Calculs sur des graphes en M/R et Spark

Camelia Constantin – LIP6

Prénom.Nom@lip6.fr

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 tes tâches aux workers

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

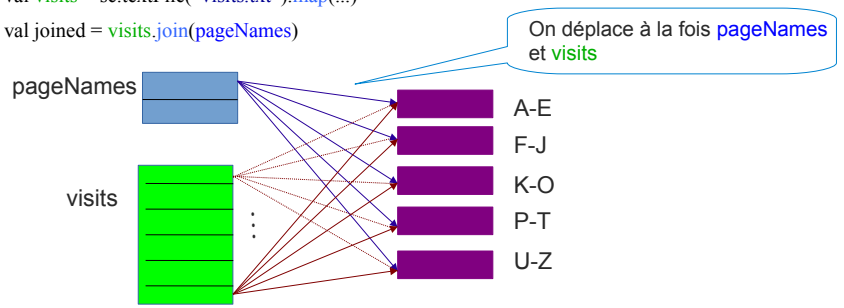
```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

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).
- accessibles 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

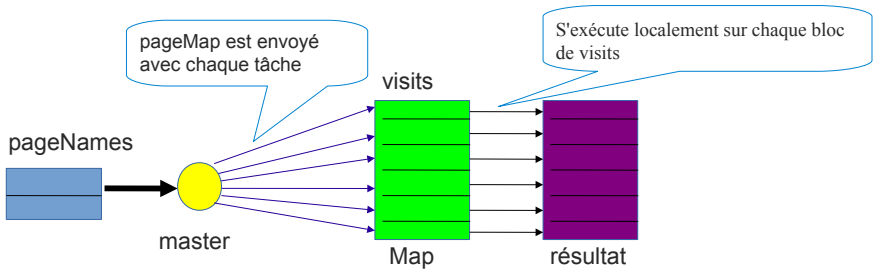
Exemple : Join

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



Si l'une des deux tables est de petite taille

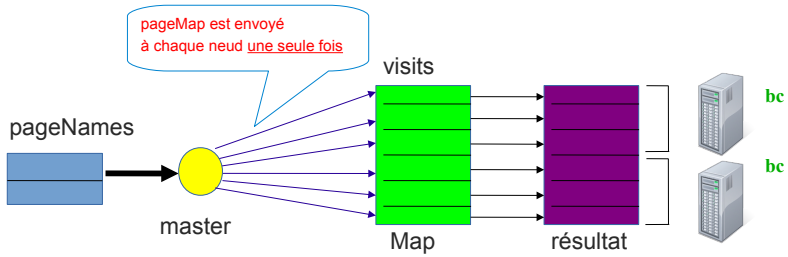
```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()//les stocker comme un tableau associatif d'objets  sur le driver
val visits = sc.textFile("visits.txt").map(...)
val joined = visits.map(v => (v._1, (pageMap(v._1), v._2)))
```



Tasks « Map » Tasks « Reduce »

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)))
```



Tasks « Map » Tasks « Reduce »

Accumulateurs

Souvent on a besoin d'agréger plusieurs valeurs pendant l'exécution

→ ***Accumulateurs***

- Variables qui généralisent les compteurs en M/R et peuvent être incrémentées avec une opération associative
- Utilisés 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*

Accumulateur

- Les tâches peuvent le modifier avec add, ne peuvent pas lire sa valeur
- Accumulateur numérique créé avec *SparkContext.longAccumulator()* ou avec *SparkContext.doubleAccumulator()*
- Seulement le programme driver peut lire les valeurs d'un accumulateur en utilisant la méthode *.value()* (une Exception est générée si les tâches essaient de le lire)

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0)
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
scala> accum.value
res2: Long = 10
```

- mis à jour dans les trasformations uniquement lorsque la RDD est calculée

```
scala> val accum = sc.longAccumulator
scala> data.map{x=> accum.add(x);x}
//accum restera 0 tant qu'aucune action ne déclenchera l'exécution de map
```

Exemple

```
val badRecords = sc.longAccumulator
val badBytes = sc.doubleAccumulator

records.filter(r => {
      if (isBad(r)) {
            badRecords += 1
            badBytes += r.size
            false
            } else {
                  true
            }
      }).save(...)
printf("Total bad records: %d, avg size: %f\n", badRecords.value,
badBytes.value / badRecords.value)
```

Accumulator [Long]

Accumulator [Double]

Accumulateurs de type utilisateur

Définir un objet qui hérite de la classe *AccumulatorV2[IN, OUT]* et implanter ses méthodes abstraites, comme :

- *reset* (remettre la valeur à « zéro »)
- *add* pour additionner deux valeurs, etc.

Exemple : pour une classe Vector avec des valeurs réelles

```
class Vector(val data: Array[Double]) {…}
object VectorAP extends AccumulatorV2[Vector, Vector] {
      private val myVector: Vector = Vector.createZeroVector
      def reset(): Unit = {  myVector.reset() }
      def add(v: Vector): Unit = { myVector.add(v)}
      …..
}
```

On peut utiliser maintenant :

```
scala> val accum = new VectorAP //créer un accumulateur du nouveau type
scala> sc.register(accum, "MyVectorAcc")//l'enregistrer dans spark context
```


Partitionnement de données

- RDD : collections de données de grande taille, ne peuvent pas être stockées dans un seul nœud, doivent être partitionnées sur plusieurs noeuds
- Coût de communication élevé dans un programme distribué : utiliser 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
  - utilise pour des données qui sont re-utilisées plusieurs fois dans des opérations basées sur des clés (eg. **Join**)
  - Inutile de partitionner à l'avance une RDD qui sera utilisée une seule fois

Partitions

- Sont évaluées en mode "lazy"(déclenché par une action, à chaque action Spark recalcule le DAG des transformations)
- Chaque nœud contient une ou plusieurs partitions, chaque partition est stockée sur une seule machine, on ne peut pas choisir le nœud pour une partition
- Une tâche par partition
- Le nombre de partitions peut être configuré(pas assez : pas assez de concurrence, mauvaise utilisation des ressources, trop de partitions : plus de temps pour affecter les tâches que pour leur exécution), par défaut : nb total de cœurs du cluster.

Créer des partitions

- *PartitionBy* : retourne une nouvelle RDD (ne modifie pas celle qui 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 en parallèle sur la nouvelle RDD (doit être ≥ au nombre de coeurs du cluster)

Déjà existants en Spark :

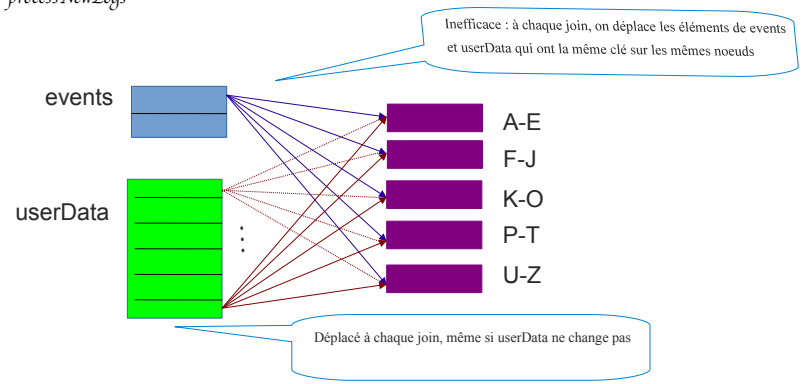
- HashPartitioner(nbPartitions: Int) :
  - les données dont les clés ont la même valeur % nbPartitions apparaissent dans la même partition
- RangePartitioner(nbPartitions: Int, rdd: RDD[_ <: Product2[K, V]], ascending: Boolean = true) :
  - les données avec les clés dans la même plage de valeurs apparaissent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

Exemple

```
val sc = new SparkContext(...)
//charger les informations une seule fois à partir d'un fichier Hadoop SequenceFile,
//fichier lu eeffectivement lors de l'exécution de count(), utiliser persist() pour éviter de le relire
//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)) =>
      !userInfo.topics.contains(linkInfo.topic)
  }.count()  // Combien d'utilisateurs ont visité un lien qui n'appartient pas à un sujet de leur intérêt
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```


Exemple

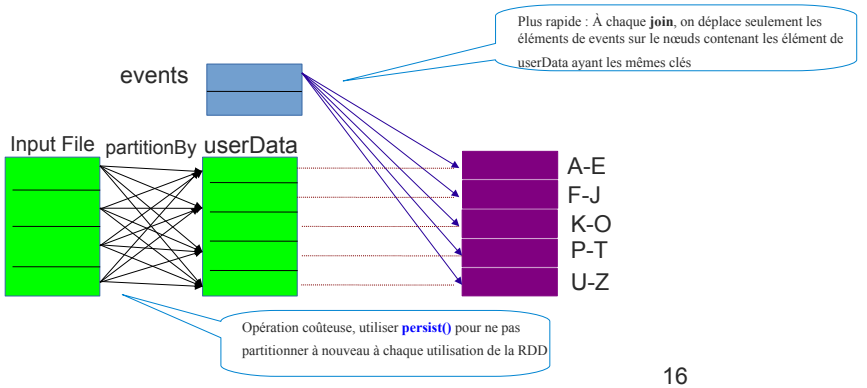
- les entrées de userData sont distribuées en fonction du bloc HDFS où elles ont été trouvées (Spark ne connaît pas l'emplacement de l'entrée correspondante à un certain UserID). Utilisée pour toutes les exécutions de *processNewLogs*, **ne change pas**
- events : RDD locale à *processNewLogs*, utilisée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



Solution : utiliser le partitionnement

- créer 5 partitions pour userData (partitionBy()= transformation, produit toujours une nouvelle RDD)
- pas besoin de partitionner events (locale à processNewLogs(), utilisée une seule fois)

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



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` (nb partitions=degré de parallélisme)

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.partitioner
res0: Option[Partitioner] = None

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


Utilisation du partitionnement

Opérations qui utilisent le partitionnement

- cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, and lookup.
- Si l'opération utilise une seule RDD (e.g reduceByKey) partitionnée les valeurs pour chaque clés sont calculées localement, seulement la valeur finale sera envoyée au master
- Une opération binaire sur deux RDD partitionnées, les données d'au moins une des deux ne seront pas envoyées dans le réseau (aucune donnée ne sera envoyée si elles ont le même partitionnement et sont sur les mêmes machines)

Utilisation du partitionnement

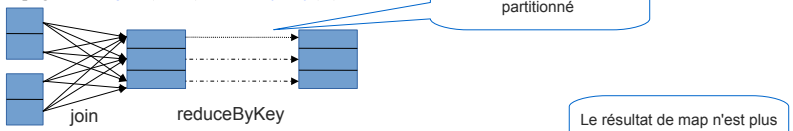
Opérations qui changent le partitionnement

- cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, partitionBy, sort (e.g sortByKey produit un partitionnement range, groupByKey un partitionnement par hachage)
- mapValue, flatMapValue, filter (si la RDD à laquelle on les applique a un `Partitioner`)
- L'objet `Partitioner` est créé automatiquement sur les RDD créées par des opérations qui partitionnent les données. Pour les opérations binaires on obtient :
  - le même `Partitioner` que l'un des deux opérandes ayant un `Partitioner`
  - le `Partitioner` du premier opérande
  - si aucun opérande n'est partitionné : `HashPartitioner`

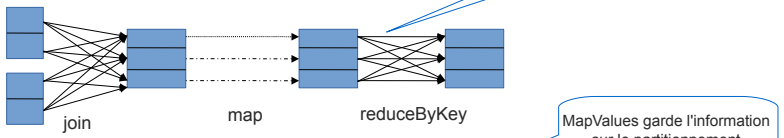
Toute autre opération produit des résultats sans `Partitionner` `(e.g map())`

Exemples

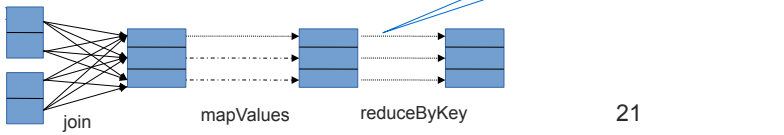
pageNames.join(visits).reduceByKey(...)



pageNames.join(visits).map(...).reduceByKey



pageNames.join(visits).mapValues(...).reduceByKe



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 partitionnées de la même manière

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

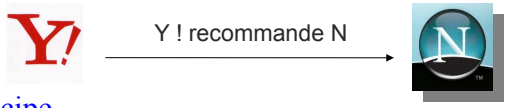
```
class DomainPartitioner(numParts: Int) extends Partitioner {
    override def numPartitions: Int = numParts
    override def getPartition(key: Any): Int = {
            val domain = new java.net.URL(key.toString).getHost()
            val code = (domain.hashCode % numPartitions)
            if (code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


Algorithmes de calcul sur les graphes

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

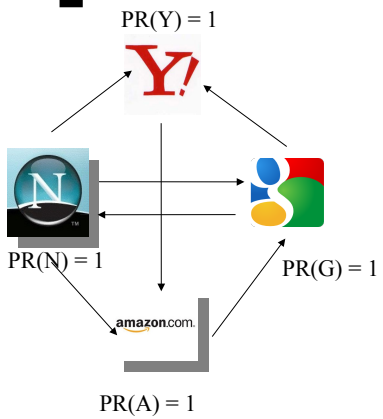
Recommandation donnée par **Y!** à **N** :

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

Importance de **N**etscape est la somme de ses recommandations

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

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

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

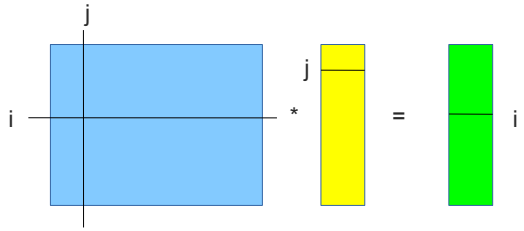
- m_{ij}: fraction de l'importance de j qui est donnée à i ($m_{ij} = 1/|out(j)|$, si j a des liens sortants, $p_{ij} = 0$ dans le cas contraire)
- ligne i = fractions d'importance reçues par i
- colonne j = distribution de l'importance de j (pour les pages j avec des liens sortants, la somme des éléments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

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

PR_i est l'importance de la page i

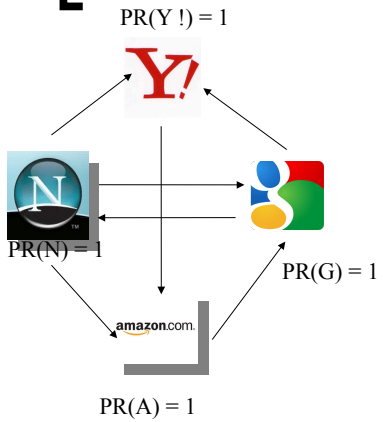
Exemple

Mise à jour de PR_i : $PR_i = \sum \frac{PR_j}{|out(j)|}$



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

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

		Y	N	A	G	
PR(Y)	Y		1/3		1/2	PR(Y)
PR(N)	N				1/2	PR(N)
PR(A)	A	1	1/3			PR(A)
PR(G)	G		1/3	1		PR(G)

PR = M * PR

Algorithme de calcul itératif

- Un graphe avec n nœuds
- Initialisation : $PR^0 = [1,\ldots,1]$
- À chaque itération k, recalculer $PR^{(k)}$

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

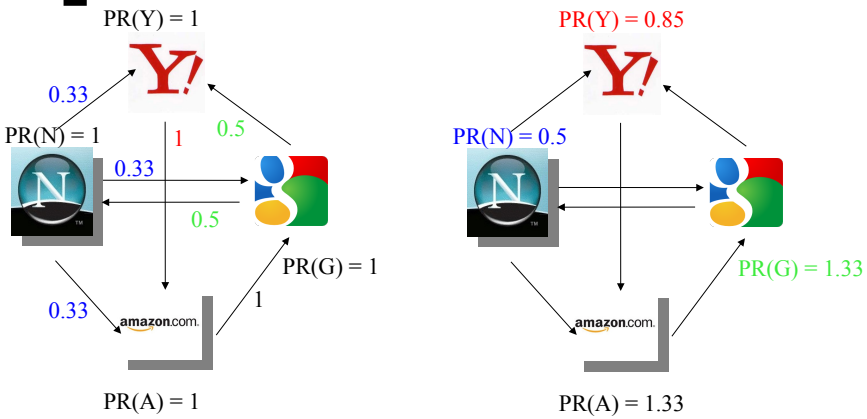
- Arrêt du calcul (convergence) :

$$\frac{\sum |PR_i^k - PR_i^{k-1}|}{\|PR^k\|_1} < \varepsilon, \varepsilon \in (0,1)$$

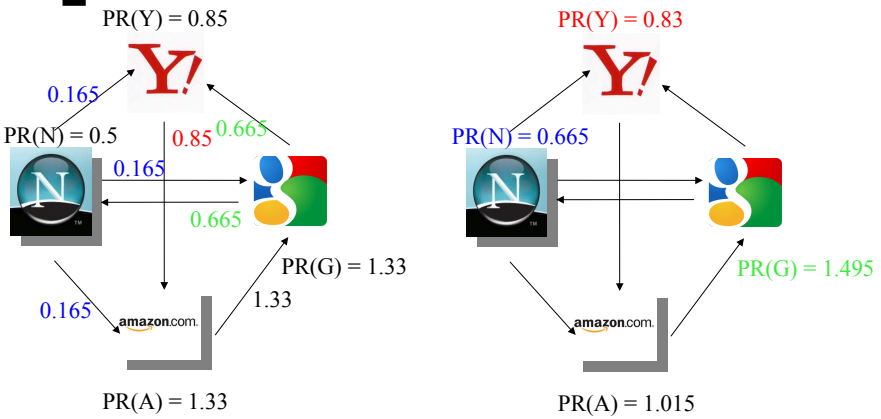
Le vecteur PR obtenu à la convergence satisfait la condition :

$$PR = M * PR$$

Exemple – Itération 1



Exemple – Itération 2



Algorithme itératif complet

Un graphe avec n nœuds

- Initialisation : $PR^0 = [1,\ldots,1]=I_n$
- À chaque itération k, recalculer $PR^{(k)}$

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

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

  arrêt lorsque : $\frac{\sum |PR_i^k - PR_i^{(k-1)}|}{\|PR^k\|_1} < \varepsilon, \varepsilon \in (0,1)$

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

PageRank en MapReduce

method MAP(nodeid n)

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

for all nodeid m in out(n) **do**

EMIT(m, p)

method REDUCE(nodeid m, [p_1, p_2…])

s ← (1-d)

for all p in [p_1, p_2…] **do**

s ← s + p

PR(m) ← s

Calcul de PageRank en M/R

Map :

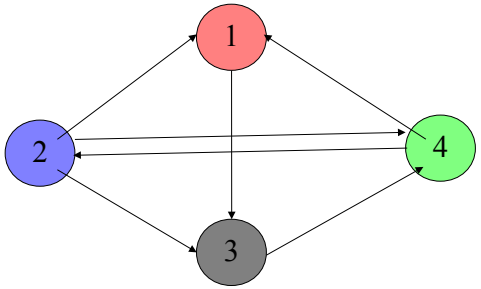
- Un Map task travaille sur une portion de la matrice M et du vecteur $PR^{(k-1)}$ et produit une partie de $PR^{(k)}$
- La fonction Map s'applique à un seul élément mij (1/|out(j)|) de la matrice M et produit la paire (i, d*mij*PRj) =>tous les termes de la somme qui permet de calculer le nouveau PRi 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

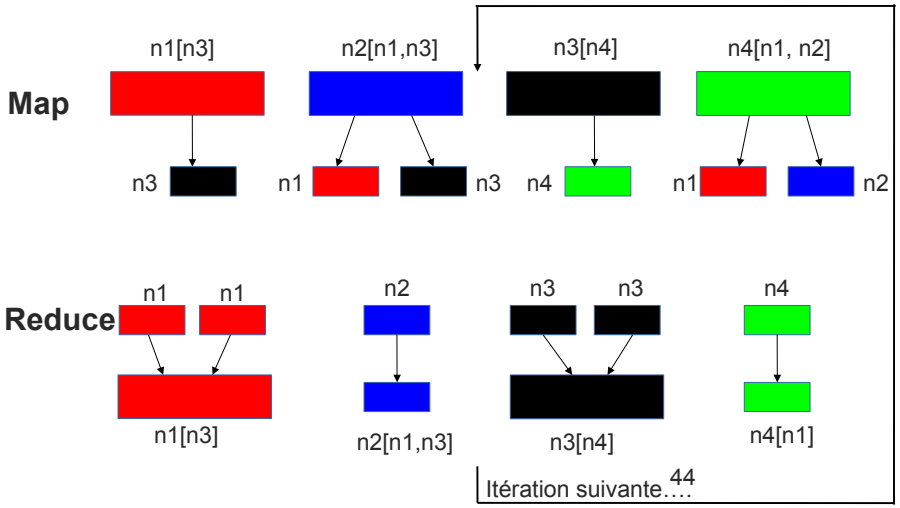
Reduce :

- La fonction Reduce aditionne les termes avec la même clé i, et produit la paire (i, PRi)

Exemple



PageRank en MapReduce

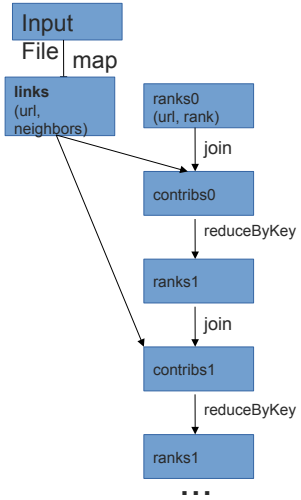


Exemple : PageRank sous Spark

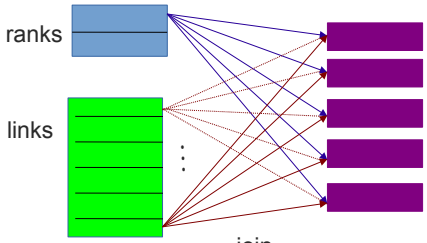
- Le score de chaque page est initialisé à 1
- À chaque itération, une page p envoie le score rank(p)/|numVoisins(p)| à ses voisins
- Le score de chaque page q est calculé comme étant 0.15+0.85*scoresReçus(q)

```
val links = // RDD  (url, neighbors), liste des voisins de chaque page
var ranks = // RDD (url, rank), liste des scores de chaque page
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



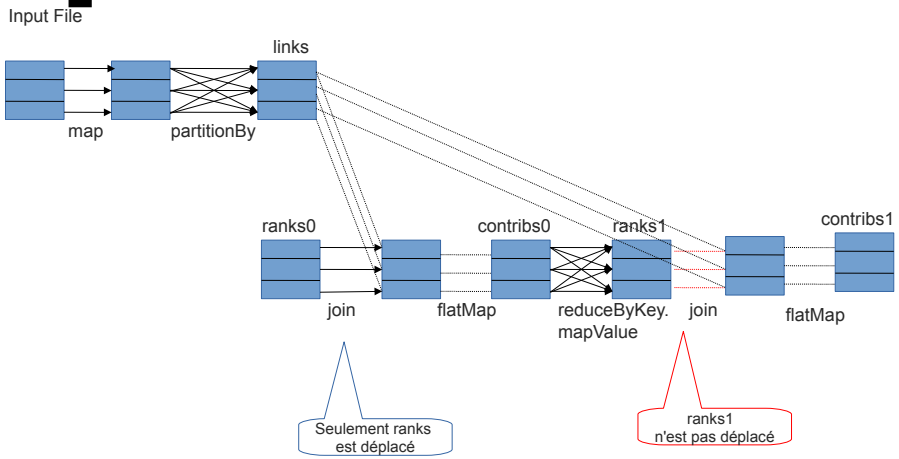
À chaque join on déplace links et ranks (regroupés en fonction de leurs clés avec HashPartitioner)



Exécution avec le partitionnement existant dans Spark

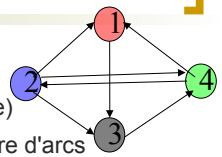
```
val links = sc.textFile(...).map(...).partitionBy(new
    HashPartitioner(8)).persist() //links beaucoup plus grand que ranks, ne sera
 //plus envoyé, gardé en RAM pour toutes les itérations
var ranks = links.mapValues(v=>1.0) //utiliser mapValues pour garder le
    //partitionnement
for (i <- 1 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*_)
    //utiliser mapValues à la place de map, garde le partitionnement de
    reduceByKey et optimise join() suivant
}
```


Nouvelle exécution



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 nœud on stocke également le nombre de liens sortants
- Le vecteur PR est également de grandes dimensions (prop. au nombre de noeuds)



	1	2	3	4
1	0	1/3	0	1/2
2	0	0	0	1/2
3	1	1/3	0	0
4	0	1/3	1	0

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

Partitionnement en bandes

- **B** Map tasks

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) : $PR_b^k=((1,PR_b^k(1)), \ldots, (n,PR_b^k(n))$

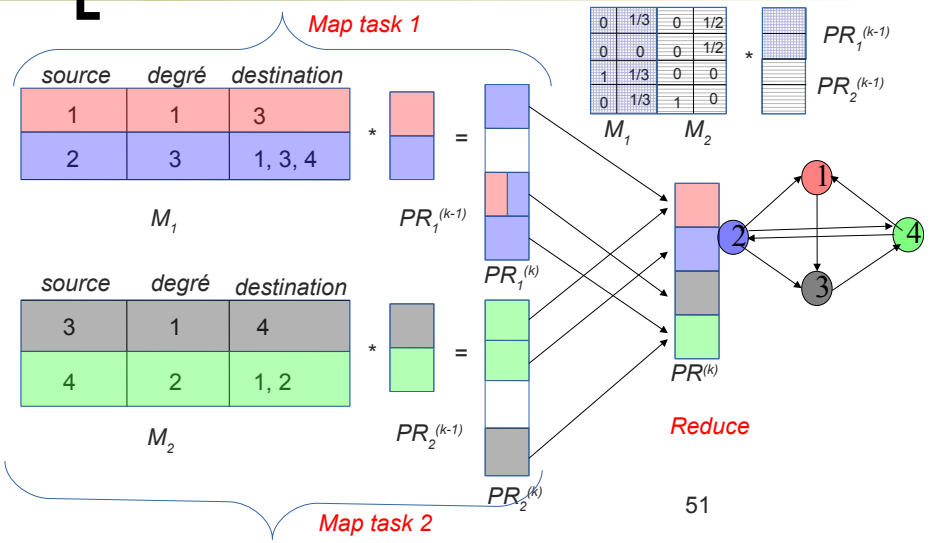
Reduce

- 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

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

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)

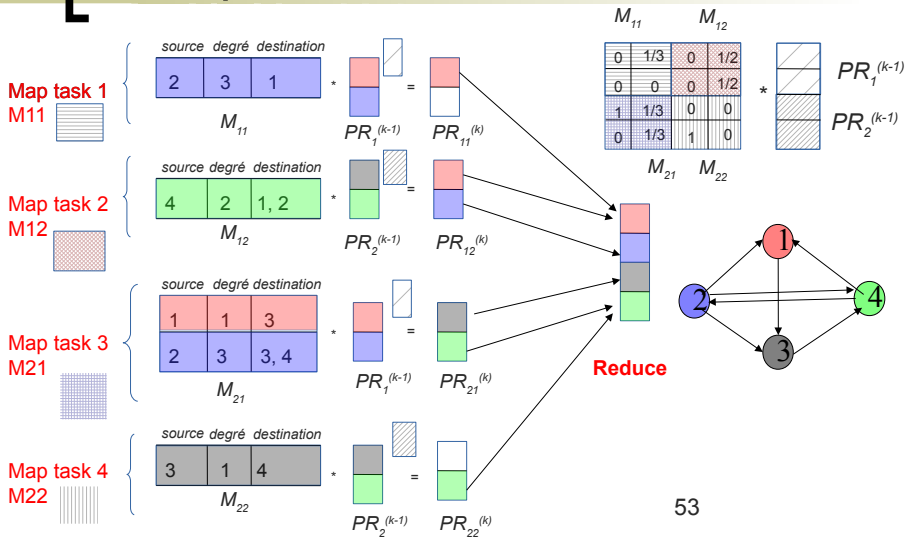
- Chaque task produit une version locale du vecteur PR_i^k :
- $PR_{ib}^k=((1,PR_{ib}^k(1)), \ldots, (i,PR_{ib}^k(i))$

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 crées pour influencer le score PR → solutions pour détecter ces liens et corriger les scores

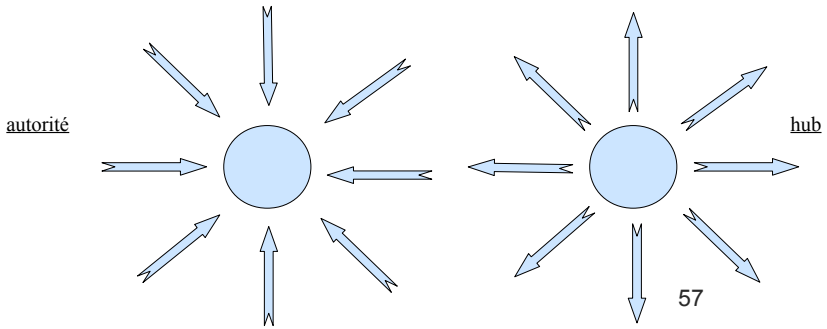
HITS

HITS

- HITS = Hyperlink 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

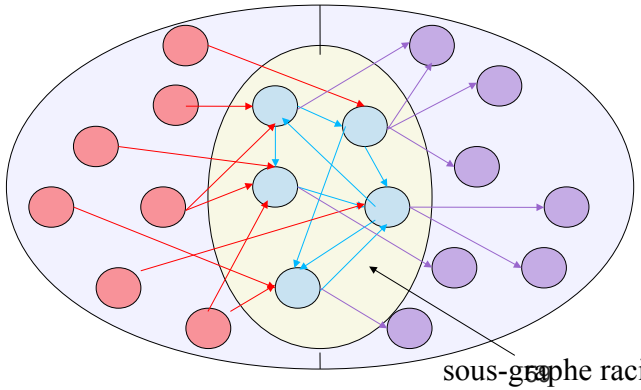
(ex: le site officiel de l'université est « pointé » par les sites des différents masters et des universités collaborant)

=> relation de renforcement mutuel

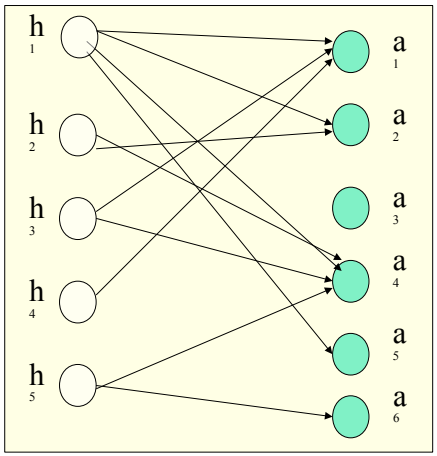
Trouver les autorités et hubs

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



Exemple



Score de hub:
h1 = a1 + a2 + a4 +a5
h4 = a1
donc h1 meilleur hub

Score d'autorité:
a2 = h1 + h2
a4 = h1 + h2 + h3 + h5
donc a4 meilleure autorité

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)

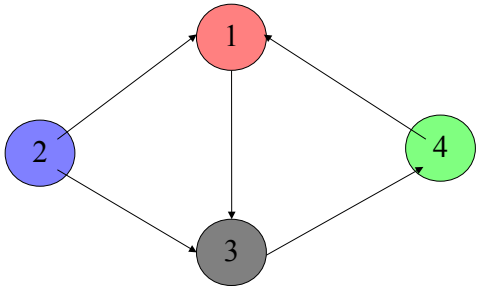
- à chaque itération, lorsque tous les scores ont été calculés, on les normalise (on peut utiliser n'importe quelle norme, on veut les valeurs relatives):

$$a(p)^k = \frac{a(p)^k}{\sqrt{\sum \left(a(p')^k\right)^2}} \qquad h(p)^k = \frac{h(p)^k}{\sqrt{\sum \left(h(p')^k\right)^2}}$$

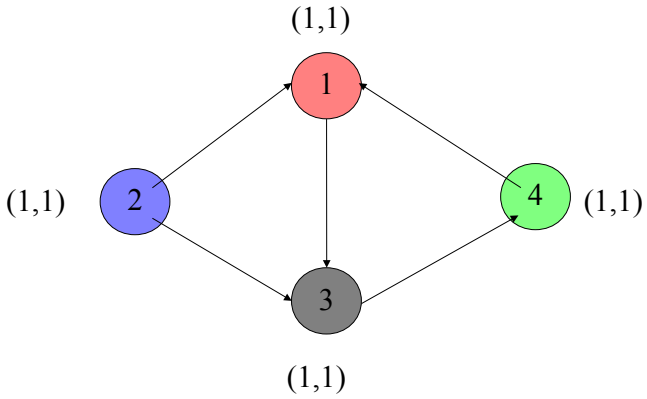
- test de convergence pour un seuil de tolérance ε :

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

Exemple



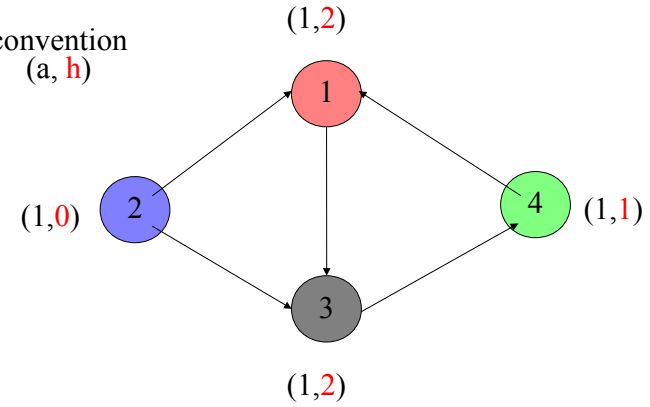
Exemple



Initialisation

Exemple

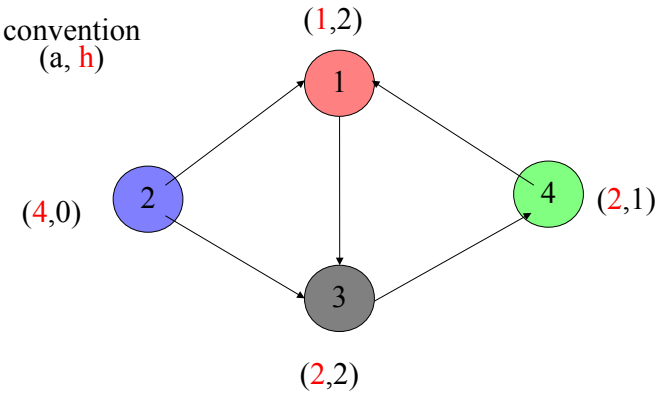
convention
(a, h)



Calcul des scores d'autorité

Exemple

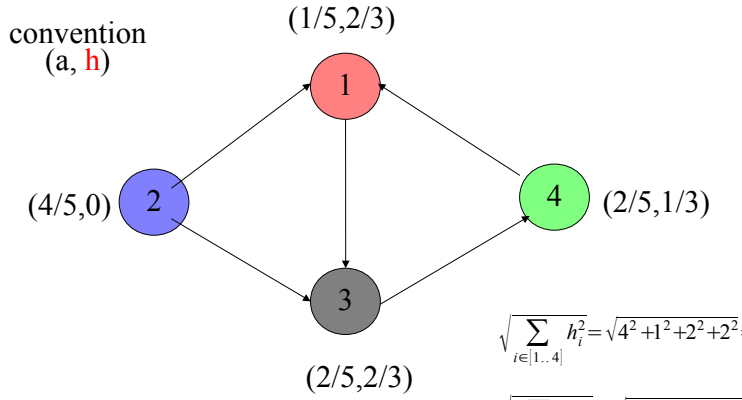
convention
(a, h)



Calcul des scores de hub

Exemple

convention
(a, h)



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

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

Normalisation des scores

Algorithme séquentiel

$a^0 := \left(\frac{1}{\sqrt{n}}, \ldots, \frac{1}{\sqrt{n}}\right) \in \mathbb{R}^n$

$h^0 = \left(\frac{1}{\sqrt{N}}, \ldots, \frac{1}{\sqrt{N}}\right) \in \mathbb{R}^n$

$k := 1$

do $\forall\ p$:

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

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

$$a(p)^k = \frac{a(p)^k}{\sqrt{\sum \left(a(p')^k\right)^2}}$$

$$h(p)^k = \frac{h(p)^k}{\sqrt{\sum \left(h(p')^k\right)^2}}$$

while $\sum \left(h_i^k - h_i^{(k-1)}\right)^2 > \varepsilon$

Algorithme parallèle

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

Compter les triangles

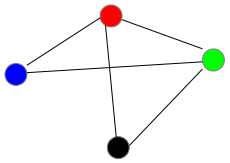
Pourquoi compter les triangles ?

Coefficient de clustérisation :

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

$$= \frac{|\{(u,w) \in E | u \in \Gamma(v) \wedge w \in \Gamma(v)\}|}{\binom{d_v}{2}}$$

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



cc( ) = 1/1

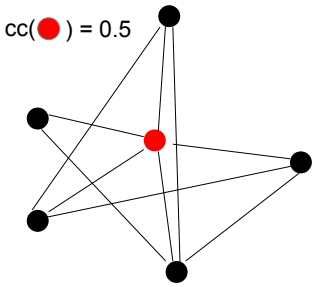
cc( ) = 2/3

cc( ) = 1/1

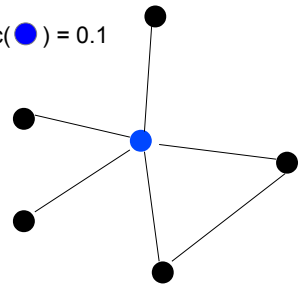
cc( ) = 2/3

Coefficient de clustérisation

Montre la densité de la connectivité autour d'un noeud



vs.



Comment compter les triangles ?

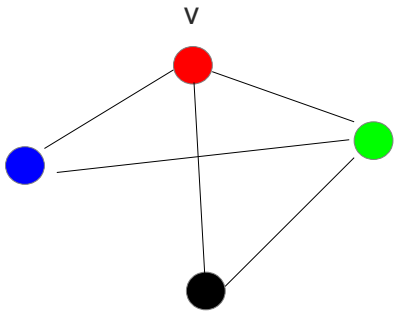
Algorithme séquentiel (graphe non dirigé) :

```
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)

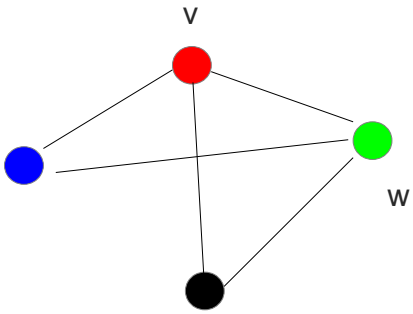
Exemple



À partir du nœud v

Triangles = 0

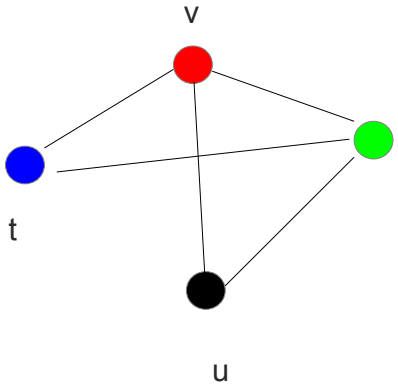
Exemple



À partir du nœud v

Triangles = 0.5

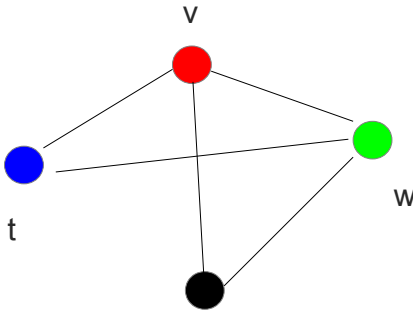
Exemple



À partir du nœud v

Triangles = 1

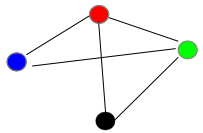
Exemple



À partir du nœud v

Triangles = 1.5

Algorithme parallèle



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

foreach $u \in \Gamma(v)$ emit {(v,u)}

Reduce 1 : Input : {(v,u) | $u \in \Gamma(v)$ }

foreach (u,w) : $u,w \in \Gamma(v)$

emit {((u,w), v)} //tous les arcs possibles dans le graphe

Exemple : ((●●)●) ((●●)●) ((●●)●),.....

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

Reduce 2 : Input : {(u,w)| v1,v2,...vk,$?}

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

Triangles[vi] +=1/2

((●●)● $) → Triangles(●)+1/2

(●●)● → ∅

Adaptation de l'algorithme

- On génère tous les chemins à vérifier en parallèle, le temps d'exécution est $max_{v \in V}(\sum d_v^2)$ => pour les nœuds avec beaucoup de voisins (millions) les reducer tasks correspondants peuvent être très lents

Amélioration :

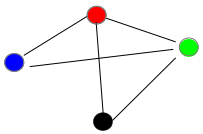
- ordonner les nœuds par leur degré (pour ceux qui ont le même degré par leur identifiant)
- compter chaque triangle une seule fois, à partir du nœud minimum
- complexité : $O(m^{3/2})$ (m = nombre d'arcs dans le graphe)

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
```


Algorithme parallèle



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

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

Exemple : (●●) (●●) (●●) (●●)(●●)(●●)

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

foreach (u,w) : $u,w \in S$

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

Exemple : ((●●)●) ((●●)●) ((●●)●))

● < ● < ● < ●

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

Reduce 2 : Input : {(u,w)| v1,v2,...vk,$?}

foreach (u,w) if $ is part of the input, then : Triangles[vi] ++

Exemple : ((●●)● $) → Triangles(●)++

(●●)● → ∅

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

Mining of Massive Datasets (Chapitre 5) : http://infolab.stanford.edu/~ullman/mmds/bookL.pdf