Hadoop

Laurent Lapasset

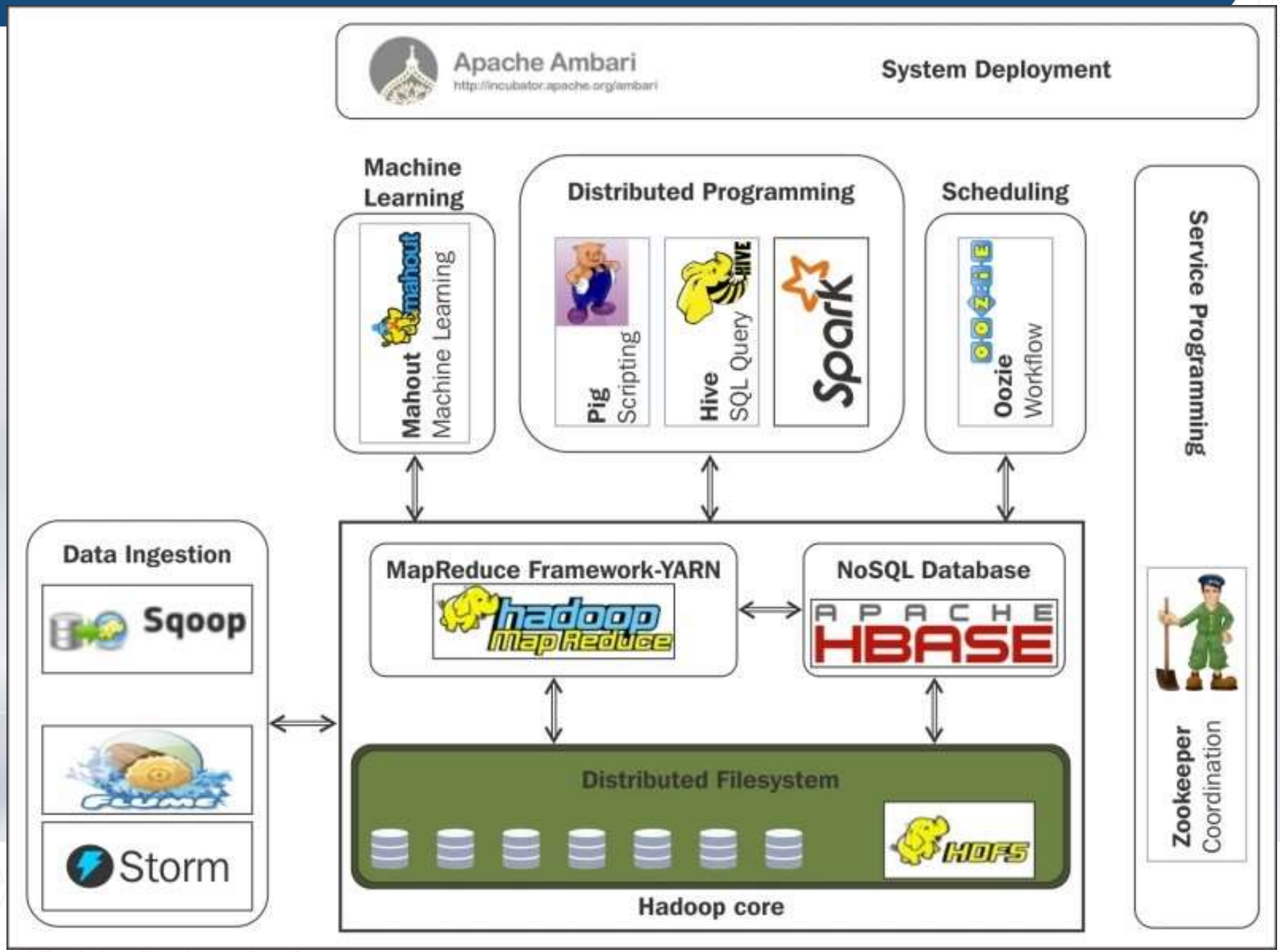
laurent.lapasset@recherche.enac.fr

2024

Les utilisateurs de Hadoop

- Facebook : 10 To/j ; le + grand cluster Hadoop au monde 35/40 pétaoctets (2010)
- eBay : fait appel à Hadoop pour ses besoins en matière d'analyse décisionnelle portant sur de très gros volumes de données.
- Google : réalise la majeure partie de ses développements à 100% en interne
- Twitter : 7To/j ; traitements distribués sur Hadoop
- Yahoo! : 2011 lancement d'une société (Hortonworks) proposant une offre de services autour d'Hadoop.
- LinkedIn : recours à Hadoop pour ses besoins en analytics.
- Airbus : 40To pour chaque vol d'essai

Un écosystème riche

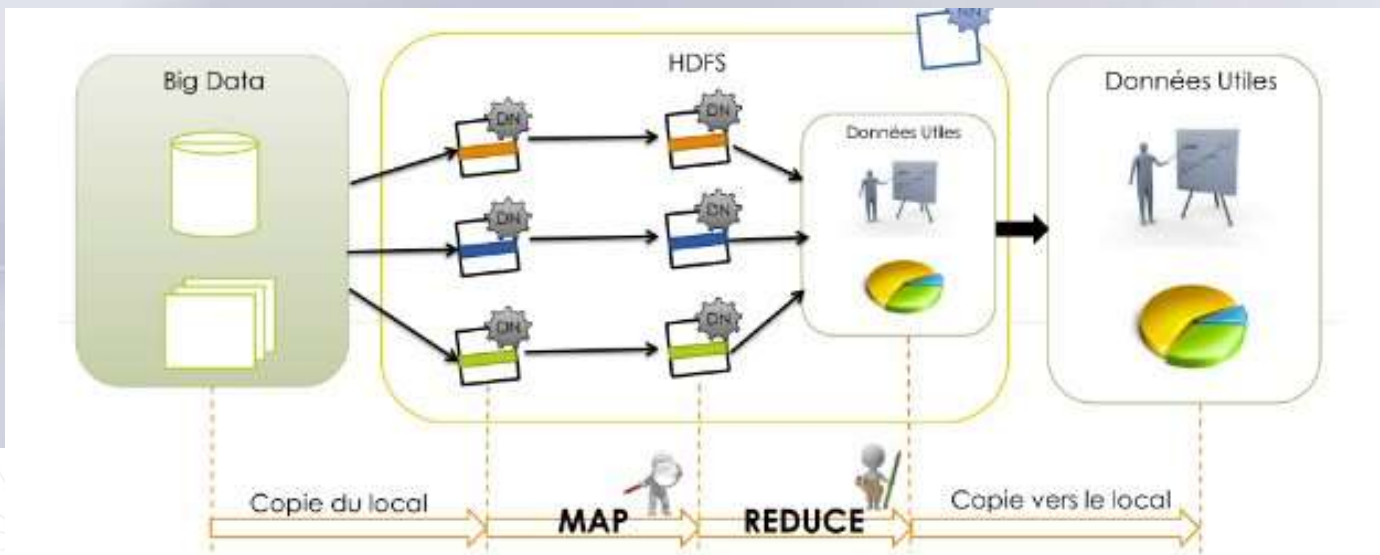


Types de traitements

- **Deux grandes familles de solutions :**
  - Traitement en mode batch (par exemple Hadoop)
    - Les données sont initialement stockées dans le cluster
    - Diverses requêtes sont exécutées sur ces données
    - Les données ne changent pas / les demandes de changement
  - Traitement en mode streaming (par exemple Storm)
    - Les données arrivent en continu en mode streaming
    - Les traitements sont exécutés à la volée sur ces données
    - Modification des données / les requêtes ne changent pas

Les principes Hadoop

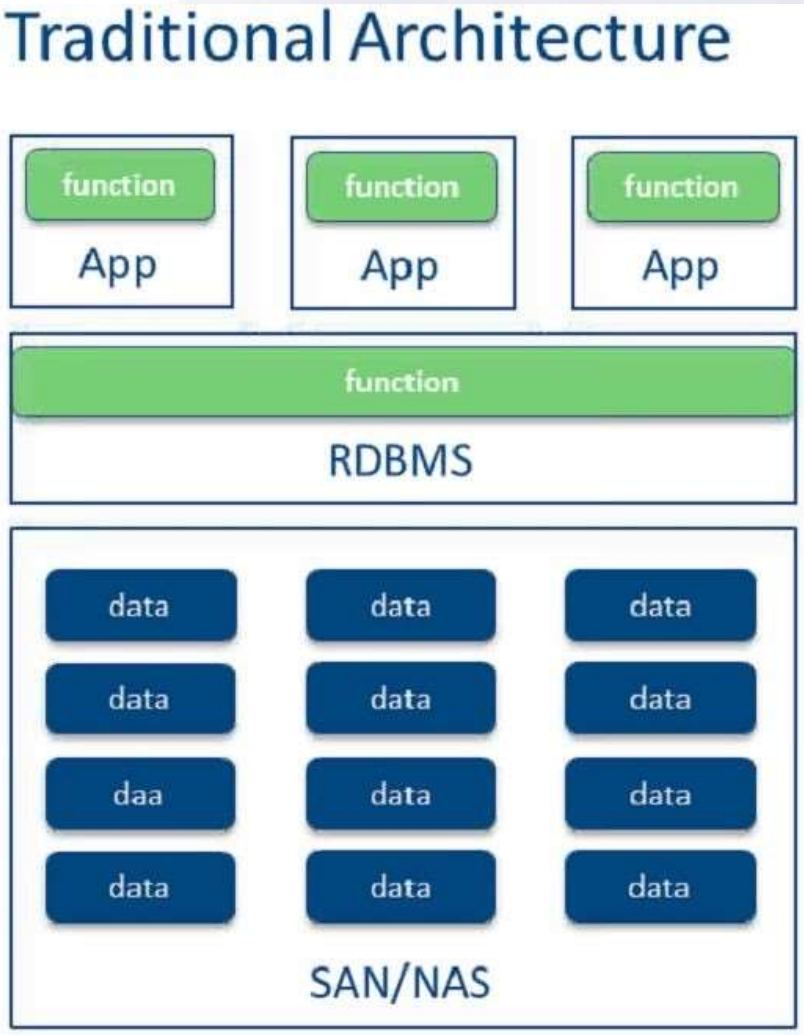
- **Deux parties principales**
  - Stockage de données : HDFS (Hadoop Distributed File System)
  - Traitement de données : Map-Reduce
- **Principe**
  - Copie de données dans HDFS – les données sont divisées et stockées sur un ensemble de nœuds
  - Traiter les données là où elles sont stockées (Map) et récolte les résultats (Reduce)
  - Copie des résultats depuis HDFS



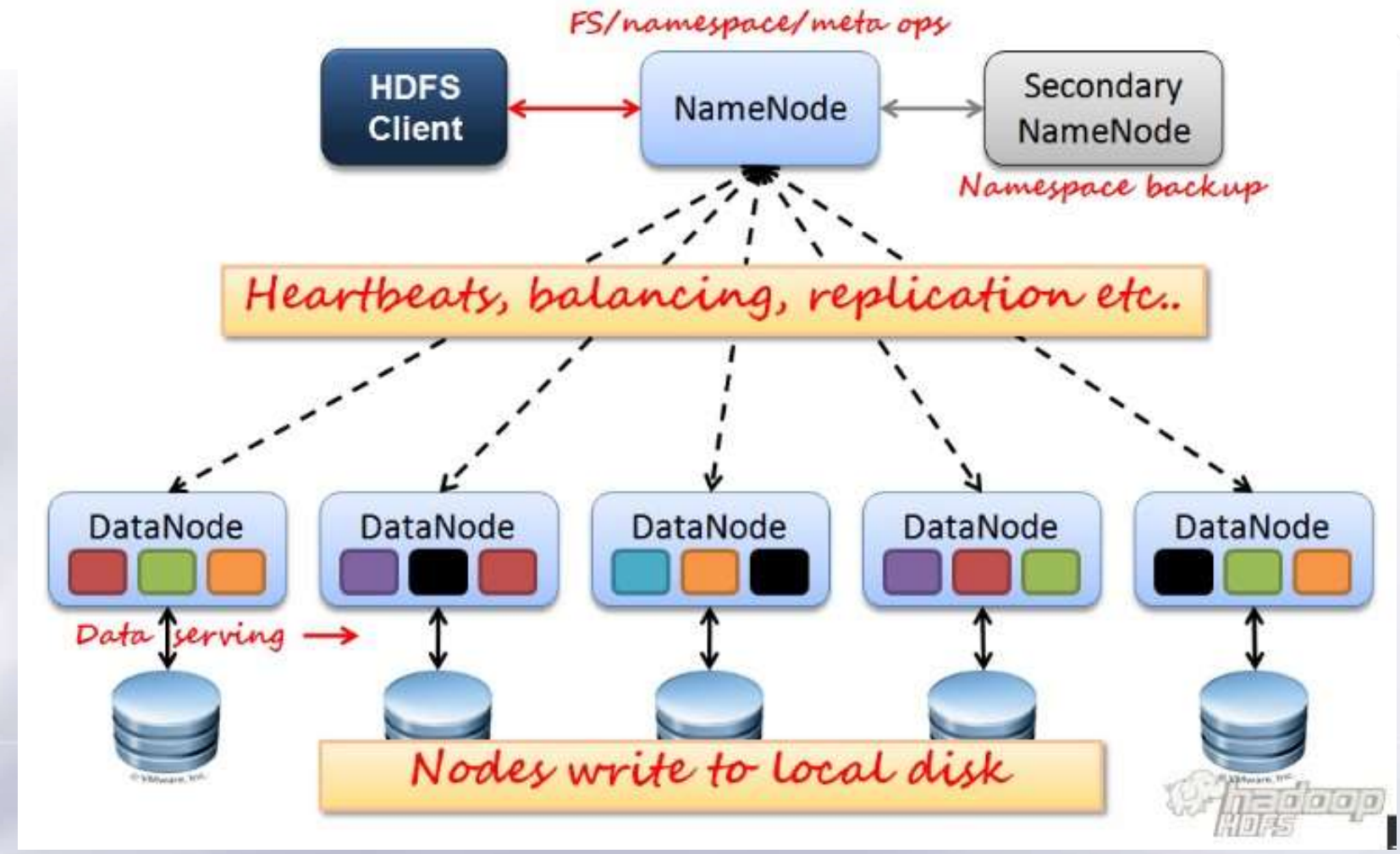
HDFS: Hadoop Distributed File System

- Un nouveau système de fichiers pour lire et écrire des données dans un cluster
- Les fichiers sont découpés en blocs et répartis entre les nœuds
- La taille d'un bloc par défaut : 64 Mb
- Les blocs sont répliqués dans le cluster (3 fois par défaut)
- Write-once-read-many : conçu pour une seule écriture / plusieurs lectures
- Le HDFS repose sur les systèmes de fichiers locaux

Architecture traditionnelle et Hadoop



Architecture HDFS



Le système de stockage de fichier HDFS

Objectifs :

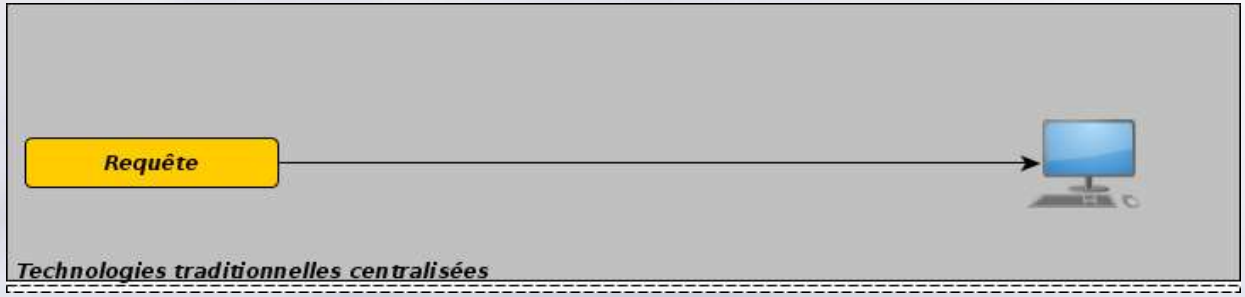
- Tolérant aux pannes d'une manière native (Fault tolerant)
- “Scalable”
- Modèle d'accès immuable
- Déplace les calculs vers les données
- Simple à mettre en place en assurant la portabilité sur différentes plateformes.

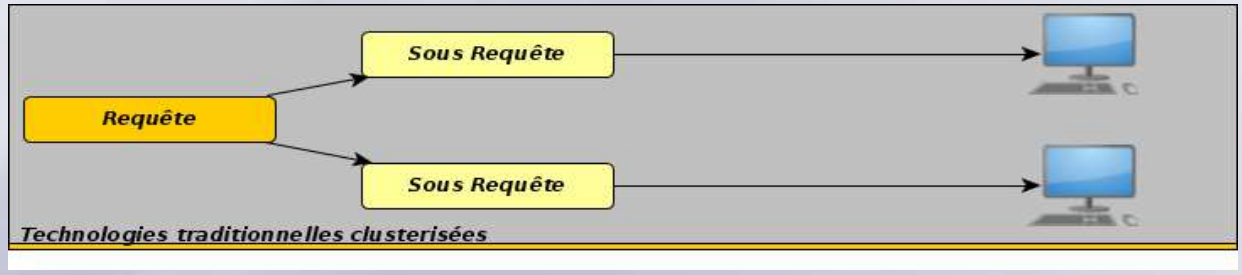
Fonctionnalités :

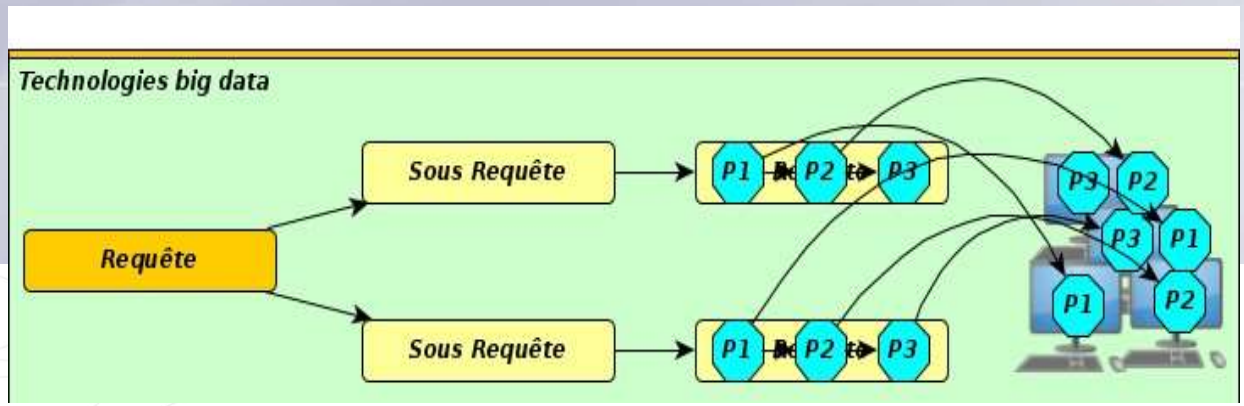
- Gestion des fichiers par blocs
- Réplication et distribution
- Gestion des droits
- Accès aux données en continu (Streaming)
- Stockage des grands jeux de données

Approche générale

- **Principe de base : diviser pour mieux régner**
  - Répartir les E/S et le calcul entre plusieurs devices







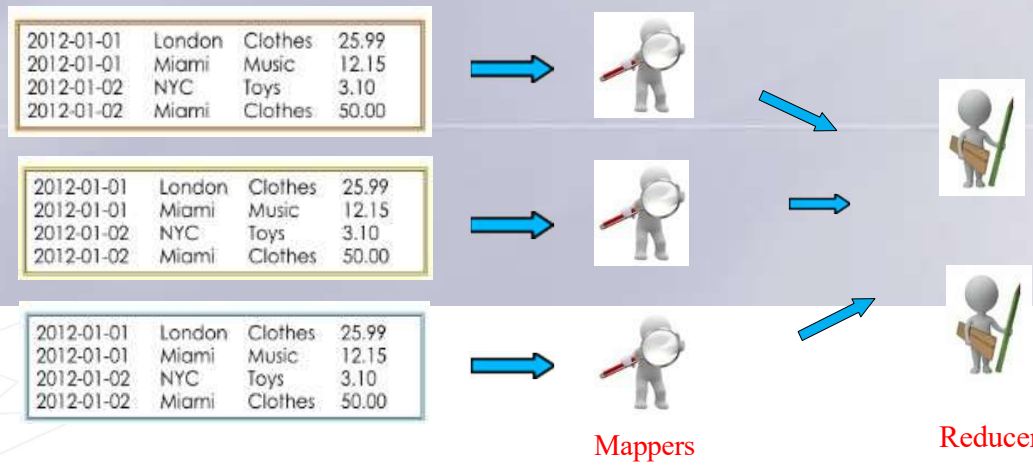
Calcul réparti avec Map Reduce

- **Exemple : nous devons gérer de nombreux magasins dans le monde entier**
  - Un « gros » document enregistre toutes les ventes
    - Pour chaque vente : jour - ville - produit - prix
  - Objectif : calculer le total des ventes par magasin
- **La méthode traditionnelle**
  - Une table de hachage mémorise le total de chaque magasin (<city, total>)
  - Nous itérons à travers tous les enregistrements
    - Pour chaque enregistrement, si nous trouvons la ville dans la table de hachage, nous ajoutons le prix

Map Reduce : un exemple

- **Que se passe-t-il si la taille du document est de 100++ To ?**
  - Les entrées/sorties sont lentes
  - Saturation de la mémoire de l'hôte
  - Le traitement est trop long

- **Map-Reduce**
  - Diviser le document en plusieurs blocs
  - Plusieurs machines de calcul sur les blocs
  - Mappers : exécution en parallèle sur les blocs
  - Reducers : agréger les résultats des Mappers



Map Reduce : un exemple

- **Mappers**

Lire les paires <city, prix> à partir d'un bloc de document

  - Envoyez-les aux **réducteurs** selon la ville

- **Reducers**

  - Chaque réducteur est responsable d'un ensemble de villes
  - Chaque **reduce** calcule le total pour chaque ville

Hadoop : caractéristiques

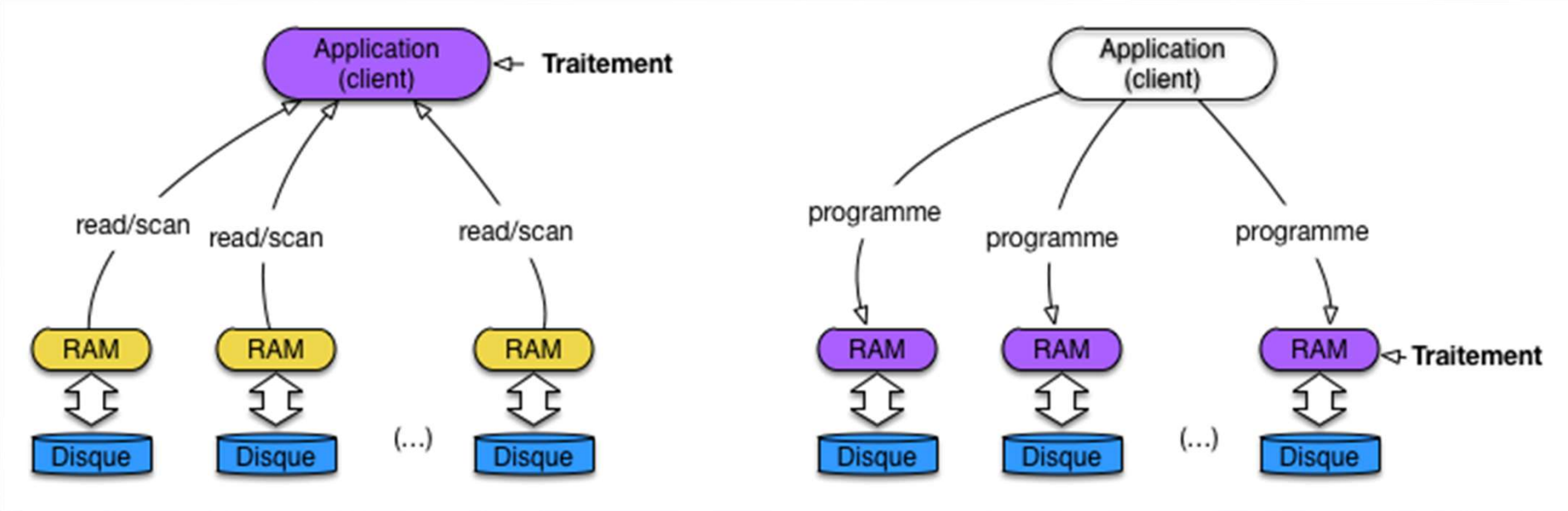
- **Exécution d'applications Map-Reduce dans un cluster**
  - Le cluster regroupe des dizaines, des centaines ou des milliers de nœuds
  - Chaque nœud fournit des capacités de stockage et de calcul
- **Scalability**
  - Il permet le stockage de très grands volumes de données
  - Elle permet le calcul parallèle de ces données
  - Il est possible d'ajouter des nœuds
- **Tolérance aux fautes**
  - Si un noeud tombe
    - La persistance des données est assurée (au besoin les requêtes ou les calculs sont soumis à nouveau)
    - Les données devraient être encore disponibles (les données sont reproduites)

Hadoop : principes

- **Deux parties principales**
  - Stockage des données : HDFS (Hadoop Distributed File System)
  - Traitement des données : Map-Reduce

- **Principe**
  - Copier les données vers HDFS – les données sont divisées et stockées sur un ensemble de nœuds
  - Traitez les données là où elles sont stockées (Carte) et collectez les résultats(Réduire)
  - Copier les résultats depuis HDFS

Hadoop : principe de localité



Programmation sur Hadoop

- **Entité de base: la paire key-value (KV)**
- **La function map :**

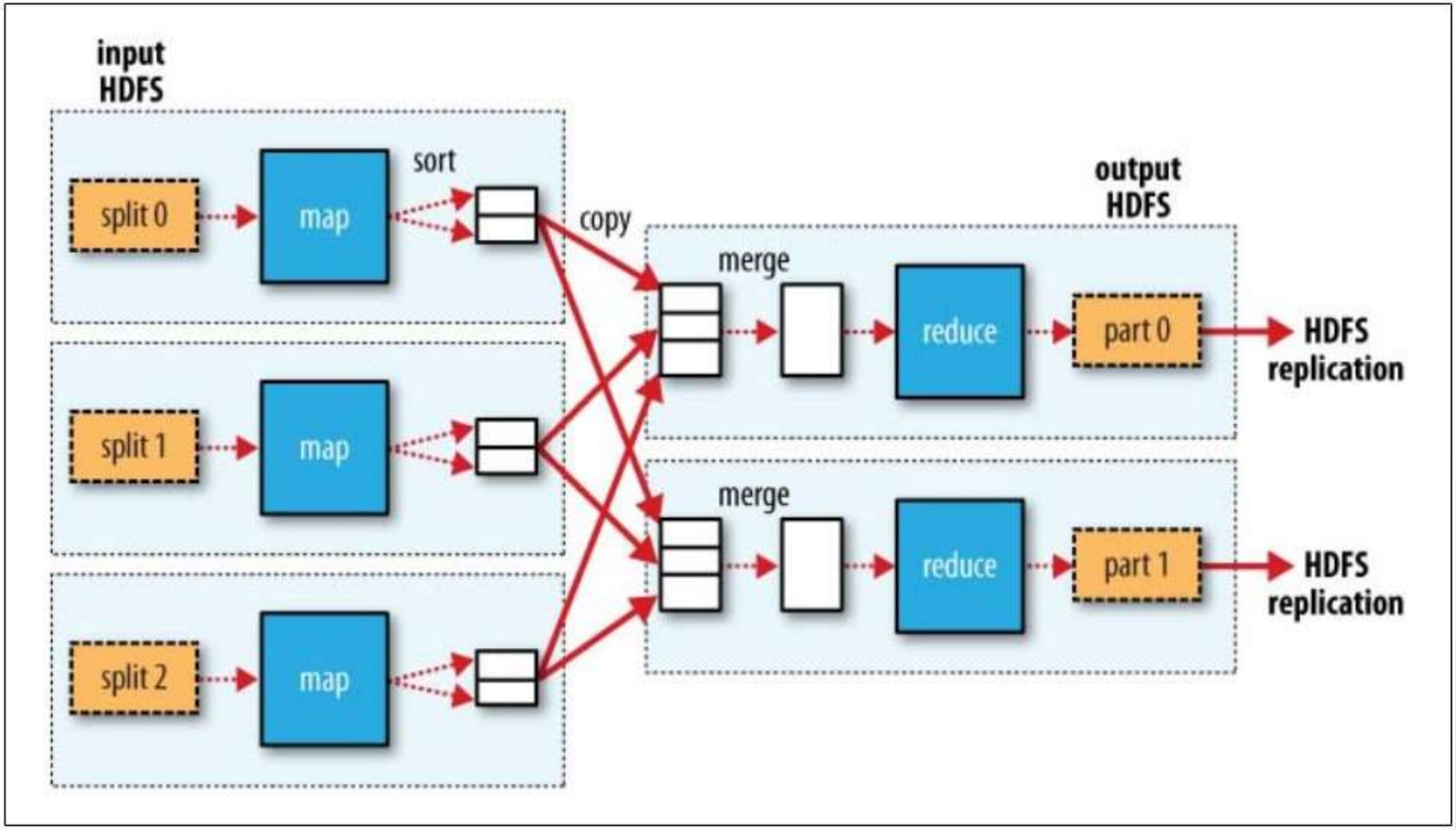
```
Map (void * document) {
        Int cles = 1;
        Foreach mot in document
                calcul_Intermediaire(mot, cles);
}
```

  - Input : KV
  - Output : {KV}
  - La fonction map reçoit successivement un **ensemble** de paires KV du block local
- **La function reduce**

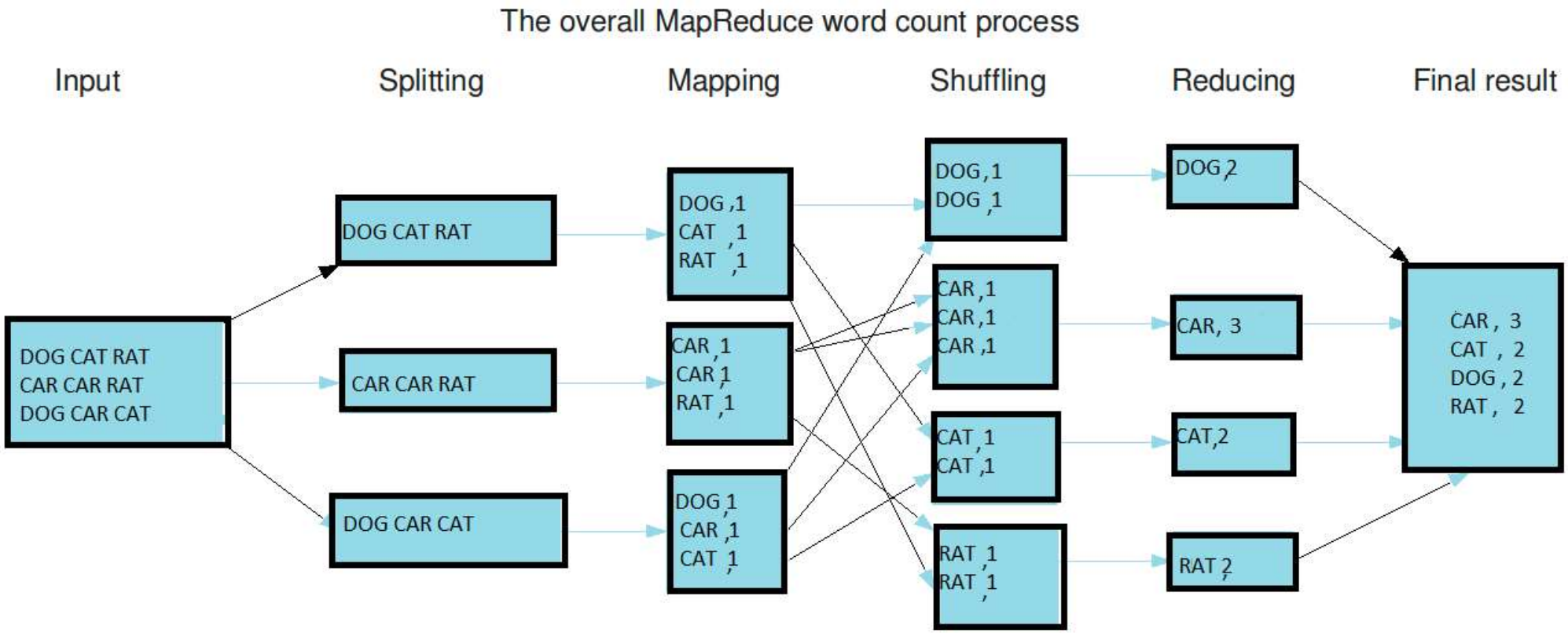
```
Reduce (int cles, Interator values) {
        Int result = 0;
        Foreach v in values
                Result += v;
}
```

  - Input : K{V}
  - Output : {KV}
  - Chaque clé reçue par le reduce est unique

Plan d'exécution



Map Reduce sur un exemple



Programmation avec Hadoop

- **L'application classique : WordCount**
  - Input : un gros fichier texte (ou un ensemble de fichiers textes)
    - Chaque ligne est lue avec des paires KV <line-number, line>
  - Output : nombre d'occurences de chaque mot

Map

- **map(key, value) → List(key_i, $value_i$)**
  - <1, Hello World Bye World> →

< Hello, 1>

< World, 1>

< Bye, 1>

< World, 1>

```
public static class MonTokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException
    {
        String tokens[] = value.toString().split(" ");
        for (String tok : tokens)
            {
                word.set( tok);
                context.write(word, one);
            }
    }
}
```

Shuffle et tri

- Sort : groupe les KVs où le K est identique
- Shuffle : distribut les KVs aux reducers
- Opérations réalisées par le framework

Reduce

- **reduce(key, List(value$_i$)) → List(key$_i$, value$_i$)**

< Bye, [1]> → < Bye, 1>

< Goodbye, [1]> → < Goodbye, 1>

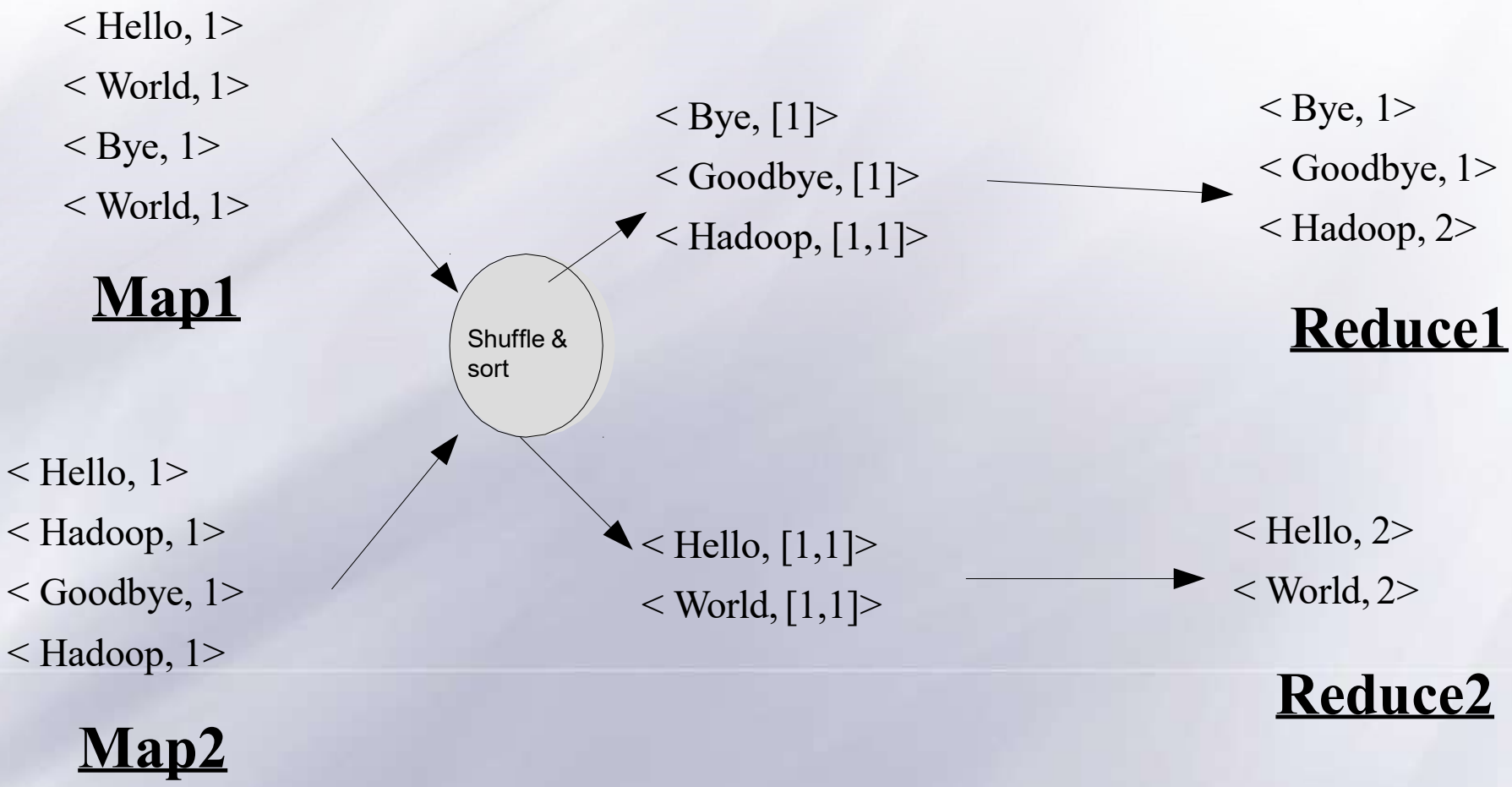
< Hadoop, [1,1]> → < Hadoop, 2>

< Hello, [1,1]> → < Hello, 2>

< World, [1,1]> → < World, 2>

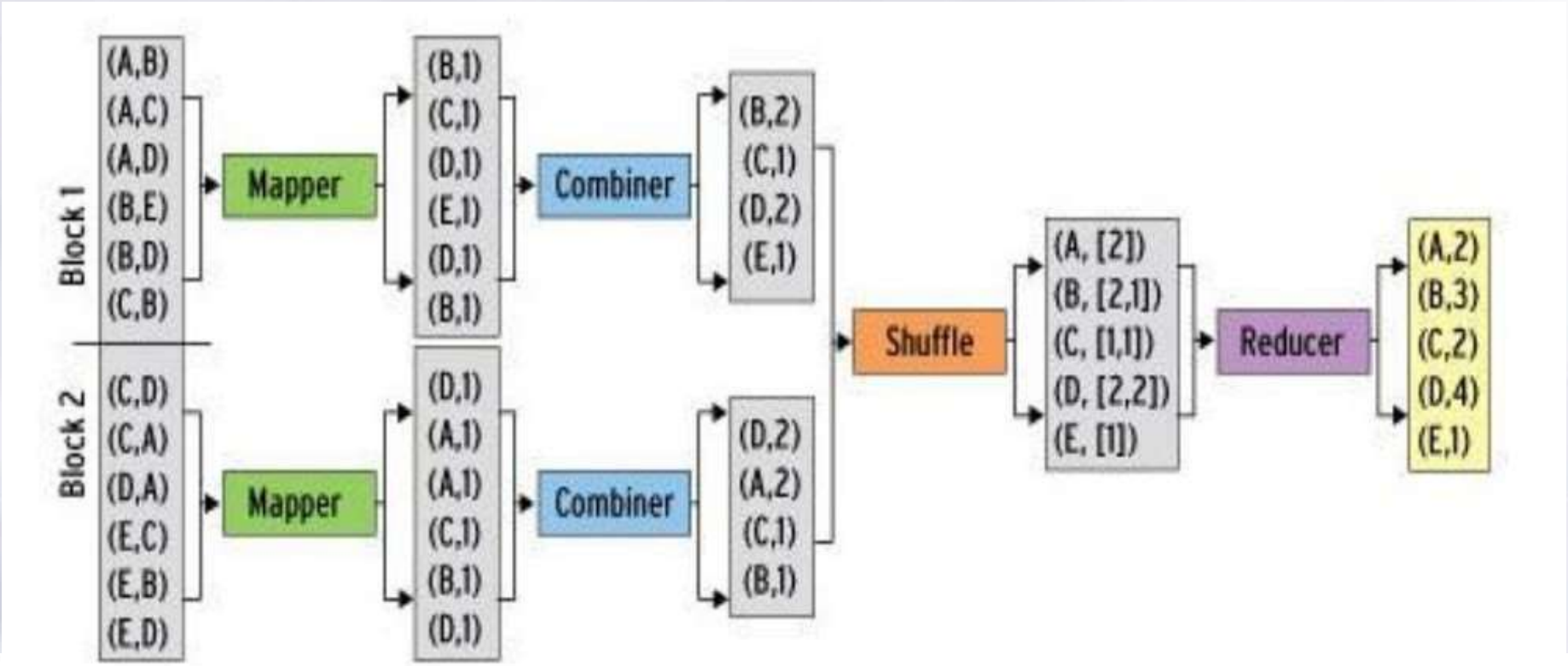
```
public static class MonIntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable>
{
    private IntWritable result = new IntWritable();

    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);
    }
}
```

Plusieurs opérations de réduction



Exemples de réductions

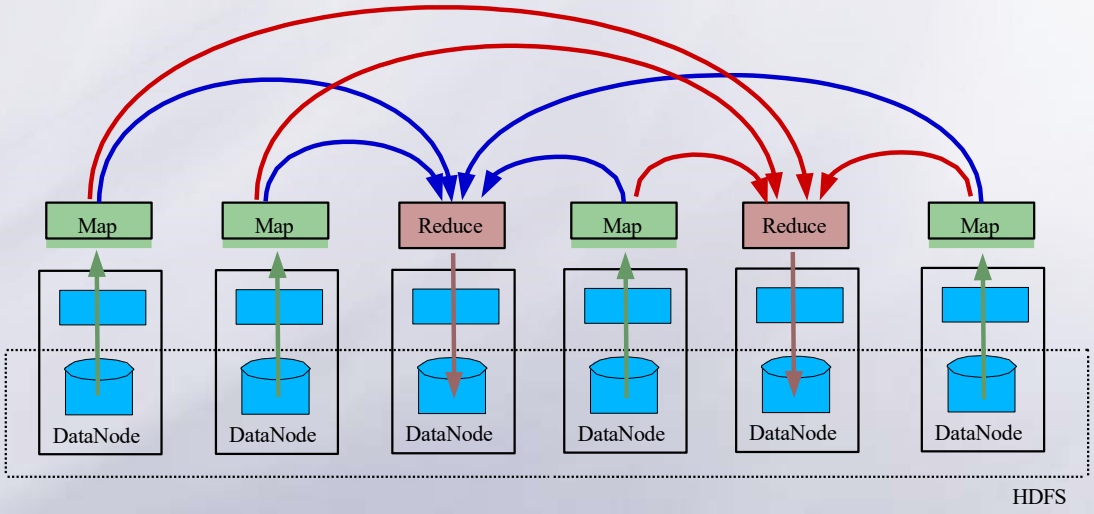
- **Réduire le transfert de données entre les Mappers et les Réducteurs**
  - Exécuté à la sortie du Mapper
  - Souvent la même fonction comme Réducteur



Programme principal

```
public class WordCount {
    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(MonTokenizerMapper.class);
    job.setCombinerClass(MonIntSumReducer.class);
    job.setReducerClass(MonIntSumReducer.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);
    }
}
```

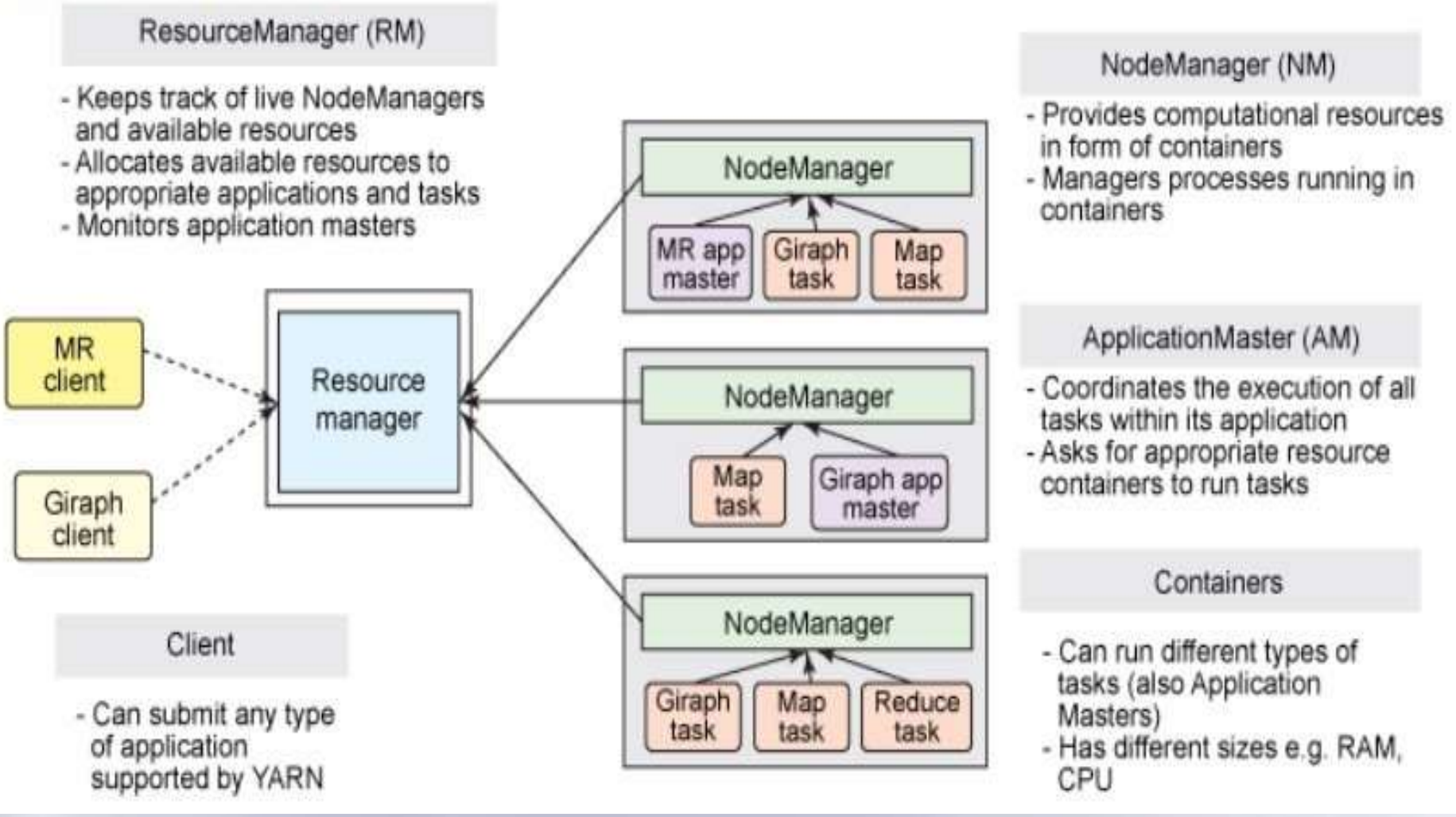
Execution dans un cluster



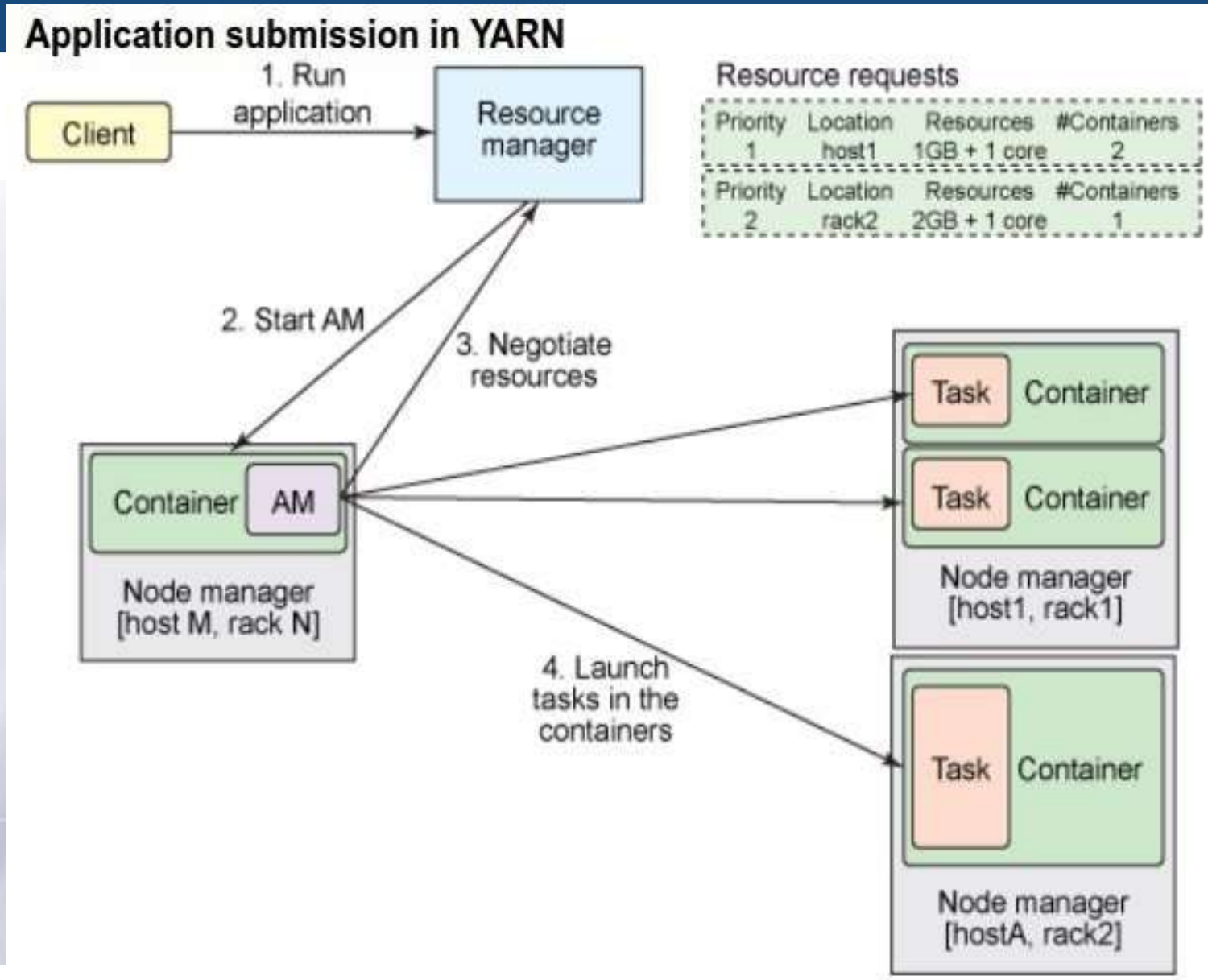
- HDFS s'exécute sur tous les nœuds : démon DataNode
- Chaque réduction génère un bloc de résultats stocké dans HDFS

Yarn : Yet Another Resource Negociator

Architecture of YARN



Yarn : Yet Another Resource Negociator



Hadoop en action

➢ Pre-requisite

- Java 8 installé (JAVA_HOME defined)
- Configurer ssh

➢ Installation d'hadoop

- tar xzf hadoop-X.Y.Z.tar.gz
- Définir les variables d'environment :

- export HADOOP_HOME=<path>/hadoop-X.Y.Z
- export PATH=$HADOOP_HOME/bin:$HADOOP_HOME/sbin:$PATH

Hadoop en action

➢ Dévelopement

- Sans eclipse
  - hadoop com.sun.tools.javac.Main <java-source-file>
- Avec eclipse
  - Créer un projet Java
  - Ajouter le jars dans le build path

    $HADOOP_HOME/share/hadoop/common/hadoop-common-X.Y.Z.jar
    $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-common-X.Y.Z.jar
    $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-X.Y.Z.jar
- Votre application doit être packagée dans un jar
  - jar cf wc.jar -C hadoop-wordcount/bin .

Hadoop – single node

- **Administration**
  - Format HDFS
    - hdfs namenode -format
  - Start HDFS
    - start-dfs.sh
    - vous pouvez alors vérifier avec jps que les daemons sont là (DataNode/NameNode/SecondaryNameNode)
- **File management in HDFS**
  - hdfs -put <local-file> <hdfs-file>
  - Autre commandes : get, cat, rm, mkdir, rmdir ...
- **Execution**
  - hadoop jar <jar-file> <java-class-name> <input-dir> <output-dir>

Hadoop – single node - exemple

- hdfs namenode -format
- start-dfs.sh
- jps
- hdfs dfs -mkdir /input
- hdfs dfs -put filesample.txt /input
- hadoop com.sun.tools.javac.Main WordCount.java
- jar cf wc.jar *.class
- hadoop jar wc.jar WordCount /input /output
- hdfs dfs -cat /output/*
- stop-dfs.sh

Hadoop – cluster mode - yarn

Configuration

- Fichiers à configurer dans <hadoop-home>/etc/hadoop
  - <hadoop-home>/etc/hadoop/hadoop-env.sh
  - <hadoop-home>/etc/hadoop/mapred-env.sh
    - JAVA_HOME
  - <hadoop-home>/etc/hadoop/core-site.xml
  - <hadoop-home>/etc/hadoop/hdfs-site.xml
  - <hadoop-home>/etc/hadoop/mapred-site.xml
  - <hadoop-home>/etc/hadoop/yarn-site.xml
  - <hadoop-home>/etc/hadoop/masters
    - The master node
  - <hadoop-home>/etc/hadoop/slaves
    - The slave nodes

Hadoop – cluster mode - yarn

- **Déploiment**
  - hdfs namenode -format
  - start-dfs.sh
  - start-yarn.sh
  - mr-jobhistory-daemon.sh --config <hadoop-home>/etc/hadoop start historyserver
- **Démarrage des daemons**
  - Sur les noeuds esclaves : un DataNode daemon (hdfs) et un NodeManager daemon (yarn)
  - Sur le noeud maître : un NameNode, un SecondaryNameNode daemon (hdfs) et un ResourceManager daemon (yarn)
- **Surveillance**
  - HDFS : master:50070
  - YARN : master:8088
  - JobHistory : master:19888

Merci de votre attention !