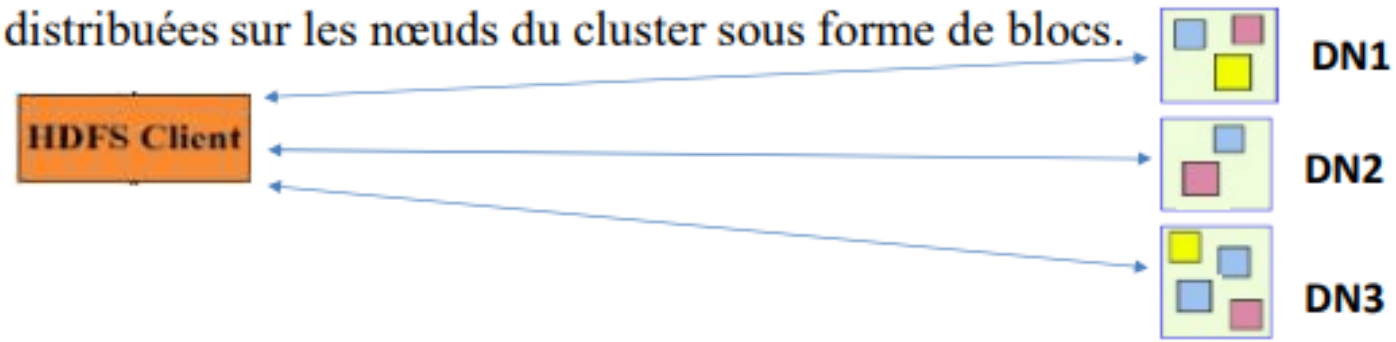
MapReduce/YARN

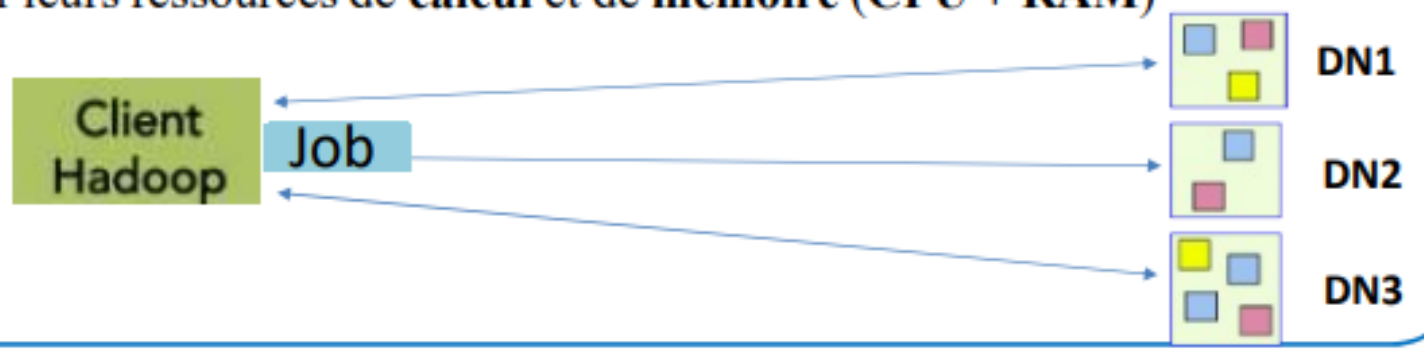
Hadoop: MapReduce

- **Principe:**

- Les données sont distribuées sur les nœuds du cluster sous forme de blocs.



- Ces données ne seront pas déplacées, via le réseau, vers un programme devant les traiter.
- Pour de meilleures performances, chaque bloc de données est traité **localement** *(principe de **data locality**)*, minimisant les besoins d'échanges réseaux entre les machines.
- Un programme (*job*) doit donc être exécuté sur les nœuds contenant les données à traiter, où il va exploiter leurs ressources de **calcul** et de **mémoire (CPU + RAM)**

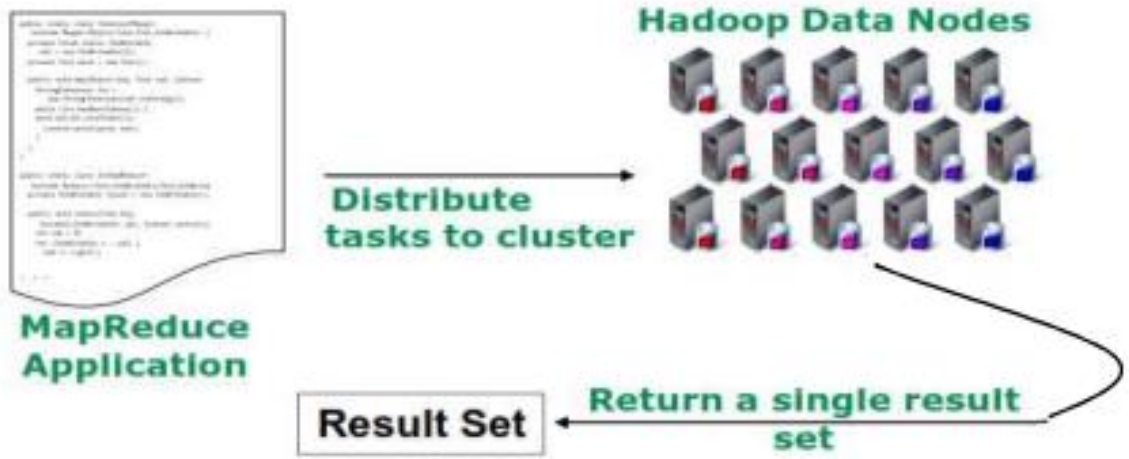


Hadoop: MapReduce

- **Principe:**

- Le programme doit être écrit selon un **modèle** bien déterminé: **MapReduce**.

- Le système de fichiers distribués (HDFS) est au cœur de **MapReduce**. Il est responsable de la distributions des données à travers le cluster, en faisant en sorte que l'ensemble du cluster ressemble à un système de fichiers géant.



Hadoop: MapReduce

- **Intérêt:**

- Simplifier le développement de programme devant traiter des données distribuées.
- Le développeur n'a pas à se soucier du travail de **parallélisation** et de **distribution** du **traitement (*job*)**. **MapReduce** permet au développeur de ne s'intéresser qu'à la partie **algorithmique** à appliquer aux données.
- Un programme **MapReduce** contient deux fonctions principales **Map ()** et **Reduce ()** contenant les traitements à appliquer aux données.

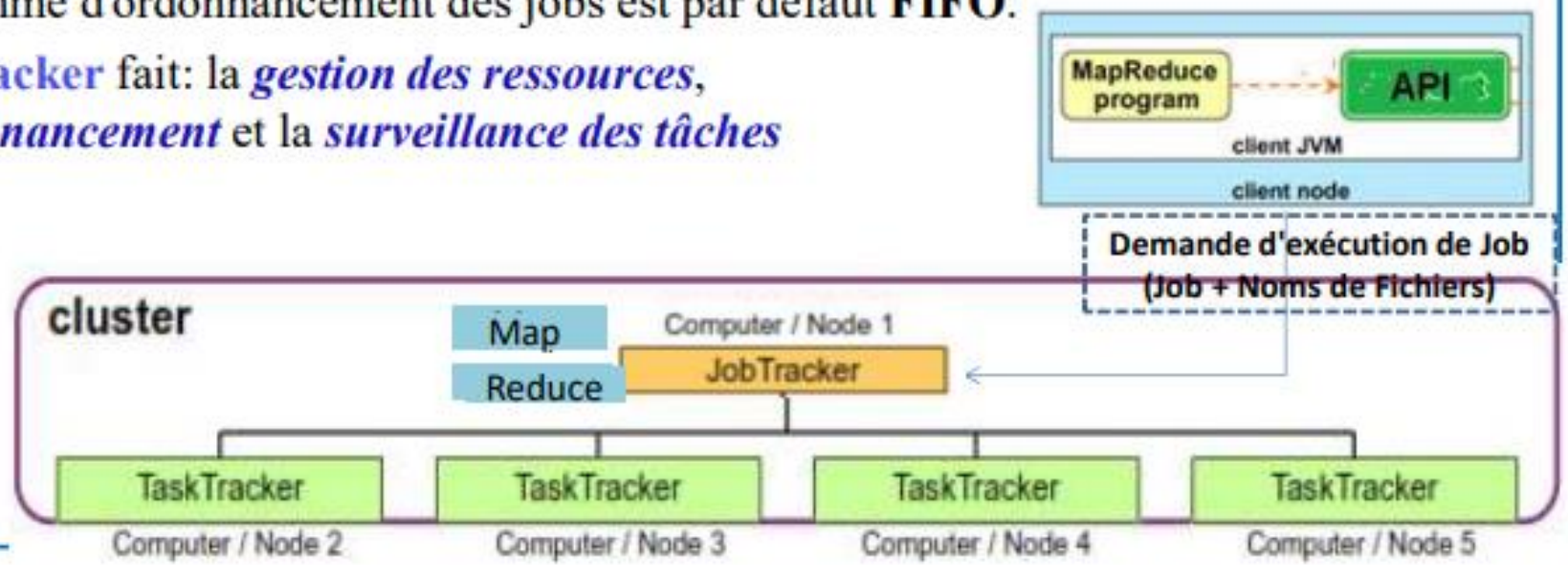
- **Architecture MR1 (dans Hadoop 1)**

- **Maître/Esclave**: L'unique maître (**JobTracker**) contrôle l'exécution des deux fonctions sur plusieurs esclaves (**TaskTrackers**).

MapReduce: JobTracker

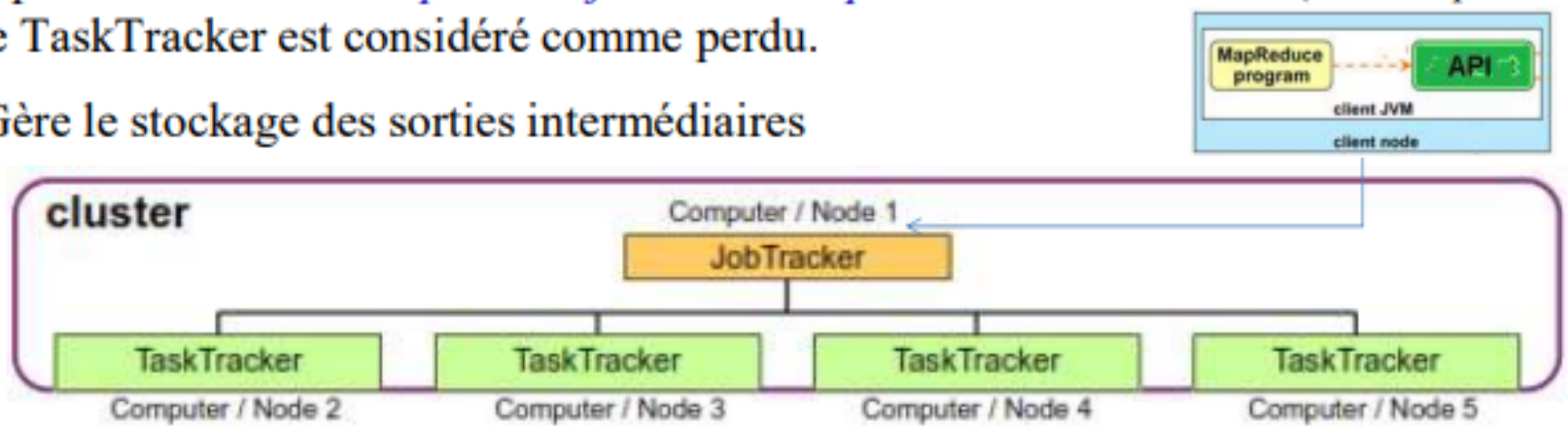
- Reçoit les **jobs MapReduce** envoyés par les clients (*Applications*)
- Communique avec le **NameNode** pour avoir les emplacements des données à traiter
- Passe les tâches **Map** et **Reduce** aux nœuds **TaskTrackers**
- Surveille les tâches et le statut des **TaskTrackers**
- Relance une tâche si elle échoue.
- Surveille l'état d'avancement des jobs et fournit des informations à ce sujet aux applications clientes.
- Algorithme d'ordonnancement des jobs est par défaut **FIFO**.

➔ **JobTracker** fait: la ***gestion des ressources***, l'***ordonnancement*** et la ***surveillance des tâches***



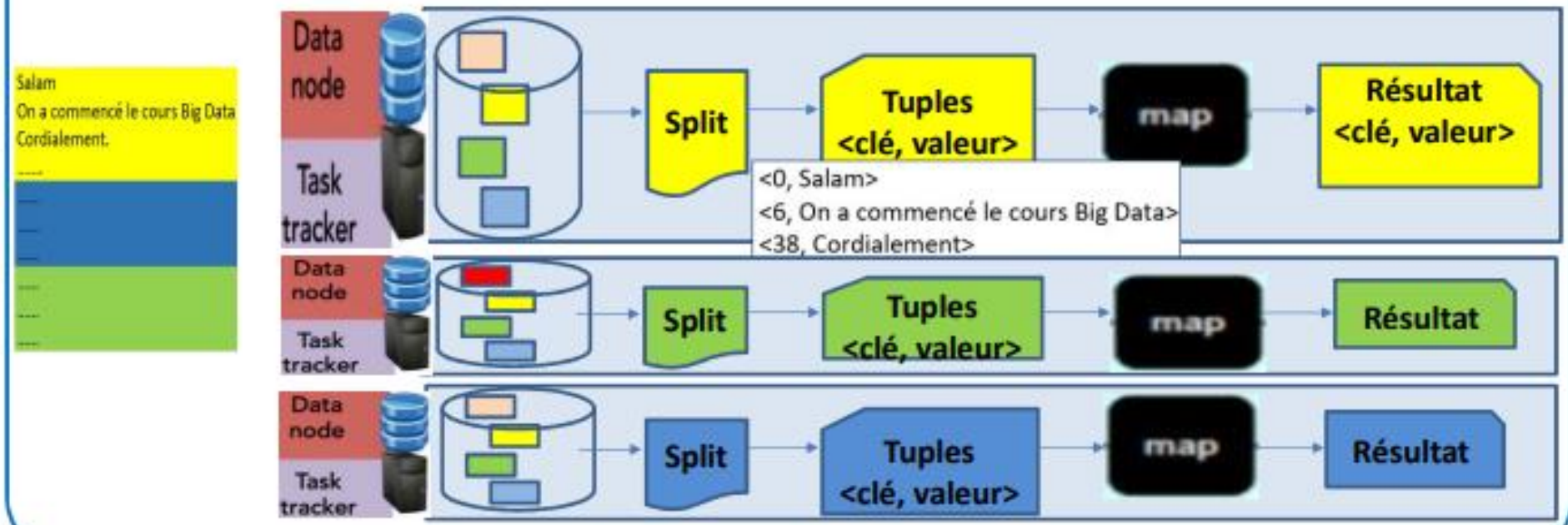
MapReduce: TaskTracker

- Exécute les tâches **Map** et **Reduce**
- Chaque TaskTracker est configuré avec un nombre de "**slots**", qui représentent sa le nombre de tâches **Map** et **Reduce** qu'il peut exécuter (**mapreduce.tasktracker.map.tasks.maximum** et **mapreduce.tasktracker.reduce.tasks.maximum** dans **mapred-site.xml**).
- Pour lancer une tâche Map, le JobTracker cherche un slot Map libre sur un nœud possédant les données à traiter. S'il n'existe pas, il cherche sur un nœud du même rack.
- Communique son statut au **JobTracker** via des **heartbeat** (*id, slots libres*, …)
- Après une durée de *mapreduce.jobtracker.expire.trackers.interval* (=10 min par défaut) le TaskTracker est considéré comme perdu.
- Gère le stockage des sorties intermédiaires



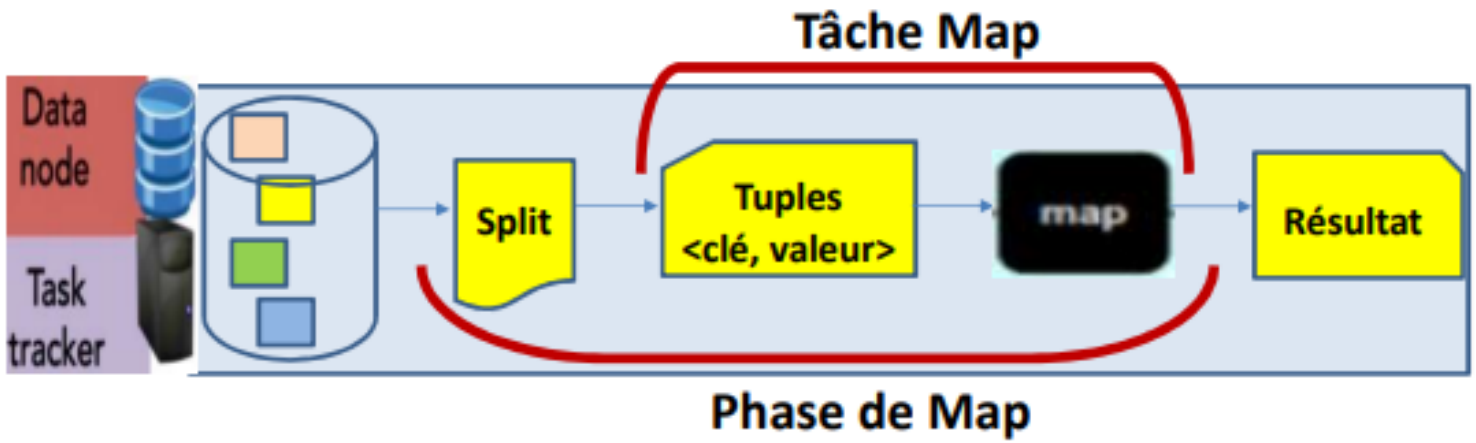
MapReduce: Modèle de programmation - Map

- Les fichiers d'entrée sont divisés en pièces **logiques** appelées "**Input Splits**" (**Splits**)
- Un **split** est une division **logique** des fichiers qui sont stockés sous forme de **blocs** HDFS. Ces blocs représentent une division **physique** des fichiers.
- Un **split** **désigne** une partie d'un fichier (*début*, *fin*, *id blocs+emplacments*, …)
- La taille d'un **split** peut être définie dans le job à exécuter (*par défaut: taille de bloc*)
  (max(mapred.min.split.size, min(mapred.max.split.size, dfs.block.size))



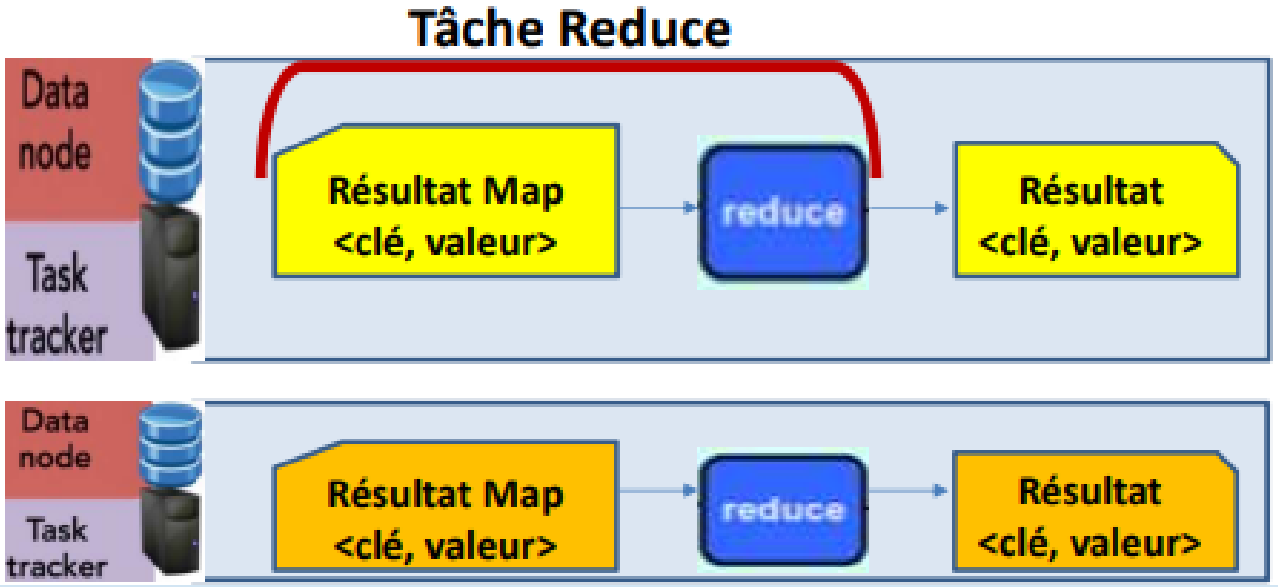
MapReduce: Modèle de programmation - Map

- Les données (lignes) correspondantes à un split sont transformées en tuples *<Clé, Valeur>*
- Chaque **split** est traité par une **tâche Map** qui applique la fonction **Map** de l'utilisateur sur chaque **tuple** du split.
- Les nœuds esclaves (**TaskTracker**) exécutent les tâches Map pour traiter les **splits** individuellement en parallèle (sous le contrôle global du **JobTracker**)
- Chaque nœud esclave stocke les résultats de ses tâches Map dans son **système de fichiers local** s'ils dépassent un seuil dans la RAM.



MapReduce: Modèle de programmation - Reduce

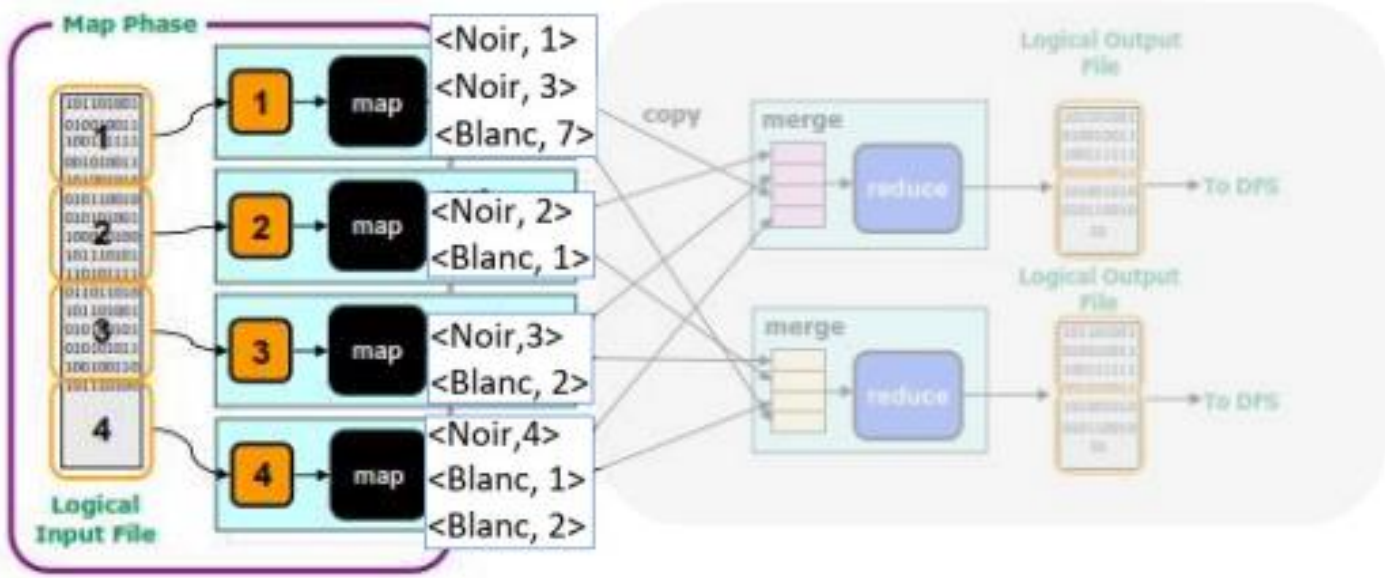
- Les résultats des tâches **Map** sont agrégées par des **tâches Reduce** sur des nœuds esclaves (sous le contrôle du **JobTracker**).
- Plusieurs **tâches Reduce** parallélisent l'agrégation
- Une **tâche Reduce** appliquent le fonction **Reduce** de l'utilisateur sur les résultats (**<clé, valeur>**) de tâches **Map**.
- Les sorties sont stockées dans **HDFS** (**et donc répliquées**)



MapReduce: La phase de Map

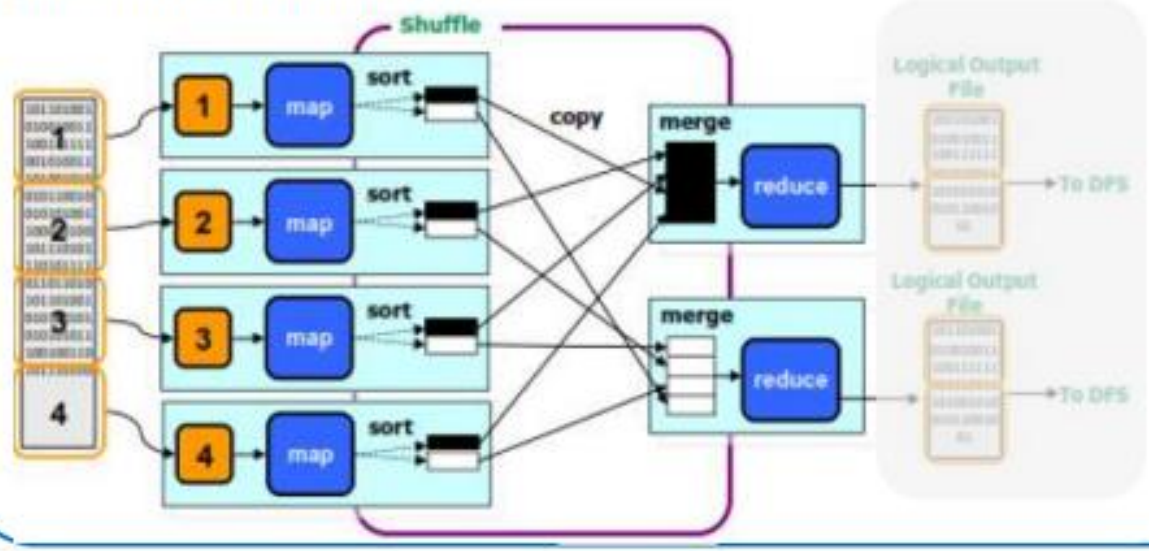
Mapper

- Petit programme (*généralement*), distribué dans le cluster et local aux données
- Traite une partie des données en entrée (*Split*)
- Chaque **Map analyse**, **filtre** ou **transforme** un split qui est un ensemble de pairs **<key, value>**.
- Produit des pairs **<clé, valeur>** groupés



MapReduce: La phase de Shuffle (réorganisation)

- Le traitement de cette phase est **préprogrammé** dans **MapReduce**
- Le résultat ({*<clé, valeur>*}) produit par chaque **Map** est **localement regroupée** par clé.
- Les pairs de **même clé** sont envoyés (**copiés**) au **même nœud**, choisi par le **JobTracker,** qui exécutera la phase **Reduce** sur ces données.
- Avant l'exécution de la tâche **Reduce**, les pairs de même clé sont **fusionnés** sur ce nœud pour former **un seul pair**

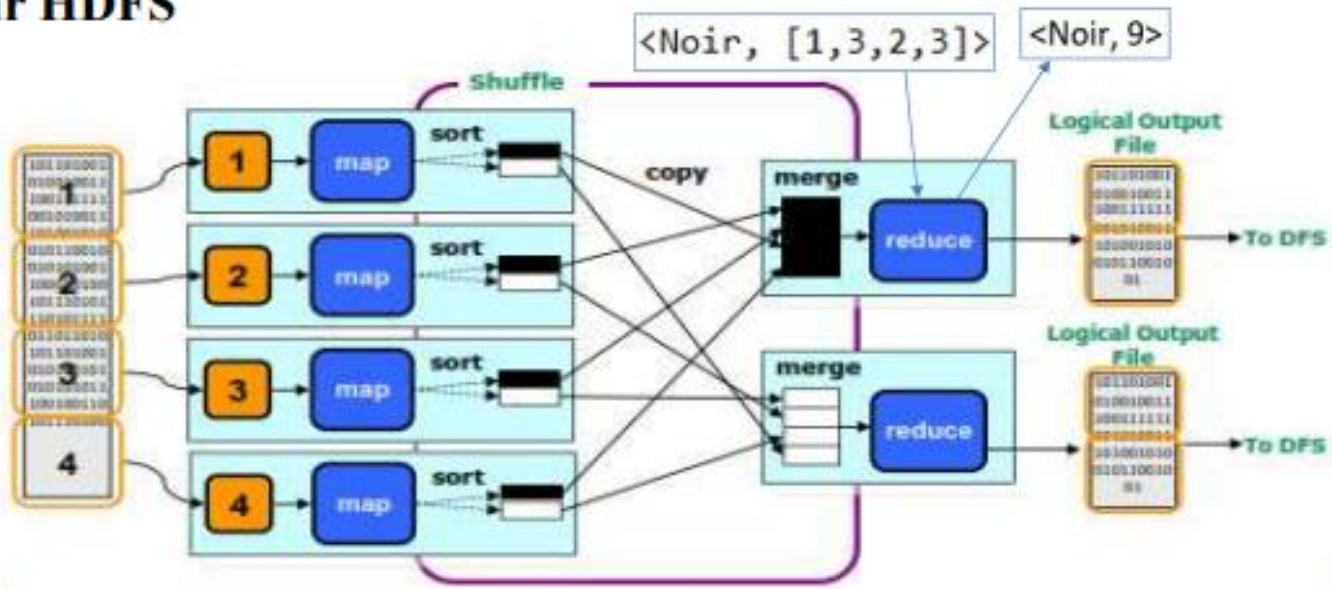


<noir, 1>
<noir, 3>
<noir ,2>
<noir, 3>
<noir, [1,3,2,3]>

MapReduce: La phase de Reduce

Reducer

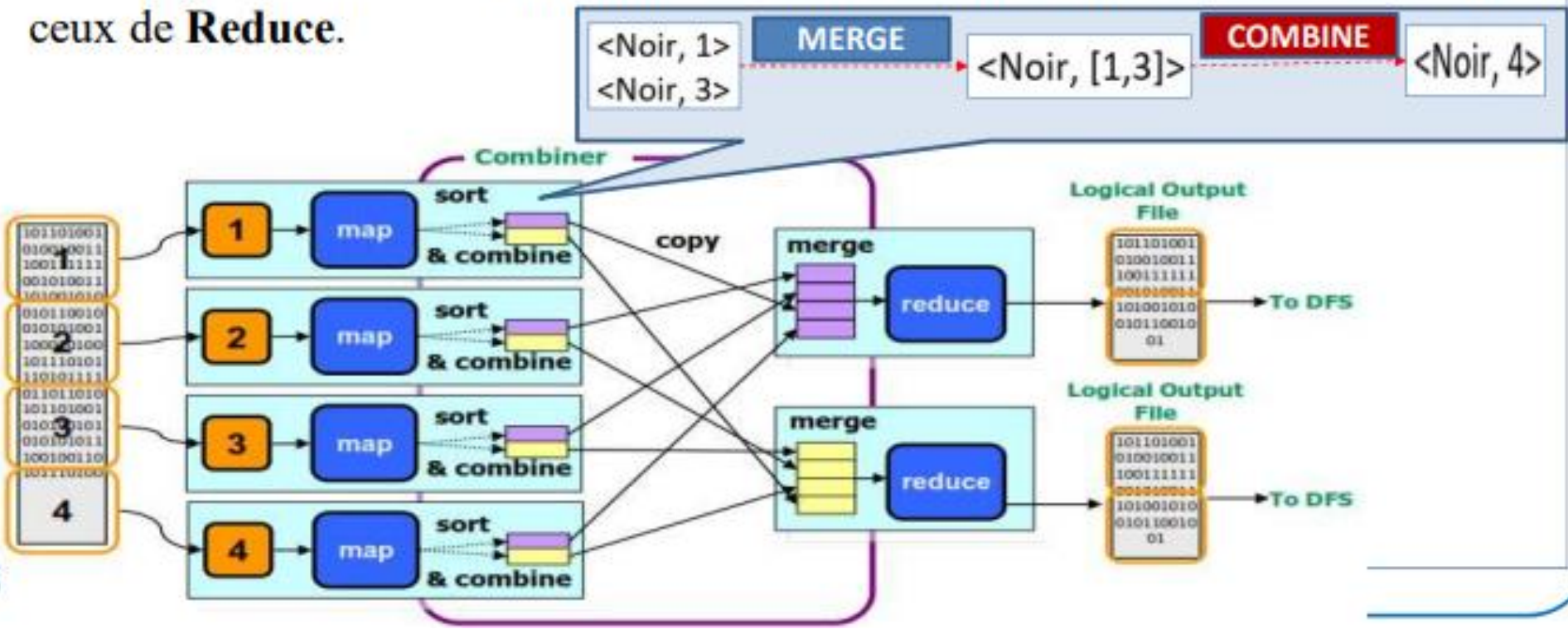
- Petit programme (généralement) qui traite toutes les valeurs de la clé dont il est responsable. Ces valeurs sont passées au **Reducer** sous forme d'un tableau.
- Chaque **Tâche Reduce** écrit son résultat **dans son propre fichier de sortie sur HDFS**



MapReduce: La phase de Combine *(optionnel)*

Combiner

- Les résultats de **Map** sont **triés** et **fusionnés (par clé)** en local puis traités par une fonction généralement identique à **Reduce** exécutée par les nœuds ayant réalisé l'opération de **Map**.
- Ceci permettra aussi de **minimiser le trafic réseau** entre les nœuds de **Map** et ceux de **Reduce**.



Exemple MapReduce

Objectif: On a un fichier texte sur HDFS contenant les noms d'animaux (*un nom/ligne*) qui se répètent.

On veut calculer le nombre de chaque félin (*tigre*, *lion*, *panthère*, ….)

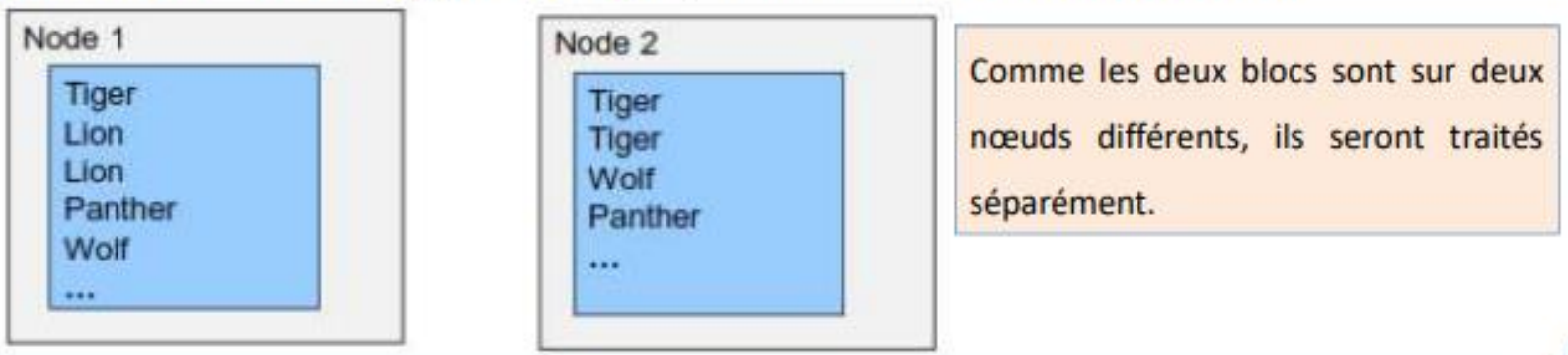
- En SQL, on aurait pu avoir le résultat par la requête:

```
SELECT NAME, COUNT(NAME) FROM   ANIMALS
WHERE   NAME  IN  ('Tiger', 'Lion', …)
GROUP BY NAME;
```

Table

Animals (Name Varchar2(20), …..)

- Le fichier a été divisé, par exemple, en deux blocs sur deux nœuds:



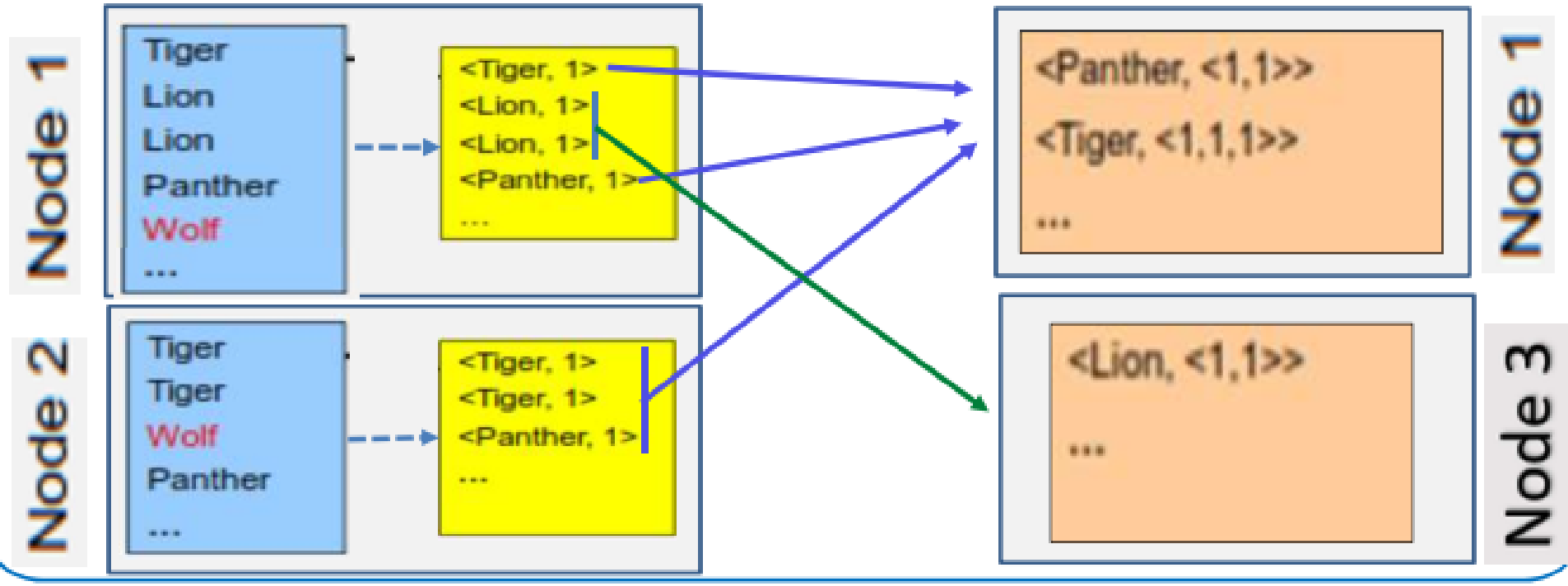
Comme les deux blocs sont sur deux nœuds différents, ils seront traités séparément.

Exemple MapReduce: La tâche Shuffle

- Déplacer tous les **pairs ayant la même clé** vers le **même nœud** cible

clé[0] ∈ ['A', 'L'] → pairs vers **Noeud3**, clé[0] ∈ ['M', 'Z'] → pairs vers **Noeud1**

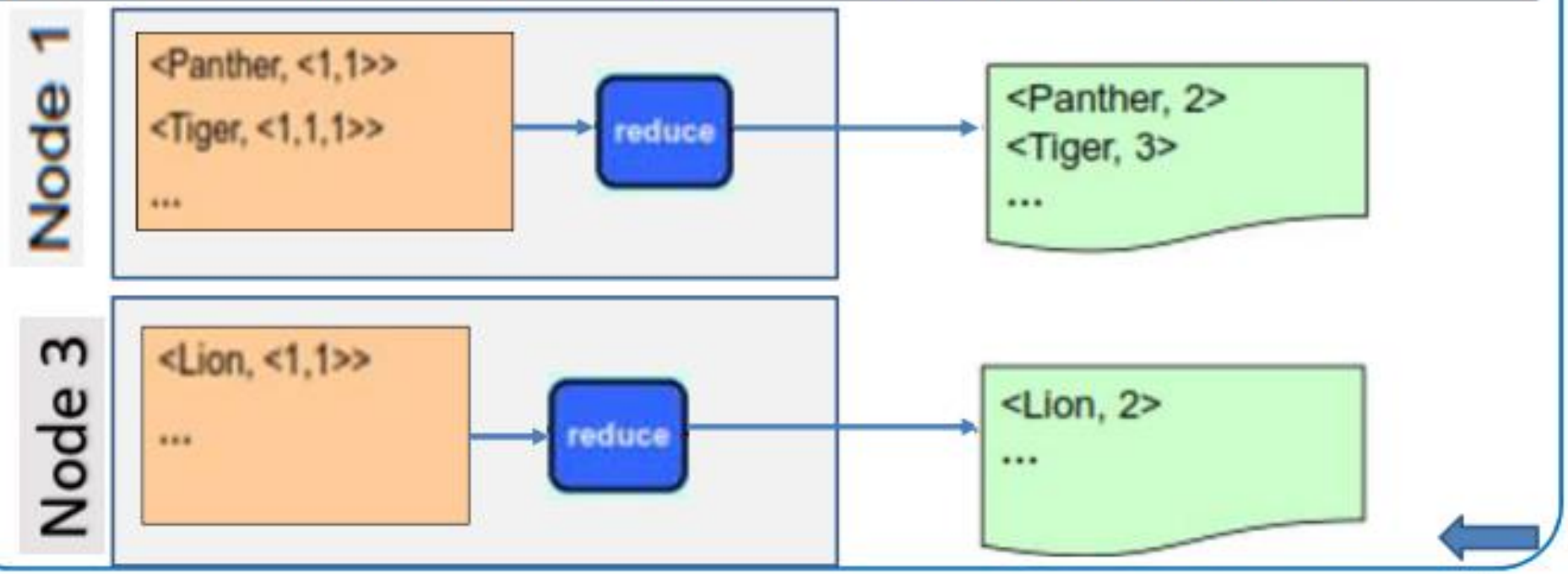
- Les tâches **Reduce** peuvent s'exécuter sur n'importe quel nœud.
- Le nombre de tâches **Map** et **Reduce** n'est pas forcément le même.
- Généralement le nombre de tâches **Reduce** est plus petit que celui des **Map**.



Exemple MapReduce: La tâche Reduce

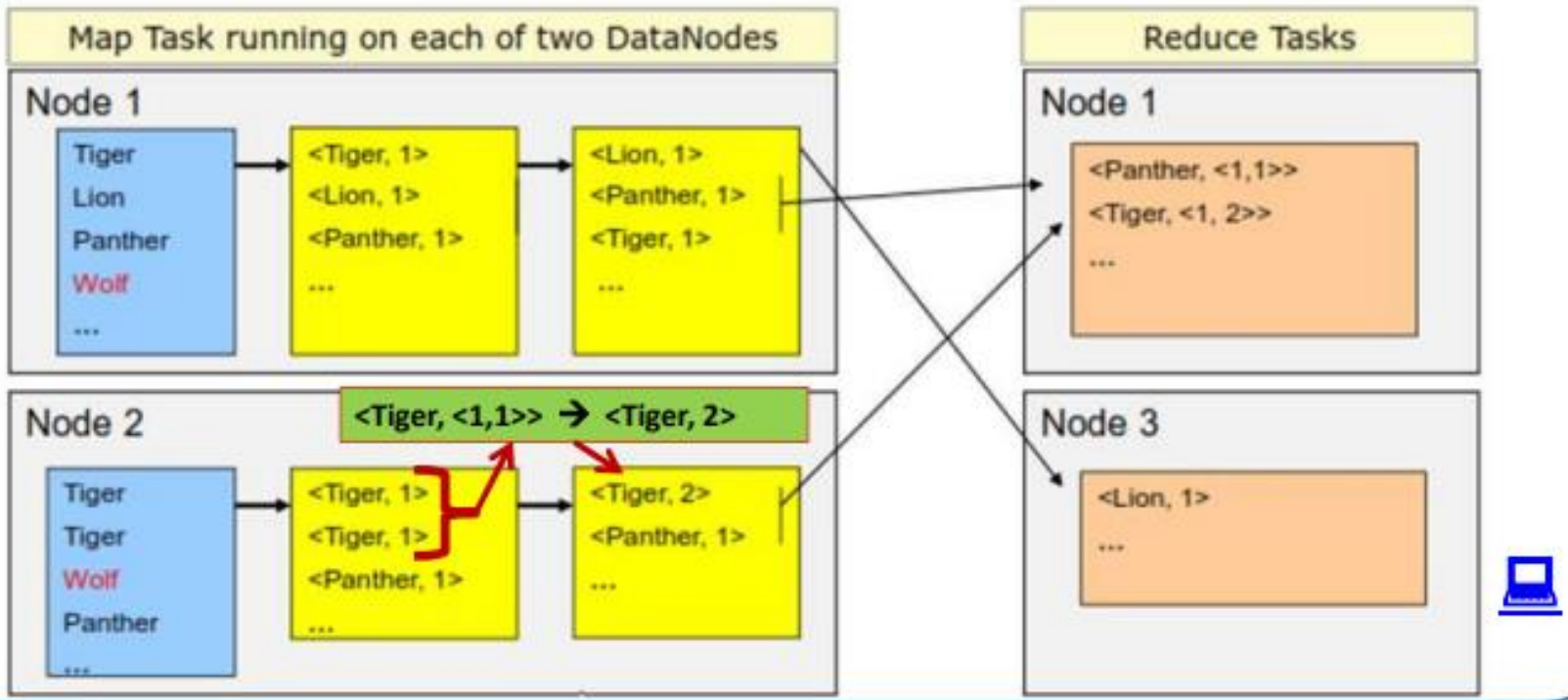
- La tâche **Reduce** calcule des valeurs agrégées pour chaque clé, dans notre cas, le nombre d'occurrence de la clé (*nom d'animal*) c'est ***la somme des valeurs fusionnées***.
- La sortie (résultat) d'une tâche **Reduce** est écrite dans un fichier HDFS

```
public void reduce(Text key, Iterable<IntWritable> values, Context sortie) { int sum = 0;
for (IntWritable V: values) { sum += V.get(); } sortie.write(key , new IntWritable(sum)); }
```



Exemple MapReduce: La tâche Combine

- Pour améliorer les performances, les résultats de **Map** sont **agrégées** sur les nœuds l'ayant produit (**avant** la phase **Shuffle**)
- On réduit la quantité de données envoyées sur le réseau

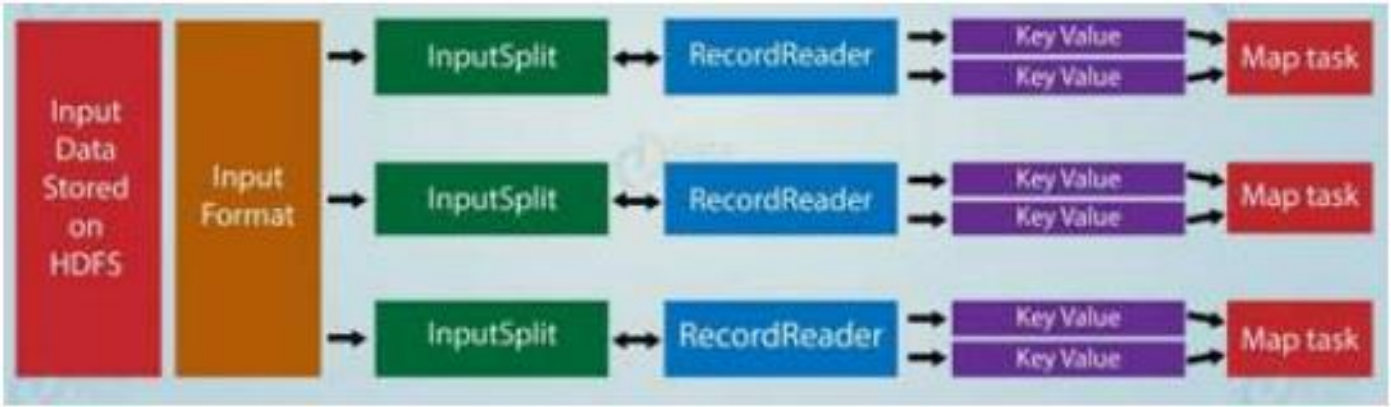


MapReduce: SPLITS

- Les fichiers dans Hadoop sont stockés dans des blocs (128 Mo).
- **MapReduce** divise logiquement les données en fragments ou **Splits**.
- Une tâche **Map** est exécutée sur chaque **Split**.
- La plupart des fichiers sont sous forme d'enregistrements séparés par des caractères bien définis.
- Le caractère le plus courant est le caractère de fin de ligne.
- La classe **InputFormat** est chargée de prendre un fichier HDFS et le transformer en **Splits** (**InputSplit**). (La méthode **getSplits** retourne une liste d'objet **InputSplit** dont chacun représente un split <*chemin du fichier, début, fin, …*>)

MapReduce: Quelques Classes

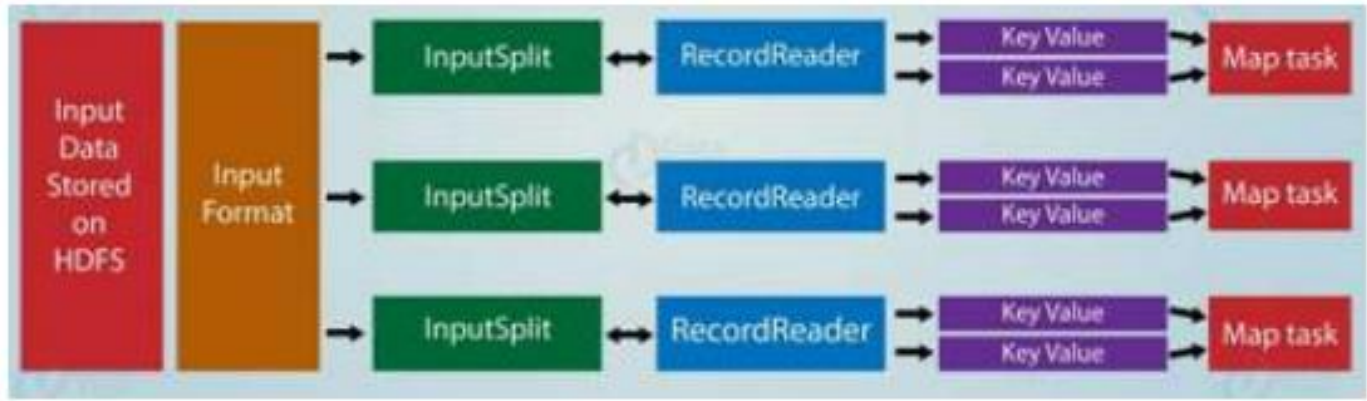
Trois classes principales lisent des données dans MapReduce:



InputFormat: divise les fichiers à traiter en plusieurs **splits** dont la taille est **128M** ou la valeur du paramètre ***mapred.min.split.size*** dans ***mapred-site.xml***. On peut aussi préciser la taille dans le code du job MapReduce. On peut également créer une classe **InputFormat** personnalisée pour indiquer comment un fichier peut être divisé en **splits**.

InputSplit: Chaque split est représenté par un objet de la classe **InputSplit** qui est une représentation logique des données. Les données traitées par une tâche **Map** sont représentées par un objet **InputSplit** (*taille, informations sur les nœuds contenant les données du split*)

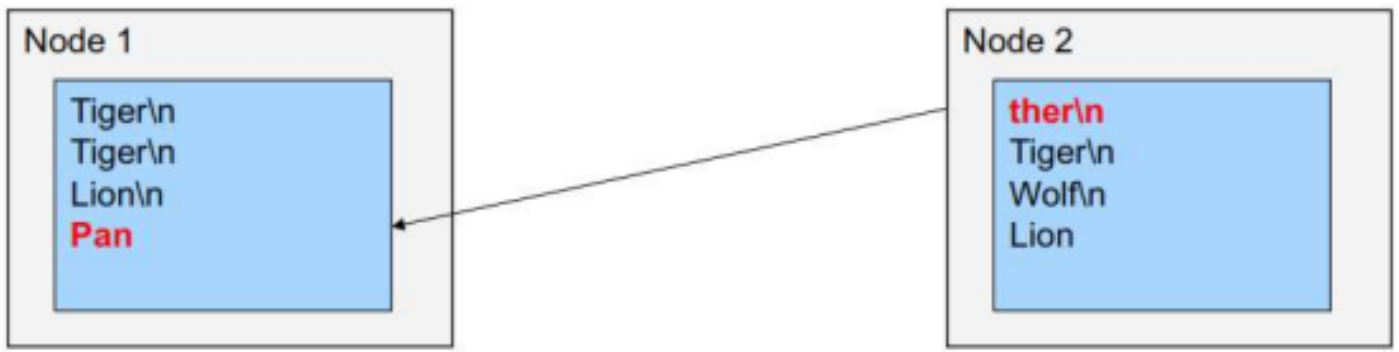
MapReduce: Quelques Classes



- **RecordReader**: Elle communique avec **InputSplit** pour convertir les lignes (**LineRecordReader**) des fichiers (**Splits**) en pairs ***<clé, valeur>.*** Par défaut, **RecordReader** utlise **TextInputFormat** pour faire cette conversion.

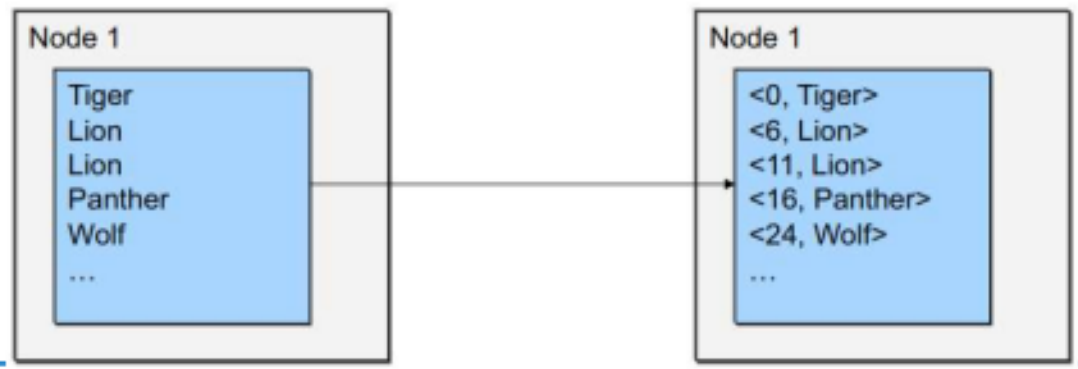
MapReduce: RecordReader

- La plupart du temps, la fin d'un split ne se trouve pas à la fin d'un bloc
- Les fichiers sont lus dans des **Records** par la classe **RecordReader**.
- Dans le cas où le dernier enregistrement d'un bloc se termine dans autre bloc, HDFS enverra la partie manquante du dernier enregistrement via le réseau

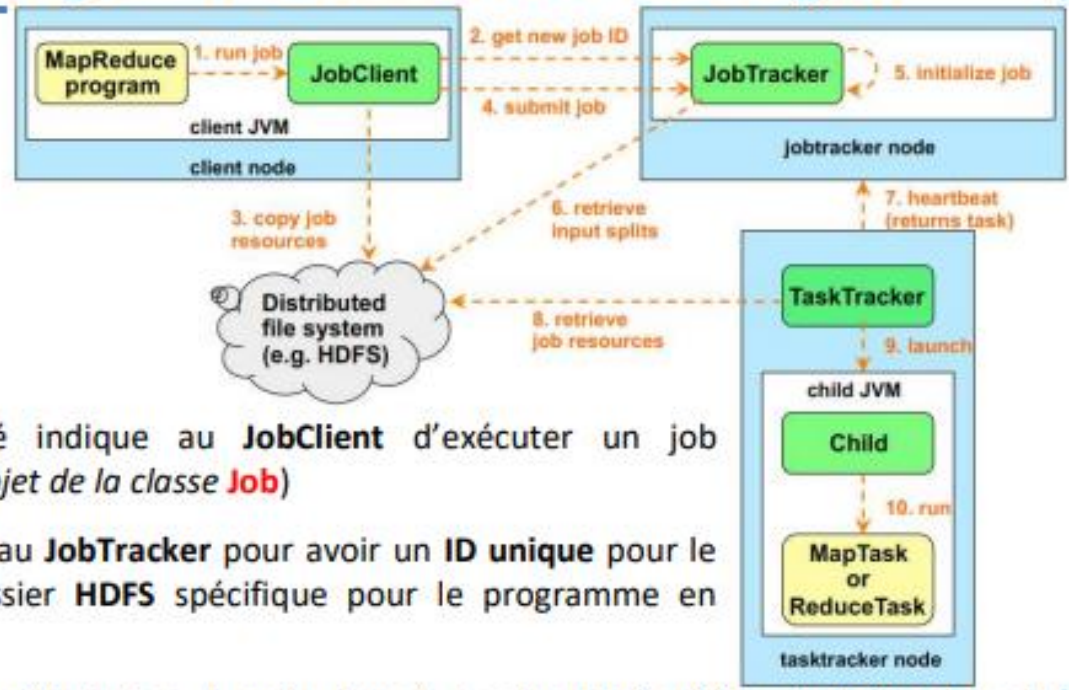


MapReduce: InputFormat

- Les tâches **MapReduce** lisent les fichiers en définissant une classe **InputFormat**.
  - Les tâches **Map** attendent des pairs ***<clé, valeur>***
- Pour lire des fichiers texte de lignes délimitées, Hadoop fournit la classe **TextInputFormat**.
- Elle retourne un pair ***<clé, valeur>*** par ligne
- La **valeur** est le contenu de la ligne
- La **clé** est le décalage par rapport au début de la première ligne.

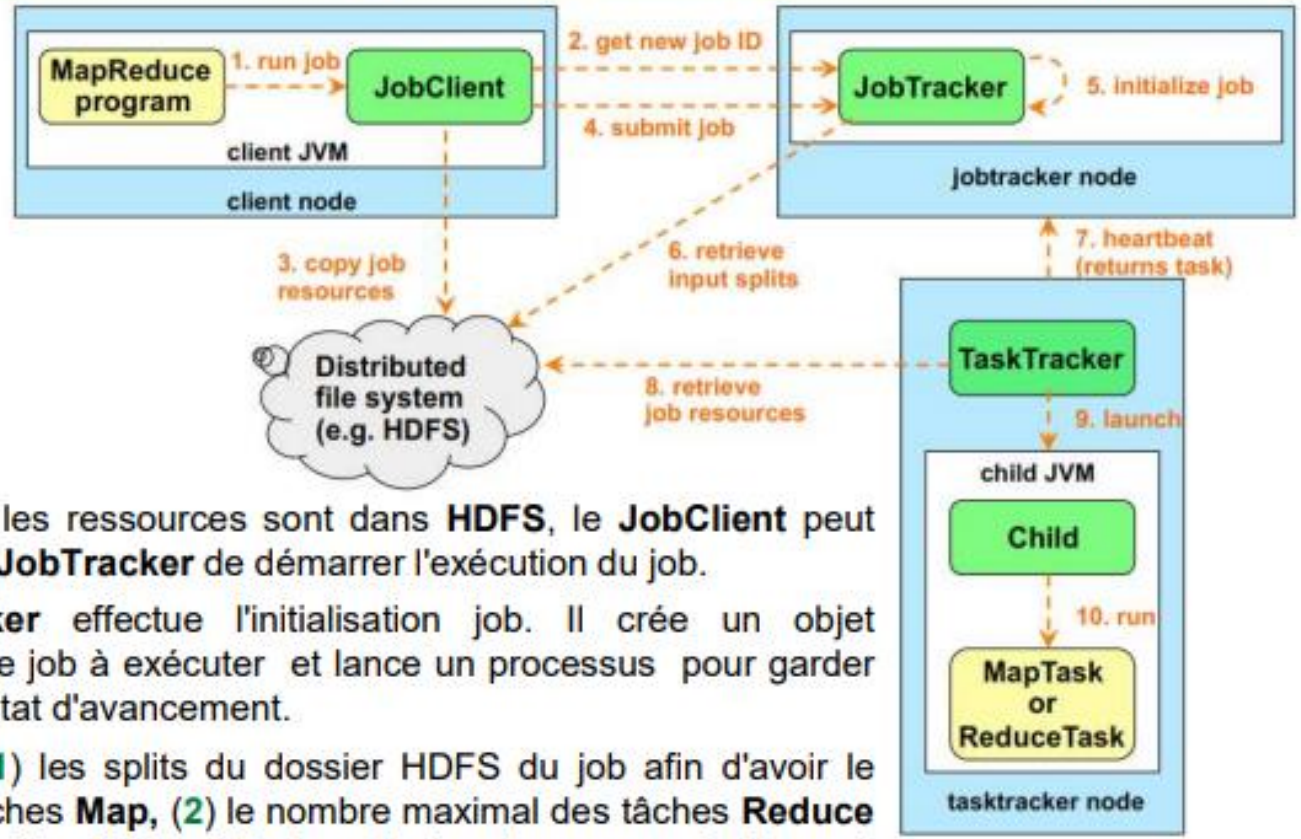


MapReduce: Etapes d'exécution d'un job



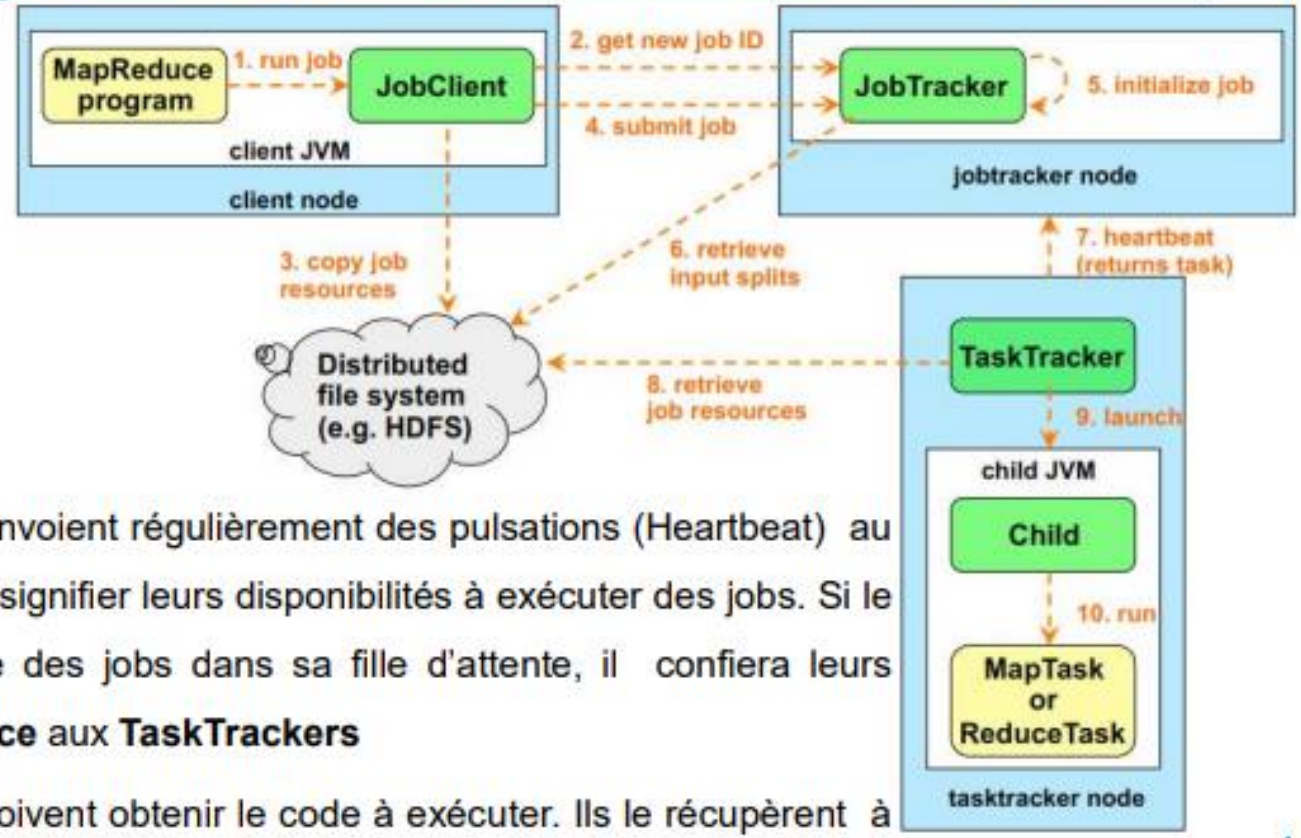
1. Le programme **MapReduce** créé indique au **JobClient** d'exécuter un job **MapReduce** (*via l'utilisation d'un objet de la classe* **Job**)
2. Le **JobClient** envoie une demande au **JobTracker** pour avoir un **ID unique** pour le job. Le **JobTracker** créera un dossier **HDFS** spécifique pour le programme en utilisant son **ID**.
3. Le **JobClient** calcule le nombre de splits du job et copie, dans le dossier **HDFS** créé par le **JobTracker** et nommé d'après l'**ID du job**, les ressources du job, telles qu'un fichier **jar** contenant le code Java des tâches **Map** et **Reduce,** un fichier XML de configuration du *job (classe de Map et Reduce, type de clé et valeur des résultats de Reduce, …)* et les splits calculés.
   Le JAR du job est copié avec une réplication haute (*paramètre* **mapreduce.submit.replication** dans **mapred-site.xml**, par **défaut=10**), de sorte qu'il y est beaucoup de copies à travers le cluster pour les **TaskTrackers**.

MapReduce: Etapes d'exécution d'un job



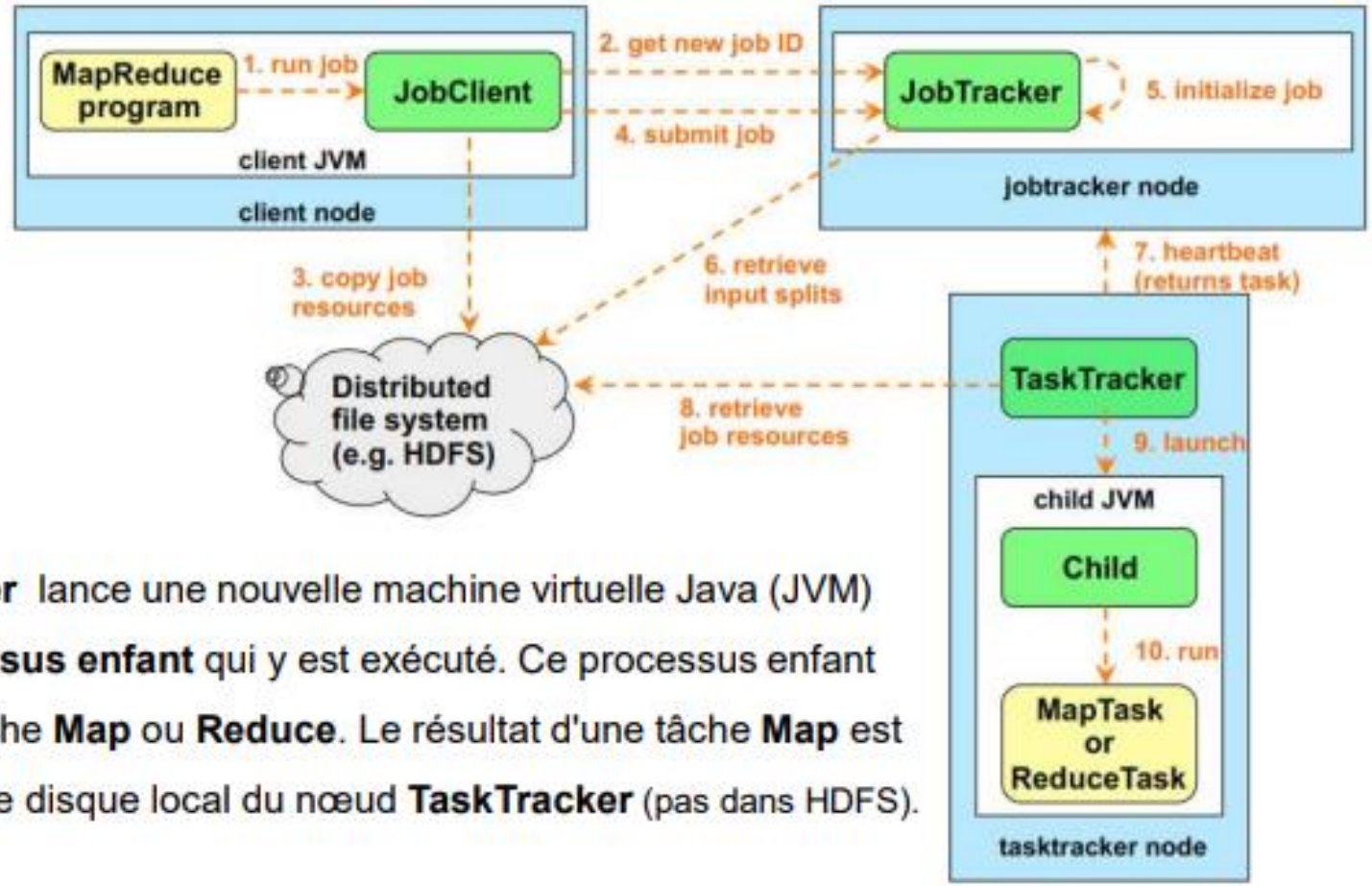
4. Une fois que les ressources sont dans **HDFS**, le **JobClient** peut demander au **JobTracker** de démarrer l'exécution du job.
5. Le **JobTracker** effectue l'initialisation job. Il crée un objet représentant le job à exécuter et lance un processus pour garder trace de son état d'avancement.
6. Il récupère: (1) les splits du dossier HDFS du job afin d'avoir le nombre de tâches **Map,** (2) le nombre maximal des tâches **Reduce** à créer (paramètre **Mapred.Reduce.tasks** dans **mapred-site.xml, par défaut=1** càd 100% de la taille du cluster)

MapReduce: Etapes d'exécution d'un job



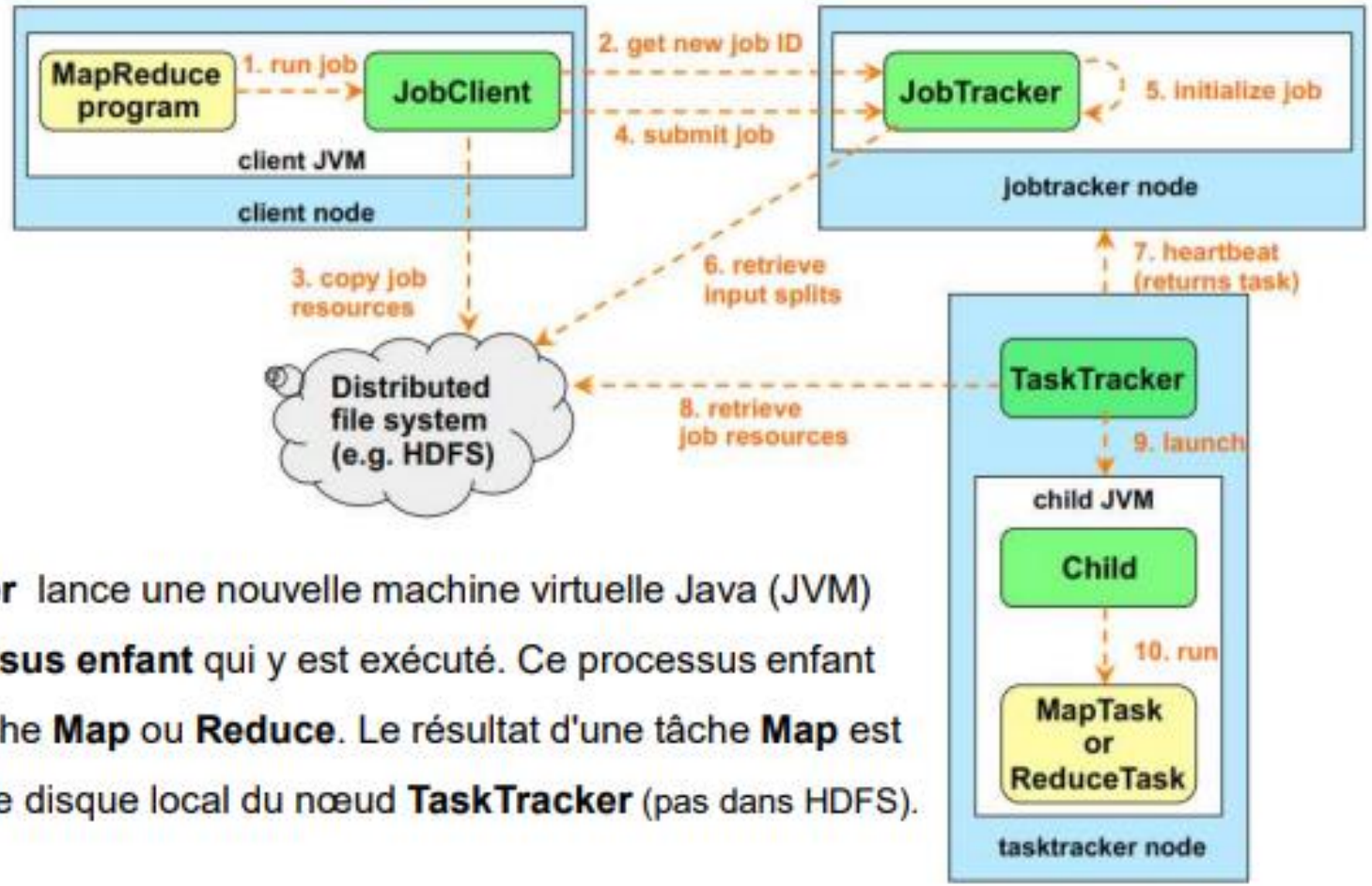
7. Les **TaskTrackers** envoient régulièrement des pulsations (Heartbeat) au **JobTracker** pour lui signifier leurs disponibilités à exécuter des jobs. Si le **Jobtracker** possède des jobs dans sa fille d'attente, il confiera leurs tâches **Map** et **Reduce** aux **TaskTrackers**
8. Les **TaskTrackers** doivent obtenir le code à exécuter. Ils le récupèrent à partir du dossier HDFS du job contenant les ressources du job copiées dans l'étape 3.

MapReduce: Etapes d'exécution d'un job



9. Le **TaskTracker** lance une nouvelle machine virtuelle Java (JVM) avec un **processus enfant** qui y est exécuté. Ce processus enfant exécute une tâche **Map** ou **Reduce**. Le résultat d'une tâche **Map** est enregistré sur le disque local du nœud **TaskTracker** (pas dans HDFS).

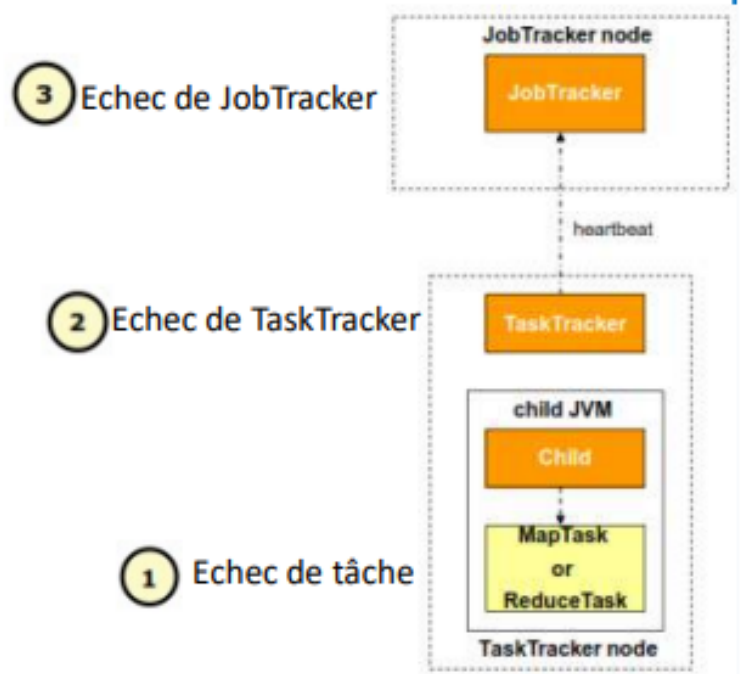
MapReduce: Etapes d'exécution d'un job



9. Le **TaskTracker** lance une nouvelle machine virtuelle Java (JVM) avec un **processus enfant** qui y est exécuté. Ce processus enfant exécute une tâche **Map** ou **Reduce**. Le résultat d'une tâche **Map** est enregistré sur le disque local du nœud **TaskTracker** (pas dans HDFS).

MapReduce: Tolérance aux pannes

- Le moyen principal utilisé par Hadoop pour assurer la tolérance aux pannes consiste à redémarrer des tâches en cas d'échec.
- Le **JobTracker** sait les tâches **Map** et **Reduce** attribuées à chaque **TaskTracker**.
- Si un **TaskTracker** n'envoie pas une pulsation au **JobTracker** pendant une période donnée (par défaut, 10 minute), le JobTracker le considère bloqué.
- En cas d'échec d'un **TaskTracker** pendant l'exécution de ses tâches **Map**, elles seront **toutes** attribuées aux autres **TaskTrackers**.



- En cas d'échec d'un **TaskTracker** pendant l'exécution de ses tâches **Reduce**, les autres **TaskTrackers** ré-exécuteront **seulement** les tâches **Reduce en cours** sur le **TaskTracker** en échec.

MapReduce 1: Maitre/Esclave

- Dans l'architecture **MapReduce**, l'exécution de jobs est contrôlée par deux types de processus:
  - Le **JobTracker**, l'unique maître, qui coordonne et attribue toutes les tâches **Map** et **Reduce** exécutées sur le cluster,
  - Un certain nombre de **TaskTrackers**, processus subordonnés, qui exécutent les tâches assignées et rendent compte périodiquement de leur avancement au **JobTracker**.

MapReduce 1: Responsabilités du JobTracker

- Dans **MapReduce 1**, le **JobTracker** est chargé de **deux responsabilités** distinctes:

1. **Gestion des ressources et ordonnancement des jobs** de calcul dans le cluster, ce qui implique de gérer la liste des **nœuds actifs**, la liste des **Map** et **Reduce** et des emplacements (**Slots**) disponibles et occupés, ainsi que d'affecter les emplacements disponibles aux jobs et tâches appropriés, en fonction de la politique d'ordonnancement sélectionnée (FIFO, ….).

2. **Coordination de toutes les tâches** en cours d'exécution sur un cluster, ce qui implique de donner aux **TaskTrackers** l'ordre de démarrer des tâches **Map** et **Reduce**, de surveiller l'exécution des tâches, de relancer les tâches ayant échoué, d'exécuter **spéculativement** des tâches lentes, … etc.

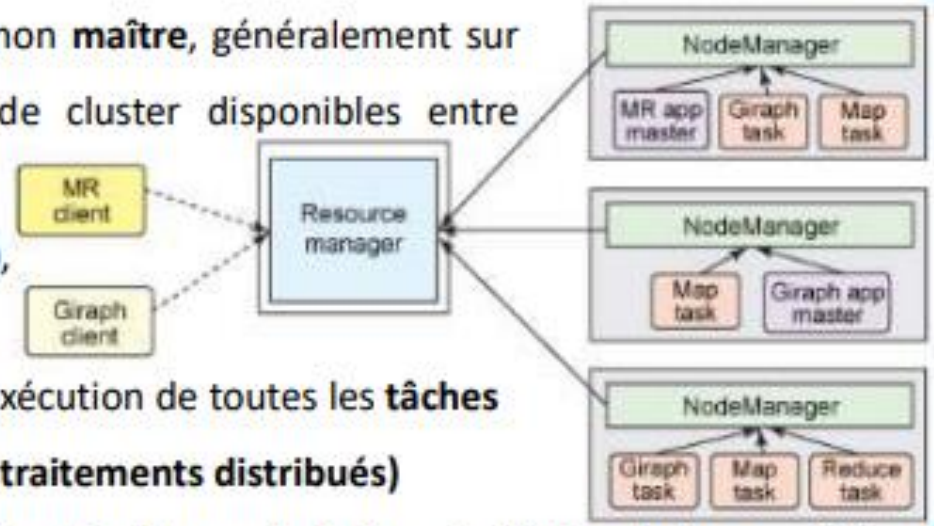
MapReduce 1 – Limites

- Les limitations les plus sérieuses de **MapReduce** V1 sont:
  - **L'évolutivité**: Les **responsabilités** du **JobTracker** posaient d'importants problèmes d'évolutivité: **MapReduce1** rencontre des goulots d'étranglement d'évolutivité lorsqu'elle s'approche de 4 000 nœuds et 40 000 tâches, ceci est dû au fait que le **JobTracker** gère à la fois les **jobs** et leurs **tâches**.

    YARN surmonte ces limitations grâce à son architecture de **Ressource Manager/Application Master** séparés: il est conçu pour évoluer jusqu'à 10 000 nœuds et 100 000 tâches.
  - **L'utilisation des ressources**: Les **TaskTrackers** dans **MapReduce 1** sont configurés avec un nombre de **Slots** divisés **séparément** en slots pour les tâches **Map** et d'autres pour les tâches **Reduce**. **Yarn** utilise des **conteneurs** (**container**) pouvant être utilisés pour exécuter n'importe quel type de tâche.
  - N'est pas **Multi-tenant**: pas de prise en charge de job non **MapReduce**.

YARN (*Yet Another Resource Negotiator*)

- **MapReduce V1** a subi une refonte complète avec **YARN**, scindant les deux fonctionnalités principales de **JobTracker** (*gestion des ressources et planification / surveillance des jobs*) en démons distincts.
- **Resource Manager (RM)**
  - L'unique **Resource Manager** s'exécute tant que démon **maître**, généralement sur une machine dédiée. Il partage les ressources de cluster disponibles entre diverses applications (jobs) concurrentes.
  - Lorsqu'un utilisateur soumet une **application (job)**, une instance d'un processus léger appelé **ApplicationMaster** est démarrée pour coordonner l'exécution de toutes les **tâches de cette application (selon plusieurs paradigmes de traitements distribués)**

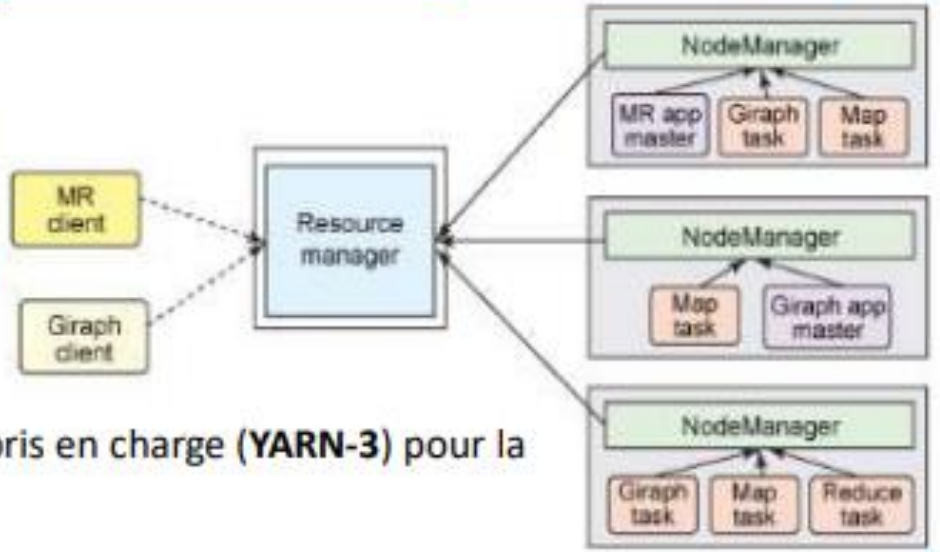


(*surveillance, redémarrage de tâches ayant échoué, exécution spéculative de tâches lentes, ….*) Ces responsabilités étaient précédemment attribuées au **JobTracker** pour **tous les jobs**.

  - L'**ApplicationMaster** et les **tâches** de son application s'exécutent dans des **conteneurs** de ressources contrôlés par les **NodeManagers**

YARN (Yet Another Resource Negotiator)

- **NodeManager (NM)**
  - Il dispose d'un nombre de **conteneurs** de ressources créés dynamiquement. La capacité d'un conteneur dépend de la quantité de ressources qu'il contient (*mémoire, CPU, disque, E/S réseau*)



  - Actuellement, seuls la mémoire et le processeur sont pris en charge (**YARN-3**) pour la la configuration des conteneurs..
  - Le **NodeManager** démarre les conteneurs, les surveille et communique leur statuts au **ResourceManager**
  - Le nombre de conteneurs sur un nœud dépend de paramètres de la configuration et le nombre total de ressources du nœud (*nombre total de processeurs et taille mémoire*)
  - **ApplicationMaster** peut **exécuter** **n'importe quel type de tâche** dans un **conteneur**. Par exemple, **ApplicationMaster MapReduce** demande à un conteneur de lancer une tâche **Map** ou **Reduce**, tandis qu'un **ApplicationMaster Giraph** demande à un conteneur d'exécuter une tâche **Giraph**.

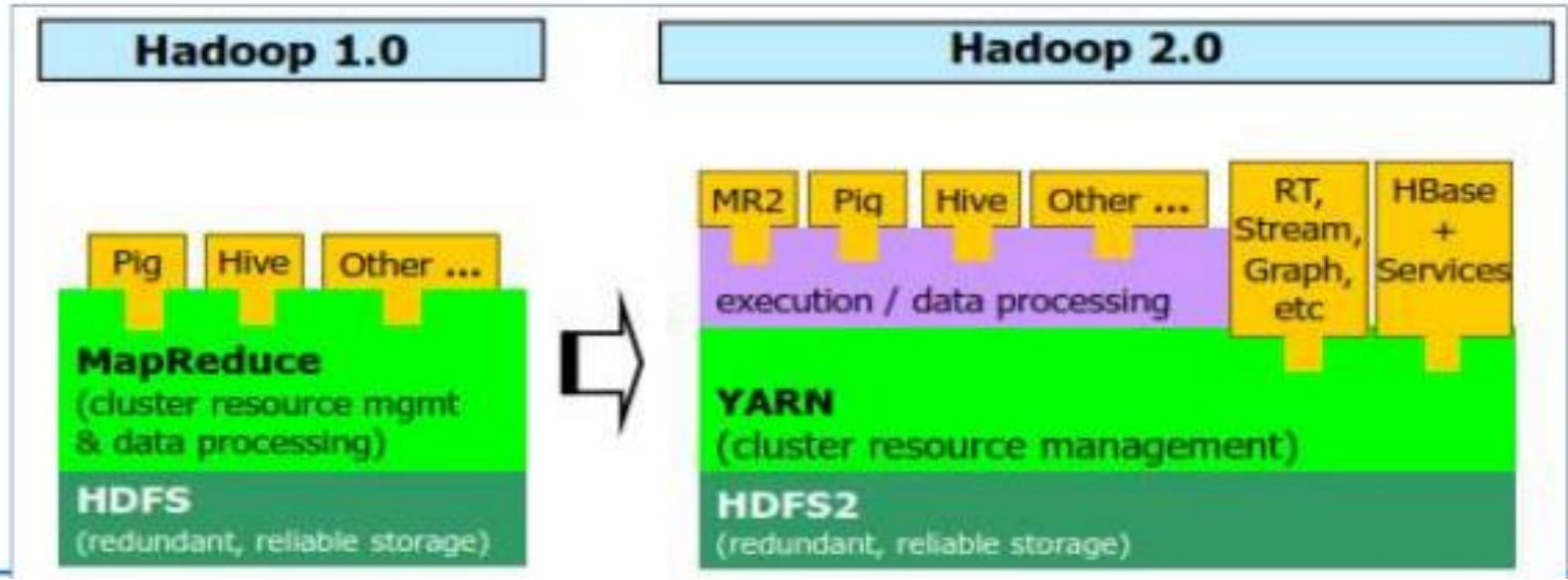
YARN (Yet Another Resource Negotiator)

Hadoop v1 Vers Hadoop v2

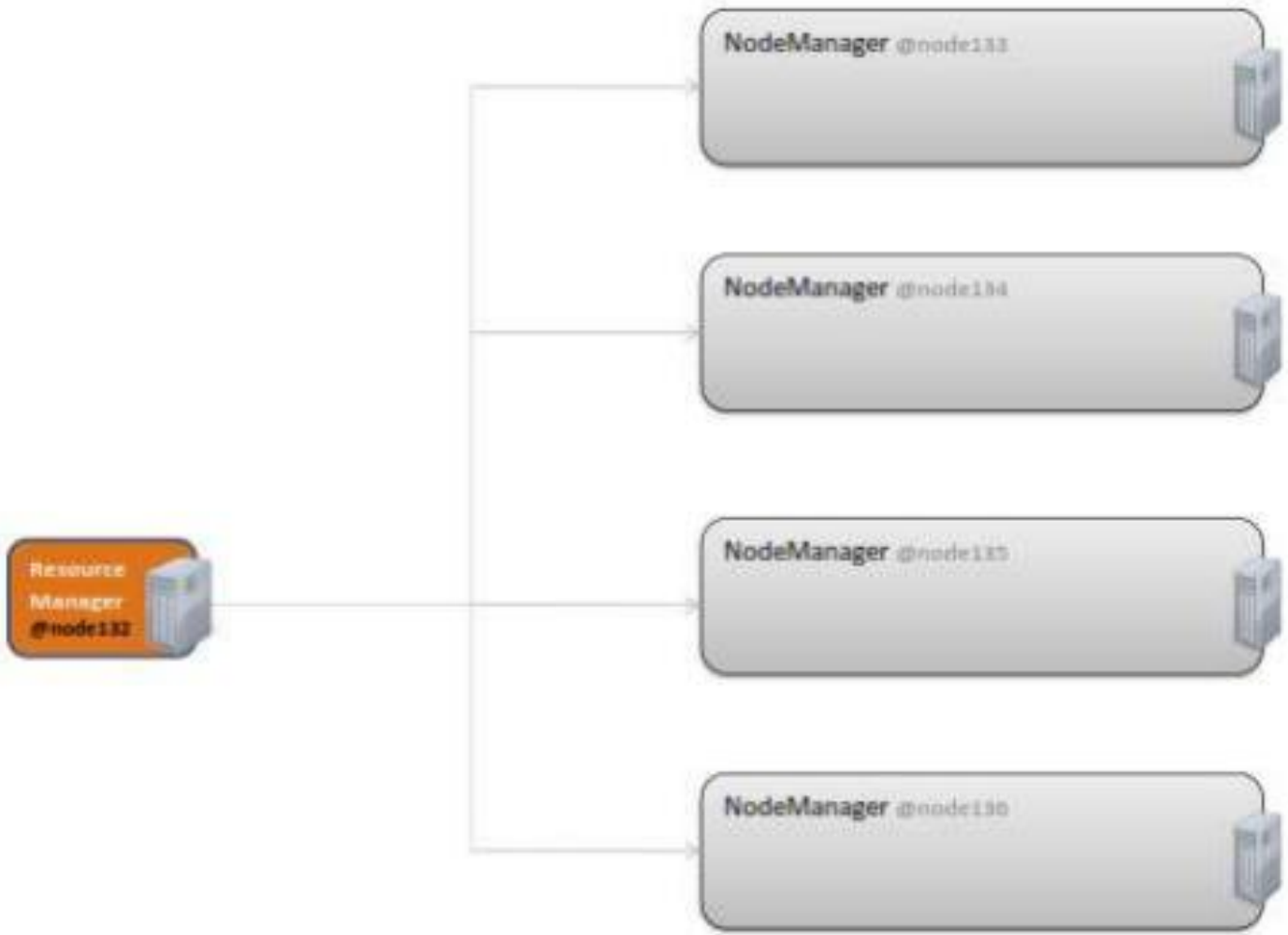
MapReduce 1	YARN
Jobtracker	Resource manager, application master,
Tasktracker	Node manager
Slot	Container

Système à usage unique
Traitement par lots

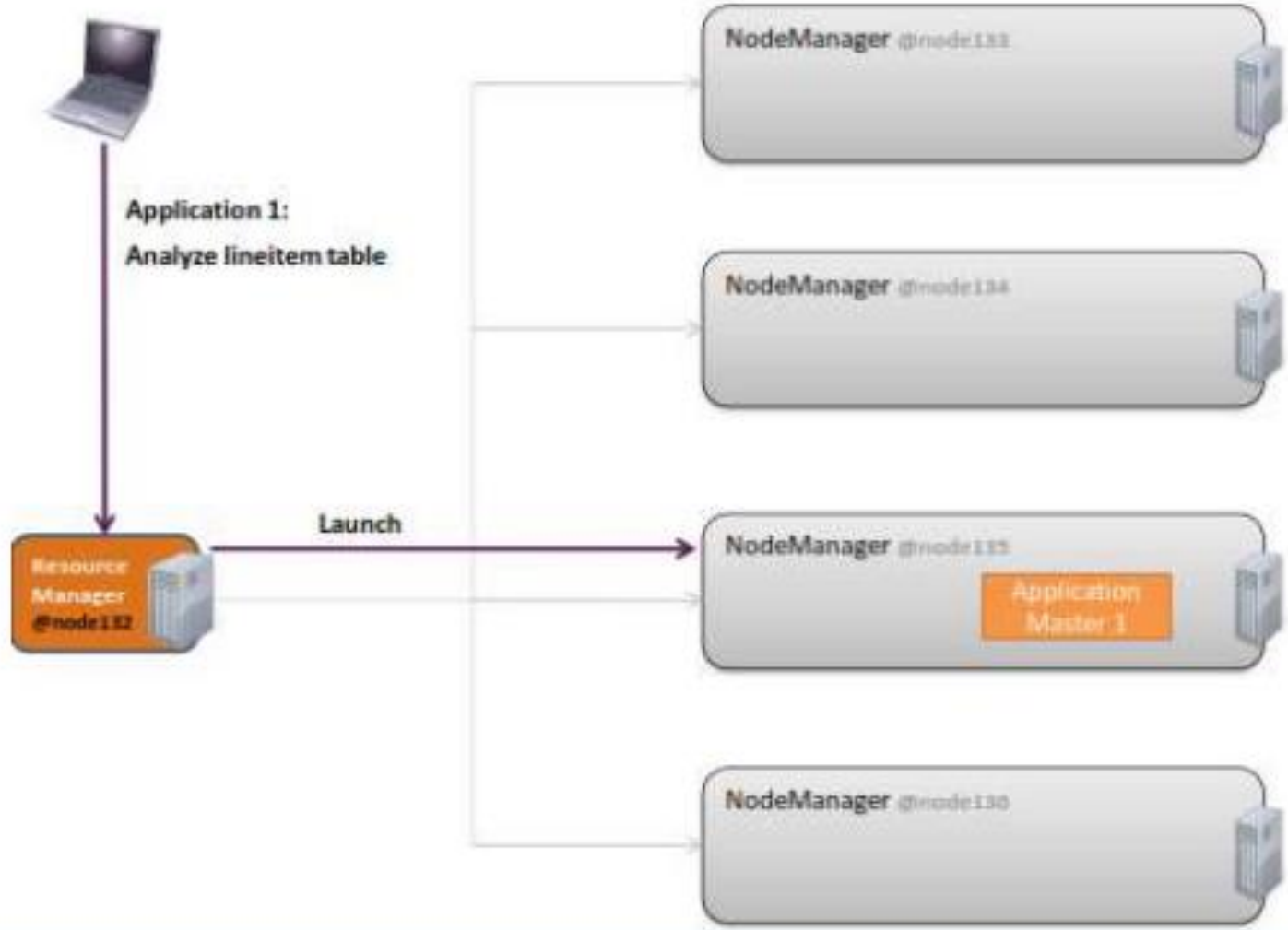
Plateforme à usages multiples
Traitement par lots, interactif, en ligne, streaming



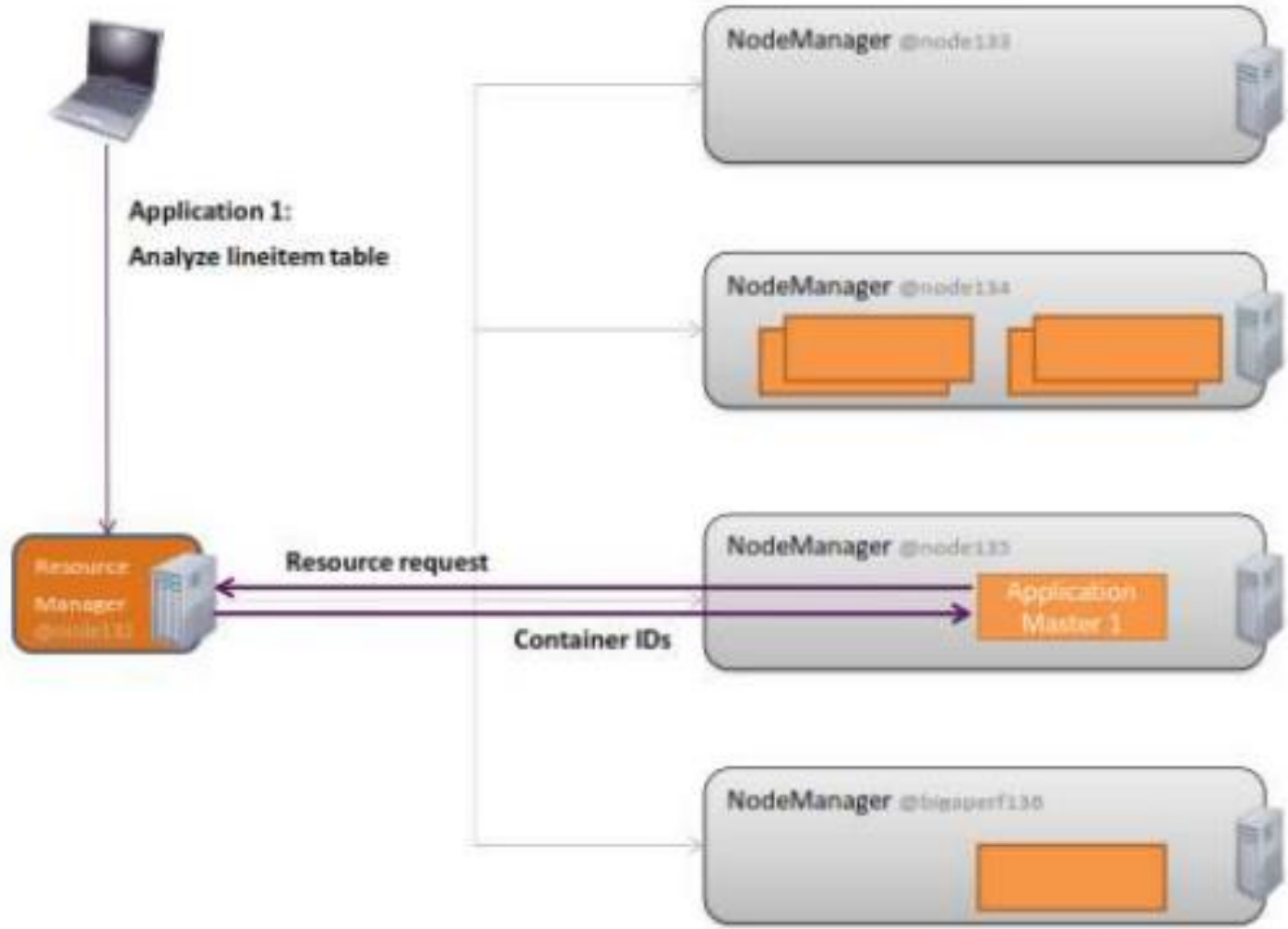
YARN: Exécuter une application dans Yarn (1 / 7)



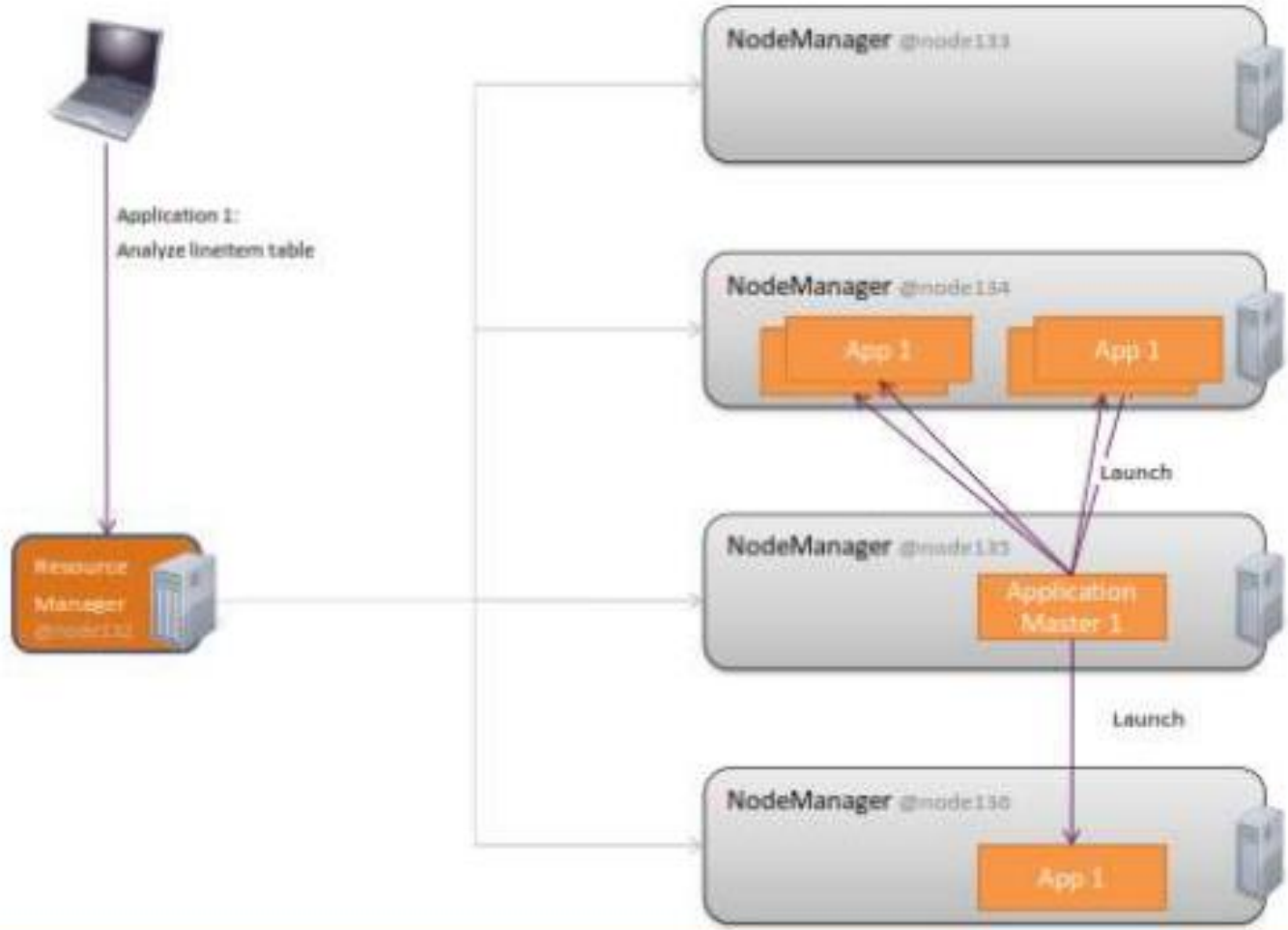
YARN: Exécuter une application dans Yarn (2 / 7)



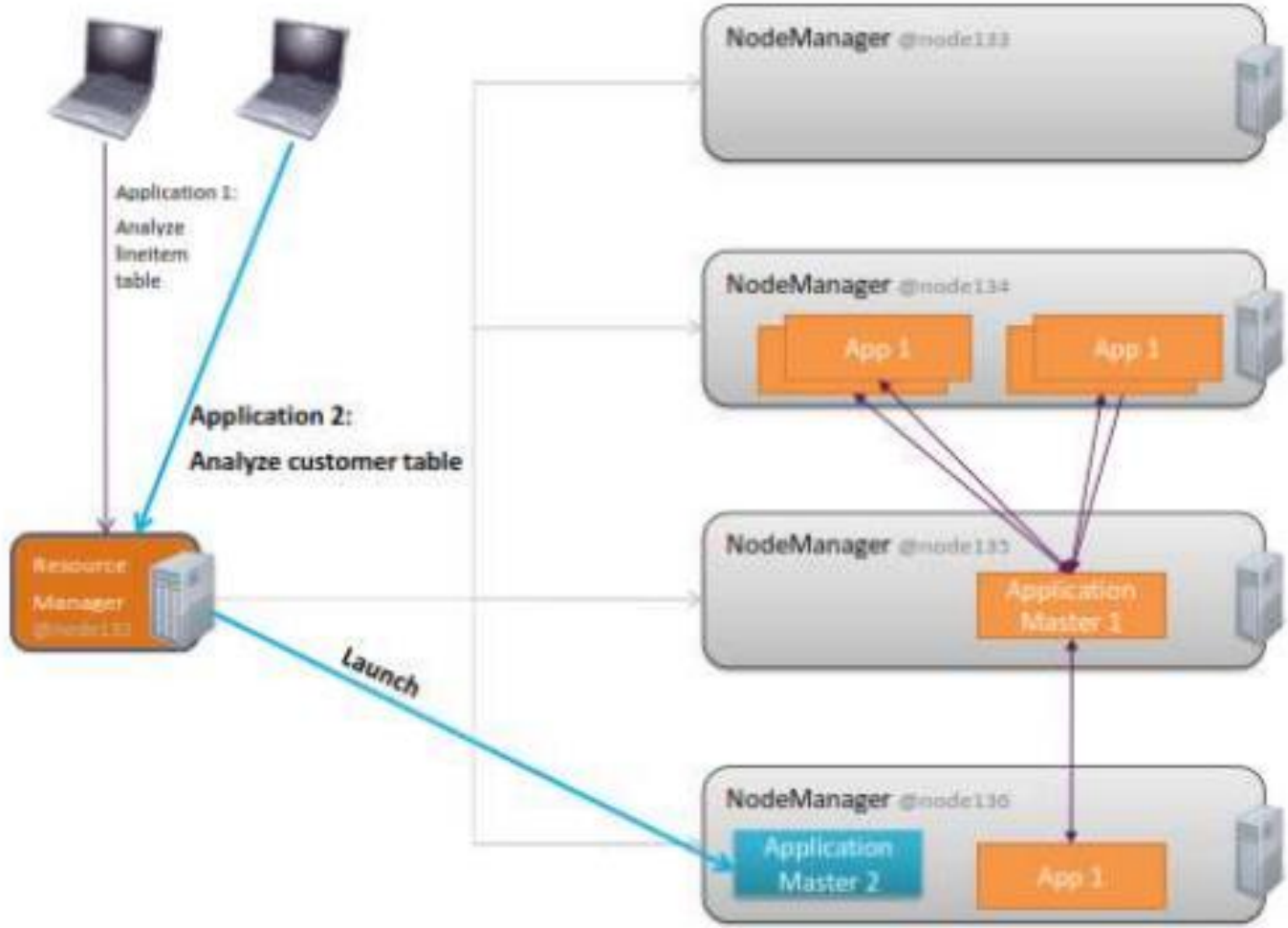
YARN: Exécuter une application dans Yarn (3 / 7)



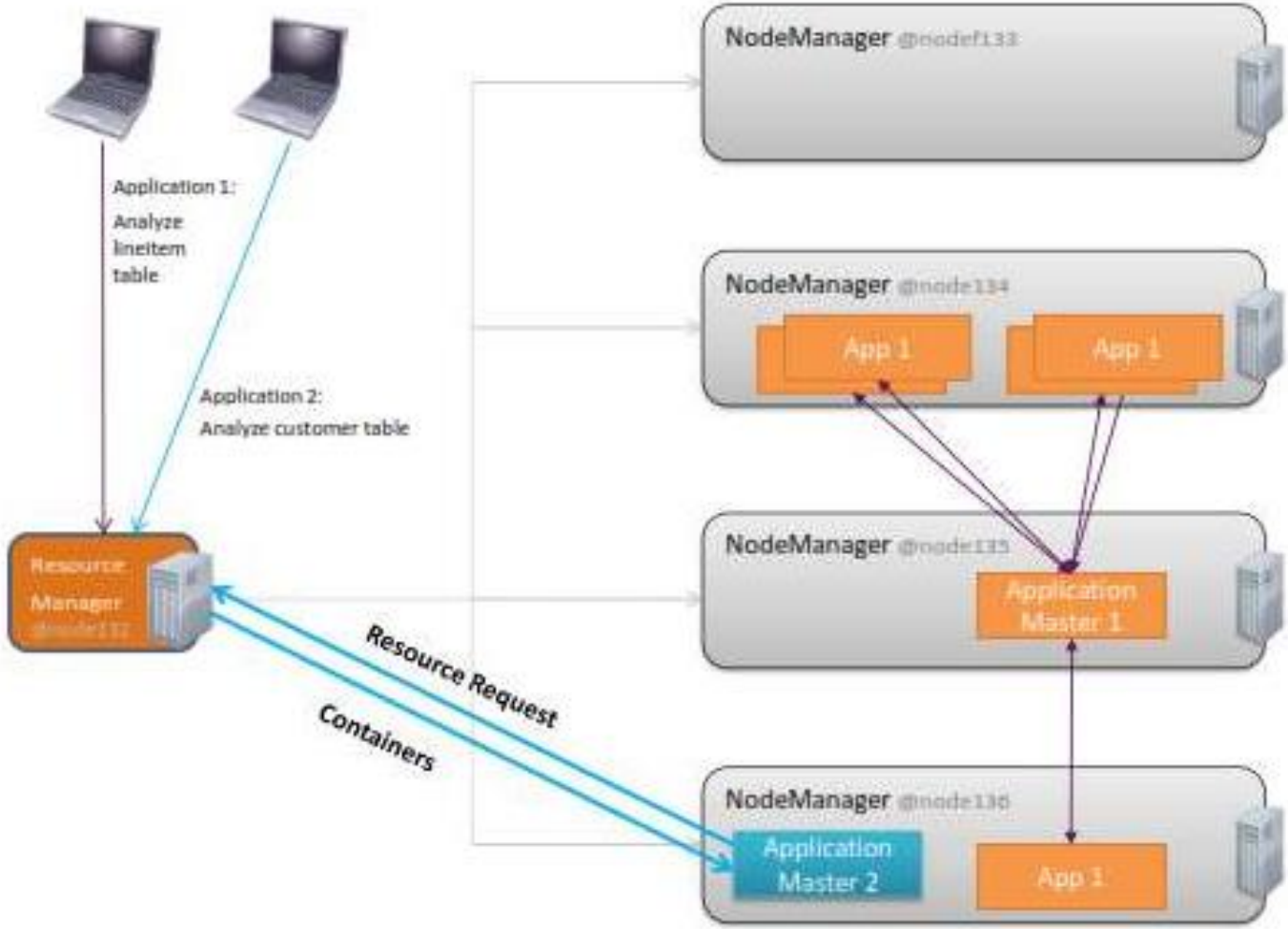
YARN: Exécuter une application dans Yarn (4 / 7)



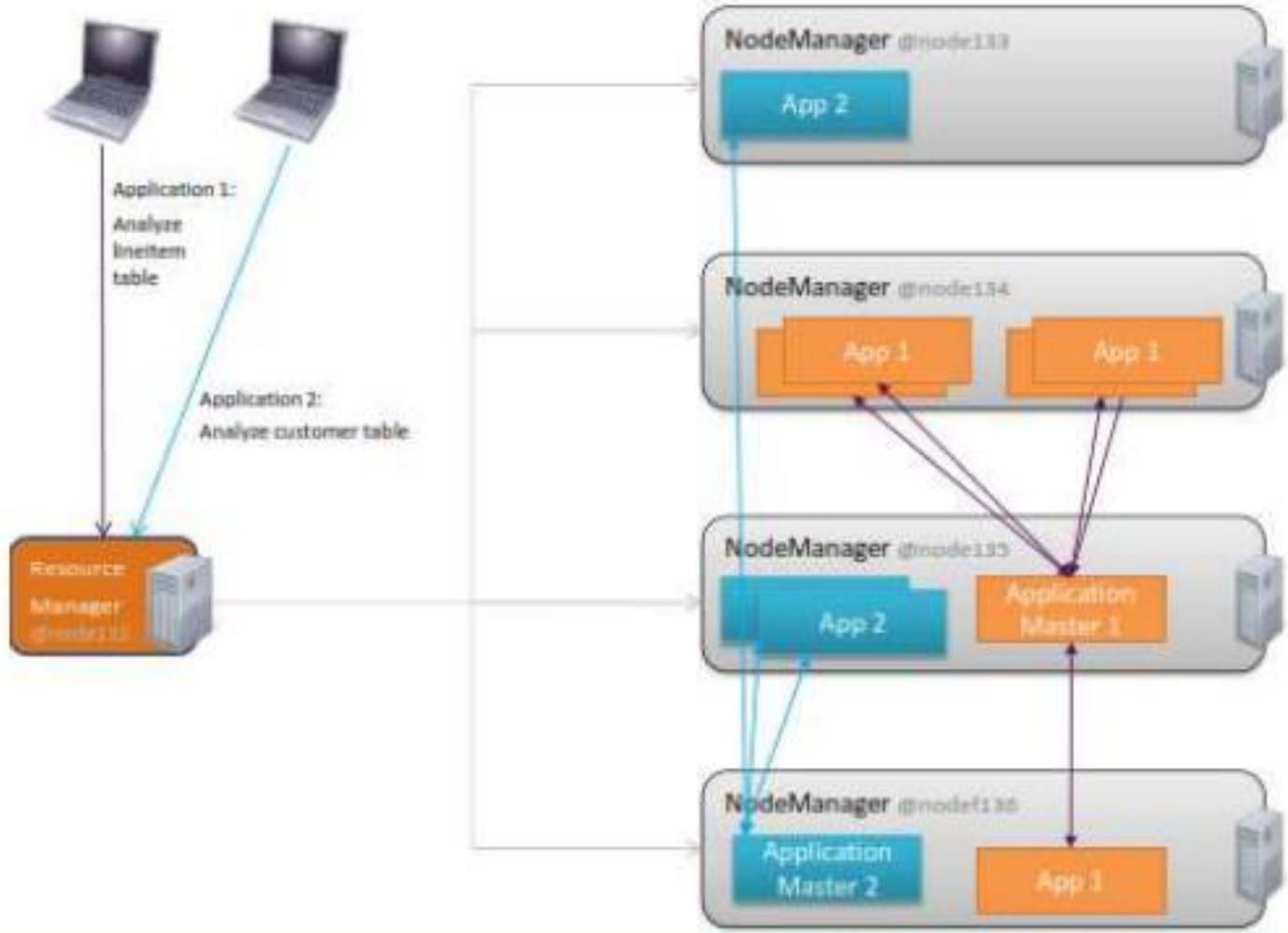
YARN: Exécuter une application dans Yarn (5 / 7)



YARN: Exécuter une application dans Yarn (6 / 7)



YARN: Exécuter une application dans Yarn (7 / 7)



Critique de MapReduce

A l'ère numérique actuelle, l'approche MapReduce peut devenir le mode normal de traitement des données et Hadoop l'outil par défaut du traitement de données.

Mais MapReduce présente plusieurs limites :

1. Est un langage de **très bas niveau** = très proche de la machine ;

=> Il impose au développeur de savoir interagir avec le cluster : ce qui peut être très difficile pour un développeur nouveau dans le monde du traitement parallèle, ou pour des utilisateurs non informaticiens.

Critique de MapReduce

2. Si on souhaite mettre en place une solution complexe, il est nécessaire d'enchaîner une série de jobs MapReduce et de les exécuter séquentiellement.

=> Il est difficile d'exprimer des opérations complexes en n'utilisant que MapReduce.

Critique de MapReduce

3. Après une opération Map ou Reduce, le résultat est écrit sur disque.

Les Mappers et Reducers communiquent entre eux à travers les données écrites sur disque.

=> Opérations de lectures/écritures **très coûteuses en temps**!

=> MapReduce est une **bonne solution** pour les traitements à **passe unique**.

=> MapReduce **n'est pas la meilleure solution** pour les traitements **à plusieurs passes**.

Solutions

Afin de résoudre ces limites, plusieurs langages et frameworks ont été développés notamment par Apache.

Problème 1 : langage de **très bas niveau**

=> Solutions : Langages d'abstraction

Problème 2 : difficulté de résoudre des problèmes relativement complexes

+

Problème 3 : lecture/écriture sur disque

=> Solution : **open source Frameworks**

Solutions

Dans le cadre de ce cours, nous nous intéressons à :

- **Apache Pig et Apache Hive comme** langages d'abstraction
- **Apache Spark** comme open source Framework

Apache Spark

- Framework open source complet et unifié adapté au **traitement batch et temps réel** de divers types de données.

- Développé en 2009 par AMPLab – University of California Berkeley.

- En 2010 il est passé open source sous forme de projet Apache.

Apache Spark

La principale innovation de Spark est qu'il maintient **les résultats intermédiaires en mémoire** plutôt que sur disque : ce qui est très utile en particulier lorsqu'il est nécessaire de travailler **plusieurs fois** sur le même jeu de données.

Le moteur d'exécution de Spark est conçu pour travailler aussi bien en mémoire que sur disque.

Spark essaie de stocker le plus possible en mémoire avant de basculer sur disque.

Il est capable de travailler avec une partie des données en mémoire, une autre sur disque.

Apache Spark

Spark permet de développer des pipelines de traitement de données complexes, à plusieurs étapes, en s'appuyant sur des graphes orientés acycliques (DAG).

Il permet de partager les données en mémoire entre les graphes, de façon à ce que plusieurs jobs puissent travailler sur le même jeu de données.

Apache Spark

- Il permet à des applications sur clusters Hadoop d'être exécutées jusqu'à 100 fois plus vite en mémoire, 10 fois plus vite sur disque.

- Il est écrit en Scala et s'exécute sur la Machine Virtuelle Java (JVM).

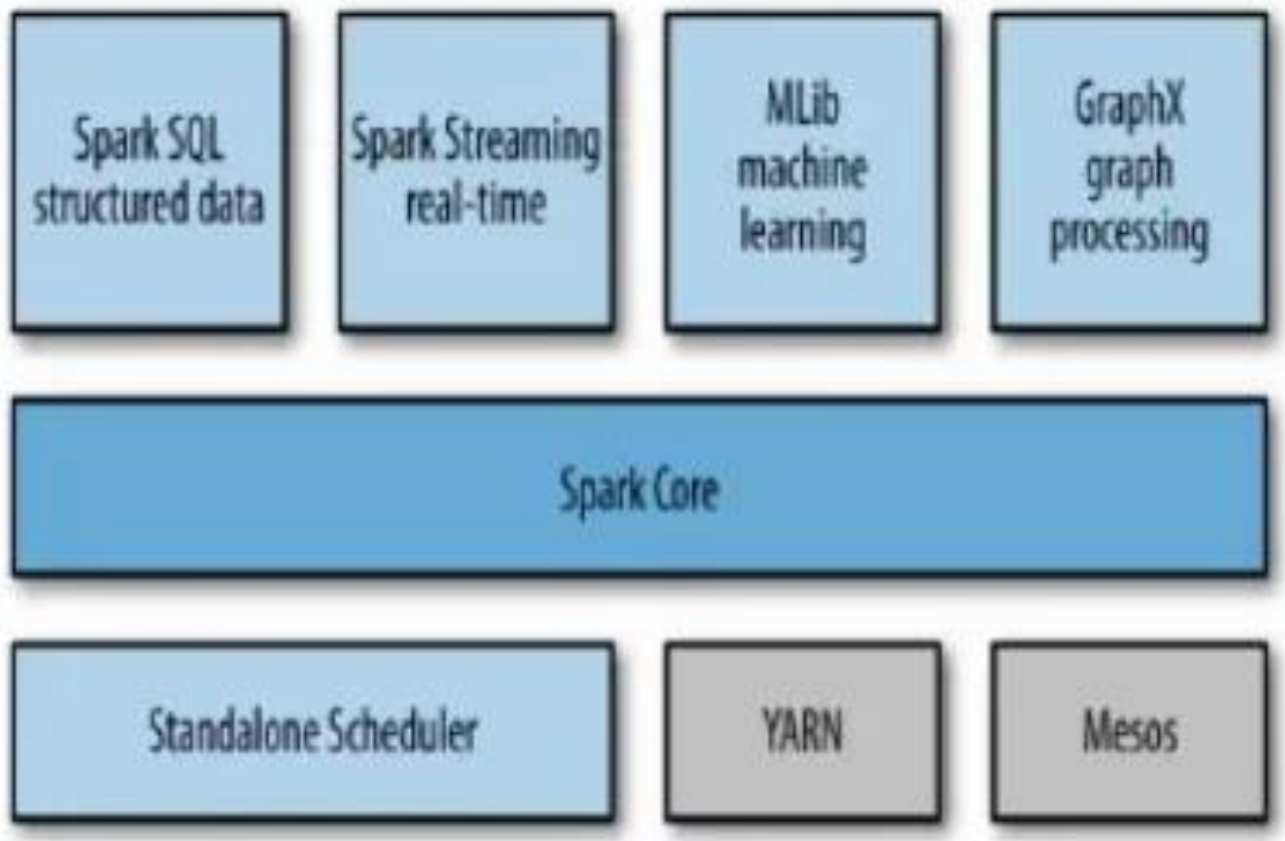
- Il est possible de l'utiliser de façon interactive pour requêter les données depuis un shell.

Apache Spark

Il est possible de déployer des applications Spark sur un cluster Hadoop v1 existant (avec SIMR – Spark-Inside-MapReduce), ou sur un cluster Hadoop v2 YARN.

Spark n'est pas considéré comme un remplaçant de Hadoop mais comme **une alternative au MapReduce d'Hadoop**.

Ecosystème de Spark



Ecosystème de Spark

Spark Core :

- Moteur de base de Spark
- Gestion de tâches
- Gestion de mémoire
- Récupération d'erreurs
- Interaction avec le stockage
- Définition des classes de RDD

Ecosystème de Spark

A part les API principales de Spark, son écosystème contient des librairies additionnelles telles que :

Spark streaming : est utilisé pour le traitement temps-réel des données en flux. Il s'appuie sur un mode de traitement en "micro batch" et utilise Dstream (c'est-à-dire une série de RDD (Resilient Distributed Dataset)).

Spark SQL : Grâce à des requêtes de type SQL, Spark SQL permet d'extraire, transformer et charger des données sous différents formats (JSON, Parquet, base de données) et les exposer pour des requêtes ad-hoc.

Ecosystème de Spark

MLlib : c'est une librairie de machine learning qui contient tous les algorithmes et utilitaires d'apprentissage classiques, comme la classification, la régression, le clustering, en plus des primitives d'optimisation sous-jacentes.

GraphX : c'est une API (en version alpha) pour les traitements de graphes. Elle étend les RDD de Spark en introduisant le Resilient Distributed Dataset Graph, un multi-graphe orienté avec des propriétés attachées aux nœuds et aux arrêtes. GraphX inclut une collection toujours plus importante d'algorithmes et de builders pour simplifier les tâches d'analyse de graphes.

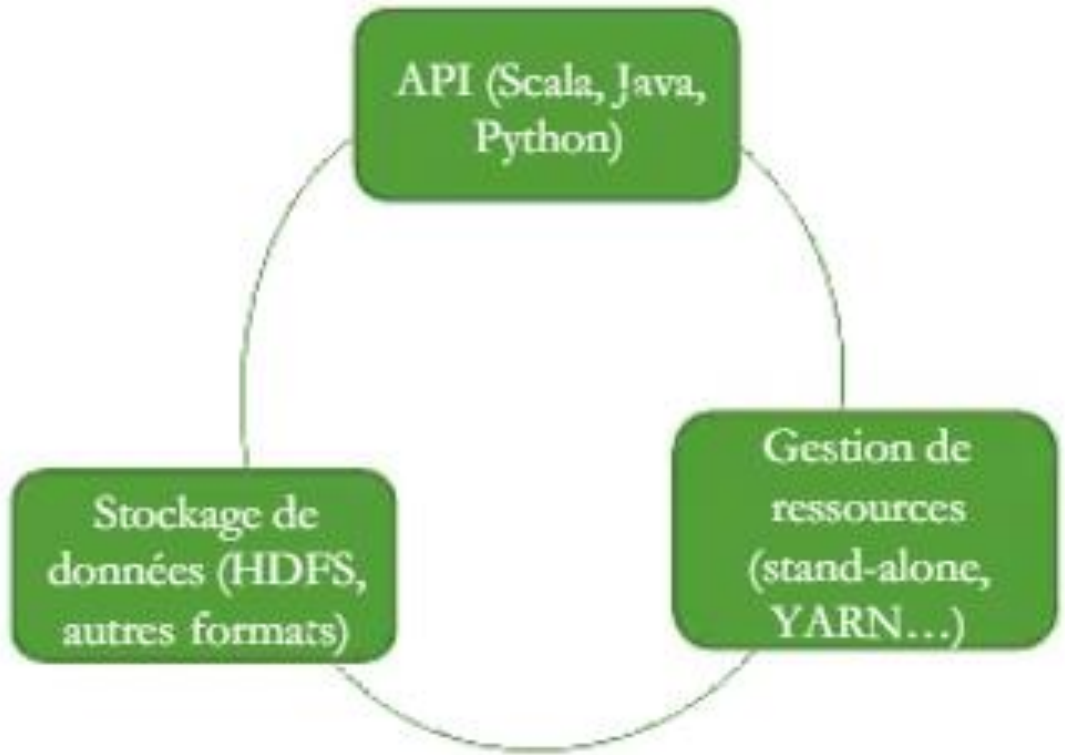
Ecosystème de Spark

Il existe aussi des adaptateurs pour intégration à d'autres produits comme Cassandra (Spark Cassandra Connector) et R (SparkR).

Avec le connecteur Cassandra, il est possible d'utiliser Spark pour accéder à des données stockées dans Cassandra et réaliser des analyses sur ces données.

Architecture de Spark

L'architecture de Spark est basée sur trois composants principaux :



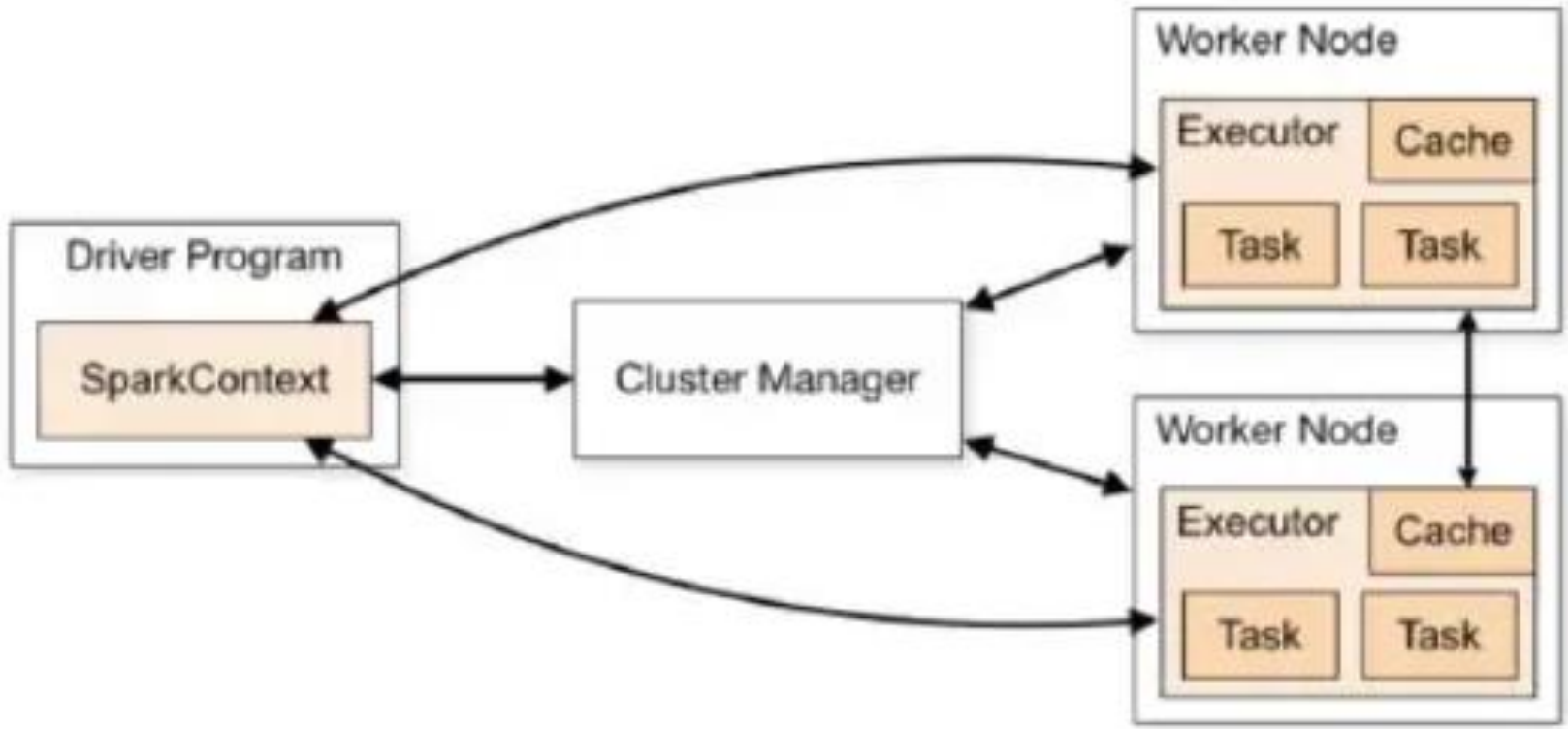
Architecture de Spark

Spark utilise le système de fichiers **HDFS** pour le stockage des données. Il peut fonctionner avec n'importe quelle source de données compatible avec Hadoop, dont HDFS, HBase, Cassandra.

L'API permet aux développeurs de créer des applications Spark en utilisant une API standard. L'API existe en Scala, Java et Python.

Spark peut être déployé comme un serveur autonome ou sur un framework de traitements distribués comme Mesos ou YARN.

Architecture de Spark



Architecture de Spark

Les applications Spark s'exécutent comme un ensemble de processus indépendants sur un cluster, coordonnés par un objet **SparkContext** du programme principal appelé **Driver Program**.

Architecture de Spark

1. Le driver contient une application qui définit un contexte : objet de la classe Sparkcontext. Ce contexte permet d'accéder à toutes les fonctionnalités de Spark.

2. Pour s'exécuter sur un cluster, le SparkContext se connecte au Cluster Manager pour l'allocation de ressources aux applications (le cluster manager peut être soit un gestionnaire standalone de Spark, soit YARN ou Mesos).

3. Une fois connecté, Spark lance des executors sur les nœuds du cluster : des processus qui lancent des traitements et stockent les données pour les applications.

Architecture de Spark

5. Spark envoie ensuite le code de l'application (Jar ou ficher python) aux executors.

6. SparkContext envoie ensuite les Tasks aux executors pour qu'ils les lancent.

7. Un executor peut exécuter plusieurs tâches grâce aux threads.

Resilient Distributed Dataset

- Un concept au cœur du framework Spark.
- Collection d'éléments **découpables** par Spark et **traitables** de manière distribuée.
- Il peut porter tout type de donnée et est stocké par Spark sur différentes partitions.