Fouilles de données et Medias sociaux

Master 2 DAC - FDMS

Sylvain Lamprier

UPMC

- Rich and big data:
  - Millions d'utilisateurs
  - Millions de contenus
  - Multimedia (texte, image, videos, etc...)
  - Millions de connections et relations (de differents types)
  - Preferences, tendences, opinions, ...
- Analyse réseau social ⇒ Traitements à large échelle
  - Taille des données
  - Complexité des données
  - Dynamicité
- Données + Traitements distribués

Traitement Distribué

- Hadoop Map Reduce
  - API Java
  - Echaînement de jobs
    - Exemples de tâches MapReduce itérées
    - Mise en oeuvre sur Hadoop
- Spark

Traitement distribué

- Apache Hadoop
  - Framework distribué
  - Utilisé par de très nombreuses entreprises
  - Traitements parallèles sur des clusters de machines
  - ⇒ Amener le code aux données
- Système de fichiers HDFS
  - Système de fichiers virtuel de Hadoop
  - Conçu pour stocker de très gros volumes de données sur un grand nombre de machines
  - Permet l'abstraction de l'architecture physique de stockage
  - Réplication des données

Map Reduce

- Exécution d'un problème de manière distribuée
  - ⇒ Découpage en sous-problèmes
  - ⇒ Execution des sous-problèmes sur les différentes machines du cluster
    - Stratégie algorithmique dite du *Divide and Conquer*
- Deux étapes principales :
  - Map: Emission de paires <clé,valeur> pour chaque donnée d'entrée lue
  - Reduce: Regroupement des valeurs de clé identique et application d'un traitement sur ces valeurs de clé commune

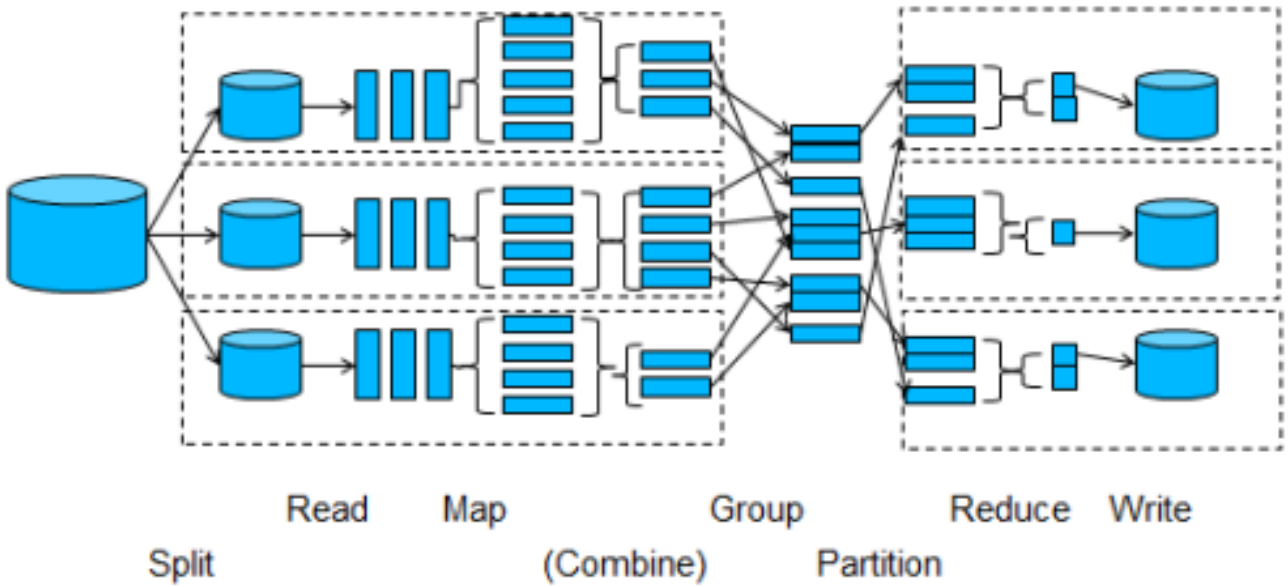
Map Reduce

- Composants d'un processus Map Reduce:
  1. Split: Divise les données d'entrée en flux parallèles à fournir aux noeuds de calcul.
  2. Read: Lit les flux de données en les découpant en unités à traiter. Par défaut à partir d'un fichier texte: unité = ligne.
  3. Map: Applique sur chaque unité envoyé par le Reader un traitement de transformation dont le résultat est stocké dans des paires <clé,valeur>.
  4. Combine: Composant facultatif qui applique un traitement de reduction anticipé à des fins d'optimisation. Cette étape ne doit pas perturber la logique de traitement.
  5. ...

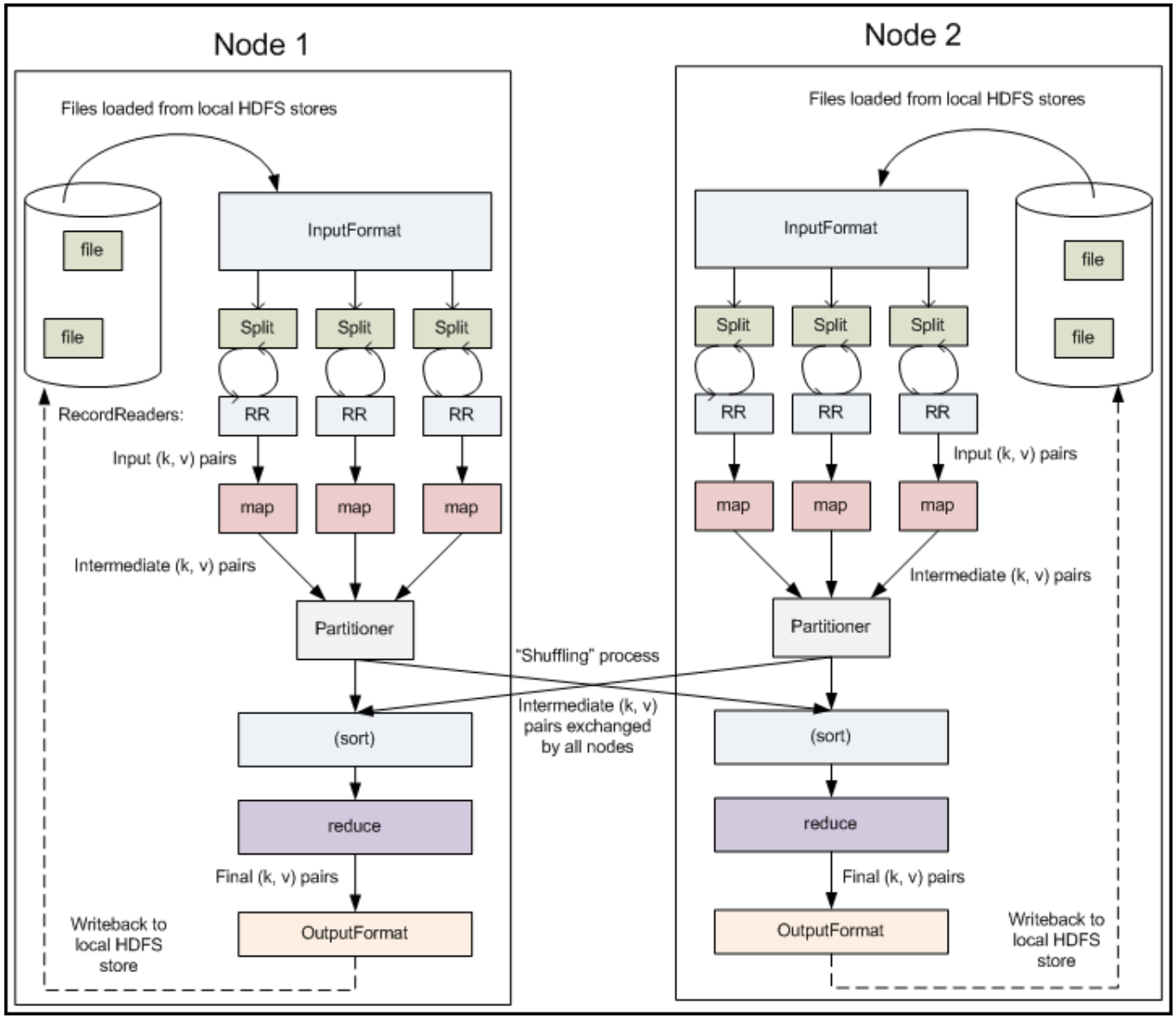
Map Reduce

- Composants d'un processus Map Reduce:
  1. ...
  2. Group: Regroupement (ou shuffle) des paires de clés communes. Réalisé par un tri distribué sur les différents noeuds du cluster.
  3. Partition: Distribue les groupes de paires sur les différents noeuds de calcul pour préparer l'opération de reduction. Généralement effectué par simple hashage et découpage en morceaux de données de tailles égales (dépend du nombre de noeuds Reduce).
  4. Reduce: Applique un traitement de réduction à chaque liste de valeurs regroupées.
  5. Write: Écrit le resultat du traitement dans le(s) fichier(s) résultats(s). On obtient autant de fichiers resultats que l'on a de noeuds de reduction.

Map Reduce



Map Reduce


Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
file
file
InputFormat
InputFormat
Split
Split
Split
Split
Split
Split
RecordReaders:
RR
RR
RR
RR
RR
RR
Input (k, v) pairs
Input (k, v) pairs
map
map
map
map
map
map
Intermediate (k, v) pairs
Intermediate (k, v) pairs
Partitioner
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
(sort)
reduce
reduce
Final (k, v) pairs
Final (k, v) pairs
OutputFormat
OutputFormat
Writeback to local HDFS store
Writeback to local HDFS store

Hadoop Map Reduce

- Hadoop : Framework Java
- Elements centraux:
  - Interface *Writable* : types de données
  - Class *InputFormat<K,V>* : format d'entrée
  - Class *Mapper<KI,VI,KO,VO>* : opération de Mapping
  - Class *Reducer<KI,VI,KO,VO>* : opération de Reduction
  - Class *OutputFormat<K,V>* : format de sortie

Map Reduce: Writable

- Interface Writable : types de données
  - Format d'échange des différents composants MapReduce
  - Hadoop nativement des classes d'écriture pour les types primitifs
    - int ⇒ IntWritable
    - long ⇒ LongWritable
    - float ⇒ FloatWritable
    - boolean ⇒ BooleanWritable
    - String ⇒ Text
    - ...
  - Pour les autres types : étendre soi-même Writable en redéfinissant
    - readFields(DataInput in) : définit la manière de lire la donnée dans un flux d'entrée in
    - write(DataOutput out) : définit la manière d'écrire la donnée dans un flux de sortie out

Hadoop Map Reduce: Writable

Exemple Writable : Point3D

```
public class Point3D implements Writable {
        public float x,y,z;

        public Point3D(float x, float y, float z) {
                this.x = x;
                this.y = y;
                this.z = z;
        }

        public void write(DataOutput out) throws IOException {
                out.writeFloat(x);
                out.writeFloat(y);
                out.writeFloat(z);
        }

        public void readFields(DataInput in) throws IOException {
                x = in.readFloat();
                y = in.readFloat();
                z = in.readFloat();
        }
}
```

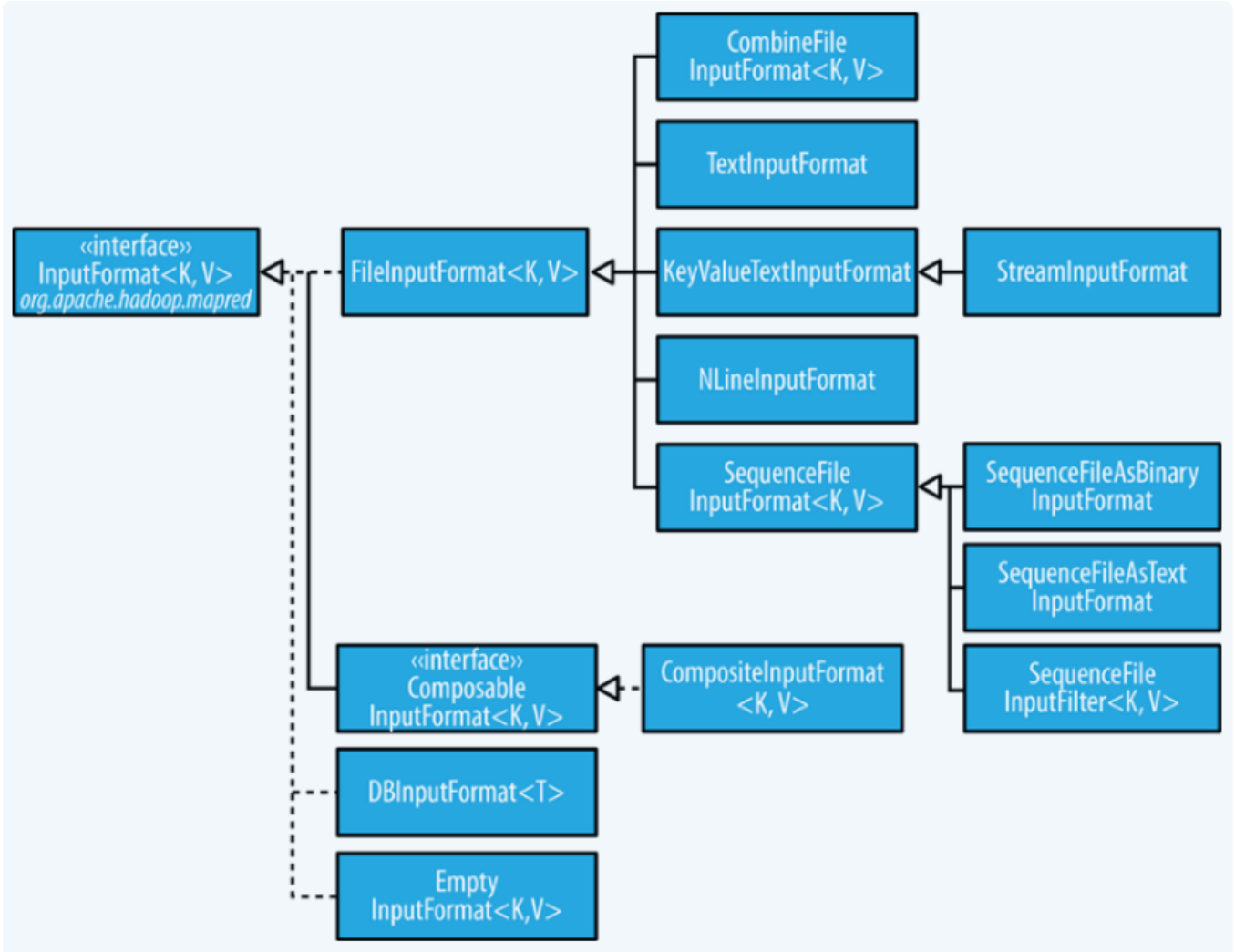
Hadoop Map Reduce: InputFormat

- Class *InputFormat<K,V>* : format d'entrée
  - Les données d'entrée sont découpées en unités de traitement
  - Définit :
    - Les fichiers à utiliser en entrée
    - La manière de découper les données d'entrée en blocs (définit le nombre de tâches map à executer)
    - La manière de découper les blocs d'entrée en unité de traitement (définit le nombre d'itérations séquentielles de chaque tâche Map)

Hadoop Map Reduce: InputFormat

- InputFormat standards :
  - TextInputFormat:
    - Données textuelles
    - Découpage par ligne
    - Clé: L'offset de la line
    - Valeur: La ligne textuelle
  - KeyValueInputFormat
    - Données textuelles
    - Découpage par ligne
    - Clé: Tout le texte qui précède la 1ière tabulation
    - Valeur: Le reste de la line
  - SequenceFileInputFormat
    - Données binaires produites par Hadoop (SequenceFileOutputFormat)
    - Découpage par couples définis dans le format
    - Clé - Valeur: objets Writable sérialisés
- Les trois formats découpent les entrées en blocs de 64Mo
  - Correspond à la taille des blocs sur HDFS
  - Modifiable selon les paramètres *mapred.min.split.size* et *mapred.max.split.size*

Hadoop Map Reduce: InputFormat



Hadoop Map Reduce: InputFormat

- Définition de FileInputFormat Personnalisés

⇒ Hériter de FileInputFormat<K,V>

⇒ Une méthode à redéfinir:

- createRecordReader(InputSplit,TaskAttemptContext) : crée le lecteur d'entrée

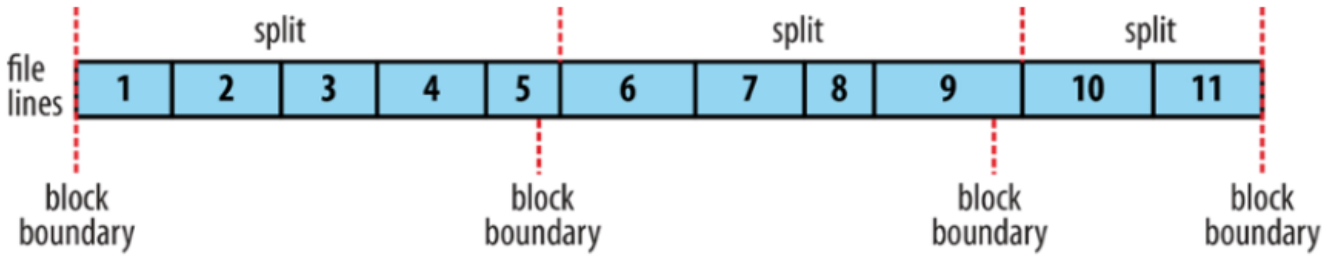
⇒ Si on ne souhaite pas que le fichier soit découpé en plusieurs blocs :

- Redéfinir isSplitable(JobContext context, Path file) pour qu'elle retourne *false*

Hadoop Map Reduce: RecordReader

- Définit la manière dont on lit les entrées
  - Un InputSplit est un bloc de traitement...
  - ... que l'on doit lire par unités
  - Problème : Les frontières des unités ne coincident pas toujours avec les frontières des blocs
    (ex: unité = ligne v.s. bloc=64Mo)

  ⇒ Le Reader termine la lecture de l'unité sur le bloc suivant. Attention à bien respecter cette logique lors de la définition d'un Reader personnalisé.



Hadoop Map Reduce: Mapper

- Pour chaque unité de traitement
  - Emission d'une paire clé-valeur

⇒ Hériter de Mapper<Kin,Vin,Kout,Vout>

Exemple de Mapper: WordCount

```
public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable>
{
        private static final IntWritable ONE=new IntWritable(1);

        protected void map(Object offset, Text value, Context context)
        throws IOException, InterruptedException
        {
                StringTokenizer tok=new StringTokenizer(value.toString(), " ");
                while(tok.hasMoreTokens())
                {
                        Text word=new Text(tok.nextToken());
                        context.write(word, ONE);
                }
        }
}
```

Hadoop Map Reduce: Reducer

- Pour chaque clé émise par le Mapper
  - Réduction de la liste de valeurs associées
  - Emission d'un résultat Clé-Valeur

⇒ Hériter de Reducer<Kin,Vin,Kout,Vout>

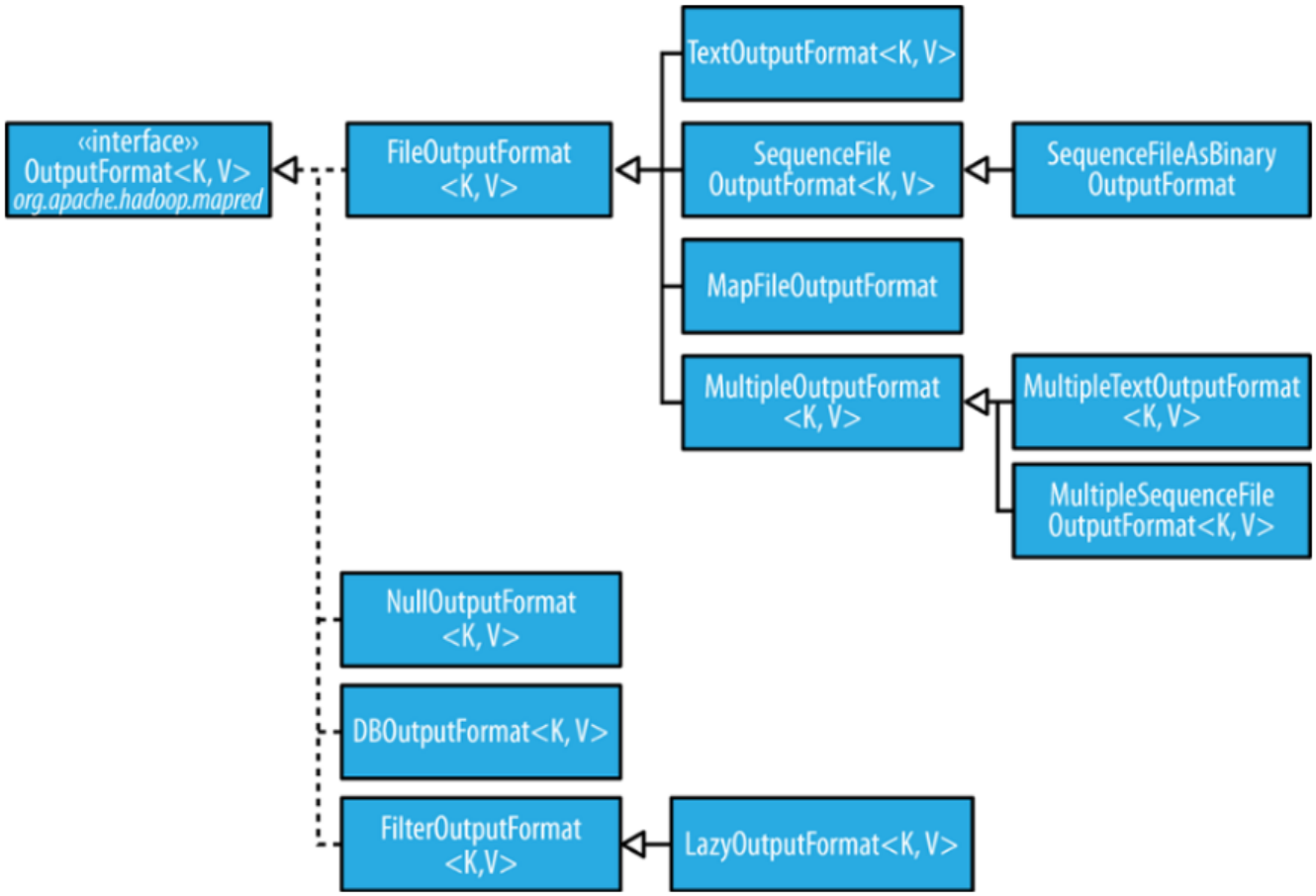
Exemple de Reducer: WordCount

```
public class WordCountReducer extends Reducer<Text, IntWritable, Text, Text>
{
        public void reduce(Text key, Iterable<IntWritable> values,Context context)
        throws IOException, InterruptedException
        {
                Iterator<IntWritable> i=values.iterator();
                int count=0;
                while(i.hasNext())
                        count+=i.next().get();
                context.write(key, new Text(count+" occurences."));
        }
}
```

Hadoop Map Reduce: OutputFormat

- Class *OutputFormat<K,V>* : format de sortie
  - Les résultats du MapReduce sont écrites dans la sortie selon
    - Le support de sortie
    - Le format de sortie (texte, binaire, etc..)
    - Le type de compression

Hadoop Map Reduce: OutputFormat



Hadoop Map Reduce: OutputFormat

- Correspondance entre InputFormat et OuputFormat:
  - TextOutputFormat ⇒ KeyValueInputFormat
  - SequenceOutputFormat ⇒ SequenceInputFormat
- SequenceOutputFormat<K,V>
  - Format de sortie du Mapper
  - Permet de serialiser divers types d'objects que l'on pourra deserialiser par un SequenceInputFormat
  - Différents types de compression permis (record ou block compression)
  - Produit des fichiers facilement "splittable" ⇒ améliore les performances d'un traitement par un futur nouveau job MapReduce

Exemple de Création d'un Job

```
public static Job createJob(String in, String out) throws IOException {
  Configuration conf=new Configuration();
  Job job=Job.getInstance(new Cluster(conf),conf);
  job.setJarByClass(Test.class);
  SequenceFileInputFormat.addInputPath(job,new Path(in));
  job.setInputFormatClass(SequenceFileInputFormat.class);
  job.setMapperClass(TestMapper.class);
  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(Point3D.class);
  job.setCombinerClass(TestReducer.class);
  job.setReducerClass(TestReducer.class);
  job.setNumReduceTasks(3);
  SequenceFileOutputFormat.setOutputPath(job,new Path(out));
  job.setOutputFormatClass(SequenceFileOutputFormat.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(Point3D.class);
  return job;
}
```

Exécution d'un Job: Création du Driver

```
public class Driver {
        public static void main(String[] args) throws   Exception
        {
                Job job=createJob(args[0],args[1]);
                job.setJarByClass(Driver.class);
                job.waitForCompletion(true);
        }
}
```

Hadoop Map Reduce: Echaînement de Jobs

- Job
  - submit() : Soumet le job
  - waitForCompletion(boolean verbose) : Soumet le job et attend sa terminaison
- ControlledJob
  - addDependingJob(ControlledJob dependingJob) : ajoute une dépendence au job (attend la terminaison du job passé en paramètre pour commencer)
  - submit() : Soumet le job
- JobControl
  - addJob(ControlledJob aJob) : ajoute un nouveau job au JobControl
  - run() : lance les jobs
  - allFinished() : retourne un booléen indiquant si tous les jobs lancés dans ce JobControl sont terminés

Pour avoir un retour sur l'execution

```
while (!jobControl.allFinished()) {
  System.out.println("Jobs in waiting state: " + jobControl.getWaitingJobList().size());
  System.out.println("Jobs in ready state: " + jobControl.getReadyJobsList().size());
  System.out.println("Jobs in running state: " + jobControl.getRunningJobList().size());
  List<ControlledJob> successfulJobList = jobControl.getSuccessfulJobList();
  System.out.println("Jobs in success state: " + successfulJobList.size());
  List<ControlledJob> failedJobList = jobControl.getFailedJobList();
  System.out.println("Jobs in failed state: " + failedJobList.size());
}
```

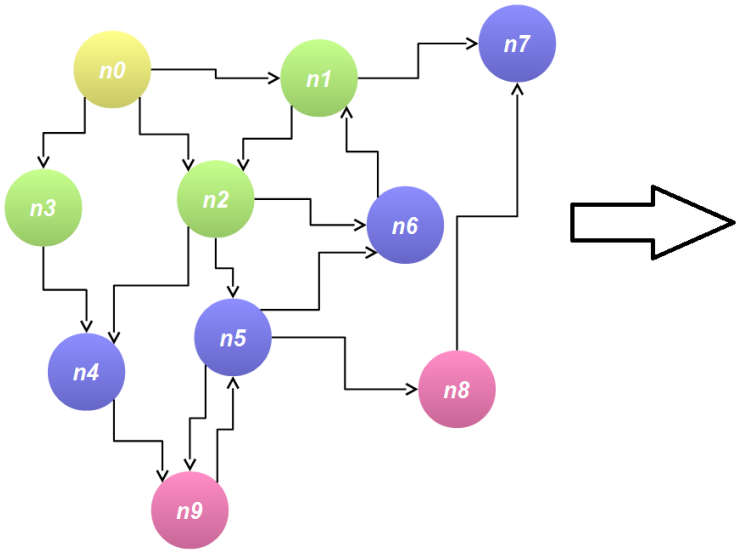
Hadoop Map Reduce: Echaînement de Jobs

- De nombreuses applications peuvent nécessiter l'enchaînement de plusieurs jobs
  - ⇒ Différents Map-Reduce à enchaîner
  - ⇒ Multiples itérations d'un même job Map-Reduce
- Comment exploite-t-on les resultats ?
  - Resultats en tant que données en RAM ⇒ Entrées fixes
  - Resultats en tant que nouvelles entrées du MapReduce ⇒ Ecriture/Lecture des resultats

4 Exemples de problèmes MapReduce incrémentaux

- Calcul des plus courtes distances à un noeud d'un graphe
- Calcul des plus courts chemins entre toutes les paires de noeuds d'un graphe
- Algorithme PageRank
- Regression Logistique

Calcul des plus courtes distances à un noeud d'un graphe



Distances à n0 :

n1 = 1
n2 = 1
n3 = 1
n4 = 2
n5 = 2
n6 = 2
n7 = 2
n8 = 3
n9 = 3

- Quelles Entrées ?
- Quelles Transformations Map ?
- Quelles Clés de Regroupement ?
- Quelles opérations de réduction ?

Calcul des plus courtes distances à un noeud d'un graphe

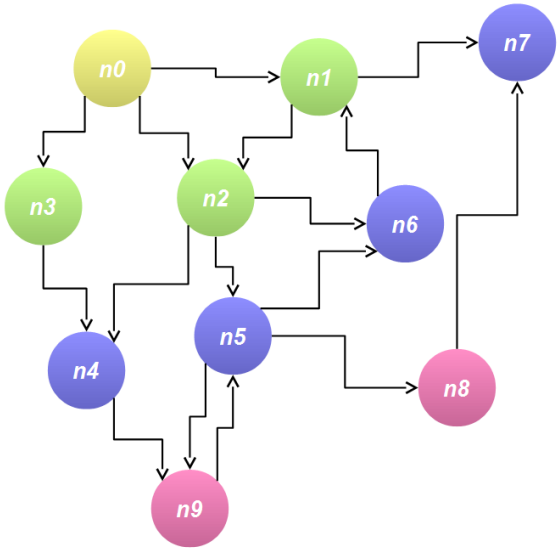
- Données :
  - Structure de Noeud n:
    - $n.id$: identifiant du noeud n
    - $n.adjacencyList$: Liste d'adjacence $S(n)$ du noeud n
    - $n.dist$: Distance du noeud n au noeud de départ *start* (initialisée à ∞ pour tous les noeuds sauf *start*)
- Formulation (BFS):
  - $start.dist = 0$
  - Pour tout noeud n atteignable par un ensemble de noeuds M, $n.dist = 1 + \min_{m \in M} m.dist$

Calcul des plus courtes distances à un noeud d'un graphe

Breadth First Search: Job MapReduce

```
Map(n){
        Emettre(n.id,n);
        Si(n.dist < ∞):
                Pour tout id ∈ n.adjacencyList:
                        dist ← n.dist + 1
                        Emettre(id, Node(id, m.dist, ∅));
}

Reduce(nid, nodes){
        min ← ∞;
        nmin ← ∅;
        adj ← ∅;
        Pour tout n ∈ nodes:
                Si(n.adjacencyList ≠ ∅): adj ← n.adjacencyList
                Si((nmin == ∅) ou (n.dist < min)):
                        min ← n.dist;
                        nmin ← n
        nmin.adjacencyList ← adj;
        Emettre(nmin);
}
```

Calcul des plus courts chemins entre toutes les paires de noeuds d'un graphe



Plus courts chemins :

n0,n1: n0n1
n0,n2: n0n2
n0,n3: n0n3
n0,n4: n0n3n4; n0n2n4
n0,n5: n0n2n5
n0,n6: n0n2n6
n0,n7: n0n1n7
n0,n8: n0n2n5n8
n0,n9: n0n3n4n9; n0n2n4n9; n0n2n5n9
n1,n0:
n1,n2: n1n2
n1,n3:
n1,n4: n1n2n4
...

- Quelles Entrées ?
- Quelles Transformations Map ?
- Quelles Clés de Regroupement ?
- Quelles opérations de réduction ?

Calcul des plus courts chemins entre toutes les paires de noeuds d'un graphe

- Données :
  - Chemins C entre paire de noeuds:
    - $C.start$: noeud de depart
    - $C.end$: noeud de fin
    - $C.dist$: distance (nbre de liens à suivre pour atteindre $C.end$ à partir de $C.start$
    - $C.shortests$: Ensemble de chemins de $C.start$ à $C.end$ de distance $C.dist$
- Formulation :
  - Pour tout couple de noeuds u, v tel qu'il existe un lien entre u et v, la distance minimale $D(u,v)$ est : $D(u,v) = 1$
  - Pour tout couple de noeuds u, v non directement lié : $D(u,v) = \min_m(D(u,m) + D(m,v))$

Calcul des plus courts chemins entre toutes les paires de noeuds d'un graphe

Calcul des plus courts chemins: Job MapReduce

```
Map(p){
        Emettre(p.start,p);
        Si(p.new): Emettre(p.end,p);
}

Reduce(node, ListPaths){
        P ← ∅; Ends ← ∅; Starts ← ∅
        Pour tout p ∈ ListPaths:
            p.new ← False;
            Si(p.start==node): Ends = Ends ∪ p;
            Sinon: Starts = Starts ∪ p;
        Pour tout ep ∈ Ends:
            P[ep.start, ep.end] = ep;
            Pour tout sp ∈ Starts:
                Si(sp.start==ep.start): continue;
                d ← sp.dist + ep.dist;
                p ← P[sp.start, ep.end];
                Si((p==null) ou (p.dist>=d)):
                    newp ← Fusion(sp, ep);
                    Si(p.dist==d):
                        p.shortests ← p.shortests ∪ newp.shortests;
                    Sinon:
                        P[sp.start, ep.end] ← newp;
        Pour tout p ∈ P: Emettre(p);
}
```

Algorithme PageRank

- *PageRank* simule le comportement d'un surfeur aléatoire:
  1. au temps t, le sufeur est sur la page p_t,
  2. avec une probabilité d, il clique aléatoirement sur un lien de p_t. p_{t+1} est alors la page pointée par ce lien.
  3. avec une probabilité $1-d$, p_{t+1} est une page Web choisie aléatoirement.
  4. retour à l'étape 1.
- Avec
  - P_i : l'ensemble des pages pointant vers la page i
  - ℓ_j : le nombre de liens sortant de la page j
- Probabilité μ_i^t que le surfeur soit sur la page i au temps t =

$$\mu^{t+1} = \frac{1-d}{N}\mathbf{1} + dA\mu^t \quad \text{avec} \quad A_{ij} = \begin{cases} \frac{1}{\ell_j} \text{ si } j \in P_i \\ 0 \text{ sinon} \end{cases} \quad \text{et} \quad N : \#\text{pages Web.}$$

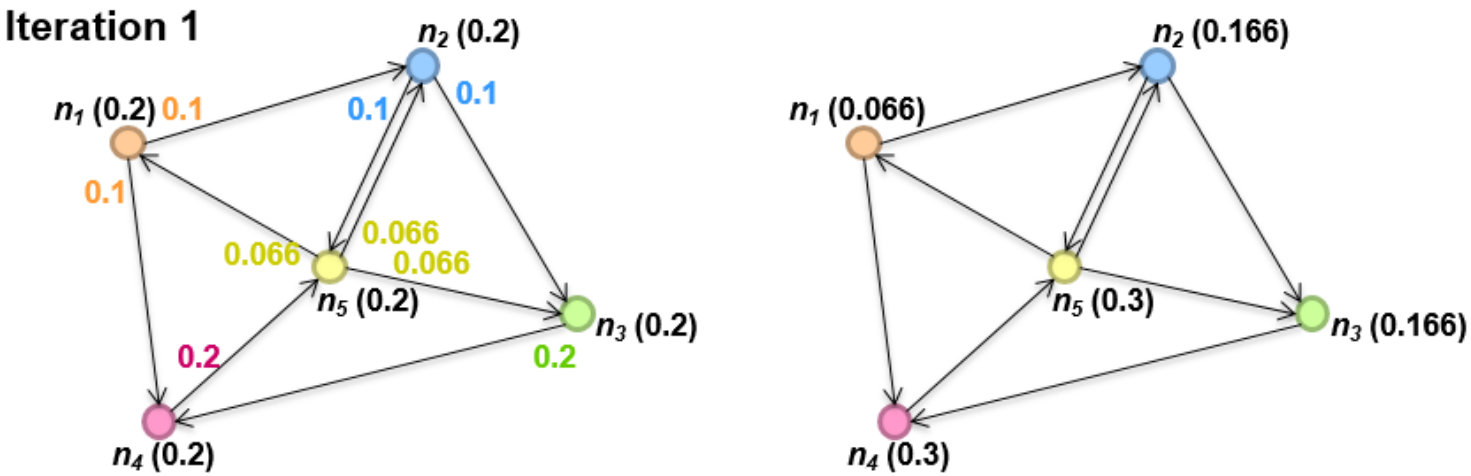
Algorithme PageRank

$$\mu^{t+1} = \frac{1-d}{N}\mathbf{1} + dA\mu^t \quad \text{avec} \quad A_{ij} = \begin{cases} \frac{1}{\ell_j} \text{ si } j \in P_i \\ 0 \text{ sinon} \end{cases} \quad \text{et} \quad N : \#\text{pages Web.}$$

- Quelles Entrées ?
- Quelles Transformations Map ?
- Quelles Clés de Regroupement ?
- Quelles opérations de réduction ?

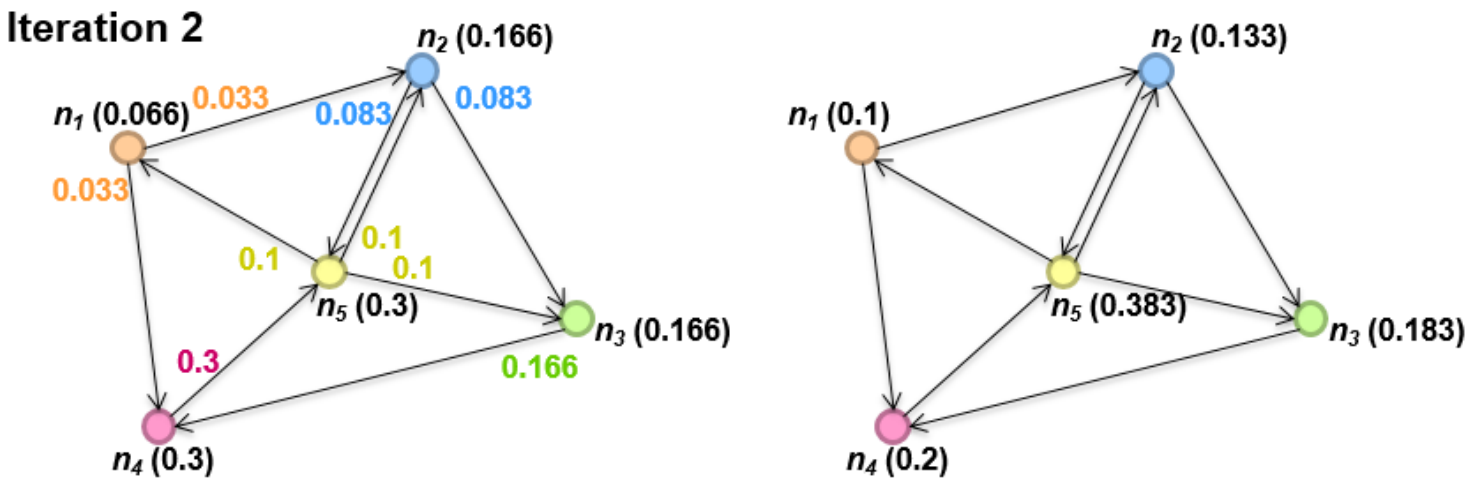
Algorithme PageRank

Avec d=1 :



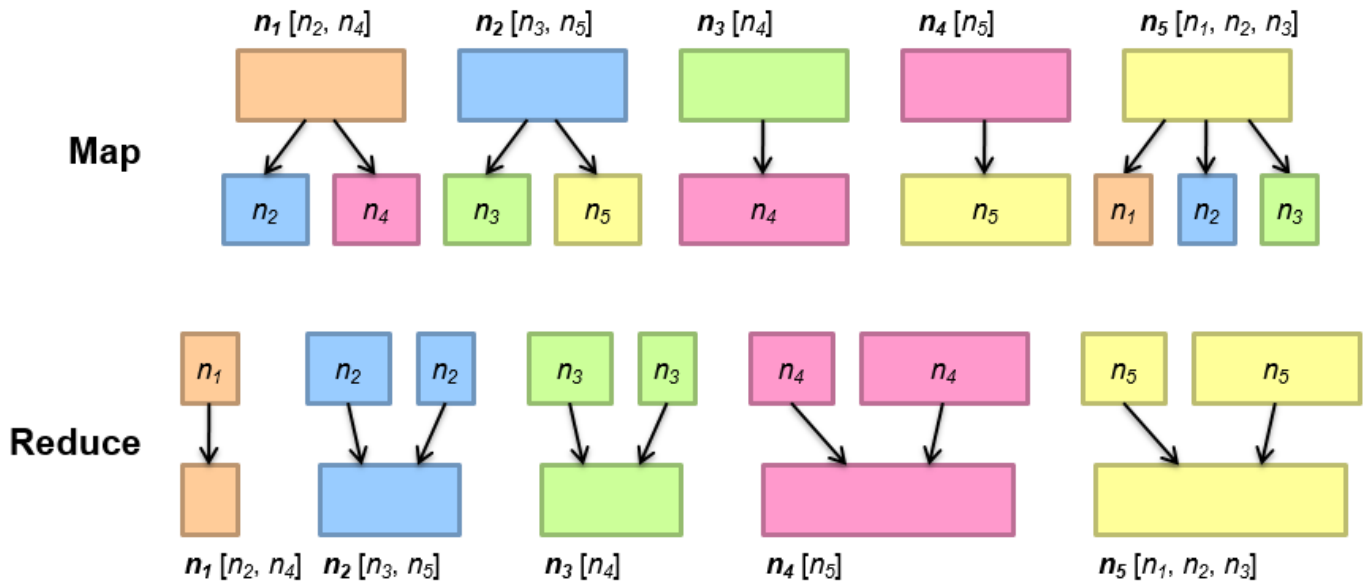
Algorithme PageRank

Avec d=1 :



Algorithme PageRank

Avec d=1 :



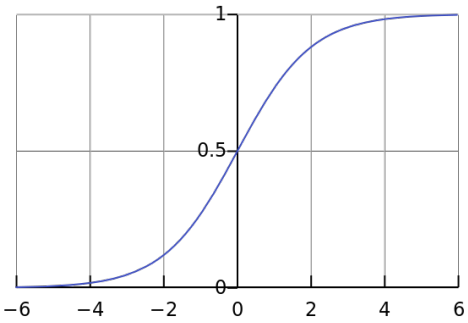
Algorithme PageRank

- Données :
  - Structure de Noeud n:
    - $n.id$: identifiant du noeud n
    - $n.adjacencyList$: Liste d'adjacence $S(n)$ du noeud n
    - $n.p$: score PageRank à l'iteration courante
- Formulation :
  - Pour tout noeud n: $n.p^{t_0} = \frac{1}{N}$
  - Pour tout noeud n atteignable par un ensemble de noeuds M, $n.p^{t+1} = \frac{1-d}{N} + d \sum_{m \in M} \frac{m.p^t}{|m.adjacencyList|}$

PageRank: Job MapReduce

```
Map(n){
        np ← n.p / |m.adjacencyList|
        n.p ← 0
        Emettre(n.id, n);
        Pour tout id ∈ n.adjacencyList :
                Emettre(id, Node(id, ∅, np));
}

Reduce(nid, nodes){
        sum ← 0;
        adj ← ∅;
        Pour tout n ∈ nodes :
                Si(n.adjacencyList ≠ ∅) : adj ← n.adjacencyList;
                sum ← sum + n.p;
        Emettre(Node(nid, adj, (1 − d)/N + d × sum));
}
```

Regression Logistique



- Hypothèse :
  - Rapport de probas conditionnelles peut être modélisé par une application linéaire
  - $\ln \frac{p(1|x)}{1-p(1|x)} = \theta.x$

  $\Rightarrow p(1|X) = \frac{e^{\theta.x}}{1+e^{\theta.x}} = \frac{1}{1+e^{-\theta.x}}$

- Maximum de vraissemblance
  - $\theta^* = \arg\max_{\theta} \sum_i^n y_i ln(\frac{1}{1+e^{-\theta.x}}) + (1-y_i) ln(1-\frac{1}{1+e^{-\theta.x}})$
  - Gradient : $\sum_i^n x_i (y_i - \frac{1}{1+e^{-\theta.x}})$

Regression Logistique

Estimation du maximum de vraissemblance par montée de gradient :

$$\theta^{t+1} = \theta^t + \alpha \sum_i^n x_i (y_i - \frac{1}{1 + e^{-\theta . x}})$$

- Quelles Entrées ?
- Quelles Transformations Map ?
- Quelles Clés de Regroupement ?
- Quelles opérations de réduction ?

Regression Logistique

- Données :
  - Item =
    - *item.x* : Vecteur de features de *nbDims* dimensions
    - *item.y* : Score à prédire
  - Entrées fixes, seul θ varie
- Formulation:
  - $\theta^{t+1} = \theta^t + \alpha \sum_i^n x_i (y_i - \frac{1}{1+e^{-\theta.x}})$

Montée de gradient: Job MapReduce

```
Map(item){
        Pour i de 1 à nbDims:
                Emettre(i, item.x_i(item.y − 1/(1+e^{−<θ,item.x>})));
}

Reduce(i, gradients){
        sumg ← 0;
        Pour tout g ∈ gradients:
                sumg ← sumg + g
        Emettre(i, theta_i + α × sumg);
}
```

Hadoop Map Reduce: Echaînement de Jobs

- Resultats en tant que nouvelles entrées du MapReduce ⇒ Ecriture/Lecture des resultats dans des SequenceFile
  - Ecriture/Lecture des resultats dans des fichier
  - De préférence SequenceFile : job.setOuputFormat(SequenceFileOutputFormat.class)
  - A l'iteration suivante, lecture à partir d'un SequenceFile : job.setInputFormat(SequenceFileInputFormat.class)
- Resultats en tant que données en RAM ⇒ Entrées fixes
  - Entrées fixes
  - Paramètres lus à partir de fichiers resultats
  - Eventuellement spécifier une sorte vide : job.setOuputFormat(NullOutputFormat.class)

Hadoop Map Reduce vs Spark

- Hadoop Map Reduce: Inconvénients
  - Beaucoup de lectures/ecritures sur disque
  - Pas de mise en mémoire vive
- Spark
  - Mise en oeuvre d'une mémoire partagée
  - Simplification des process

  ⇒ Jusqu'à 10x fois plus rapide sur disque

  ⇒ Jusqu'à 100x fois plus rapide sur disque

  ⇒ Code plus compact

  ⇒ Mieux adapté pour l'apprentissage statistique

- Contexte Spark
  - Définit les opérations Spark à effectuer sur les collections de données

Spark: Initialisation

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf sparkConf = new SparkConf().setAppName(name);
JavaSparkContext sc = new JavaSparkContext(sparkConf);
```

Spark

- Resilient Distributed Datasets (RDD)
  - Collections d'éléments manipulables en parallèle
  - Deux manières de créer des RDDs:
    - Parallelisation d'une collection en mémoire

Spark: Collection parallelisée

```
import org.apache.spark.api.java.JavaRDD;
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

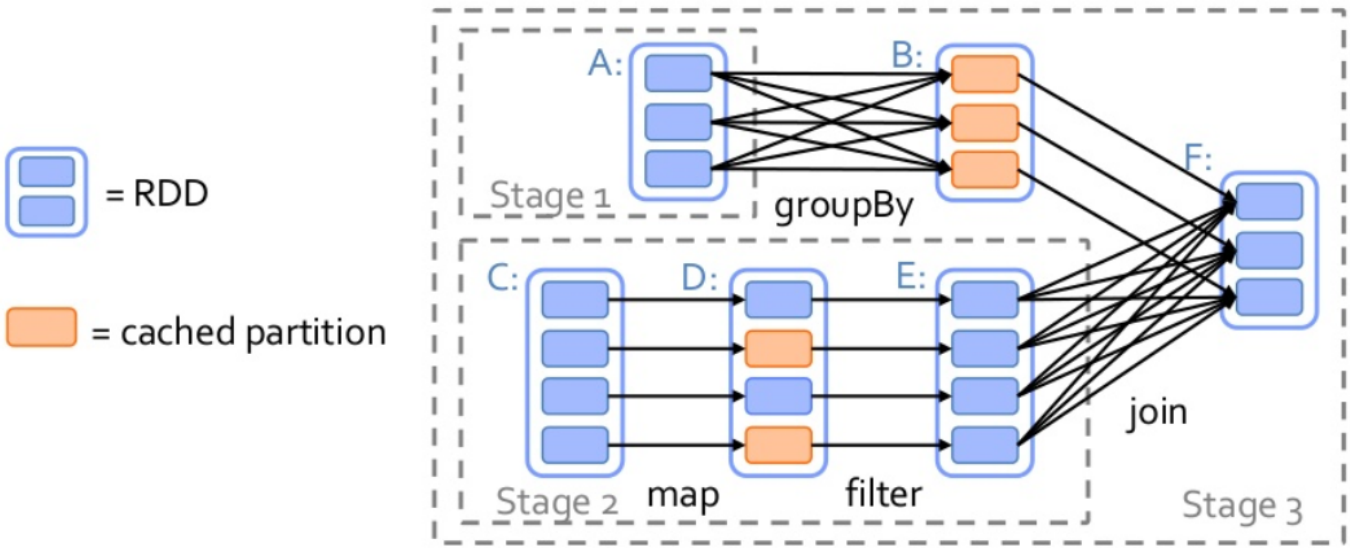
    - Référencement d'un dataset sur un système de fichier distribué comme HDFS

Spark: Initialisation

```
import org.apache.spark.api.java.JavaRDD;

// Fichier(s) texte à lire ligne par ligne
JavaRDD<String> distFile = sc.textFile("data");

// Fichiers texte entiers
JavaRDD<String> distFile = sc.wholeTextFiles("data");

// Fichier(s) SequenceFile avec K et V des types de données étendant Writable
JavaRDD<String> distFile = sc.sequenceFile[K, V]("data");

// Autres types de fichiers avec Reader Hadoop
JavaRDD<String> distFile = sc.newHadoopRDD("data");
```

- Deux types d'operations sur les RDD
- Transformations
  - Création de nouveaux RDD par application d'une transformation
    - map, flatMap, filter, reduceByKey, join, etc...
  - "Lazy" operations => effectué uniquement lorsqu'une action est requise sur le resultat de la transformation
    - Moins de résultats à retourner
    - Plannification des tâches facilitée
- Actions
  - Opération produisant un résultat
    - reduce, takeSample, count(), etc...
    - saveAsTextFile, saveAsSequenceFile, etc...

- Persistence des RDD
  - Si besoin de réutiliser le résultat d'une transformation plusieurs fois
  - *rdd.persist(StorageLevel.level)*: stocke le RDD pour utilisation ultérieure
- Différents types de stockage:
  - MEMORY_ONLY : Stocke les RDD en tant qu'objets Java déserialisés en mémoire vive. Si une partie ne tient pas en mémoire, elle sera re-calculée lorsque l'on en aura besoin
  - MEMORY_AND_DISK : Stocke les RDD en tant qu'objets Java déserialisés en mémoire vive, mais si une partie ne tient pas en mémoire, on l'écrit sur disque
  - DISK_ONLY : Stocke les RDD sur disque
  - MEMORY_ONLY_SER et MEMORY_AND_DISK_SER : Identique à MEMORY_ONLY et MEMORY_AND_DISK mais stocke les objets après sérialisation. Plus léger en mémoire mais plus lent à reconstruire.


= RDD
= cached partition
A:
B:
F:
Stage 1
groupBy
C:
D:
E:
join
Stage 2
map
filter
Stage 3

- Fonctions
  - Définition de fonction à passer aux opérations
    - Function : 1 argument en entrée, chaque valeur du dataset (map)
    - Function2 : 2 arguments en entrée, le resultat courant + la nouvelle valeur (reduction)
    - FlatMapFunction : 1 argument en entrée, une liste de valeur en sortie (flatMap)

Spark: Exemple

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new Function<String, Integer>() {
  public Integer call(String s) { return s.length(); }
});
int totalLength = lineLengths.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer a, Integer b) { return a + b; }
});
```

- Certaines opérations travaillent avec des paires Clé-Valeur
  - JavaPairRDD, PairFunction, Tuple2

Spark: Exemple

```
JavaRDD<String> lines = jsc.textFile("JavaHdfsLR.java");
JavaRDD<String> words = lines.flatMap(
                  new FlatMapFunction<String, String>() {
                    public Iterable<String> call(String s) {
                      return Arrays.asList(s.split(" "));
                    }
                  }
);
JavaPairRDD<String, Integer> ones = words.mapToPair(
                                  new PairFunction<String, String, Integer>() {
                                    public Tuple2<String, Integer> call(String s) {
                                      return new Tuple2<String,Integer>(s, 1);
                                    }
                                  }
);
JavaPairRDD<String, Integer> counts = ones.reduceByKey(
                                  new Function2<Integer, Integer, Integer>() {
                                    public Integer call(Integer i1, Integer i2) {
                                      return i1 + i2;
                                    }
                                  }
);
```

- Variables partagées
  - Mise en place de variables auxquelles les différents process en parallèle peuvent avoir accès

  ⇒ Très utile pour des tâches d'apprentissage statistique avec optimisation itérative de paramètres

  - Deux types de variables partagées
    - Variables Broadcast: variable en lecture seule, partagée par tous les process
    - Accumulateurs: variable en écriture seule, à laquelle on ne peut qu'ajouter des éléments (seul le Driver peut lire le contenu de la variable)

- Deux types de variables partagées
  - Variables Broadcast

Spark: Variables Broadcast

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});
broadcastVar.value();
```

  - Accumulateurs

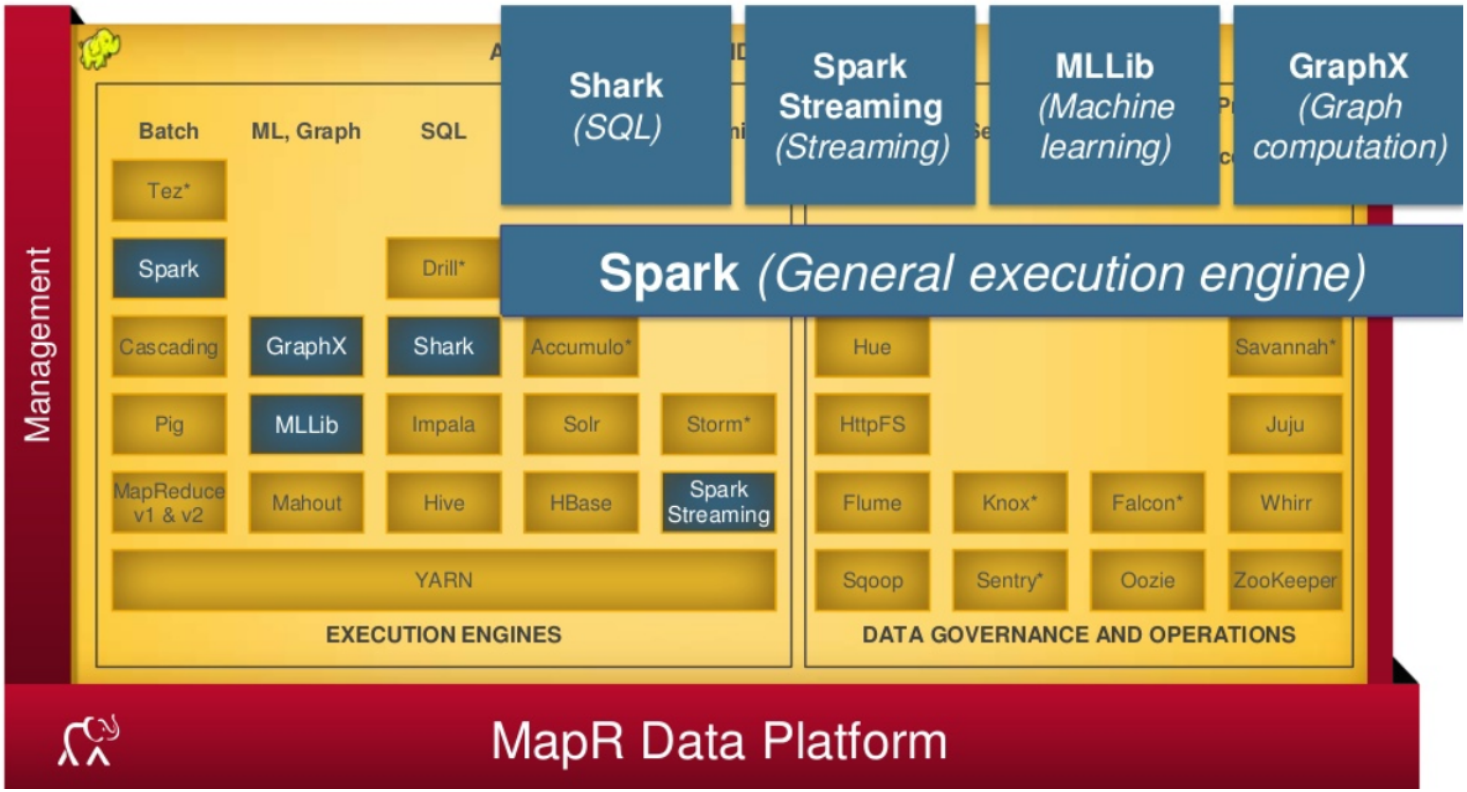
Spark: Accumulateurs

```
class VectorAccumulatorParam implements AccumulatorParam<Vector> {
  public Vector zero(Vector initialValue) {
    return initialValue;
  }
  public Vector addInPlace(Vector v1, Vector v2) {
    v1.addInPlace(v2); return v1;
  }
}
Vector initialValue=new Vector(...);
Accumulator<Vector> vecAccum = sc.accumulator(initialValue, new VectorAccumulatorParam());
// De n'importe quel process :
vacAccum.add(new Vector(...));
// Du driver :
vacAccum.value();
```

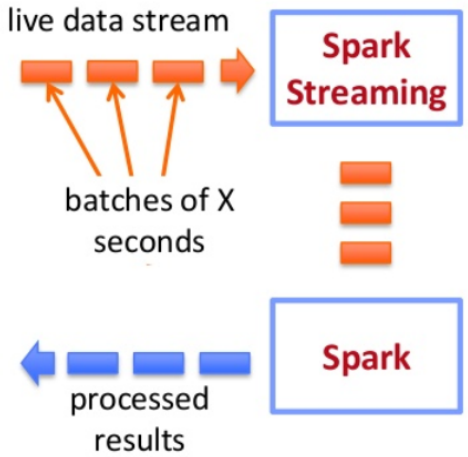
Exemple Spark: Regression Logistique

```
class ComputeGradient extends Function<DataPoint, Vector> {
  private Vector w;
  ComputeGradient(Vector w) { this.w = w; }
  public Vector call(DataPoint p) {
    return p.x.times(p.y - (1 / (1 + Math.exp(-w.dot(p.x)))));
  }
}

JavaRDD<DataPoint> points = spark.textFile(...).map(new ParsePoint()).cache();
Vector w = Vector.random(D); // current separating plane
for (int i = 0; i < ITERATIONS; i++) {
  Vector gradient = points.map(new ComputeGradient(w)).reduce(new AddVectors());
  w = w.subtract(gradient);
}
System.out.println("Final separating plane: " + w);
```

Spark


Shark
(SQL)
Spark
Streaming
(Streaming)
MLLib
(Machine
learning)
GraphX
(Graph
computation)
Spark (General execution engine)
Management
Batch
ML, Graph
SQL
Tez*
Spark
Drill*
Cascading
GraphX
Shark
Accumulo*
Pig
MLLib
Impala
Solr
Storm*
MapReduce
v1 & v2
Mahout
Hive
HBase
Spark
Streaming
YARN
EXECUTION ENGINES
Hue
Savannah*
HttpFS
Juju
Flume
Knox*
Falcon*
Whirr
Sqoop
Sentry*
Oozie
ZooKeeper
DATA GOVERNANCE AND OPERATIONS
MapR Data Platform

Spark Streaming




```
tagCounts = hashTags.window(Minutes(1), Seconds(5)).count()
```

sliding window operation | window length | sliding interval