au besoin.

Hadoop MapReduce: Fonctionnement



Gestion de Ressources d'Hadoop

Hadoop MapReduce: Limites

- Le **Job Tracker** s'exécute sur une seule machine, et fait plusieurs tâches (gestion de ressources, ordonnancement et monitoring des tâches…);
  - Problème de scalabilité et de goulot d'étranglement : les nombreux datanodes existants ne sont pas exploités**:** max de 5000 noeuds et 40,000 tasks s'executant simultanément (Yahoo).
- Si le **Job Tracker** tombe en panne, tous les jobs doivent redémarrer;
  - Problème de disponibilité: SPoF (*single point of failure)*
- Le nombre de map slots et de reduce slots est prédéfini:
  - Problème d'exploitation: si on a plusieurs map jobs à exécuter, et que les map slots sont pleins, les reduce slots ne peuvent pas être utilisés, et vice-versa.
- Le **Job Tracker** est fortement intégré à Map Reduce!
  - Problème d'interoperabilité: impossible d'exécuter des applications non-MR sur HDFS

Solution: Hadoop 2.0

Hadoop v1.0

MapReduce
Data Processing
& Resource Management

HDFS
Distributed File Storage

Hadoop v2.0

MapReduce

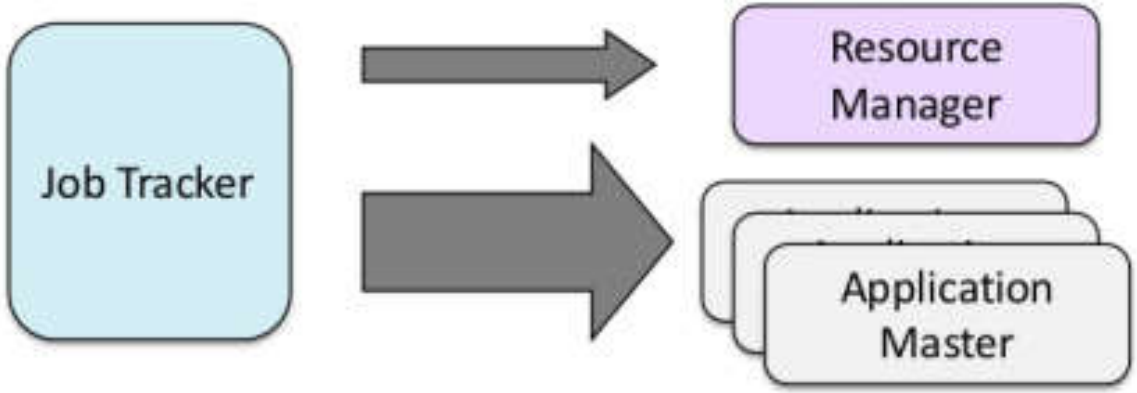
Other Data Processing Frameworks

YARN
Resource Management

HDFS
Distributed File Storage

YARN: C'est quoi?

- **Yarn:** *Yet Another Resource Negociator.*
- Un framework permettant d'exécuter n'importe quel type d'application distribuée sur un cluster Hadoop, pas uniquement les applications MapReduce.
- Pas de notion de slots: Les nœuds ont des ressources (CPU, mémoire..) allouées aux applications à la demande.
- **Idée clé:** Séparer la gestion des ressources de celle des tâches MR.



YARN- Les démons

Resource Manager (RM)

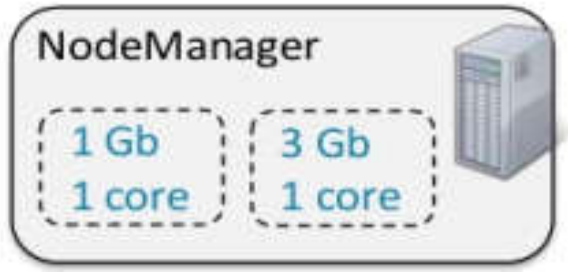
- Tourne sur un nœud master
- Ordonnanceur global des ressources
- Permet l'arbitrage des ressources entre plusieurs applications.

Node Manager (NM)

- S'exécute sur les nœuds slaves
- Communique avec RM

YARN- Les démons

Containers

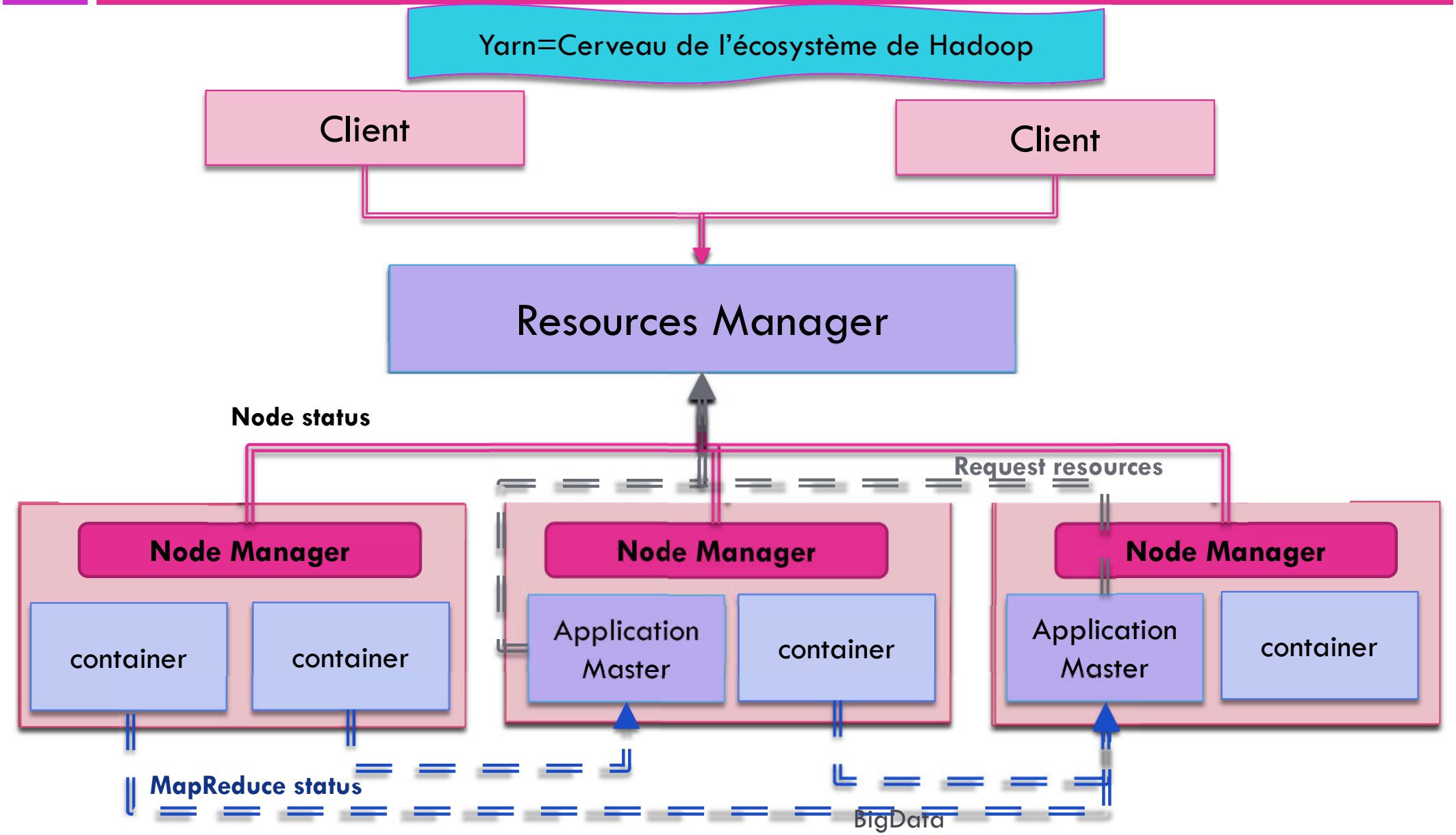


- Crées par le RM à la demande
- Se voit allouer des ressources (mémoire, CPU) sur le noeud slave: Il n'y a plus de slots prédéfinis.
- Les applications tournent sur un ou plusieurs containers.

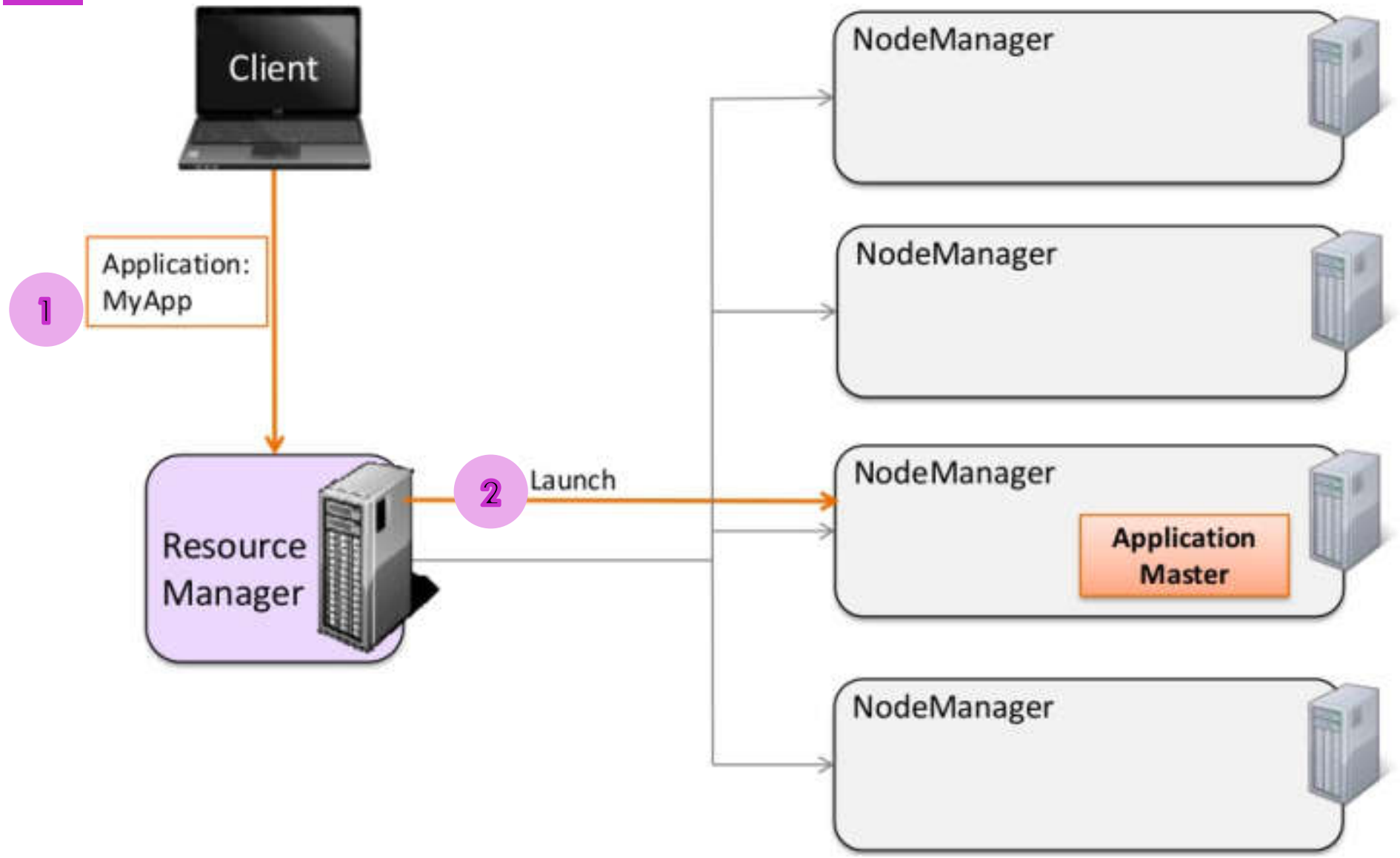
Application Master (AM)

- Un seul par application
- S'exécute sur un container
- Demande plusieurs containers pour exécuter les tâches d'applications.

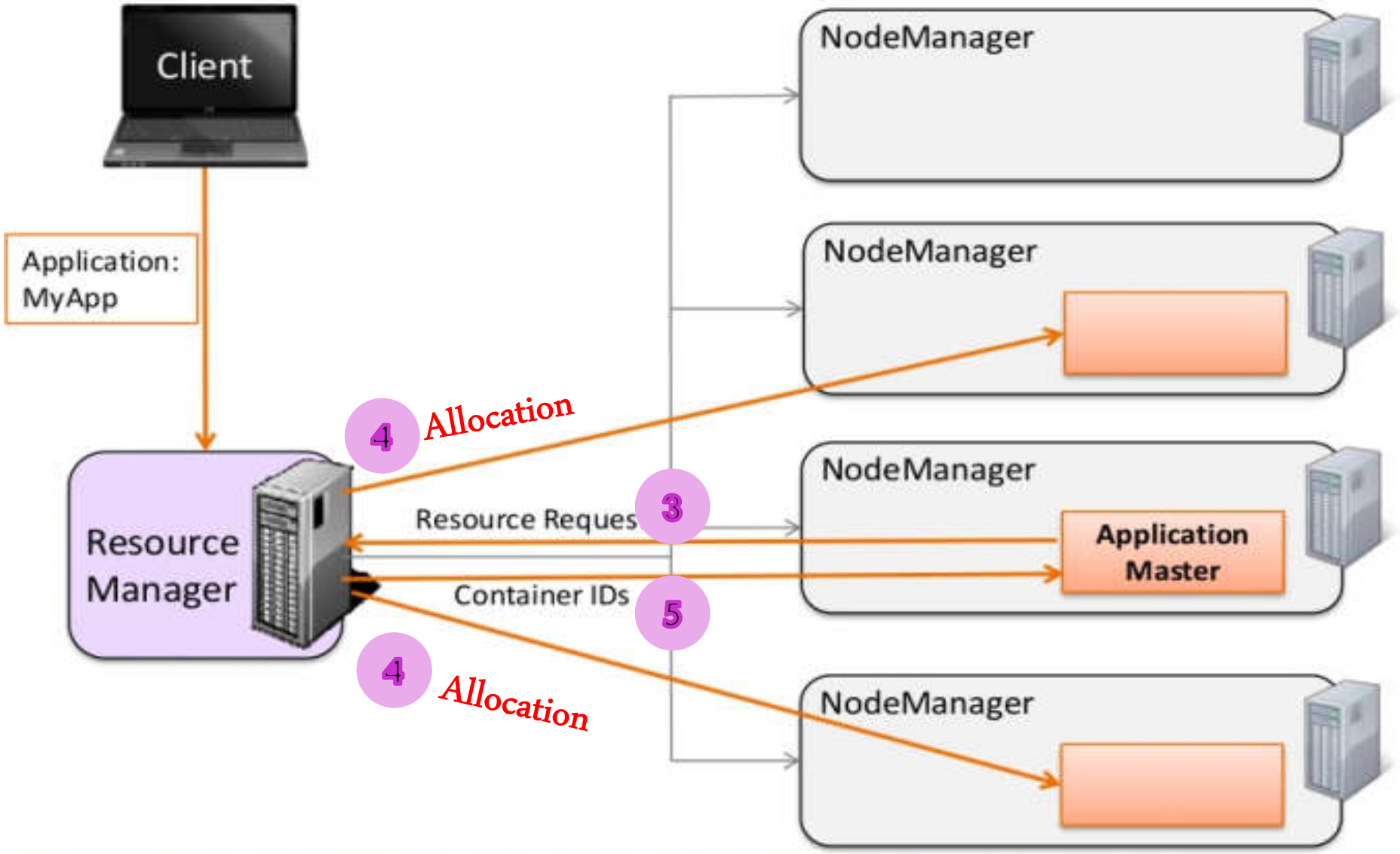
YARN- Fonctionnement



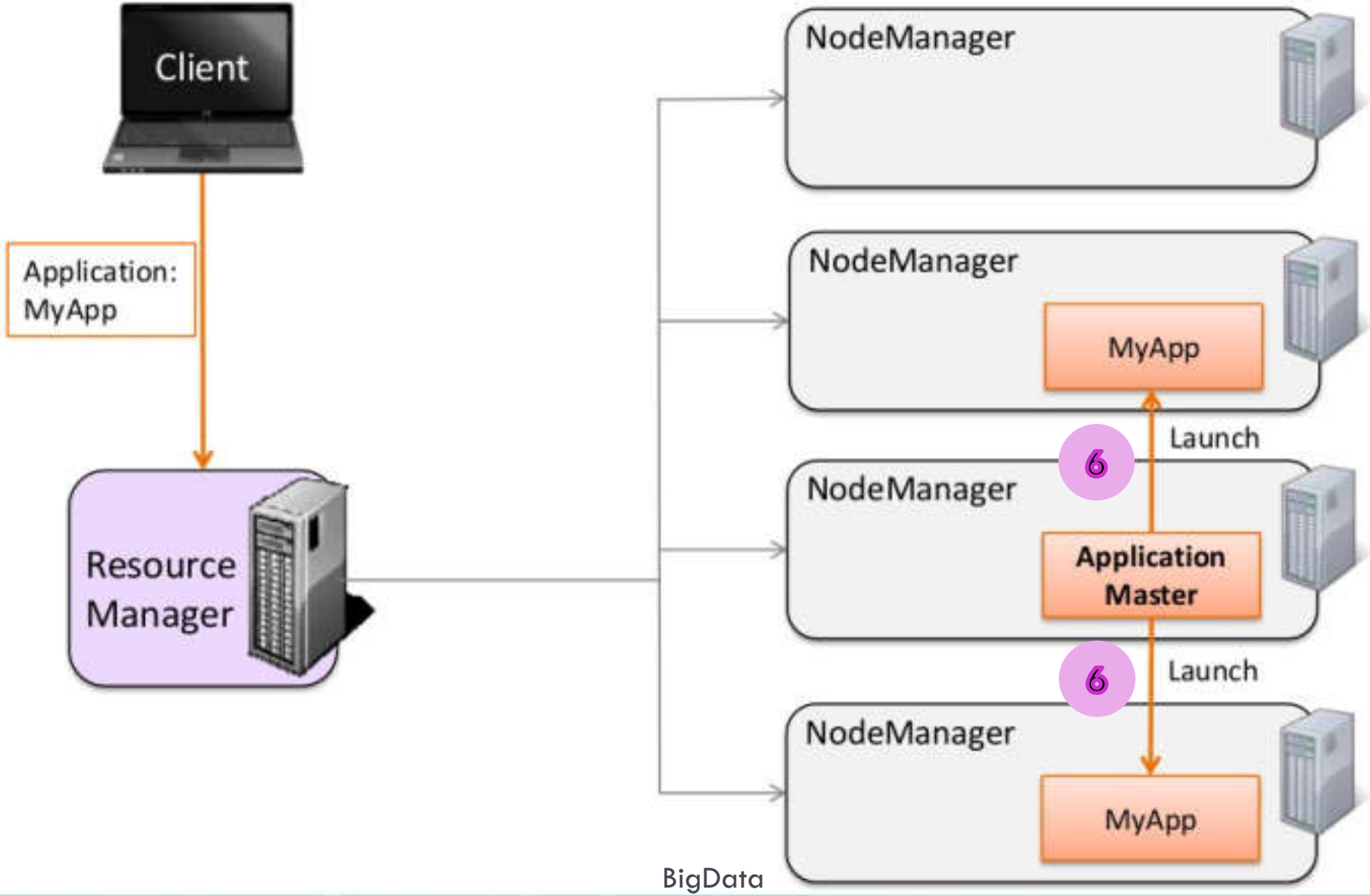
Cluster YARN: Exécution d'une application



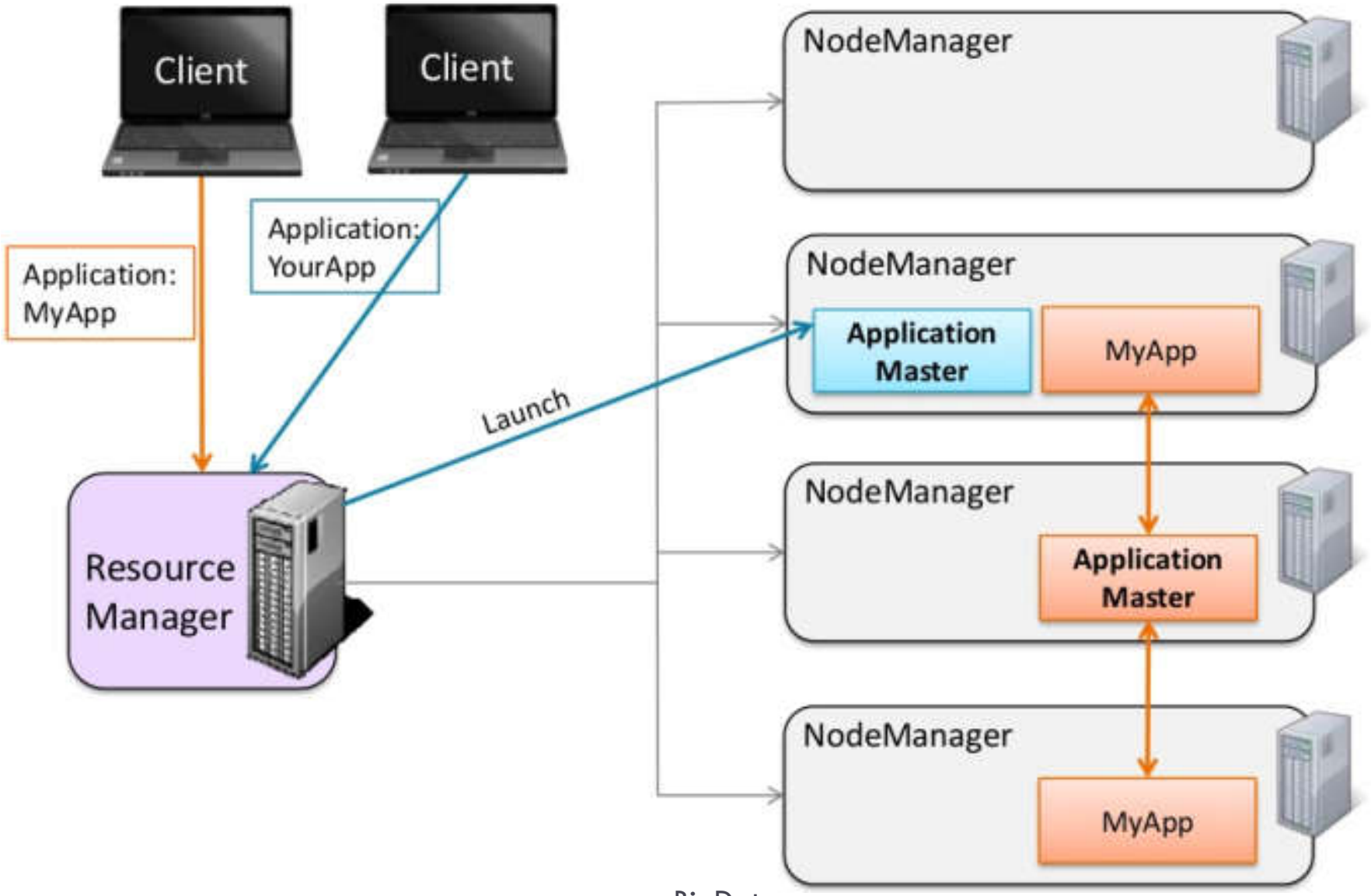
Cluster YARN: Exécution d'une application



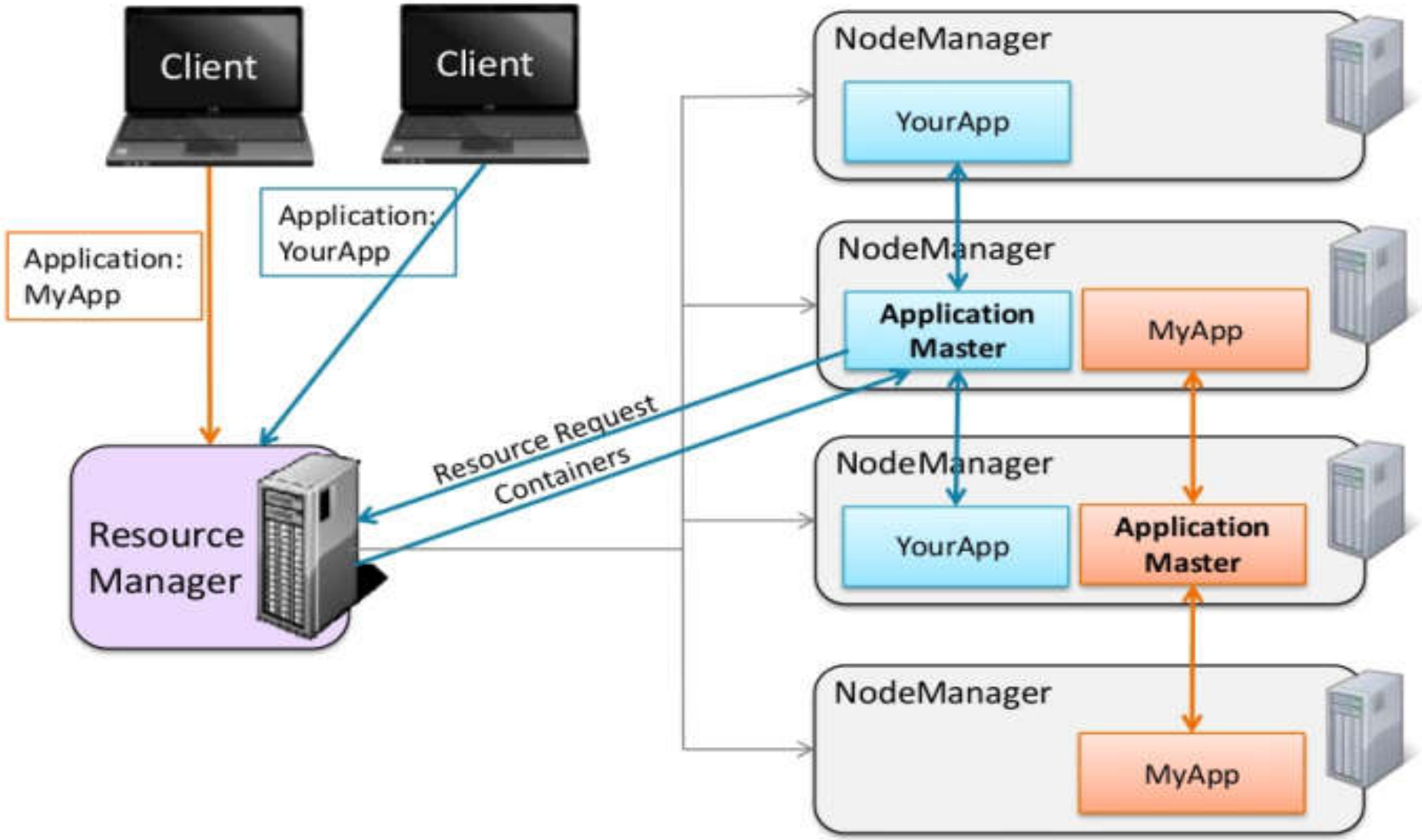
Cluster YARN: Exécution d'une application



Cluster YARN: Exécution d'une application



Cluster YARN: Exécution d'une application



YARN et Map-Reduce v2

- Yarn ne sait pas quel type d'application s'exécute: peut être MR ou autres applications non-MR comme:
  - Distributed Shell
  - Impala
  - Apache Giraph
  - Spark
  - Autres : https://cwiki.apache.org/confluence/display/HADOOP2/PoweredByYarn

Quelques applications pouvant s'exécuter de manière native sur Hadoop (Source : HortonWorks)

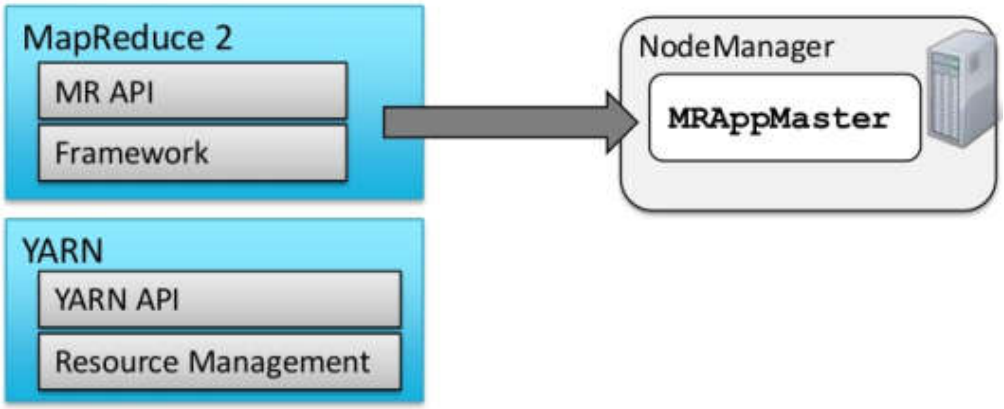
YARN et Map-Reduce v2

MR2: C'est quoi?

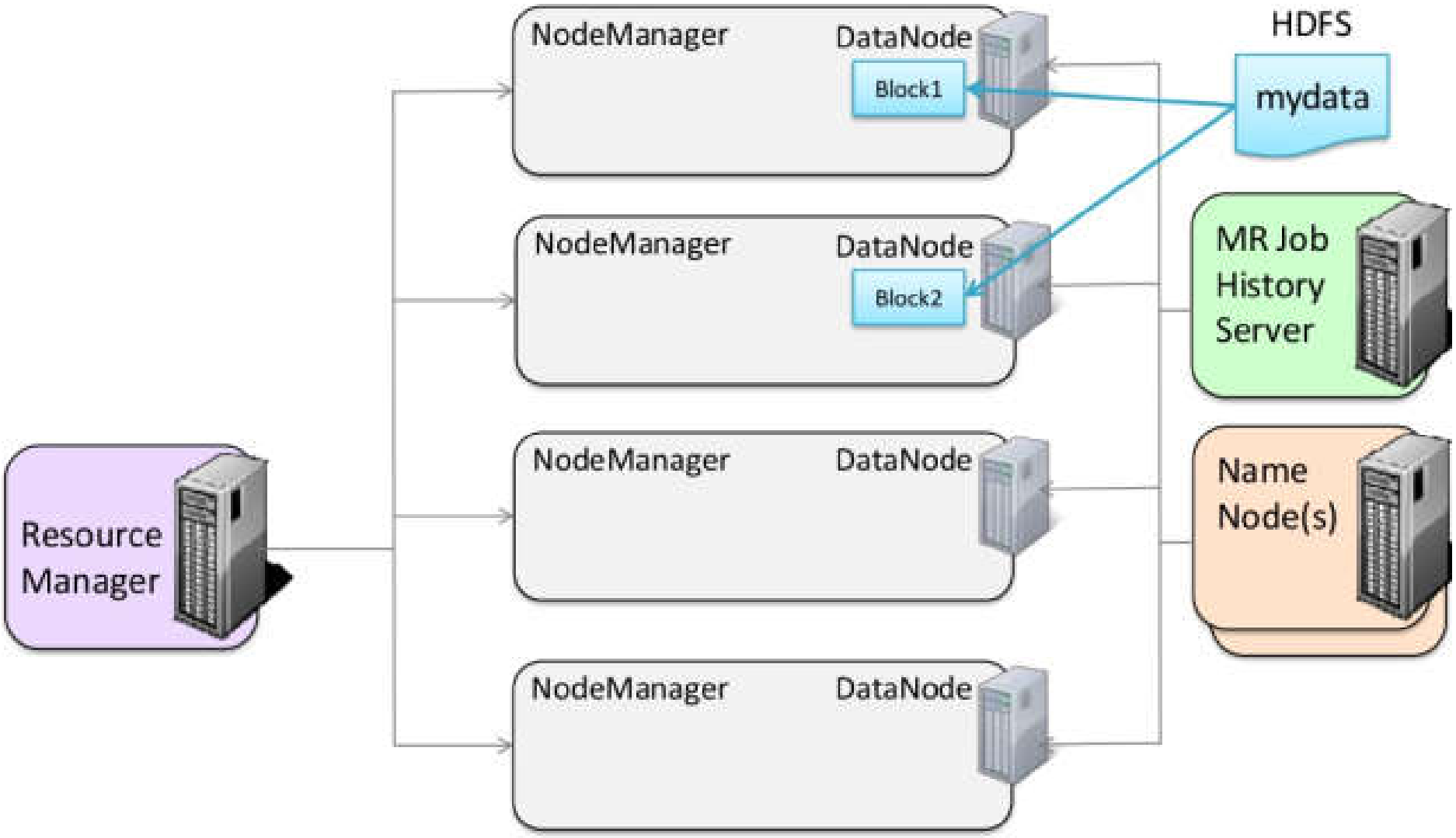
- Avec Yarn, il n'y a plus de l'unique JobTracker pour exécuter les jobs et des taskTracker pour exécuter les tasks des jobs.
- L'ancienne version MRv1 a été écrite pour s'adapter avec YARN et a été appelée MRv2.

MR2 utilise Hadoop:

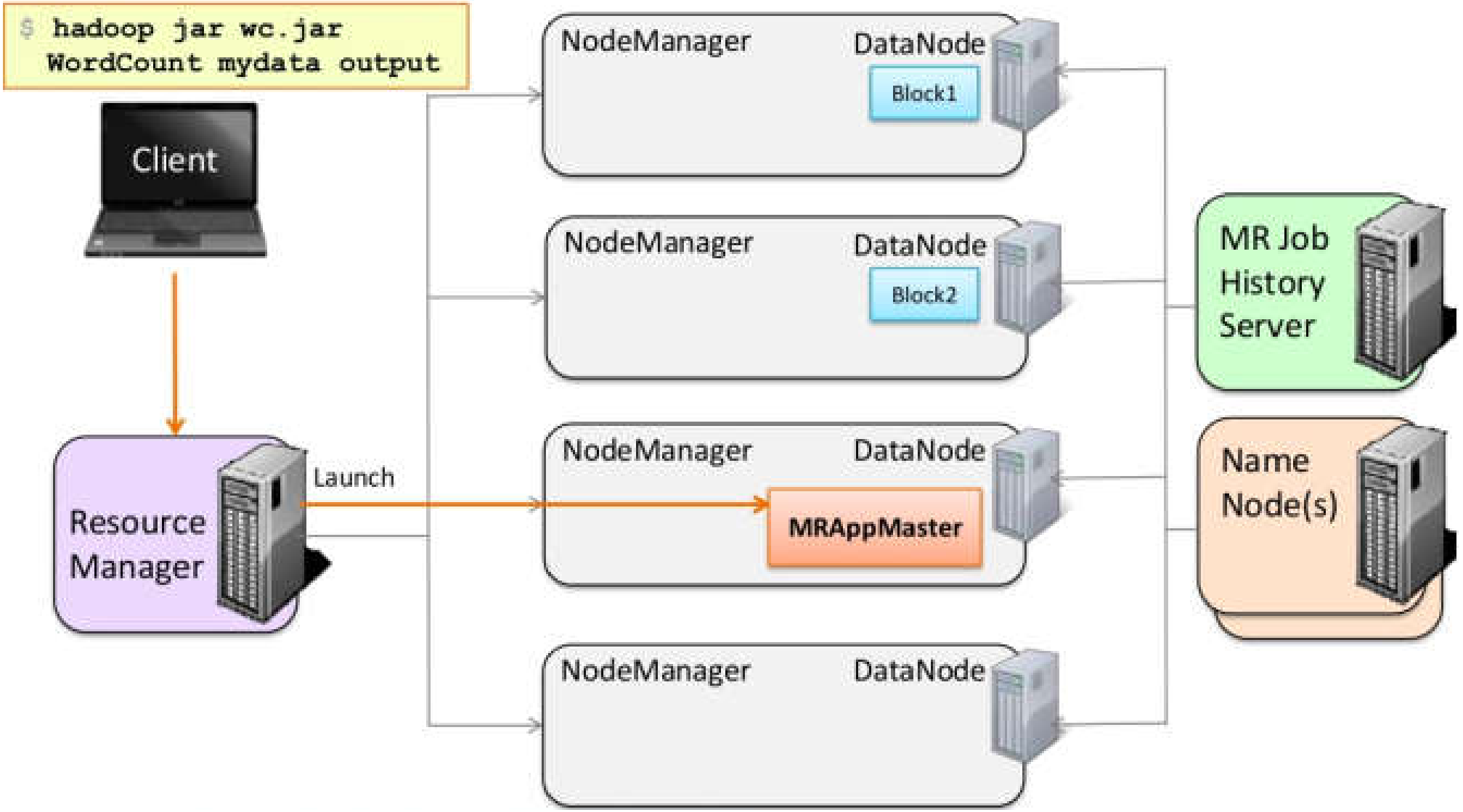
- Hadoop inclus un MRAppMaster (MapReduce ApplicationMaster) pour gérer les jobs MR.



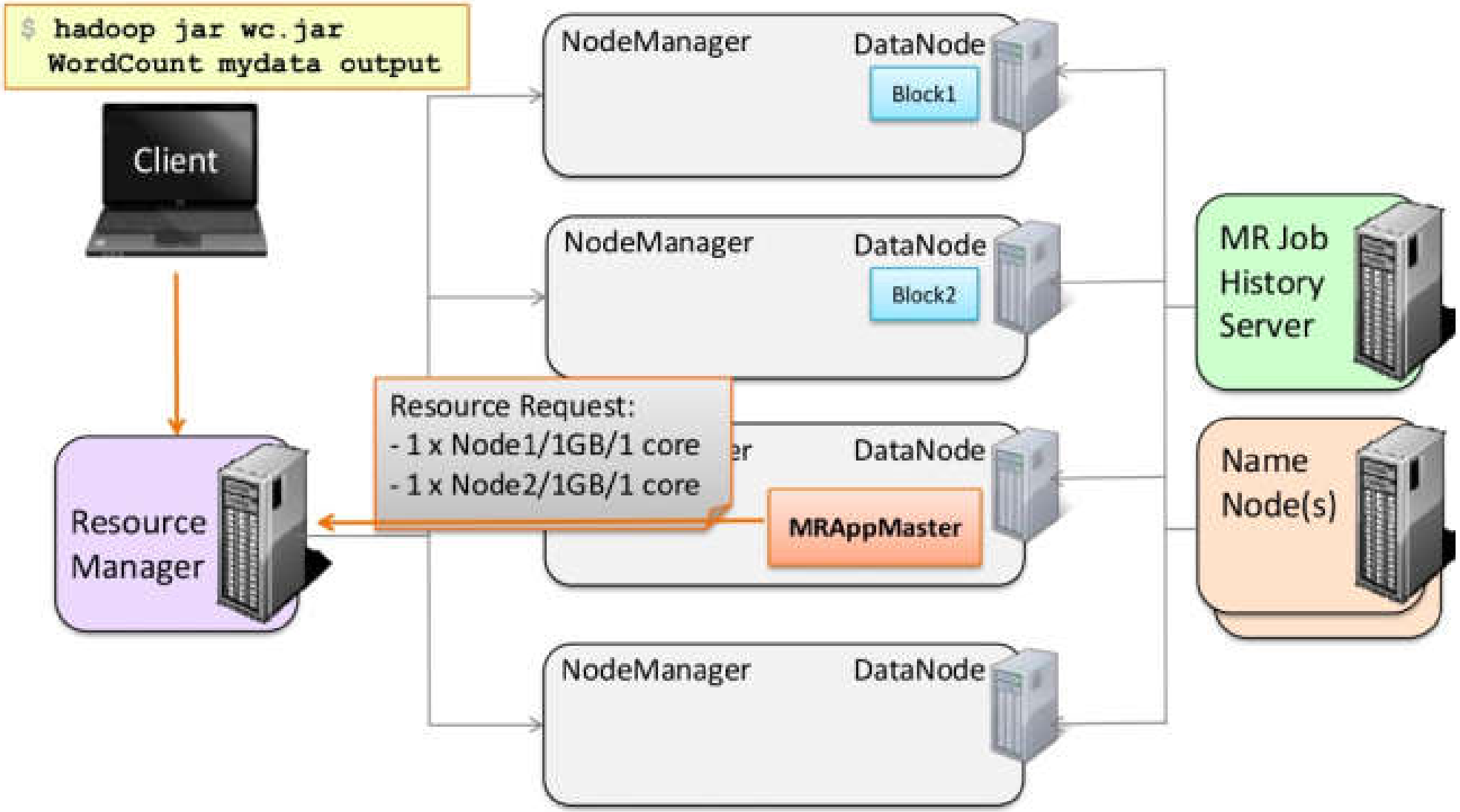
Cluster YARN: Exécution d'un job Map-Reduce



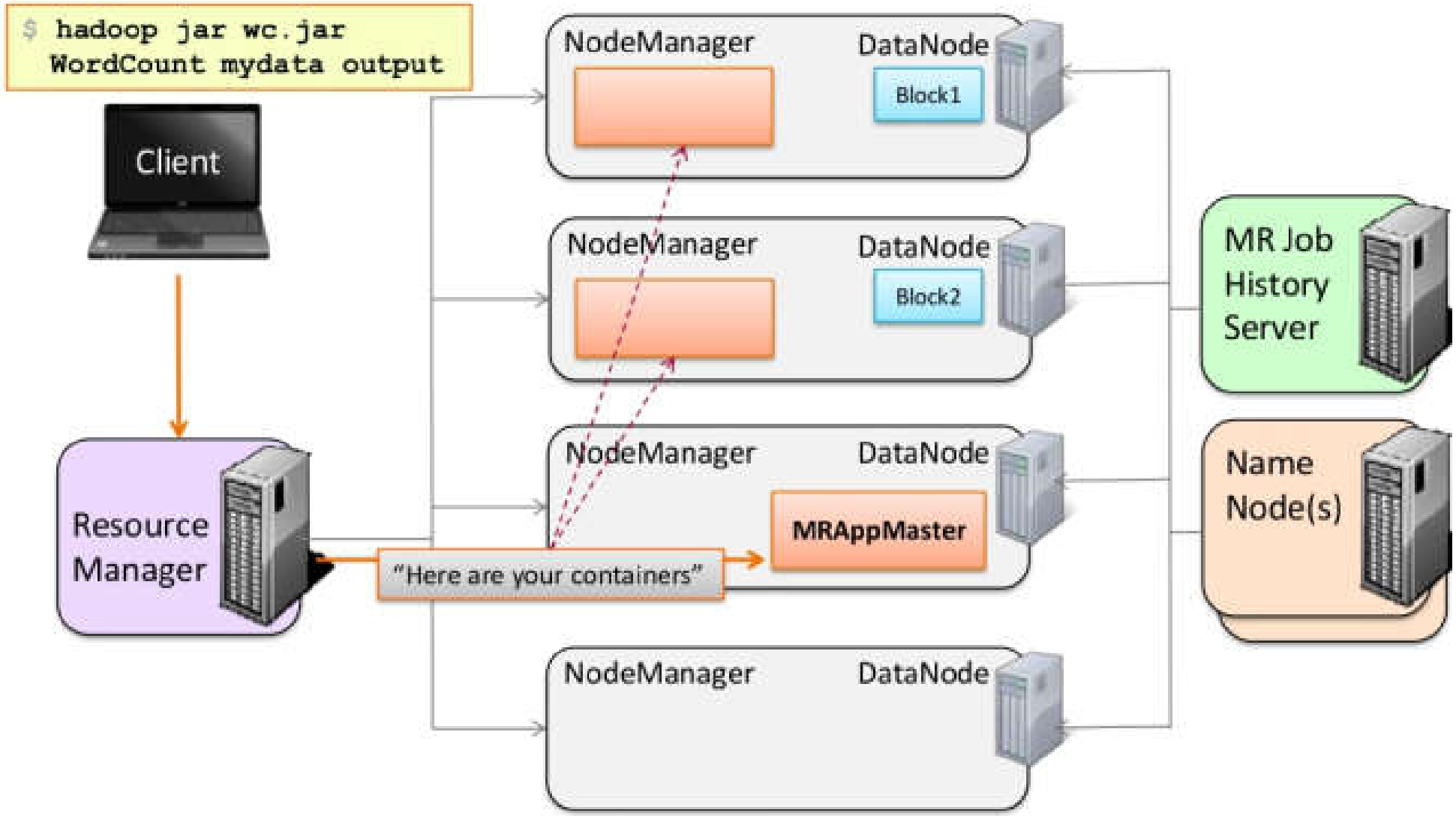
Cluster YARN: Exécution d'un job Map-Reduce



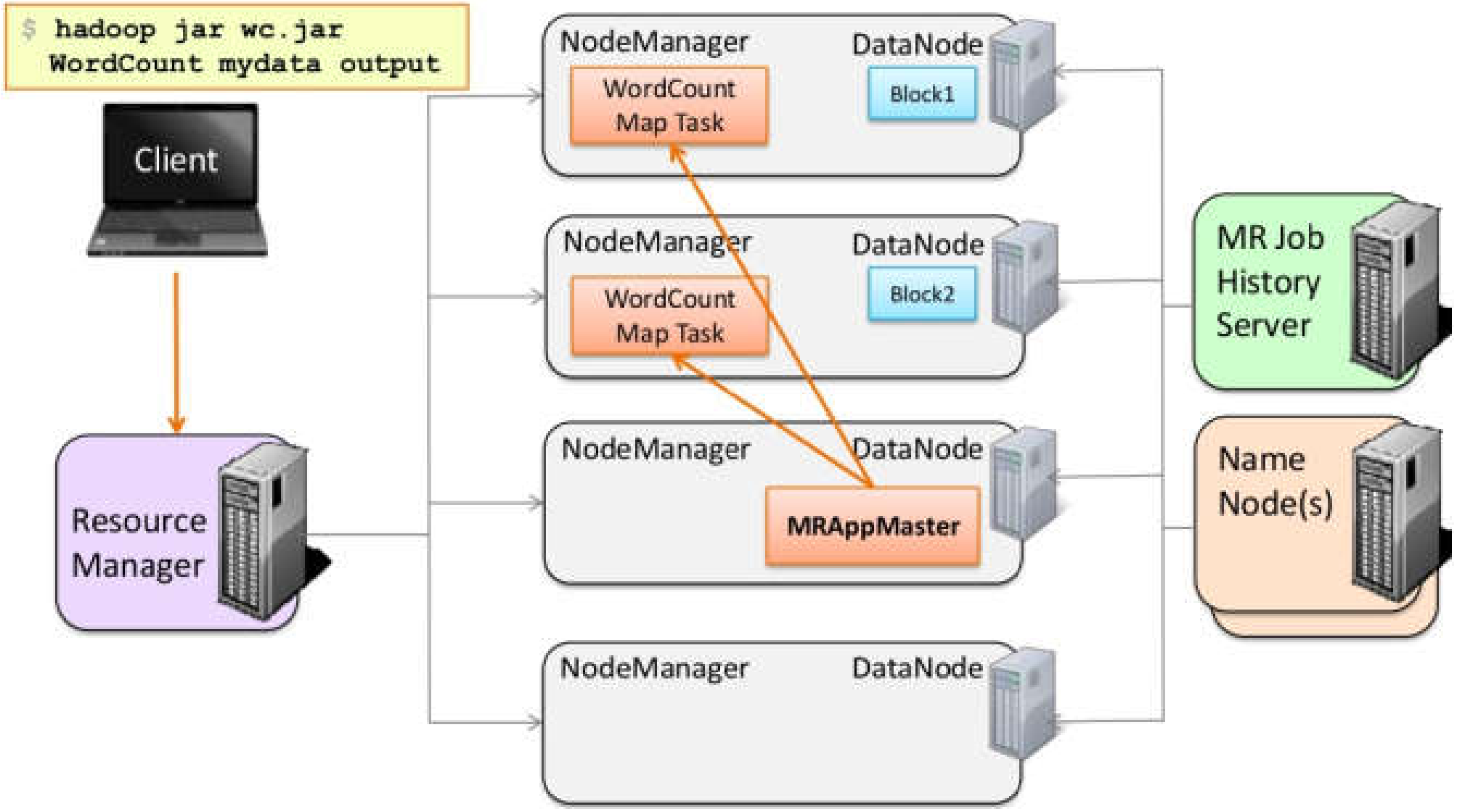
Cluster YARN: Exécution d'un job Map-Reduce



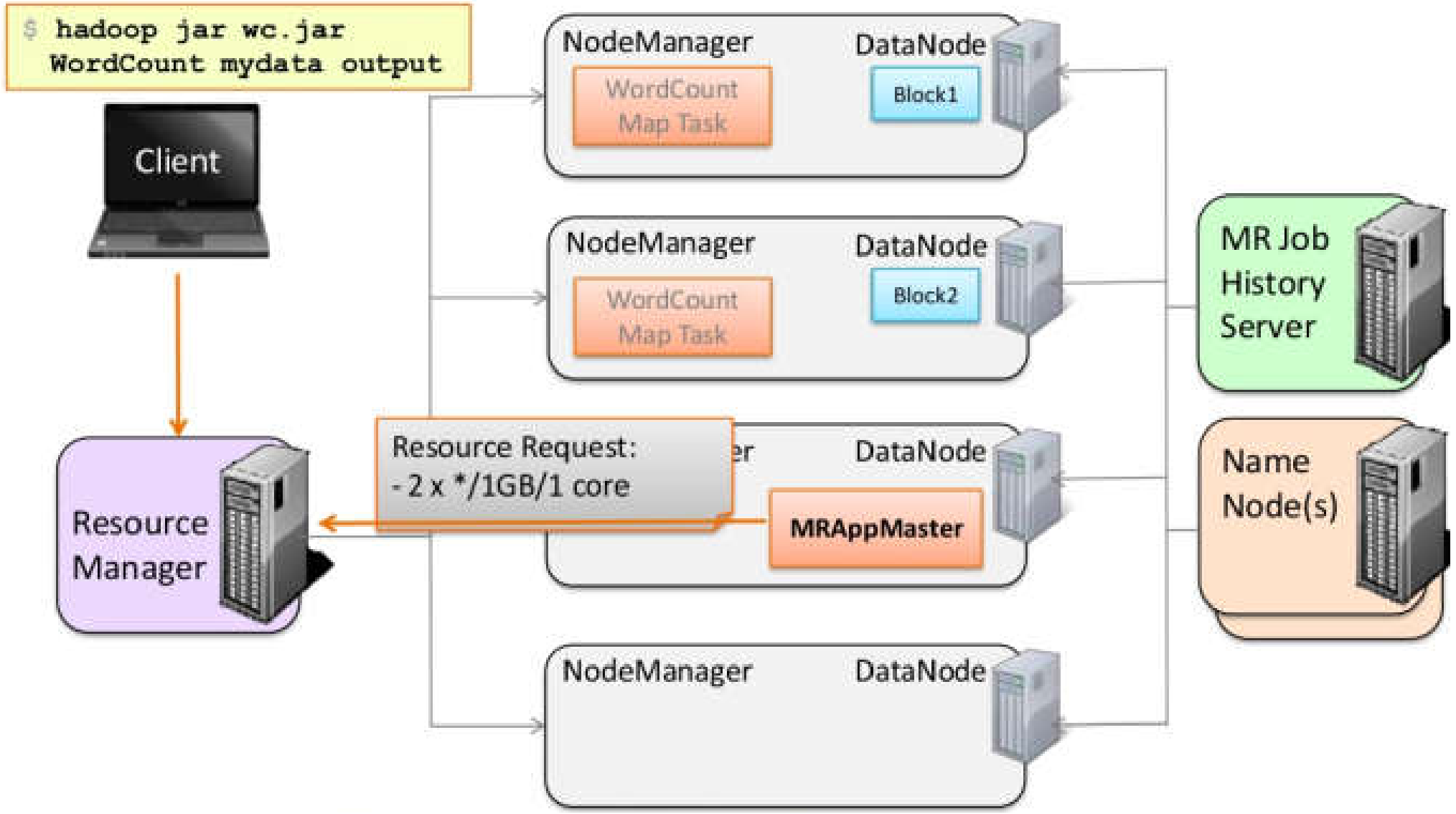
Cluster YARN: Exécution d'un job Map-Reduce



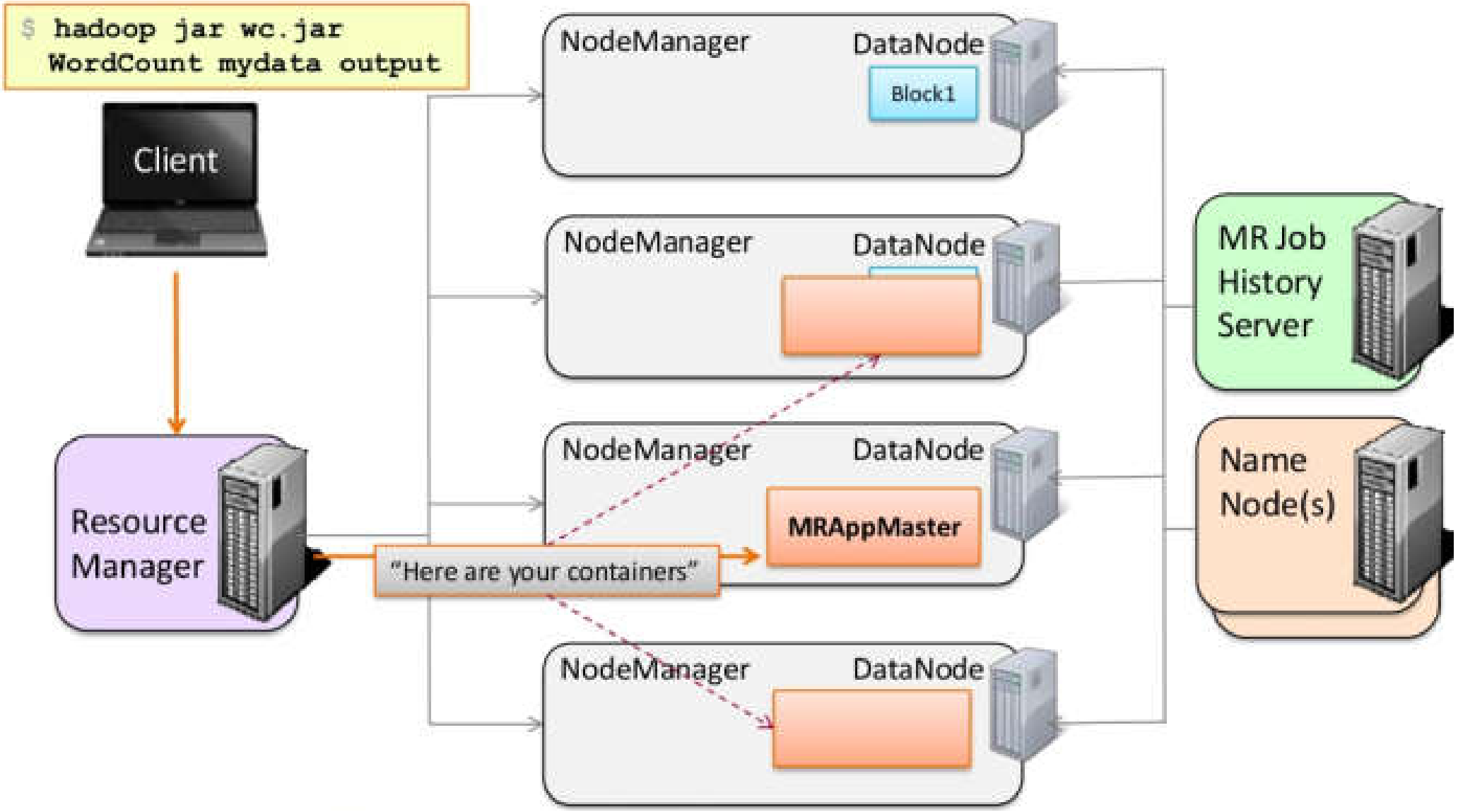
Cluster YARN: Exécution d'un job Map-Reduce



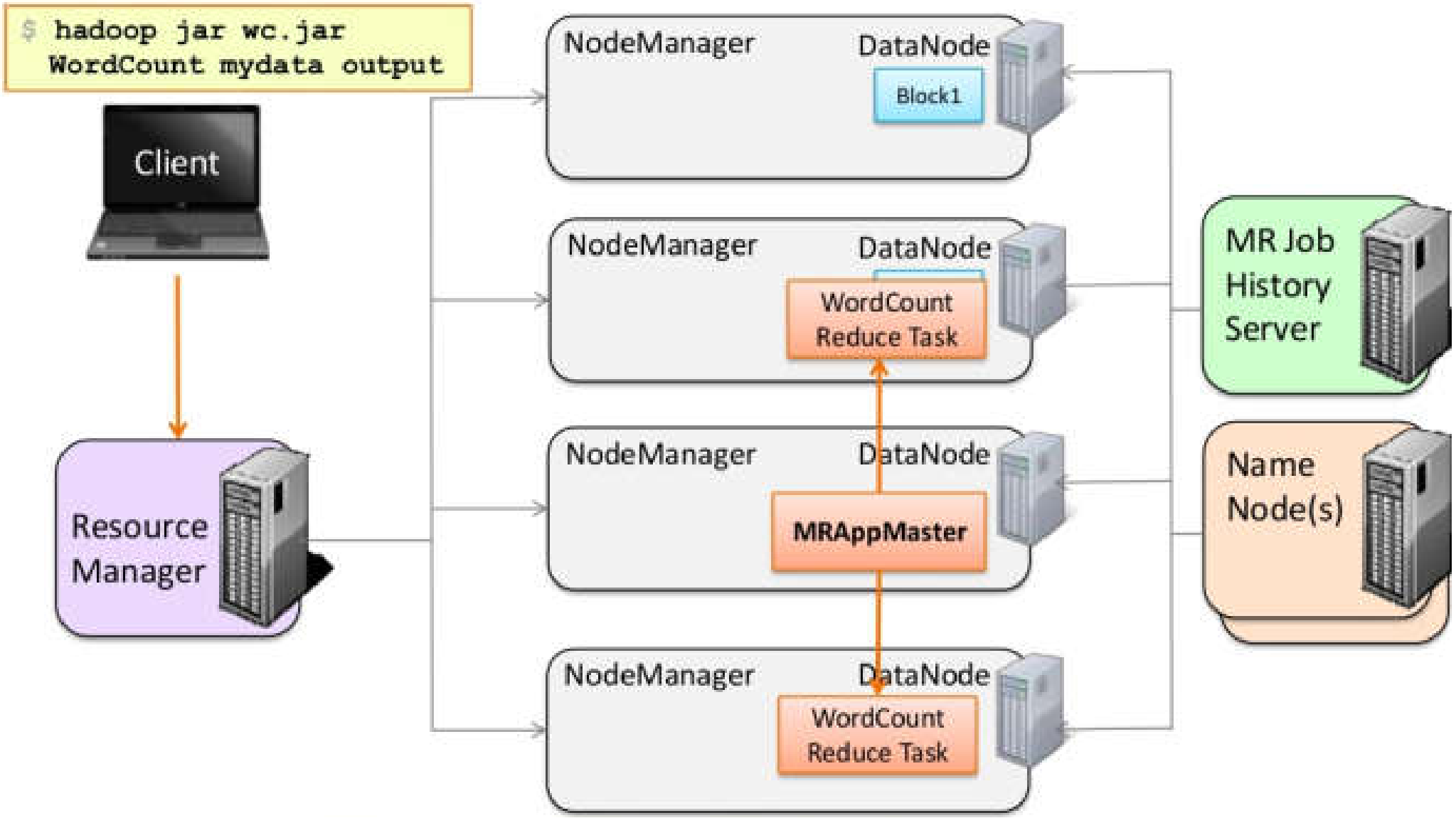
Cluster YARN: Exécution d'un job Map-Reduce



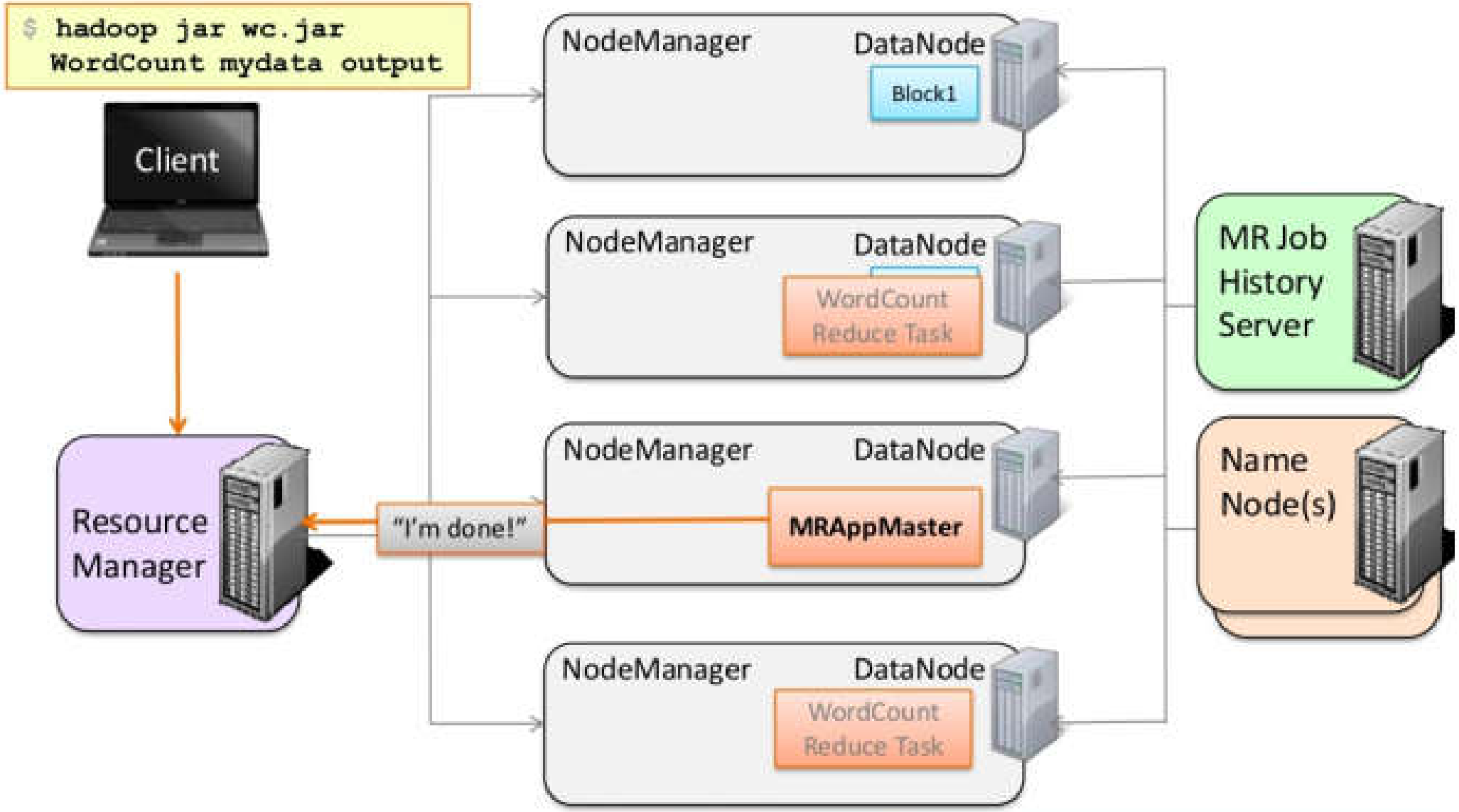
Cluster YARN: Exécution d'un job Map-Reduce



Cluster YARN: Exécution d'un job Map-Reduce

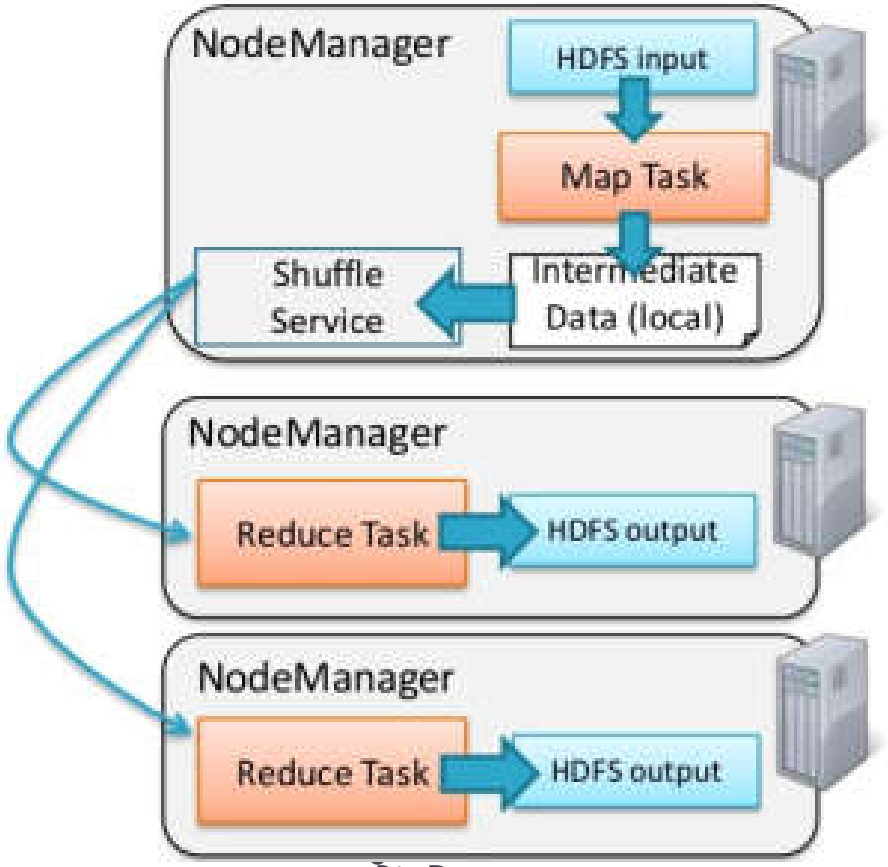


Cluster YARN: Exécution d'un job Map-Reduce



Cluster YARN: Exécution d'un job Map-Reduce

- dans Yarn, Shuffle s'exécute comme un service auxiliaire
  - Il s'éxecute dans le NodeManager JVM comme un service persistant.



API Java d'Hadoop

API Java d'Hadoop

- Comme indiqué précédemment, Hadoop est développé en Java. Les tâches MAP/REDUCE sont donc implémentables par le biais d'interfaces Java (il existe cependant des wrappers très simples permettant d'implémenter ses tâches dans n'importe quel langage).
- Un programme Hadoop se compile au sein d'un .jar.
- Pour développer un programme Hadoop, on va créer trois classes distinctes:
  - Une classe dite « DRIVER » qui contient la fonction main du programme. Cette classe se chargera d'informer Hadoop des types de données clef/valeur utilisées, des classes se chargeant des opérations MAP et REDUCE, et des fichiers HDFS à utiliser pour les entrées/sorties.
  - Une classe MAPPER (qui effectuera l'opération MAP).
  - Une classe REDUCER (qui effectuera l'opération REDUCE)

Word Count en Java: l'opération Map

100

```
private final static IntWritable one = new IntWritable(1);

public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
 StringTokenizer itr = new
                               StringTokenizer(value.toString());
 while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  context.write(word, one);
 }
}
```

- Map renvoie essentiellement 1 par mot trouvé
- Mais plein de types nouveaux!

Word Count en Java : IntWritable

101

```
private final static IntWritable one = new IntWritable(1);

public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
 StringTokenizer itr = new
                              StringTokenizer(value.toString());
 while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  context.write(word, one);
 }
}
```

- La classe IntWritable est une classe enveloppante pour les entiers (comme Integer en Java, mais permettant aussi de modifier les valeurs entières contenues), définie dans org.apache.hadoop.io.
- Un objet x instance de cette classe est donc un pointeur sur un entier, que l'on peut modifier par x.set(val) (val étant un entier). On peut déterminer la valeur pointée par x par x.get().

Word Count en Java: Type des arguments


```
private final static IntWritable one = new IntWritable(1);

public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
 StringTokenizer itr = new
                       StringTokenizer(value.toString());
 while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  context.write(word, one);
  }
}
```

- La clé key est de type général Object.
- La classe Text est une classe définie par Hadoop (dans org.apache.hadoop.io.BinaryComparable) qui permet de coder du texte au standard d'encodage UTF8: c'est une classe sérialisable, et qui permet la comparaison (compareTo)
- La fonction map prend également en entrée un contexte d'exécution (de type Context) dans lequel la fonction map va écrire ses résultats.
- La fonction map doit pouvoir lancer des exceptions, de type IOException ou InterruptedException.

Word Count en Java: Découpage en mots

103

```
private final static IntWritable one = new IntWritable(1);

public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
 StringTokenizer itr = new
                       StringTokenizer(value.toString());
 while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  context.write(word, one);
  }
}
```

- La classe StringTokenizer de java.lang.Object permet de découper une chaîne de caractères en mots.
- On passe au StringTokenizer **itr** nouvellement créé le texte donné en entrée. En fait, TextInputFormat de la classe Text définit par défaut un mode d'entrée du texte, ligne par ligne. Donc il y aura ici un processus map par ligne de texte.
- On itère ensuite sur les mots trouvés, séparés par un espace dans value (converti en chaîne de caractères).

Word Count en Java:

Émission des valeurs intermédiaires


```
private final static IntWritable one = new IntWritable(1);

public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
 StringTokenizer itr = new
                               StringTokenizer(value.toString());
 while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  context.write(word, one);
  }
}
```

- A chaque fois que l'on trouve un nouveau mot, on écrit ce mot dans un Text word, par word.set, et on écrit le résultat de la fonction (ou processus) map correspondant dans le contexte context par context.write.
- Le résultat associé est une valeur 1 (représenté par le pointeur sur entier one) pour chaque mot trouvé word.

Word Count en Java: Classe Mapper


```
public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{
 public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
          ...
```

- Une fonction map doit être définie dans une classe héritant de Mapper (défini dans org.apache.hadoop.mapreduce.Mapper).
- Cette classe est une classe *générique* Java : Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT> :
  - KEYIN qui sera le type des clés d'entrée pour la fonction map
  - VALUEIN pour le type des valeurs d'entrée,
  - KEYOUT pour le type des clés de sortie
  - VALUEOUT pour le type des valeurs de sortie
- Mapper<Object, Text, Text, IntWritable> désigne une classe qui va comprendre une méthode map, prenant des entrées de type Object*Text et rendra des valeurs de type Text*IntWritable.

Word Count en Java: Reduce


```
private IntWritable result = new IntWritable() ;

public void reduce(Text key, Iterable<IntWritable> values,
                   Context context)
              throws IOException, InterruptedException {
 int sum = 0 ;
 for (IntWritable val : values) {
  sum += val.get() ;
  }
 result.set(sum) ;
 context.write(key, result) ;
 }
}
```

- La fonction reduce prend en entrée :
  - un Text key (clé de sortie de fonctions map de la première phase),
  - une liste (Iterable<IntWritable>) de valeurs entières, qui correspondent à toutes les valeurs associées à la clé key,
  - et un Context context qui permet d'écrire une paire de valeurs de sortie, de type Text*IntWritable pour chaque mot, le nombre de fois qu'il apparaît dans le texte)

Word Count en Java: Reduce


```
private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,
                   Context context)
                throws IOException, InterruptedException {
 int sum = 0;
 for (IntWritable val : values) {
  sum += val.get();
  }
 result.set(sum);
 context.write(key, result);
 }
}
```

- Pour chaque mot (key), reduce récupère la liste des entiers associés (en fait tous des 1 ici!), en sortie de map, et un Context dans lequel écrire les résultats.

Word Count en Java: Reduce


```
private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,
                   Context context)
                throws IOException, InterruptedException {
 int sum = 0;
 for (IntWritable val : values) {
  sum += val.get();
  }
 result.set(sum);
 context.write(key, result);
 }
}
```

- On itère sur la liste (key, value) en sortie de map et on somme les value, dans sum.

Word Count en Java: Reduce

109

```
private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,
                   Context context)
              throws IOException, InterruptedException {
 int sum = 0;
 for (IntWritable val : values) {
  sum += val.get();
  }
 result.set(sum);
 context.write(key, result);
 }
}
```

- On écrit ensuite la paire (key,result) (la deuxième composante contient la somme, i.e. le nombre d'occurrence du mot key) dans le context de sortie.

Word Count en Java: Classe Reducer


```
public static class IntSumReducer
  extends Reducer<Text,IntWritable,Text,IntWritable> {
 public void reduce(Text key,
                    Iterable<IntWritable> values,
                    Context context)
             throws IOException, InterruptedException {
            ...
```

- La classe générique Reducer<K2,V2,K3,V3> : prend en entrée la liste des valeurs de sortie de Mapper (de type V2), et effectue une réduction, par clé (de sortie toujours de Mapper, ici de type K2).
- La classe IntSumReducer encapsule la méthode de réduction reduce. avec K2=Text, V2= liste de IntWritable (en fait Iterable<IntWritable>), plus un contexte (pour les résultats), contenant des paires (Text,IntWritable).

Word Count en Java: Classe Main


```
public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration() ;
 Job job = Job.getInstance(conf, "word count") ;
 job.setJarByClass(WordCount.class) ;
 job.setMapperClass(TokenizerMapper.class) ;
 job.setReducerClass(IntSumReducer.class) ;
 job.setOutputKeyClass(Text.class) ;
 job.setOutputValueClass(IntWritable.class) ;
 FileInputFormat.addInputPath(job, new Path(args[0])) ;
 FileOutputFormat.setOutputPath(job, new Path(args[1])) ;
 System.exit(job.waitForCompletion(true) ? 0 : 1) ;
 }
}
```

- Le main démarre par définir la configuration de Hadoop, qui lit en particulier le fichier core-site.xml et ajoute d'autres données propres à l'application.

Word Count en Java: Classe Main


```
public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration();
 Job job = Job.getInstance(conf, "word count");
 job.setJarByClass(WordCount.class);
 job.setMapperClass(TokenizerMapper.class);
 job.setReducerClass(IntSumReducer.class);
 job.setOutputKeyClass(Text.class);
 job.setOutputValueClass(IntWritable.class);
 FileInputFormat.addInputPath(job, new Path(args[0]));
 FileOutputFormat.setOutputPath(job, new Path(args[1]));
 System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

- Puis on définit la classe principale, et les classes pour map et reduce.

Word Count en Java: Classe Main


```
public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration();
 Job job = Job.getInstance(conf, "word count");
 job.setJarByClass(WordCount.class);
 job.setMapperClass(TokenizerMapper.class);
 job.setReducerClass(IntSumReducer.class);
 job.setOutputKeyClass(Text.class);
 job.setOutputValueClass(IntWritable.class);
 FileInputFormat.addInputPath(job, new Path(args[0]));
 FileOutputFormat.setOutputPath(job, new Path(args[1]));
 System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

- On définit ensuite le type des valeurs des clés de sortie des processus reduce et le type des valeurs associées à ces clés, en sortie des processus reduce.

Word Count en Java: Classe Main


```
public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration();
 Job job = Job.getInstance(conf, "word count");
 job.setJarByClass(WordCount.class);
 job.setMapperClass(TokenizerMapper.class);
 job.setReducerClass(IntSumReducer.class);
 job.setOutputKeyClass(Text.class);
 job.setOutputValueClass(IntWritable.class);
 FileInputFormat.addInputPath(job, new Path(args[0]));
 FileOutputFormat.setOutputPath(job, new Path(args[1]));
 System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

- Enfin, on précise les chemins dans la partition HDFS dans lesquels trouver les arguments en entrée, et dans lesquels écrire en sortie.
- main appelle à la fin job.waitForCompletion qui met en route les processus et attend leur terminaison

Modes d'Exécution de Hadoop

- Hadoop propose trois modes d'exécution
- Local (standalone)
  - tout s'exécute au sein d'une seule JVM, en local.
  - mode recommandé en phase de développement (des petits tests et débogages)
- Local (pseudo-distributed):
  - le fonctionnement en mode cluster est simulé par le lancement des tâches dans différentes JVM exécutées localement.
  - La configuration est presque comme celle d'un cluster
- Fully-distributed (cluster)
  - c'est le mode d'exécution réel d'Hadoop.
  - Il permet de faire fonctionner le système de fichiers distribué et les tâches sur un ensemble de machines.
  - Une topologie **maître-esclave** est mise en place.

Exécution en mode standalone

- Pour pouvoir exécuter le programme que l'on vient d'écrire (WordCount.java) on le compile et on crée tout d'abord une archive .jar.

```
> echo $CLASSPATH
/Users/Eric/Cours/X/INF431/Hadoop/hadoop-2.5.1./share/hadoop/common/
hadoop-common-2.5.1.jar :/Users/Eric/Cours/X/INF431/Hadoop/hadoop-2.5.1./
share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.5.1.jar
> mkdir Classes
> javac -d Classes WordCount.java
> ls Classes/org/myorg/WordCount
WordCount.class          WordCount$Map.class      WordCount$Reduce.class
> jar -cvf Classes.jar -C Classes .
manifeste ajoute
ajout : org/(entree = 0) (sortie = 0)(stockage : 0 %)
ajout : org/myorg/(entree = 0) (sortie = 0)(stockage : 0 %)
ajout : org/myorg/WordCount$Map.class(entree = 1938) (sortie = 799)
ajout : org/myorg/WordCount$Reduce.class(entree = 1611) (sortie = 648)
ajout : org/myorg/WordCount.class(entree = 1538) (sortie = 753)
```

Exécution en mode standalone

- On crée maintenant des répertoires HDFS pour pouvoir contenir le texte d'entrée et les résultats en sortie :

```
> ls Fichiers
fichier1.txt fichier2.txt
> cat Fichiers/fichier1.txt
Ceci est le cours INF431
> cat Fichiers/fichier2.txt
Ceci est le TD INF431
```

- Et on copie ces fichiers sur la partition HDFS:

```
> bin/hdfs dfs -put Fichiers
> bin/hdfs dfs -ls Fichiers
-rw-r--r--   1 Eric supergroup      26 2014-12-04 15:28 Fichiers/fichier1.txt
-rw-r--r--   1 Eric supergroup      23 2014-12-04 15:28 Fichiers/fichier2.txt
```

Exécution en mode standalone

- Puis on exécute le code :

```
> hadoop jar Classes.jar org.myorg.WordCount Fichiers output
```

- Dans le répertoire de sortie des résultats, output, qui a été crée, on trouve tous les mots avec leur nombre d'occurrence.

```
> bin/hdfs dfs -ls output
Found 2 items
-rw-r--r--   1 Eric supergroup          0 2014-12-04 16:08 output2/_SUCCESS
-rw-r--r--   1 Eric supergroup         40 2014-12-04 16:08 output2/part-r-00000
> bin/hdfs dfs -cat output/*
Ceci 2
INF431 2
TD 1
cours 1
est 2
le 2
```

Fin

Happy Hadooping

☺