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

Jonathan Lejeune

UPMC

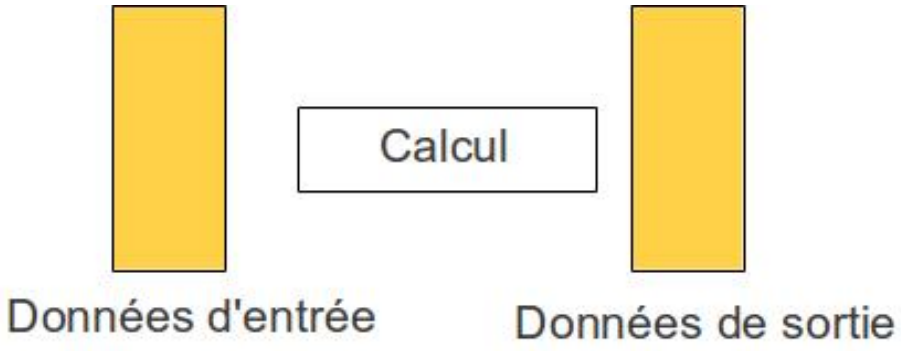
8 octobre 2013

PSIA 2013

Inspiré du cours des années précédentes de Luciana Arantes

Motivations

- calcul "data-intensive"
  - des volumes de données de plus en plus gros (> 1 To)
    - exemples : parsage 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

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 ?)**

Solution word-count parallèle

```
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**

le Map-Reduce

- 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

le Map-Reduce

Une transparence pour le programmeur :

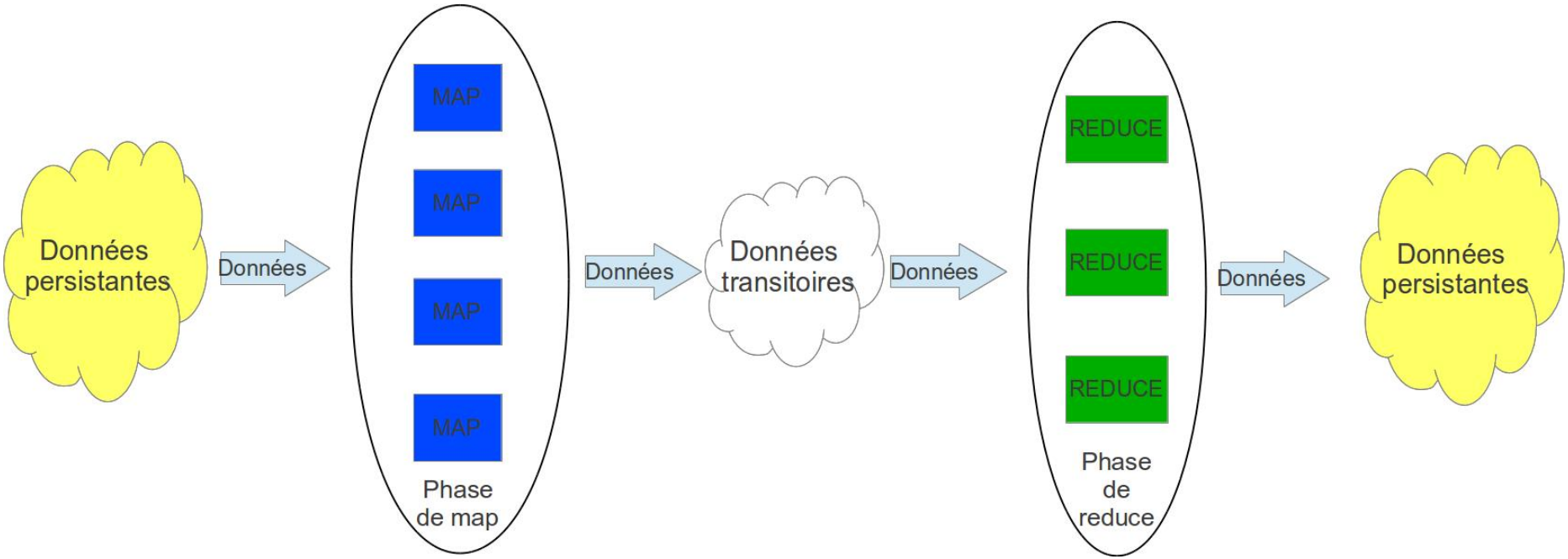
- une parallélisation automatique sur l'ensemble d'unités de calcul en terme de :
  - distribution des traitements
  - distribution des données
- équilibrage de charge
- stockage et transfert de données
- tolérance aux pannes
- ...

le Map-Reduce

- 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 <clé,valeur> 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



le Map-Reduce

- 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 adaptez vos fonction Map et Reduce en fonction du problème que vous voulez résoudre.

la phase de Map

- 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.) = 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>

$$\textbf{Map : (K1,V1)} \rightarrow \textbf{list(K2,V2)}$$

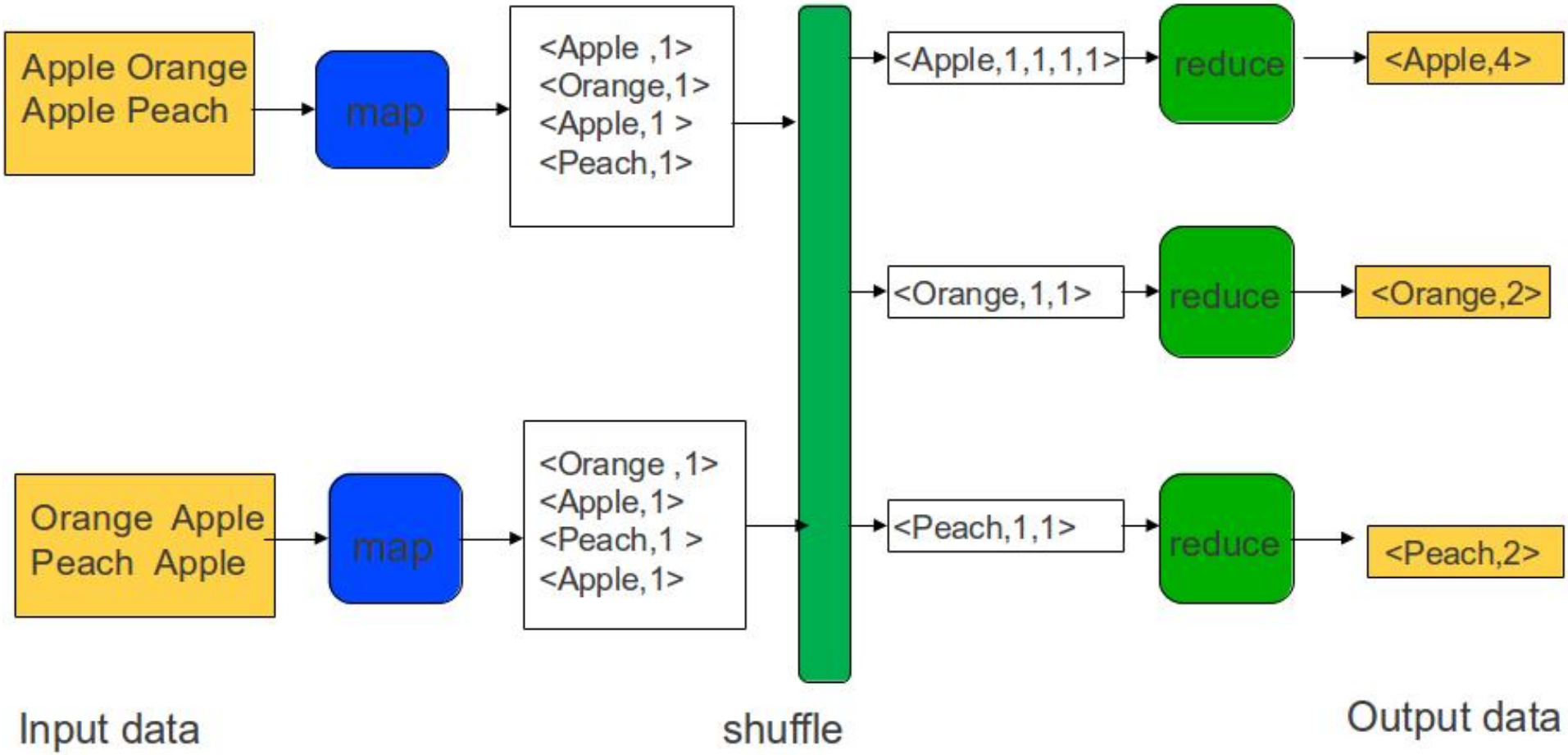
la phase de Reduce

- 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 <clé,valeur> de type <K3,V3>
- Chaque paire <K3,V3> émise est enregistrée dans l'ensemble de données de sortie

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

Remarque : bien souvent K2 = K3

Exemple de flux de données avec le wordcount



Les fonctions Map et Reduce du wordcount

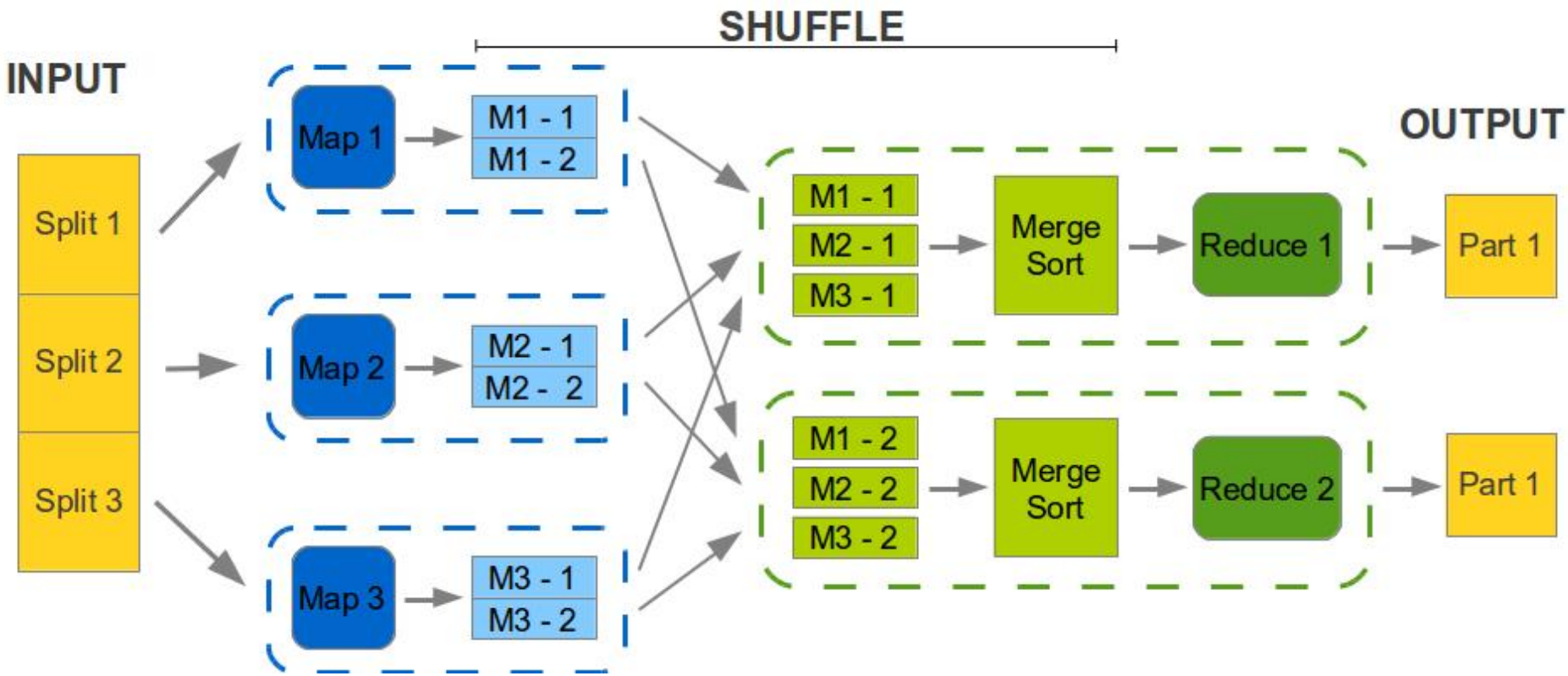
```
void Map(key, string value) {
    // key : id of the line
    // value : content of the line
        for each word w in value {
            EmitIntermediate(w, "1");
        }
 }

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));
}
```

la phase de Shuffle

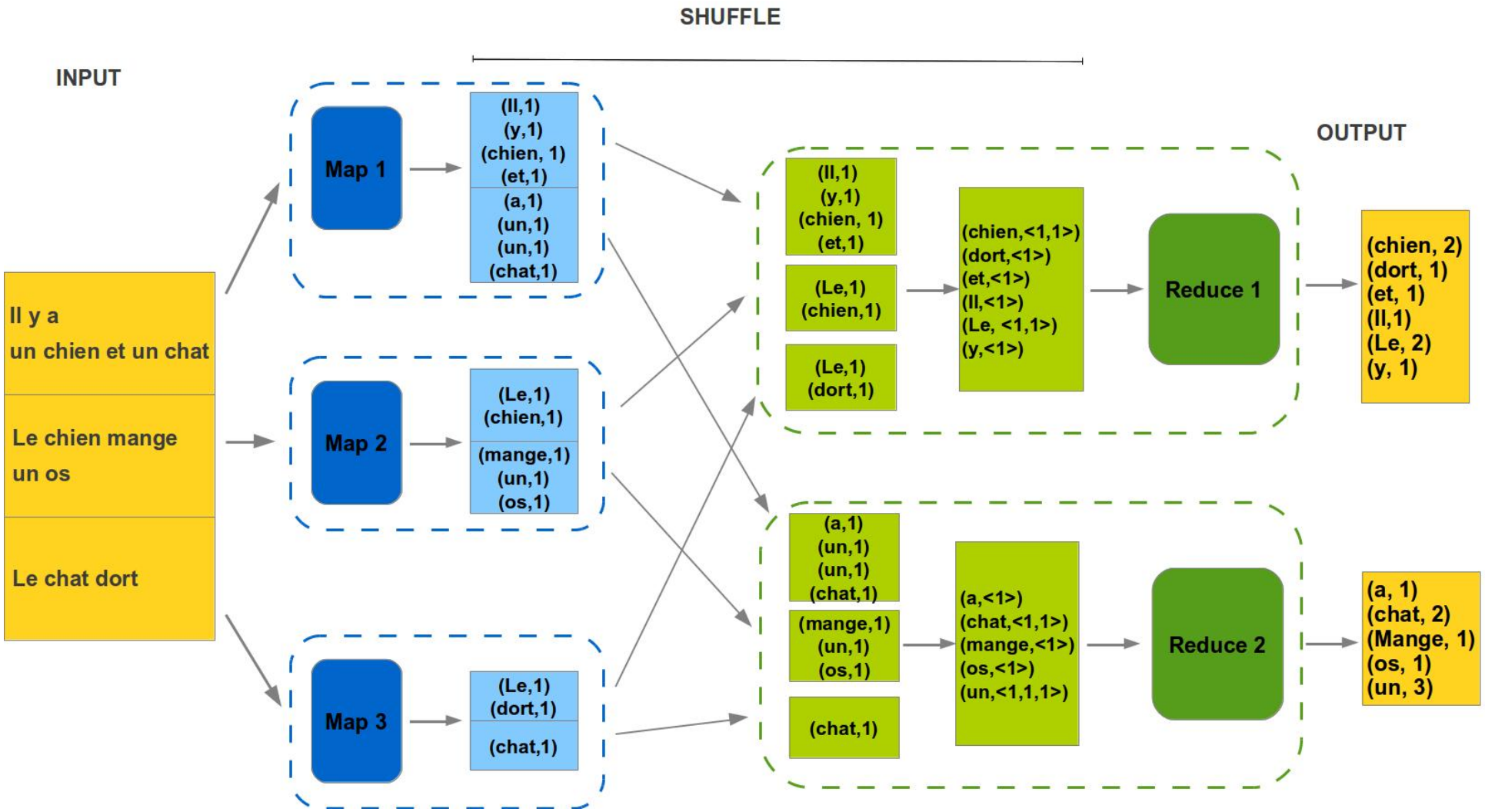
- transmission des données de la phase map vers la phase reduce
- Responsabilité des maps :
  - stockage local partitionné des couples clé/valeur de sortie
  - assignement déterministe des clés parmi *nbReduce* partitions (= **partitionnement**)
    $\Rightarrow$ une valeur de clé est associée à un unique 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 :
      $\Rightarrow$ **un ordre doit être défini pour chaque type de clé**

Flux de données détaillé

IMPORTANT



Flux de données détaillé du WordCount



Autres exemples

- **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>)

Environnement d'exécution

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

Utilisateurs d'Hadoop

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

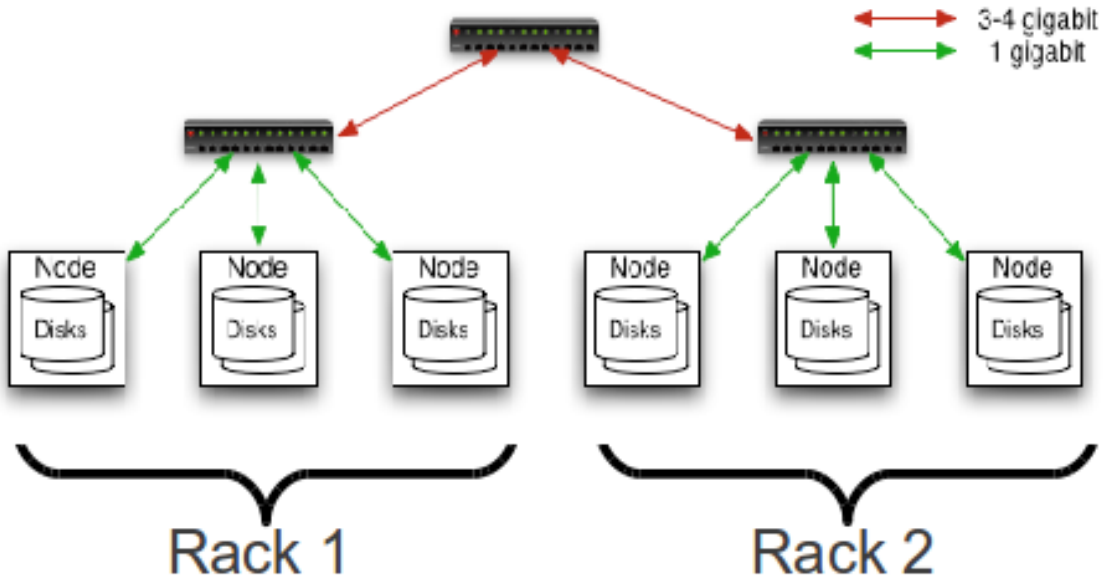
Plus d'information sur http ://wiki.apache.org/hadoop/PoweredBy

Aperçu d'Hadoop

- 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

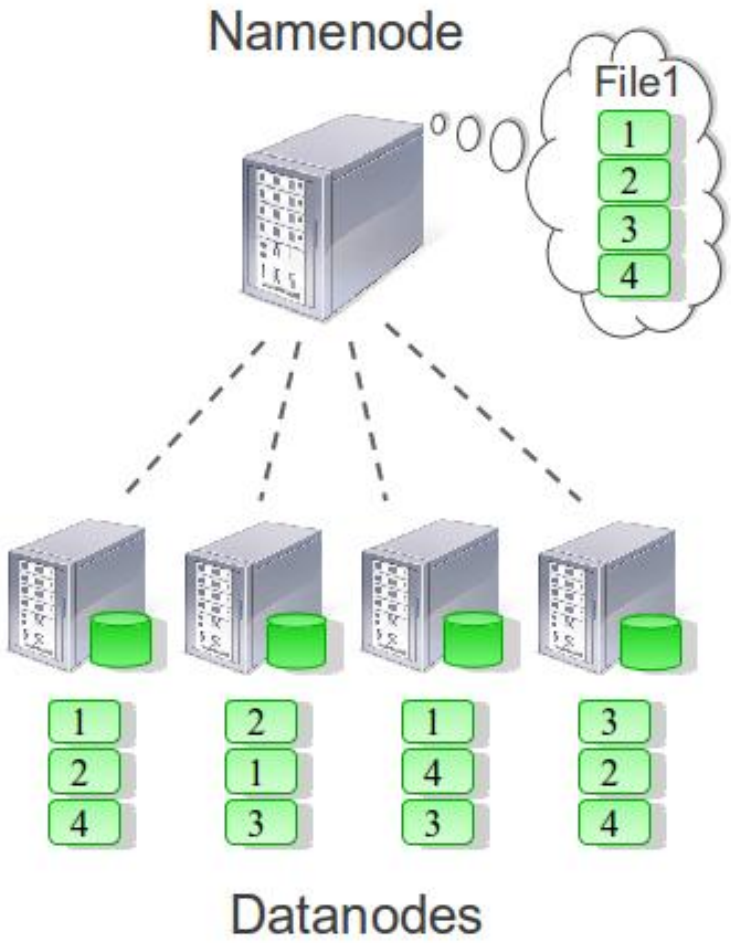


généralement une architecture à 2 niveaux

HDFS : Hadoop Distributed File System

- Optimisé pour stocker de très gros fichiers
- les fichiers sont divisés en blocs (taille par défaut 64 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



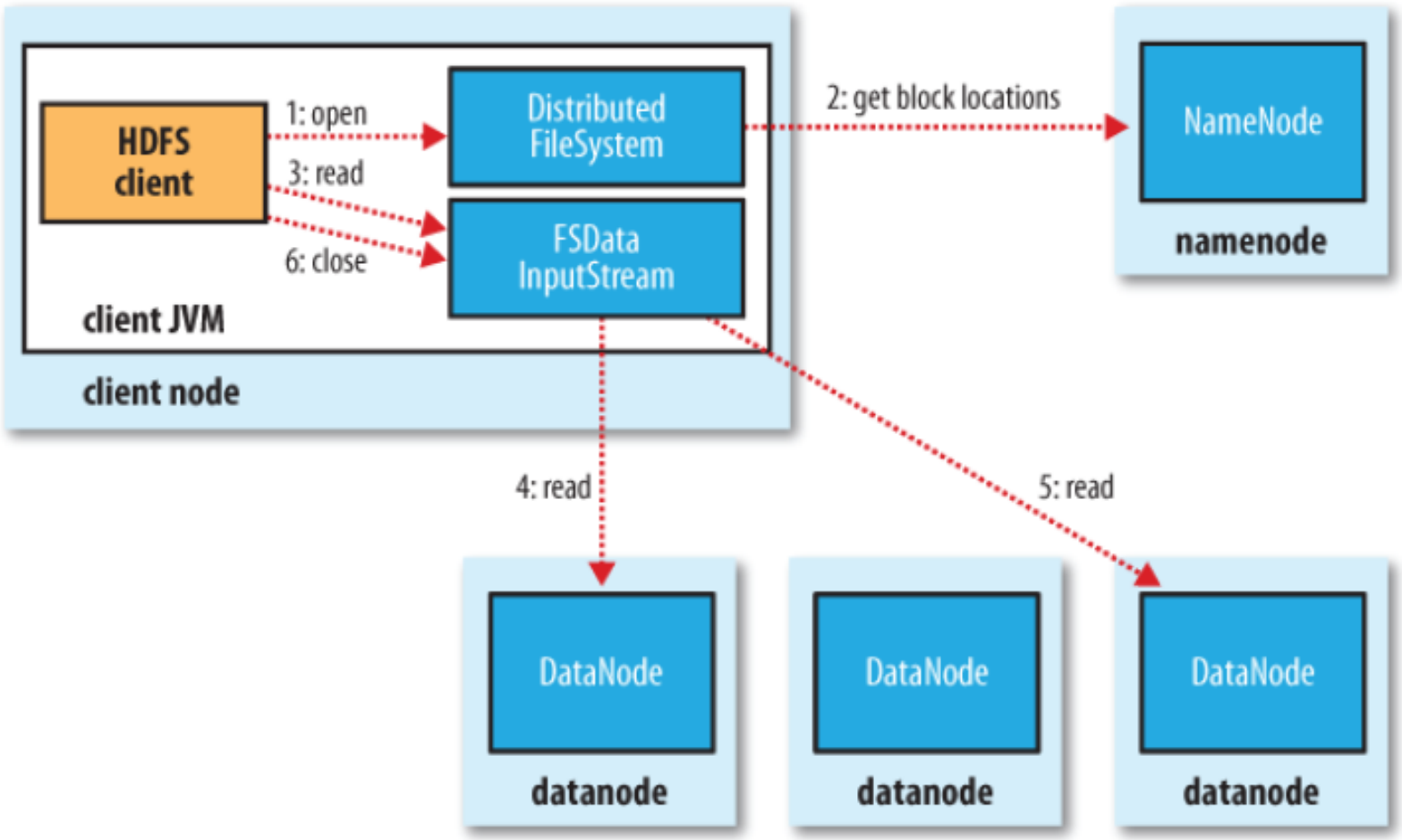
HDFS : le Namenode

- 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)

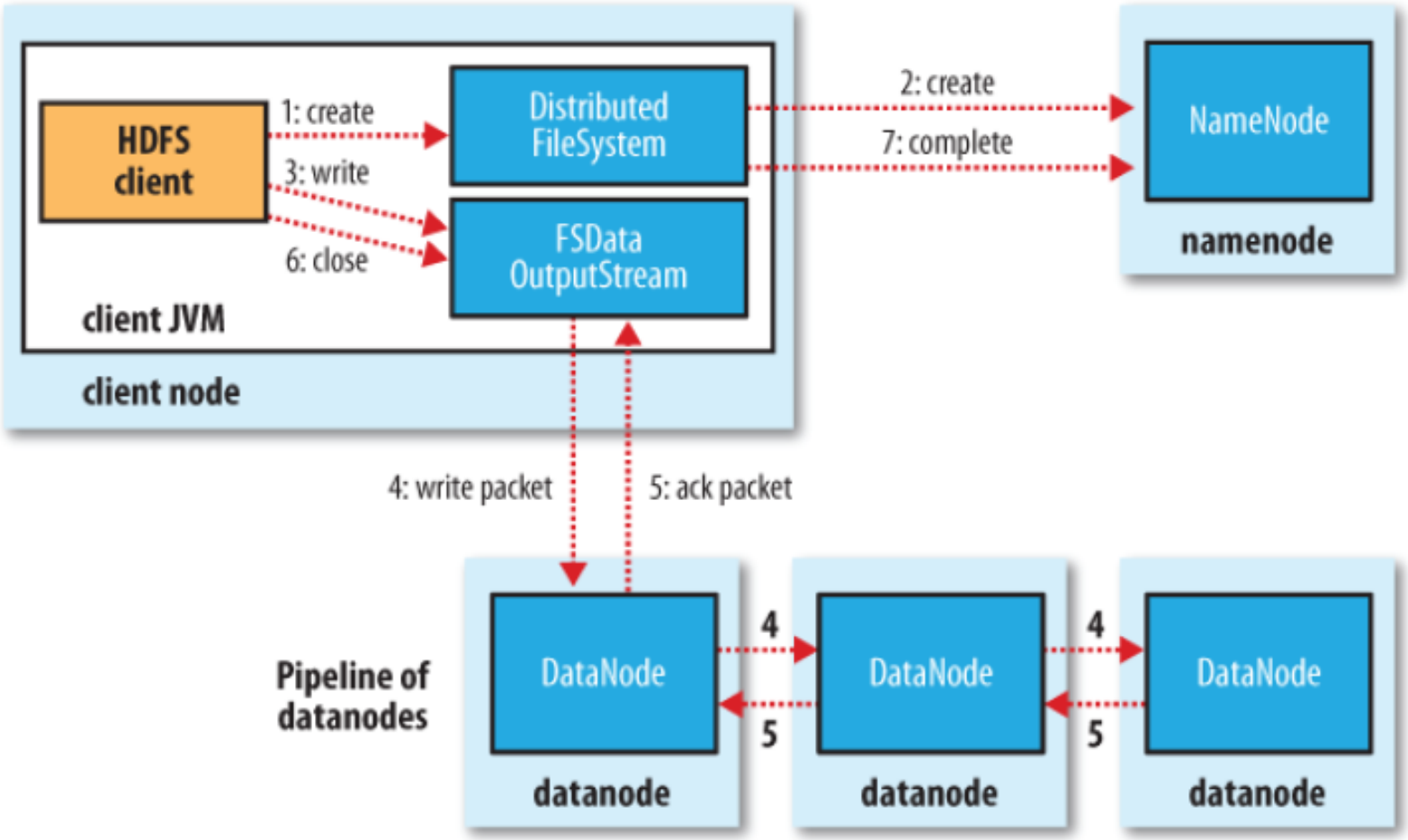
HDFS : le datanode

- 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



HDFS : écriture et pipeline de données

- 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

HDFS :stratégie de placement des blocs

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

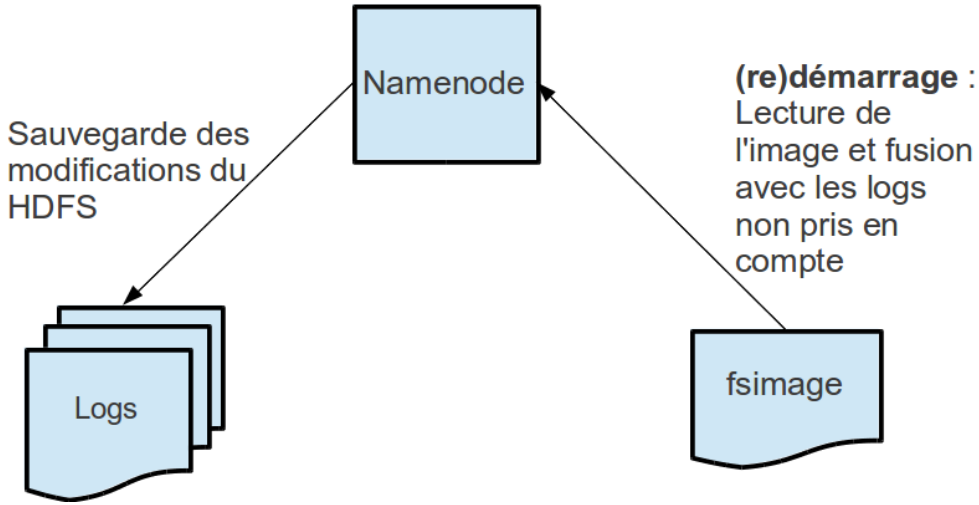
HDFS : tolérance aux fautes

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

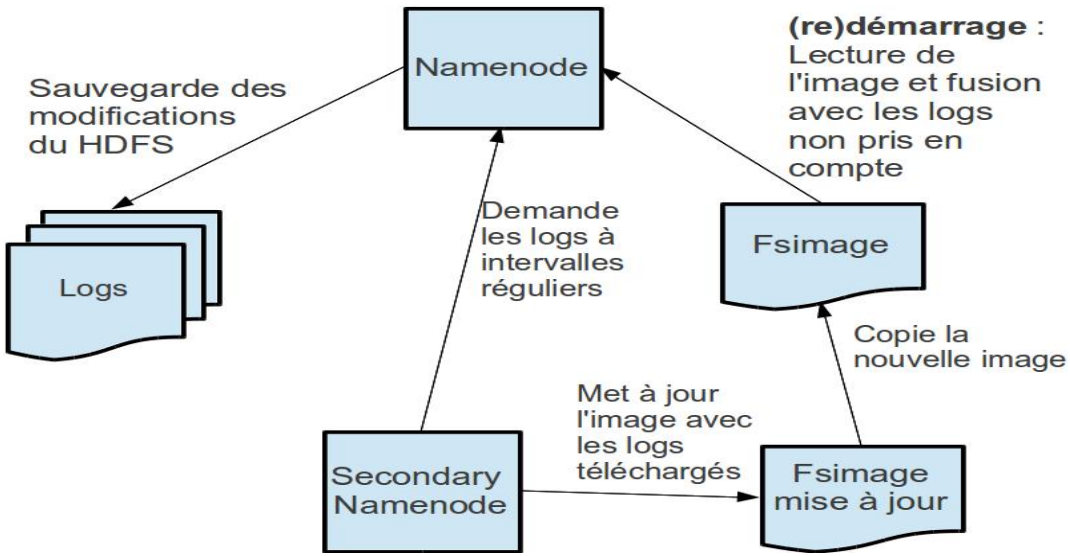
Problème : rares redémarrages du Namenode

⇒ 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

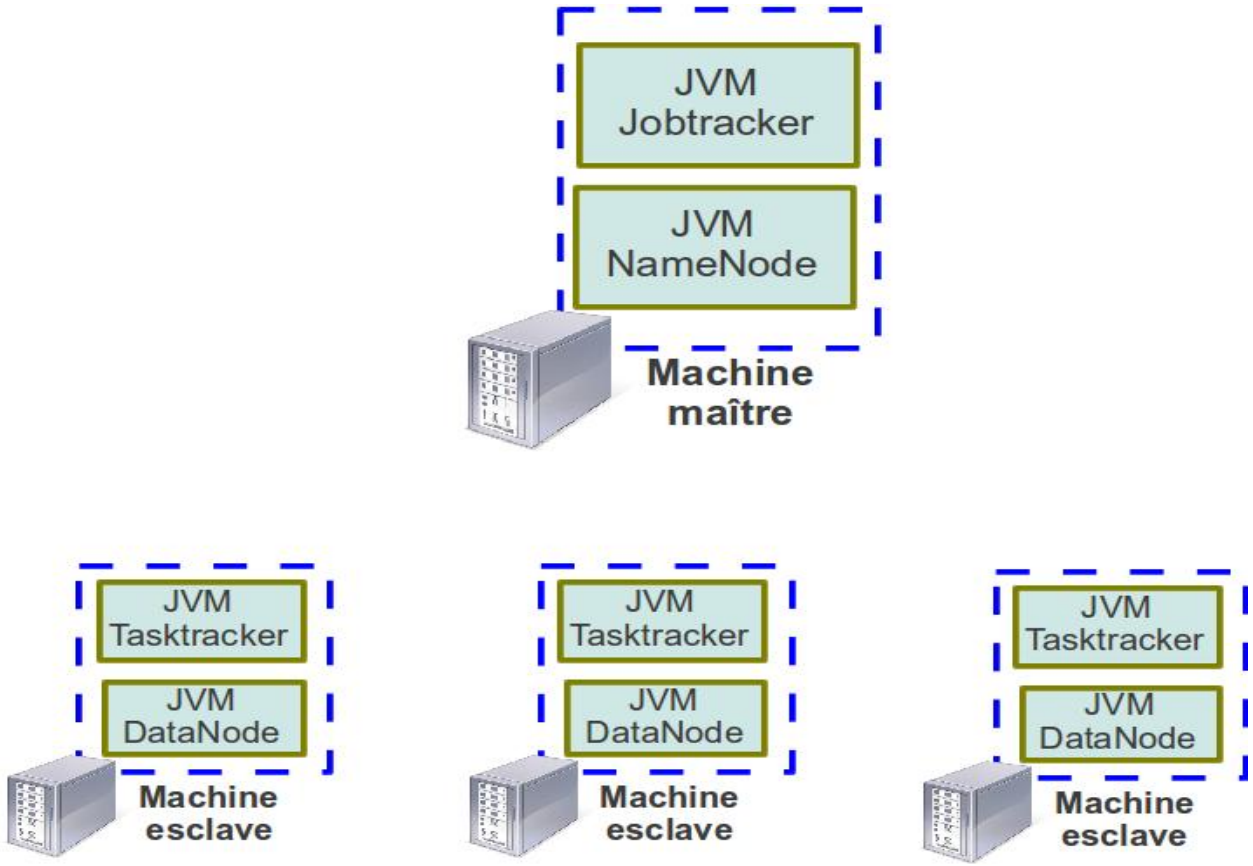


framework Map-Reduce

- Gère les programmes clients Map-Reduce
  - exécution
  - ordonnancement
  - déploiement
- Une architecture maître-esclave
  - le maitre Map-Reduce : le **Jobtracker**
  - les esclaves Map-Reduce : les **Tasktrackers**

framework Map-Reduce et HDFS

- exécution sur une même machine physique du **Jobtracker** et du **Namenode**
- exécution sur une même machine physique d'un **Tasktracker** avec un **Datanode**



framework Map-Reduce : le Jobtracker

- reçoit les jobs des clients
- détermine les splits des données d'entrée d'un job
- crée les tâches de map en fonction du nombre de splits
- crée les tâches de reduce en fonction de la configuration du job
- assigne les tâches aux Tasktrackers
- réaffecte les tâches défaillantes
- maintient des informations sur l'état d'avancement des jobs

framework Map-Reduce : le tasktracker

- exécute les tâches données par le Jobtracker
- exécution des tâches dans une autre JVM (**Child**)
- a une capacité en termes de nombres de tâches qu'il peut exécuter simultanément (**slot**) :
  - paramétré par l'administrateur de la plate-forme
  - dépend du nombre de cœurs et de la capacité mémoire de la machine physique
- heartbeat avec le Jobtracker
  - message-aller :
    - demande ou non de nouvelle tâches en fonction des slots disponibles
    - envoi de diverses informations (avancement des tâches en cours, espace mémoire, nombre de cœurs, charge, etc.)
  - message-retour :
    - une liste d'ordres comme : lancer une tâche, annuler une tâche, écriture définitive de la sortie sur le HDFS (**commit**), réinitialisation du tasktracker

framework Map-Reduce : initialisation d'un job

À la réception d'un job client, le Jobtracker :

- crée un objet Job avec un identifiant unique (**JobId**)
  - détermine les splits des données d'entrée
  - crée une liste de tâches (map et reduce) à distribuer avec un identifiant unique pour chaque tâche (**TaskId**)
    - les tâches map en fonction du nombre de splits
    - les tâches reduce en fonction d'un paramètre de configuration
- associe deux "fausses" tâches map :
  - tâche **setup** :
    - création d'un répertoire temporaire de travail sur le HDFS
    - doit être exécutée avant les "vraies" maps
  - tâche **cleanup** :
    - suppression du répertoire temporaire de travail sur le HDFS
    - doit être exécutée après l'ensemble des reduces
  - seront exécutées sur un tasktracker (le Jobtracker ne fait que ordonnancer :))
- insère cet objet dans une file d'attente interne
- sera mise à jour en fonction de l'état d'avancement

framework Map-Reduce : Assignation des tâches

A la réception d'un heartbeat, le Jobtracker :

- choisit les tâches de setup/cleanup en premier lieu
- sinon, choisit un job j dans la file d'attente
  - choisit parmi la liste des tâches de j, par ordre de préférence :
    - **les tâches de map** : en fonction de la localité du Tasktracker, choisit un map où son split associé est le plus proche possible de celui-ci
    - **les tâches de reduce** : à partir d'une certaine proportion de map terminés. Pas de prise en compte de la localité des données
- crée pour chaque tâche choisie une tentative de tâche avec un identifiant (**TaskAttemptId**)
- maintient pour chaque tâche la liste des tentatives et leur Tastracker associé

Le nombre de tentatives de tâche renvoyé ne peut pas excéder le nombre de slots max indiqué dans le heartbeat

framework Map-Reduce : exécution des tâches

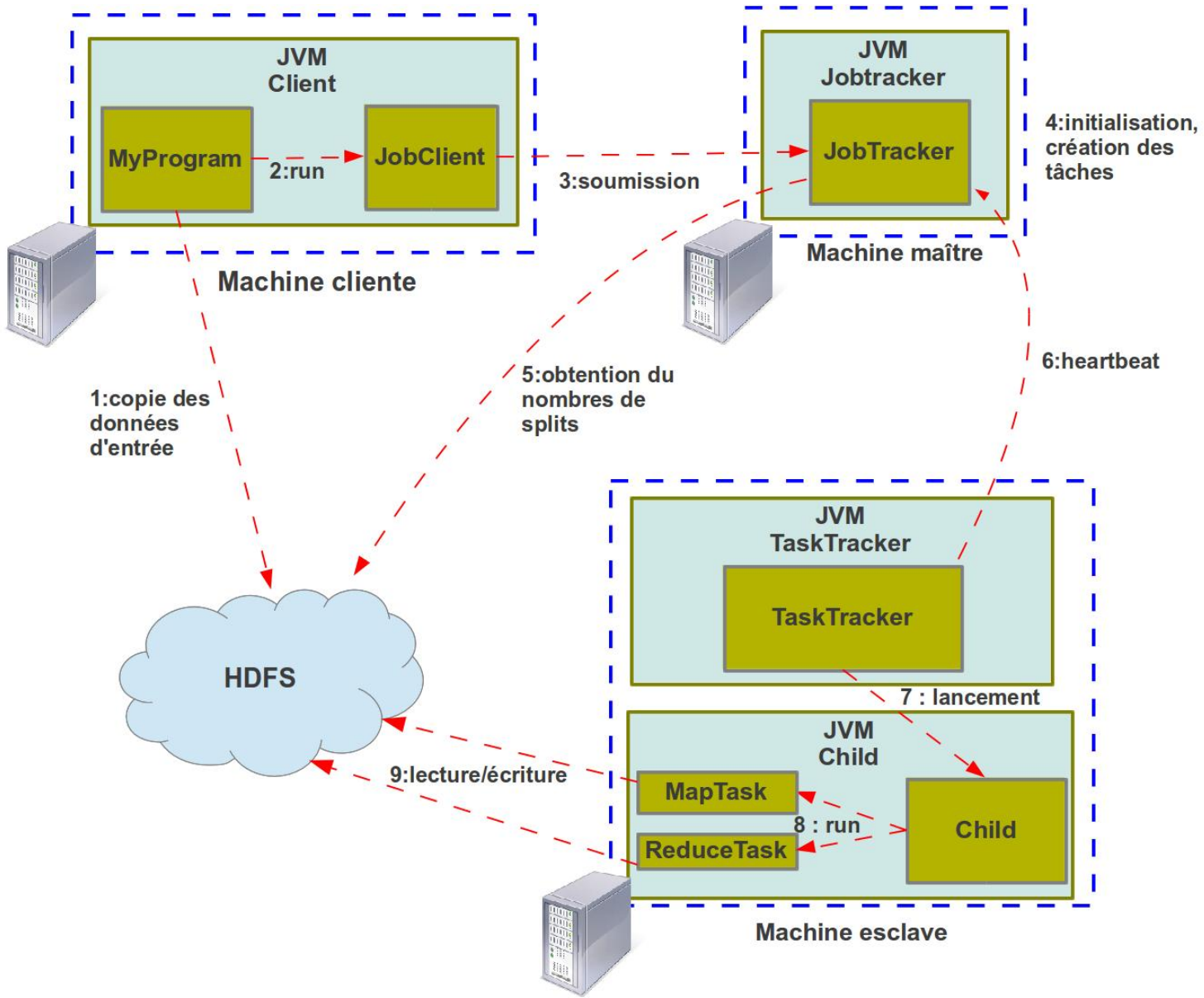
A la réception du retour de heartbeat, le tasktracker :

- crée une nouvelle JVM (**Child**) pour chaque tâche à exécuter.

Une fois un child lancé :

- reporte régulièrement à la JVM Tasktracker l'avancement et l'état de la tâche

framework Map-Reduce : schéma

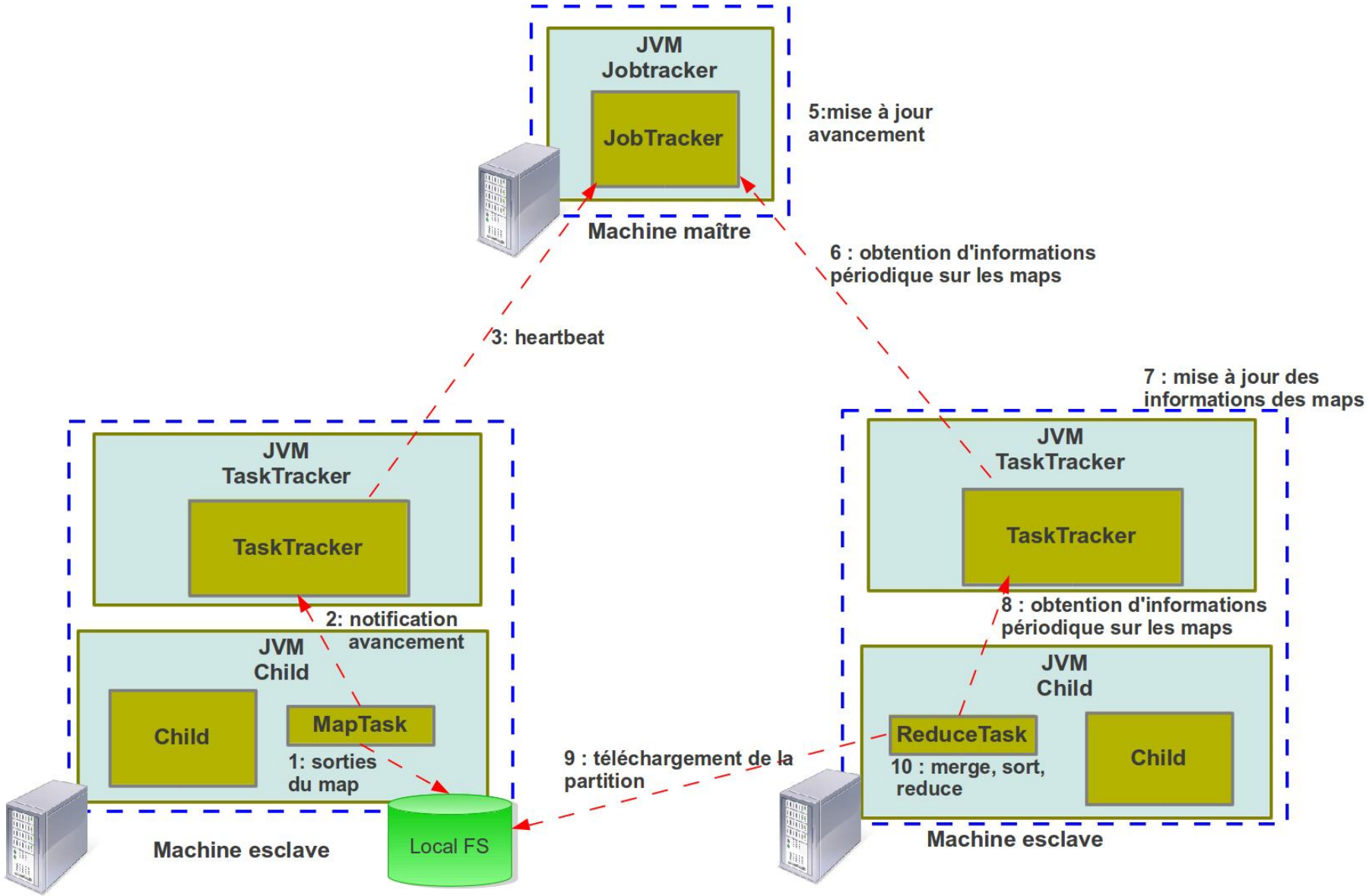


framework Map-Reduce : Shuffle

- les sorties de maps sont stockées sur les systèmes de fichiers locaux des machines
- le Jobtracker connaît l'avancement des tâches ainsi que leur localisation
- En plus du heartbeat, un thread de la JVM Tasktracker demande périodiquement au Jobtracker des informations sur les tâches de maps en cours
  - l'état d'avancement
  - la machine physique hébergeant la 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

Remarque : le Tasktracker ne doit pas effacer les sorties de maps avant que le job ne se termine (en cas de reduces défaillants)

framework Map-Reduce : schéma du shuffle



framework Map-Reduce : tolérance aux fautes

crash du Jobtracker :

- ça ne doit pas arriver :)

crash d'un Tasktracker :

- plus de heartbeat (détection par le Jobtracker)
- Le Jobtracker :
  - le marque comme défaillant et l'efface de son pool de Tasktrackers disponibles
  - réaffecte de nouvelles tentatives de tâches à d'autres Tasktrackers

crash d'un Child :

- report de l'erreur à la JVM Tasktracker, avant de se terminer
- propagation de l'erreur au Jobtracker via le heartbeat $\Rightarrow$ réaffectation d'une nouvelle tentative sur un autre Tasktracker
- si une tâche échoue plus de 4 fois, le job entier est annulé.
- le Jobtracker peut blacklister le Tastracker associé si son nombre de crashs de Child est supérieur au taux moyen de défaillance du cluster

framework Map-Reduce : exécution spéculative

Permet d'anticiper les défaillances sur des tâches jugées trop lentes :

- d'après Hadoop, une tâche lente est une tâche potentiellement défaillante
- détecté par le Jobtracker :
  - affectation d'une nouvelle tentative de tâche sur un autre tasktracker
- ce mécanisme peut être désactivé pour un job particulier

framework Map-Reduce : écriture des données finales

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

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

solution :

- chaque tentative écrit dans un fichier portant leur TaskAttemptId dans le répertoire temporaire de travail (créé par le **setup**)
- lorsqu'une tentative se termine, attendre la permission du jobtracker pour copier son résultat dans le répertoire définitif (état CommitPending)
- le Jobtracker donne une telle permission à la première tentative détectée CommitPending
- envoie de l'ordre de commit par le message retour du heartbeat
- les autres tentatives sont détruites après la fin du commit

Programmation Map Reduce : définir ses types de clé/valeur

- 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.

Programmation Map Reduce : format d'entrée et de sortie

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

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, Co
              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

- défini la politique de répartition des sorties de map
- classe héritant de la classe abstraite *Partitioner* $< KEY, VALUE >$
- 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 = new Job(conf, "MonJob");
   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);
   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 :

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

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

Exemples :

```
hadoop fs -ls <path>
hadoop fs -mv <src> <dst>
hadoop fs -cp <src> <dst>
hadoop fs -cat <src>
hadoop fs -copyFromLocal <localsrc> ... <dst>
hadoop fs -mkdir <path>
hadoop fs -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

Références

[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/