

## Calculs sur des graphes en M/R et Spark

Camelia Constantin – LIP6

Prénom.Nom@lip6.fr

## Spark : aspect avancés

### Variables Broadcast

Les opérations sur les RDD prennent comme arguments des fonctions (fermetures)

→ les fermetures et les variables qu'elles utilisent sont représentées par des objets Java qui sont sérialisés et envoyés avec les tâches aux workers

Dans certains cas, des variables de grande taille doivent être accessibles et partagées entre plusieurs tasks ou entre plusieurs opérations

#### => Variables broadcast

- on garde une copie d'une variable en lecture seule sur chaque machine
- on n'envoie pas une copie de la variable avec chaque tâche
- algorithmes efficaces de distribution des variables broadcast afin de réduire le coût de communication

=> peuvent être utilisées pour donner à chaque nœud une copie des données de grande taille

### Exemple

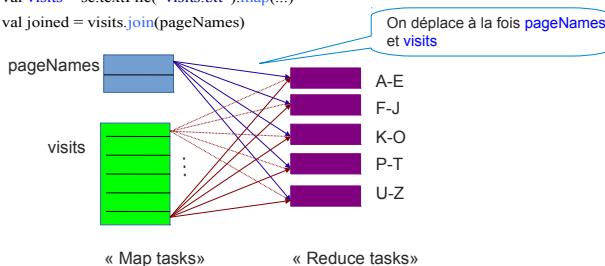
```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

#### Variables broadcast

- créées avec `SparkContext.broadcast(valeurInitiale)`
- `broadcastVar` doit être utilisée à la place de `Array(1, 2, 3)` (la variable sera envoyée aux nœuds une seule fois).
- Accessible à l'intérieur des tâches avec la méthode `.value` (la première tâche à accéder la variable obtient sa valeur)
- La variable ne doit pas être modifiée après broadcast (la variable serait modifiée sur un seul nœud) afin que tous les nœuds aient la même valeur

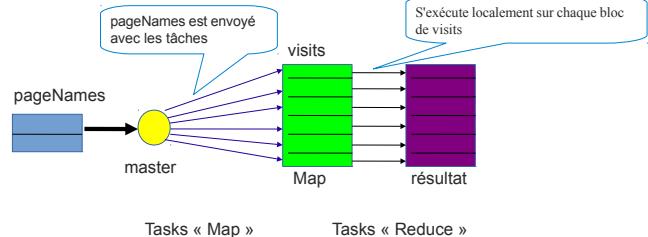
## Exemple : Join

```
// Charger une RDD avec des paires (URL, noms)
val pageNames = sc.textFile("pages.txt").map(...)
// Charger une RDD avec des paires (URL, visites)
val visits = sc.textFile("visits.txt").map(...)
val joined = visits.join(pageNames)
```



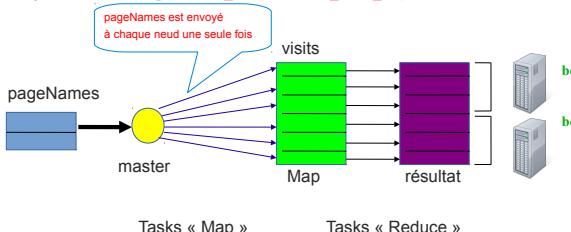
## Si l'une des deux tables est de petite taille

```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap() //les stocker comme un tableau d'objets associatif sur le driver
val visits = sc.textFile("visits.txt").map(...)
val joined = visits.map(v => (v._1, (pageMap(v._1), v._2)))
```



## Meilleure version avec broadcast

```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()
val bc = sc.broadcast(pageMap) //est de type Broadcast[Map[...]]
val visits = sc.textFile("visits.txt").map(...)
val joined = visits.map(v => (v._1, (bc.value(v._1), v._2)))
```



## Accumulateurs

Souvent on a besoin d'agrégé plusieurs valeurs pendant son exécution

### → Accumulateurs

- Variables qui généralisent les compteurs en M/R et peuvent être incrémentées avec une opération associative
- Utilisées pour implanter des compteurs et sommes efficacement en parallèle

Les accumulateurs existants en Spark sont de type numérique et collections mutables (qui peuvent changer pendant l'exécution du programme)

On peut également définir des accumulateurs d'un type utilisateur

## Exemple

- Les tasks peuvent les modifier en utilisant add ou +, ne peuvent pas lire leurs valeurs → chaque modification faite par un task est prise en considération une seule fois
- Seulement le programme driver peut lire les valeurs des accumulateurs en utilisant la méthode .value (une Exception est générée si les tasks essaient de le lire)

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: spark.Accumulator[Int] = 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Int = 10
```

```
Méthode accumulator de SparkContext :
public <T>Accumulator<T> accumulator(T initialValue, String name, AccumulatorParam<T> param)
```

## Autre exemple

```
val badRecords = sc.accumulator(0, Accumulator[Int])
val badBytes = sc.accumulator(0.0, Accumulator[Double])

records.filter(r => {
    if(isBad(r)) {
        badRecords += 1
        badBytes += r.size
        false
    } else {
        true
    }
}).save(...)

printf("Total bad records: %d, avg size: %fn", badRecords.value,
badBytes.value / badRecords.value)
```

## Accumulateurs d'un type utilisateur

Définir un objet qui hérite de l'interface [AccumulatorParam\[T\]](#) (T est le type utilisateur) et implanter les méthodes

- `zero` (« valeur zéro » pour le nouveau type)
- `addInPlace` pour additionner deux valeurs

**Exemple :** pour une classe Vector avec des valeurs réelles

```
class Vector(val data: Array[Double]) {...}

object VectorAP extends AccumulatorParam[Vector] {
    def zero(v: Vector) = new Vector(new Array(v.data.size)) //crée un vecteur de la taille donnée et initialise
                                                               //ses entrées à 0

    def addInPlace(v1: Vector, v2: Vector) = {
        for (i ← 0 to v1.data.size-1) v1.data(i) += v2.data(i)
        return v1
    }
}
```

On peut utiliser maintenant : `sc.accumulator(new Vector(new Array(1,1)), "My Accumulator", VectorAP)`

## Partitionnement de données

- Essayer de réduire le coût de communication en contrôlant le partitionnement des RDD entre les nœuds

### Partitionnement

- Défini pour des pairs (clé, valeur), divise les données en utilisant une fonction applicable sur la clé
- Le partitionnement a un coût => seulement pour des données qui sont re-utilisées plusieurs fois dans des opérations basées sur des clés (eg. `join`)

## Exemple

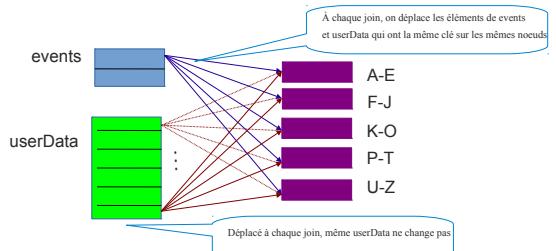
```
val sc = new SparkContext(...)

//charger les informations une seule fois à partir d'un fichier Hadoop SequenceFile
//UserInfo : liste de sujets qui représentent l'intérêt de l'utilisateur
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

//méthode appelée périodiquement pour obtenir le log des actions de l'utilisateur pendant les 5 dernières minutes
//LinkInfo : information sur les liens visités par l'utilisateur
def processNewLogs(logFileName: String) {
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
    val joined = userData.join(events) // RDD (UserID, (UserInfo, LinkInfo))
    val offTopicVisits = joined.filter {
        case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
            userInfo.topics contains(linkInfo.topic)
    }.count()
    println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

## Exemple

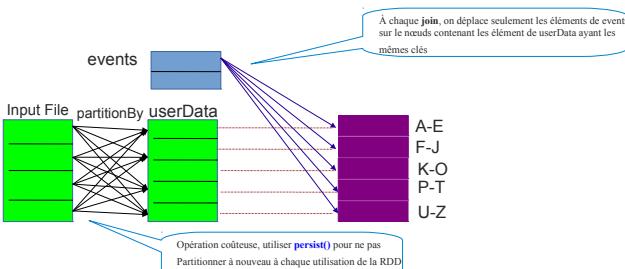
- les entrées de `userData` sont distribuées en fonction du bloc HDFS où elles ont été trouvées (Spark ne connaît pas l'emplacement de l'entrée correspondante à un certain UserID). Utilisée pour toutes les exécutions de `processNewLogs`, **ne change pas**
- `events` : RDD locale à `processNewLogs`, utilisée une seule fois, **change** à chaque exécution de `processNewLogs`



## Solution : utiliser le partitionnement

- créer 5 partitions pour `userData`
- Pas besoin de partitionner `events`

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").partitionBy(new HashPartitioner(5)).persist()
```



## Créer des partitions

- `PartitionBy` : retourne une nouvelle RDD (ne modifie pas celle qu'existe)
- Utiliser `persist()` après `partitionBy` pour ne pas exécuter le partitionnement à chaque accès
- Le nombre de partitions détermine le nombre de tasks qui vont exécuter les opérations sur la nouvelle RDD en parallèle (doit être  $\geq$  au nombre de coeurs de cluster)

### Déjà existants en Spark :

- `HashPartitioner(nbPartitions: Int)` :
  - les données dont les clés ont la même valeur % nbPartitions apparaissent dans la même partition
- `RangePartitioner(nbPartitions: Int, rdd: RDD[_ <: Product2[K, V]], ascending: Boolean = true)` :
  - les données avec les clés dans la même plage de valeurs apparaissent dans la même partition

# Partitionnement

Chaque RDD a un objet `Partitioner` qui est optionnel

Pour chaque opération qui implique un déplacement de données :

- Pour une RDD qui a un `Partitioner`, l'opération prend en compte ce `Partitioner`, les valeurs pour chaque clé sont calculées d'abord localement sur chaque machine et seulement le résultat est envoyé au master
- Appliquée à deux RDD, elle prend en compte le `Partitioner` d'une des deux RDD, s'il existe (les données de cette RDD ne seront pas déplacées). Aucune donnée n'est déplacée si :
  - Les deux RDD ont le même `Partitioner` et sont distribuées sur les mêmes machines ou l'une d'entre elles n'est pas encore calculée
  - Spark utilise par défaut `HashPartitioner`

# Utilisation du partitionnement

## Opérations qui utilisent le partitionnement

- `cogroup`, `groupWith`, `join`, `leftOuterJoin`, `rightOuterJoin`, `groupByKey`, `reduceByKey`, `combineByKey`, `and lookup`.

## Opérations qui changent le partitionnement

- `cogroup`, `groupWith`, `join`, `leftOuterJoin`, `rightOuterJoin`, `groupByKey`, `reduceByKey`, `combineByKey`, `partitionBy`, `sort`
- si la RDD à laquelle on l'applique a un `Partitioner` : `mapValue`, `flatMapValue`, `filter`
- L'objet `Partitioner` est créé automatiquement sur les RDD créés par des opérations qui partitionnent les données, pour les opérations binaires on obtient :
  - le même `Partitioner` que l'un des deux opérande ayant un `Partitioner`
  - le `Partitioner` du premier opérande
  - si aucun opérande n'est partitionné : `HashPartitioner`

Toute autre opération produit des résultats sans `Partitioner`

# Connaître le partitionnement existant

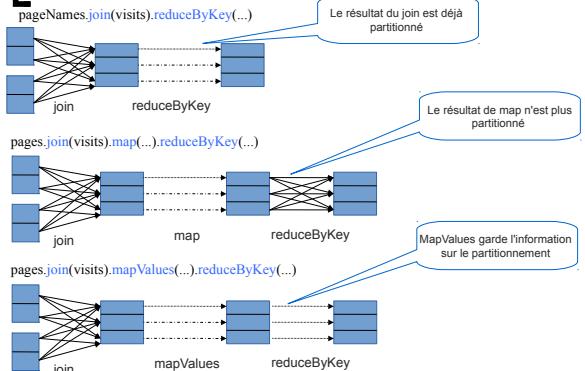
On utilise la méthode `.partitioner` sur une RDD

```
scala> val a = sc.parallelize(List((1, 1), (2, 2)))
scala> val b = sc.parallelize(List((1, 1), (2, 2)))
scala> val joined = a.join(b)

scala> a.partitionner
res0: Option[Partitioner] = None

scala> joined.partitionner
res1: Option[Partitioner] = Some(HashPartitioner@286d41c0)
```

# Exemples

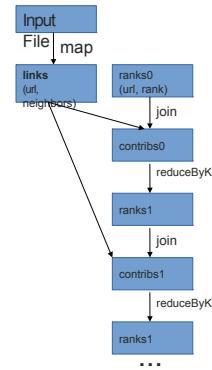


## [ Exemple : PageRank ]

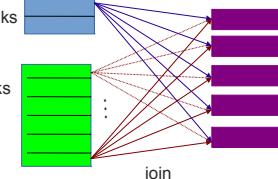
- Le score de chaque page est initialisé à 1
- À chaque itération, une page p envoie le score  $\text{rank}(p)/\text{out}(p)$  à ses voisins
- Le score de chaque page q est calculé comme étant  $0.15 + 0.85 * \text{contib}(q)$

```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs
for (i <= 1 to ITERATIONS) {
    val contribs = links.join(ranks).flatMap {
        case (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    ranks = contribs.reduceByKey(_ + _).mapValues(.15 + .85*_)
}
```

## [ Exécution sans partitionnement ]



À chaque join on déplace links et ranks (regroupés en fonction de leurs clés avec HashPartitioner)

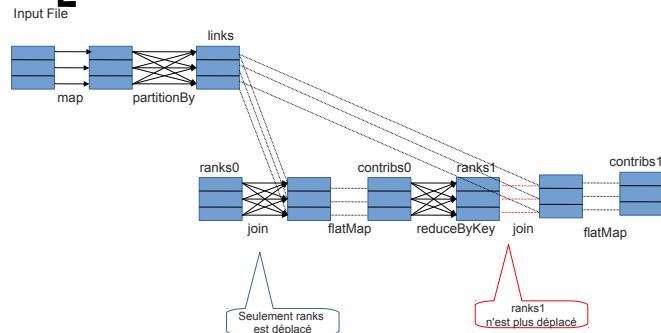


## [ Solution ]

Partitionner links :

```
val links = sc.textFile(...).map(...).partitionBy(new HashPartitioner(8)).persist()
var ranks = // RDD of (url, rank) pairs
for (i <= 1 to ITERATIONS) {
    val contribs = links.join(ranks).flatMap {
        case (url, (links, rank)) =>
            links.map(dest => (dest, rank/links.size))
    }
    ranks = contribs.reduceByKey(_ + _).mapValues(.15 + .85*_)
}
```

## [ Nouvelle exécution ]



## [ Définir son propre partitionnement ]

Extension de la classe `Partitioner`, écrire les méthodes :

- `NumPartitions` : retourne le nombre de partitions
- `getPartition` : retourne la partition d'une clé donnée (entre 0 et NumPartitions)
- `equals` : utile pour décider si deux RDD sont partitionnés de la même manière

*Exemple : regrouper les pages du même domaine dans la même partition*

```
class DomainPartitioner(numParts: Int) extends Partitioner {  
    override def numPartitions: Int = numParts  
    override def getPartition(key: Any): Int = {  
        val domain = new java.net.URL(key.toString).getHost()  
        val code = (domain.hashCode % numPartitions)  
        if (code < 0) {code + numPartitions // Make it non-negative}  
        } else {code}  
    }  
    override def equals(other: Any): Boolean = other match {  
        case dnp: DomainNamePartitioner => dnp.numPartitions == numPartitions  
        case _ => false  
    }  
}
```

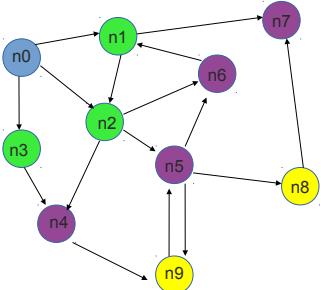
## [ Algorithmes de calcul sur les graphes ]

## [ Plus court chemin ]

### [ Problème ]

- Trouver la longueur du plus court chemin à partir d'un nœud donné  $s$  vers les autres nœuds
- *Centralisé*: l'algorithme de Dijkstra
- *MapReduce* : BFS en parallèle à partir du nœud initial  $s$ 
  - Intuition (pour un graphe non pondéré):
    - Pour tous les voisins  $p$  de  $s$ :  
 $DISTANCE(p) = 1$
    - Pour tous les nœuds  $n$  atteignables à partir d'un ensemble de nœuds  $M$   
 $DISTANCE(n) = 1 + \min_{m \in M} DISTANCE(m)$

## BFS



## Algorithme

Données :

- Clé : nœud n
- Valeur : distance d à partir de s, liste de voisins AdjacencyList => node N
- Initialisation : d=  $\infty$  pour tous les nœuds sauf s

```
1: class MAPPER
2:   method MAP(nid n, node N)
3:     d ← N.DISTANCE
4:     EMIT(nid n, N)                                ▷ Pass along graph structure
5:     for all nodeid m ∈ N.ADJACENCYLIST do
6:       EMIT(nid m, d + 1)                          ▷ Emit distances to reachable nodes
7:
8: class REDUCER
9:   method REDUCE(nid m, [d1, d2, ...])
10:    dmin ←  $\infty$ 
11:    M ←  $\emptyset$ 
12:    for all d ∈ counts [d1, d2, ...] do
13:      if IsNode(d) then
14:        M ← d
15:      else if d < dmin then
16:        dmin ← d
17:    M.DISTANCE ← dmin
18:    EMIT(nid m, node M)                            ▷ Recover graph structure
19:                                              ▷ Look for shorter distance
20:                                              ▷ Update shortest distance
```

## BFS

- Plusieurs itérations sont nécessaires pour explorer tout le graphe( à chaque itération on découvre des nouveaux nœuds)
- Lorsqu'un nœud est trouvé pour la première fois on obtient la plus courte distance (pas vrai pour des graphes pondérés où les arcs peuvent avoir des poids différents)
- *Arrêt de l'algorithme* : lorsque la distance de tous les nœuds est différente de  $\infty$  (on suppose que le graphe est connecté). Nombre d'itérations : le diamètre du graphe
- *Comparaison*:
  - BFS : À chaque itération calcule les distances vers tous les nœuds =>beaucoup de calculs inutiles, pas de structure globale
  - Dijkstra : plus efficace, à chaque fois qu'un nœud est exploré son plus court chemin a déjà été trouvé, nécessite une structure globale

## PageRank

## [PageRank: principe]

Liens hypertexte = recommandations



### Principe

- Les pages avec beaucoup de recommandations sont plus importantes
- Importance aussi de *qui* donne la recommandation

*être recommandé par Yahoo! est mieux que par X*

*la recommandation compte moins si Yahoo! recommande beaucoup de pages → l'importance d'une page dépend du nombre et de la qualité (importance de celui qui recommande) de ses liens entrants*

## [PageRank simplifié]

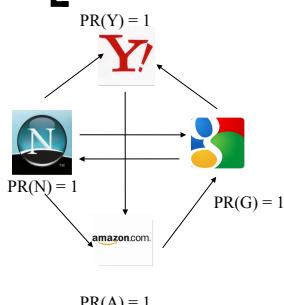
Recommandation donnée par **Y!** à **N** :

$$\frac{PR(Y)}{|out(Y)|} \quad \text{où} \quad \left\{ \begin{array}{l} PR(Y) = \text{l'importance de } Y \\ |out(Y)| = \text{nombre de liens sortants de } Y \end{array} \right.$$

Importance de Netscape est la somme de ses recommandations

$$PR(N) = \sum \frac{PR(P)}{|out(P)|} \quad P = \text{pages qui recommandent N}$$

## [Exemple]



$$PR(A) = PR(N)/3 + PR(Y)/3$$

$$PR(Y) = PR(N)/3 + PR(G)/2$$

$$PR(N) = PR(G)/2$$

$$PR(G) = PR(A) + PR(N)/3$$

## [Calcul des valeurs PR]

Résolution système linéaire

- 4 équations avec 4 inconnues
- pas de solution unique

→ ajouter la contrainte  $PR(A) + PR(Y) + PR(N) + PR(G) = 1$  pour assurer l'unicité

Observation:

système linéaire de grandes dimensions, beaucoup de pages sans liens sortants => les méthodes de calcul directes (ex. méthode de Gauss) sont plus coûteuses que les *méthodes itératives*

## Représentation matricielle

On considère  $n$  pages, pour chaque page  $i$ , on note:

- $out(i)$  est l'ensemble de pages  $j$  référencées par  $i$

$M(m_{ij})$  est la matrice d'adjacence associée au graphe du Web

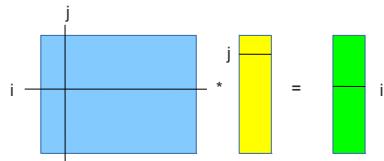
- $m_{ij}$ : fraction de l'importance de  $j$  qui est donnée à  $i$  ( $m_{ij} = 1/|out(j)|$ , si  $j$  a des liens sortants,  $p_{ij} = 0$  dans le cas contraire)
- ligne  $i$  = fractions d'importance reçues par  $i$
- colonne  $j$  = distribution de l'importance de  $j$  (pour les pages  $j$  avec des liens sortants, la somme des éléments sur les colonnes est 1)

$PR(PR_1, PR_2, \dots, PR_n)$  est le vecteur des inconnues (importance)

$PR_i$  est l'importance de la page  $i$

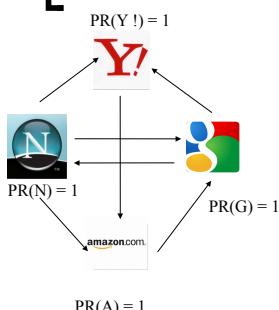
## Exemple

$$\text{Mise à jour de } PR_i : PR_i = \sum_{j \in out(i)} \frac{PR_j}{|out(j)|}$$



$$PR_i = \sum m_{ij} * PR_j$$

## Exemple



$$PR(A) = PR(N)/3 + PR(Y)$$

$$PR(Y) = PR(N)/3 + PR(G)/2$$

$$PR(N) = PR(G)/2$$

$$PR(G) = PR(A) + PR(N)/3$$

$$PR = \begin{matrix} \begin{matrix} PR(Y!) & PR(Y) & PR(N) & PR(A) & PR(G) \end{matrix} \\ \begin{matrix} PR(N) \\ PR(Y!) \\ PR(A) \\ PR(G) \end{matrix} \end{matrix} = \begin{matrix} \begin{matrix} Y & Y & N & A & G \end{matrix} \\ \begin{matrix} 1/3 & 1/3 & 1/2 & 1/2 \\ 1/3 & 1/3 & 1 & 1/3 \\ 1/3 & 1/3 & 1 & 1/3 \end{matrix} \end{matrix} * \begin{matrix} \begin{matrix} PR(Y!) \\ PR(N) \\ PR(A) \\ PR(G) \end{matrix} \end{matrix}$$

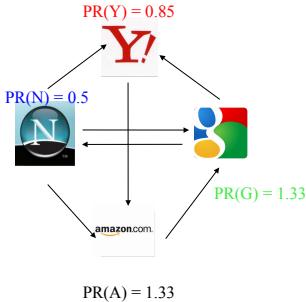
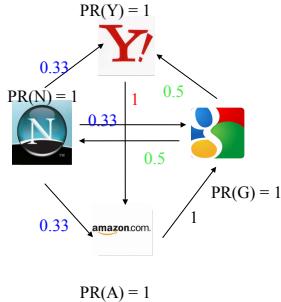
## Algorithme de calcul itératif

- Un graphe avec  $n$  nœuds
  - Initialisation :  $PR^0 = [1, \dots, 1]$
  - À chaque itération  $k$ , recalculer  $PR^{(k)}$
- $$PR^{(k)} = M * PR^{(k-1)}$$
- Arrêt du calcul (convergence) :
- $$\sum |PR_i^k - PR_i^{k-1}| < \varepsilon \in (0,1)$$

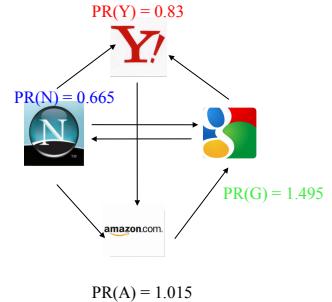
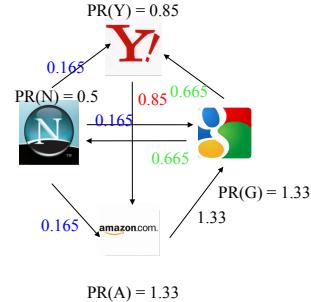
Le vecteur  $PR$  obtenu à la convergence satisfait la condition :

$$PR = M * PR$$

## [ Exemple – Itération 1 ]



## [ Exemple – Itération 2 ]



## [ Algorithme itératif complet ]

- Un graphe avec  $n$  nœuds
- Initialisation :  $PR^0 = [1, \dots, 1] = I_n$

À chaque itération  $k$ , recalculer  $PR^{(k)}$

$$PR^{(k)} = d * M * PR^{(k-1)} + (1-d)I_n$$

$$\text{(ou équivalent : } \forall i : PR_i^k = d * \sum_j \frac{PR_j}{|\text{out}(j)|} + (1-d) \text{ )}$$

$$\text{tant que : } \sum_i |PR_i^k - PR_i^{k-1}| < \epsilon \in (0, 1)$$

- le facteur de décroissance  $d$  (valeur habituelle 0.85) est utilisé pour assurer l'uniquité du vecteur PR calculé et la convergence du calcul itératif

## [ PageRank en MapReduce ]

method MAP(nodeid n)

$$p \leftarrow d * \frac{PR(n)}{|\text{out}(n)|}$$

for all nodeid m in out(n) do

EMIT(m, p)

method REDUCE(nodeid m, [p<sub>1</sub>, p<sub>2</sub>, ...])

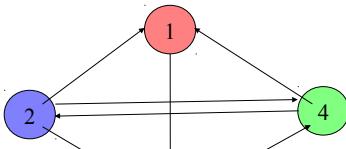
$$s \leftarrow (1-d)$$

for all p in [p<sub>1</sub>, p<sub>2</sub>, ...] do

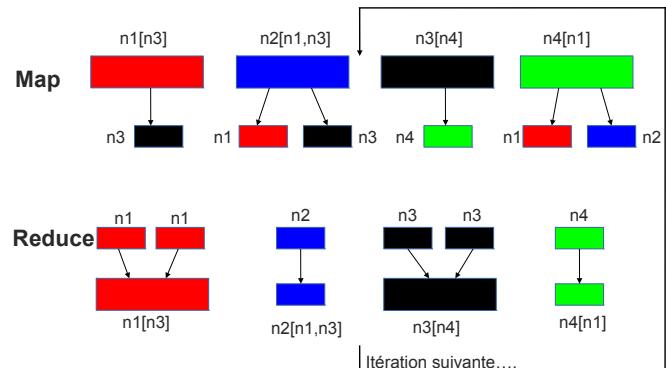
$$s \leftarrow s + p$$

$$PR(m) \leftarrow s$$

## Exemple

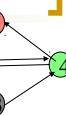


## [ PageRank en MapReduce ]



## [ Encodage de la matrice ]

- Matrice creuse (beaucoup d'entrées sont 0)
- Stocker les entrées non nulles (listes d'adjacence)
- L'espace de stockage est proportionnel au nombre d'arcs
- Pour chaque noeud on stocke également le nombre de liens sortants
- Le vecteur PR est également de grandes dimensions (prop. au nombre de noeuds)



Nœud source	degré	Nœuds destination
1	1	3
2	3	1, 3, 4
3	1	4
4	2	1, 2

## [ Calcul de PageRank ]

### Map :

- Un Map task travaille sur une portion de la matrice M et du vecteur  $PR(k-1)$  et produit une partie de  $PR(k)$
- La fonction Map s'applique à un seul élément  $mij$  de la matrice M et produit la paire  $(i, d^{mij} \cdot PR_j) \Rightarrow$  tous les termes de la somme qui permet de calculer le nouveau  $PR_i$  auront la même clé
- On utilise également un combiner pour agréger localement les valeurs produites par le même Map task

=> Définir des stratégies de partitionnement de la matrice et des vecteurs qui tiennent compte de la capacité mémoire des nœuds de calcul exécutant les tasks

### Reduce :

- La fonction Reduce additionne les termes avec la même clé i, et produit la paire  $(i, PR_i)$

## Partitionnement en bandes

### Map

- La matrice M est partitionnée en  $B$  bandes verticales (une bande correspond à un ensemble de nœuds source)
- Le vecteur  $PR^{(k-1)}$  est partitionné en  $B$  bandes horizontales  
=> chaque task reçoit une bande  $M_b$  de M et la bande correspondante de  $PR_b^{(k-1)}$
- Chaque task produit une version locale du vecteur  $PR^k$  (de même taille que le vecteur  $PR^k$ ) :  $PR_b^k = ((1, PR_b^k(1)), \dots, (n, PR_b^k(n)))$

### Reduce

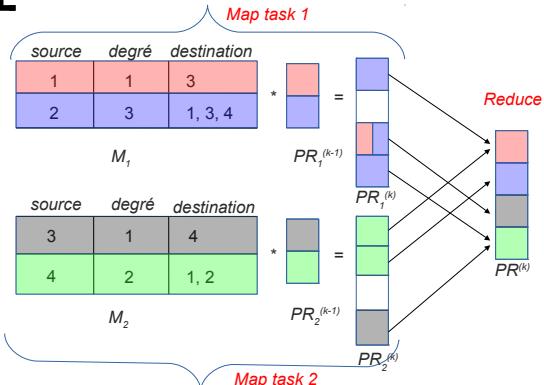
- Aggrégation somme des vecteurs  $PR_b^k$  en fonction des clés

**Avantage de cette méthode :** on stocke seulement une partie(bande) de M et de

$PR^{(k-1)}$  dans la mémoire locale d'un nœud de calcul

**Inconvénient :** on doit stocker le vecteur  $PR_b^k$  entièrement (possiblement de même taille que  $PR^k$ ) => problème si pas assez de mémoire

## Exemple



## Partitionnement en blocks

### $B*B$ Map tasks

### Map

- La matrice M est partitionnée en  $B*B$  blocks
- Le vecteur  $PR^{(k-1)}$  est partitionné en  $B$  bandes horizontales  
=> chaque task reçoit un block  $M_{ib}$  de M et une bande de  $PR_b^{(k-1)}$  ( $PR_b^{(k-1)}$  est transmise  $B$  fois) : à chaque task qui reçoit un block  $M_{ib}$ , pour i de 1 à B)

$M_{11}$	....	$M_{1B}$
:	:	:
$M_{B1}$	....	$M_{BB}$

- Chaque task produit une version locale du vecteur  $PR_i^k$  :
- $PR_{ib}^k = ((1, PR_{ib}^k(1)), \dots, (i, PR_{ib}^k(i)))$

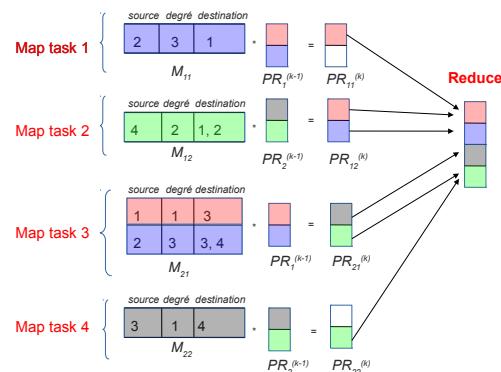
**Reduce :** somme des vecteurs  $PR_{ib}^k$  en fonction des clés

**Avantage de cette méthode :** on stocke seulement une partie(block) de M et de

$PR^{(k-1)}$  dans la mémoire locale d'un nœud de calcul, ainsi qu'une partie du vecteur final => tout peut être stocké en mémoire

**Inconvénient :** chaque bande  $PR_b^{(k-1)}$  du vecteur  $PR^{(k-1)}$  doit être répliquée plusieurs fois

## Exemple



## Quelques problèmes liés au score PR

### Mesure de popularité générique

- ne tient pas compte de la sémantique des informations associées aux entités classées (documents, utilisateurs ...), le sujet de la requête, les préférences de l'utilisateur → versions de PR améliorées

### Une seule mesure d'importance

- autres méthodes : *HITS*

### Influencé par des liens de spam

- Liens artificiels créés pour influencer le score PR → solutions pour détecter ces liens et corriger les scores

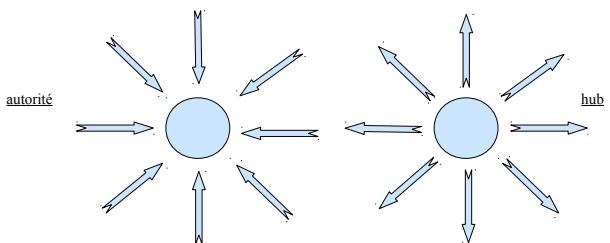
## HITS

## HITS

- HITS = Hypertext Induced Topic Search
- Une autre technique d'utilisation du graphe du Web pour le classement
- Part de l'idée qu'il existe 2 rôles pour une page: **hub et autorité**
- Chaque page peut être un hub, une autorité, ou les 2
- On donne par conséquent **2 scores** (scores de hub et d'autorité) à chaque page au lieu d'un seul
- Les scores sont calculés *au moment* de la requête

## Quelques définitions

- Une autorité est une page qui fournit des informations importantes et fiables sur un sujet donné (comme dans PageRank)
- Un hub est une page qui contient une collection de liens vers des autorités sur un sujet donné



## [ Idée de l'algorithme ]

Les bonnes pages de hubs ont des liens vers les pages d'autorité les plus intéressantes

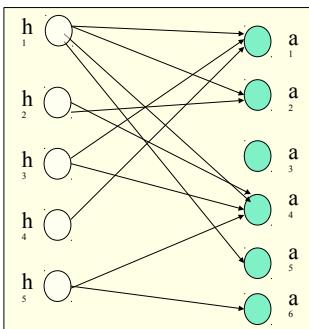
(ex.: le fan de Star Wars aura de nombreux liens vers des pages sur des sites importants dédiés au film)

Les pages d'autorité importantes sont pointées par des hubs importants

(ex: le site officiel de l'université est « pointé » par les sites des différents masters et des universités collaborant)

=> relation de renforcement mutuel

## [ Exemple ]



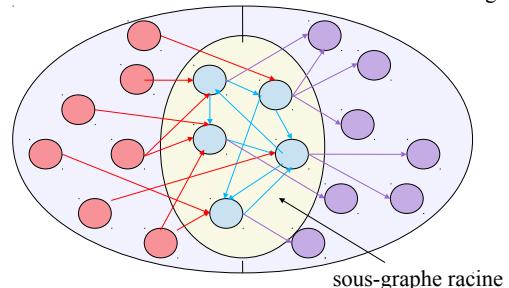
**Score de hub:**  
 $h_1 = a_1 + a_2 + a_4 + a_5$   
 $h_4 = a_1$   
donc  $h_1$  meilleur hub

**Score d'autorité:**  
 $a_2 = h_1 + h_2$   
 $a_4 = h_1 + h_2 + h_3 + h_5$   
donc  $a_4$  meilleure autorité

## [ Trouver les autorités et hubs ]

on construit d'abord un sous-graphe du web centré sur le sujet souhaité (en fonction de la requête)

ensuite on calcule les scores de hubs et d'autorités dans ce sous-graphe



## [ Calcul des scores ]

Algorithme itératif :

- pour chaque page p on maintient un **score de hub**  $h(p)$  et un **score d'autorité**  $a(p)$
- à chaque itération on calcule d'abord les scores d'autorité à partir des scores de hub précédents:

$$a(p)^k = \sum_{q \text{ pointe vers } p} h(q)^{k-1}$$

- avec ce nouveau score on calcule les scores de hub :

$$h(p)^k = \sum_{p \text{ pointe vers } q} a(q)^k$$

## [ Calcul des scores (suite) ]

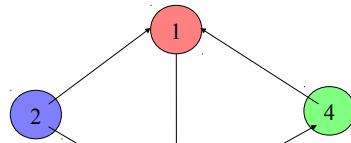
- à chaque itération, lorsque tous les scores ont été calculés, on les normalise (on peut utiliser n'importe quelle norme, on veut les valeurs relatives):

$$a(p)^k = \frac{a(p)^k}{\sqrt{\sum |a(p)|^k}} \quad h(p)^k = \frac{h(p)^k}{\sqrt{\sum |h(p)|^k}}$$

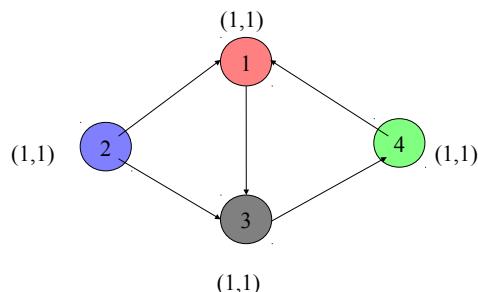
- test de convergence pour un seuil de tolérance  $\varepsilon$  :

$$\sum (h_i^k - h_i^{(k-1)})^2 < \varepsilon \quad \sum (a_i^k - a_i^{(k-1)})^2 < \varepsilon$$

## [ Exemple ]

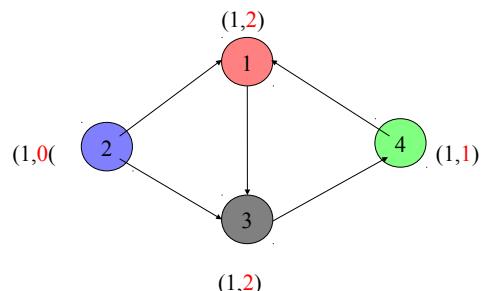


## [ Exemple ]



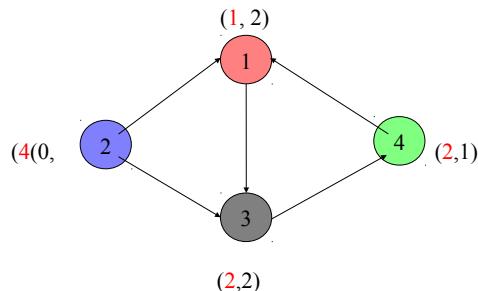
Initialisation

## [ Exemple ]

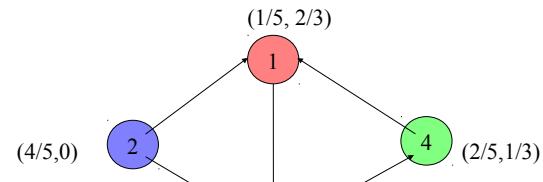


Calcul des scores d'autorité

## Exemple



## Exemple



$$\sqrt{\sum_{i \in [1..4]} h_i^2} = \sqrt{4^2 + 1^2 + 2^2 + 2^2} = \sqrt{25} = 5$$

$$\sqrt{\sum_{i \in [1..4]} a_i^2} = \sqrt{0^2 + 2^2 + 2^2 + 1^2} = \sqrt{9} = 3$$

Normalisation des scores

## Algorithme séquentiel

```

 $a^0 := \left( \frac{1}{\sqrt{n}}, \dots, \frac{1}{\sqrt{n}} \right) \in \mathbb{R}^n$ 
 $h^0 := \left( \frac{1}{\sqrt{N}}, \dots, \frac{1}{\sqrt{N}} \right) \in \mathbb{R}^n$ 
 $k := 1$ 
do    $\forall p :$ 
     $a(p)^k := \sum_{q \text{ pointe vers } p} h(q)^{k-1}$ 
     $h(p)^k := \sum_{p \text{ pointe vers } q} a(q)^k$ 
     $a(p)^k := \frac{a(p)^k}{h(p)^k}$ 
     $h(p)^k := \sqrt{\sum (a(p')^k)^2}$ 
while  $\sum (h_i^k - h_i^{k-1})^2 > \epsilon$ 

```

## Algorithme parallèle

... à réaliser en TME à partir de l'algorithme de PageRank

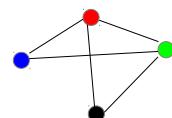
# Compter les triangles

## Pourquoi compter les triangles ?

Coefficient de clustérisation :

- Pour un graphe non dirigé  $G=(V,E)$
- $cc(v) = \text{fraction des voisins de } v \text{ qui sont eux-mêmes des voisins}$

$\binom{d_v}{2}$  est le nombre total d'arcs possibles entre les voisins de  $v$



$$cc(\text{blue}) = 1/1$$

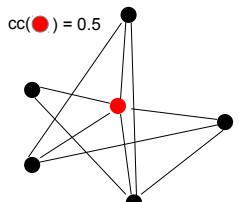
$$cc(\text{red}) = 2/3$$

$$cc(\text{black}) = 1/3$$

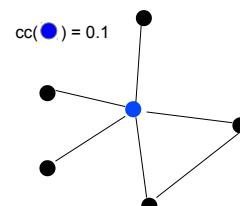
$$cc(\text{green}) = 1/1$$

## Coefficient de clustérisation

Montre la densité de la connectivité autour d'un noeud



vs.



## Comment compter les triangles ?

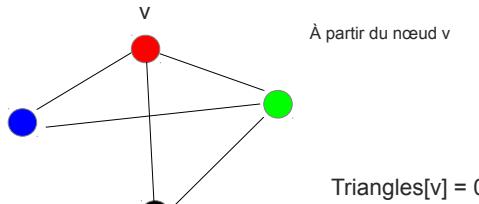
Algorithme séquentiel :

```
Triangles ← 0
foreach v in V
    foreach u,w in Adjacency(v)
        if (u,w) in E
            Triangles += 1/2
return (Triangles / 3)
```

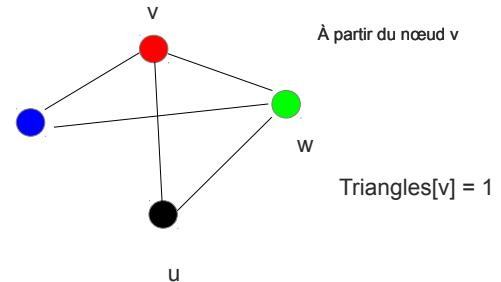
Complexité de l'algorithme :  $O\left(\sum d_v^2\right)$

Chaque triangle est compté 3 fois (une fois par noeud)

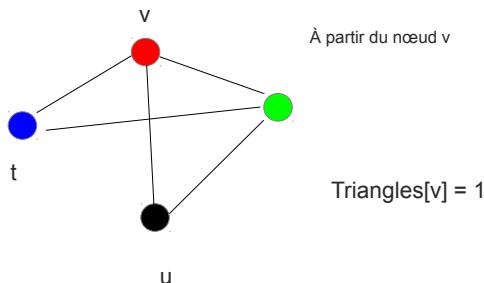
### [Exemple]



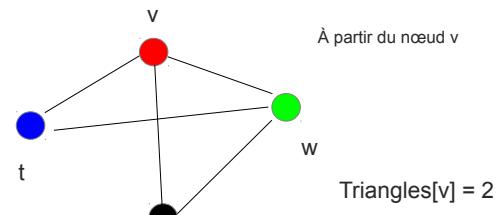
### [Exemple]



### [Exemple]

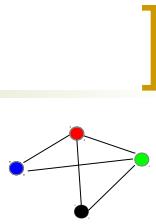


### [Exemple]



## Algorithme parallèle

**Map 1 :** Input :  $\{(v,u) \mid u \in \Gamma(v)\}$   
 foreach  $u \in \Gamma(v)$  emit  $\{(v,u)\}$



**Reduce 1 :** Input :  $\{(v,u) \mid u \in \Gamma(v)\}$   
 foreach  $(u,w) \in \Gamma(v)$  emit  $\{(u,w), v\}$   
 $((\bullet, \bullet), \bullet) ((\bullet, \bullet), \bullet) ((\bullet, \bullet), \bullet)$

**Map 2 :** emit  $\{((u,w), \$) \mid w \in \Gamma(u)\}$

**Reduce 2 :** Input :  $\{((u,w), v1, v2, \dots, vk, \$)\}$   
 foreach  $(u,w)$  if  $\$$  is part of the input, then :  
 $\text{Triangles}[vi] += 1/2$   
 $((\bullet, \bullet), \bullet, \$) \rightarrow \text{Triangles}(\bullet) + 1/2$   
 $((\bullet, \bullet), \bullet) \rightarrow \emptyset$

## Adaptation de l'algorithme

- On génère tous les chemins à vérifier en parallèle, le temps d'exécution est  $\max_{v \in V} |\sum d_v^k| \Rightarrow$  pour les nœuds avec beaucoup de voisins (millions) les reduce tasks correspondants peuvent être très lents

Amélioration :

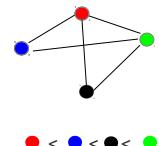
- ordonner les nœuds par leur degré (pour ceux qui ont le même degré par leur identifiant)
- compter chaque triangle une seule fois, à partir du nœud minimum
- Complexité :  $O(m^{3/2})$  ( $m = \text{nombre d'arcs dans le graphe}$ )

## Algorithme amélioré

Algorithme séquentiel :

```
Triangles ← 0
foreach v in V
    foreach u,w in Adjacency(v)
        if  $u > v \&& w > u$ 
            if  $(u,w)$  in E
                Triangles++
```

Return Triangles



**Map 1 :** Input :  $\{(v,u) \mid u \in \Gamma(v)\}$

if  $u > v$  then emit  $\{(v,u)\}$

$(\bullet, \bullet), (\bullet, \bullet), (\bullet, \bullet), (\bullet, \bullet), (\bullet, \bullet), (\bullet, \bullet)$

**Reduce 1 :** Input :  $\{(v,u) \mid u \in S \subset \Gamma(v)\}$

foreach  $(u,w) \in S$

if  $w > u$  then emit  $\{((u,w), v)\}$

$((\bullet, \bullet), \bullet), ((\bullet, \bullet), \bullet), ((\bullet, \bullet), \bullet)$

**Map 2 :** emit  $\{((u,w), \$) \mid w < u\}$

**Reduce 2 :** Input :  $\{((u,w), v1, v2, \dots, vk, \$)\}$

foreach  $(u,w)$  if  $\$$  is part of the input, then :

$\text{Triangles}[vi] +=$

$((\bullet, \bullet), \bullet, \$) \rightarrow \text{Triangles}(\bullet) +=$

$((\bullet, \bullet), \bullet) \rightarrow \emptyset$

## Références

<https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ach04.html>  
<http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-amp-camp-2012-advanced-spark.pdf>  
[https://www.cs.berkeley.edu/~matei/papers/2012/nsdi\\_spark.pdf](https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf)  
Mining of Massive Datasets (Chapitre 5) : <http://infolab.stanford.edu/~ullman/mmds/bookL.pdf>