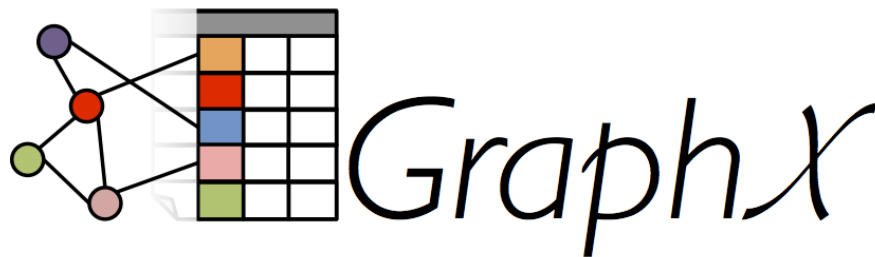


GraphX Programming Guide

- [Overview](#)
- [Getting Started](#)
- [The Property Graph](#)
 - [Example Property Graph](#)
- [Graph Operators](#)
 - [Summary List of Operators](#)
 - [Property Operators](#)
 - [Structural Operators](#)
 - [Join Operators](#)
 - [Neighborhood Aggregation](#)
 - [Aggregate Messages \(aggregateMessages\)](#)
 - [Map Reduce Triplets Transition Guide \(Legacy\)](#)
 - [Computing Degree Information](#)
 - [Collecting Neighbors](#)
 - [Caching and Uncaching](#)
- [Pregel API](#)
- [Graph Builders](#)
- [Vertex and Edge RDDs](#)
 - [VertexRDDs](#)
 - [EdgeRDDs](#)
- [Optimized Representation](#)
- [Graph Algorithms](#)
 - [PageRank](#)
 - [Connected Components](#)
 - [Triangle Counting](#)
- [Examples](#)



Overview

GraphX is a new component in Spark for graphs and graph-parallel computation. At a high level, GraphX extends the Spark [RDD](#) by introducing a new [Graph](#) abstraction: a directed multigraph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., [subgraph](#), [joinVertices](#), and [aggregateMessages](#)) as well as an optimized variant of the [Pregel](#) API. In addition, GraphX includes a growing collection of graph [algorithms](#) and [builders](#) to simplify graph analytics tasks.

Getting Started

To get started you first need to import Spark and GraphX into your project, as follows:

```
import org.apache.spark._
import org.apache.spark.graphx._
// To make some of the examples work we will also need RDD
import org.apache.spark.rdd.RDD
```

If you are not using the Spark shell you will also need a `SparkContext`. To learn more about getting started with Spark refer to the [Spark Quick Start Guide](#).

The Property Graph

The [property graph](#) is a directed multigraph with user defined objects attached to each vertex and edge. A directed multigraph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex. The ability to support parallel edges simplifies modeling scenarios where there can be multiple relationships (e.g., co-worker and friend) between the same vertices. Each vertex is keyed by a *unique* 64-bit long identifier (`VertexId`). GraphX does not impose any ordering constraints on the vertex identifiers. Similarly, edges have corresponding source and destination vertex identifiers.

The property graph is parameterized over the vertex (VD) and edge (ED) types. These are the types of the objects associated with each vertex and edge respectively.

GraphX optimizes the representation of vertex and edge types when they are primitive data types (e.g., int, double, etc...) reducing the in memory footprint by storing them in specialized arrays.

In some cases it may be desirable to have vertices with different property types in the same graph. This can be accomplished through inheritance. For example to model users and products as a bipartite graph we might do the following:

```
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
```

Like RDDs, property graphs are immutable, distributed, and fault-tolerant. Changes to the values or structure of the graph are accomplished by producing a new graph with the desired changes. Note that substantial parts of the original graph (i.e., unaffected structure, attributes, and indices) are reused in the new graph reducing the cost of this inherently functional data structure. The graph is partitioned across the executors using a range of vertex partitioning heuristics. As with RDDs, each partition of the graph can be recreated on a different machine in the event of a failure.

Logically the property graph corresponds to a pair of typed collections (RDDs) encoding the properties for each vertex and edge. As a consequence, the graph class contains members to access the vertices and edges of the graph:

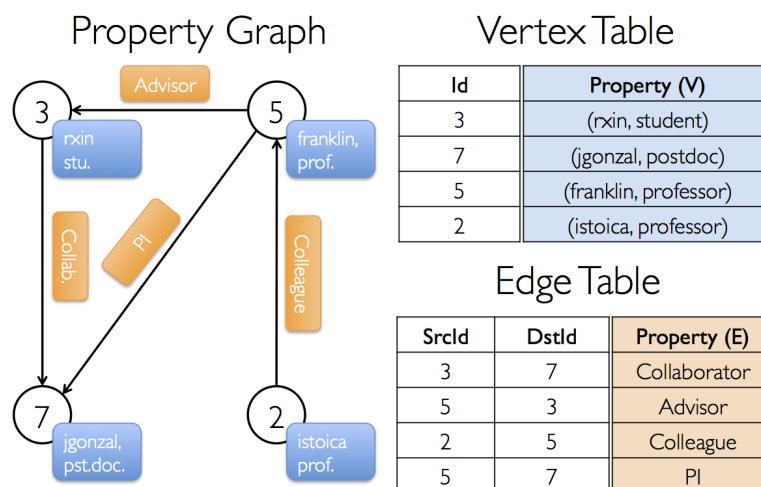
```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

The classes `VertexRDD[VD]` and `EdgeRDD[ED]` extend and are optimized versions of `RDD[(VertexId, VD)]` and `RDD[Edge[ED]]` respectively. Both `VertexRDD[VD]` and `EdgeRDD[ED]` provide additional functionality built around graph

computation and leverage internal optimizations. We discuss the VertexRDD [VertexRDD](#) and EdgeRDD [EdgeRDD](#) API in greater detail in the section on [vertex and edge RDDs](#) but for now they can be thought of as simply RDDs of the form: `RDD[(VertexId, VD)]` and `RDD[Edge[ED]]`.

Example Property Graph

Suppose we want to construct a property graph consisting of the various collaborators on the GraphX project. The vertex property might contain the username and occupation. We could annotate edges with a string describing the relationships between collaborators:



The resulting graph would have the type signature:

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

There are numerous ways to construct a property graph from raw files, RDDs, and even synthetic generators and these are discussed in more detail in the section on [graph builders](#). Probably the most general method is to use the [Graph object](#). For example the following code constructs a graph from a collection of RDDs:

```
// 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)
```

In the above example we make use of the [Edge](#) case class. Edges have a `srcId` and a `dstId` corresponding to the source and destination vertex identifiers. In addition, the `Edge` class has an `attr` member which stores the edge property.

We can deconstruct a graph into the respective vertex and edge views by using the `graph.vertices` and `graph.edges` members respectively.

```
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
```

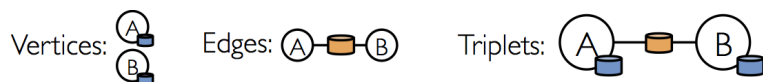
Note that `graph.vertices` returns an `VertexRDD[(String, String)]` which extends `RDD[(VertexId, (String, String))]` and so we use the scala `case` expression to deconstruct the tuple. On the other hand, `graph.edges` returns an `EdgeRDD` containing `Edge[String]` objects. We could have also used the case class type constructor as in the following:

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

In addition to the vertex and edge views of the property graph, GraphX also exposes a triplet view. The triplet view logically joins the vertex and edge properties yielding an `RDD[EdgeTriplet[VD, ED]]` containing instances of the `EdgeTriplet` class. This *join* can be expressed in the following SQL expression:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

or graphically as:



The `EdgeTriplet` class extends the `Edge` class by adding the `srcAttr` and `dstAttr` members which contain the source and destination properties respectively. We can use the triplet view of a graph to render a collection of strings describing relationships between users.

```
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(_))
```

Graph Operators

Just as RDDs have basic operations like `map`, `filter`, and `reduceByKey`, property graphs also have a collection of basic operators that take user defined functions and produce new graphs with transformed properties and structure. The core operators that have optimized implementations are defined in `Graph` and convenient operators that are expressed as a compositions of the core operators are defined in `GraphOps`. However, thanks to Scala implicits the operators in `GraphOps` are automatically available as members of `Graph`. For example, we can compute the in-degree of each vertex (defined in `GraphOps`) by the following:

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

The reason for differentiating between core graph operations and [GraphOps](#) is to be able to support different graph representations in the future. Each graph representation must provide implementations of the core operations and reuse many of the useful operations defined in [GraphOps](#).

Summary List of Operators

The following is a quick summary of the functionality defined in both [Graph](#) and [GraphOps](#) but presented as members of [Graph](#) for simplicity. Note that some function signatures have been simplified (e.g., default arguments and type constraints removed) and some more advanced functionality has been removed so please consult the API docs for the official list of operations.

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // 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]]
  // Functions for caching graphs =====
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // 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]
  // 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]
  // 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]
  // 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]
// 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]
}

```

Property Operators

Like the RDD map operator, the property graph contains the following:

```

class Graph[VD, ED] {
  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]
}

```

Each of these operators yields a new graph with the vertex or edge properties modified by the user defined map function.

Note that in each case the graph structure is unaffected. This is a key feature of these operators which allows the resulting graph to reuse the structural indices of the original graph. The following snippets are logically equivalent, but the first one does not preserve the structural indices and would not benefit from the GraphX system optimizations:

```

val newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }
val newGraph = Graph(newVertices, graph.edges)

```

Instead, use `mapVertices` to preserve the indices:

```

val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))

```

These operators are often used to initialize the graph for a particular computation or project away unnecessary properties. For example, given a graph with the out degrees as the vertex properties (we describe how to construct such a graph later), we initialize it for 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)
```

Structural Operators

Currently GraphX supports only a simple set of commonly used structural operators and we expect to add more in the future. The following is a list of the basic structural operators.

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
              vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
}
```

The `reverse` operator returns a new graph with all the edge directions reversed. This can be useful when, for example, trying to compute the inverse PageRank. Because the reverse operation does not modify vertex or edge properties or change the number of edges, it can be implemented efficiently without data movement or duplication.

The `subgraph` operator takes vertex and edge predicates and returns the graph containing only the vertices that satisfy the vertex predicate (evaluate to true) and edges that satisfy the edge predicate *and connect vertices that satisfy the vertex predicate*. The subgraph operator can be used in number of situations to restrict the graph to the vertices and edges of interest or eliminate broken links. For example in the following code we remove broken links:

```
// 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")),
                      (4L, ("peter", "student"))))

// 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"),
                      Edge(4L, 0L, "student"), Edge(5L, 0L, "colleague")))

// 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)

// Notice that there is a user 0 (for which we have no information) connected to users
// 4 (peter) and 5 (franklin).
graph.triplets.map(
  triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1
).collect.foreach(println(_))

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

// The valid subgraph will disconnect users 4 and 5 by removing user 0
validGraph.vertices.collect.foreach(println(_))
```



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

Note in the above example only the vertex predicate is provided. The `subgraph` operator defaults to `true` if the vertex or edge predicates are not provided.

The `mask` operator constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph. This can be used in conjunction with the `subgraph` operator to restrict a graph based on the properties in another related graph. For example, we might run connected components using the graph with missing vertices and then restrict the answer to the valid subgraph.

```
// 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)
```

The `groupEdges` operator merges parallel edges (i.e., duplicate edges between pairs of vertices) in the multigraph. In many numerical applications, parallel edges can be *added* (their weights combined) into a single edge thereby reducing the size of the graph.

Join Operators

In many cases it is necessary to join data from external collections (RDDs) with graphs. For example, we might have extra user properties that we want to merge with an existing graph or we might want to pull vertex properties from one graph into another. These tasks can be accomplished using the *join* operators. Below we list the key join operators:

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

The `joinVertices` operator joins the vertices with the input RDD and returns a new graph with the vertex properties obtained by applying the user defined map function to the result of the joined vertices. Vertices without a matching value in the RDD retain their original value.

Note that if the RDD contains more than one value for a given vertex only one will be used. It is therefore recommended that the input RDD be made unique using the following which will also *pre-index* the resulting values to substantially accelerate the subsequent join.

```
val nonUniqueCosts: RDD[(VertexId, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```


The more general `outerJoinVertices` behaves similarly to `joinVertices` except that the user defined `map` function is applied to all vertices and can change the vertex property type. Because not all vertices may have a matching value in the input RDD the `map` function takes an `Option` type. For example, we can setup a graph for PageRank by initializing vertex properties with their `outDegree`.

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

You may have noticed the multiple parameter lists (e.g., `f(a)(b)`) curried function pattern used in the above examples. While we could have equally written `f(a)(b)` as `f(a,b)` this would mean that type inference on `b` would not depend on `a`. As a consequence, the user would need to provide type annotation for the user defined function:

```
val joinedGraph = graph.joinVertices(uniqueCosts,
  (id: VertexId, oldCost: Double, extraCost: Double) => oldCost + extraCost)
```

Neighborhood Aggregation

A key step in many graph analytics tasks is aggregating information about the neighborhood of each vertex. For example, we might want to know the number of followers each user has or the average age of the the followers of each user. Many iterative graph algorithms (e.g., PageRank, Shortest Path, and connected components) repeatedly aggregate properties of neighboring vertices (e.g., current PageRank Value, shortest path to the source, and smallest reachable vertex id).

To improve performance the primary aggregation operator changed from `graph.mapReduceTriplets` to the new `graph.AggregateMessages`. While the changes in the API are relatively small, we provide a transition guide below.

Aggregate Messages (aggregateMessages)

The core aggregation operation in GraphX is `aggregateMessages`. This operator applies a user defined `sendMsg` function to each *edge triplet* in the graph and then uses the `mergeMsg` function to aggregate those messages at their destination vertex.

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

The user defined `sendMsg` function takes an `EdgeContext`, which exposes the source and destination attributes along with the edge attribute and functions (`sendToSrc`, and `sendToDst`) to send messages to the source and destination attributes. Think of `sendMsg` as the *map* function in map-reduce. The user defined `mergeMsg` function takes two messages destined to the same vertex and yields a single message. Think of `mergeMsg` as the *reduce* function in map-reduce. The `aggregateMessages` operator returns a `VertexRDD[Msg]` containing the aggregate message (of type `Msg`) destined to each vertex. Vertices that did not receive a message are not included in the returned `VertexRDD`.

In addition, `aggregateMessages` takes an optional `tripletsFields` which indicates what data is accessed in the `EdgeContext` (i.e., the source vertex attribute but not the destination vertex attribute). The possible options for the `tripletsFields` are defined in `TripletFields` and the default value is `TripletFields.All` which indicates that the user defined `sendMsg` function may access any of the fields in the `EdgeContext`. The `tripletFields` argument can be used to notify GraphX that only part of the `EdgeContext` will be needed allowing GraphX to select an optimized join strategy. For example if we are computing the average age of the followers of each user we would only require the source field and so we would use `TripletFields.Src` to indicate that we only require the source field

In earlier versions of GraphX we used byte code inspection to infer the `TripletFields` however we have found that bytecode inspection to be slightly unreliable and instead opted for more explicit user control.

In the following example we use the `aggregateMessages` operator to compute the average age of the more senior followers of each user.

```
import org.apache.spark.graphx.{Graph, VertexRDD}
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(_))
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/graphx/AggregateMessagesExample.scala" in the Spark repo.

The `aggregateMessages` operation performs optimally when the messages (and the sums of messages) are constant sized (e.g., floats and addition instead of lists and concatenation).

Map Reduce Triplets Transition Guide (Legacy)

In earlier versions of GraphX neighborhood aggregation was accomplished using the `mapReduceTriplets` operator:

```
class Graph[VD, ED] {  
  def mapReduceTriplets[Msg](  
    map: EdgeTriplet[VD, ED] => Iterator[(VertexId, Msg)],  
    reduce: (Msg, Msg) => Msg  
  ): VertexRDD[Msg]  
}
```

The `mapReduceTriplets` operator takes a user defined map function which is applied to each triplet and can yield *messages* which are aggregated using the user defined `reduce` function. However, we found the user of the returned iterator to be expensive and it inhibited our ability to apply additional optimizations (e.g., local vertex renumbering). In `aggregateMessages` we introduced the `EdgeContext` which exposes the triplet fields and also functions to explicitly send messages to the source and destination vertex. Furthermore we removed bytecode inspection and instead require the user to indicate what fields in the triplet are actually required.

The following code block using `mapReduceTriplets`:

```
val graph: Graph[Int, Float] = ...  
def msgFun(triplet: Triplet[Int, Float]): Iterator[(Int, String)] = {  
  Iterator((triplet.dstId, "Hi"))  
}  
def reduceFun(a: String, b: String): String = a + " " + b  
val result = graph.mapReduceTriplets[String](msgFun, reduceFun)
```

can be rewritten using `aggregateMessages` as:

```
val graph: Graph[Int, Float] = ...  
def msgFun(triplet: EdgeContext[Int, Float, String]) {  
  triplet.sendToDst("Hi")  
}  
def reduceFun(a: String, b: String): String = a + " " + b  
val result = graph.aggregateMessages[String](msgFun, reduceFun)
```

Computing Degree Information

A common aggregation task is computing the degree of each vertex: the number of edges adjacent to each vertex. In the context of directed graphs it is often necessary to know the in-degree, out-degree, and the total degree of each vertex. The `GraphOps` class contains a collection of operators to compute the degrees of each vertex. For example in the following we compute the max in, out, and total degrees:

```
// 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)
```

Collecting Neighbors

In some cases it may be easier to express computation by collecting neighboring vertices and their attributes at each vertex. This can be easily accomplished using the `collectNeighborIds` and the `collectNeighbors` operators.

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

These operators can be quite costly as they duplicate information and require substantial communication. If possible try expressing the same computation using the `aggregateMessages` operator directly.

Caching and Uncaching

In Spark, RDDs are not persisted in memory by default. To avoid recomputation, they must be explicitly cached when using them multiple times (see the [Spark Programming Guide](#)). Graphs in GraphX behave the same way. **When using a graph multiple times, make sure to call `Graph.cache()` on it first.**

In iterative computations, *uncaching* may also be necessary for best performance. By default, cached RDDs and graphs will remain in memory until memory pressure forces them to be evicted in LRU order. For iterative computation, intermediate results from previous iterations will fill up the cache. Though they will eventually be evicted, the unnecessary data stored in memory will slow down garbage collection. It would be more efficient to uncache intermediate results as soon as they are no longer necessary. This involves materializing (caching and forcing) a graph or RDD every iteration, uncaching all other datasets, and only using the materialized dataset in future iterations. However, because graphs are composed of multiple RDDs, it can be difficult to unpersist them correctly. **For iterative computation we recommend using the Pregel API, which correctly unpersists intermediate results.**

Pregel API

Graphs are inherently recursive data structures as properties of vertices depend on properties of their neighbors which in turn depend on properties of *their* neighbors. As a consequence many important graph algorithms iteratively recompute the properties of each vertex until a fixed-point condition is reached. A range of graph-parallel abstractions have been proposed to express these iterative algorithms. GraphX exposes a variant of the Pregel API.

At a high level the Pregel operator in GraphX is a bulk-synchronous parallel messaging abstraction *constrained to the topology of the graph*. The Pregel operator executes in a series of super steps in which vertices receive the *sum* of their inbound messages from the previous super step, compute a new value for the vertex property, and then send messages to neighboring vertices in the next super step. Unlike Pregel, messages are computed in parallel as a function of the edge triplet and the message computation has access to both the source and destination vertex attributes. Vertices that do not receive a message are skipped within a super step. The Pregel operators terminates iteration and returns the final graph when there are no messages remaining.

Note, unlike more standard Pregel implementations, vertices in GraphX can only send messages to neighboring vertices and the message construction is done in parallel using a user defined messaging

function. These constraints allow additional optimization within GraphX.

The following is the type signature of the [Pregel operator](#) as well as a *sketch* of its implementation (note: to avoid `stackOverflowError` due to long lineage chains, `pregel` support periodically checkpoint graph and messages by setting `"spark.graphx.pregel.checkpointInterval"` to a positive number, say 10. And set checkpoint directory as well using `SparkContext.setCheckpointDir(directory: String)`):

```
class GraphOps[VD, ED] {
  def pregel[A]
    (initialMsg: A,
     maxIter: Int = Int.MaxValue,
     activeDir: EdgeDirection = EdgeDirection.Out)
    (vprog: (VertexId, VD, A) => VD,
     sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
     mergeMsg: (A, A) => A)
  : Graph[VD, ED] = {
    // Receive the initial message at each vertex
    var g = mapVertices( (vid, vdata) => vprog(vid, vdata, initialMsg) ).cache()

    // compute the messages
    var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // Loop until no messages remain or maxIterations is achieved
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // Receive the messages and update the vertices.
      g = g.joinVertices(messages)(vprog).cache()
      val oldMessages = messages
      // Send new messages, skipping edges where neither side received a message. We must cache
      // messages so it can be materialized on the next line, allowing us to uncache the previous
      // iteration.
      messages = GraphXUtils.mapReduceTriplets(
        g, sendMsg, mergeMsg, Some((oldMessages, activeDirection))).cache()
      activeMessages = messages.count()
      i += 1
    }
    g
  }
}
```

Notice that `Pregel` takes two argument lists (i.e., `graph.pregel(list1)(list2)`). The first argument list contains configuration parameters including the initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges). The second argument list contains the user defined functions for receiving messages (the vertex program `vprog`), computing messages (`sendMsg`), and combining messages `mergeMsg`.

We can use the `Pregel` operator to express computation such as single source shortest path in the following example.

```
import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.graphx.util.GraphGenerators

// A graph with edge attributes containing distances
val graph: Graph[Long, Double] =
```

```

GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // 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"))

```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/graphx/SSSPExample.scala" in the Spark repo.

Graph Builders

GraphX provides several ways of building a graph from a collection of vertices and edges in an RDD or on disk. None of the graph builders repartitions the graph's edges by default; instead, edges are left in their default partitions (such as their original blocks in HDFS). [Graph.groupEdges](#) requires the graph to be repartitioned because it assumes identical edges will be colocated on the same partition, so you must call [Graph.partitionBy](#) before calling `groupEdges`.

```

object GraphLoader {
  def edgeListFile(
    sc: SparkContext,
    path: String,
    canonicalOrientation: Boolean = false,
    minEdgePartitions: Int = 1)
    : Graph[Int, Int]
}

```

[GraphLoader.edgeListFile](#) provides a way to load a graph from a list of edges on disk. It parses an adjacency list of (source vertex ID, destination vertex ID) pairs of the following form, skipping comment lines that begin with #:

```

# This is a comment
2 1
4 1
1 2

```

It creates a `Graph` from the specified edges, automatically creating any vertices mentioned by edges. All vertex and edge attributes default to 1. The `canonicalOrientation` argument allows reorienting edges in the positive direction (`srcId < dstId`), which is required by the [connected components](#) algorithm. The `minEdgePartitions` argument specifies the minimum number of edge partitions to generate; there may be more edge partitions than specified if, for example, the HDFS file has more blocks.

```
object Graph {
  def apply[VD, ED](
    vertices: RDD[(VertexId, VD)],
    edges: RDD[Edge[ED]],
    defaultVertexAttr: VD = null)
    : Graph[VD, ED]

  def fromEdges[VD, ED](
    edges: RDD[Edge[ED]],
    defaultValue: VD): Graph[VD, ED]

  def fromEdgeTuples[VD](
    rawEdges: RDD[(VertexId, VertexId)],
    defaultValue: VD,
    uniqueEdges: Option[PartitionStrategy] = None): Graph[VD, Int]
}
```

`Graph.apply` allows creating a graph from RDDs of vertices and edges. Duplicate vertices are picked arbitrarily and vertices found in the edge RDD but not the vertex RDD are assigned the default attribute.

`Graph.fromEdges` allows creating a graph from only an RDD of edges, automatically creating any vertices mentioned by edges and assigning them the default value.

`Graph.fromEdgeTuples` allows creating a graph from only an RDD of edge tuples, assigning the edges the value 1, and automatically creating any vertices mentioned by edges and assigning them the default value. It also supports deduplicating the edges; to deduplicate, pass Some of a `PartitionStrategy` as the `uniqueEdges` parameter (for example, `uniqueEdges = Some(PartitionStrategy.RandomVertexCut)`). A partition strategy is necessary to colocate identical edges on the same partition so they can be deduplicated.

Vertex and Edge RDDs

GraphX exposes RDD views of the vertices and edges stored within the graph. However, because GraphX maintains the vertices and edges in optimized data structures and these data structures provide additional functionality, the vertices and edges are returned as `VertexRDD` `VertexRDD` and `EdgeRDD` `EdgeRDD` respectively. In this section we review some of the additional useful functionality in these types. Note that this is just an incomplete list, please refer to the API docs for the official list of operations.

VertexRDDs

The `VertexRDD[A]` extends `RDD[(VertexId, A)]` and adds the additional constraint that each `VertexId` occurs only *once*. Moreover, `VertexRDD[A]` represents a *set* of vertices each with an attribute of type A. Internally, this is achieved by storing the vertex attributes in a reusable hash-map data-structure. As a consequence if two `VertexRDDs` are derived from the same base `VertexRDD` `VertexRDD` (e.g., by `filter` or `mapValues`) they can be joined in constant time without hash evaluations. To leverage this indexed data structure, the `VertexRDD` `VertexRDD` exposes the following additional functionality:

```
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]
// Show only vertices unique to this set based on their VertexId's
def minus(other: RDD[(VertexId, VD)])
// 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]
}
```

Notice, for example, how the filter operator returns an `VertexRDD`. Filter is actually implemented using a `BitSet` thereby reusing the index and preserving the ability to do fast joins with other `VertexRDD`s. Likewise, the `mapValues` operators do not allow the map function to change the `VertexId` thereby enabling the same `HashMap` data structures to be reused. Both the `leftJoin` and `innerJoin` are able to identify when joining two `VertexRDD`s derived from the same `HashMap` and implement the join by linear scan rather than costly point lookups.

The `aggregateUsingIndex` operator is useful for efficient construction of a new `VertexRDD` from an `RDD[(VertexId, A)]`. Conceptually, if I have constructed a `VertexRDD[B]` over a set of vertices, *which is a super-set* of the vertices in some `RDD[(VertexId, A)]` then I can reuse the index to both aggregate and then subsequently index the `RDD[(VertexId, A)]`. For example:

```
val setA: VertexRDD[Int] = VertexRDD(sc.parallelize(0L until 100L).map(id => (id, 1)))
val rddB: RDD[(VertexId, Double)] = sc.parallelize(0L until 100L).flatMap(id => List((id, 1.0), (id, 2.0)))
// There should be 200 entries in rddB
rddB.count
val setB: VertexRDD[Double] = setA.aggregateUsingIndex(rddB, _ + _)
// There should be 100 entries in setB
setB.count
// Joining A and B should now be fast!
val setC: VertexRDD[Double] = setA.innerJoin(setB)((id, a, b) => a + b)
```

EdgeRDDs

The `EdgeRDD[ED]`, which extends `RDD[Edge[ED]]` organizes the edges in blocks partitioned using one of the various partitioning strategies defined in `PartitionStrategy`. Within each partition, edge attributes and adjacency structure, are stored separately enabling maximum reuse when changing attribute values.

The three additional functions exposed by the `EdgeRDD` are:

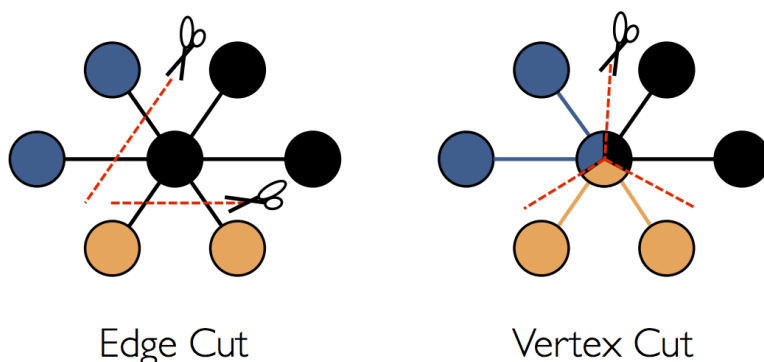
```
// Transform the edge attributes while preserving the structure
def mapValues[ED2](f: Edge[ED] => ED2): EdgeRDD[ED2]
// Reverse the edges reusing both attributes and structure
def reverse: EdgeRDD[ED]
```

```
// 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]
```

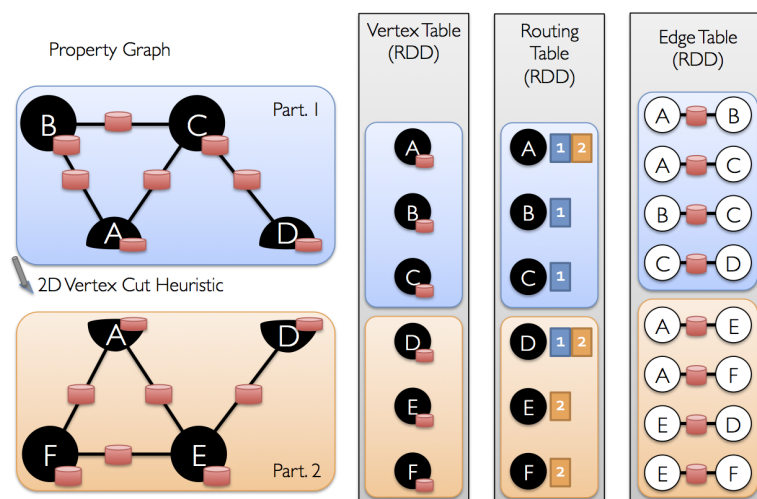
In most applications we have found that operations on the `EdgeRDD` are accomplished through the graph operators or rely on operations defined in the base RDD class.

Optimized Representation

While a detailed description of the optimizations used in the GraphX representation of distributed graphs is beyond the scope of this guide, some high-level understanding may aid in the design of scalable algorithms as well as optimal use of the API. GraphX adopts a vertex-cut approach to distributed graph partitioning:



Rather than splitting graphs along edges, GraphX partitions the graph along vertices which can reduce both the communication and storage overhead. Logically, this corresponds to assigning edges to machines and allowing vertices to span multiple machines. The exact method of assigning edges depends on the `PartitionStrategy` and there are several tradeoffs to the various heuristics. Users can choose between different strategies by repartitioning the graph with the `Graph.partitionBy` operator. The default partitioning strategy is to use the initial partitioning of the edges as provided on graph construction. However, users can easily switch to 2D-partitioning or other heuristics included in GraphX.



Once the edges have been partitioned the key challenge to efficient graph-parallel computation is efficiently joining vertex attributes with the edges. Because real-world graphs typically have more edges than vertices, we move vertex attributes to the edges. Because not all partitions will contain edges adjacent to all vertices we internally maintain a routing table which identifies where to broadcast vertices when implementing the join required for operations like `triplets` and `aggregateMessages`.

Graph Algorithms

GraphX includes a set of graph algorithms to simplify analytics tasks. The algorithms are contained in the `org.apache.spark.graphx.lib` package and can be accessed directly as methods on `Graph` via `GraphOps`. This section

describes the algorithms and how they are used.

PageRank

PageRank measures the importance of each vertex in a graph, assuming an edge from u to v represents an endorsement of v 's importance by u . For example, if a Twitter user is followed by many others, the user will be ranked highly.

GraphX comes with static and dynamic implementations of PageRank as methods on the [PageRank object](#). Static PageRank runs for a fixed number of iterations, while dynamic PageRank runs until the ranks converge (i.e., stop changing by more than a specified tolerance). [GraphOps](#) allows calling these algorithms directly as methods on Graph.

GraphX also includes an example social network dataset that we can run PageRank on. A set of users is given in `data/graphx/users.txt`, and a set of relationships between users is given in `data/graphx/followers.txt`. We compute the PageRank of each user as follows:

```
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"))
```

Find full example code at `"examples/src/main/scala/org/apache/spark/examples/graphx/PageRankExample.scala"` in the Spark repo.

Connected Components

The connected components algorithm labels each connected component of the graph with the ID of its lowest-numbered vertex. For example, in a social network, connected components can approximate clusters. GraphX contains an implementation of the algorithm in the [ConnectedComponents object](#), and we compute the connected components of the example social network dataset from the [PageRank section](#) as follows:

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

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
```

```

}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))

```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/graphx/ConnectedComponentsExample.scala" in the Spark repo.

Triangle Counting

A vertex is part of a triangle when it has two adjacent vertices with an edge between them. GraphX implements a triangle counting algorithm in the [TriangleCount object](#) that determines the number of triangles passing through each vertex, providing a measure of clustering. We compute the triangle count of the social network dataset from the [PageRank section](#). Note that *TriangleCount* requires the edges to be in canonical orientation (*srcId* < *dstId*) and the graph to be partitioned using [Graph.partitionBy](#).

```

import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))

```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/graphx/TriangleCountingExample.scala" in the Spark repo.

Examples

Suppose I want to build a graph from some text files, restrict the graph to important relationships and users, run page-rank on the sub-graph, and then finally return attributes associated with the top users. I can do all of this in just a few lines with GraphX:

```

import org.apache.spark.graphx.GraphLoader

// Load my user data and parse into tuples of user id and attribute list
val users = (sc.textFile("data/graphx/users.txt")
  .map(line => line.split(",")).map( parts => (parts.head.toLong, parts.tail) ))

// Parse the edge data which is already in userId -> userId format

```

```

val followerGraph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")

// Attach the user attributes
val graph = followerGraph.outerJoinVertices(users) {
  case (uid, deg, Some(attrList)) => attrList
  // Some users may not have attributes so we set them as empty
  case (uid, deg, None) => Array.empty[String]
}

// Restrict the graph to users with usernames and names
val subgraph = graph.subgraph(vpred = (vid, attr) => attr.size == 2)

// Compute the PageRank
val pagerankGraph = subgraph.pageRank(0.001)

// Get the attributes of the top pagerank users
val userInfoWithPageRank = subgraph.outerJoinVertices(pagerankGraph.vertices) {
  case (uid, attrList, Some(pr)) => (pr, attrList.toList)
  case (uid, attrList, None) => (0.0, attrList.toList)
}

println(userInfoWithPageRank.vertices.top(5)(Ordering.by(_._2._1)).mkString("\n"))

```

Find full example code at ["examples/src/main/scala/org/apache/spark/examples/graphx/ComprehensiveExample.scala"](#) in the Spark repo.