

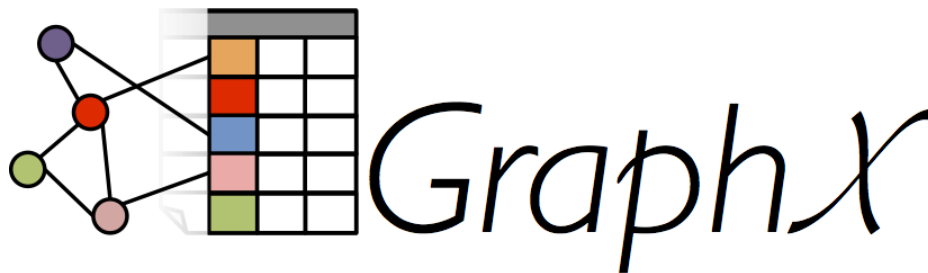


## Hands-on Exercises

☰ Graph Analytics With GraphX ▼

### In This Chapter

- 1. Background on Graph-Parallel Computation (Optional)
- 2. Introduction to the GraphX API
  - 2.1. The Property Graph
  - 2.2. Graph Views
- 3. Graph Operators
  - 3.1. The Map Reduce Triplets Operator
  - 3.2. Subgraph
- 4. Constructing an End-to-End Graph Analytics Pipeline on Real Data
  - 4.1. Getting Started (Again)
  - 4.2. Load the Wikipedia Articles
  - 4.3. Look at the First Article
  - 4.4. Clean the Data
  - 4.5. Making a Vertex RDD
  - 4.6. Making the Edge RDD
  - 4.7. Making the Graph
  - 4.8. Running PageRank on Wikipedia



GraphX is the new (alpha) Spark API for graphs (e.g., Web-Graphs and Social Networks) and graph-parallel computation (e.g., PageRank and Collaborative Filtering). At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: 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

([\(\[\\(\\[\\\(`mapFunc:org.apache.spark.graphx.EdgeTriplet\\\[VD,ED\\\]>Iterator\\\[\\\(org.apache.spark.graphx.VertexId,A\\\)\\\],reduceFunc:`\\]\\(http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph@mapReduceTriplets\\[A\\]\\(mapFunc:org.apache.spark.graphx.EdgeTriplet\\[VD,ED\\]>Iterator\\[\\(org.apache.spark.graphx.VertexId,A\\)\\],reduceFunc:\\(A,A\\)>A,activeSetOpt:Option\\[\\(org.apache.spark.graphx.VertexRDD\\[\\_\\],org.apache.spark.graphx.EdgeDirection\\)\\]\\)</a>\\)</p>
</div>
<div data-bbox=\\)\]\(http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.GraphOps@joinVertices\[U\]\(RDD\[\(VertexId,U\)\]\)\(\(VertexId,VD,U\)>VD\)\(ClassTag\[U\]\):Graph\[VD,ED\]\)</a>\), and <code>mapReduceTriplets</code></p>
</div>
<div data-bbox=\)](http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph@subgraph((EdgeTriplet[VD,ED])=>Boolean,(VertexId,VD)>Boolean):Graph[VD,ED])</a>, <code>joinVertices</code></p>
</div>
<div data-bbox=)

`(A,A)>A,activeSetOpt:Option[(org.apache.spark.graphx.VertexRDD[_],org.apache.spark.graphx.EdgeDirection)])`)

(`implicitevidence$10:scala.reflect.ClassTag[A]):org.apache.spark.graphx.VertexRDD[A]`)) as well as an optimized variant of the Pregel

(<http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html#pregel>) API. In addition, GraphX includes a growing collection

of graph algorithms ([http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html#graph\\_algorithms](http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html#graph_algorithms)) and builders

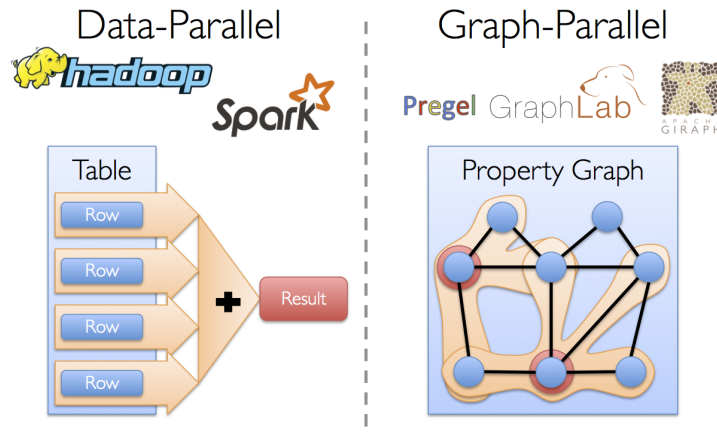
([http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html#graph\\_builders](http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html#graph_builders)) to simplify graph analytics tasks.

In this chapter we use GraphX to analyze Wikipedia data and implement graph algorithms in Spark. The GraphX API is currently only available in Scala but we plan to provide Java and Python bindings in the future.

## 1. Background on Graph-Parallel Computation (Optional)

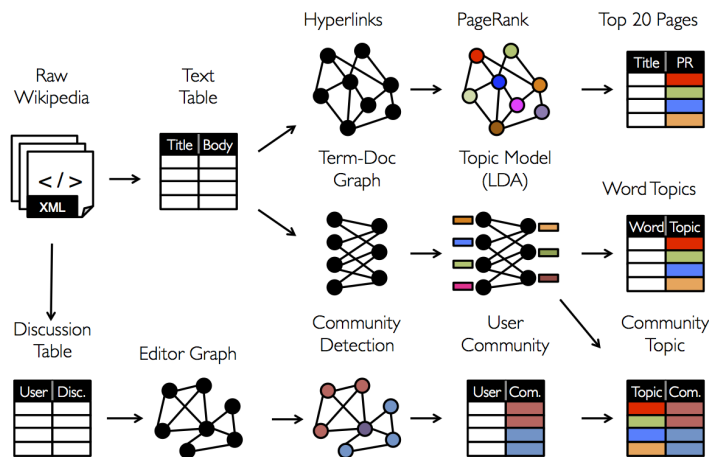
If you want to get started coding right away, you can skip this part or come back later.

From social networks to language modeling, the growing scale and importance of graph data has driven the development of numerous new *graph-parallel* systems (e.g., Giraph (<http://giraph.apache.org>) and GraphLab (<http://graphlab.org>)). By restricting the types of computation that can be expressed and introducing new techniques to partition and distribute graphs, these systems can efficiently execute sophisticated graph algorithms orders of magnitude faster than more general *data-parallel* systems.

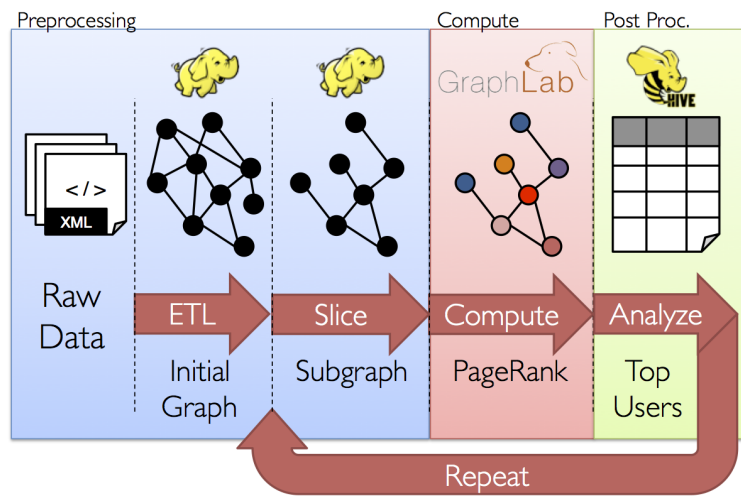


The same restrictions that enable graph-parallel systems to achieve substantial performance gains also limit their ability to express many of the important stages in a typical graph-analytics pipeline. Moreover while graph-parallel systems are optimized for iterative diffusion algorithms like PageRank they are not well suited to more basic tasks like constructing the graph, modifying its structure, or expressing computation that spans multiple graphs.

These tasks typically require data-movement outside of the graph topology and are often more naturally expressed as operations on tables in more traditional data-parallel systems like Map-Reduce. Furthermore, how we look at data depends on our objectives and the same raw data may require many different table and graph views throughout the analysis process:



Moreover, it is often desirable to be able to move between table and graph views of the same physical data and to leverage the properties of each view to easily and efficiently express computation. However, existing graph analytics pipelines compose graph-parallel and data-parallel systems, leading to extensive data movement and duplication and a complicated programming model.



The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API. The GraphX API enables users to view data both as graphs and as collections (i.e., RDDs) without data movement or duplication. By incorporating recent advances in graph-parallel systems, GraphX is able to optimize the execution of graph operations.

## 2. Introduction to the GraphX API

To get started you first need to import GraphX. Start the Spark-Shell (by running the following on the root node):

```
/root/spark/bin/spark-shell
```

and paste the following in your Spark shell:

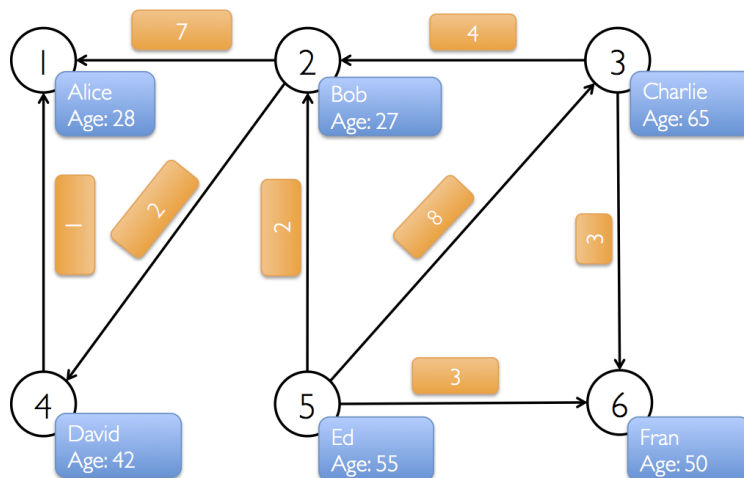


```
1 import org.apache.spark.graphx._
2 import org.apache.spark.rdd.RDD
```

### 2.1. The Property Graph

The property graph (<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph>) is a directed multigraph (a directed graph with potentially multiple parallel edges sharing the same source and destination vertex) with properties attached to each vertex and edge. Each vertex is keyed by a *unique* 64-bit long identifier (VertexID). Similarly, edges have corresponding source and destination vertex identifiers. The properties are stored as Scala/Java objects with each edge and vertex in the graph.

Throughout the first half of this tutorial we will use the following toy property graph. While this is hardly *big data*, it provides an opportunity to learn about the graph data model and the GraphX API. In this example we have a small social network with users and their ages modeled as vertices and likes modeled as directed edges.



We begin by creating the property graph from arrays of vertices and edges. Later we will demonstrate how to load real data. Paste the following code into the spark shell.



```
1 val vertexArray = Array(  
2   (1L, ("Alice", 28)),  
3   (2L, ("Bob", 27)),  
4   (3L, ("Charlie", 65)),  
5   (4L, ("David", 42)),  
6   (5L, ("Ed", 55)),  
7   (6L, ("Fran", 50))  
8 )  
9 val edgeArray = Array(  
10  Edge(2L, 1L, 7),  
11  Edge(2L, 4L, 2),  
12  Edge(3L, 2L, 4),  
13  Edge(3L, 6L, 3),  
14  Edge(4L, 1L, 1),  
15  Edge(5L, 2L, 2),  
16  Edge(5L, 3L, 8),  
17  Edge(5L, 6L, 3)  
18 )
```

Here we use the Edge (<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Edge>) 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 (in this case the number of likes).

Using `sc.parallelize` (introduced in the Spark tutorial) construct the following RDDs from the `vertexArray` and `edgeArray` variables.

We have made the code blocks editable so that you have space to compose your answer before pasting into the spark shell.



```
1 val vertexRDD: RDD[(Long, (String, Int))] = // Implement  
2 val edgeRDD: RDD[Edge[Int]] = // Implement
```

View Solution

```
1 val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)  
2 val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
```

Now we are ready to build a property graph. The basic property graph constructor takes an RDD of vertices (with type `RDD[(VertexId, V)]`) and an RDD of edges (with type `RDD[Edge[E]]`) and builds a graph (with type `Graph[V, E]`). Try the following:



```
1 val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
```

The vertex property for this graph is a tuple `(String, Int)` corresponding to the *User Name* and *Age* and the edge property is just an `Int` corresponding to the number of *Likes* in our hypothetical social network.

There are numerous ways to construct a property graph from raw files, RDDs, and even synthetic generators. 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. As with RDDs, each partition of the graph can be recreated on a different machine in the event of a failure.

## 2.2. Graph Views

In many cases we will want to extract the vertex and edge RDD views of a graph (e.g., when aggregating or saving the result of calculation). As a consequence, the graph class contains members (`graph.vertices` and `graph.edges`) to access the vertices and edges of the graph. While these members extend `RDD[(VertexId, V)]` and `RDD[Edge[E]]` they are actually backed by optimized representations that leverage the internal GraphX representation of graph data.

Use `graph.vertices` to display the names of the users that are at least 30 years old. The output should contain (in addition to lots of log messages):

```
David is 42
Fran is 50
Ed is 55
Charlie is 65
```

Here is a hint:



```
1 graph.vertices.filter {
2   case (id, (name, age)) => /* implement */
3 }.collect.foreach {
4   case (id, (name, age)) => /* implement */
5 }
```

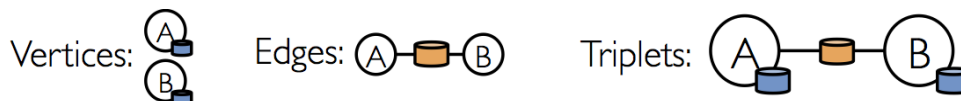
#### View Solution

```
1 // Solution 1
2 graph.vertices.filter { case (id, (name, age)) => age > 30 }.collect.foreach {
3   case (id, (name, age)) => println(s"$name is $age")
4 }
5
6 // Solution 2
7 graph.vertices.filter(v => v._2._2 > 30).collect.foreach(v => println(s"${v._2._1} is ${v._2._2}"))
8
9 // Solution 3
10 for ((id,(name,age)) <- graph.vertices.filter { case (id,(name,age)) => age > 30 }.collect) {
11   println(s"$name is $age")
12 }
```

Here we use the new String Interpolation feature in Scala 2.10:

```
1 val name = "Joey"
2 println(s"$name is ${ 3 * 10 }")
```

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` (<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.EdgeTriplet>) class. This *join* can be expressed graphically as:



The `EdgeTriplet` (<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.EdgeTriplet>) class extends the `Edge` (<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Edge>) class by adding the `srcAttr` and `dstAttr` members which contain the source and destination properties respectively.

Use the `graph.triplets` view to display who likes who. The output should look like:

Bob likes Alice  
Bob likes David  
Charlie likes Bob  
Charlie likes Fran  
David likes Alice  
Ed likes Bob  
Ed likes Charlie  
Ed likes Fran

Here is a partial solution:



```
1 for (triplet <- graph.triplets.collect) {  
2   /**  
3    * Triplet has the following Fields:  
4    *   triplet.srcAttr: (String, Int) // triplet.srcAttr._1 is the name  
5    *   triplet.dstAttr: (String, Int)  
6    *   triplet.attr: Int  
7    *   triplet.srcId: VertexId  
8    *   triplet.dstId: VertexId  
9    */  
10 }
```

✔ View Solution

```
1 for (triplet <- graph.triplets.collect) {  
2   println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")  
3 }
```

If someone likes someone else more than 5 times than that relationship is getting pretty serious. For extra credit, find the lovers.



✔ View Solution

```
1 for (triplet <- graph.triplets.filter(t => t.attr > 5).collect) {  
2   println(s"${triplet.srcAttr._1} loves ${triplet.dstAttr._1}")  
3 }
```

### 3. Graph Operators

Just as RDDs have basic operations like count, map, filter, and reduceByKey, property graphs also have a collection of basic operations. The following is a list of some of the many functions exposed by the Graph API.



```

1  /** Summary of the functionality in the property graph */
2  class Graph[VD, ED] {
3    // Information about the Graph
4    val numEdges: Long
5    val numVertices: Long
6    val inDegrees: VertexRDD[Int]
7    val outDegrees: VertexRDD[Int]
8    val degrees: VertexRDD[Int]
9
10   // Views of the graph as collections
11   val vertices: VertexRDD[VD]
12   val edges: EdgeRDD[ED]
13   val triplets: RDD[EdgeTriplet[VD, ED]]
14
15   // Change the partitioning heuristic
16   def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
17
18   // Transform vertex and edge attributes
19   def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
20   def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
21   def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
22   def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
23
24   // Modify the graph structure
25   def reverse: Graph[VD, ED]
26   def subgraph(
27     epred: EdgeTriplet[VD, ED] => Boolean = (x => true),
28     vpred: (VertexID, VD) => Boolean = ((v, d) => true))
29     : Graph[VD, ED]
30   def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
31
32   // Join RDDs with the graph
33   def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph[VD, ED]
34   def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])(
35     mapFunc: (VertexID, VD, Option[U]) => VD2)
36     : Graph[VD2, ED]
37
38   // Aggregate information about adjacent triplets
39   def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]
40   def mapReduceTriplets[A: ClassTag](
41     mapFunc: EdgeTriplet[VD, ED] => Iterator[(VertexID, A)],
42     reduceFunc: (A, A) => A)
43     : VertexRDD[A]
44
45   // Iterative graph-parallel computation
46   def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
47     vprog: (VertexID, VD, A) => VD,
48     sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID, A)],
49     mergeMsg: (A, A) => A)
50     : Graph[VD, ED]
51
52   // Basic graph algorithms
53   def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
54   def connectedComponents(): Graph[VertexID, ED]
55   def triangleCount(): Graph[Int, ED]
56   def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
57 }

```

These functions are split between Graph

(<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph>) and GraphOps

(<http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.GraphOps>). However, thanks to the “magic” of 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:



```
1 val inDegrees: VertexRDD[Int] = graph.inDegrees
```

In the above example the `graph.inDegrees` operators returned a `VertexRDD[Int]` (recall that this behaves like `RDD[(VertexId, Int)]`). What if we wanted to incorporate the in and out degree of each vertex into the vertex property? To do this we will use a set of common graph operators.

First we define a User class to better organize the vertex property and build a new graph with the user property (paste the following code into the spark shell):



```

1 // Define a class to more clearly model the user property
2 case class User(name: String, age: Int, inDeg: Int, outDeg: Int)
3 // Create a user Graph
4 val initialUserGraph: Graph[User, Int] = graph.mapVertices{ case (id, (name, age)) => User(name, age, 0, 0) }

```

Notice that we initialized each vertex with 0 in and out degree. Now we join the in and out degree information with each vertex building the new vertex property (paste the following code into the spark shell):



```

1 // Fill in the degree information
2 val userGraph = initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {
3   case (id, u, inDegOpt) => User(u.name, u.age, inDegOpt.getOrElse(0), u.outDeg)
4 }.outerJoinVertices(initialUserGraph.outDegrees) {
5   case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg, outDegOpt.getOrElse(0))
6 }

```

Here we use the `outerJoinVertices` method of `Graph` which has the following (confusing) type signature:



```

1 def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
2   (mapFunc: (VertexID, VD, Option[U]) => VD2)
3   : Graph[VD2, ED]

```

Notice that `outerJoinVertices` takes *two* argument lists. The first contains an RDD of vertex values and the second argument list takes a function from the id, attribute, and Optional matching value in the RDD to a new vertex value. Note that it is possible that the input RDD may not contain values for some of the vertices in the graph. In these cases the `Option` argument is empty and `optOutDeg.getOrElse(0)` returns 0.

Using the `degreeGraph` print the number of people who like each user:

```

User 1 is called Alice and is liked by 2 people.
User 2 is called Bob and is liked by 2 people.
User 3 is called Charlie and is liked by 1 people.
User 4 is called David and is liked by 1 people.
User 5 is called Ed and is liked by 0 people.
User 6 is called Fran and is liked by 2 people.

```



✔ View Solution

```

1 for ((id, property) <- userGraph.vertices.collect) {
2   println(s"User $id is called ${property.name} and is liked by ${property.inDeg} people.")
3 }

```

Print the names of the users who are liked by the same number of people they like.



✔ View Solution

```

1 userGraph.vertices.filter {
2   case (id, u) => u.inDeg == u.outDeg
3 }.collect.foreach {
4   case (id, property) => println(property.name)
5 }

```



## 3.1. The Map Reduce Triplets Operator

Using the property graph from Section 2.1, suppose we want to find the oldest follower of each user. The `mapReduceTriplets` ([http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph@mapReduceTriplets\[A\]\(mapFunc:org.apache.spark.graphx.EdgeTriplet\[VD,ED\]=>Iterator\[\(org.apache.spark.graphx.VertexId,A\)\],reduceFunc:\(A,A\)=>A,activeSetOpt:Option\[\(org.apache.spark.graphx.VertexRDD\[\\_\],org.apache.spark.graphx.EdgeDirection\)\)](http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph@mapReduceTriplets[A](mapFunc:org.apache.spark.graphx.EdgeTriplet[VD,ED]=>Iterator[(org.apache.spark.graphx.VertexId,A)],reduceFunc:(A,A)=>A,activeSetOpt:Option[(org.apache.spark.graphx.VertexRDD[_],org.apache.spark.graphx.EdgeDirection))])) (`(implicitevidence$10:scala.reflect.ClassTag[A]):org.apache.spark.graphx.VertexRDD[A]`) operator allows us to do this. It enables neighborhood aggregation, and its simplified signature is as follows:



```
1 class Graph[VD, ED] {
2   def mapReduceTriplets[MsgType](
3     // Function from an edge triplet to a collection of messages (i.e., Map)
4     map: EdgeTriplet[VD, ED] => Iterator[(VertexId, MsgType)],
5     // Function that combines messages to the same vertex (i.e., Reduce)
6     reduce: (MsgType, MsgType) => MsgType
7   ): VertexRDD[MsgType]
8 }
```

The map function is applied to each edge triplet in the graph, yielding messages destined to the adjacent vertices. The reduce function aggregates messages destined to the same vertex. The operation results in a `VertexRDD` containing the aggregate message for each vertex.

We can find the oldest follower for each user by sending a message containing the name and age of each follower and aggregating the messages by taking the message from the older follower:



```
1 // Find the oldest follower for each user
2 val oldestFollower: VertexRDD[(String, Int)] = userGraph.mapReduceTriplets[(String, Int)](
3   // For each edge send a message to the destination vertex with the attribute of the source vertex
4   edge => Iterator((edge.dstId, (edge.srcAttr.name, edge.srcAttr.age))),
5   // To combine messages take the message for the older follower
6   (a, b) => if (a._2 > b._2) a else b
7 )
```

Display the oldest follower for each user:

```
David is the oldest follower of Alice.
Charlie is the oldest follower of Bob.
Ed is the oldest follower of Charlie.
Bob is the oldest follower of David.
Ed does not have any followers.
Charlie is the oldest follower of Fran.
```



```
1 userGraph.vertices.leftJoin(oldestFollower) { (id, user, optOldestFollower) =>
2   /**
3    * Implement: Generate a string naming the oldest follower of each user
4    * Note: Some users may have no messages optOldestFollower.isEmpty if they have no followers
5    *
6    * Try using the match syntax:
7    *
8    * optOldestFollower match {
9    *   case None => "No followers! implement me!"
10   *   case Some((name, age)) => "implement me!"
11   * }
12   *
13   */
14 }.collect.foreach {
15   case (id, str) => println(str)
16 }
```

```

1 userGraph.vertices.leftJoin(oldestFollower) { (id, user, optOldestFollower) =>
2   optOldestFollower match {
3     case None => s"${user.name} does not have any followers."
4     case Some((name, age)) => s"${name} is the oldest follower of ${user.name}."
5   }
6 }.collect.foreach { case (id, str) => println(str) }

```

As an exercise, try finding the average follower age of the followers of each user.



View Solution

```

1 val averageAge: VertexRDD[Double] = userGraph.mapReduceTriplets[(Int, Double)](
2   // map function returns a tuple of (1, Age)
3   edge => Iterator((edge.dstId, (1, edge.srcAttr.age.toDouble))),
4   // reduce function combines (sumOfFollowers, sumOfAge)
5   (a, b) => ((a._1 + b._1), (a._2 + b._2))
6   ).mapValues((id, p) => p._2 / p._1)
7
8 // Display the results
9 userGraph.vertices.leftJoin(averageAge) { (id, user, optAverageAge) =>
10   optAverageAge match {
11     case None => s"${user.name} does not have any followers."
12     case Some(avgAge) => s"The average age of ${user.name}'s followers is $avgAge."
13   }
14 }.collect.foreach { case (id, str) => println(str) }

```

## 3.2. Subgraph

Suppose we want to study the community structure of users that are 30 or older. To support this type of analysis GraphX includes the subgraph

([In the following we restrict our graph to the users that are 30 or older.](http://spark.incubator.apache.org/docs/latest/api/graphx/index.html#org.apache.spark.graphx.Graph@subgraph((EdgeTriplet[VD,ED])=>Boolean,(VertexId,VD)>Boolean):Graph[VD,ED])</a> operator that 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 <i>and connect vertices that satisfy the vertex predicate</i>.</p>
</div>
<div data-bbox=)



```

1 val olderGraph = userGraph.subgraph(vpred = (id, user) => user.age >= 30)

```

Lets examine the communities in this restricted graph:



```

1 // compute the connected components
2 val cc = olderGraph.connectedComponents
3
4 // display the component id of each user:
5 olderGraph.vertices.leftJoin(cc.vertices) {
6   case (id, user, comp) => s"${user.name} is in component ${comp.get}"
7 }.collect.foreach { case (id, str) => println(str) }

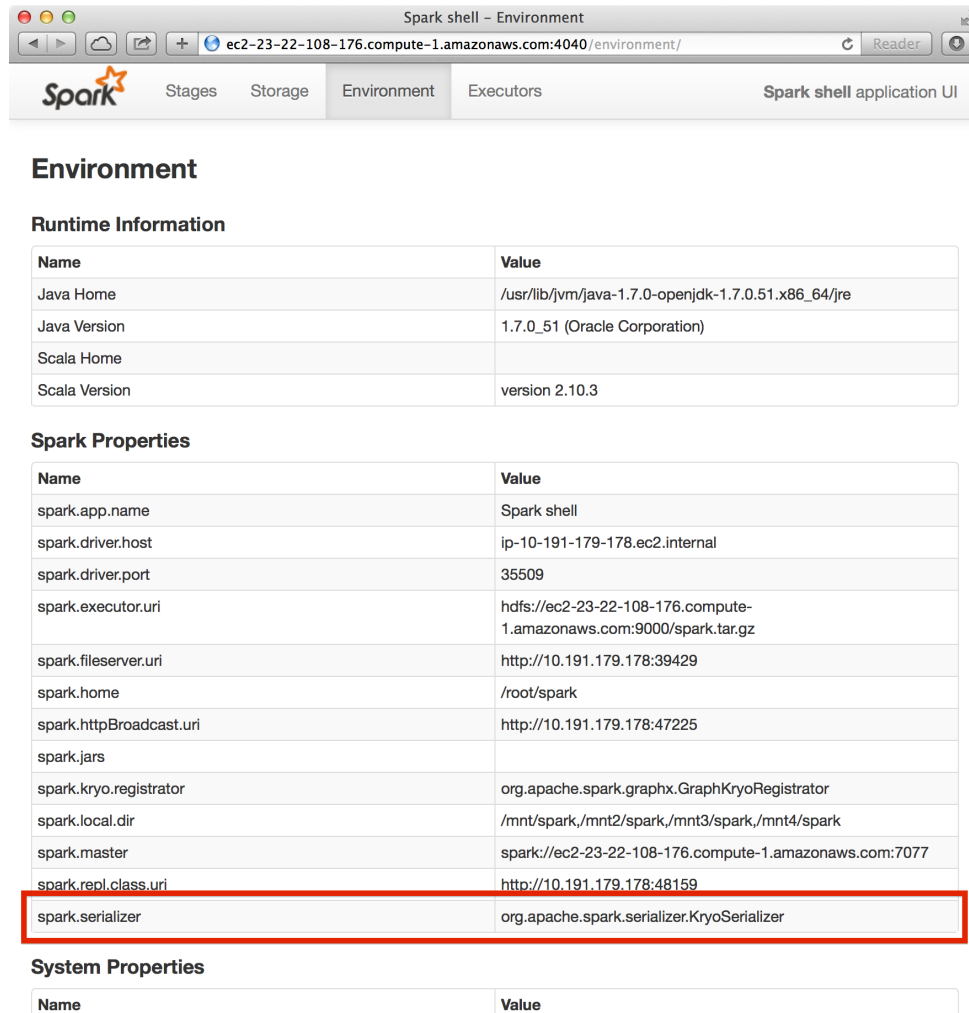
```

Connected components are labeled (numbered) by the lowest vertex Id in that component. Notice that by examining the subgraph we have disconnected David from the rest of his community. Moreover his connections to the rest of the graph are through younger users.

## 4. Constructing an End-to-End Graph Analytics Pipeline on Real Data

Now that we have learned about the individual components of the GraphX API, we are ready to put them together to build a real analytics pipeline. In this section, we will start with Wikipedia link data, use Spark operators to clean the data and extract structure, use GraphX operators to analyze the structure, and then use Spark operators to examine the output of the graph analysis, all from the Spark shell.

GraphX requires the Kryo serializer to achieve maximum performance. To see what serializer is being used check the Spark Shell UI by going to `http://<MASTER_URL>:4040/environment/` and checking the `spark.serializer` property:



The screenshot shows the Spark Shell Environment UI. The 'Environment' tab is selected, displaying runtime information and Spark properties. The 'spark.serializer' property is highlighted with a red box, showing its value as 'org.apache.spark.serializer.KryoSerializer'.

Name	Value
Java Home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.51.x86_64/jre
Java Version	1.7.0_51 (Oracle Corporation)
Scala Home	
Scala Version	version 2.10.3

Name	Value
spark.app.name	Spark shell
spark.driver.host	ip-10-191-179-178.ec2.internal
spark.driver.port	35509
spark.executor.uri	hdfs://ec2-23-22-108-176.compute-1.amazonaws.com:9000/spark.tar.gz
spark.fileserver.uri	http://10.191.179.178:39429
spark.home	/root/spark
spark.httpBroadcast.uri	http://10.191.179.178:47225
spark.jars	
spark.kryo.registrator	org.apache.spark.graphx.GraphKryoRegistrator
spark.local.dir	/mnt/spark,/mnt2/spark,/mnt3/spark,/mnt4/spark
spark.master	spark://ec2-23-22-108-176.compute-1.amazonaws.com:7077
spark.repl.class.uri	http://10.191.179.178:48159
spark.serializer	org.apache.spark.serializer.KryoSerializer

Name	Value
------	-------

By default Kryo Serialization is not enabled. In this exercise we will walk through the process of enabling Kryo Serialization for the spark shell. First exit the current Spark Shell (either type `exit` or `ctrl-c`).

Open a text editor (e.g., the one true editor `emacs` or `vim`) and add the following to `/root/spark/conf/spark-env.sh`:

```
SPARK_JAVA_OPTS+= '
-Dspark.serializer=org.apache.spark.serializer.KryoSerializer
-Dspark.kryo.registrator=org.apache.spark.graphx.GraphKryoRegistrator '
export SPARK_JAVA_OPTS
```

or if you are feeling lazy paste the following command in the terminal (not the spark shell):

```
echo -e "SPARK_JAVA_OPTS+= ' -Dspark.serializer=org.apache.spark.serializer.KryoSerializer -Dspark.kryo.registrator=org.apache.spark.graphx.GraphKryoRegistrator ' \nexport SPARK_JAVA_OPTS" >> /root/spark/conf/spark-env.sh
```

Then run the following command which will update the conf on all machines in the cluster:

```
/root/spark-ec2/copy-dir.sh /root/spark/conf
```

Finally restart the cluster (by again running the following in the terminal):

```
/root/spark/sbin/stop-all.sh
sleep 3
/root/spark/sbin/start-all.sh
```

After starting the Spark shell below, if you check `http://<MASTER_URL>:4040/environment/` the serializer property `spark.serializer` property should be set to `org.apache.spark.serializer.KryoSerializer`.

## 4.1. Getting Started (Again)

Start the spark shell:

```
/root/spark/bin/spark-shell
```

Import the standard packages.



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

## 4.2. Load the Wikipedia Articles

The first step in our analytics pipeline is to ingest our raw data into Spark. Load the data (located at `"/wiki_links/part*" in HDFS) into an RDD:`



```
1 // We tell Spark to cache the result in memory so we won't have to repeat the
2 // expensive disk IO. We coalesce down to 20 partitions to avoid excessive
3 // communication.
4 val wiki: RDD[String] = sc.textFile("/wiki_links/part*").coalesce(20)
```

## 4.3. Look at the First Article

Display the contents of the first article:



✔ View Solution

```
1 wiki.first
2 // res0: String = AccessibleComputing [[Computer accessibility]]
```

## 4.4. Clean the Data

The next steps in the pipeline are to clean the data and extract the graph structure. In this example, we will be extracting the link graph but one could imagine other graphs (e.g., the keyword by document graph and the contributor graph). From the sample article we printed out, we can already observe some structure to the data. The first word in the line is the name of the article, and the rest of string contains the links in the article.

Now we are going to use the structure we've already observed to do the first round of data-cleaning. We define the `Article` class to hold the different parts of the article which we parse from the raw string, and filter out articles that are malformed or redirects.

```
1 // Define the article class
2 case class Article(val title: String, val body: String)
3
4 // Parse the articles
5 val articles = wiki.map(_.split('\t')).
6   // two filters on article format
7   filter(line => (line.length > 1 && !(line(1) contains "REDIRECT"))).
8   // store the results in an object for easier access
9   map(line => new Article(line(0).trim, line(1).trim)).cache
```

We can see how many cleaned articles are left, computing and caching the cleaned article RDD in the process:

```
1 articles.count
```

## 4.5. Making a Vertex RDD

At this point, our data is in a clean enough format that we can create our vertex RDD. Remember we are going to extract the link graph from this dataset, so a natural vertex attribute is the title of the article. We are also going to define a mapping from article title to vertex ID by hashing the article title. Finish implementing the following:

```
1 // Hash function to assign an Id to each article
2 def pageHash(title: String): VertexId = {
3   title.toLowerCase.replace(" ", "").hashCode().toLong
4 }
5
6 // The vertices with id and article title:
7 val vertices: RDD[(VertexId, String)] = /* implement */
```

✔ View Solution

```
1 // Hash function to assign an Id to each article
2 def pageHash(title: String): VertexId = {
3   title.toLowerCase.replace(" ", "").hashCode().toLong
4 }
5 // The vertices with id and article title:
6 val vertices = articles.map(a => (pageHash(a.title), a.title)).cache
```

Again, let's force the vertices RDD by counting it:

```
1 vertices.count
```

## 4.6. Making the Edge RDD

The next step in data-cleaning is to extract our edges to find the structure of the link graph. We know that the MediaWiki syntax (the markup syntax Wikipedia uses) indicates a link by enclosing the link destination in double square brackets on either side. So a link looks like "[[Article We Are Linking To]]". Based on this knowledge, we can write a regular expression to extract all strings that are enclosed on both sides by "[[" and "]" respectively, and then apply that regular expression to each article's contents, yielding the destination of all links contained in the article.

Copy and paste the following into your Spark shell:

```

1 val pattern = "\\(\\|.\\+?\\|\\|\\)".r
2 val edges: RDD[Edge[Double]] = articles.flatMap { a =>
3   val srcVid = pageHash(a.title)
4   pattern.findAllIn(a.body).map { link =>
5     val dstVid = pageHash(link.replace("[", "").replace("]", ""))
6     Edge(srcVid, dstVid, 1.0)
7   }
8 }

```

This code extracts all the outbound links on each page and produces an RDD of edges with unit weight.

## 4.7. Making the Graph

We are finally ready to create our graph. Note that at this point, we have been using core Spark dataflow operators, working with our data in a table view. Switching to a graph view of our data is now as simple as calling the Graph constructor with our vertex RDD, our edge RDD, and a default vertex attribute. The default vertex attribute is used to initialize vertices that are not present in the vertex RDD, but are mentioned by an edge (that is, pointed to by a link). In the Wikipedia data there are often links to nonexistent pages.

We will use an empty title string as the default vertex attribute to represent the target of a broken link. Note that this concern arises because our dataset is imperfect (“dirty”), as might occur in a real world analytics pipeline.

Complete the following by restricting the graph to vertices with non-empty titles:



```
1 val graph = Graph(vertices, edges, "").subgraph(vpred = /* implement */).cache
```

 View Solution

```
1 val graph = Graph(vertices, edges, "").subgraph(vpred = {(v, d) => d.nonEmpty}).cache
```

Let's force the graph to be computed by counting some of their attributes (this might take two minutes):



```
1 graph.vertices.count
```

The first time the graph is created, GraphX constructs index data structures for all the vertices in the graph and detects and allocates missing vertices. Computing the triplets will require an additional join but this should run quickly now that the indexes have been created.



```
1 graph.triplets.count
```

## 4.8. Running PageRank on Wikipedia

We can now do some actual graph analytics. For this example, we are going to run PageRank (<http://en.wikipedia.org/wiki/PageRank>) to evaluate what the most important pages in the Wikipedia graph are. PageRank (PageRank) is part of a small but growing library of common graph algorithms already implemented in GraphX. However, the implementation is simple and straightforward, and just consists of some initialization code, a vertex program and message combiner to pass to Pregel.



```
1 val prGraph = graph.staticPageRank(5).cache
```

`Graph.staticPageRank` returns a graph whose vertex attributes are the PageRank values of each page. However, this means that while the resulting graph `prGraph` only contains the PageRank of the vertices and no longer contains the original vertex properties including the title. Luckily, we still have our graph that contains that information. Here, we can perform a join of the vertices in the `prGraph` that have the information about relative ranks of the vertices with the vertices in the graph that have the information about the mapping from vertex to article title. This yields a new graph that has combined both pieces of information, storing them both in a tuple as the new vertex attribute. We can then perform further table-based operators on this new list of vertices, such as finding the ten most important vertices (those with the highest pageranks) and printing out their corresponding article titles. Putting this all together, and we get the following set of operations to find the titles of the ten most important articles.



```
1 val titleAndPrGraph = graph.outerJoinVertices(prGraph.vertices) {  
2   (v, title, rank) => (rank.getOrElse(0.0), title)  
3 }  
4  
5 titleAndPrGraph.vertices.top(10) {  
6   Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)  
7 }.foreach(t => println(t._2._2 + ": " + t._2._1))
```

Finally, let's find the most important page within the subgraph of Wikipedia that mentions Berkeley in the title:



```
1 val berkeleyGraph = graph.subgraph(vpred = (v, t) => t.toLowerCase contains "berkeley")  
2  
3 val prBerkeley = berkeleyGraph.staticPageRank(5).cache  
4  
5 berkeleyGraph.outerJoinVertices(prBerkeley.vertices) {  
6   (v, title, r) => (r.getOrElse(0.0), title)  
7 }.vertices.top(10) {  
8   Ordering.by((entry: (VertexId, (Double, String))) => entry._2._1)  
9 }.foreach(t => println(t._2._2 + ": " + t._2._1))
```

This brings us to the end of the GraphX chapter of the tutorial. We encourage you to continue playing with the code and to check out the GraphX Programming Guide (<http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html>) for further documentation about the system.

Bug reports and feature requests are welcomed.



([movie-recommendation-with-mllib.html](#))



([tachyon.html](#))

🔗 Submit an issue on GitHub (<https://github.com/amplab/training/issues>)

☰ Graph Analytics With GraphX ▲

## ***Hands-on Exercises***