

Intégration et qualité de données

Master Informatique – 1^{ère} année

Mourad Ouziri
mourad.ouziri@u-paris.fr

Maître de conférences
Université de Paris



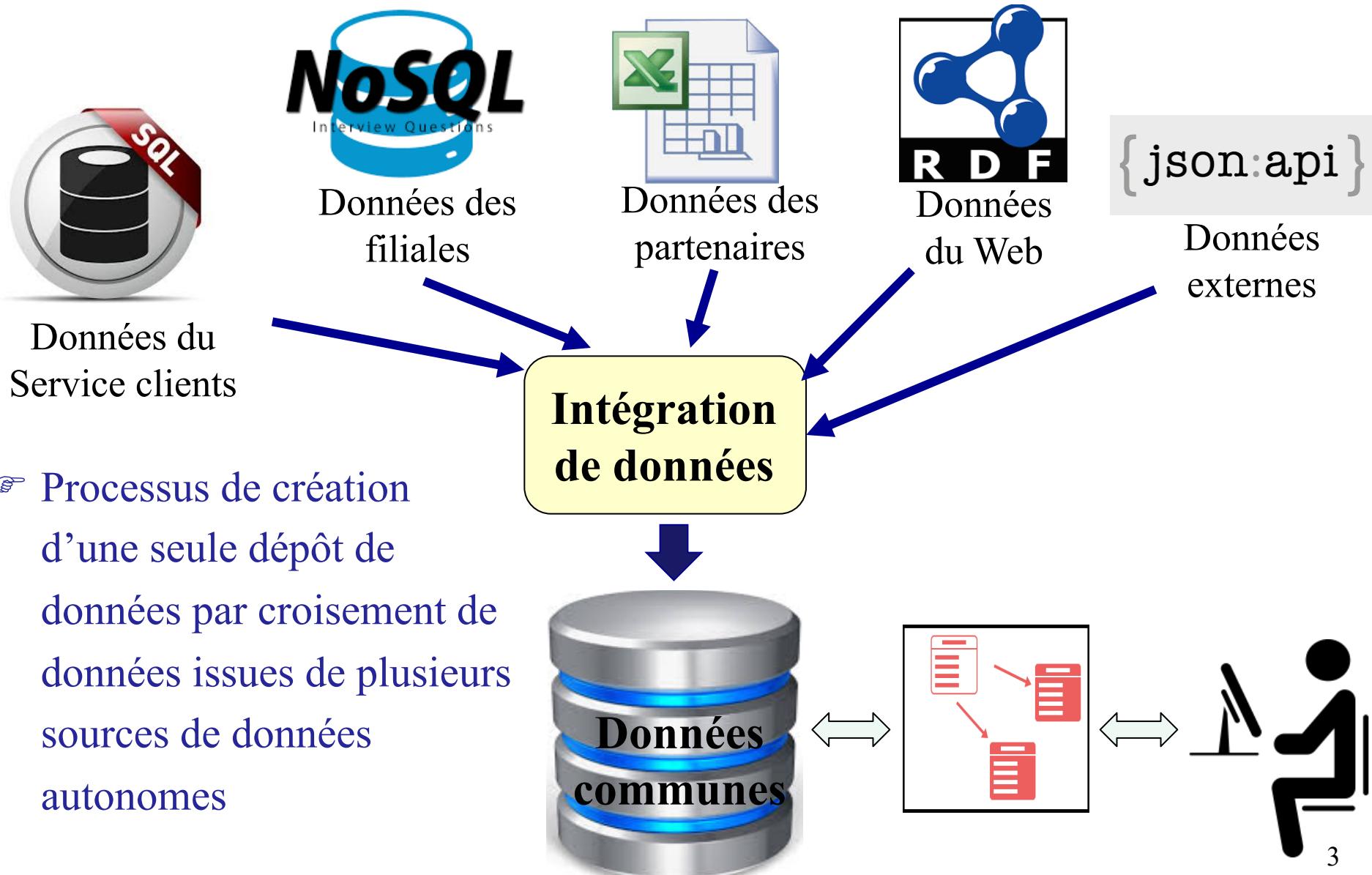
Partie 1 (suite) :

Intégration de données en environnement
Big Data

avec Apache Spark

Intégration de données

Définition



Introduction

Big data

☞ Qu'est ce que le Big Data ?

Le big data désigne des ensembles de données issus de multiples sources hétérogènes constituant une très grande quantité de données difficilement traitable avec les outils classiques de gestion de base de données même en utilisant les machines les plus puissantes [*wikipedia complétée, sept. 2018*]

Introduction

Volume de données

☞ Unités de mesures du volume de données

signe	préfixe	facteur	exemple représentatif
k	kilo	10^3	une page de texte
M	méga	10^6	vitesse de transfert par seconde
G	giga	10^9	DVD, clé USB
T	téra	10^{12}	disque dur
P	péta	10^{15}	
E	exa	10^{18}	FaceBook, Amazon, Google
Z	zetta	10^{21}	internet tout entier depuis 2010

40 Zo : volume de données manipulés par les machines en 2020

Introduction

Volume de données

☞ Quelques exemples de volumétries de données

Internet : Google en 2015 : 10 Eo, Facebook en 2018 : 1 Eo de données (7 Po de nouvelles données par jour), Amazon : 1 Eo

BigScience : télescopes (1 Po/jour), CERN (2 Po lus et écrits/jour, 280 Po de stockage)

IoT : A380 génère 1.6 Go par vol (300.000 capteurs), AirFrance collecte 9 To par an chaque année, 5 milliards d'appareils IoT en 2015 => 25 milliards en 2020 (rapport *Gartner*)

☞ Sources de données : les données sont produites de partout...

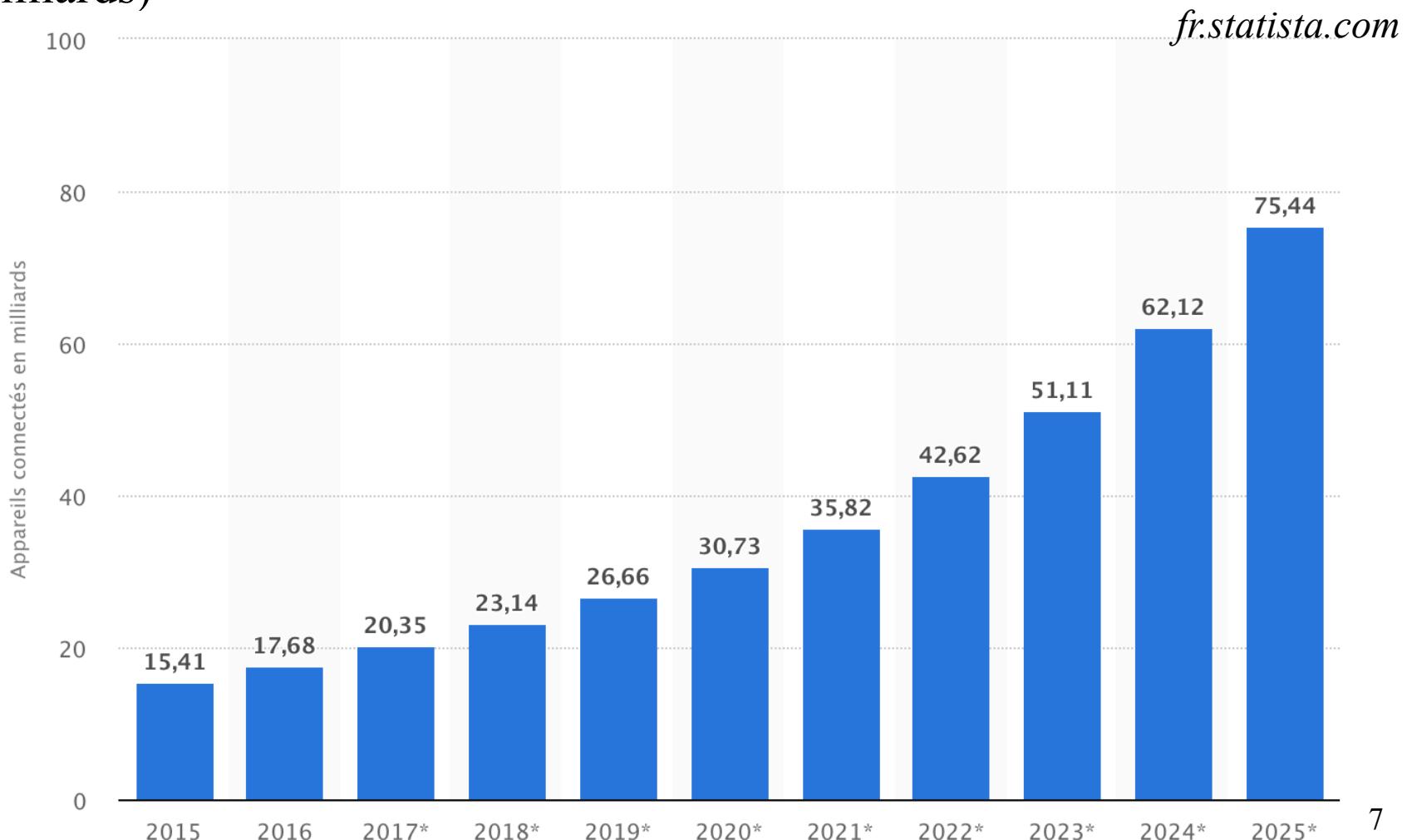
téléphones portables, réseaux sociaux, transactions e-commerce, médias, objets connectés, open data, ...

☞ Enfin, c'est le déluge !!!

Introduction

Volume de données IoT

- 👉 Nombre d'appareils connectés dans le monde de 2015 à 2025 (en milliards)



Introduction

Volume de données

☞ Les outils de stockage et de traitement de données ainsi que les machines les plus puissantes ne sont plus en mesure de traiter ces quantités de données pharamineuses en temps raisonnable



Introduction

Limites des machines classiques

- ☞ Une seule machine ne peut plus stocker les données ni fournir des opérations de lecture et écriture avec un débit acceptable, encore moins réaliser des traitements de recherche et de calculs
- ☞ Illustration : temps de chargement de données volumineuses...

Temps de chargement de 1Eo avec un disque SSD (6Go/s)

=

plus de 5 ans !!!



Introduction

Volume de données

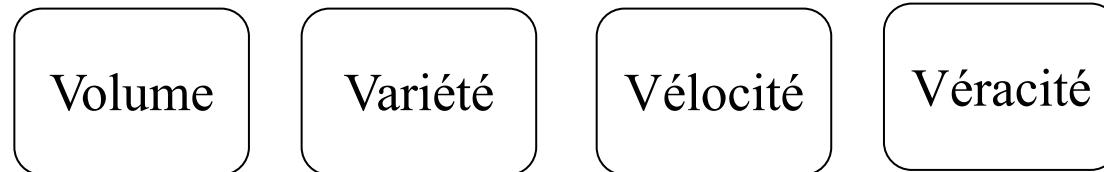
- ☞ Seulement 0.5% des données étaient analysées en 2012 (*the Guardian*)
- ☞ 80% de données inexploitées en 2012, 65% en 2020 (IDC)
- ☞ Alors que la donnée est devenue l'or noir de l'économie et constitue un capital immatériel des entreprises



Introduction

4V du Big Data

- Caractéristiques des données en Big Data (4V !) :



- Volume : les organisations sont submergées de données, toutes sont potentiellement utiles !
- Variété : les données sont issues de multiples sources de stockage (bases de données, fichiers, API, etc.) ayant des modèles (tabulaire, json, texte, images, etc.), schémas (adresse, $fn+ln$), des formats (date, tél, png, jpeg), encodages de valeurs (caractères, sexe), unités de mesure (distance, poids) et vocabulaires différents
- Vélocité : de grandes quantités de données arrivent en temps réel et doivent être traités immédiatement pour certains cas d'usage (détection de fraude/intrusion, trafic routier, etc.)
- Véracité : porte sur la confiance (degré de vérité) pouvant être accordée à la donnée

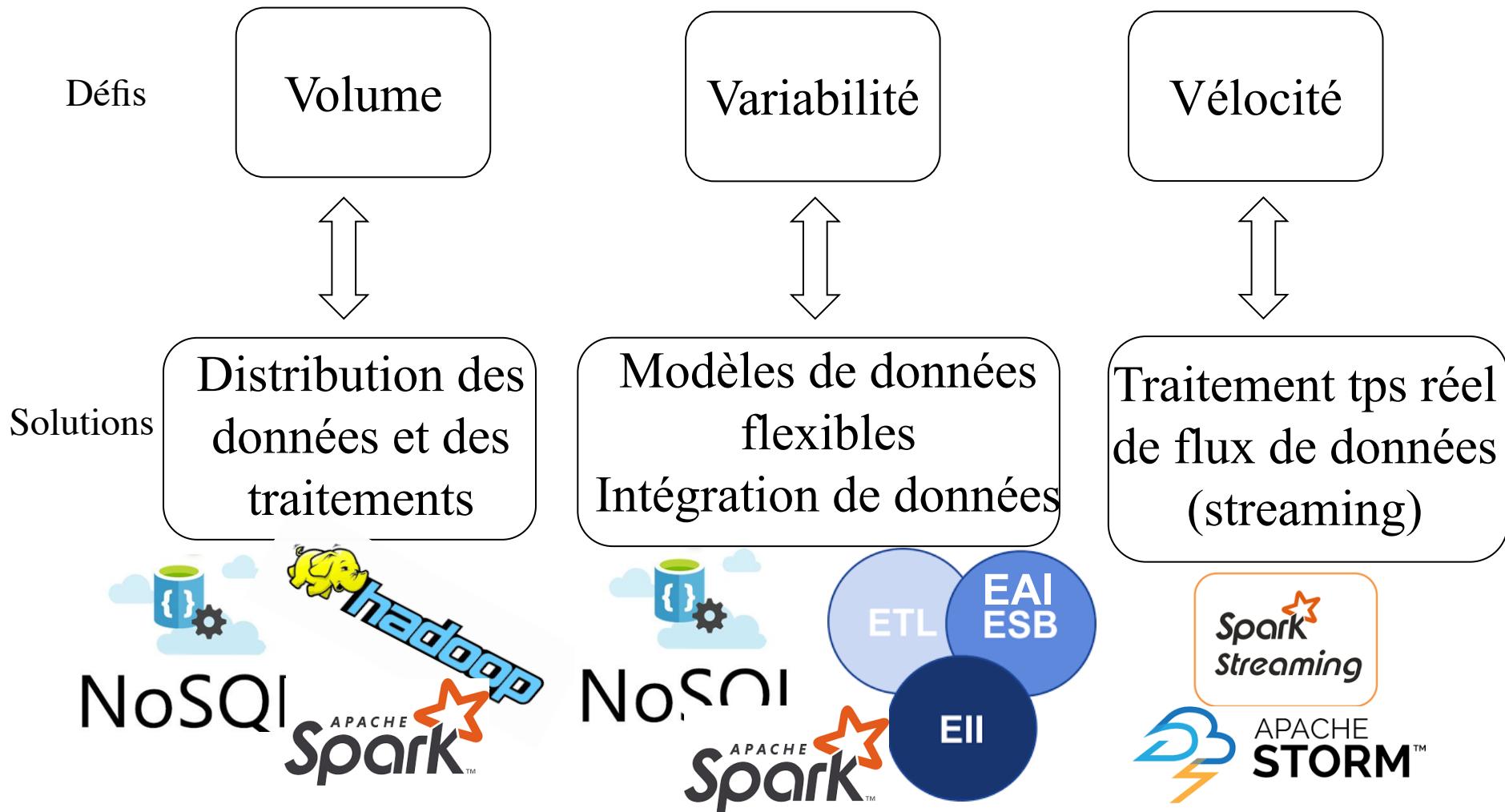
La solution Big Data

Hadoop, Spark, NoSQL

Introduction

Outils du Big Data

- Défis et solutions (*non exhaustif*) :



4V : Volume de données

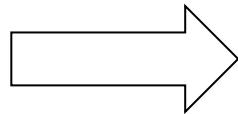
Stockage de données massives et
calculs distribués

4V : Volume de données

Passage à l'échelle

☞ Objectif visé : la *scalabilité horizontale (scale-out)*...

1 machine



n machines



Performances

=

p

Nombre de machines

x

n

Performances

≈

p x n

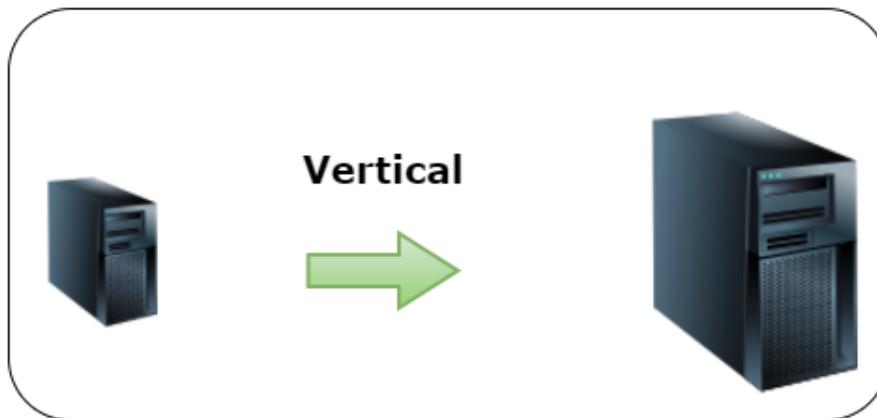
$p \in \{\text{espace de stockage, temps de calcul}\}$

4V : Volume de données

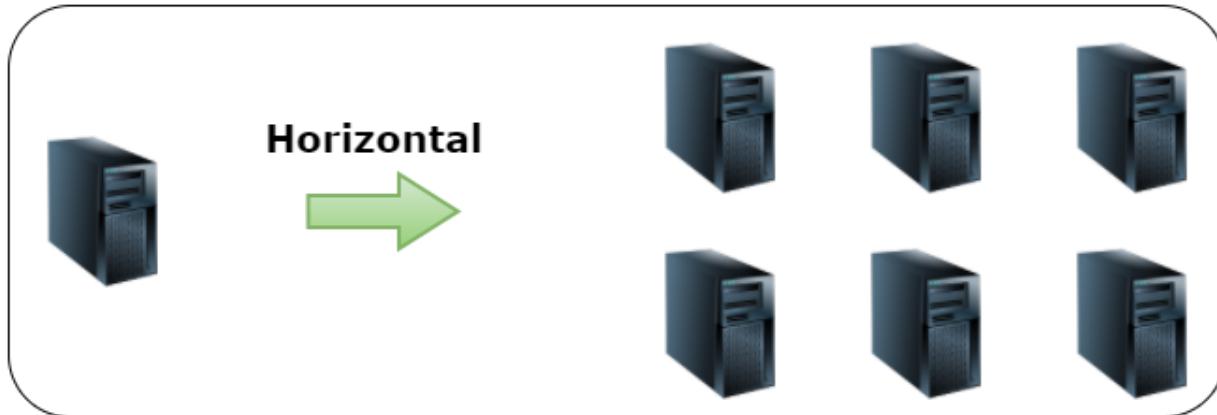
Passage à l'échelle

☞ Objectif visé : la *scalabilité horizontale (scale-out)*...

Scale-up



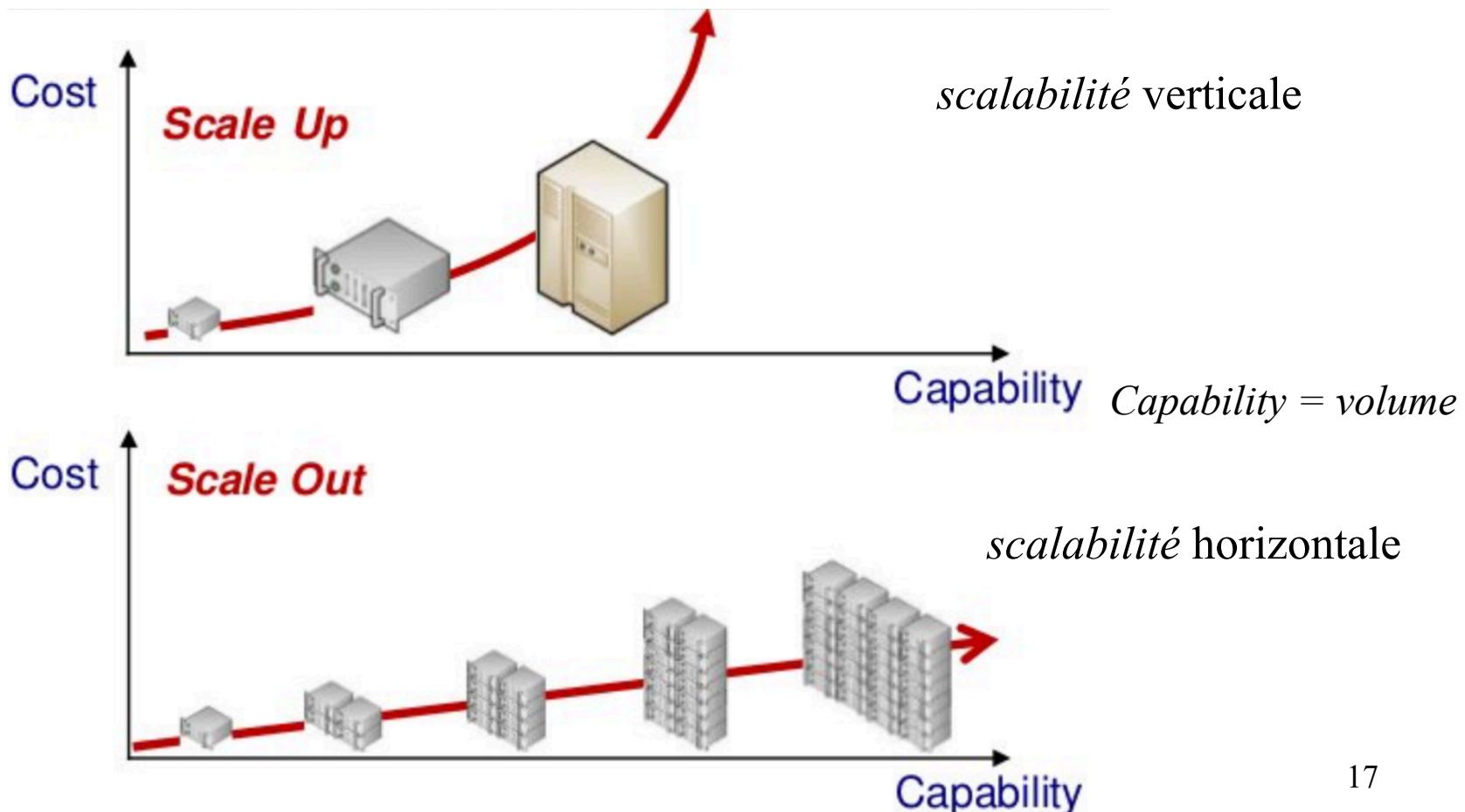
Scale-out



4V : Volume de données

Passage à l'échelle

- ☞ Le coût de la *scalabilité* horizontale évolue linéairement contre une évolution exponentielle de celle de la *scalabilité* verticale

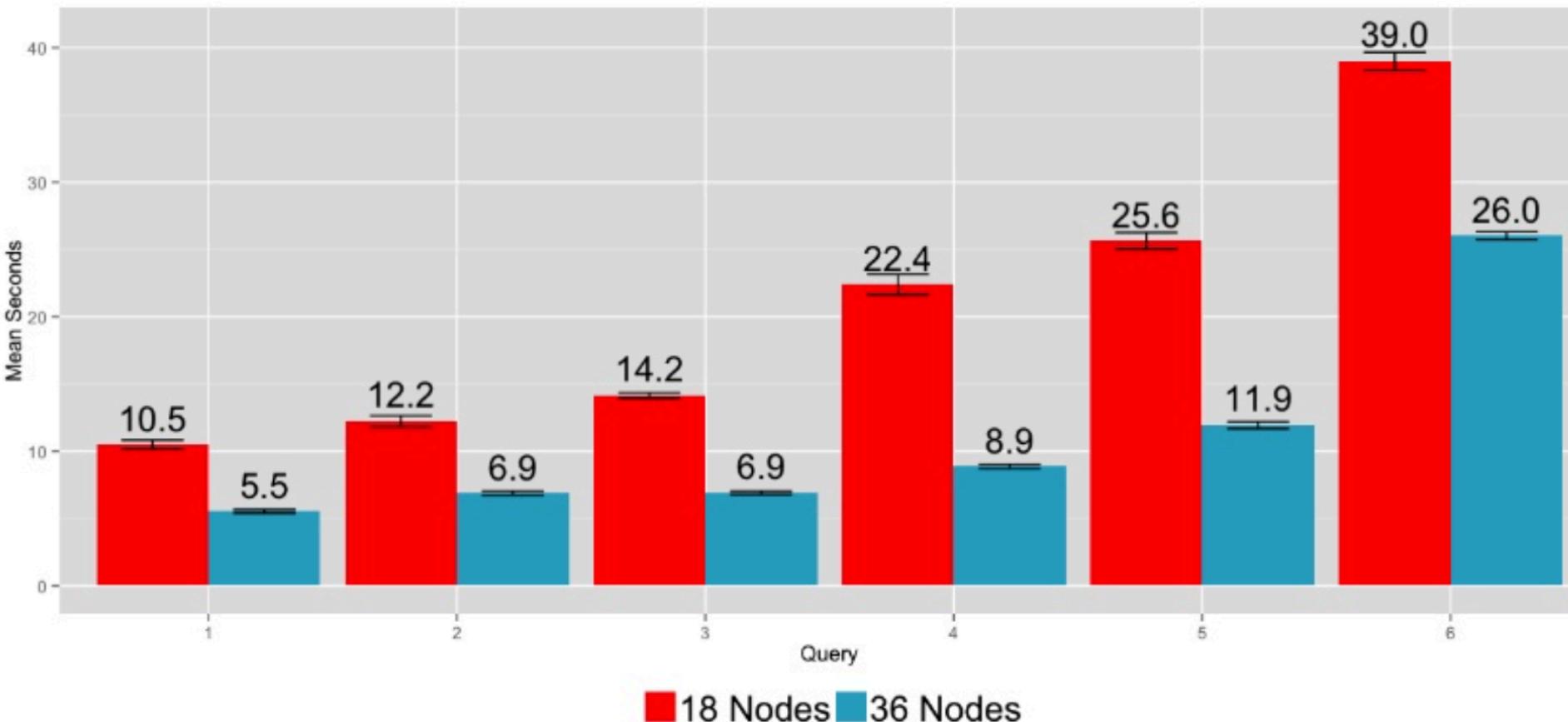


4V : Volume de données

Passage à l'échelle

☛ Performances de la *scalabilité* horizontale avec Impala (1)

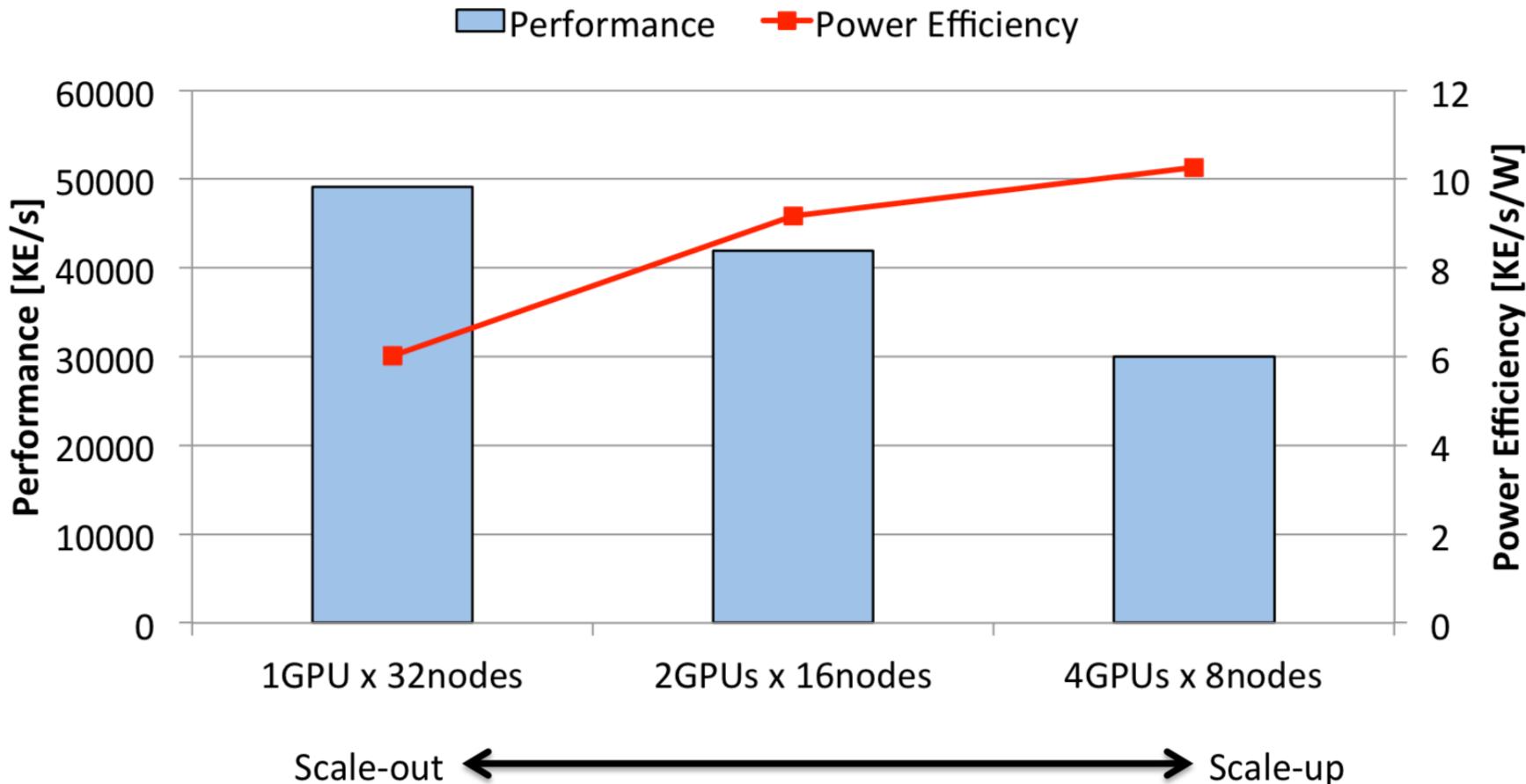
Source : cloudera



4V : Volume de données

Passage à l'échelle

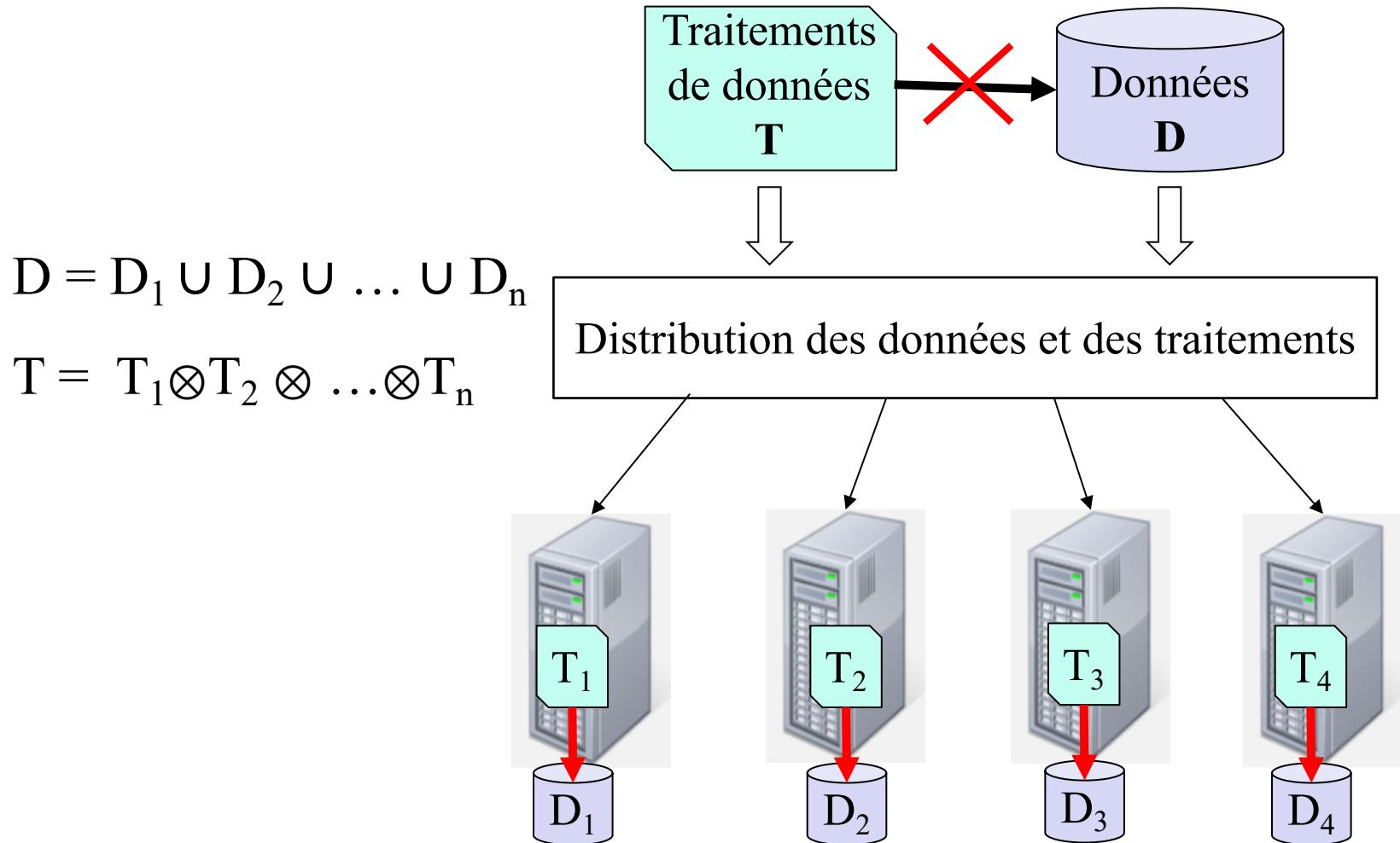
☛ Performances de la *scalabilité horizontale* versus *scalabilité verticale*



4V : Volume de données

Passage à l'échelle (*scalabilité horizontale*)

☞ La *scalabilité horizontale* par distribution des données et des calculs



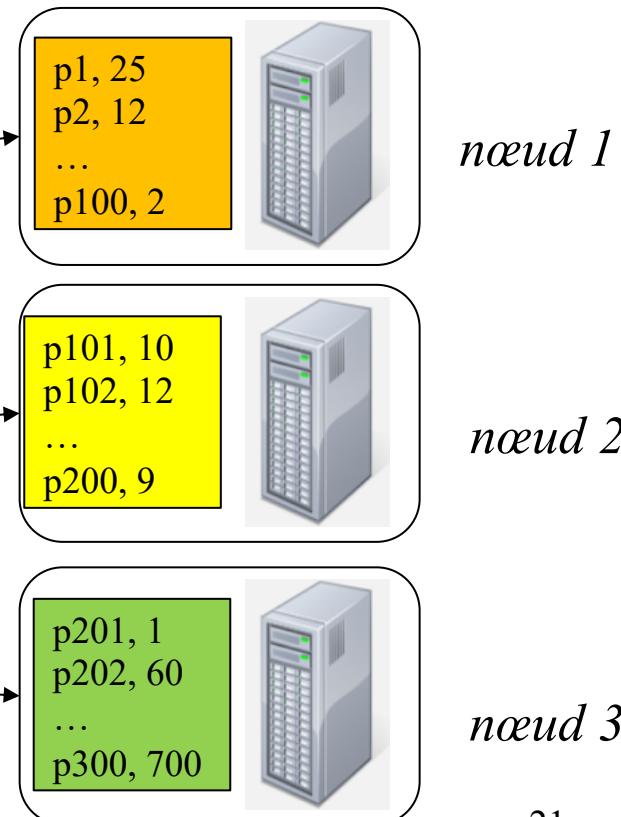
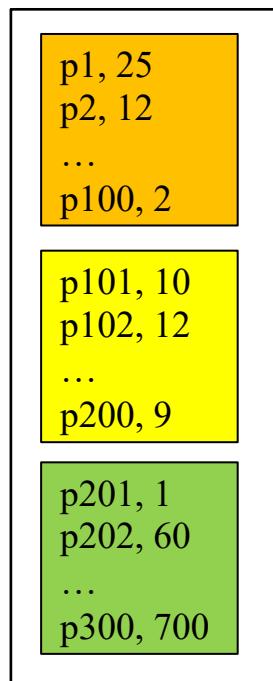
4V : Volume de données

Scalabilité horizontale : distribution des données

Partitionnement de données volumineuses (*sharding*)

- Partitionner les données et les répartir sur plusieurs machines du cluster (nœuds)
- Permet de stocker et de charger des données volumineuses de manière parallèle (parallélisation par distribution, *shared nothing*)

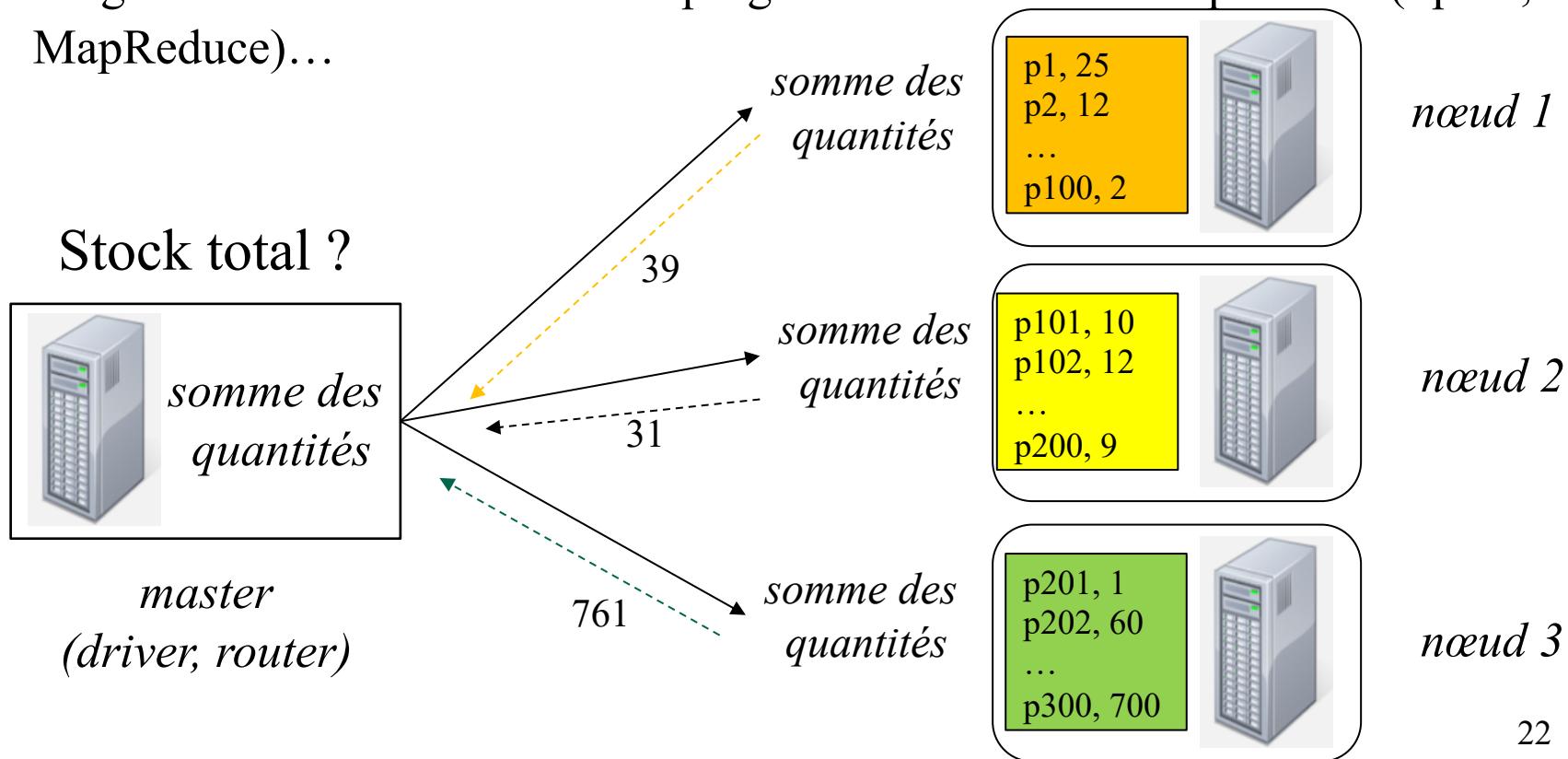
bigfile.data



4V : Volume de données

Scalabilité horizontale : distribution des calculs

- ☞ Distribuer les calculs sur les nœuds de stockage du cluster
 - ☞ Pousser les traitements/calculs là où se trouvent les données (*data locality*)
 - ☞ Répartition de charge (load balancing)
 - ☞ Programmer selon un modèle de programmation/traitement parallèle (Spark, MapReduce)...



4V : Volume de données

Passage à l'échelle (*scalabilité*)

☞ Temps de chargement de 1Eo de données...

Avec une machine SSD (6Go/s)

=

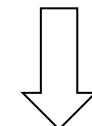
plus de 5 ans !!!

Avec un cluster de 100.000 machines SATA (1Go/s)

=

un peu plus de 2 heures et demi

Machines SSD



Cluster de machines (data center)



4V : Volume de données

Data center

- ☞ Data center : clusters de machines de calculs et de stockage
 - ☞ Constitué d'un grand nombre de machines interconnectées avec du haut débit
 - ☞ Machines peu coûteuses et hétérogènes (différents OS)
 - ☞ Présentent des risques de pannes non négligeables (*design for failure*)
 - ☞ Facilité d'ajout et de retrait de machines
- ☞ Simple calcul de probabilité de panne

Cluster : $n=1000$ machines

p_i : probabilité de panne de la machine i

supposons que $p_i = 0.1$

P (au moins 1 machine tombe en panne) =

$1 - P$ (aucune machine en panne) =

$1 - (1-p_1) \times (1-p_2) \dots (1-p_n)$

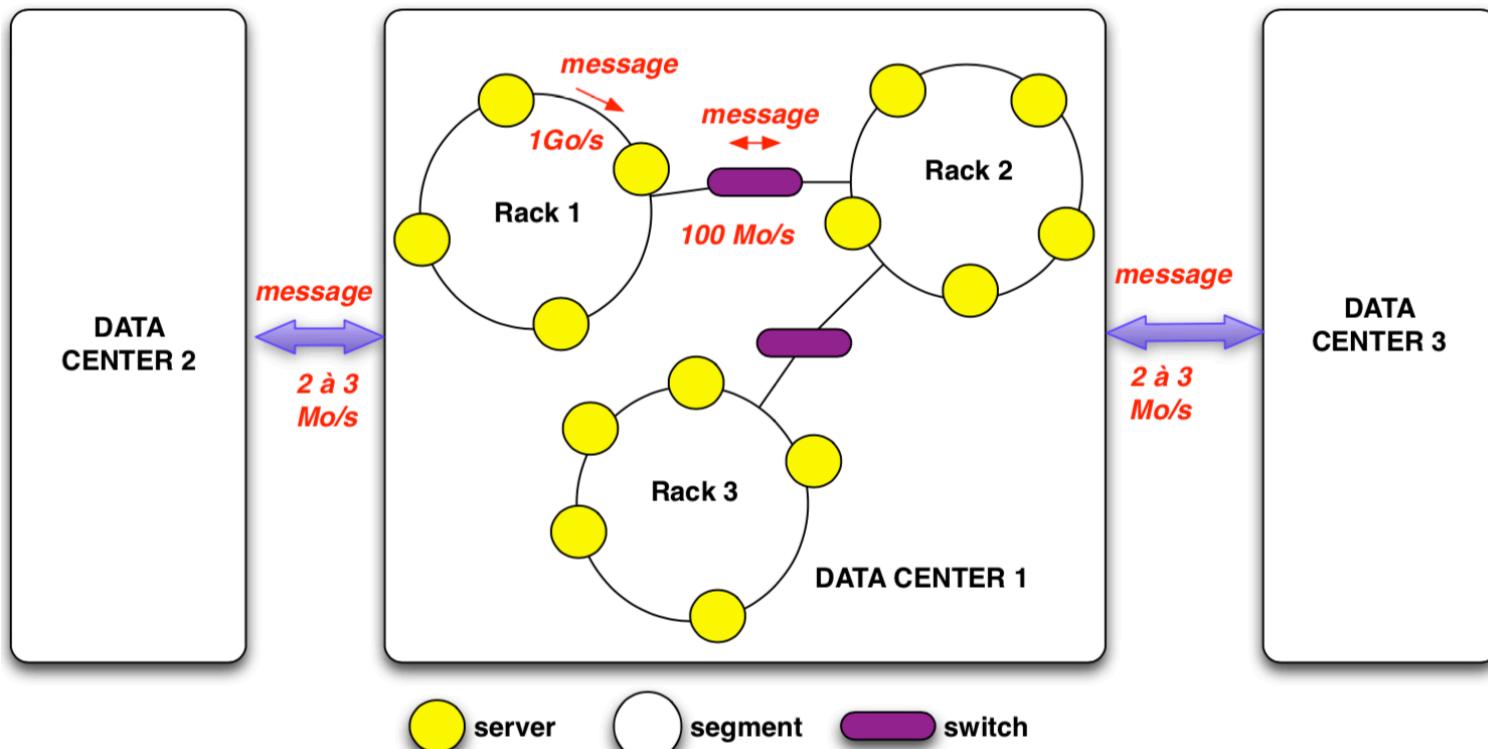
= 0.9999



4V : Volume de données

Data center

- ☞ Trois niveaux de communications LAN
 - ☞ Rack : communication très haut débit (1Go/s)
 - ☞ Data center : interconnexions de racks par routeurs (100Mb/s)
 - ☞ Interconnexion des data centers par routeurs

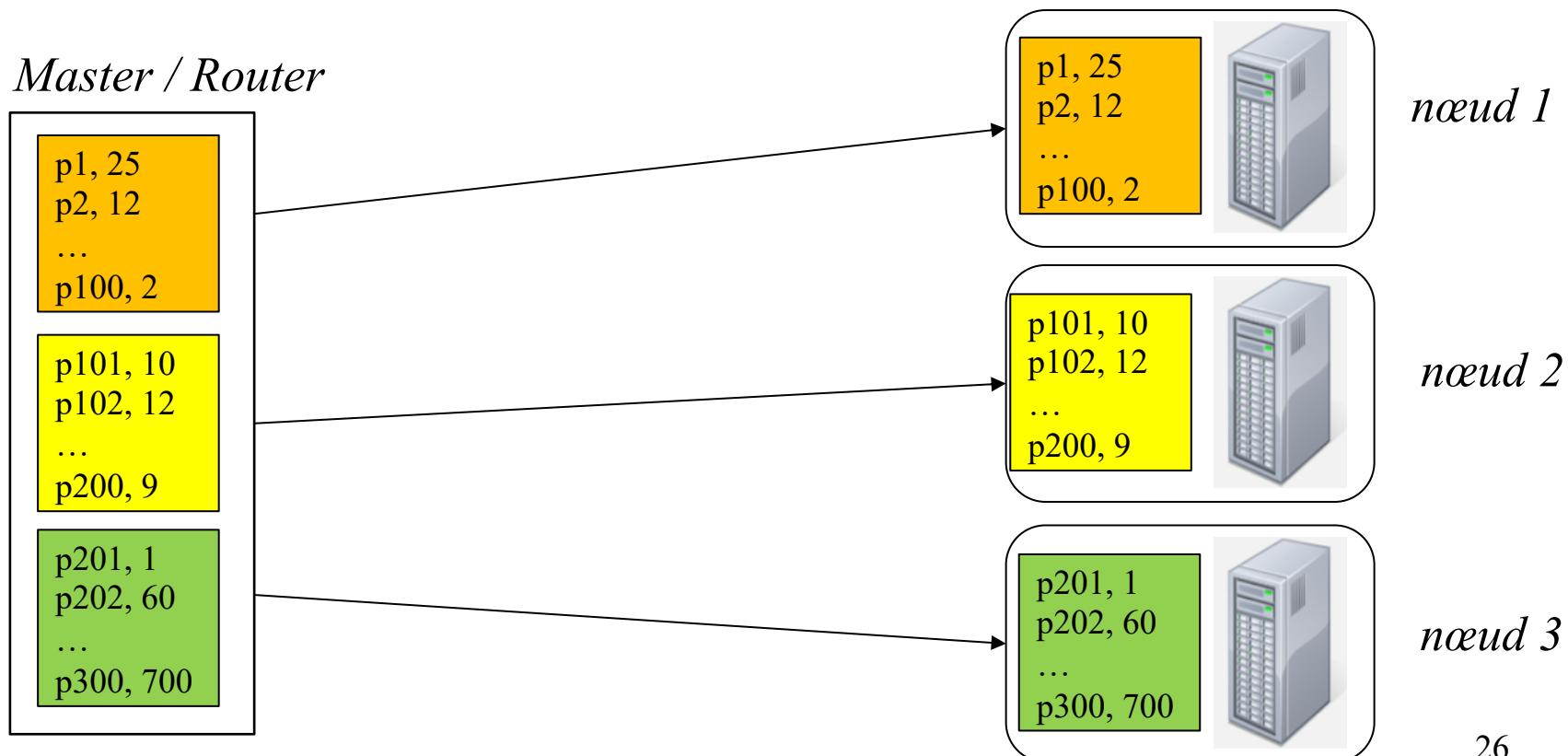


4V : Volume de données

Partitionnement/*sharding* de données

Partitionnement en blocs de taille fixe (HDFS – support de HBase)

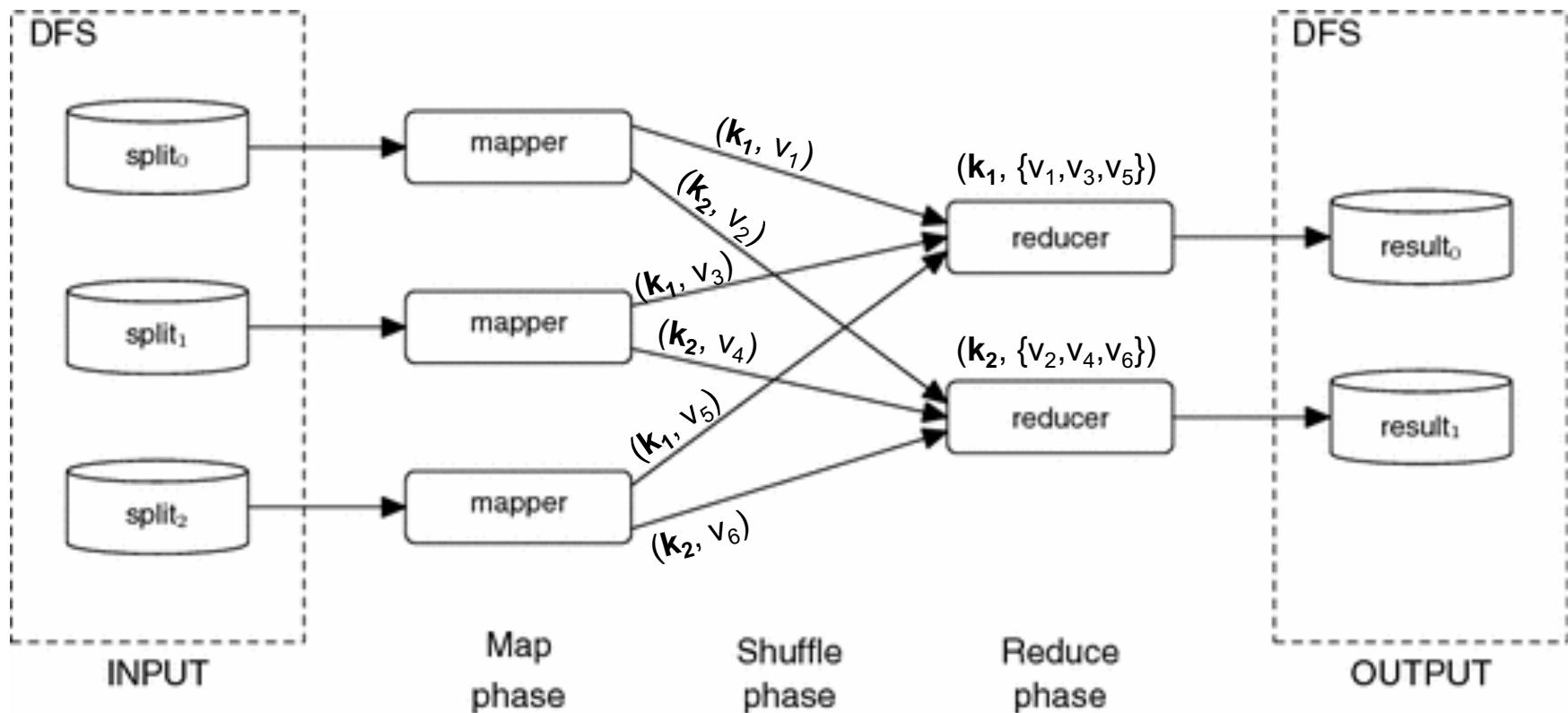
- Un fichier est découpé en blocs de taille prédefinie (64MO)
- Le Master maintient les métadonnées (fichiers-partitions-nœuds)



4V : Volume de données

MapReduce : Traitement de données partitionnées

- ☞ MapReduce : modèle de programmation distribuée (*shared nothing*)
 - ☞ Fonction *Map* : s'exécute de manière indépendante sur les partitions : (clé, valeur)
 - ☞ Fonction *Reduce* : reçoit les couples de même clé et les agrège
 - ☞ Principe de *data locality* : les traitements se font sur les nœuds de stockage



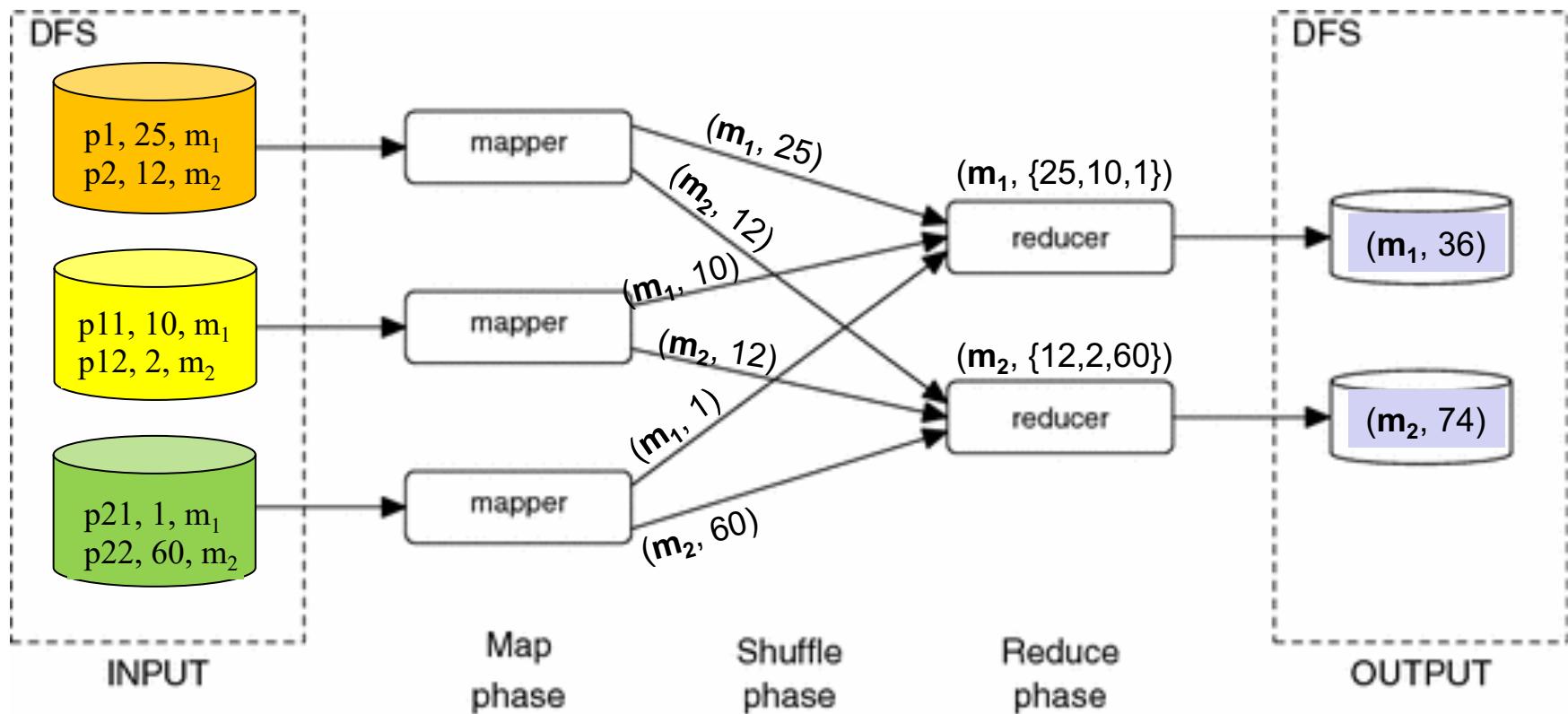
4V : Volume de données

MapReduce : Traitement de données partitionnées

☞ Exemple : calcul du stock par magasin

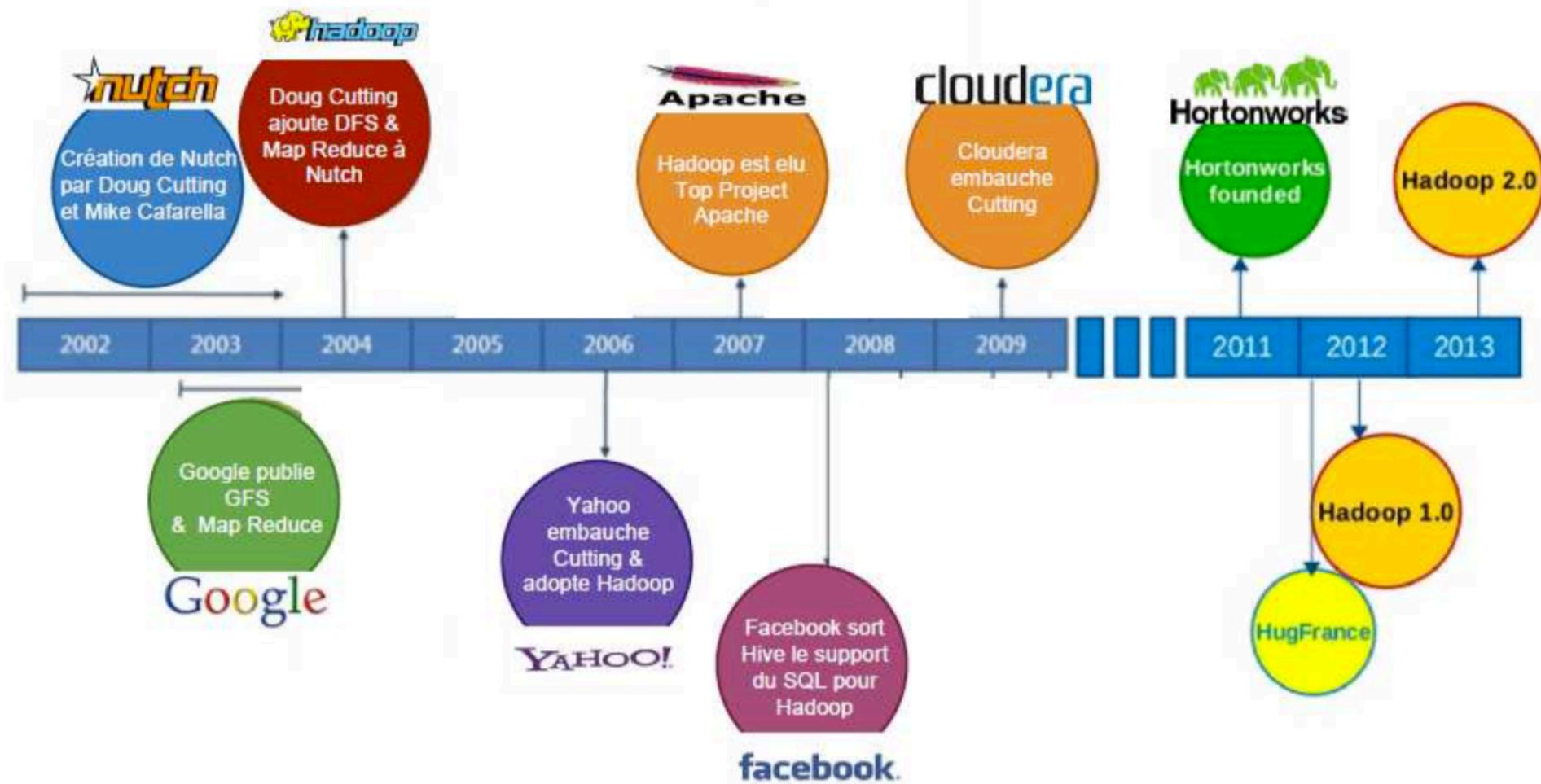
☞ Map : (ref, qté, mag) → (mag, qté)

☞ Reduce : (mag, {qté₁, ..., qté_n}) → (mag, sum{qté₁, ..., qté_n})



Hadoop

Historique



Hadoop

Architecture

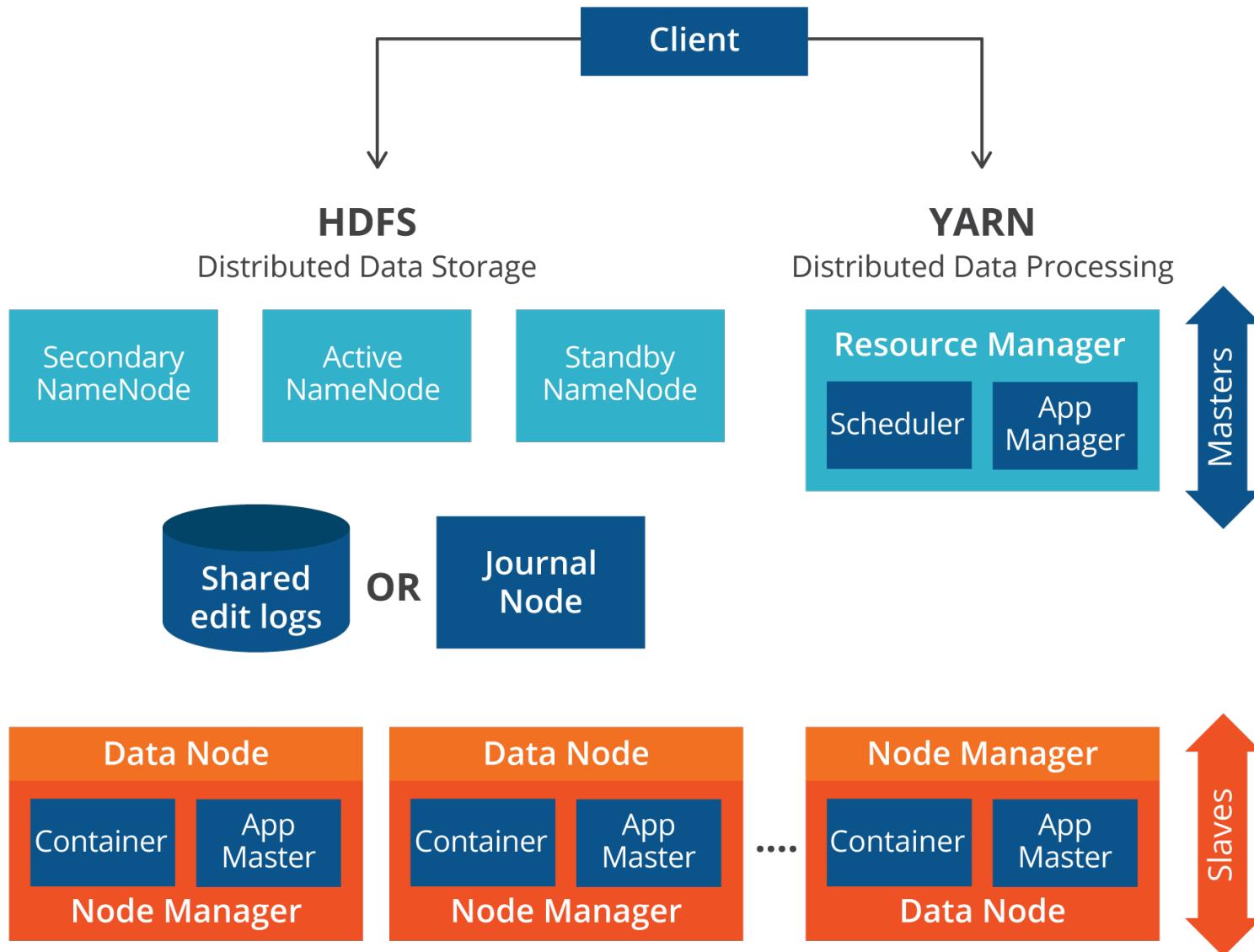
☞ Principe

- ☞ *Diviser pour régner* : diviser les (gros) fichiers de données en petits blocs et les stocker sur plusieurs machines du cluster
- ☞ *Data locality* : pousser/exécuter les traitements là où (sur les machines) se trouvent les données

☞ Architecture : une infrastructure de big data offrant les services de :

- ☞ Stockage de données volumineuses de manière distribuée sur un cluster de machines, c'est son module **HDFS** (Hadoop Distributed File System)
- ☞ Modèle de programmation distribuée sur plusieurs machines (**MapReduce**)
- ☞ Allocation de ressources d'un cluster et ordonnancement, c'est son module **Yarn**
- ☞ Scalabilité horizontale : possibilité d'ajouter des machines au fur et à mesure que les données augmentent

Hadoop Architecture



Hadoop

Raisons de sa démocratisation...

- ☞ Transparency pour le développeur Big Data :
 - ☞ Le développeur envoie ses « grands » fichiers de données sur le cluster sans se préoccuper comment seront-ils distribués ni où seront-ils stockés
 - ☞ Il développe ses programmes en Map-Reduce sans se préoccuper où (sur quelles machines) et comment hadoop les exécute
 - ☞ Il ne se préoccupe pas non plus de la gestion des pannes des machines du cluster ni de la restauration des données reprise en cas panne...

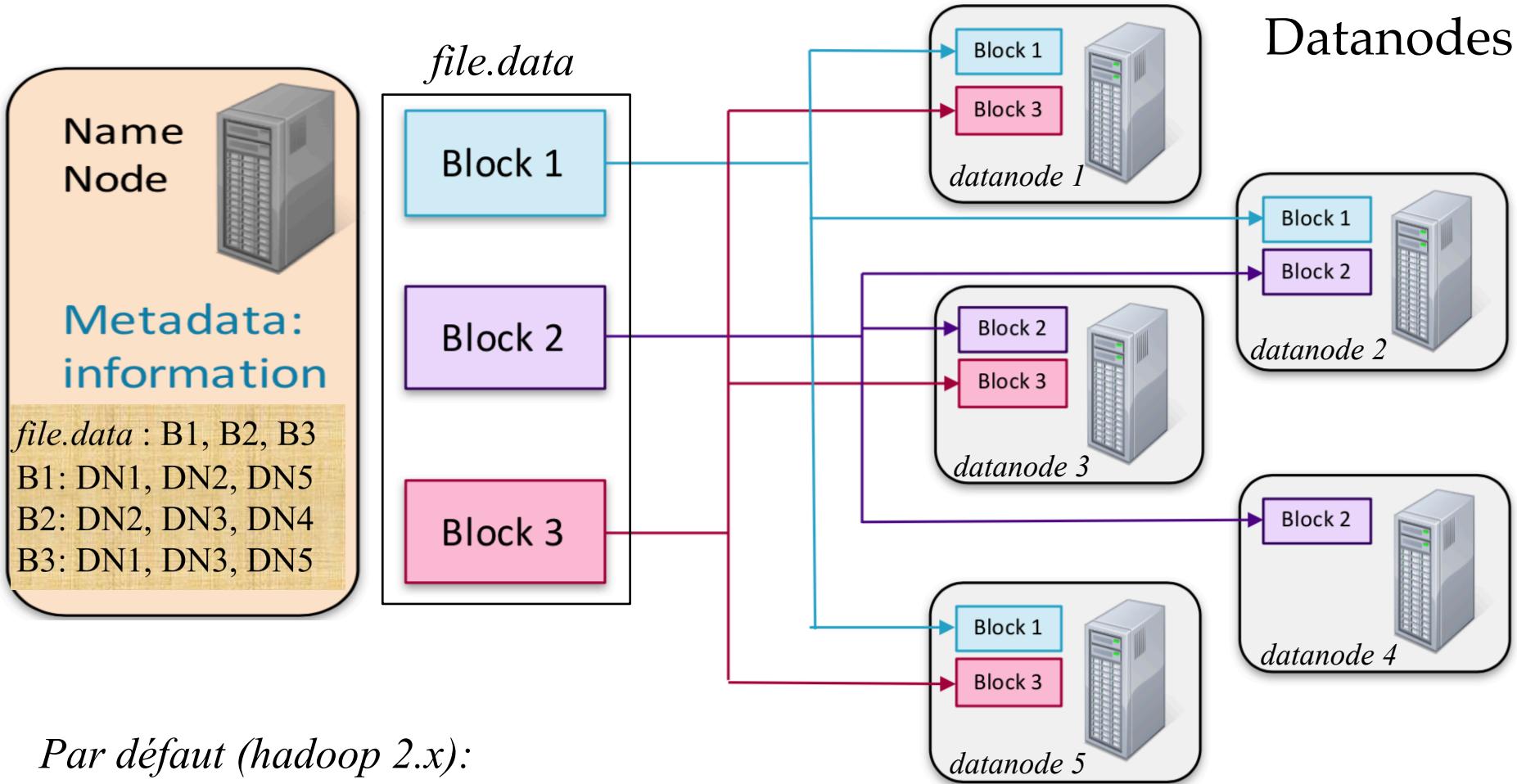
Hadoop

HDFS : SGF distribué

- ☞ Système de gestion de fichiers distribué écrit en Java : les fichiers sont partitionnés en blocs et stockés sur différentes nœuds du cluster
- ☞ S'installe en surcouche d'un SGF classique
- ☞ Horizontalement scalable
- ☞ Tolérant aux pannes des machines du cluster (par réPLICATION DES BLOCS)
- ☞ Architecture *master-slave* composée de trois modules :
 - ☞ Namenode (nœud master) : répartit les fichiers en blocs et leurs affectations aux *Datanodes* et les droits
 - ☞ Datanode (nœud slave) de stockage des blocs de fichiers (mais aussi de calcul, *data locality*)
 - ☞ Secondary Namenode : sauvegarde de secours du Namenode

Hadoop

HDFS : SGF distribué



Par défaut (*hadoop 2.x*):

Taille d'un bloc = 128Mo

Facteur de réPLICATION = 3

Hadoop

HDFS : commandes de manipulation

- ☞ Deux manières d’interagir avec le HDFS : API Java ou commandes HDFS
- ☞ Quelques commandes :
 - `hdfs dfs -ls <path>`
 - `hdfs dfs –cat/tail <src>`
 - `hdfs dfs -put <localsrc> <dest>`
 - `hdfs dfs -get <src> <localdst>`
 - `hdfs dfs -mkdir <path>`
 - `hdfs dfs -mv <src> <dst>` (rm, rmdir, etc.)
 - `hdfs fsck <path>` : vérifier l’état du répertoire
- ☞ Commandes d’administration : `hdfs dfsadmin -report`

Hadoop

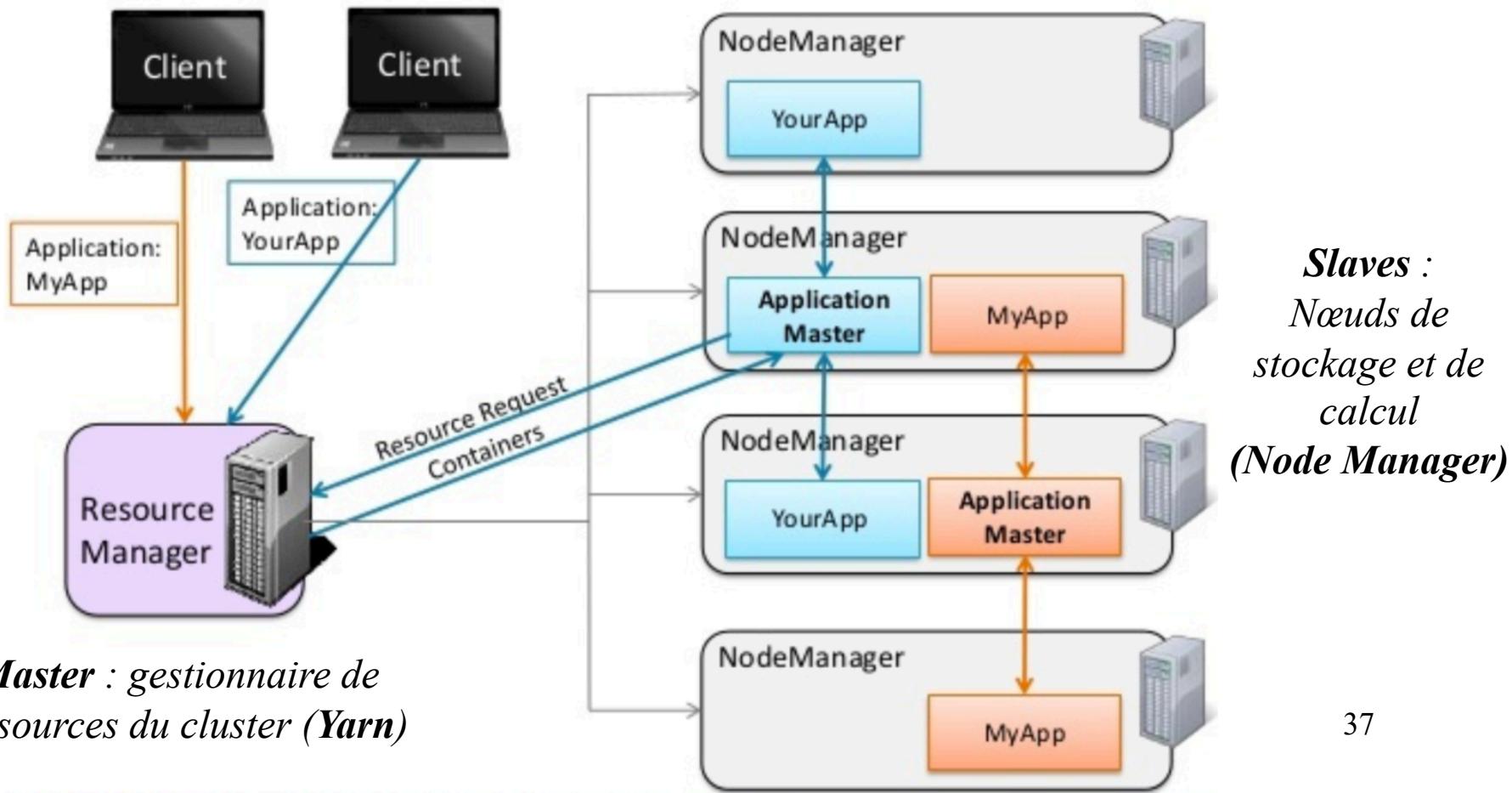
Yarn : Yet Another Resource Negotiator

- ☞ Gestion de ressources (stockage et calcul) du cluster
- ☞ Architecture *master-slave* composée de :
 - Resource Manager (*master*) : alloue les ressources (container) du cluster aux applications
 - Node Manager (*slave*) : gère l'exécution des tâches des applications *Yarn* (MapReduce, Spark, autre) sur un *datanode*
 - *Container* : unité d'allocation de ressources du cluster, c'est le couple (espace de stockage, nombre de *vcore*) sur un nœud
- ☞ Commandes d'administration :
 - `yarn node {-list / -status}`

4V : Volume de données

NoSQL superperforme les SGBD Relationnels

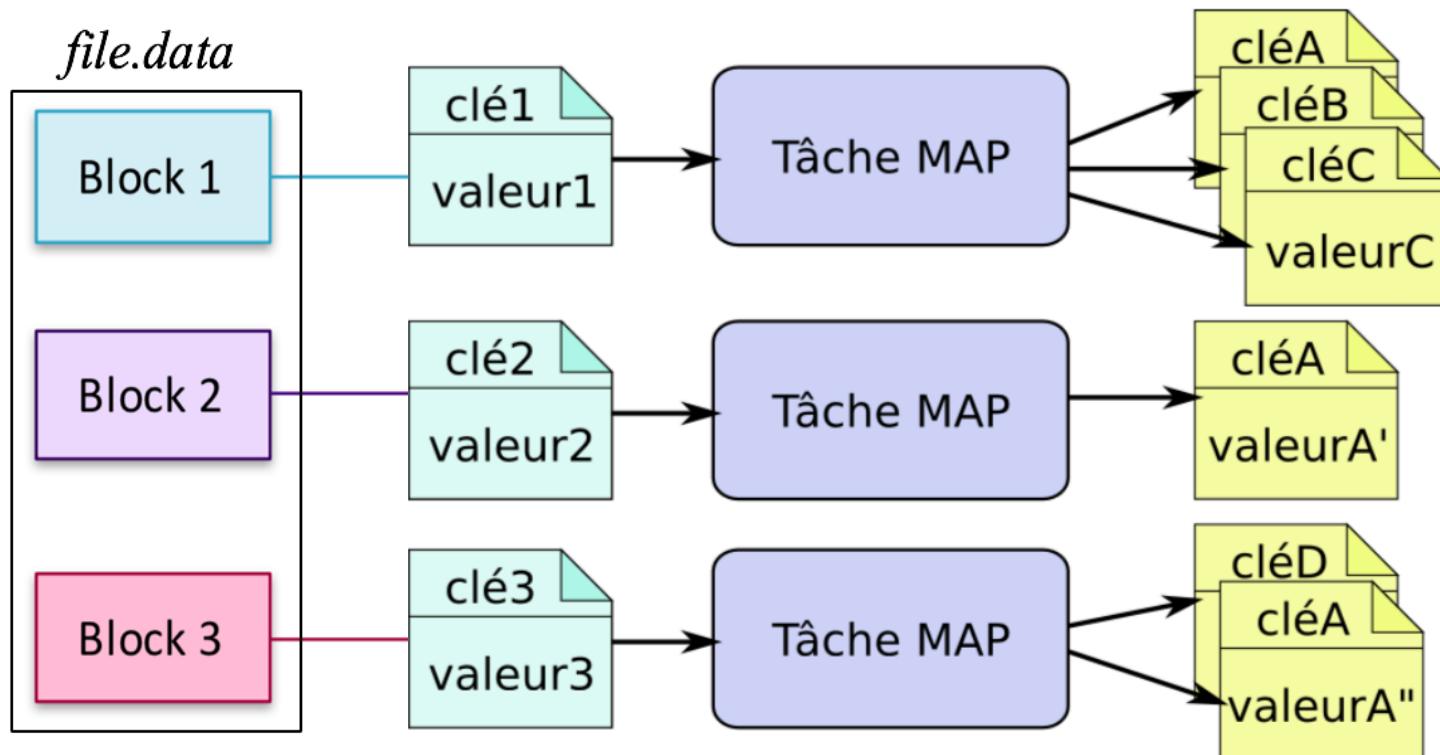
☞ Exemple de l'architecture de distribution de programmes dans Hadoop/Yarn



Hadoop

MapReduce : programmation parallèle et distribuée

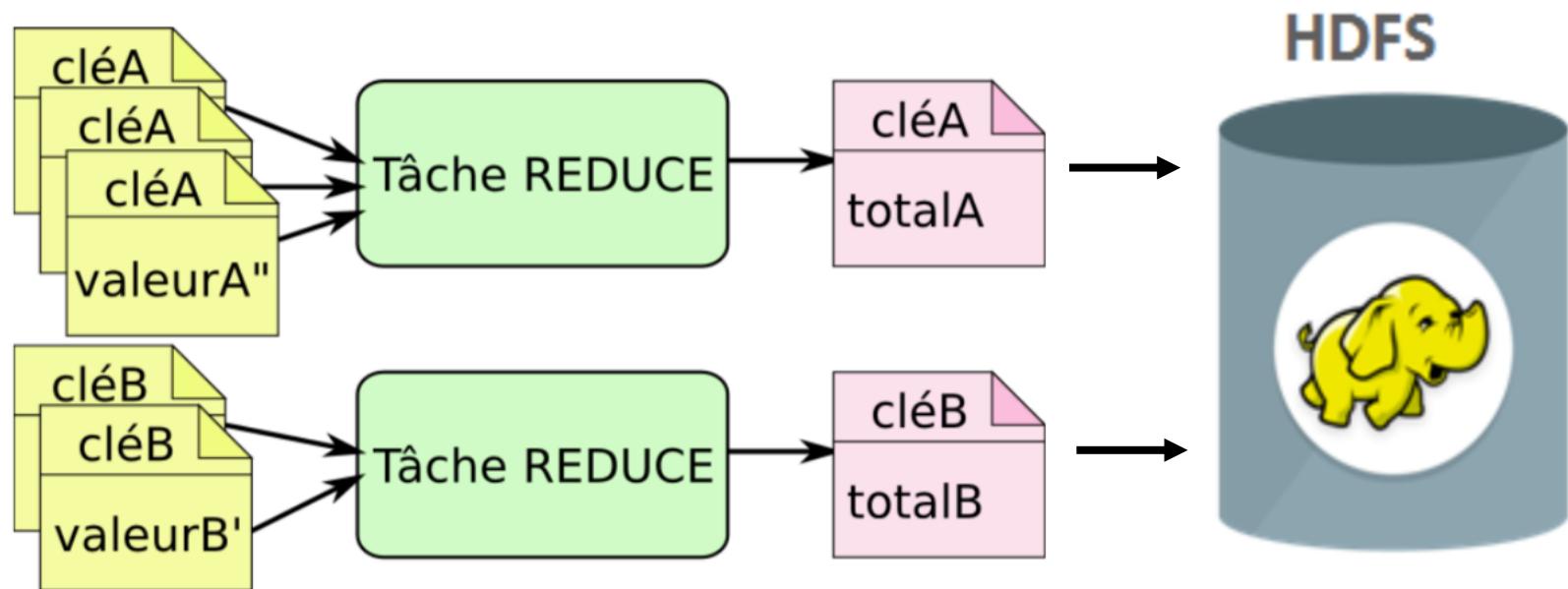
☞ Schéma du *Map*



Hadoop

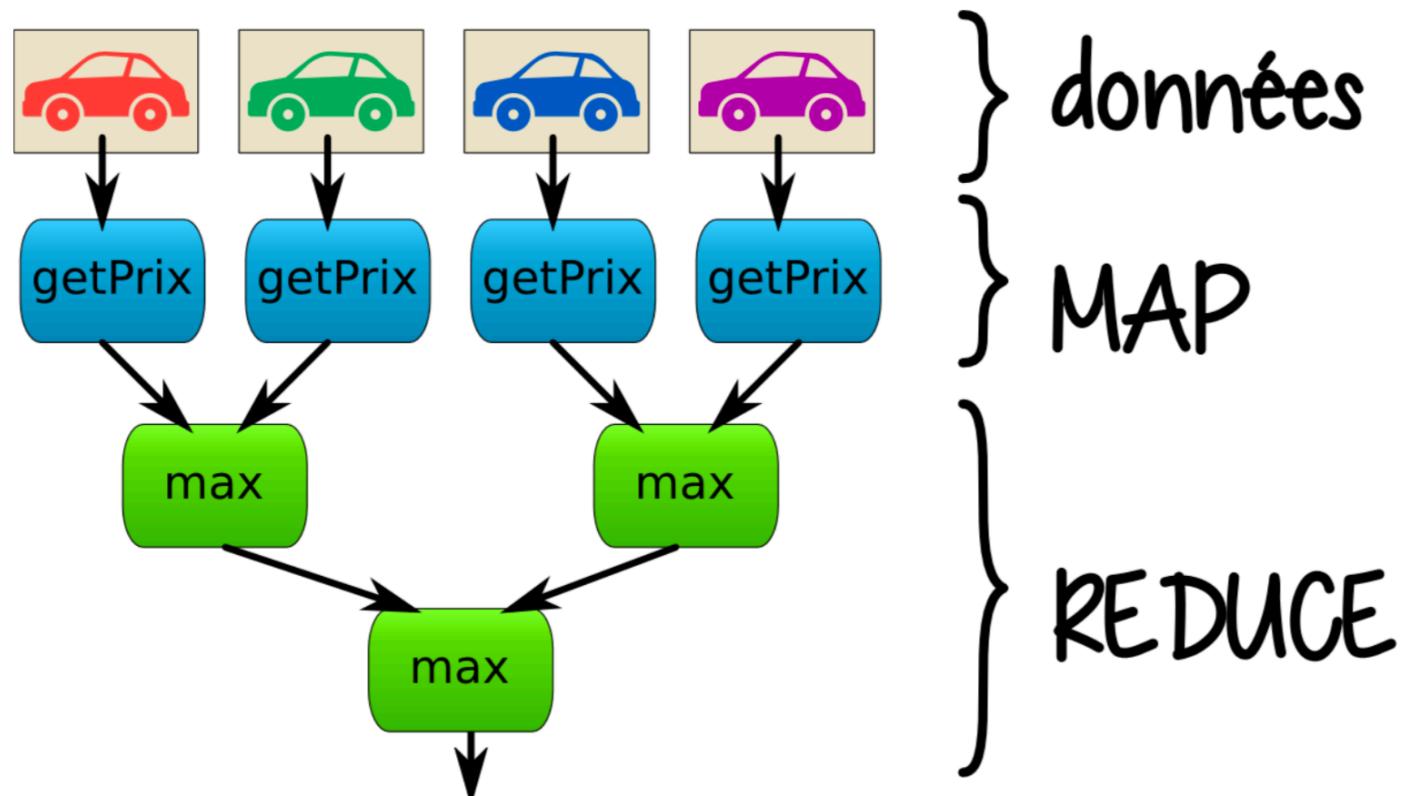
MapReduce : programmation parallèle et distribuée

☞ Schéma du *Reduce*



Hadoop

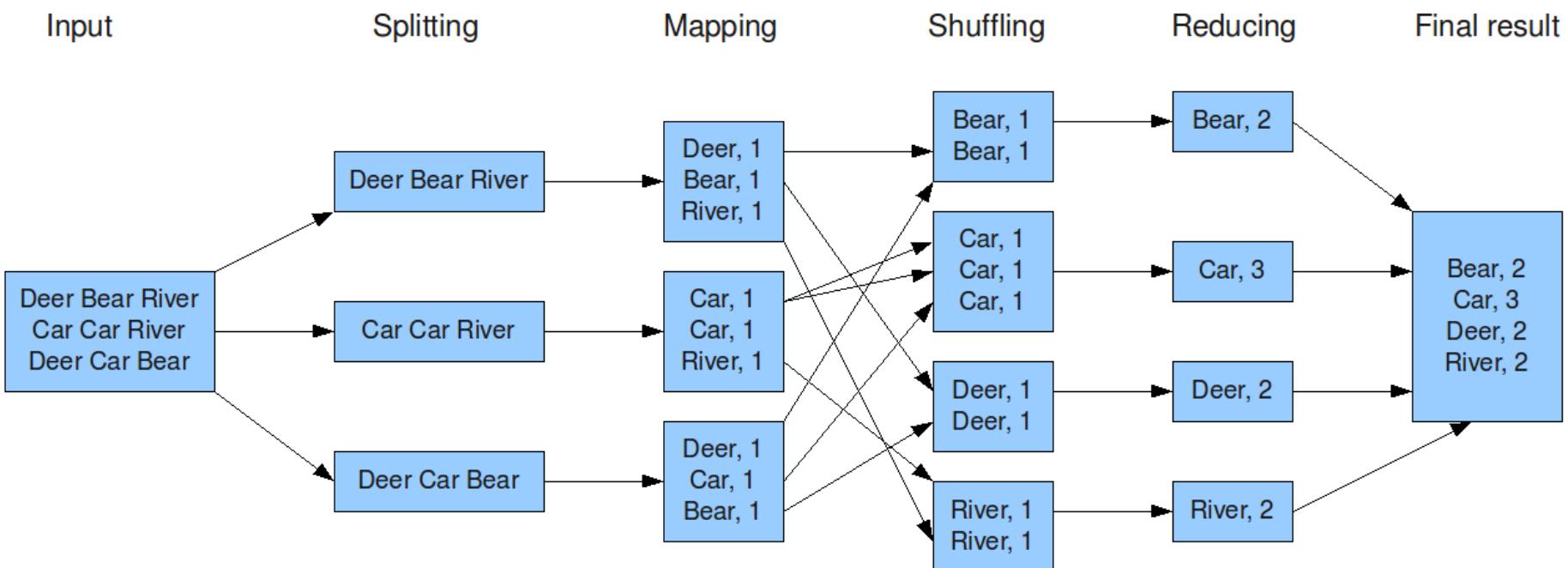
MapReduce : schéma de calcul



Hadoop

MapReduce : exemple de *WordCount*

The overall MapReduce word count process



Hadoop

MapReduce : programmation Java (fonctions *map* et *reduce* de *WordCount*)

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String cleanLine = value.toString().toLowerCase().replaceAll("\\W", " ");
        StringTokenizer itr = new StringTokenizer(cleanLine);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}

public static class SumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    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);
    }
}
```

Hadoop

MapReduce : programmation Java *WordCount* (Driver)

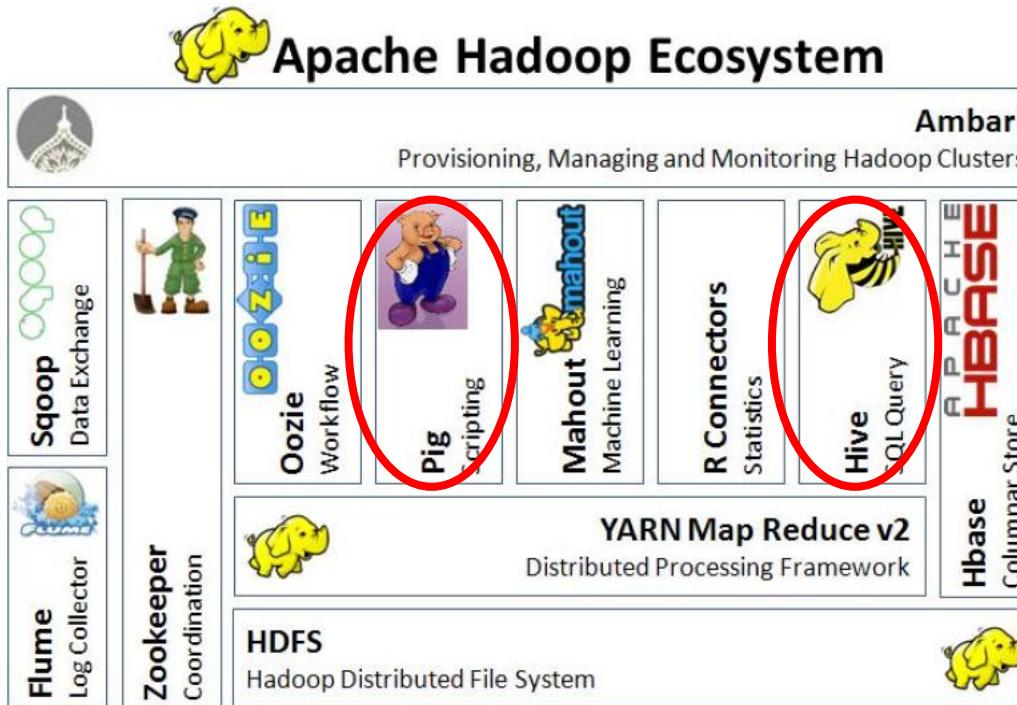
```
public static void main(String... args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
        System.err.println("Usage: wordcount <in> [<in>...] <out>");
        System.exit(2);
    }

    Job job = Job.getInstance(conf, "MapReduce-Lab : WordCount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(SumReducer.class);
    job.setReducerClass(SumReducer.class);
    // job.setPartitionerClass(WordCountPartitioner.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
        FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
        log.info("In Argument: "+otherArgs[i]);
    }
    log.info("In Argument: "+otherArgs[otherArgs.length - 1]);
    FileOutputFormat.setOutputPath(job,
        new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Hive et Pig

Faire du MapReduce sans coder en Java

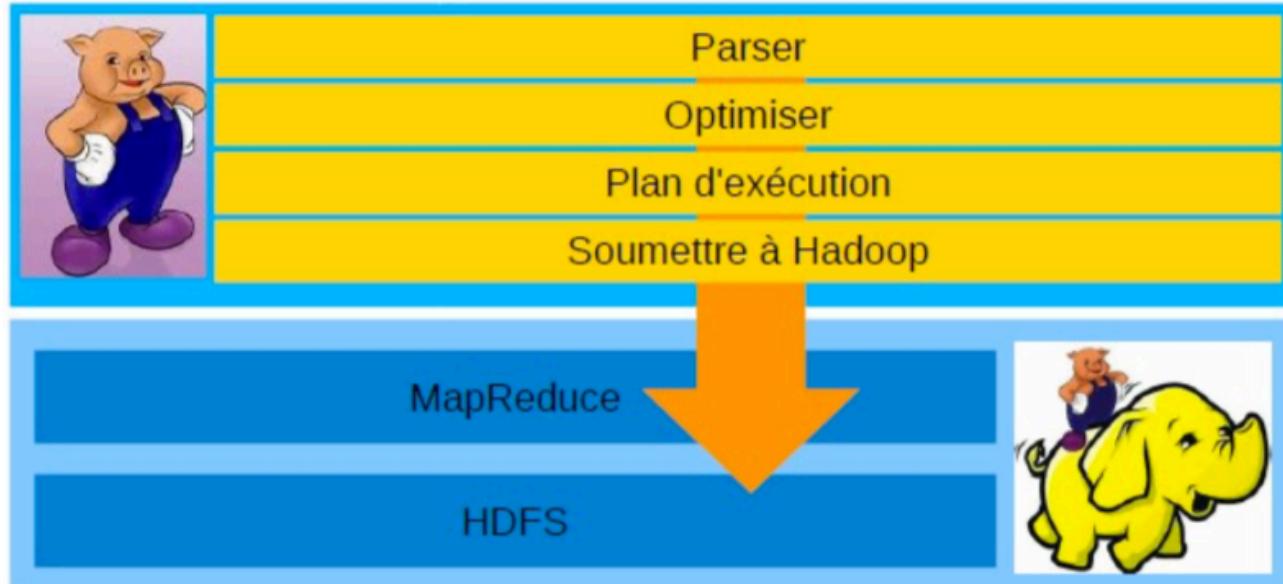
- ☞ Hive : moteur SQL permettant d'interroger les fichiers HDFS (tables virtuelles) avec des requêtes SQL, traduites en Jobs MapReduce
- ☞ Pig : langage de commandes (instructions de haut niveau) pour écrire des scripts de traitement de données HDFS, traduits en Jobs en MapReduce



Pig

Faire du MapReduce sans coder en Java

```
Lines = LOAD '/data/texts/*.txt' AS (line:chararray);
Words = FOREACH Lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
Groups = GROUP Words BY word;
Counts = FOREACH Groups GENERATE group, COUNT(Words);
STORE Counts INTO 'counts';
```



- Augmente la productivité
- 10 lignes en Pig = 200 lignes en Java
- 15 minutes en Pig = 4 heures en Java