

# Hadoop : une plate-forme d'exécution de programmes Map-Reduce

Jonathan Lejeune

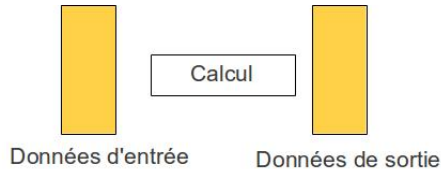
École des Mines de Nantes

Janvier 2015

CODEL 2014/2015



- calcul "data-intensive"
  - des volumes de données de plus en plus gros ( $> 1$  To)
    - exemples : parsing de documents, de pages web, de logs, etc.
    - accès à des flux de données : une écriture pour beaucoup de lectures



- Coût :
  - Du matériel de moins en moins cher et de plus en plus puissant
  - virtualisation et partage des ressources physiques  $\Rightarrow$  Cloud Computing  
= informatique à la demande

## Exemple de calcul : le word-count

- **En entrée** : un ou plusieurs (gros) fichiers textes
- **En sortie** : le nombre d'occurrences de chaque mot du texte

```
Apple Orange  
Banana Peach  
Orange Apple  
Strawberry  
Orange Apple
```

Données d'entrée

**word-count**

```
Apple 3  
Banana 1  
Peach 1  
Orange 3  
Strawberry 1
```

Données de sortie

## Solution word-count non-parallèle

En supposant que l'espace mémoire est suffisamment grand

```
word-count ( ) {  
  for each file f {  
    for each word w in f {  
      w_count[w]++;  
    }  
  }  
  save w_count to persistent storage  
}
```

Temps d'exécution très grand (jour, mois, année ?)

```
Mutex lock; // protects w_count
word-count ( ) {
    for each file f in parallel {
        for each word w in f {
            lock.Lock();
            w_count[w]++;
            lock.Unlock();
        }
    }
    save w_count to persistent storage
}
```

**Goulot d'étranglement due au mutex global**

- Un paradigme de programmation et un environnement d'exécution proposé par Google (2003)
- parallélisation de calcul manipulant de gros volumes de données dans des clusters de machines (centaines, milliers de CPU)
- un modèle de programmation fonctionnelle
  - aucune modification des structures de données
  - flux de données implicite
  - l'ordre des opérations n'a pas d'importance
- l'environnement d'exécution :
  - permet d'automatiser la parallélisation
  - gère les éventuelles fautes (pannes)
  - gère les communications entre les machines

## Une transparence pour le programmeur :

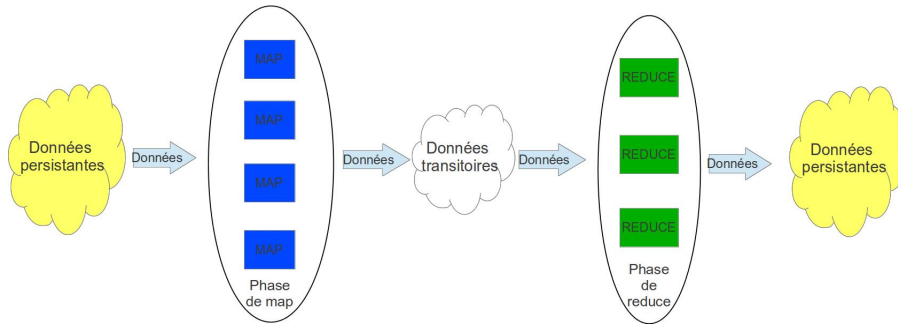
- une parallélisation automatique sur l'ensemble d'unités de calcul en terme de :
  - distribution des traitements
  - distribution des données
- équilibrage de charge
- stockage et transfert de données
- tolérance aux pannes
- ...

- La masse de données d'entrée est divisée en blocs appelés **split** qui seront traités par différentes tâches
- le calcul se fait en deux phases : **Map** et **Reduce**
- flux de donnée : un flux de paires de  $\langle \text{clé}, \text{valeur} \rangle$  entre :
  - les données d'entrée et la phase de map
  - phase de map et phase de reduce
  - la phase de reduce et les données de sortie.
- Un programme MapReduce peut se résumer à deux fonctions
  - la **fonction de map** : lit les données d'entrée, fait son traitement et produit une sortie
  - la **fonction de reduce** : lit les sorties produites par la phase de map, fait son traitement et produit les données de sortie

**Le programmeur doit juste fournir ces deux fonctions dans l'environnement d'exécution pour que son programme fonctionne !!!**



# Aperçu du flux de données



- Schéma classique d'un flux Map-Reduce
  - Lecture d'une grande quantité de données
  - **Map** : extraire une information qui vous intéresse
  - **Shuffle** : phase intermédiaire (cf. plus loin)
  - **Reduce** : agrège, filtre, transforme, etc.
  - Écriture du résultat
- En gardant ce schéma vous devez juste adapter vos fonction Map et Reduce en fonction du problème que vous voulez résoudre.

- un bloc de données (split) correspond à une tâche map
- Un élément du split (ex : ligne de fichier, tuple d'une base de données, Objet sérialisé, etc.) est associé à une clé de type K1
- A chaque clé de type K1 lue depuis le split, le processus map correspondant fait un appel à la fonction *map()*.
- la fonction *map()* produit dans le flux d'information une liste de <clé,valeur> intermédiaire de type <K2,V2>

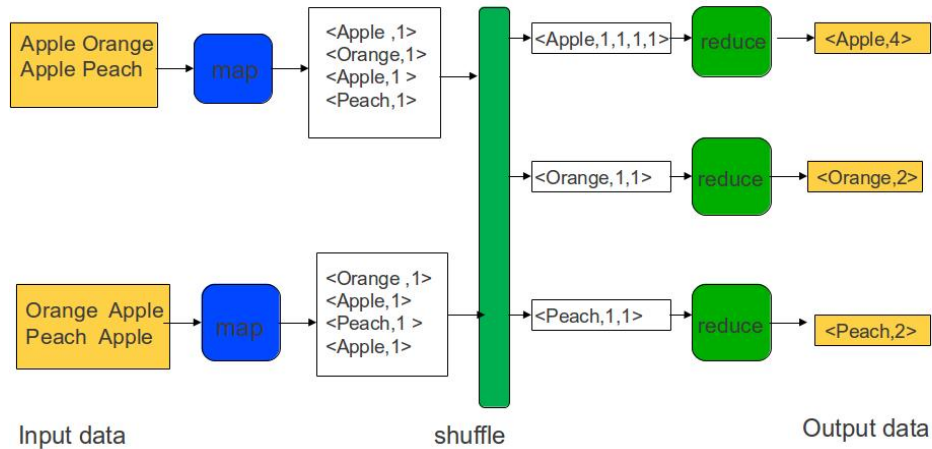
**Map : (K1,V1)  $\rightarrow$  list(K2,V2)**

- le nombre de reduces est défini a priori par l'utilisateur
- Une fois la phase de map terminée, agrégation en liste de toutes les valeurs intermédiaires de type  $V2$  associées à une clé de type  $K2$ .
- A chaque clé de type  $K2$  le processus reduce correspondant fait un appel à la fonction *reduce()*.
- la fonction *reduce()* produit dans le flux d'information une liste de  $\langle \text{clé}, \text{valeur} \rangle$  de type  $\langle K3, V3 \rangle$
- Chaque paire  $\langle K3, V3 \rangle$  émise est enregistrée dans l'ensemble de données de sortie

**Reduce :  $(K2, \text{list}(V2)) \rightarrow \text{list}(K3, V3)$**

Remarque : bien souvent  $K2 = K3$

## Exemple de flux de données avec le wordcount

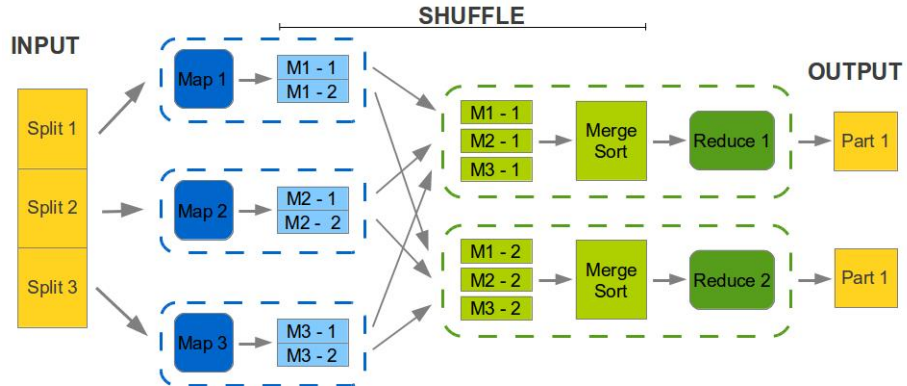


## Les fonctions Map et Reduce du wordcount

```
void Map(key, string value) {  
    // key : id of the line  
    // value : content of the line  
    for each word w in value {  
        EmitIntermediate(w, "1");  
    }  
}  
  
void Reduce(string key, list<string> values) {  
    // key: a word  
    // values : a list of contents  
    int count = 0;  
    for each v in values {  
        count += StringToInt(v);  
    }  
    Emit(key, IntToString(count));  
}
```

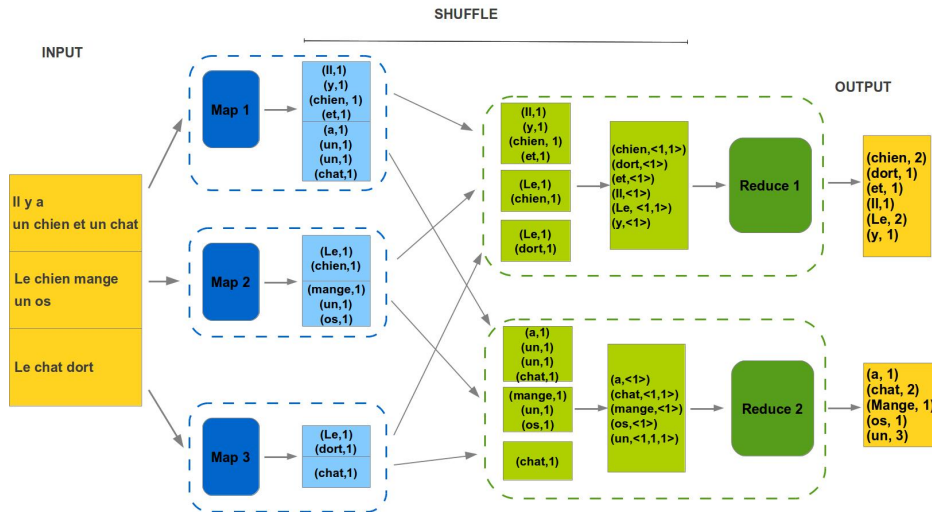
- transmission des données de la phase map vers la phase reduce
- Responsabilité des maps :
  - stockage local partitionné des couples clé/valeur de sortie
  - assignement déterministe des clés parmi *nbReduce* partitions (= **partitionnement**)  
⇒ une valeur de clé est associée à un unique reduce
- Responsabilité des reduces :
  - **copy**
    - téléchargement sur chaque map de la partition qui lui est associée
  - **merge**
    - agrégation de l'ensemble des partitions téléchargées
    - agrégation des valeurs pour une clé donnée
  - **sort**
    - tri des différentes clés définissant l'ordre de lecture par le reduce :  
⇒ **un ordre doit être défini pour chaque type de clé**

## IMPORTANT





# Flux de données détaillé du WordCount



- **Compteur de fréquence d'accès sur des URL**

- **Map** : des logs de requêtes de pages web en entrée et (URL,1) en sortie
- **Reduce** : ajoute les valeurs de chaque URL et sort le couple (URL, count)

Même principe que le WordCount :)

- **Index inversé**

- **Map** : parser chaque document et émettre une séquence de (mot, docId)
- **Reduce** : pour chaque mot, sortir directement la liste de valeur associée : (mot, list<docID>)

## Bref historique

- 2003 : Première librairie MapReduce écrite par Google
- 2004 : papier fondateur du MapReduce à OSDI  
[Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters]

## Plusieurs implémentations :

- **Apache Hadoop**
- Twister
- Skynetv
- Bash MapReduce
- etc.

plus d'informations : <http://en.wikipedia.org/wiki/MapReduce>

# La plate-forme Apache Hadoop MapReduce



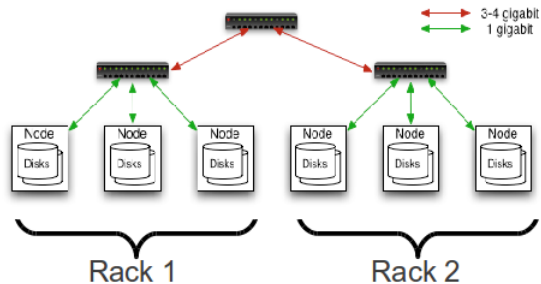
- Amazon (EMR ? Elastic MapReduce )
- Facebook
- IBM : Blue Cloud
- Joost (video distribution)
- Last.fm (free internet radio)
- New York Times
- PowerSet (search engine natural language)
- Veoh (online video distribution)
- Yahoo !

Plus d'information sur [http ://wiki.apache.org/hadoop/PoweredBy](http://wiki.apache.org/hadoop/PoweredBy)

- Plate-forme Map-reduce open-source écrite en JAVA
- créé par Doug Cutting en 2009 ( projets de la fondation Apache)
- un système de fichier distribué : Hadoop Distributed File System (HDFS)
- un ordonnanceur de programmes Map-Reduce
- une API Map-Reduce en JAVA, Python, C++
- une interface utilisateur
- une interface administrateur

# Architecture physique

- exécution sur une grille de machines
- Les machines sont groupées par rack



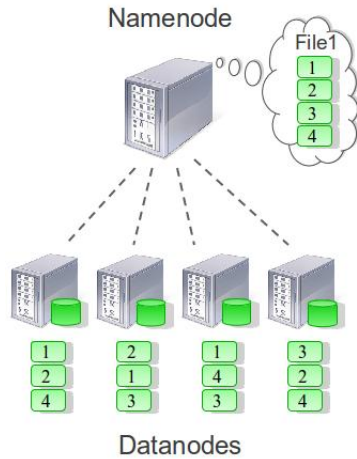
généralement une architecture à 2 niveaux

# HDFS : Hadoop Distributed File System

- Optimisé pour stocker de très gros fichiers
- les fichiers sont divisés en blocs (taille par défaut 128 Mo)
- Une architecture maître-esclave
  - le maître HDFS : le **Namenode**
  - les esclaves HDFS : les **Datanodes**
- les blocs sont stockés sur les Datanodes
- chaque bloc est répliqué sur différents Datanodes (par défaut 3 répliquas).
- lecture séquentielle
- écriture en mode append-only



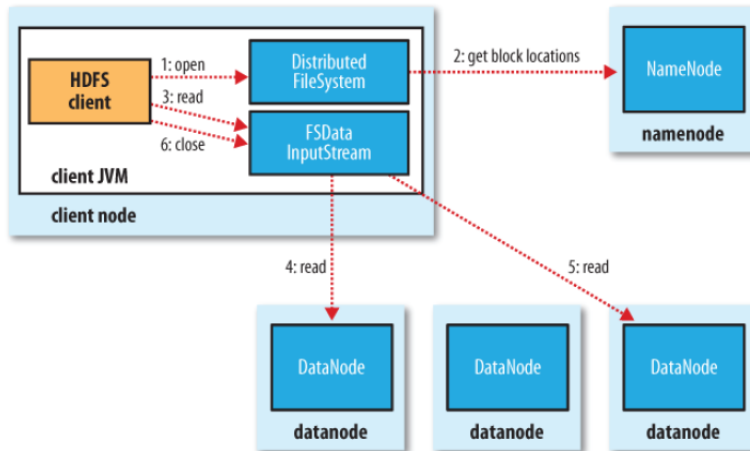
# HDFS : Schéma de stockage des blocs



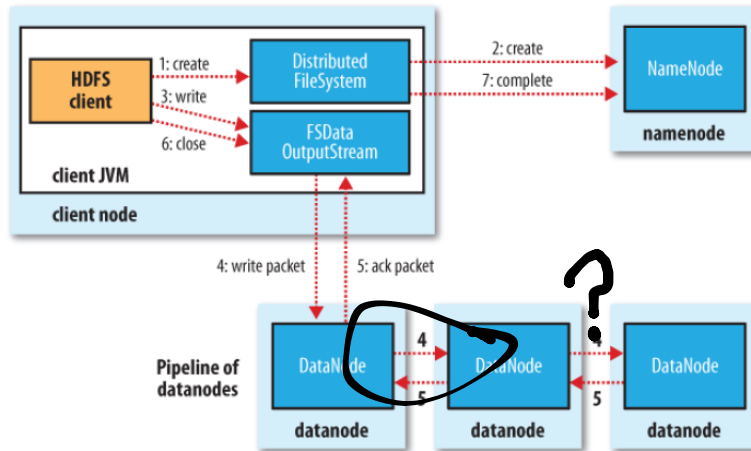
- responsable de la distribution et de la réplication des blocs
- Serveur d'informations du HDFS pour le client HDFS
- stocke et gère les méta-données :
  - liste des fichiers
  - liste des blocs pour chaque fichier
  - liste des Datanodes pour chaque bloc
  - attributs des fichiers (ex : nom, date de création, facteur de réplication)
- logs toute méta-donnée et toute transaction sur un support persistant
  - lectures/écritures
  - créations/suppressions
- démarre à partir d'une image de HDFS (fsimage)

- stocke des blocs de données dans le système de fichier local
- maintient des méta-données sur les blocs possédés (ex : CRC)
- serveur de bloc de données et de méta-données pour le client HDFS
- heartbeat avec le Namenode
  - message-aller vers le Namenode indiquant :
    - son identité
    - sa capacité totale, son espace utilisé, son espace restant
  - message-retour depuis le Namenode :
    - des commandes (copie de blocs vers d'autres Datanodes, invalidation de blocs, etc.)
- en plus du heartbeat, informe régulièrement le Namenode des blocs qu'il contient

# HDFS : Lecture d'un fichier



# HDFS : Écriture d'un fichier



- Le client :
  - récupère la liste des Datanodes sur lesquels placer les différents répliquas
  - écrit le bloc sur premier Datanode
- Le premier Datanode transfère les données au prochain Datanode dans le pipeline
- Quand tous les répliquas sont écrits, le client recommence la procédure pour le prochain bloc du fichier

Stratégie courante :

- un répliqua sur le rack local
- un second répliqua sur un autre rack
- un troisième répliqua sur un rack d'un autre datacenter
- les répliquas supplémentaires sont placés de façon aléatoire

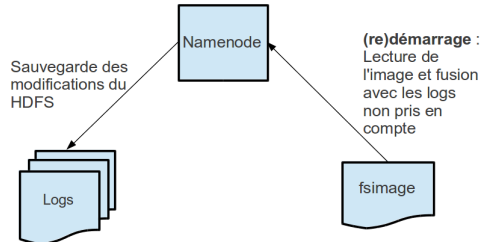
Le client lit le plus proche répliqua

## crash du Datanode :

- plus de heartbeat (détection par le Namenode)
- réplication distribuée (robustesse des données)

## crash du Namenode :

- sauvegarde des logs de transaction sur un support stable
- redémarrage sur la dernière image du HDFS et application des logs de transaction (recouvrement)





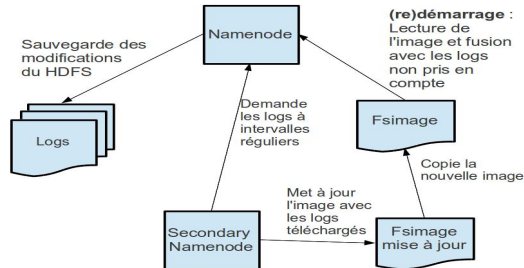
# HDFS : Secondary Namenode

Problème : rares redémarrages du Namenode

- ⇒ énorme quantité de logs de transaction : **stockage conséquent**
- ⇒ redémarrage long car prise en compte de beaucoup de changements

Solution : réduire la taille des logs

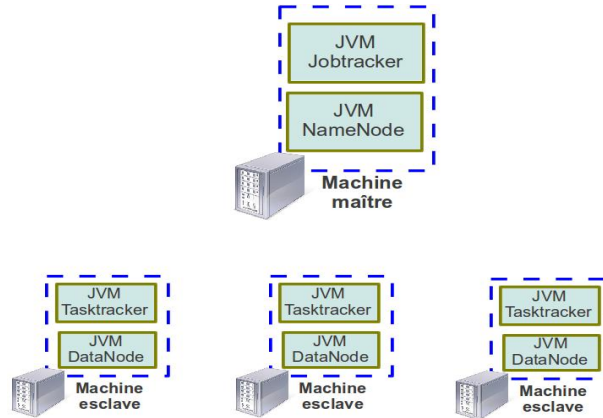
- utilisation d'un (ou plusieurs) processus distant : Le **Secondary NameNode** :
  - télécharge régulièrement les logs sur le Namenode
  - crée une nouvelle image en fusionnant les logs avec l'image HDFS
  - renvoie la nouvelle image au Namenode



- Gère les programmes clients Map-Reduce :
  - exécution
  - ordonnancement
  - déploiement
- Deux versions notables :
  - Version 0.x et 1.x : architecture purement maître-esclave
  - Version 2.x YARN : architecture maître-esclave à deux niveaux (stable depuis oct 2013)

# Hadoop Map-Reduce 1.x : architecture

- le maître Map-Reduce : le **Jobtracker**
- les esclaves Map-Reduce : les **Tasktrackers**



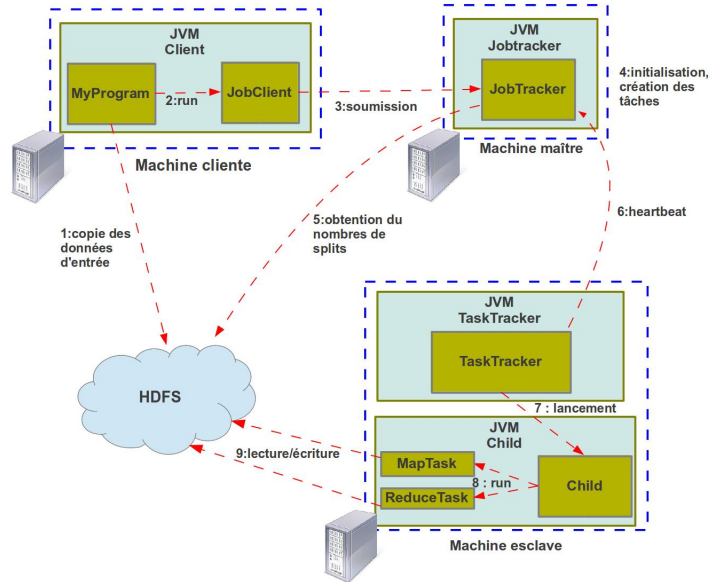
## Le jobtracker :

- gère l'ensemble des ressources du système
- reçoit les jobs des clients
- ordonnance les différentes tâches des jobs soumis
- assigne les tâches aux Tasktrackers
- réaffecte les tâches défaillantes
- maintient des informations sur l'état d'avancement des jobs

## Un tasktracker :

- exécute les tâches données par le Jobtracker
- exécution des tâches dans une autre JVM (**Child**)

# Hadoop Map-Reduce 1.x : Soumission d'un job



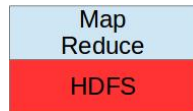
## Vers une deuxième version

Architecture trop centralisée de Hadoop 1.x :

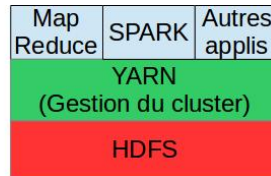
- vulnérable aux défaillances du JobTracker
- goulot d'étranglement (passage à l'échelle difficile)
- mélange de la gestion des ressources avec la gestion des jobs

Apport de Hadoop 2.x (YARN) :

- séparation de la gestion des ressources du cluster et de la gestion des jobs MapReduce
- généralisation de la gestion des ressources du cluster applicable à toute application



**Hadoop 1.x**



**Hadoop 2.x**

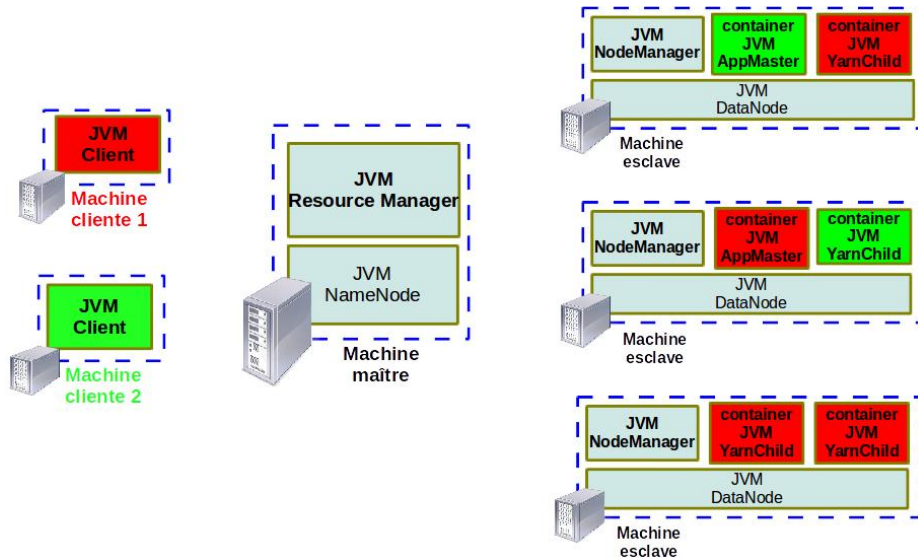
Répartition des fonctionnalités du Jobtracker :

- le **ResourceManager** (RM) :
  - JVM s'exécutant sur le nœud maître
  - contrôle toutes les ressources du cluster et l'état des machines esclaves
  - ordonnance les requêtes clientes
- un **ApplicationMaster** (AM) par application soumise (= 1 ou plusieurs Job)
  - JVM s'exécutant sur une machine esclave
  - négocie avec le RM les ressources nécessaires à l'application

Répartition des fonctionnalités du Tasktracker sur une machine esclave :

- un **NodeManager** (NM) :
  - une JVM par machine esclave
  - gère les ressources du nœud
- des **Containers** :
  - une abstraction de ressources sur un nœud, dédiée à
    - soit une JVM **ApplicationMaster**
    - soit une JVM **YarnChild** qui exécute une tâche Map ou Reduce

# Architecture de YARN avec 2 clients





- **Gérer le cluster en maximisant l'utilisation des ressources.**
- réception des requêtes clientes :
  - soumission et suppression d'applications
  - informations sur l'état courant des soumissions
- réception de requêtes administratives
- Gère l'ensemble des applications :
  - ordonnance et stocke les états des applications
  - alloue un container pour l'ApplicationMaster sur un NodeManager
  - redémarre l'application en cas de défaillance

- réception des requêtes de containers des ApplicationMaster
- Ordonnancement des requêtes de container (**YARN Scheduler**)
  - ordonnanceur modulable
  - par défaut : CapacityScheduler qui utilise un ordonnancement hiérarchique
- Connaissance de l'état de santé des NodeManagers
  - ressources allouées et disponibles sur chaque NM
  - Affectation des containers sur les NodeManagers
- gère la sécurité des communications
  - stocke et génère des clés de cryptage

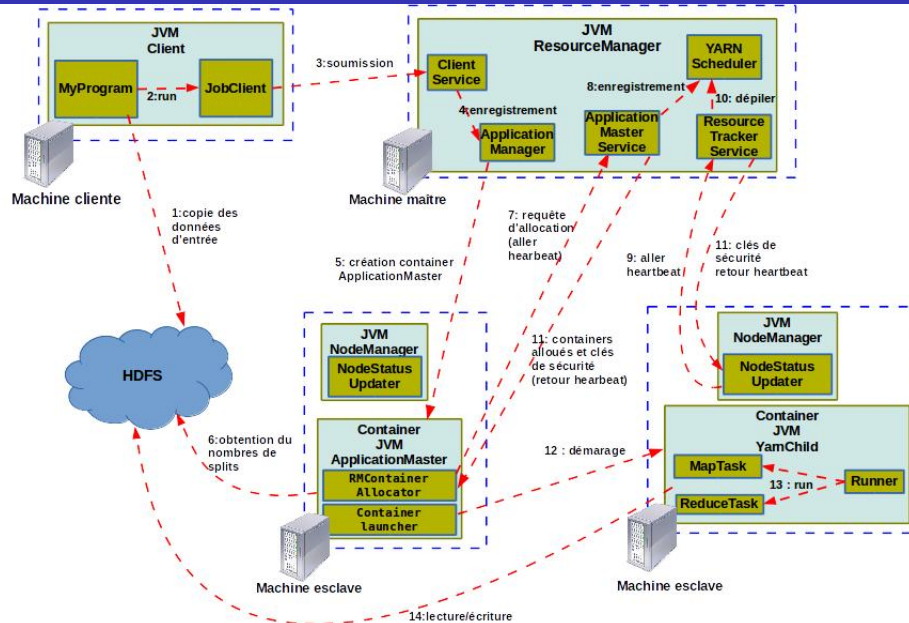
# YARN : Application Master

- négocie l'allocation des ressources et collabore avec les NodeManager pour utiliser les ressources acquises
- gère le cycle de vie de l'appli, les défaillances des containers, l'ajustement de la consommation des ressources
- heartbeat avec le ResourceManager
  - message aller :
    - progrès courant de l'application
    - containers demandés : nombre requis, préférence de localité
    - container libérés
  - message retour :
    - liste des containers nouvellement alloués et leur clé de sécurité
    - les ressources disponibles dans le cluster
    - possibilité de directive d'arrêt ou de resynchronisation
- Communique avec les NodeManager pour démarrer/arrêter les container alloués qui exécuteront les tâches de map et reduce (JVM YarnChild)

- héberge les containers
- libère les containers à la demande du RM ou du AM correspondant
  - libération des ressources (CPU,RAM)
  - possibilité de maintenir les données produite par la tache jusqu'à la fin de l'application
- monitore l'état de santé du nœud et des containers
- maintient des clés de sécurité provenant du RM afin d'authentifier les utilisation des containers
- heartbeat avec le RM :
  - message aller :
    - états des containers hébergés
    - états du nœud
  - message retour :
    - des clés de sécurités pour la communication avec les AM
    - des container à libérer
    - possibilité de directive d'arrêt ou de resynchronisation

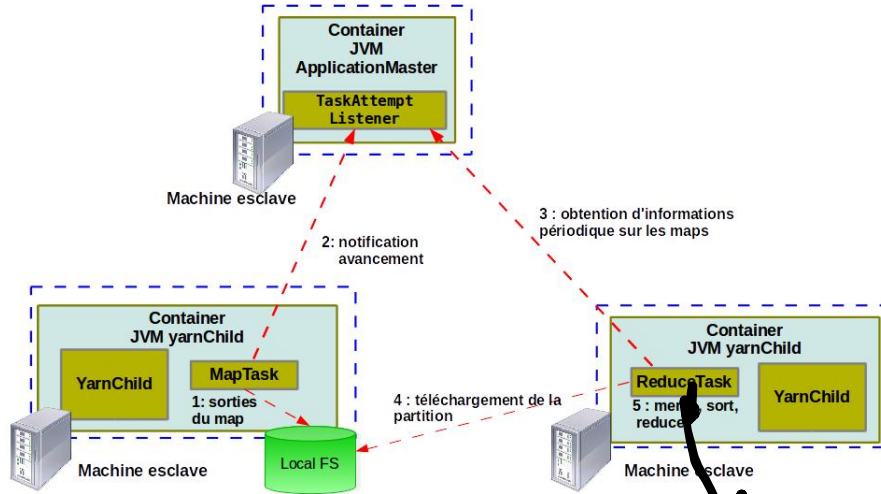
- L'ApplicationMaster définit à sa création le nombre de containers à demander au RM
  - la préférence de localisation des containers dépend de la localité des données d'entrée du job
- Choisi parmi la liste des tâches, par ordre de préférence :
  - **les tâches de map** : en les affectant les plus proches possibles de leur split respectif
  - **les tâches de reduce** : à partir d'une certaine proportion de maps terminés
- démarre une instance de la tâche choisie dans un container alloué par le RM
- Un container Yarn Child :
  - exécute une instance de tâche de map ou de reduce
  - fournit son état d'avancement à l'Application Master correspondant

# Soumission d'un Job Map Reduce sous YARN



- les sorties de maps sont stockées sur les systèmes de fichiers locaux des machines
- L'ApplicationMaster connaît l'avancement des tâches ainsi que leur localisation
- Un container Yarn Child demande périodiquement à L'ApplicationMaster des informations sur les tâches de maps en cours :
  - l'état d'avancement
  - la machine physique hébergeant les tâches
- Dès qu'un map se termine, les reduces concernés téléchargent directement leur partition sur la machine du map (phase de copie)
- une fois les données téléchargées, exécution des phases de merge et de sort

# framework Map-Reduce : schéma du shuffle




vers où?



## Crash du ResourceManager

- c'est une bonne question :) !

### crash d'un nœud esclave :

- le NodeManager n'émet plus de heartbeat au RM
  - Le RM supprime le nœud de ses esclaves opérationnels tant qu'il n'a pas redémarré 
- les containers YarnChild n'émettent plus de status à leur AppMaster
  - Au bout de 3 crash, l'AppMaster Blackliste le nœud et demande au RM l'allocation des containers sur un autre esclave
- les AppMaster n'émettent plus de heartbeat au RM
  - redémarrage d'une nouvelle instance de l'application en créant un nouvel AppMaster
- mécanisme de recovery au redémarrage du NM

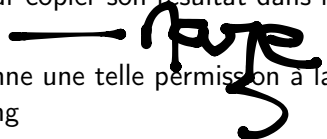
**Possibilité de démarrer de nouvelles instances de tâche en cours d'exécution :**

- Estimation faite par l'ApplicationMaster :
  - si la spéculation est jugée nécessaire, affectation d'une nouvelle instance de tâche dans un container.

**Intérêt : Permettre d'anticiper les tâches jugées trop lentes et donc potentiellement défailante**

**Écriture du résultat du calcul sur le HDFS par les reduces (un fichier par tâche de reduce)**

**Problème :** Comment assurer une cohérence puisque plusieurs instance d'une même tâche peuvent s'exécuter en même temps (ex : spéculation) ?  
**solution :**

- chaque instance écrit dans un fichier portant leur TaskAttemptId dans un dossier temporaire (créé préalablement par l'ApplicationMaster)
- lorsqu'une instance se termine, attendre la permission de l'ApplicationMaster pour copier son résultat dans le répertoire définitif (état CommitPending) — 
- l'ApplicationMaster donne une telle permission à la première instance détectée CommitPending
- envoi de l'ordre de commit par le message retour du heartbeat
- les autres instances sont détruites après la fin du commit

- les valeurs doivent implémenter l'interface *Writable* de l'API Hadoop
- les clés doivent implémenter l'interface *WritableComparable*  $< T >$  (interface implémentant *Writable* et *Comparable*  $< T >$  de Java)
- *Writable* contient deux méthodes :
  - *void write(DataOutput out)* : sérialisation
  - *void readFields(DataInput in)* : dé-sérialisation

Des Writables prédéfinis dans l'API :

*BooleanWritable*, *DoubleWritable*, *FloatWritable*, *IntWritable*,  
*LongWritable*, *Text*, etc.

- permet de définir le format des données dans :
  - **les données d'entrée** : toute classe héritant de *InputFormat*  $\langle K, V \rangle$ .  
Classe fournies par l'API Hadoop principalement utilisées :
    - *TextInputFormat*
    - *SequenceFileInputFormat*  $\langle K, V \rangle$
    - *KeyValueTextInputFormat*
  - **les données de sortie** : toute classe héritant de *OutputFormat*  $\langle K, V \rangle$ .  
Classe fournies par l'API Hadoop principalement utilisées :
    - *TextOutputFormat*
    - *SequenceFileOutputFormat*  $\langle K, V \rangle$

# Programmation Map Reduce : programme du Mapper

- classe héritant de la classe  
*Mapper* < *KEYIN*, *VALUEIN*, *KEYOUT*, *VALUEOUT* >
- surcharge de la méthode **map**, appelée à chaque lecture d'une nouvelle paire clé/valeur dans le split :

```
protected void map(KEYIN key, VALUEIN value, Context context)
```

Exemple : Mapper du WordCount

```
public class TokenizerMapper extends  
    Mapper<Object, Text, Text, IntWritable>{  
  
    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(new Text(word), new IntWritable(1));  
        }  
    }  
}
```

## Programmation Map Reduce : programme du Reducer

- classe héritant de la classe  
*Reducer* < *KEYIN*, *VALUEIN*, *KEYOUT*, *VALUEOUT* >
- surcharge de la méthode **reduce**, appelée à chaque lecture d'une nouvelle paire clé/list(valeur) :

```
protected void reduce(KEYIN key, Iterable<VALUEIN> values,  
                      Context context)
```

Exemple : Reducer du WordCount

```
public class IntSumReducer  
    extends Reducer<Text, IntWritable, Text, 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, new IntWritable(sum));  
    }  
}
```

## Programmation Map Reduce : partitionneur

- définit la politique de répartition des sorties de map
- classe héritant de la classe abstraite *Partitioner*  $\langle KEY, VALUE \rangle$
- implémentation de la méthode abstraite **getPartition**, appelée par le *context.write()* du map :

```
public abstract int getPartition(KEY key, VALUE value,
                                int numPartitions);

/* key : l'objet type clé
value : l'objet type valeur
numPartitions : nombre de partition (= nombre de reduce)
*/
```

Le partitionneur par défaut de Hadoop est un hachage de la clé (*HashPartitioner*) :

```
public class HashPartitioner<K, V> extends Partitioner<K, V>{
    public int getPartition(K key, V value,
                            int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```



## Programmation Map Reduce : squelette client

```
public class MyProgram {  
    public static void main(String[] args){  
        Configuration conf = new Configuration();  
        Job job = Job.getInstance(conf, "Mon_Job");  
        job.setJarByClass(MyProgram.class); //jar du programme  
        job.setMapperClass(...); // classe Mapper  
        job.setReducerClass(...); // classe Reducer  
        job.setMapOutputKeyClass(...); // classe clé sortie map  
        job.setMapOutputValueClass(...); // classe valeur sortie map  
        job.setOutputKeyClass(...); // classe clé sortie job  
        job.setOutputValueClass(...); //classe valeur sortie job  
        job.setInputFormatClass(...); // classe InputFormat  
        job.setOutputFormatClass(...); //classe OutputFormat  
        job.setPartitionerClass(HashPartitioner.class); //partitioner  
        job.setNumReduceTasks(...); // nombre de reduce  
        FileInputFormat.addInputPath(job, ...); //chemins entrée  
        FileOutputFormat.setOutputPath(job, ...); //chemin sortie  
        job.waitForCompletion(true); //lancement du job  
    }  
}
```

# Programmation Map Reduce : API du HDFS

- possibilité de manipuler le HDFS depuis le programme client
- HDFS représenté par un objet FileSystem unique (singleton) :

```
final FileSystem fs = FileSystem.get(conf);
```

Quelques méthodes :

```
//copie d'un fichier/dossier local client vers le HDFS
fs.copyFromLocalFile(src,dst);
//copie d'un fichier/dossier HDFS vers le fs local client
fs.copyToLocalFile(sr,dst);
//création/écrasement d'un fichier
FSDataOutputStream out = fs.create(f);
//test d'existence d'un fichier
boolean b = fs.exists(f);
//ouverture en ecriture (append only)
FSDataOutputStream out = fs.append(f);
//ouverture en lecture
FSDataInputStream in = fs.open(f);
//le reste : RTFM
```

## Commande HDFS dans un terminal

- possibilité de manipuler le HDFS depuis un terminal
- commande :

```
hdfs dfs <commande HDFS>
```

- commandes HDFS similaires celles d'Unix en ajoutant un tiret devant

Exemples :

```
hdfs dfs -ls <path>
```

```
hdfs dfs -mv <src> <dst>
```

```
hdfs dfs -cp <src> <dst>
```

```
hdfs dfs -cat <src>
```

```
hdfs dfs -copyFromLocal <localsrc> ... <dst>
```

```
hdfs dfs -mkdir <path>
```

```
hdfs dfs -copyToLocal <src> <localdst>
```

RTFM pour les autres :).

# Projets Apache utilisant Hadoop

- **Hbase** : SGBD optimisé pour les grandes tables utilisant le HDFS
- **Pig et Hive** : logiciels d'analyse de données permettant d'utiliser Hadoop avec une syntaxe proche du SQL



- **Mahout** : création d'implémentations d'algorithmes d'apprentissage automatique distribués



- **Zookeeper** : service de coordination pour applications distribuées
- **Chukwa** : système de monitoring de gros systèmes distribués
- **Spark** : alternative 100 fois plus performante au Map Reduce pour certaines applications



[1] Jeffrey Dean and Sanjay Ghemawat, MapReduce : Simplified Data Processing on Large Clusters, OSDI'04 : Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[2] Hadoop : the definitive guide, White Tom, O'Reilly, 2012, ISBN : 978-1-4493-8973-4

[3] Hadoop web site : [http ://hadoop.apache.org/](http://hadoop.apache.org/)