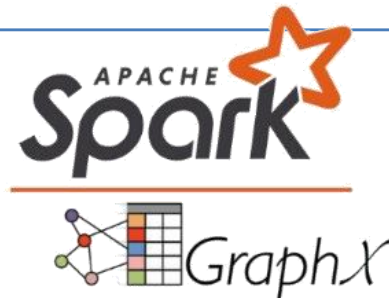


Manipuler des graphes avec GraphX

- Présentation de GraphX.
- Les différentes opérations.
- Vertex and Edge.
- Créer des graphes.
- Présentation de différents algorithmes.
- GraphX Vs GraphFrame



TP: Manipulation de l'API GraphX et l'API GraphFrame.

*Basé sur la présentation de
Camelia Constantin &
J. Gonzalez*

Présentation

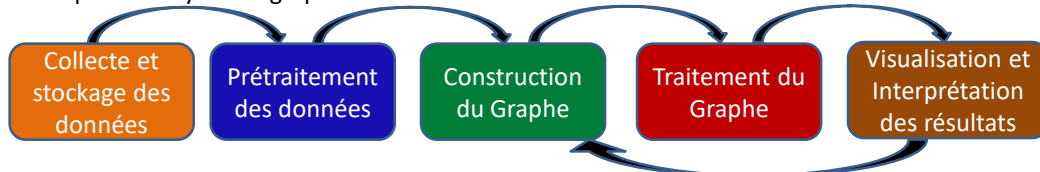
- Les graphes sont utilisés dans de nombreux domaines pour modéliser et analyser des **relations** complexes entre des **entités**, comme:
 - **Réseaux sociaux** (*Facebook, Twitter, LinkedIn, ...*): pour représenter les **relations** entre les **utilisateurs** et les analyser ensuite (*amitiés, interactions, communautés, ...*).
 - **Systemes de recommandation**: pour modéliser les préférences et les **relations** entre les **utilisateurs** et les **éléments** (*produits, films, événements, lieu, voyage, ...*) pour les analyser afin de faire des recommandations.
 - **Logistique et transport** : pour modéliser les réseaux de transport (*les itinéraires, les connexions entre les différents points, ...*) et optimiser les trajets, planifier les livraisons,...

Présentation

- **Recherche d'information** : pour représenter les liens entre les **pages Web**, les **documents**, les **mots-clés**, etc, afin d'améliorer les algorithmes de recherche, de détecter les communautés d'intérêt, etc.
- **Analyse de fraudes** : pour détecter les schémas de fraude dans les **transactions** financières, les réseaux de blanchiment d'argent, etc. Ils permettent d'identifier les connexions suspectes entre les entités.
- **Optimisation de réseau** : pour résoudre des problèmes d'optimisation tels que le calcul du plus court chemin, le flot maximal, le problème du voyageur de commerce, etc.

Présentation: Etapes d'analyse

- ❑ Etapes d'analyse des graphes massifs:



- ❑ **Collecte** de plusieurs sources (fichiers, BD, flux en temps réel, ...)
- ❑ **Stockage** de façon efficace pour permettre un accès rapide lors du traitement (*BD graphiques, SF distribués, ...*)
- ❑ **Prétraitement** des données pour les rendre plus faciles à utiliser (*suppression des doublons, normalisation des valeurs,, conversion de format, ...*)
- ❑ **Construction** du graphe massif en utilisant des structures de données qui faciliteront son traitement distribué.
- ❑ **Traitement** du graphe pour découvrir des chemins optimaux, des communautés,
- ❑ **Visualisation** des résultats sous forme de diagrammes pour faciliter leur compréhension.

Présentation: Algorithmes des graphes

- ❑ Les algorithmes de graphes sont des méthodes qui manipulent et analysent les structures de données graphiques composées de **nœuds** (ou **sommets**) et de **liens** (ou **arêtes**) entre ces nœuds pour résoudre des problèmes liés aux graphes.
- ❑ **Exemples de problèmes:**
 - **Classiques:** recherche de chemin le plus court, recherche de cycles, coloration des graphes,
 - **Spécifiques:** l'algorithme des moindres carrés alternés (**Alternating Least Squares**), l'algorithme **PageRank** pour déterminer la popularité d'une page web, ...

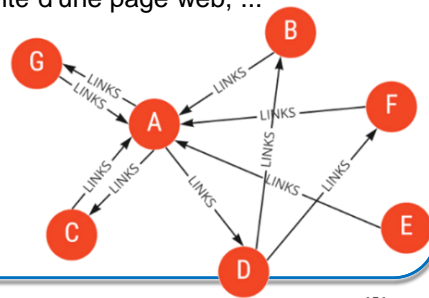
$$PR(P_i) = (1 - d) + d * \sum_{j=1}^n PR(P_j) / L(P_j)$$

P_j : page contenant un lien vers la page P_i

$L(P_j)$: nombre de liens sortant de la page P_j

d : facteur d'amortissement [0, 1] (généralement = 0.85)

Etat initial: on donne les même Rank à toutes les pages.

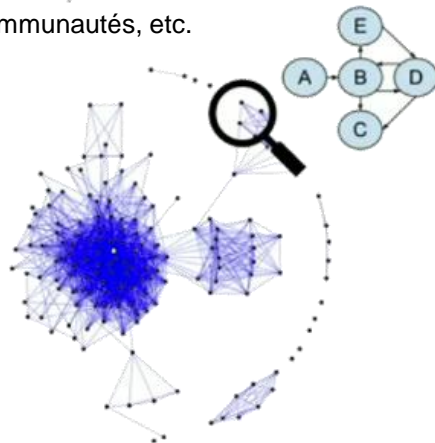


Présentation: Requêtes sur les graphes

- ❑ Les requêtes de graphes sont des opérations utilisées pour **extraire** des **informations spécifiques** à partir d'un graphe: rechercher des motifs, trouver des chemins, calculer des distances, détecter des communautés, etc.

- ❑ **Exemple de requêtes:** chercher les amis d'un utilisateur B, les éditeurs Wikipédia ayant modifié le même article, ...

- ❑ Il existe plusieurs langages de requêtes de graphes dont chacun a sa propre syntaxe et ses fonctionnalités spécifiques, comme **Cypher**, **Gremlin**, **SPARQL**, **GQL**, etc.



Présentation: Algorithmes & Requêtes sur les graphes

Deux systèmes

Algorithmes de graphe parallèles



APIs spécialisées pour simplifier la programmation sur des graphes.

- Nouvelles techniques de partitionnement du graphe avec des types d'opérations qui peuvent être utilisées.
- Exploitent la structure du graphe pour obtenir des gains en performance

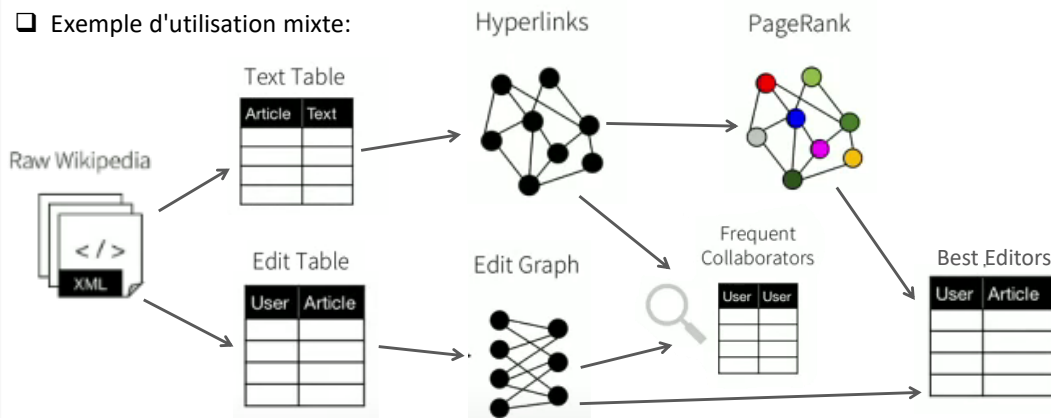
Inconvénients: difficile d'exprimer les différentes étapes d'un pipeline de traitement sur des graphes.

Requêtes sur les graphes



Présentation: Algorithmes & Requêtes sur les graphes

❑ Exemple d'utilisation mixte:



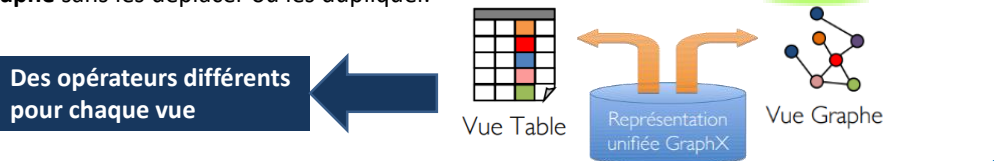
❑ Opérations relationnelles (transformation, ...)

❑ Algorithmes de Graphe

❑ Requêtes sur les Graphes

GraphX



- ❑ **GraphX** est une **bibliothèque** (en **Scala**) de traitement de **graphes** intégrée à Spark.
- ❑ Elle offre des fonctionnalités avancées pour manipuler et analyser des graphes massifs
- ❑ **GraphX** utilise un modèle de programmation basé sur les **RDD** de Spark.
- ❑ **GraphX** atténue la distinction entre les **Tables** et les **Graphes**.
- ❑ Permet aux utilisateurs de:
 - exprimer facilement et efficacement **le pipeline entier de traitement de graphe**.
 - voir les mêmes données d'une **collections (RDD)** comme **graphe** sans les déplacer ou les dupliquer.



GraphX

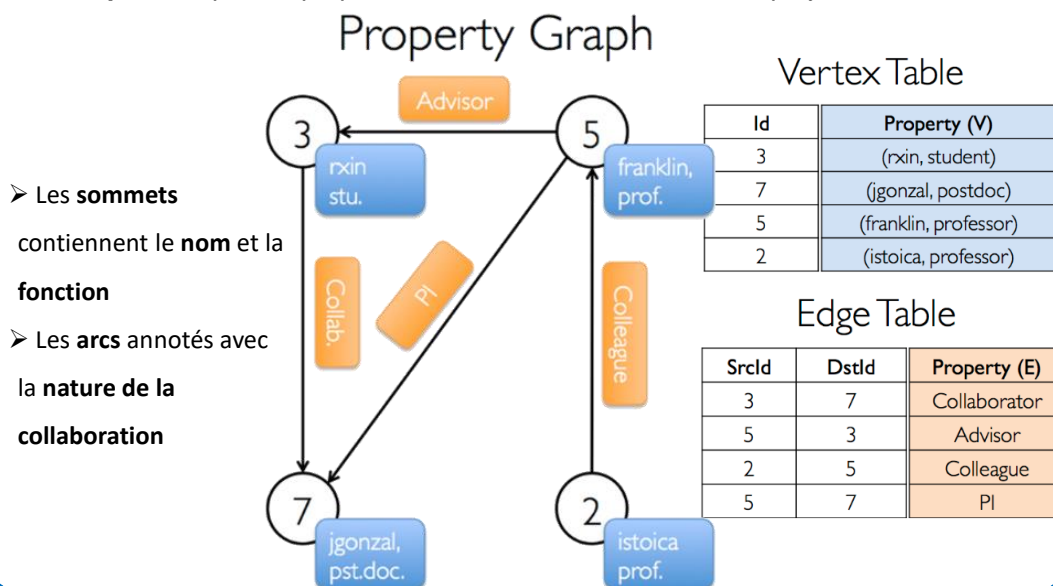
- ❑ Avantages de **GraphX**:
 - **Flexibilité**:
 - **GraphX** unifie les opérations d'ETL, l'utilisation des graphes pour le traitement (algorithmes), les requêtes de graphes et le traitement itératif dans un seul système.
 - Les abstractions **Pregel** et **GraphLab** pour manipuler les graphes peuvent 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.
 - **Vitesse**:
 - **GraphX** fournit des performances comparables aux systèmes les plus performants de traitement de graphes spécialisés.
 - **GraphX** est très performant sur les graphes de **petite et moyenne taille**.

GraphX

- ❑ Un **graphe de propriétés** est un **multi-graphe dirigé** avec des **objets (sommets)** définis par l'utilisateur et reliés par des **arêtes**.
- ❑ **Multi-graphe** signifie qu'il peut y avoir **plusieurs arêtes partageant la même source et destination** (*plusieurs relations entre nœuds*)
- ❑ Chaque **sommet** possède des attributs et une clé (ID) unique de type **VertexID** de 64 bits. 
- ❑ Chaque **arête** a l'ID du sommet **source** et l'ID du sommet **destination** 
- ❑ Dans **GraphX**, un 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 (arêtes)**
- ❑ **VD** et **ED (Vertex Data et Edge Data)** sont les types des objets associés aux **sommets** et aux **arcs** et qui représentent leurs attributs.
- ❑ Les classes **VertexRDD[VD]** et **EdgeRDD[ED]** sont des versions optimisées des **RDD** fournissant des fonctionnalités supplémentaires pour le traitement et optimisation des graphes

GraphX

- ❑ **Exemple:** Graphe de propriétés des **collaborateurs** dans un projet de recherche



GraphX

- ❑ Exemple: Les attributs d'un sommet peuvent être regroupés dans un tuple

```
import org.apache.spark.graphx.{Graph, Edge}
import org.apache.spark.rdd.RDD

type VertexId = Long // C'est le type par défaut de VertexID

//Créer une RDD pour les sommets et une RDD pour les arêtes
val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
                                                                (2L, ("istoica", "prof")), (7L, ("jgonzal", "postdoc")),
                                                                (5L, ("franklin", "prof"))))

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

//Créer le graphe
val graph = Graph(users, edges)

// Affichage des sommets et les arêtes
graph.vertices.foreach (println)
graph.edges.foreach (println)
```

(7, (jgonzal, postdoc))	Edge(5, 3, advisor)
(5, (franklin, prof))	Edge(5, 7, pi)
(2, (istoica, prof))	Edge(2, 5, colleague)
(3, (rxin, student))	Edge(3, 7, collab)

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

GraphX

- ❑ Les opérateurs des RDD sont hérités de Spark pour les deux propriétés **vertices** et **edges**:

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

GraphX

❑ Exemple en Scala (Suite): Utilisation des transformations RDD

```
// compter les utilisateurs ayant la fonction "postdocs"
println (graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count )

// Compter tous les arêtes où src<dest et la relation est "collab"
println (graph.edges.filter(e => e.srcId < e.dstId && e.attr=="collab").count)

// On peut aussi utiliser case
println (graph.edges.filter { case Edge (src, dst, rel) => src < dst &&
                                                                    rel=="collab" }.count)
```

e.attr représente l'attribut de l'arête entre les deux sommets **e.srcId** et **e.dstId**

GraphX

❑ Les attributs d'un sommet peuvent être regroupés dans un objet

```
//Le sommet est un objet Person
case class Person (name: String, pos: String)

val rdd1: RDD[(VertexId, Person)] = sc.parallelize(Seq((3L, Person("rxin", "student")),
                                                    (5L, Person("franklin", "prof")), (7L, Person("jgonzal", "postdoc")),
                                                    (2L, Person("istoica", "prof"))))

val graph1 = Graph(rdd1, edges)

// Filtrer les sommets où la fonction est "postdoc" et calculer leur nombre
println (graph1.vertices.filter { case (id, personne) => personne.pos == "postdoc" }.count)

// Affichage des sommets de graph1
graph1.vertices.foreach(println)
```

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


GraphX

❑ Une arête peut avoir plusieurs attributs regroupés dans Tuple

```
// Une arête a deux attributs (nom et poids) regroupés dans un Tuple(String, Double)
val rdd2: RDD[ Edge[(String, Double)] ] = sc.parallelize(Array(Edge(3L,7L,("collab", 0.5)),
Edge(5L, 3L, ("advisor", 0.9)), Edge(2L, 5L, ("colleague",0.4)),Edge(5L, 7L, ("PI", 1.0))))
val graph2 = Graph(users, rdd2)

// Filtrer les arêtes ayant un poids >0.5 et calculer leur nombre
println(graph2.edges.filter { e => e.attr._2 > 0.5 }.count)

case Edge (src, dst, rel)=> rel._2 > 0.5 }

// Affichage des arêtes de graph2
graph2.edges.foreach(println)
Edge(2,5,(colleague,0.4))
Edge(3,7,(collab,0.5))
Edge(5,3,(advisor,0.9))
Edge(5,7,(PI,1.0))
```

GraphX

❑ Une arête peut avoir plusieurs attributs regroupés dans un Objet

```
// Une arête a 2 attributs regroupés dans l'objet: Relation (name: String, weight: Double)
case class Relation (nom: String, poids: Double)

val rdd3: RDD[ Edge[Relation] ] = sc.parallelize(Array(Edge(3L,7L,Relation("collab",
0.5)), Edge(5L, 3L, Relation("advisor", 0.9)), Edge(2L, 5L,
Relation("colleague",0.4)),Edge(5L, 7L, Relation("PI", 1.0))))
val graph3 = Graph(users, rdd3)

// Filtrer les arêtes ayant un poids >0.5 et calculer leur nombre
println(graph3.edges.filter { e => e.attr.poids > 0.5 }.count)

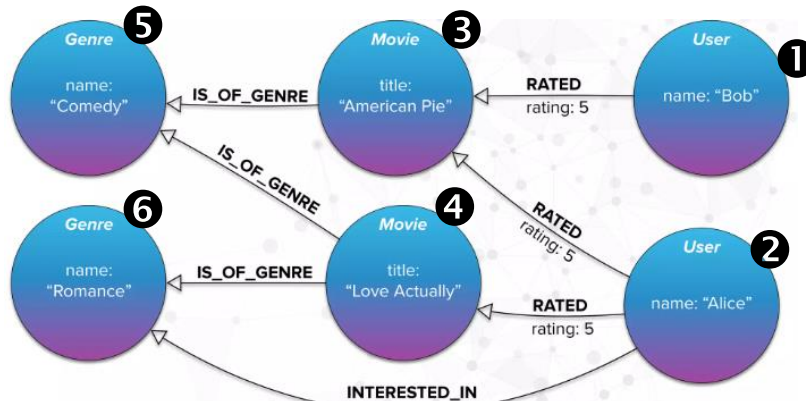
// Affichage des arêtes de graph2
graph3.edges.foreach(println)
Edge(2,5,Relation(colleague,0.4))
Edge(3,7,Relation(collab,0.5))
Edge(5,3,Relation(advisor,0.9))
Edge(5,7,Relation(pi,1.0))
```

GraphX

- Les sommets d'un graphe de propriétés peuvent être de types différents dont chacun a un identifiant unique (VertexID)

Exemple: Graphe des préférences et notations de films par les utilisateurs

3 types de nœuds et 3 types de relations



TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

166

GraphX

- Les attributs des arêtes sont regroupés dans un objet

```
case class Lien (name:String, note: Any=null) // classe des arêtes, 2ème paramètre est par défaut null
val nodes: RDD[(VertexID, (String, String))] = sc.parallelize( Seq ( (1L, ("user", "Bob")),
(2L, ("user", "Alice")), (3L, ("Movie", "American Pie")), (4L, ("Movie", "Love Actually")),
(5L, ("Genre", "Comedy")), (6L, ("Genre", "Romance"))))
val arcs: RDD[Edge[(Lien)]] = sc.parallelize( Seq ( Edge(1L, 3L, Lien("Rated", 5)),
Edge(2L, 3L, Lien("Rated", 5)), Edge(2L, 4L, Lien("Rated", 5)), Edge(2L, 6L, Lien("Interested")),
Edge(4L, 6L, Lien("Is_Of_Genre")), Edge(4L, 5L, Lien("Is_Of_Genre")),
Edge(3L, 5L, Lien("Is_Of_Genre"))))
val graph4 = Graph(nodes, arcs)
graph4.edges.foreach(println)
```

```
Edge(1,3,Lien(Rated,5))
Edge(2,3,Lien(Rated,5))
Edge(2,4,Lien(Rated,5))
Edge(2,6,Lien(Interested,null))
Edge(3,5,Lien(Is_Of_Genre,null))
Edge(4,5,Lien(Is_Of_Genre,null))
Edge(4,6,Lien(Is_Of_Genre,null))
```

TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

167

GraphX

❑ Calcul de la moyenne des notes de la relation Rated

```
val nbrated = graph4.edges.filter { case Edge(src , dst , rel) => rel.name == "Rated" }.count

val somrated = graph4.edges.filter { case Edge(src , dst , rel) => rel.name == "Rated" }
    .map{e => (e.attr.name , e.attr.note) }
    .reduceByKey((a , b)=>a.toString.toInt+b.toString.toInt).collect() (0)._2

println("Nbrated:"+ nbrated)
println("Somrated:"+somrated)
println("Moyenne"+( somrated.toString.toDouble / nbrated.toDouble))
```

```
[ ( Rated, 15 ) ]
```

```
Nbrated:3
Somrated:15
Moyenne5.0
```

Opérateurs d'informations

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

// Information about the Graph =====

val numEdges: Long

val numVertices: Long

val inDegrees: VertexRDD[Int]

val outDegrees: VertexRDD[Int]

val degrees: VertexRDD[Int]

❑ Calculer le degré entrant de chaque sommet :

```
val indeg = graph.inDegrees
```

```
indeg.collect.foreach(println(_))
```

```
(3, 1)
```

```
(7, 2)
```

```
(5, 1)
```

N.B: Les nœuds ayant un degré 0 ne sont pas présents

❑ Calculer le degré de chaque sommet :

```
val deg = graph.degrees
```

```
deg.collect.foreach(println(_))
```

```
(3, 2)
```

```
(7, 2)
```

```
(5, 3)
```

```
(2, 1)
```

N.B: tous les nœuds sont présents

❑ Calculer le degré maximal:

```
val maxdeg = graph.degrees.reduce((a, b) => if (a._2 > b._2) a else b)
```

```
println("maxdeg:" + maxdeg)
```

```
val maxindeg = graph.inDegrees.reduce((a, b) => if (a._2 > b._2) a else b)
```

```
println("maxindeg:" + maxindeg)
```

```
val maxoutdeg = graph.outDegrees.reduce((a, b) => if (a._2 > b._2) a else b)
```

```
println("maxoutdeg:" + maxoutdeg)
```

```
maxdeg: (5, 3)
```

```
maxindeg: (7, 2)
```

```
maxoutdeg: (5, 2)
```

Les vues d'informations

```
// Views of the graph as collections -----
```

```
val vertices: VertexRDD[VD]
```

```
val edges: EdgeRDD[ED]
```

```
val triplets: RDD[EdgeTriplet[VD, ED]] // Fusion des deux vues vertices et edges
```

```
/** Calculer le nombre d'arêtes par relation
```

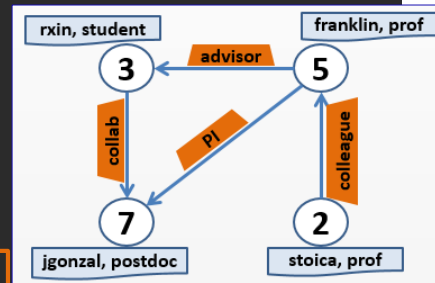
```
graph.edges.foreach(println)
```

```
graph.edges.map(e => (e.attr, 1))
```

```
.reduceByKey((a, b) => a+b)
```

```
.foreach(println)
```

```
(colleague, 1)
(PI, 1)
(advisor, 1)
(collab, 1)
```



La vue Triplet

- ❑ En plus des deux vues **vertices** et **edges**, il existe la vue **triplets**: `RDD[EdgeTriplet[VD, ED]]`
- ❑ La classe **EdgeTriplet** étend la classe **Edge** en ajoutant **srcAttr** et **dstAttr** contenant les propriétés des nœuds **source/destination**

```
// Afficher la vue triplets
```

```
graph.triplets.foreach(println)
```

```
((2, (istoica, prof)), (5, (franklin, prof)), colleague)
((3, (rxin, student)), (7, (jgonzal, postdoc)), collab)
((5, (franklin, prof)), (3, (rxin, student)), advisor)
((5, (franklin, prof)), (7, (jgonzal, postdoc)), PI)
```

```
// Concaténer le nom de chaque membre (utilisateur) destination et celui de la source
```

```
graph.triplets.map (
```

```
triplet => triplet.srcAttr._1 + " est le " + triplet.attr + " de " + triplet.dstAttr._1
).foreach( println )
```

```
istoica est le colleague de franklin
rxin est le collab de jgonzal
franklin est le advisor de rxin
franklin est le PI de jgonzal
```

Opérateurs de transformations

```
// 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 mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

- ❑ **mapXX** : produit un **nouveau graphe** de **même structure** avec **XX** modifié par la fonction **map**
- ❑ La **structure n'est pas affectée** (le graphe résultat garde la même structure que celui d'origine)
- ❑ On peut utiliser ces opérateurs pour:
 - **initialiser** le graphe **pour un calcul** en **ajoutant des propriétés** aux sommets pour effectuer ce calcul.
 - **enlever des propriétés** inutiles

```
/** Enlever la 2ème propriété des sommets
```

```
val mapgraph = graph.mapVertices((id, attr) => attr._1)
mapgraph.vertices.foreach(println)
```

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

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

Opérateurs de 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]
```

Inverser les arêtes d'un graphe orienté. $A \rightarrow B$ devient $B \rightarrow A$

Construire un sous-graphe selon deux prédicats des sommets et des arêtes.

```
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
```

- ❑ **mask**: retourne un **sous-graphe** correspondant à l'**intersection** d'un **graphe donné** et d'un **graphe masque**. On garde les arêtes d'intersection (*mêmes sommets et même relations*).
- ❑ **groupEdges**: pour un **multi-graphe**, fusionne les différentes arêtes entre deux sommets en une seule

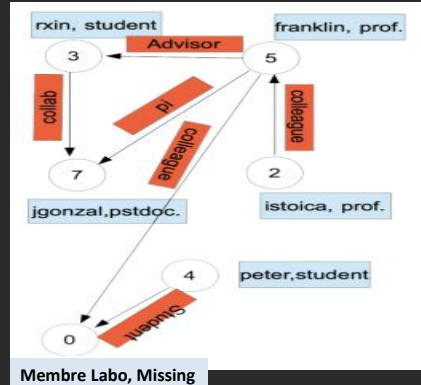
Opérateurs de structure

```
//RDD pour les sommets et les arêtes
val rdd_users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
    (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
    (4L, ("peter", "student"))))

val rdd_link: 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") ))

// utilisateur par défaut pour les arêtes
// dont l'un des sommets n'est pas défini
val defaultUser = ("Membre Labo", "Missing")

// Construire le graphe
val new_graph = Graph(rdd_users, rdd_link, defaultUser)
```



TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

174

Opérateurs de structure

❑ Créer un sous-graphe en éliminant des sommets spécifique

```
new_graph.triplets.foreach(println)
```

```
((2,(istoica,prof)),(5,(franklin,prof)),colleague)
((3,(rxin,student)),(7,(jgonzal,postdoc)),collab)
((4,(peter,student)),(0,(Membre Labo,Missing)),student)
((5,(franklin,prof)),(0,(Membre Labo,Missing)),colleague)
((5,(franklin,prof)),(3,(rxin,student)),advisor)
((5,(franklin,prof)),(7,(jgonzal,postdoc)),pi)
```

```
//Sous-graphe ayant des sommets dont la 2ème propriétés != "Missing"
```

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

```
validGraph.edges.collect.foreach(println(_))
```

```
Edge(2, 5, colleague)
Edge(3, 7, collab)
Edge(5, 3, advisor)
Edge(5, 7, pi)
```

TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

175

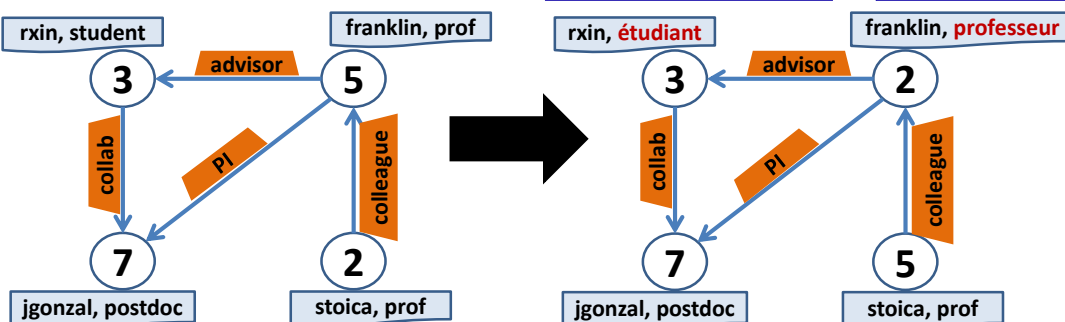
Opérateurs de Jointure

```
// 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]
```

- ❑ Dans certains cas, on a besoin de faire une **jointure de collections externes** (RDD) avec un **graphe** afin de:
 - Ajouter des **propriétés supplémentaires** aux sommets d'un graphe existant
 - Copier des **propriétés** de sommets d'un **graphe vers un autre**.
- ❑ La clé de jointure est de type **VertexId**
- ❑ Les deux opérateurs retournent un **nouveau graphe** en appliquant la fonction **map** aux propriétés obtenues après jointure.

Opérateurs de Jointure

```
/** ***** Modifier la deuxième propriété des sommet 3 et 5
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(_))
```



- ❑ les sommets sans correspondants dans la RDD gardent leur valeur initiale.
- ❑ Pas de changement de type des propriétés des sommets

Opérateurs de transformations

- ❑ Créer un nouveau graphe où chaque arête et sommet sont étiquetée avec un poids

```
val inputGraph = graph.outerJoinVertices(graph.outDegrees)
  ((vid, _, degOut) => degOut.getOrElse(0))
```

graph

Edge(2,5,colleague)	(3,(rxin,student))	→ (3,1)
Edge(3,7,collab)	(7,(jgonzal,postdoc))	→ (5,2)
Edge(5,3,advisor)	(5,(franklin,prof))	→ (2,1)
Edge(5,7,PI)	(2,(istoica,prof))	→ (2,1)

→

(3,(rxin,student),1)
(7,(jgonzal,postdoc),)
(5,(franklin,prof),2)
(2,(istoica,prof),1)

→

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

→

graph5

Edge(2,5,1.0)	(3,1)
Edge(3,7,1.0)	(7,0)
Edge(5,3,0.5)	(5,2)
Edge(5,7,0.5)	(2,1)

←

inputGraph.triplets

((2,1),(5,2),colleague)
((3,1),(7,0),collab)
((5,2),(3,1),advisor)
((5,2),(7,0),PI)

```
// Créer un graphe où chaque arête est étiquetée par un poids et chaque sommet contient 1
val graph5 = inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr)
graph5.edges.foreach(println)
```

TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

178

Opérateurs d'agrégation

```
// Aggregate information about adjacent triplets =====
```

```
def collectNeighborIds (edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
```

```
def collectNeighbors (edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId, VD)]]
```

- ❑ Ces opérateurs permettent de rassembler des informations sur les voisins de chaque sommet.

```
// Rassembler des informations sur les sommets destinations des arêtes
```

```
// sortant de chaque sommet.
```

```
val resultat = graph.collectNeighbors(EdgeDirection.Out)
```

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

```
3(7,(jgonzal,postdoc))
```

```
7
```

```
5(3,(rxin,student))(7,(jgonzal,postdoc))
```

```
2(5,(franklin,prof))
```

TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

179

Opérateurs d'agrégation

// Aggregate information about adjacent triplets

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

- ❑ **aggregateMessage**: applique **sendMsg** (\approx **map**) à chaque triplet puis utilise **mergeMsg** (\approx **reduce**) pour agréger ces messages pour le sommet destination.
- ❑ **EdgeContext** : expose les **attributs** d'une **arête** (**source, destination, relation**) et les fonctions **sendToSrc** et **sendToDst** pour envoyer des **messages** aux **sommets source** et de **destination**.
- ❑ **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** (*valeurs possibles* : **All, Dst, Src, EdgeOnly, None**)

Opérateurs d'agrégation

- ❑ On veut modéliser un graphe de followers dans un réseau social

// Sommets et Arêtes

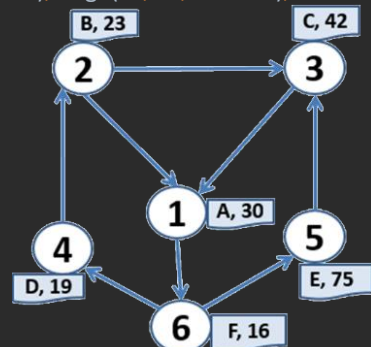
```
val user: RDD[(VertexId, (String, Int))] = sc.parallelize(Array((1L, ("A", 30)), (2L, ("B", 23)),
(3L, ("C", 42)), (4L, ("D", 19)), (5L, ("E", 75)), (6L, ("F", 16))))
```

```
val lien: RDD[Edge[String]] = sc.parallelize(Array(Edge(1L, 6L, "follow"), Edge(2L, 1L, "follow"),
Edge(2L, 3L, "follow"), Edge(3L, 1L, "follow"), Edge(4L, 2L, "follow"), Edge(5L, 3L, "follow"),
Edge(6L, 4L, "follow"), Edge(6L, 5L, "follow")))
```

// Construire le graphe

```
val gsocial = Graph(user, lien)
gsocial.triplets.foreach(println)
```

```
Edge(1, 6, follow)
Edge(2, 1, follow)
Edge(2, 3, follow)
Edge(3, 1, follow)
Edge(4, 2, follow)
Edge(5, 3, follow)
Edge(6, 4, follow)
Edge(6, 5, follow)
```



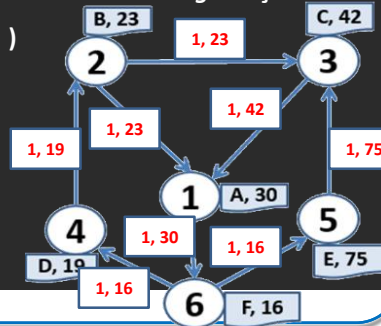
Opérateurs d'agrégation

- On veut calculer pour chaque nœud: le nombre de followers et l'âge du plus vieux

```
val olderFollowers: VertexRDD[(Int, Int)] = gsocial.aggregateMessages [(Int, Int)] (
  triplet => { // La fonction Map
    // Envoyer un message (tuple) au sommet destination contenant 1 (compteur) et l'âge
    triplet.sendToDst( (1, triplet.srcAttr._2) )
  },
  // La fonction Reduce : Calcul de la somme du compteur et le maximum des âges reçus
  (a, b) => (a._1 + b._1, if (a._2 > b._2) a._2 else b._2 )
)
```

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

```
(4, (1, 16))
(1, (2, 42))
(6, (1, 30))
(3, (2, 75))
(5, (1, 16))
(2, (1, 19))
```



TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

182

Opérateurs d'agrégation

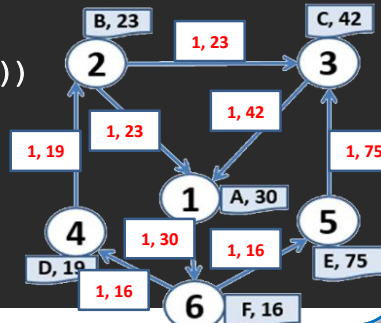
- On veut calculer pour chaque nœud: le nombre de followers et leur âge moyen

```
val Followers: VertexRDD[(Int, Double)] = gsocial.aggregateMessages [(Int, Double)] (
  triplet => { // La fonction Map
    // Envoyer un message (tuple) au sommet destination contenant 1 (compteur) et l'âge
    triplet.sendToDst( (1, triplet.srcAttr._2) )
  },
  // La fonction Reduce : Calcul de la somme du compteur et la somme des âges reçus
  (a, b) => (a._1 + b._1, a._2 + b._2 )
)
```

```
.mapValues( (id, value) => (value._1, value._2 / value._1) )
```

```
Followers.collect().foreach(println)
```

```
(4, (1, 16.0))
(1, (2, 32.5))
(6, (1, 30.0))
(3, (2, 49.0))
(5, (1, 16.0))
(2, (1, 19.0))
```



TRAITEMENT BIG DATA \ N.EL FADDOULI

CC-BY NC SA

183

Algorithmes de Graphes

```
// 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]
```

- ❑ **PageRank** mesure l'importance de chaque sommet dans un graphe, en supposant qu'une arête de **u** à **v** représente une approbation de l'importance de **v** par **u**.
- ❑ L'algorithme **connectedComponents** (composants connectés) étiquette chaque sommet du graphe (*non orienté*) avec l'ID du sommet le plus petit auquel il est connecté (*directement ou indirectement*). Il identifie l'ensemble des sommets qui forme un cluster.
- ❑ **triangleCount** calcule le nombre de triangles dans un graphe ce qui est un indicateur de sa densité. On obtient, pour chaque sommet, le nombre de triangles auxquels il appartient.
- ❑ **stronglyConnectedComponents** permet de trouver un groupe de sommets fortement connectés dans un graphe orienté (*chaque nœud est accessible depuis n'importe quel autre nœud du même groupe*) → **Détection de communauté, ...**

TRAITEMENT BIG DATA \ N.EL FADDOULI CC-BY NC SA

184

Utilisation de PageRank

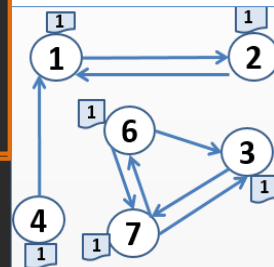
- ❑ **PageRank** des utilisateurs d'un réseau social

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

// Créer un graphe à partir d'arêtes stockées dans un fichier texte
val gfollow = GraphLoader.edgeListFile(sc, "D:\\followers.txt")
// Lancer PageRank
val ranks = gfollow.pageRank(0.0001).vertices
// jointure entre ranks et une RDD des utilisateurs
val users = sc.textFile("D:\\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) }
// Afficher le résultat
ranksByUsername.foreach(println)
```

users

```
(1,BarackObama)
(2,ladygaga)
(3,jeresig)
(4,justinbieber)
(6,matei_zaharia)
(7,odersky)
```



```
(justinbieber,0.15007622780470478)
(BarackObama,1.4596227918476916)
(matei_zaharia,0.7017164142469724)
(jeresig,0.9998520559494657)
(odersky,1.2979769092759237)
(ladygaga,1.3907556008752426)
```

TRAITEMENT BIG DATA \ N.EL FADDOULI CC-BY NC SA

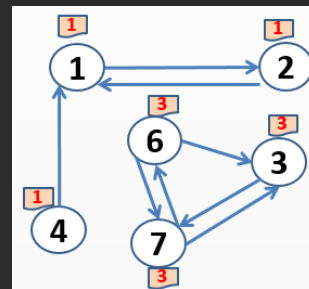
185

Utilisation de ConnectedComponents

☐ Identifier les clusters d'utilisateurs d'un réseau social

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

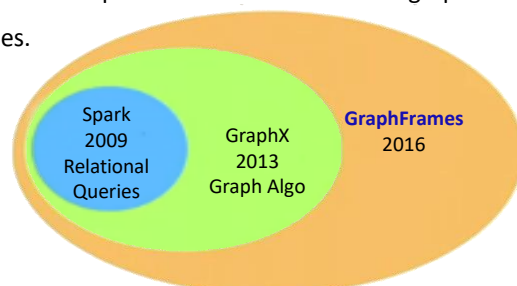
// Créer un graphe à partir d'arêtes stockées dans un fichier texte
val gfollow = GraphLoader.edgeListFile(sc, "D:\\followers.txt")
// Lancer connectedComponents
val ccgraph = gfollow.connectedComponents()
// Afficher les sommets dans l'ordre croissant selon leur attribut
// calculé par connectedComponents
ccgraph.vertices.sortBy(e => e._2).foreach(println)
```



```
(4, 1)
(1, 1)
(2, 1)
(6, 3)
(3, 3)
(7, 3)
```

GraphFrame

- ☐ **GraphFrame** est une bibliothèque de traitement de graphes développée en **Java**, **Scala** et **Python**.
- ☐ Elle est construite au-dessus de la bibliothèque **DataFrame** de traitement de données relationnelles dans Spark.
- ☐ **GraphFrame** permet de manipuler des graphes en utilisant une syntaxe similaire à celle des **DataFrames**, ce qui facilite l'intégration avec d'autres opérations de traitement de données.
- ☐ **GraphFrame** expose des opérations pour lancer des requêtes sur les données d'un graphe et aussi pour exécuter des algorithmes des graphes.



GraphFrame

- ❑ Un graphe dans **GraphFrame** est construit à partir de deux **Dataframe** pour:
 - Les **sommets (Vertex)** du graphe sous forme de tuples (**id, info1, info2, ...**) dont chacun a un identifiant unique (**id**) et contient les **propriétés** d'un sommet.

Exemple: ("a", "Alice", 34) , ("b", "Bob", 36) dont le schéma est (*id, name, age*)
 - Les **relations (Edge)** entre les sommets sous forme d'un tuples (**src, dst, relation**) contenant les **identifiants** des **sommets source** et **destination** ainsi que la **relation** qui les relie. On peut aussi ajouter des propriétés de l'arrête (**poids, ...**)

Exemple: ("a", "b", "Friend")
- ❑ Comme les **Dataframe**, les graphes dans **Graphframe** sont:
 - **Immuables:** les modifications (valeurs ou structure) produisent un nouveau graphe
 - **Distribués:** le graphe est partitionné sur les différents nœuds du cluster.
 - **Résistant aux pannes:** chaque partition du graphe peut être recréé sur une autre machine pour la tolérance aux pannes