



GraphX

Camelia Constantin

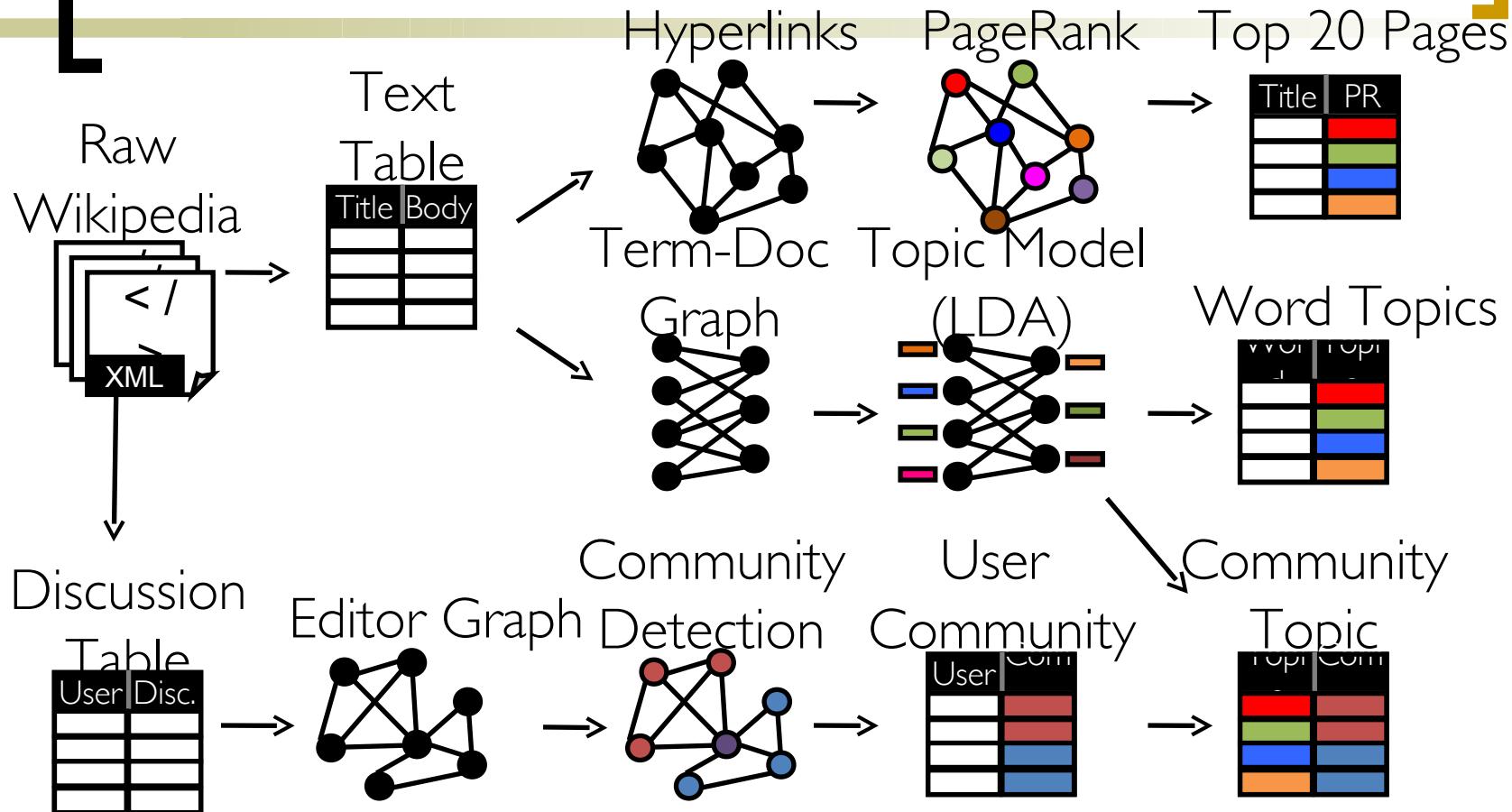
Prénom.Nom@lip6.fr

basé sur la présentation de J. Gonzalez

De nombreux algorithmes de calculs parallèles dans les graphes

- Collaborative Filtering
 - Alternating Least Squares
 - Stochastic Gradient Descent
 - Tensor Factorization
- Structured Prediction
 - Loopy Belief Propagation
 - Max-Product Linear Programs
 - Gibbs Sampling
- Semi-supervised ML
 - Graph SSL
 - CoEM
- Community Detection
 - Triangle-Counting
 - K-core Decomposition
 - K-Truss
- Graph Analytics
 - PageRank
 - Personalized PageRank
 - Shortest Path
 - Graph Coloring
- Classification
 - Neural Networks

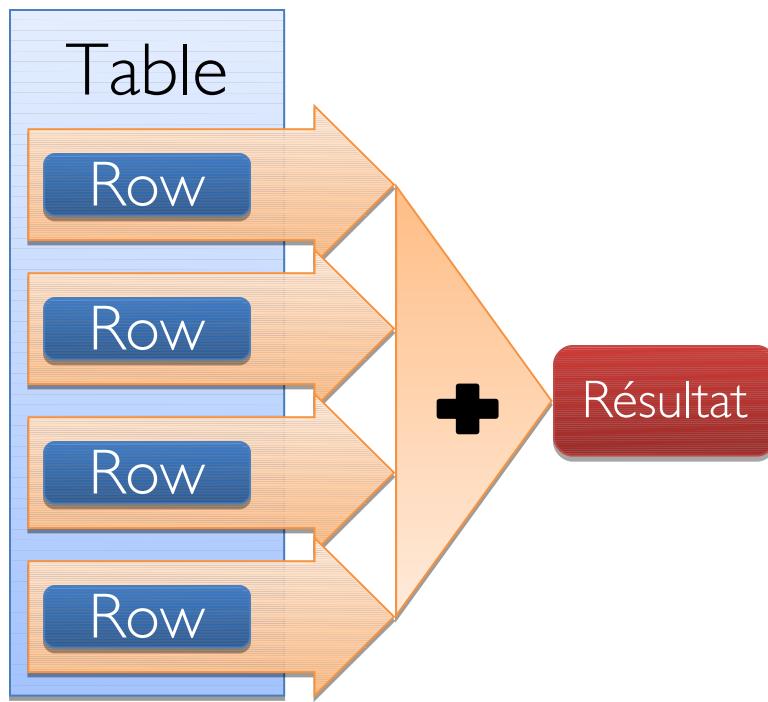
Les Graphes sont au centre de l'analyse de données Web



Les mêmes données peuvent avoir différentes “vues” table ou “vues” graphe (souvent utile de changer entre les deux vues)

Systèmes orientés graphe pour chaque vue

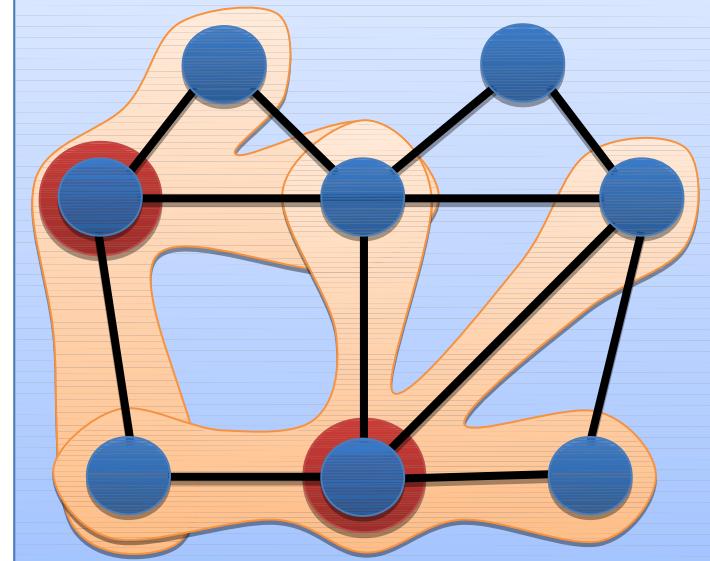
Data-parallel



Graph-Parallel



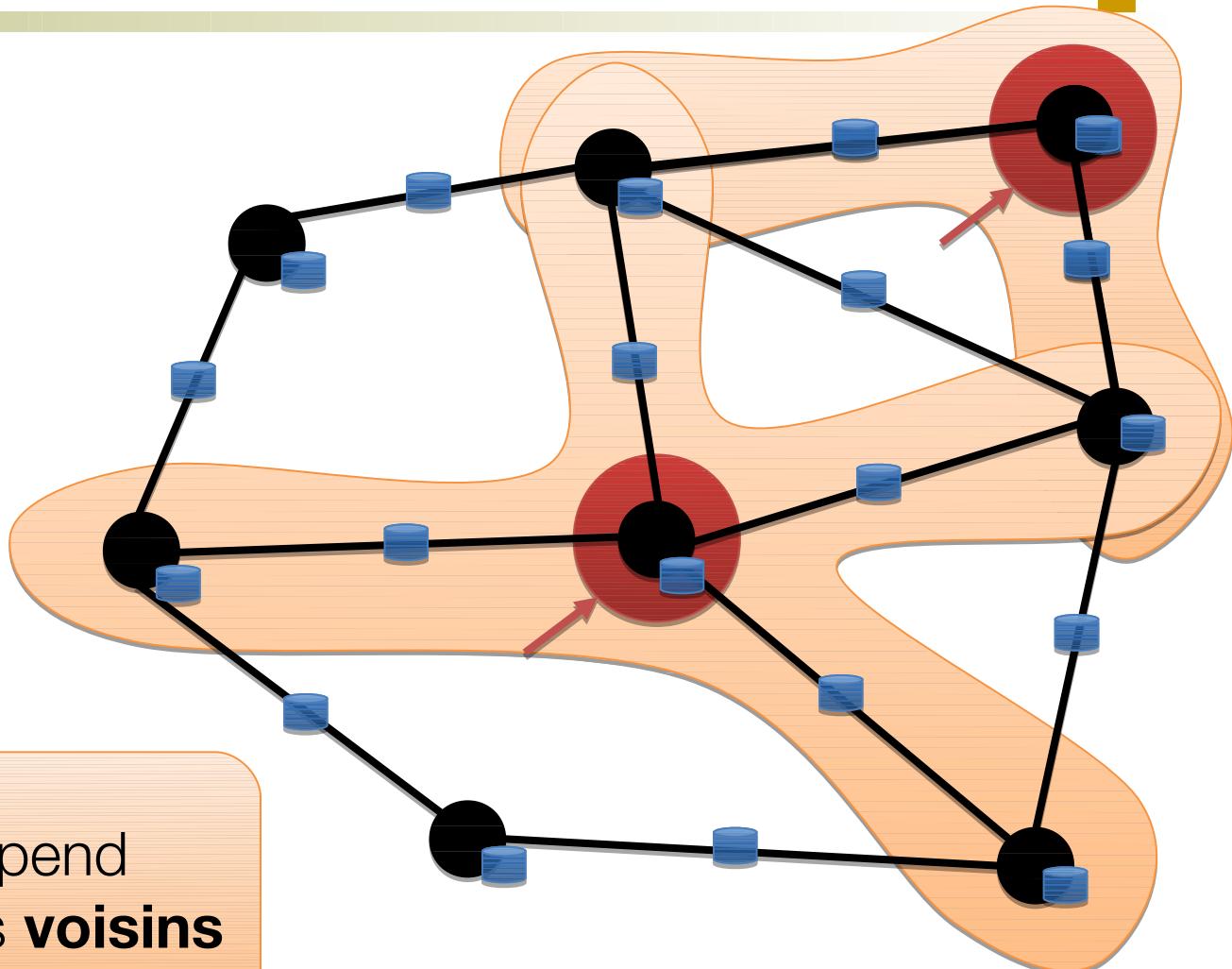
Graphe de propriétés



L'abstraction Graph-Parallel

Modèle /
Alg.
Etat

Le calcul dépend
uniquement des **voisins**

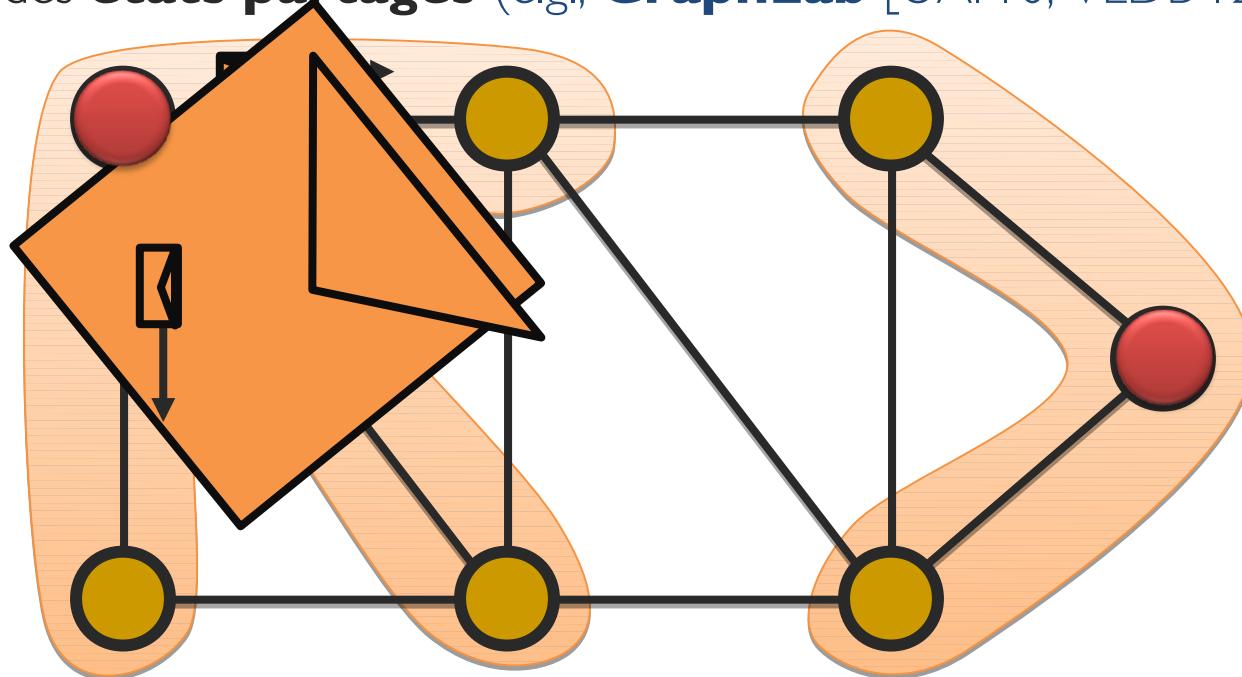


L'abstraction Graph-Parallel

Un programme défini par l'utilisateur s'exécute **sur chaque sommet**

Le graphe constraint les **interactions** le long des arêtes:

- en utilisant des **messages** (e.g. **Pregel** [PODC'09, SIGMOD'10])
- ou via des **états partagés** (e.g., **GraphLab** [UAI'10, VLDB'12])



Parallélisme: lance plusieurs programmes d'arêtes simultannément

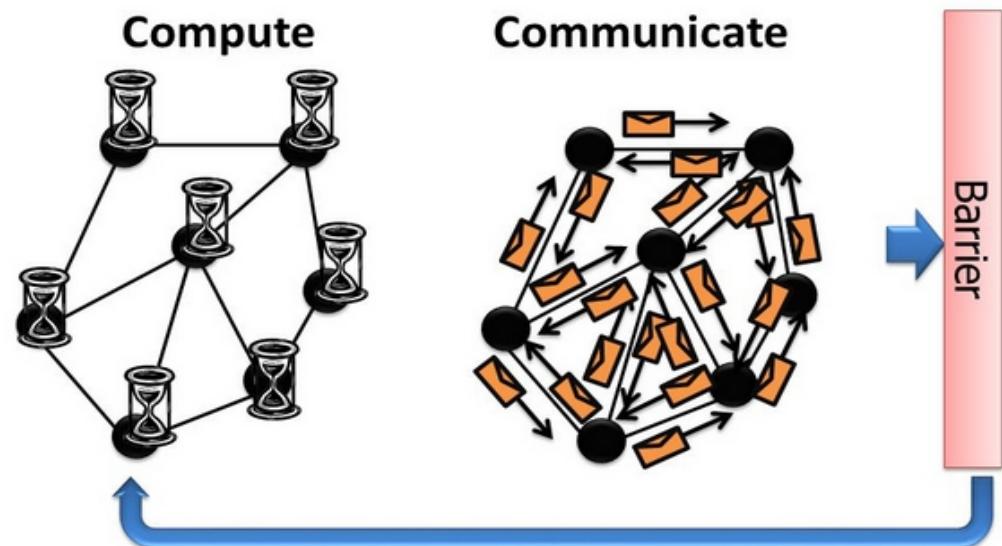
Modèle BSP

BSP: modèle de programmation data-parallel

Séquence de super-étapes:

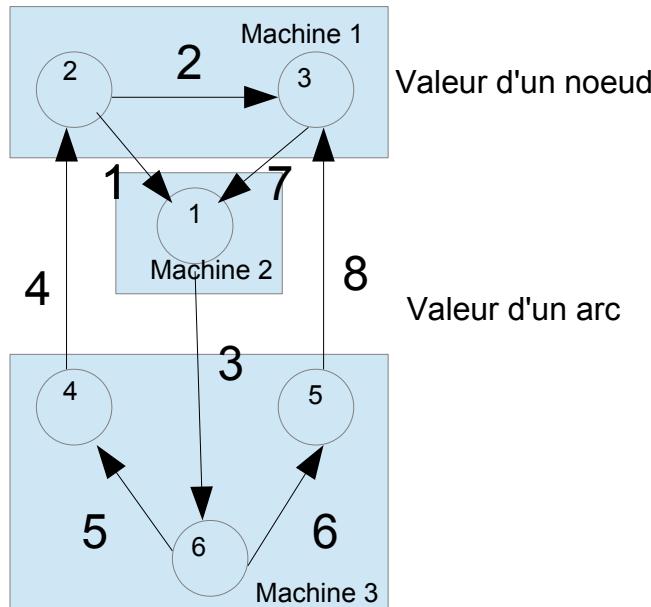
- Exécution locale :
 - Obtenir les données
 - Calculs locals asynchrones
 - Échange de données
- Barrière de synchronisation globale
- Arrêt (Halt) : aucun nœud qui calcule ou communique

- Bulk Synchronous Parallel Model:



Bulk Synchronous Parallelism (BSP)

Penser comme un nœud !



Graphe partitionné sur les machines en coupant les arcs

- arc coupé → communication entre deux machines (coût de communication)

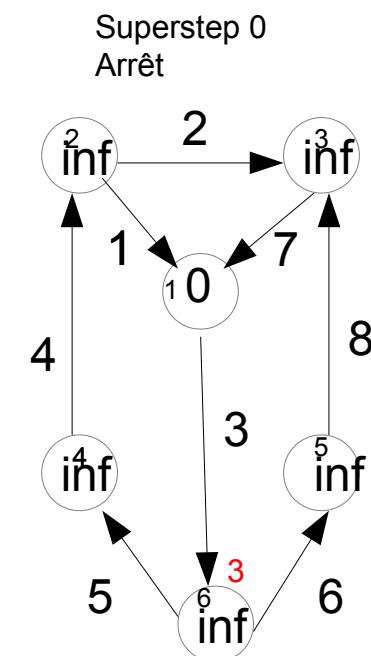
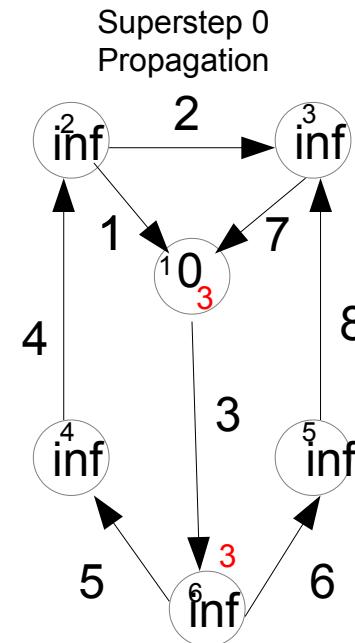
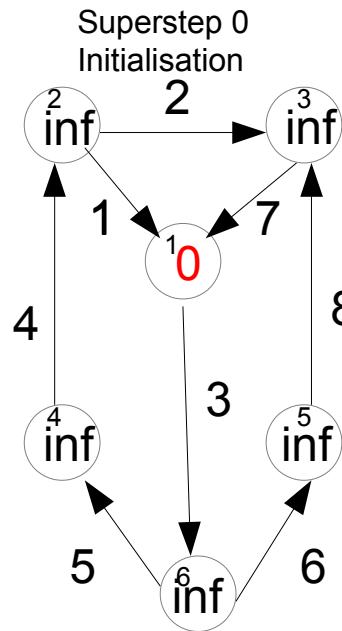
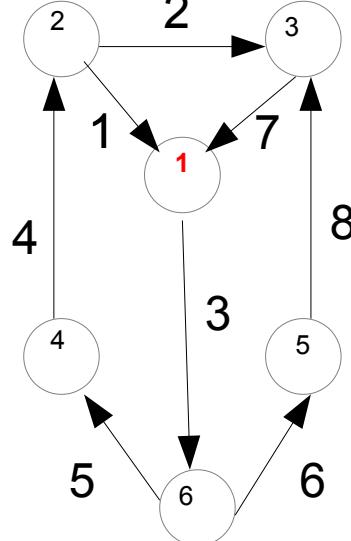
Un nœud/processus peut:

- commencer/arrêter son calcul
- ajouter/enlever un arc
- connaître son ID
- obtenir/modifier sa valeur
- obtenir/compter ses arcs
- Obtenir/modifier la valeur de ses arcs
 - Utiliser l'ID de l'arc
 - Utiliser l'ID de l'autre extrémité de l'arc

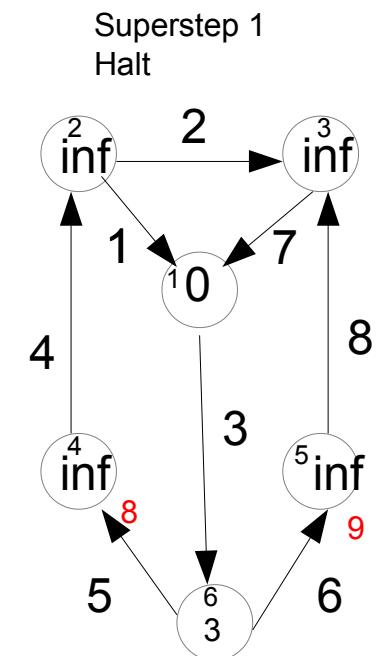
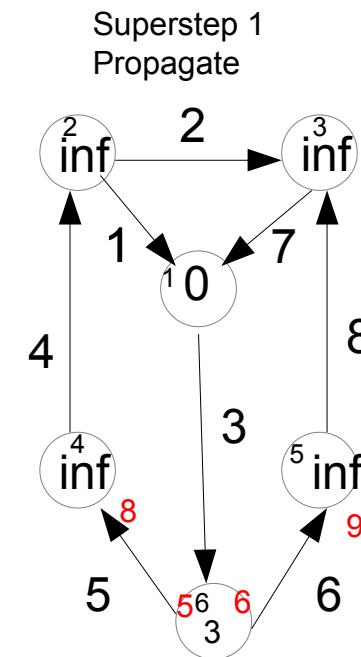
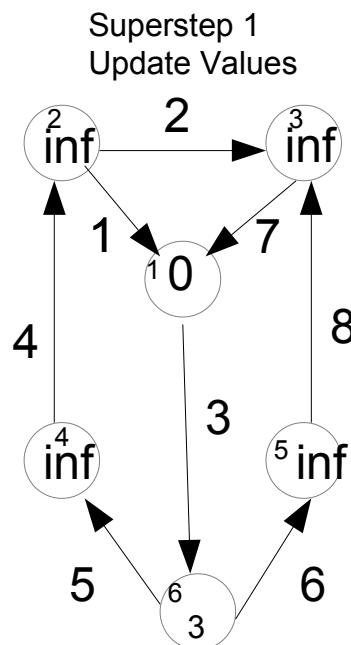
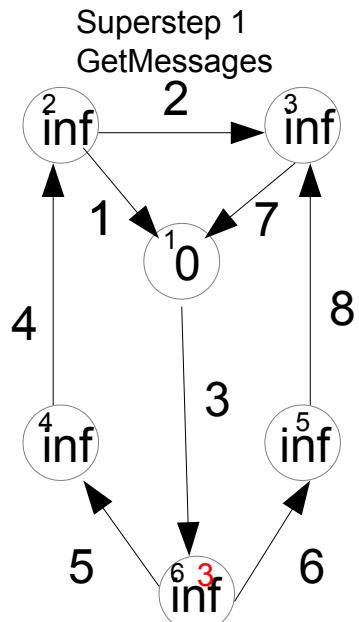
Un arc peut:

- connaître son ID
- obtenir/modifier sa valeur
- obtenir l'ID de son extrémité finale

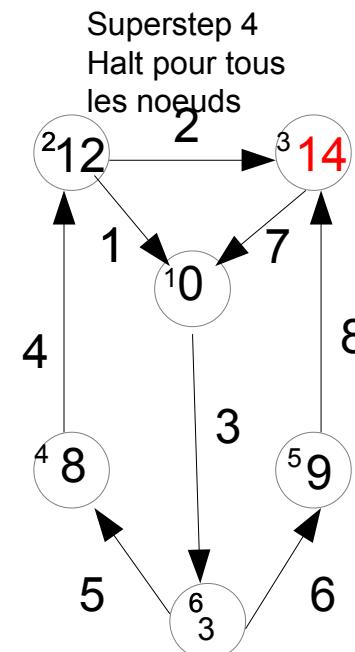
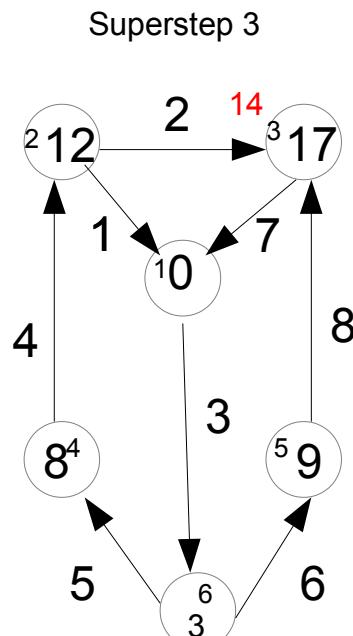
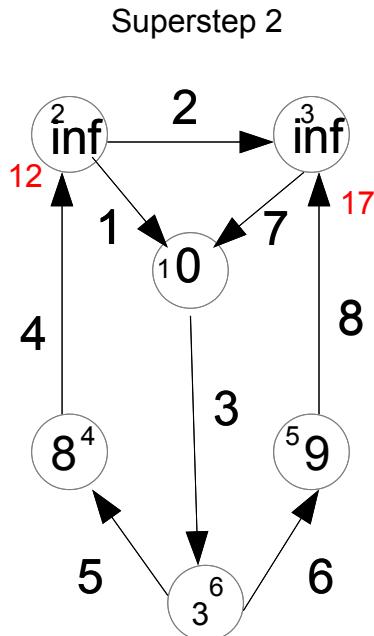
Plus court chemin (BSP)



Plus court chemin (BSP)



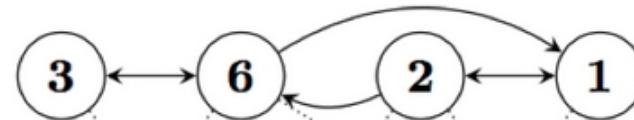
Plus court chemin (BSP)



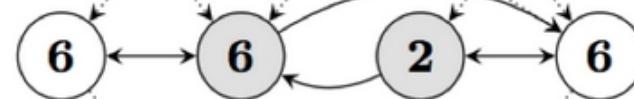
Résultat :
 $(1,0.0) (2,12.0) (3,14.0) (4,8.0) (5,9.0) (6,3.0)$

[Exemple 2]

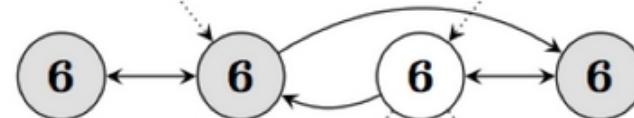
Trouver le max des id des noeuds



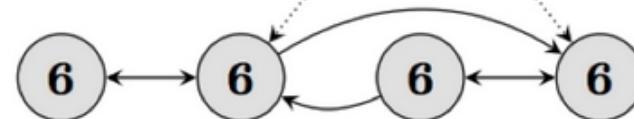
Superstep 0



Superstep 1



Superstep 2



Superstep 3

Pregel et PageRank

```
class PageRankVertex
    : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};
```

Sommet : processus qui exécute Compute() pendant chaque superstep

Superstep 0 (Initialisation):
• $1/\text{NumVertices}()$ pour chaque noeud

[Systèmes Graph-Parallel]

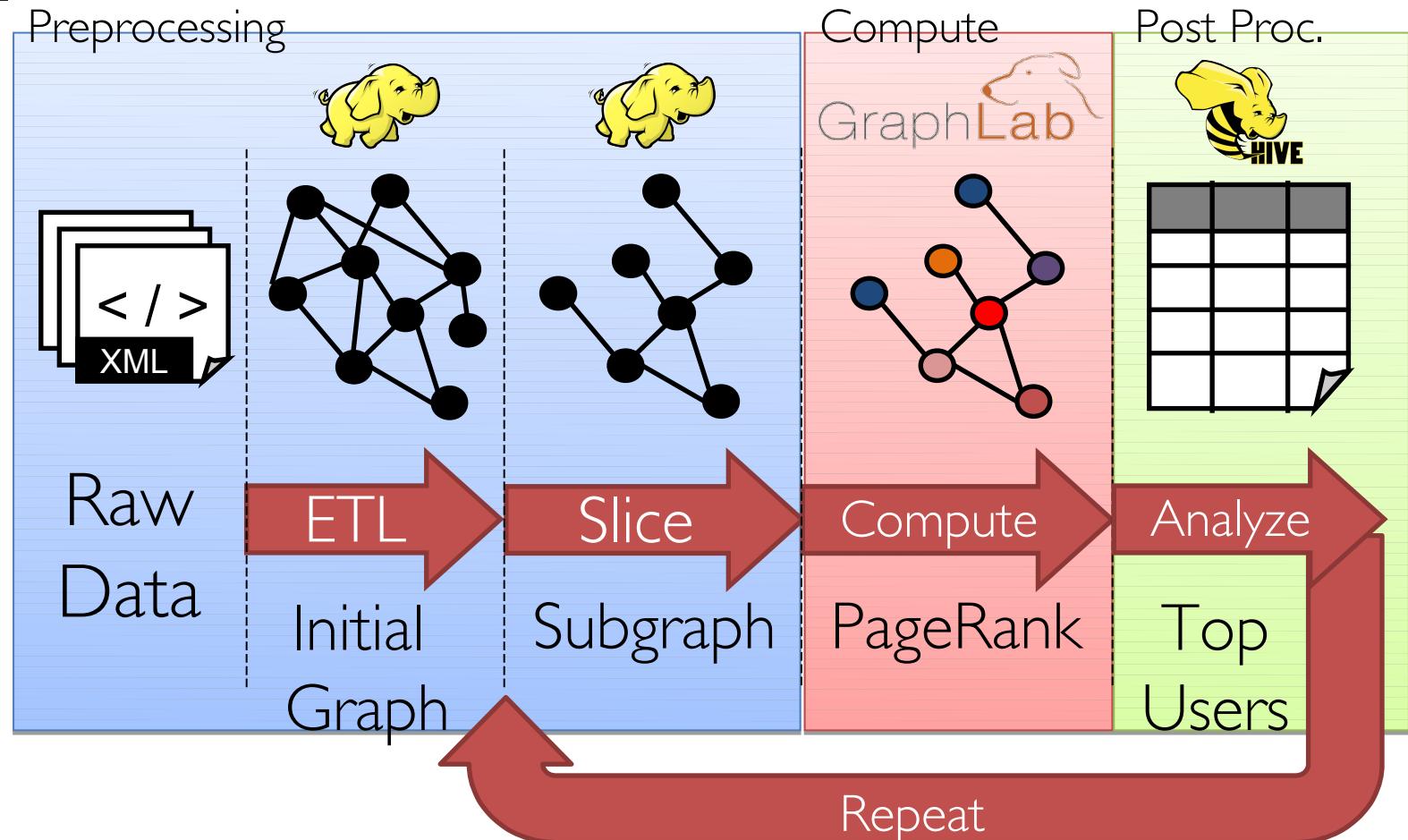
Pregel
oogle



- Proposent des [APIs spécialisées](#) pour simplifier la programmation sur des graphes.
- Nouvelles techniques de partitionnement du graphe, restriction des types d'opérations qui peuvent être utilisées.
- Exploitent la structure du graphe pour obtenir des gains en performance de plusieurs ordres de magnitude comparé aux systèmes de données parallèles (Data-Parallel) plus génériques.

Inconvénients: difficile d'exprimer les différentes étapes d'un pipeline de traitement sur des graphes (construire/modifier le graphe, calculs sur plusieurs graphes)

[Pipeline d'analyse de graphes]



Difficultés pour la programmation et l'utilisation

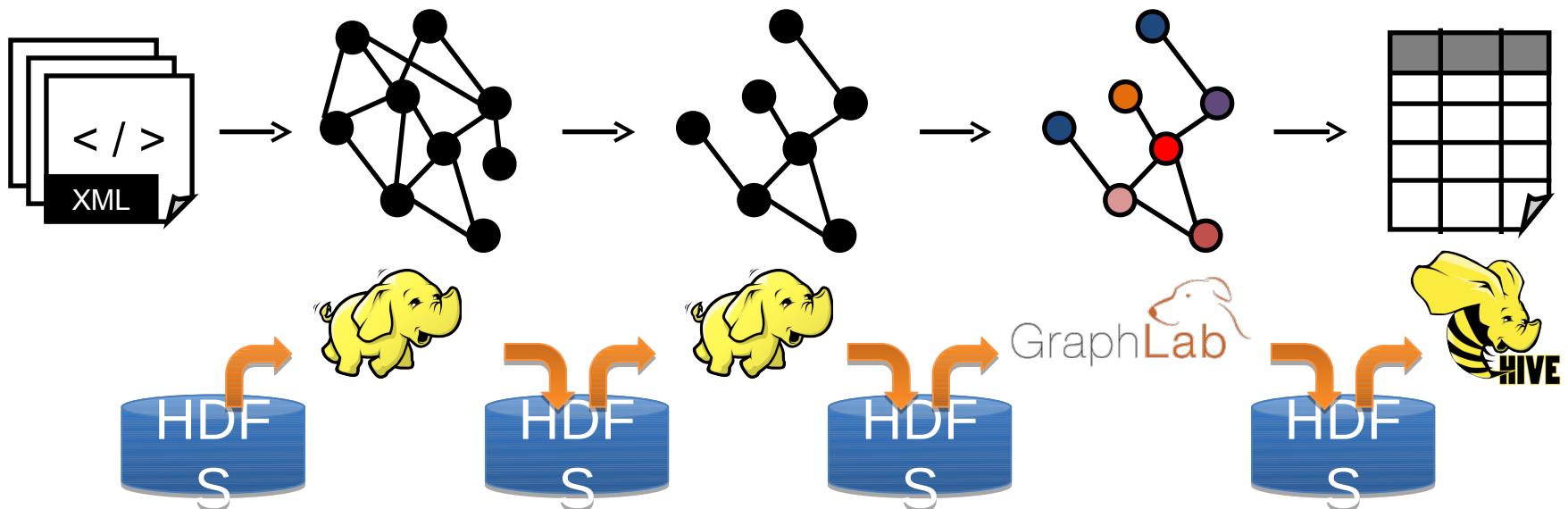
Les utilisateurs doivent **apprendre, déployer,**
et **gérer** de multiples systèmes



Conduit à des interfaces compliquées à implanter et souvent complexes à utiliser, est particulièrement inefficace

Inefficacité

D'importants déplacements de données et de duplications à travers le réseau et le système de fichiers

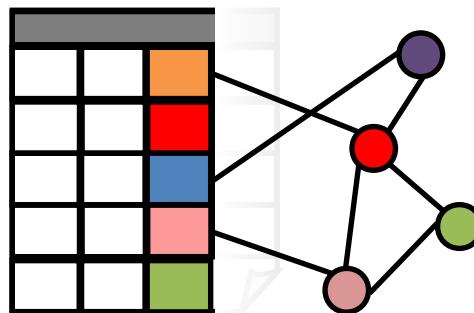


Ré-utilisation limitée de structures de données internes d'une étape à l'autre

L'approche unifiée GraphX

Nouvelle API

Atténue la distinction entre les Tables et les Graphes



New System

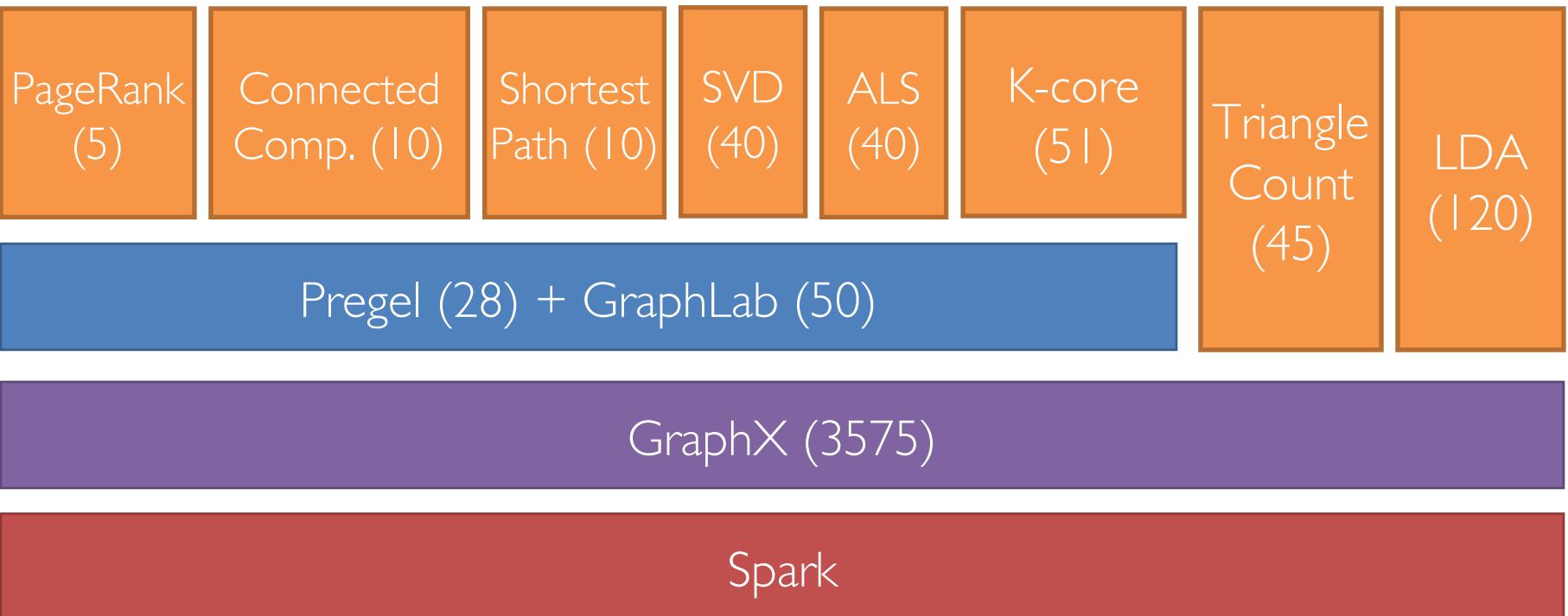
Combine les systèmes Data-Parallel et Graph-Parallel



Permet aux utilisateurs:

- d'exprimer facilement et efficacement le pipeline entier de l'analyse de graphe.
- de voir les données à la fois comme collections (RDD) et comme graphe sans déplacement/duplication

[GraphX (Lignes de Code)]

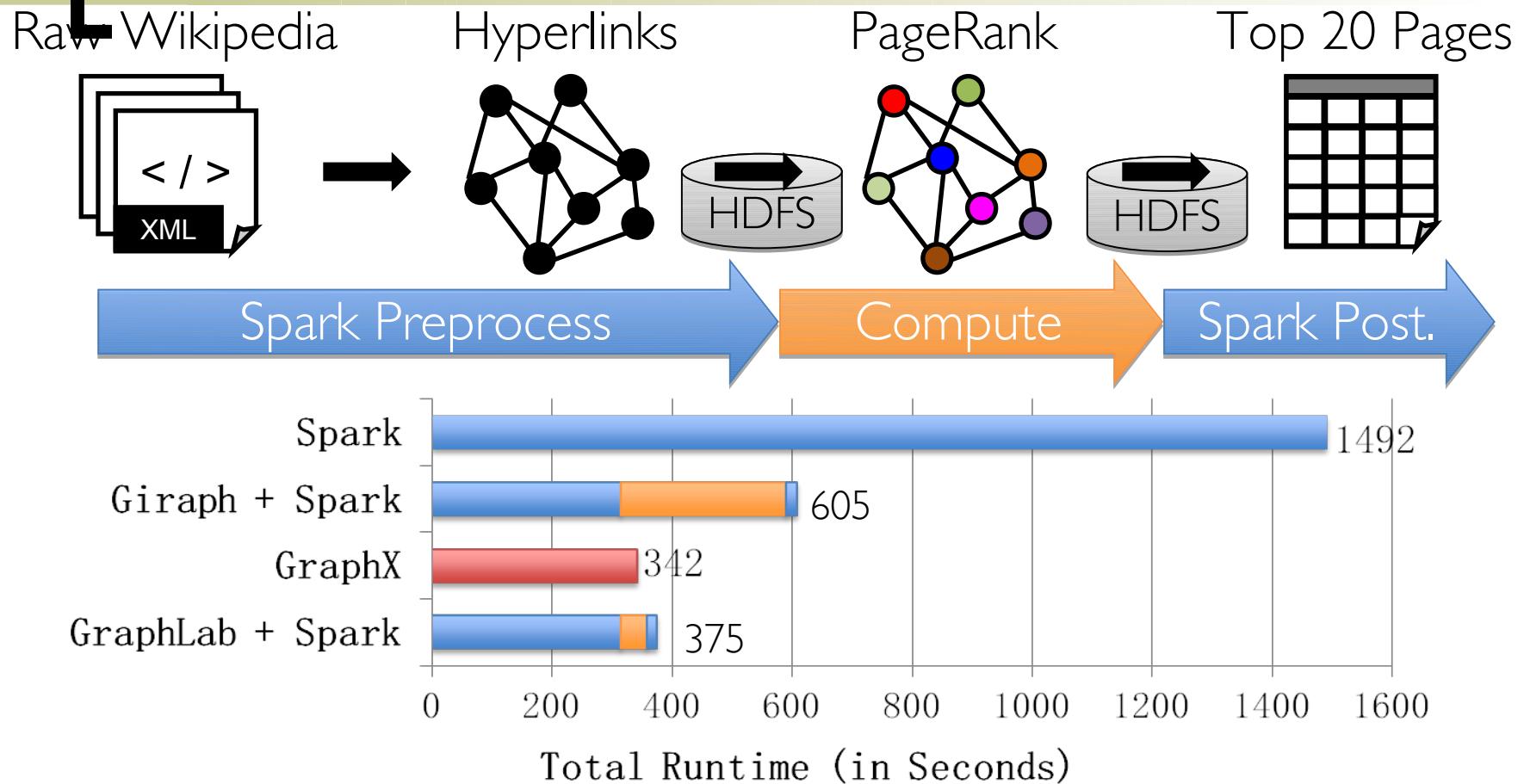


GraphX: avantages

Les abstractions Pregel et GraphLab peuvent être réalisées avec les opérateurs GraphX en moins de 50 lignes de code!

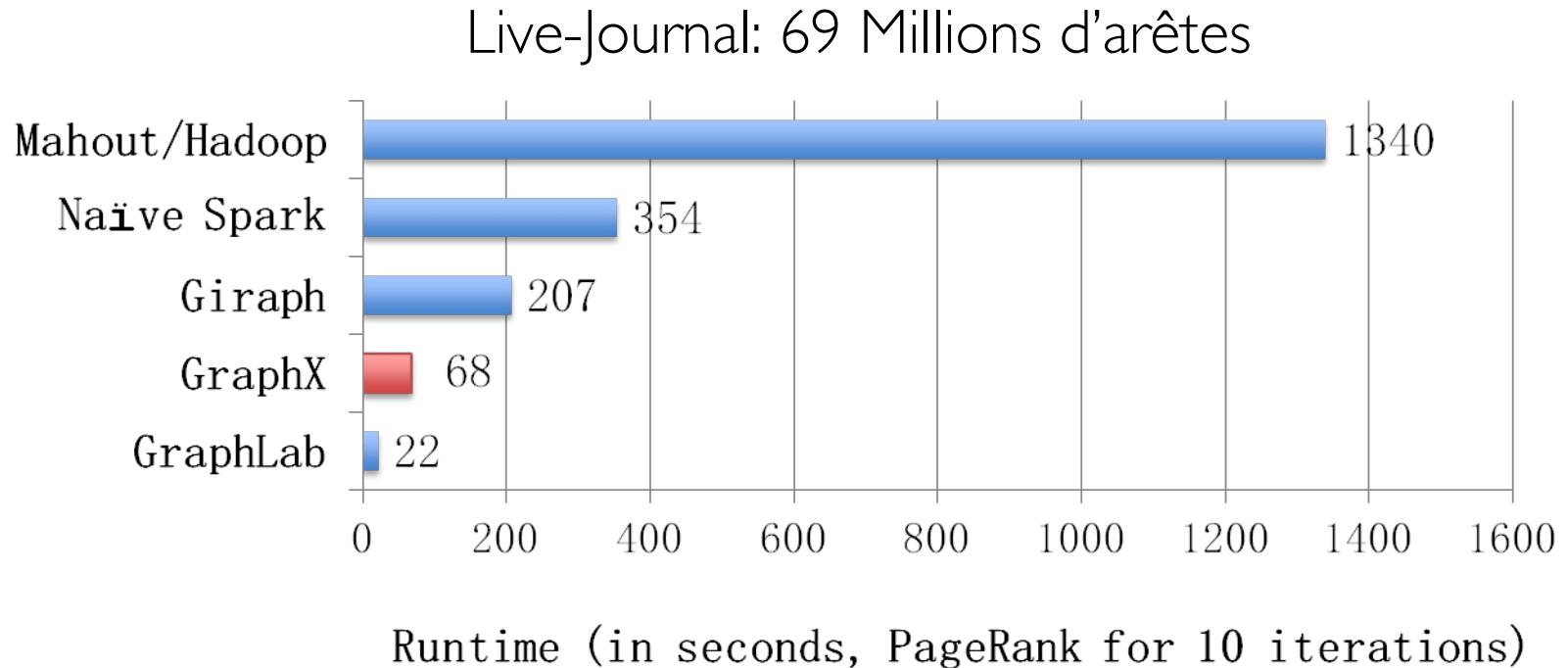
En composant ces opérateurs on peut construire les pipelines entiers d'analyses de graphe.

Un exemple de pipeline



Le temps de traitement de GraphX pour tout le pipeline est plus [rapide](#) que Spark+GraphLab

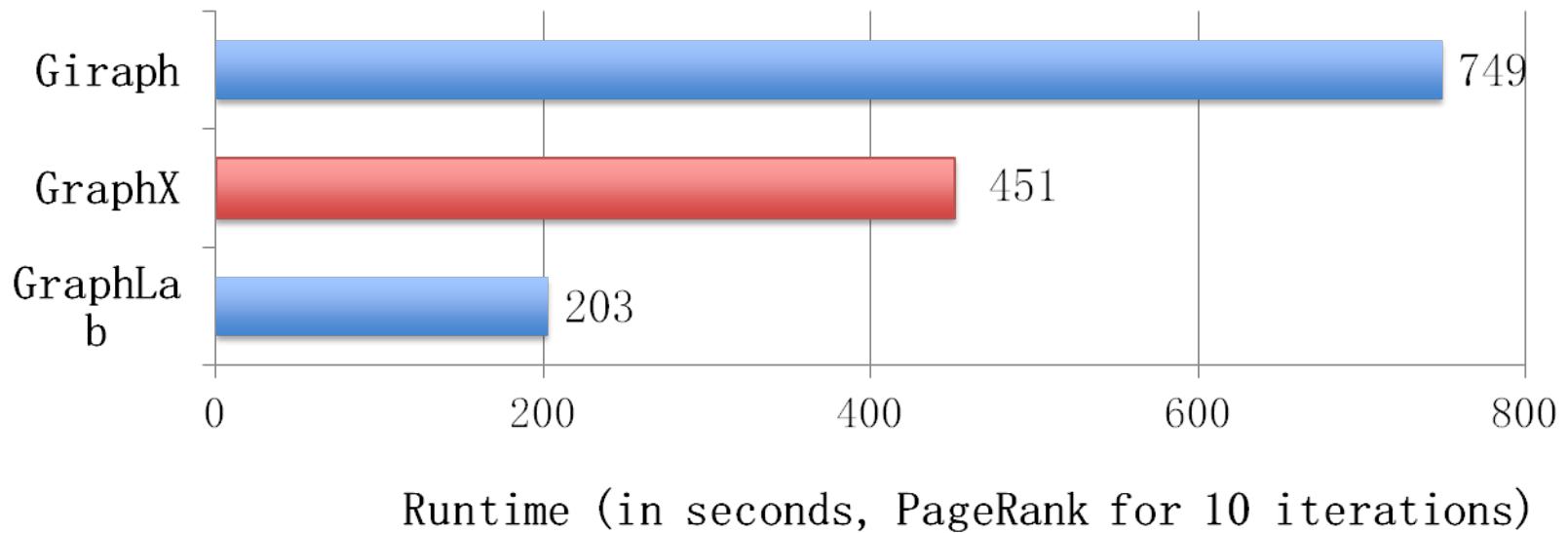
Comparaisons des performances



GraphX est environ 3x plus lent que GraphLab

GraphX passe à l'échelle sur de plus grands graphes

Graphe Twitter: 1.5 milliard d'arêtes

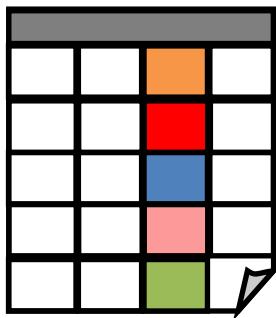


GraphX est environ 2x plus lent que GraphLab

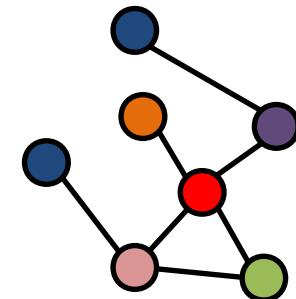
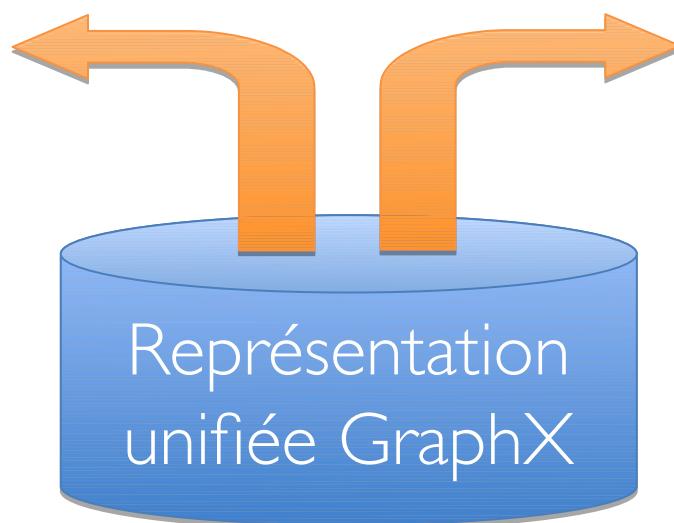
- » Surcoûts liés à Scala + Java: Lambdas, GC time, ...
- » Pas de parallélisation de la mémoire partagée: 2x plus de comm.

Différentes vues

Les Tables et les Graphes sont des *vues composites* des mêmes données *physiques*



Vue Table



Vue Graphe

Chaque vue a ses propres *opérateurs* qui exploitent la *sémantique* de la vue pour obtenir une *exécution efficace*

Voir un Graphe comme une Table(RDD)

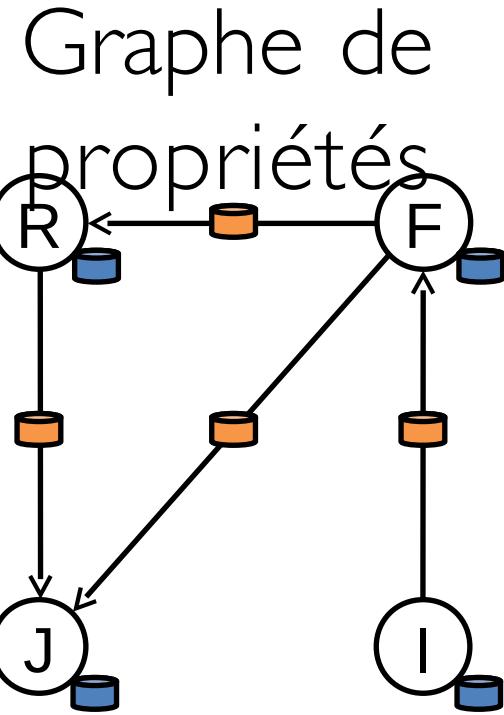


Table des sommets

<i>Id</i>	<i>Propriété (V)</i>
<i>Rxin</i>	<i>(Stu., Berk.)</i>
<i>Jegonzal</i>	<i>(PstDoc, Berk.)</i>
<i>Franklin</i>	<i>(Prof., Berk)</i>
<i>Istoica</i>	<i>(Prof., Berk)</i>

Table des arêtes

<i>Src_Id</i>	<i>Dst_Id</i>	<i>Propriété (E)</i>
<i>rxin</i>	<i>jegonzal</i>	<i>Friend</i>
<i>franklin</i>	<i>rxin</i>	<i>Advisor</i>
<i>istoica</i>	<i>franklin</i>	<i>Coworker</i>
<i>franklin</i>	<i>jegonzal</i>	<i>PI</i>

Gestion du graphe

- Comme les RDD, les graphes de propriétés sont:
- **Non-modifiable**: les changements de valeurs ou de la structure du graphe se font en produisant un nouveau graphe
- **Distribués**: le graphe est partitionné en utilisant un ensemble d'heuristiques pour le partitionnement des noeuds
- **Résistant aux pannes**: comme avec RDD, chaque partition sur le graphe peut être recréé sur une autre machine pour la tolérance aux pannes

[Le graphe de propriétés]

- Multigraphe dirigé avec des objets définis par l'utilisateur attachés à chaque arête et à chaque sommet
- Multi-graphe signifie qu'il peut y avoir plusieurs arêtes partageant la même source et destination (dans le cas où on veut modéliser plusieurs relations entre nœuds)
- Chaque sommet a une clé unique VertexID de 64 bits (**VertexId**)
- Chaque arête a l'ID du sommet source et l'ID du sommet destination

[Le graphe de propriétés]

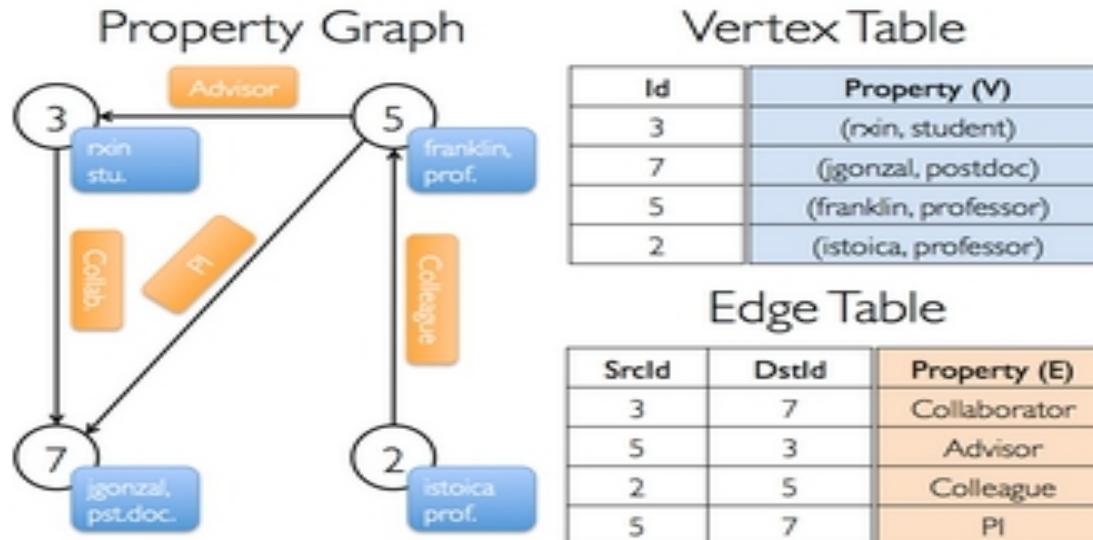
Le graphe de propriétés correspond à deux RDD :

- VertexRDD [VD] , version optimisée de RDD [(VertexID, VD)] , pour les sommets
- EdgeRDD [ED] versions optimisée de RDD [Edge [ED]] , pour les arcs
- VD et ED : types des objets associés aux sommets/arcs
- VertexRDD [VD] et EdgeRDD [ED] fournissent des fonctionnalités supplémentaires pour les calculs de graphe

Exemple de graphe de propriétés

Exemple du graphe de propriétés des collaborateurs du projet GraphX:

- Les sommets contiennent le nom et la fonction
- Les arcs annotés avec la nature de la collaboration



Graphe obtenu:

```
val userGraph: Graph[(String, String), String]
```

Construction du graphe

```
type VertexId = Long
```

```
//Créer une RDD pour les sommets
```

```
val vertices :RDD[(VertexId,(String, String))] = sc.parallelize(Array((3L,("rxin","student")), (7L,("jgonzal","postdoc")), (5L,("franklin","prof")), (2L,("istoica", "prof"))))
```

```
class Edge[ED] (val srcId : VertexId, val dstId : VertexId, val attr : ED)
```

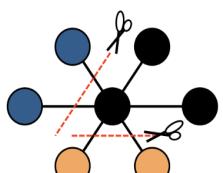
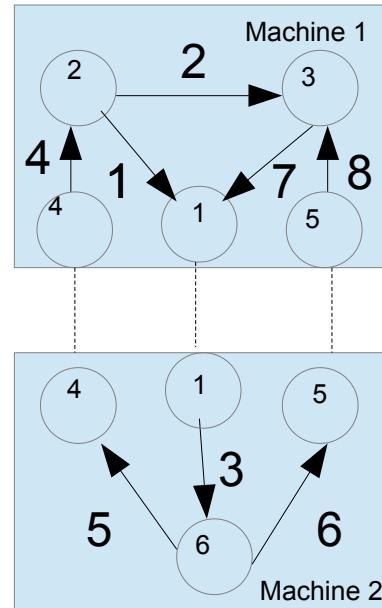
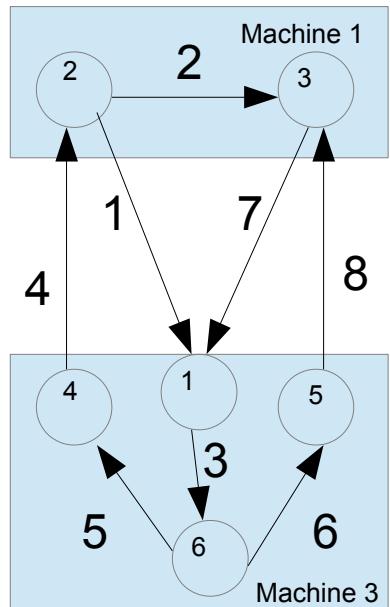
```
//Créer une RDD pour les arcs
```

```
val edges :RDD[Edge[String]] = sc.parallelize(Array(Edge(3L,7L, "collab"), Edge(5L, 3L, "advisor" ), Edge(2L,5L, "colleague"), Edge(5L,7L, "pi"))))
```

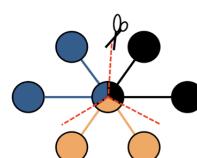
```
val graph = Graph(vertices, edges)
```

Le graphe obtenu a la signature : *Graph[(String, String), String]*

Partitionnement de graphe

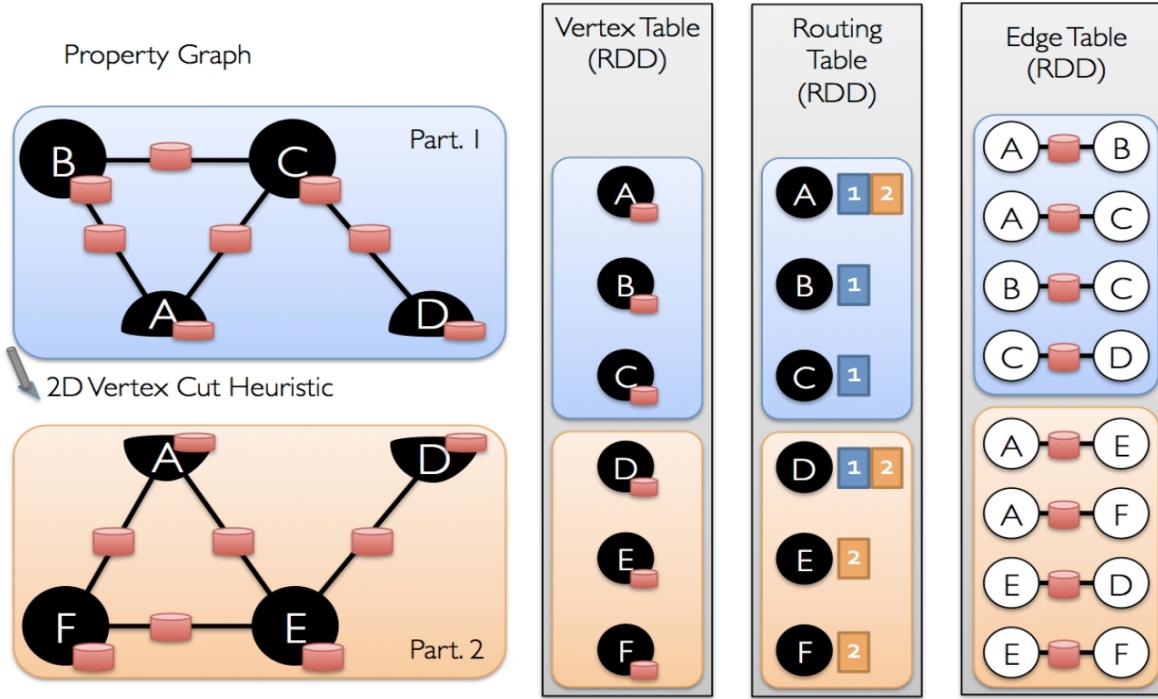


Edge Cut



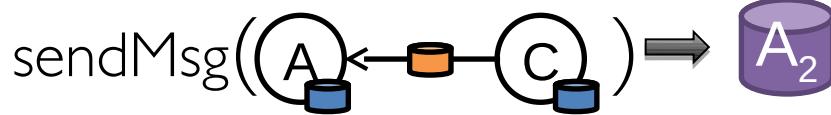
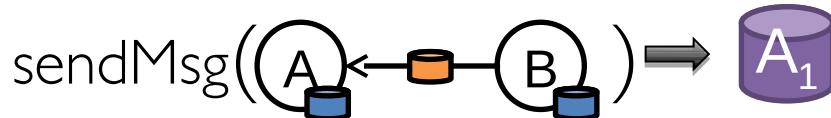
Vertex Cut

GraphX : Partitionnement de graphe



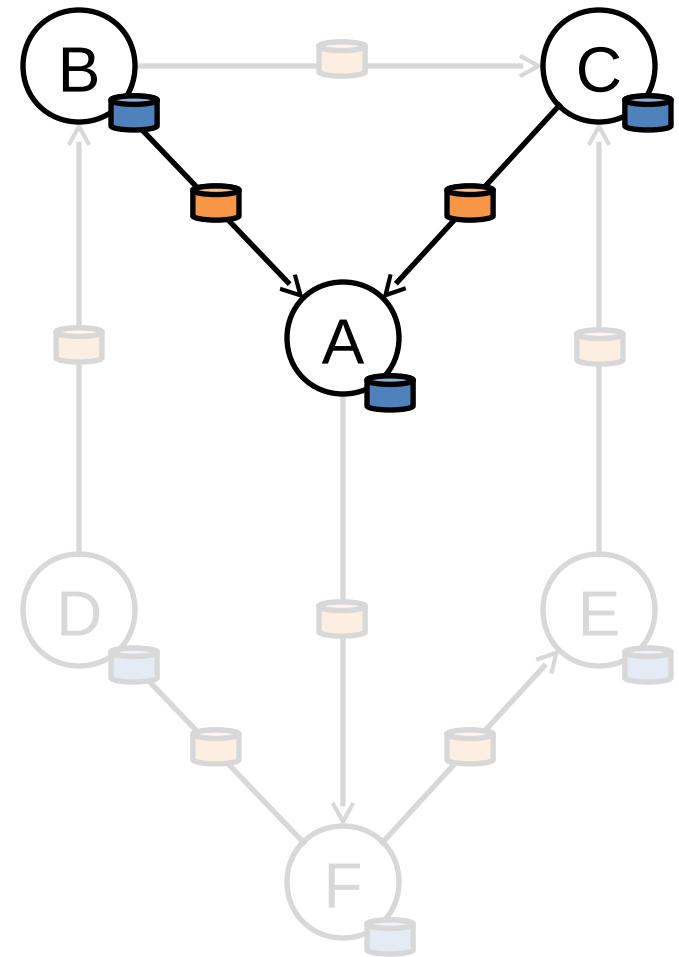
Exemple de stratégie d'exécution: aggregateMessages

Map-Reduce pour chaque sommet



Plan d'exécution :

- Déplacer les attributs des nœuds source et destination au même endroit que les partitions des arrêtes correspondantes (utilise les tables de routage)
- Générer les triplets (source arrête destination) correspondants
- Appliquer sendMsg et mergeMsg



[Opérateurs RDD]

Les opérateurs de table (RDD) sont hérités de Spark:

map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
leftOuterJoin	cross	save
rightOuterJoin	zip	...

Opérateurs de Graphe

```
class Graph [ V, E ] {  
    def Graph(vertices: Table[ (Id, V) ],  
              edges: Table[ (Id, Id, E) ])  
        // Vues -----  
        val vertices: VertexRDD[VD]  
        val edges: EdgeRDD[ED]  
        val triplets: RDD[EdgeTriplet[VD, ED]]  
        // Transformations -----  
        def reverse: Graph[V, E]  
        def subgraph(pV: (Id, V) => Boolean,  
                    pE: Edge[V, E] => Boolean): Graph[V, E]  
        def mapV(m: (Id, V) => T ): Graph[T, E]  
        def mapE(m: Edge[V, E] => T ): Graph[V, T]  
        // Joins -----  
        def joinV(tbl: Table [(Id, T)]): Graph[(V, T), E]  
        def joinE(tbl: Table [(Id, Id, T)]): Graph[V, (E, T)]  
        // Computation -----  
        def aggregateMessages(sendMsg: EdgeContext[VD, ED, Msg] => Unit,  
                             mergeMsg: (Msg, Msg) => Msg,  
                             tripletFields: TripletFields = TripletFields.All): VertexRDD[Msg]  
    }  
}
```

Utilisation des vues

Decomposition du graphe en vue sommets ou vue arêtes avec
`graph.vertices` et `graph.edges`

```
val graph: Graph[(String, String), String] // Constructed from above
// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count
// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

Résultats : 1, 1

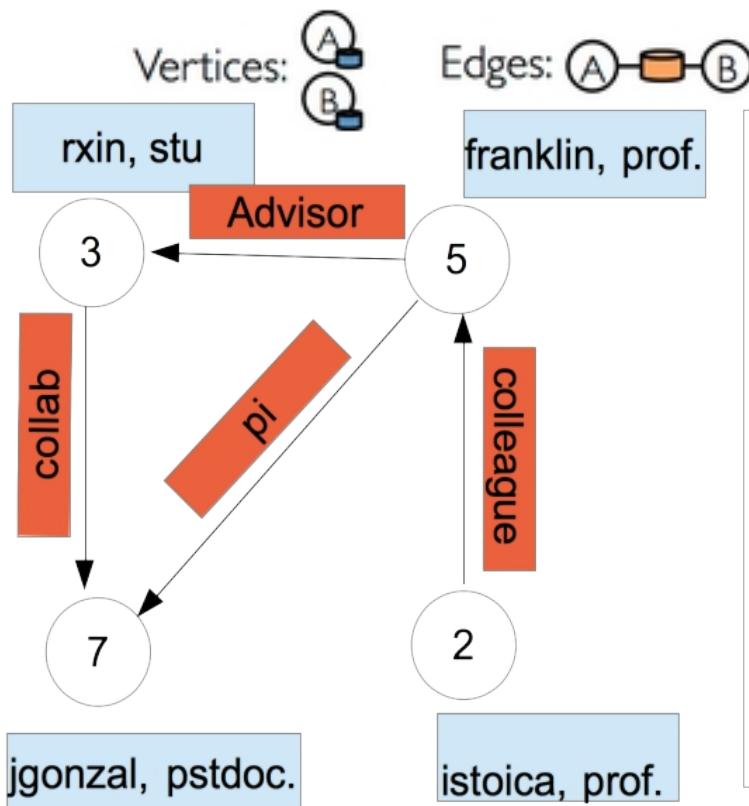
`graph.vertices` retourne un `VertexRDD[(String, String)]` qui étend `RDD[(VertexID, (String, String))]` => on peut utiliser le `case` pour décomposer le tuple (idem pour les arêtes)

`graph.edges` retourne `EdgeRDD[String]` (`RDD[Edge[String]]`) =>
on pouvait également utiliser `case` comme suit:

```
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

Vue triplets

- En plus de la vue sommets et la vue arêtes, il existe une vue *triplets* RDD [EdgeTriplet [VD, ED]]
- La classe EdgeTriplet étend la classe Edge en ajoutant **srcAttr** et **dstAttr** contenant les propriétés des noeuds source/destination



srcAttr	dstAttr	attr
(rxin, student)	(jgonzal, pdoc)	collab.
(franklin, prof)	(jgonzal, pdoc)	pi
(istoica, prof.)	(franklin, prof)	colleague
(franklin, prof)	(rxin, student)	advisor

RDD

Vue triplets

Exemple : afficher les relations entre les utilisateurs :

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

Résultat :

*rxin is the collab of jgonzal
franklin is the advisor of rxin
istoica is the colleague of franklin
franklin is the pi of jgonzal*

Opérateurs d'information

```
// Information about the Graph =====  
=====  
val numEdges: Long  
val numVertices: Long  
val inDegrees: VertexRDD[Int]  
val outDegrees: VertexRDD[Int]  
val degrees: VertexRDD[Int]  
// Views of the graph as collections =====  
=====  
val vertices: VertexRDD[VD]  
val edges: EdgeRDD[ED]  
val triplets: RDD[EdgeTriplet[VD, ED]]
```

[Exemple(1)]

- Calculer le degré entrant de chaque sommet :

```
val graph: Graph[(String, String), String]
// Utiliser l'opérateur GraphOps.inDegrees
val inDegrees: VertexRDD[Int] = graph.inDegrees
inDegrees.collect.foreach(println(_))
```

- Résultat :

(3,1)
(5,1)
(7,2)

[Exemple(2)]

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
    if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

■ Calculer le degré entrant/sortant/total maximum de chaque sommet :

Résultat:

maxDegrees : (5,3)

maxInDegree : (7,2)

maxOutDegree : (5,2)

Opérateurs de transformation(1)

```
// Change the partitioning heuristic -----
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]

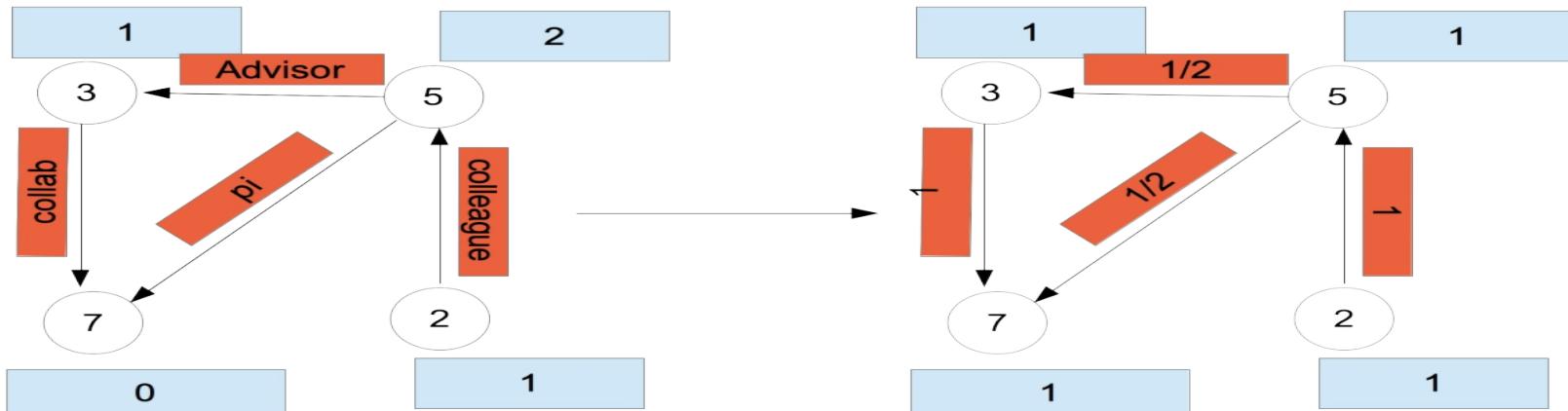
// Transform vertex and edge attributes -----
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
  : Graph[VD, ED2]
```

- mapXX : produit un nouveau graphe de même structure avec XX modifié par la fonction de l'utilisateur map
- La structure n'est pas affectée (le graphe résultat réutilise les index structurels du graphe d'origine)
- Ces opérateurs sont souvent utilisés pour initialiser le graphe pour un calcul ou enlever des propriétés inutiles

[Exemple : mapTriplets]

Créer un nouveau graphe où chaque arc est étiqueté avec un poids

```
val outputGraph: Graph[Double, Double] =  
graph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) => 1.0)
```



Opérateurs sur structure

```
// Modify the graph structure =====

def reverse: Graph[VD, ED]
def subgraph(
    epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
    vpred: (VertexID, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
```

[Opérateurs sur structure (2)]

- `reverse`: retourne un nouveau graphe en inversant la direction des arêtes
- `subgraph`: prend des prédictats sur sommets et arêtes et retourne le graphe contenant les sommets satisfaisants les prédictats et reliés par les arêtes satisfaisant les prédictats (si l'on veut restreindre à certains nœuds/arcs)
- `mask`: retourne un sous-graphe correspondant à l'intersection d'un graphe donné et d'un graphe-masque
- `groupEdges`: pour un multi-graphe, fusionne les différentes arêtes entre 2 sommets en une seule

Exemple : subgraph (1)

```
//RDD pour les sommets
```

```
val users: RDD[(VertexId, (String, String))] =  
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")), (5L, ("franklin",  
"prof")), (2L, ("istoica", "prof")), (4L, ("peter", "student"))))
```

```
//RDD pour les arcs
```

```
val relationships: RDD[Edge[String]] = sc.parallelize(Array(Edge(3L, 7L, "collab"),  
  Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"),  
Edge(4L, 0L, "student"), Edge(5L, 0L, "colleague")))
```

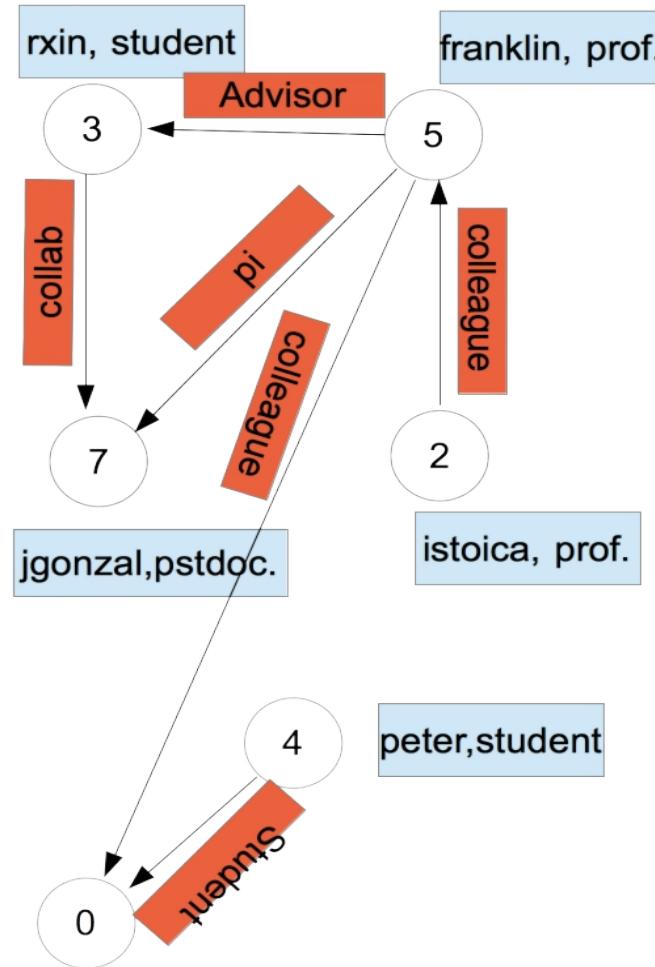
```
// Définir un utilisateur par défaut pour les arcs dont un des sommets n'est pas défini
```

```
val defaultUser = ("John Doe", "Missing")
```

```
// Construire le graphe
```

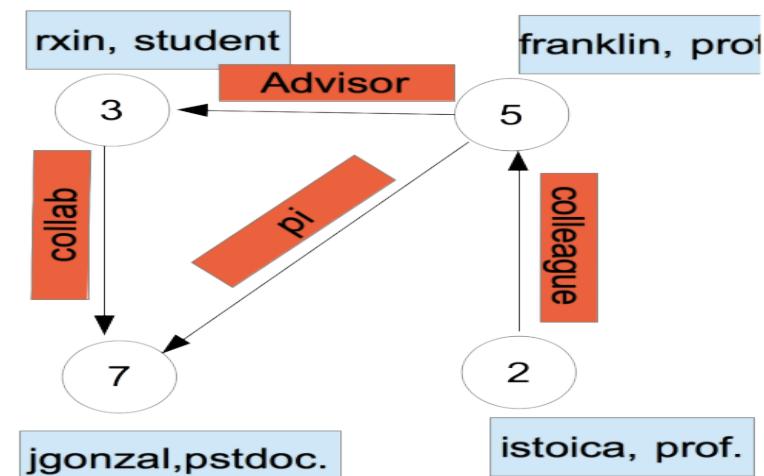
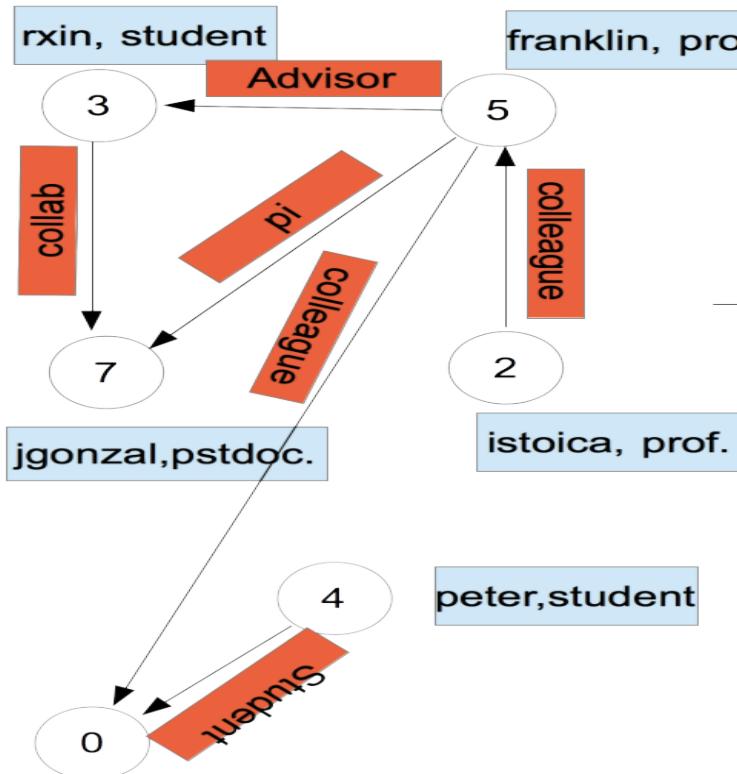
```
val graph = Graph(users, relationships, defaultUser)
```

[Exemple : subgraph (2)]



[Exemple : subgraph (3)]

```
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
```



Exemple : subgraph (4)

```
valGraph.vertices.collect.foreach(println(_))
```

Résultat :

```
(2,(istoica,prof))
(3,(rxin,student))
(4,(peter,student))
(5,(franklin,prof))
(7,(jgonzal,postdoc))
```

```
validGraph.triplets.map( triplet => triplet.srcAttr._1 + " is the " + triplet.attr + "
of " + triplet.dstAttr._1).collect.foreach(println(_))
```

Résultat

rxin is the collab of jgonzal

franklin is the advisor of rxin

istoica is the colleague of franklin

franklin is the pi of jgonzal

[Exemple : mask et subgraph]

Calculer les composantes connexes en utilisant tous les arrêtes, y compris celles passant par des nœuds « inconnus » mais ne pas garder ces nœuds dans le résultat.

```
// Run Connected Components
val ccGraph = graph.connectedComponents() // No longer contains missing field
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

Opérateurs de jointure

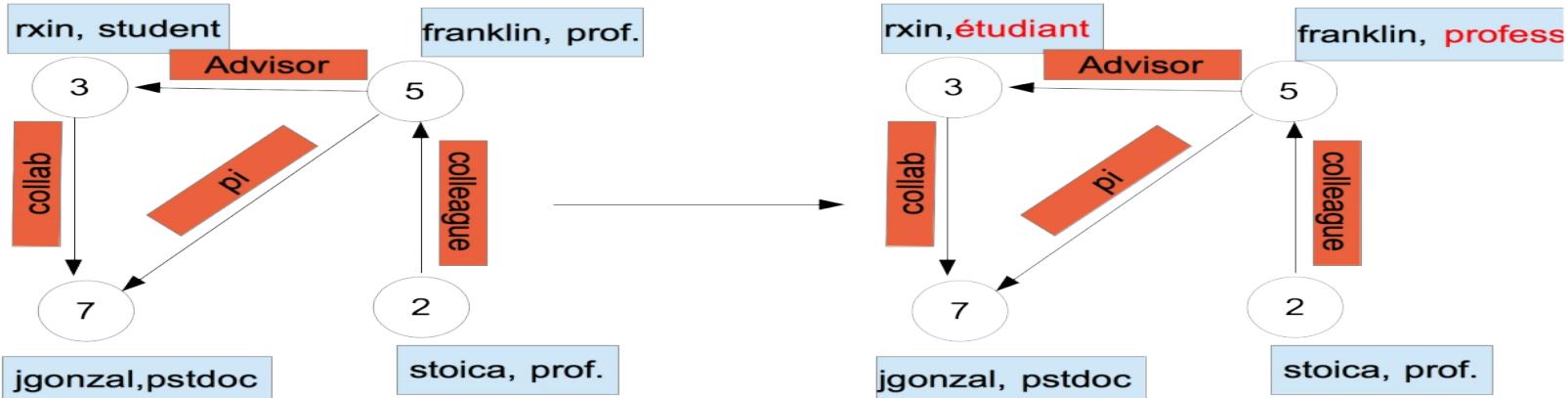
```
// Join RDDs with the graph =====  
  
def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph[VD, ED]  
def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])  
  (mapFunc: (VertexID, VD, Option[U]) => VD2)  
  : Graph[VD2, ED]
```

Souvent il est nécessaire de joindre des données de collections externes (RDD) avec des graphes

Ex.: on souhaite ajouter d'autres propriétés à un graphe existant, ou copier des propriétés d'un graphe à l'autre

`joinVertices`: joint les sommets avec le RDD en entrée et retourne un nouveau graphe avec les propriétés obtenues en appliquant la fonction map aux sommets joignant

Exemple : joinVertices



```
val joinRDD:RDD[(VertexId, String)] = sc.parallelize(Array((3L, "étudiant"), (5L, "professeur")))
val joinedGraph = graph.joinVertices(joinRDD)((id, oldVal, newVal) => (oldVal._1, newVal))
joinedGraph.vertices.collect.foreach(println(_))
```

Résultat :

(2,(istoica, prof)) (3,(rxin,étudiant)) (5,(franklin,professeur)) (7,(jgonzal, pstdoc))

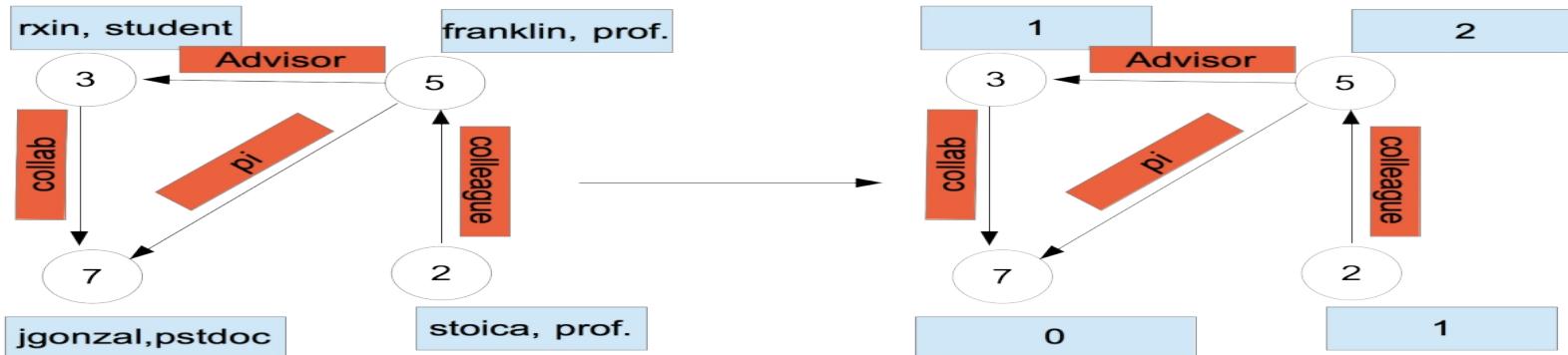
- les sommets qui n'ont pas de correspondants dans la RDD gardent leur valeur initiale.
- `joinVertices` ne peut pas changer le type des propriétés des sommets (ex. : toujours (String, String))

Exemple : outerJoinVertices

Créer un nouveau graphe, à chaque sommet on associe son degré sortant :

```
val degreesRDD : VertexRDD[Int] = graph.outDegrees
```

```
val joinedGraph: Graph[Int, String] = graph.outerJoinVertices(degreesRDD)((vid, _, degOpt)  
=> degOpt.getOrElse(0))
```



```
joinedGraph.vertices.collect.foreach(println(_))
```

Résultat :

(2,1) (3,1) (5,2) (7,0)

- La fonction `map` est appliquée à tous les sommets, elle prend comme paramètre un type **Option**.
- `outerjoinVertices` peut changer le type des propriétés des sommets (ex. : Int au lieu de (String, String))

Opérateurs d'agrégation

```
// Aggregate information about adjacent triplets =====  
  
def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexID]]  
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]  
def aggregateMessages [Msg: ClassTag] (  
    sendMsg: EdgeContext[VD, ED, Msg] => Unit,  
    mergeMsg: (Msg, Msg) => Msg,  
    tripletFields: TripletFields = TripletFields.ALL)  
: VertexRDD[A]
```

- rassembler sur les sommets des informations sur leur voisinage

[Exemple : collectNeighbors]

```
class GraphOps[VD, ED] {  
    def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]  
    def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[ Array[(VertexId, VD)] ]  
}
```

- rassembler sur chaque sommet ses voisins et leur attributs
- pas efficaces, essayer d'utiliser **aggregateMessages**

```
val resultat:VertexRDD[Array[(VertexId, (String, String))]] =  
graph.collectNeighbors(EdgeDirection.Either)
```

```
resultat.collect.foreach(sommet=>(print(sommet._1),  
sommet._2.foreach(print(_)),println()))
```

Résultat :

```
2(5,(franklin,prof))  
3(7,(jgonzal,postdoc))(5,(franklin,prof))  
5(3,(rxin,student))(2,(istoica,prof))(7,(jgonzal,postdoc))  
7(3,(rxin,student))(5,(franklin,prof))
```

Exemple : aggregateMessages

```
class Graph[VD, ED] {  
    def aggregateMessages [Msg: ClassTag] (  
        sendMsg: EdgeContext[VD, ED, Msg] => Unit,  
        mergeMsg: (Msg, Msg) => Msg,  
        tripletFields: TripletFields = TripletFields.All)  
        : VertexRDD[Msg]  
}
```

- **aggregateMessage**: applique une fonction sendMsg (\approx map) définie par l'utilisateur à chaque triplet puis utilise mergeMsg (\approx reduce) pour agréger ces messages pour le sommet destination
- **EdgeContext** : représente une arrête avec les sommets correspondants et permet d'envoyer des messages aux sommets
- **VertexRDD[Msg]** : contient les messages agrégés (de type Msg) pour chaque sommet. Les sommets qui n'ont pas reçu de message ne sont pas inclus dans le résultat
- **tripletFields** : indique quelles informations de EdgeContext sont accessibles pour sendMsg (autres valeurs possibles : Dst, Src, EdgeOnly, None)

[

Exemple

]

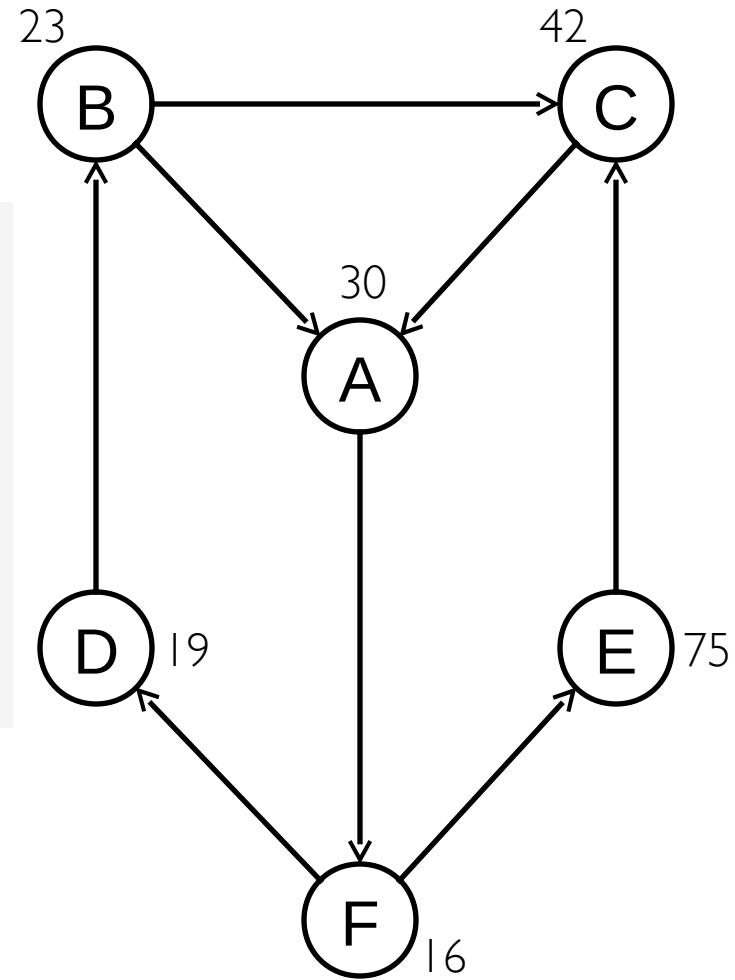
Calculer le nombre de *followers* plus âgés pour chaque sommet et la somme de leurs âges

```
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
```

- `olderFollowers.collect.foreach(println(_))`

Résultats :

(A, (1, 42.0))
 (C, (1, 75.0))
 (F, (1, 30.0))



VertexRDD

VertexRDD[VD] :

- représente un ensemble de nœuds, chacun ayant un attribut de type VD.
- chaque VertexID doit être unique, n'apparaît pas explicitement
- les attributs des nœuds sont stockés dans un hash-map => permet de faire les jointures en temps constants entre deux VertexRDD dérivées à partir de la même VertexRDD

```
class VertexRDD[VD] extends RDD[(VertexID, VD)] {  
    // Filter the vertex set but preserves the internal index  
    def filter(pred: Tuple2[VertexId, VD] => Boolean): VertexRDD[VD]  
    // Transform the values without changing the ids (preserves the internal index)  
    def mapValues[VD2](map: VD => VD2): VertexRDD[VD2]  
    def mapValues[VD2](map: (VertexId, VD) => VD2): VertexRDD[VD2]  
    // Remove vertices from this set that appear in the other set  
    def diff(other: VertexRDD[VD]): VertexRDD[VD]  
    // Join operators that take advantage of the internal indexing to accelerate joins (substantially)  
    def leftJoin[VD2, VD3](other: RDD[(VertexId, VD2)])(f: (VertexId, VD, Option[VD2]) => VD3): VertexRDD[VD3]  
    def innerJoin[U, VD2](other: RDD[(VertexId, U)])(f: (VertexId, VD, U) => VD2): VertexRDD[VD2]  
    // Use the index on this RDD to accelerate a 'reduceByKey' operation on the input RDD.  
    def aggregateUsingIndex[VD2](other: RDD[(VertexId, VD2)] , reduceFunc: (VD2, VD2) => VD2): VertexRDD[VD2]  
}
```

EdgeRDD

- Les arcs sont organisés en blocks, partitionnés avec une des stratégies (*CanonicalRandomVertexCut, EdgePartition1D, EdgePartition2D, RandomVertexCut*)
- Les attributs sont stockés séparément de la structure d'adjacence afin de pouvoir les changer facilement
- Trois fonctions additionnelles, en plus des fonctions héritées de la classe RDD :

```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
// Revere the edges reusing both attributes and structure
def reverse: EdgeRDD[ED]
// Join two 'EdgeRDD`'s partitioned using the same partitioning strategy.
def innerJoin[ED2, ED3](other: EdgeRDD[ED2])(f: (VertexId, VertexId, ED, ED2) => ED3): EdgeRDD[ED3]
```

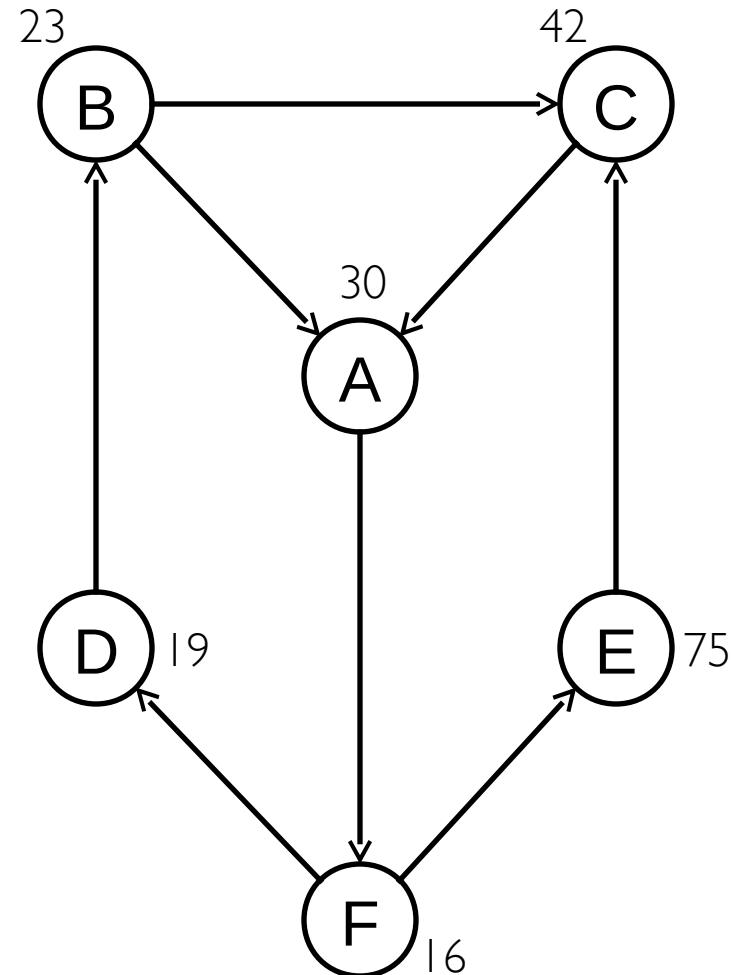
Exemple: moyenne des âges des followers plus âgés

```
// Divide total age by number of older followers to get average age of older followers  
  
val avgAgeOfOlderFollowers: VertexRDD[Double] =  
  
  olderFollowers.mapValues( id, value ) =>  
    value match { case (count, totalAge) => totalAge / count }  
  
// Display the results  
  
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

- avgAgeOfOlderFollowers.collect.foreach(println(_))

Résultats :

(A, 42.0)
(C, 75.0)
(F, 30.0)



Autres opérateurs

```
// Iterative graph-parallel computation =====
def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
    vprog: (VertexID, VD, A) => VD,
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)],
    mergeMsg: (A, A) => A)
    : Graph[VD, ED]
```

```
// Basic graph algorithms =====
def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
def connectedComponents(): Graph[VertexID, ED]
def triangleCount(): Graph[Int, ED]
def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
```

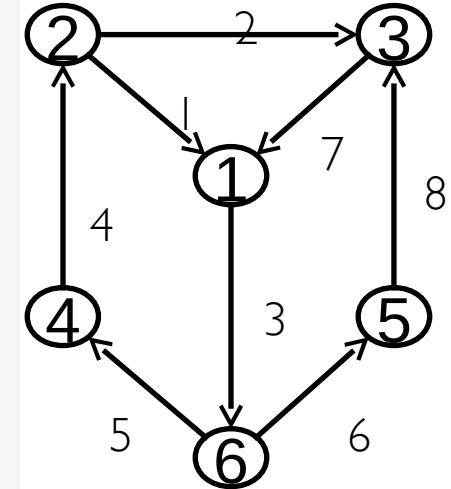
API Pregel

```
// Iterative graph-parallel computation =====  
def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(  
    vprog: (VertexID, VD, A) => VD,  
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)],  
    mergeMsg: (A, A) => A)  
: Graph[VD, ED]
```

- Algorithmes itératifs qui calculent les propriétés des sommets en fonction des propriétés de leur voisins
- Arrêt du calcul : lorsqu'il n'y a plus de messages (à chaque étape les sommets qui n'ont pas reçu de message n'envoient pas de messages)
- **vprog** : fonction exécutée par chaque sommet pour agréger ses propriétés avec les messages reçus à chaque étape
- **sendMsg** : pour envoyer un message à chaque voisin
- **mergeMsg** : fonction pour agréger les messages reçus

Plus courts chemins

```
val sourceId: VertexId = 1 // The ultimate source
// Initialize the graph such that all vertices except the root have distance infinity.
val initialGraph = graph.mapVertices((id, _) =>
  if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel[Double.PositiveInfinity](
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => { // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b) // Merge Message
)
println(sssp.vertices.collect.mkString("\n"))
```



Résultats :

(1,0.0) (2,12.0) (3,14.0) (4,8.0) (5,9.0) (6,3.0)

Utilisation de PageRank en GraphX

```
import org.apache.spark.graphx.GraphLoader

// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
    val fields = line.split(",")
    (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
    case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

Conclusion et Observations

Vues spécifiques à un domaine: Tables et Graphes

- » Les tables et les graphes sont les objets de base composable
- » opérateurs spécialisés qui exploitent la sémantique des vues

Un système unique qui couvre efficacement le pipeline

- » minimise les mouvements et la duplication de données
- » Plus besoin d'apprendre et gérer différents systèmes

Les graphes vus par l'oeil des BD

- » Graph-Parallel Pattern → Triplet joints en algèbre relationnelle
- » Systèmes de graphe → optimisations de jointures distribuées

Programmer avec GraphX

<http://spark.apache.org/docs/latest/graphx-programming-guide.html>

[Pour démarrer]

il faut d'abord importer GraphX dans la console Spark:

```
import org.apache.spark.graphx._  
import org.apache.spark.rdd.RDD
```

[TME]

Exercice 1: Construire un graphe à partir des fichiers facebook_edges_prop.csv et facebook_users_prop.csv (voir la section "Graph Builders").

Le fichier facebook_edges_prop.csv contient sur chaque ligne les informations suivantes pour un arc:

source, destination, type_relation, nombre_messages_échangés

Le fichier facebook_users_prop.csv contient les informations suivantes sur chaque utilisateur:

utilisateur, prénom, nom, âge

[TME]

- Le type de la structure Spark qui stocke les sommets est :

`org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, (String, String, Int))]`

- Le type de la structure Spark qui stocke les arêtes est :

`org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[(String, Int)]]`

- Le type de la structure Spark qui stocke le graphe est :

`org.apache.spark.graphx.Graph[(String, String, Int),(String, Int)]`

TME

Afficher les prénoms des amis de Kendall. On considère le graphe non-dirigé.

```
graph.triplets.filter{t =>t.srcAttr._1 == "Kendall" || t.dstAttr._1 ==  
"Kendall"}.map(t=>if(t.srcAttr._1=="Kendall") t.dstAttr._1 else  
t.srcAttr._1).collect.foreach(u=>println(u))
```

Afficher pour l'utilisateur 'Kendall' son nom et son âge (utiliser la vue graph.vertices).

```
graph.vertices.filter{case(id,(p,n,a)) => p=="Kendall"}.collect.foreach(u=>println(u._1 +"  
" + u._2._2 + " " + u._2._3))
```