GraphX

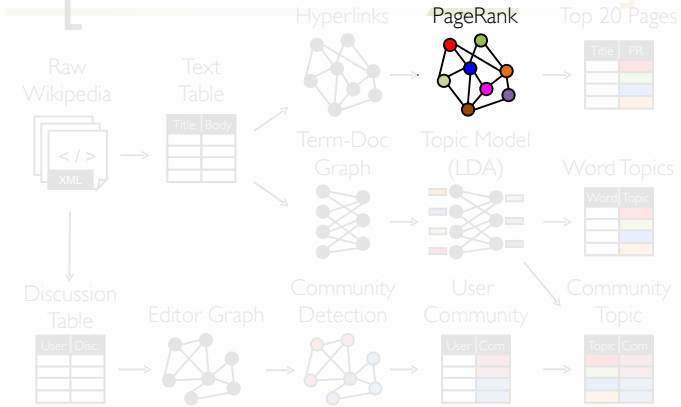
Cours Master Recherche

Camelia Constantin

Prénom.Nom@lip6.fr

basé sur la présentation de J. Gonzalez

Graphs are Central to Analytics



Exemple du calcul de PageRank:

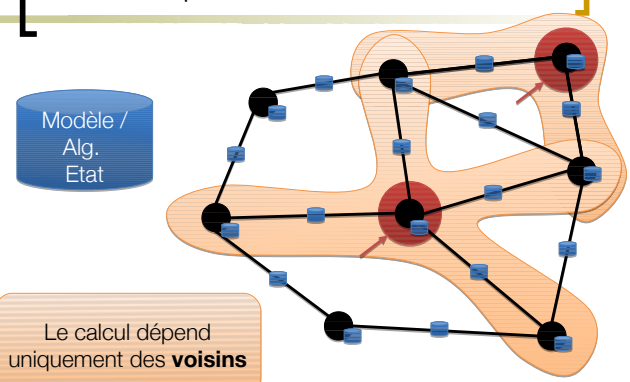
$$R[i] = 0.15 + \sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]$$

Rang de l'util. *i*

Somme pondérée des rangs des voisins

- ***Les calculs des mises à jour des rangs peuvent être fait en parallèle***
- ***On itère jusqu'à la convergence***

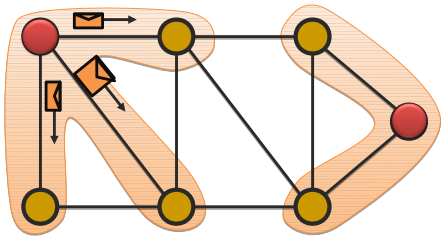
Graph-Parallel Pattern



L'abstraction **Graph-Parallel**

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

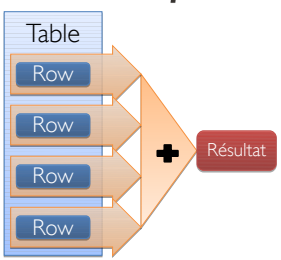
De nombreux algorithmes de calculs parallèles dans les graphes

- 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

Des systèmes séparés pour supporter chaque vue

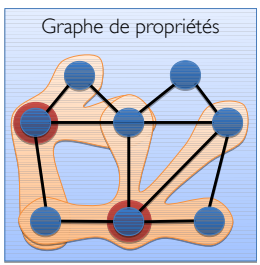
Data-parallel

hadoop Spark

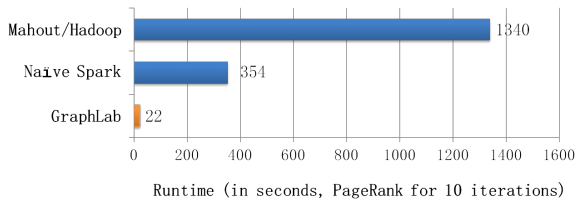


Graph-Parallel

Pregel GraphLab GIRAPH



Exemple: calcul de PageRank sur le graphe de Live-Journal



GraphLab est *60x plus rapide que* Hadoop
GraphLab is *16x plus rapide que* Spark

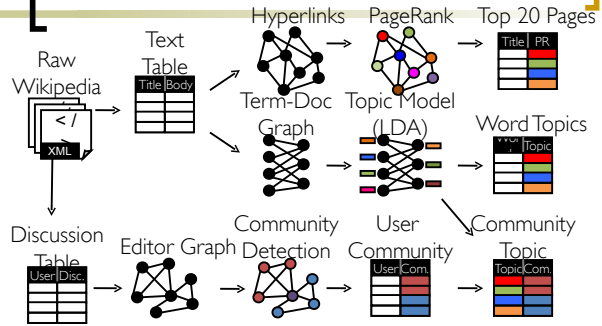
Systèmes Graph-Parallel

- 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 comparativement aux systèmes de données parallèles (Data-Parallel) plus génériques.

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

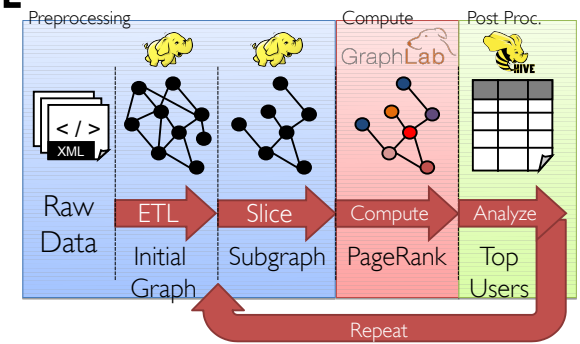
- qui utilise des systèmes data-parallel et graph-parallel →

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)

Exemple de pipeline



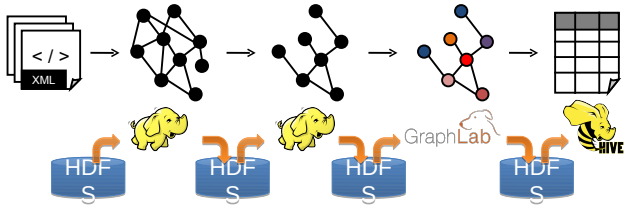
Difficultés pour la programmation et l'utilisation

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

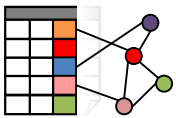


Ré-utilisation limité de structures de données internes d'une étape à l'autre

Solution: L'approche unifiée GraphX

Nouvelle API
Atténue la distinction entre les Tables et les Graphes

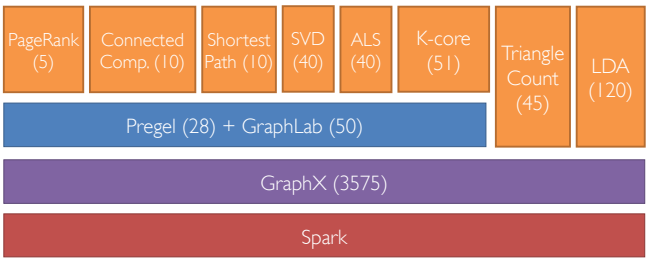
New System
Combine les systèmes Data-Parallel et Graph-Parallel



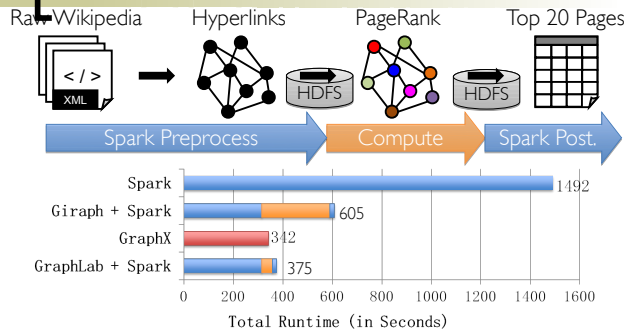
Permet aux utilisateurs:

- 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)



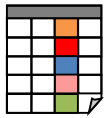
Un exemple de pipeline



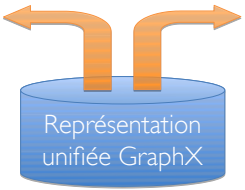
Le temps de traitement de GraphX pour tout le pipeline est plus rapide que Spark+GraphLab

Différentes vues

Les Tables et les Graphes sont des vues composables *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)

Graphe de propriétés

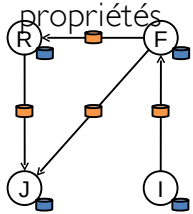


Table des sommets

Id	Propriété (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Table des arêtes

Src_Id	Dst_Id	Propriété (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Opérateurs de Tables

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) ])
  // Table Views -----------------
  def vertices: Table[ (Id, V) ]
  def edges: Table[ (Id, Id, E) ]
  def triplets: Table [ ((Id, V), (Id, V), E) ]
  // 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]
    }
```

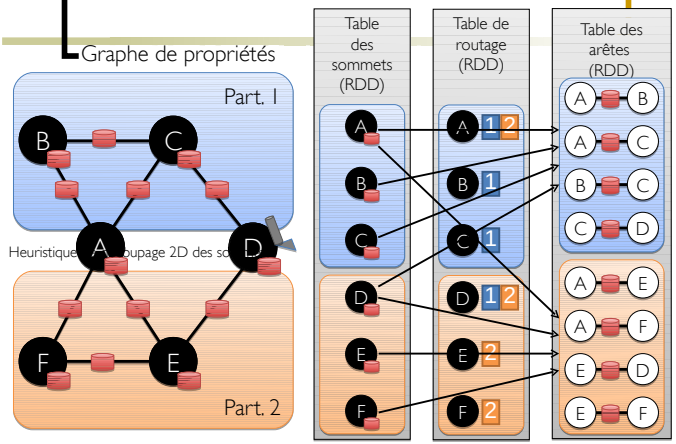
GraphX: avantages

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.

Conception d'un système GraphX

Graphes distribués comme des Tables (RDDs)



Exemple de stratégie d'exécution

Exemple de fonction :

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

- Appliquer une fonction sendMsg à chaque arrête du graphe (envoie un message à un des deux noeuds reliés par l'arrête) → sendMsg est équivalent à map en M/R
- À chaque noeud on aggrège les messages reçus avec mergeMsg → mergeMsg est équivalent à reduce en M/R

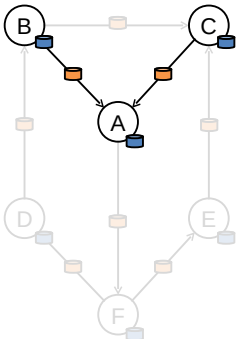
aggregateMessages

Map-Reduce pour chaque sommet

sendMsg(A ← B) ⇒ A_1

sendMsg(A ← C) ⇒ A_2

mergeMsg(A_1, A_2) ⇒ A



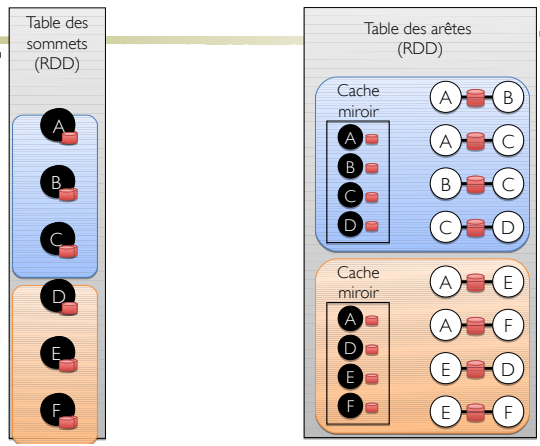
Plan d'exécution pour aggregateMessages

- Déplacer les attributs des nœuds source et destination

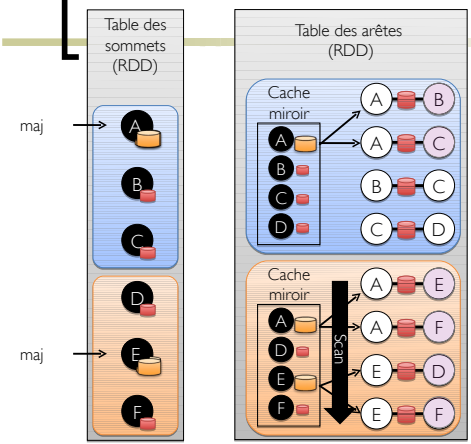
au même endroit que les partitions des arrêtes correspondantes (utilise les tables de routage)

- Générer les triplets (source arrête destination) correspondants
- Appliquer sendMsg et mergeMsg
  - Optimisation :
    - Dans les opérations itératives le nombre de nœuds qui changent d'une itération à une autre décroît → déplacer seulement les nœuds qui ont changé
    - Analyse du code source de la fonction map au moment de l'exécution afin de déterminer si les attributs des nœuds source/destinations sont utilisés (si aucun n'est utilisé la jointure est éliminée)

Déplacement des noeuds

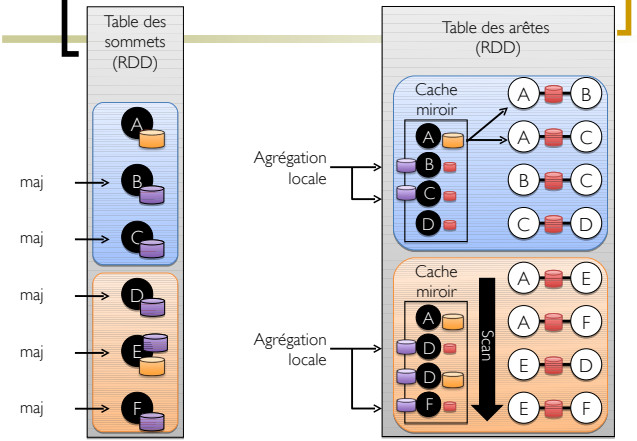


Mises à jour incrémentale

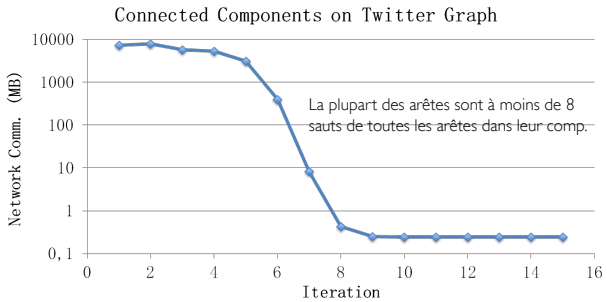


Les arrêtes sont regroupées par leur Nœud source : Bitmap index scan sur les nœuds remplace un scan séquentiel sur les arrêtes

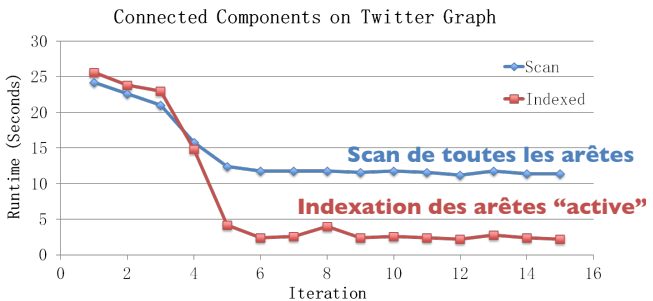
Agrégation locale des valeurs changées



Réduction des coûts de communication due à la mise en cache des mises à jour



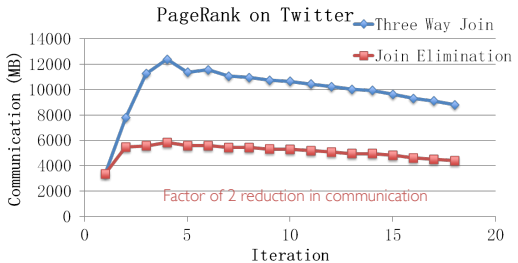
Bénéfice de l'indexation des arêtes *actives*



Elimination de jointures

Identifier et éviter les jointures pour les attributs de triplets inutilisés

Ex: PageRank accède seulement à l'attribut source



Optimisations de requêtes additionnelles

Indexation et Bitmaps:

» Pour accélérer les jointures à travers les graphes

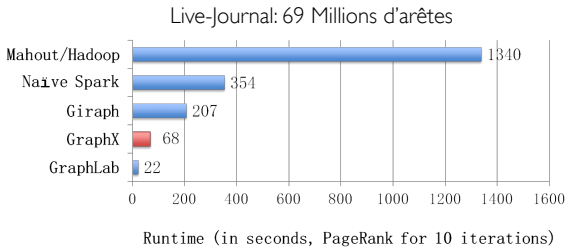
» Pour efficacement construire des sous-graphes

Ré-utilisation des index et des données:

» Ré-utilisation des tables de routage à travers les graphes et les sous-graphes

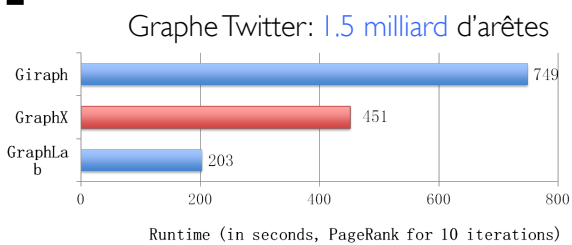
» Ré-utilisation de l'information d'adjacence des arêtes et des index

Comparaisons des performances



GraphX est environ 3x plus lent que GraphLab

GraphX passe à l'échelle sur de plus grands graphes



GraphX est environ 2x plus lent que GraphLab

» Surcoûts liés à Scala + Java: Lambdas, GC time, …

» Pas de parallélisation de la mémoire partgée: 2x plus de comm.

Conclusion et Observations

Vues spécifiques à un domaine: Tables et Graphes

» Les tables et les graphes sont les objects de base composables

» 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
```

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 destination

Graphe avec sommets/arêtes de différentes types

Pour un graphe donné, on peut avoir un/plusieurs types de sommets (VD) et un/plusieurs types d'arêtes (ED)

L'héritage permet de définir des nœuds avec différents types.

Par exemple pour un graphe biparti utilisateurs/produits:

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double) extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

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

Le graphe de propriétés correspond à deux RDD (pour les nœuds et les arcs)
La classe du Graphe contient les structures pour accéder aux sommets et aux arêtes du graphe

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Les classes `VertexRDD[VD]` et `EdgeRDD[ED]` étendent et sont des versions optimisées de `RDD[(VertexID,VD)]` et `RDD[Edge[ED]]`

`VertexRDD[VD]` et `EdgeRDD[ED]` fournissent des fonctionnalités supplémentaires pour les calculs de graphe

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 hérités de la classe RDD :

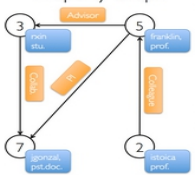
```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
// Revere the edges reusing both attributes and structure
def reverse: EdgeRDD[ED]
// Join two `EdgeRDD`s partitioned using the same partitioning strategy.
def innerJoin[ED2, ED3](other: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

Exemple de graphe de propriétés

Exemple du graphe de propriétés des collaborateurs du projet graphX:

- Les sommets contiennent le nom et la fonction
- Les arêtes annotées avec la nature de la collaboration

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Graphe obtenu:

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

Construction du graphe à partir d'une collection de RDD

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

Utilisation des vues

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

```
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
```

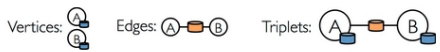
graph.vertices retourne un `VertexRDD[(String,String)]` qui étend `RDD[(VertexID,(String,String))]` => on peut utiliser le `case` pour décomposer le tuple (idem pour les arêtes)

On pouvait également utiliser le `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 y a 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



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(_))
```

Opérateurs de graphe

- Les graphes de propriétés ont une collection d'opérateurs de base produisant un nouveau graphe avec des propriétés et structures différentes
- Les opérateurs de base sont définis dans `Graph` et les opérateurs composés dans `GraphOps`
- Cependant grâce à *implicit* en Scala, les opérateurs de `GraphOps` sont disponibles dans `Graph`
- Ex: calcul du in-degree des différents sommets:

```
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

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]]
```

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
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[E
  : 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)

Opérateurs de transformation(2)

- Ces opérateurs sont souvent utilisés pour initialiser le graphe pour un calcul
- Ex: initialisation du graphe pour PageRank:

```
// Given a graph where the vertex property is the out degree
val inputGraph: Graph[Int, String] =
  graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
val outputGraph: Graph[Double, Double] =
  inputGraph.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 prédicats sur sommets et arêtes et retourne le graphe contenant les sommets satisfaisants les prédicats et reliés par les arêtes satisfaisant les prédicats
- `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

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

Opérateurs d'agrégation

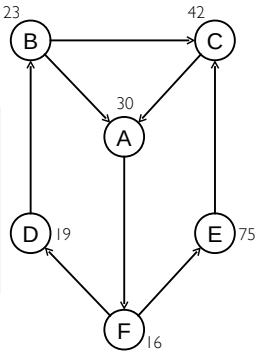
```
// Aggregate information about adjacent triplets ==================================
def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexID]]
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]
def aggregateMessages[Msg: ClassTag](
    sendMsg: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

aggregateMessage: applique une fonction sendMsg (≈map) définie par l'utilisateur à chaque triplet arête puis utilise mergeMsg (≈reduce) pour agréger ces messages pour le sommet destination

Exemple

Calculer le nombre de followers plus âgés pour chaque noeud, 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
)
```



Exemple: moyenne des âges des followers plus âgés

```
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// Create a graph with "age" as the vertex property.  Here we use a random graph for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDouble )
// Compute the number of older followers and their total age
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
)
// 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(_))
```

Opérateurs de calcul itératif graph-parallel

```
// 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 de graphe

```scala
// 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]
```

Références

- GraphX: Graph Processing in a Distributed Dataflow Framework :
- https://amplab.cs.berkeley.edu/wp-content/uploads/2014/09/graphx.pdf
- GraphX: Unifying Data-Parallel and Graph-Parallel Analytics :http://arxiv.org/pdf/1402.2394.pdf
- https://amplab.cs.berkeley.edu/wp-content/uploads/2014/02/graphx@strata2014_final.pdf
- http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph
- http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.GraphOps
- GraphX Programming Guide :
  http://spark.apache.org/docs/latest/graphx-programming-guide.html#vertex_and_edge_rdds