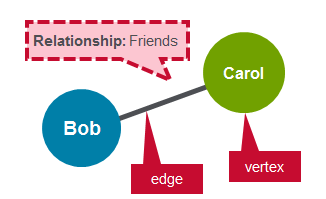
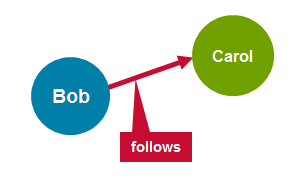
GraphX is the Apache Spark component for graph-parallel computations, built upon a branch of mathematics called graph theory. It is a distributed graph processing framework that sits on top of Spark core.

**Overview of some graph concepts**

A graph is a mathematical structure used to model relations between objects. A graph is made up of vertices and edges that connect them. The vertices are the objects and the edges are the relationships between them.



A directed graph is a graph where the edges have a direction associated with them. An example of a directed graph is a Twitter follower. User Bob can follow user Carol without implying that user Carol follows user Bob.

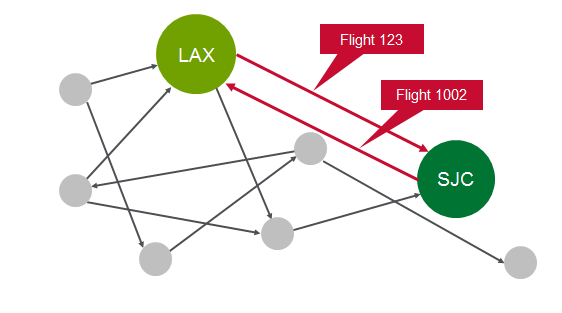


A  **regular graph** is a graph where each vertex has the same number of edges. An example of a regular graphs is Facebook friends. If Bob is a friend of Carol, then Carol is also a friend of Bob.

**Graphx Property Graph**

GraphX extends the Spark RDD with a Resilient Distributed Property Graph.

The [property graph](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph) is a directed multigraph which can have multiple edges in parallel. Every edge and vertex has user defined properties associated with it. The parallel edges allow multiple relationships between the same vertices.



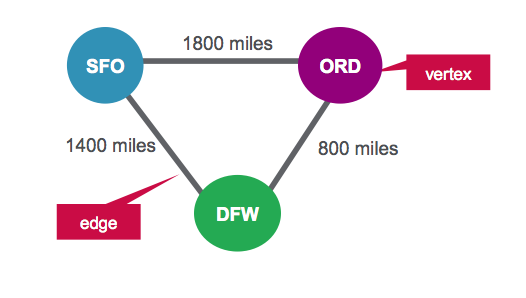
In this activity, you will use GraphX to analyze flight data.

# Scenario

As a starting simple example, we will analyze 3 flights, for each flight we have the following information:

|  |  |  |
| --- | --- | --- |
| Originating Airport | Destination Airport | Distance |
| SFO | ORD | 1800 miles |
| ORD | DFW | 800 miles |
| DFW | SFO | 1400 miles |

In this scenario, we are going to represent the airports as vertices and routes as edges. For our graph we will have three vertices, each representing an airport. The distance between the airports is a route property, as shown below:



**Vertex Table for Airports**

|  |  |
| --- | --- |
| ID | Property |
| 1 | SFO |
| 2 | ORD |
| 3 | DFW |

**Edges Table for Routes**

|  |  |  |
| --- | --- | --- |
| SrcId | DestId | Property |
| 1 | 2 | 1800 |
| 2 | 3 | 800 |
| 3 | 1 | 1400 |

# Software

This tutorial will run on the MapR v5.0 Sandbox, which includes Spark 1.3

* You can download the code and data to run these examples from here:
  + [https://github.com/caroljmcdonald/sparkdataframeexample](https://github.com/caroljmcdonald/sparkdataframeexample" \t "_new)
* The examples in this post can be run in the spark-shell, after launching with the spark-shell command.
* You can also run the code as a standalone application as described in the tutorial on [Getting Started with Spark on MapR Sandbox](https://www.mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial" \t "_new).

**Launch the Spark Interactive Shell**

Log into the MapR Sandbox, as explained in [Getting Started with Spark on MapR Sandbox](https://www.mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial" \t "_new), using userid user01, password mapr. Copy the sample data files to your sandbox home directory /user/user01 using scp. Start the spark shell with:   
$ spark-shell

|  |
| --- |
|  |

## Define Vertices

First we will import the GraphX packages.

(In the code boxes, comments are in Green and output is in Blue)

|  |
| --- |
| import org.apache.spark.\_  import org.apache.spark.rdd.RDD  // import classes required for using GraphX  import org.apache.spark.graphx.\_ |

We define airports as vertices. Vertices have an Id and can have properties or attributes associated with them. Each vertex consists of :

* Vertex id 🡪 Id (Long)
* Vertex Property 🡪 name (String)

Vertex Table for Airports

|  |  |
| --- | --- |
| ID | Property(V) |
| 1 | SFO |

We define an RDD with the above properties that is then used for the Vertexes .

|  |
| --- |
| // create vertices RDD with ID and Name  val vertices=Array((1L, ("SFO")),(2L, ("ORD")),(3L,("DFW")))  val vRDD= sc.parallelize(vertices)  vRDD.take(1)  // Array((1,SFO))  // Defining a default vertex called nowhere  val nowhere = "nowhere" |

## Define Edges

Edges are the routes between airports. An edge must have a source, a destination, and can have properties. In our example, an edge consists of :

* Edge origin id 🡪 src (Long)
* Edge destination id 🡪 dest (Long)
* Edge Property distance 🡪 distance (Long)

Edges Table for Routes

|  |  |  |
| --- | --- | --- |
| srcid | destid | Property(E) |
| 1 | 12 | 1800 |

We define an RDD with the above properties that is then used for the Edges . The edge RDD has the form (src id, dest id, distance ).

|  |
| --- |
| // create routes RDD with srcid, destid , distance  val edges = Array(Edge(1L,2L,1800),Edge(2L,3L,800),Edge(3L,1L,1400))  val eRDD= sc.parallelize(edges)  eRDD.take(2)  // Array(Edge(1,2,1800), Edge(2,3,800)) |

## Create Property Graph

To create a graph, you need to have a Vertex RDD, Edge RDD and a Default vertex.

Create a property graph called graph.

|  |
| --- |
| // define the graph  val graph = Graph(vRDD,eRDD, nowhere)  // graph vertices  graph.vertices.collect.foreach(println)  // (2,ORD)  // (1,SFO)  // (3,DFW)  // graph edges  graph.edges.collect.foreach(println)  // Edge(1,2,1800)  // Edge(2,3,800)  // Edge(3,1,1400) |

1. How many airports are there?

|  |
| --- |
| // How many airports?  val numairports = graph.numVertices  // Long = 3 |

1. How many routes are there?

|  |
| --- |
| // How many routes?  val numroutes = graph.numEdges  // Long = 3 |

1. which routes > 1000 miles distance?

|  |
| --- |
| // routes > 1000 miles distance?  graph.edges.filter { case Edge(src, dst, prop) => prop > 1000 }.collect.foreach(println)  // Edge(1,2,1800)  // Edge(3,1,1400) |

1. The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.

|  |
| --- |
| // triplets  graph.triplets.take(3).foreach(println)  ((1,SFO),(2,ORD),1800)  ((2,ORD),(3,DFW),800)  ((3,DFW),(1,SFO),1400) |

1. Sort and print out the longest distance routes

|  |
| --- |
| // print out longest routes  graph.triplets.sortBy(\_.attr, ascending=false).map(triplet =>  "Distance " + triplet.attr.toString + " from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").collect.foreach(println)  Distance 1800 from SFO to ORD.  Distance 1400 from DFW to SFO.  Distance 800 from ORD to DFW. |

# Analyze real Flight data with GraphX

# Scenario

Our data is from <http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time>. We are using flight information for January 2015. For each flight we have the following information:

|  |  |  |
| --- | --- | --- |
| Field | Description | Example Value |
| dOfM(String) | Day of month | 1 |
| dOfW (String) | Day of week | 4 |
| carrier (String) | Carrier code | AA |
| tailNum (String) | Unique identifier for the plane - tail number | N787AA |
| flnum(Int) | Flight number | 21 |
| org\_id(String) | Origin airport ID | 12478 |
| origin(String) | Origin Airport Code | JFK |
| dest\_id (String) | Destination airport ID | 12892 |
| dest (String) | Destination airport code | LAX |
| crsdeptime(Double) | scheduled departure time | 900 |
| deptime (Double) | actual departure time | 855 |
| depdelaymins (Double) | departure delay in minutes | 0 |
| crsarrtime (Double) | scheduled arrival time | 1230 |
| arrtime (Double) | actual arrival time | 1237 |
| arrdelaymins (Double) | Arrival delay minutes | 7 |
| crselapsedtime (Double) | Elapsed time | 390 |
| dist (Int) | Distance | 2475 |

In this scenario, we are going to represent the airports as vertices and routes as edges. We are interested in visualizing airports and routes and would like to see the number of airports that have departures or arrivals.

You can download the code and data to run these examples from here:

* [https://github.com/caroljmcdonald/sparkdataframeexample](https://github.com/caroljmcdonald/sparkdataframeexample" \t "_new)

Log into the MapR Sandbox, as explained in [Getting Started with Spark on MapR Sandbox](https://www.mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial" \t "_new), using userid user01, password mapr. Copy the sample data files to your sandbox home directory /user/user01 using scp. Start the spark shell with:   
$ spark-shell

## Define Vertices

First we will import the GraphX packages.

(In the code boxes, comments are in Green and output is in Blue)

|  |
| --- |
| import org.apache.spark.\_  import org.apache.spark.rdd.RDD  import org.apache.spark.util.IntParam  // import classes required for using GraphX  import org.apache.spark.graphx.\_  import org.apache.spark.graphx.util.GraphGenerators |

Below we a Scala case classes to define the Flight schema corresponding to the csv data file.

|  |
| --- |
| // define the Flight Schema  case class Flight(dofM:String, dofW:String, carrier:String, tailnum:String, flnum:Int, org\_id:Long, origin:String, dest\_id:Long, dest:String, crsdeptime:Double, deptime:Double, depdelaymins:Double, crsarrtime:Double, arrtime:Double, arrdelay:Double,crselapsedtime:Double,dist:Int) |

The function below parses a line from the data file into the Flight class.

|  |
| --- |
| // function to parse input into Flight class  def parseFlight(str: String): Flight = {  val line = str.split(",")  Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5).toLong, line(6), line(7).toLong, line(8), line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble, line(13).toDouble, line(14).toDouble, line(15).toDouble, line(16).toInt)  } |

Below we load the data from the csv file into a [Resilient Distributed Dataset (RDD)](https://spark.apache.org/docs/0.8.1/api/core/org/apache/spark/rdd/RDD.html). RDDs can have [transformations](https://spark.apache.org/docs/1.3.0/programming-guide.html#transformations) and [actions](https://spark.apache.org/docs/1.3.0/programming-guide.html#actions), the first() action returns the first element in the RDD.

|  |
| --- |
| // load the data into a RDD  val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")  // MapPartitionsRDD[1] at textFile  // parse the RDD of csv lines into an RDD of flight classes  val flightsRDD = textRDD.map(parseFlight).cache() |

We define airports as vertices. Vertices can have properties or attributes associated with them. Each vertex has the following property:

* Airport name (String)

Vertex Table for Airports

|  |  |
| --- | --- |
| ID | Property(V) |
| 10397 | ATL |

We define an RDD with the above properties that is then used for the Vertexes .

|  |
| --- |
| // create airports RDD with ID and Name  val airports = flightsRDD.map(flight => (flight.org\_id, flight.origin)).distinct  airports.take(1)  // Array((14057,PDX))  // Defining a default vertex called nowhere  val nowhere = "nowhere"  // Map airport ID to the 3-letter code to use for printlns  val airportMap = airports.map { case ((org\_id), name) => (org\_id -> name) }.collect.toList.toMap  // Map(13024 -> LMT, 10785 -> BTV,…) |

## Define Edges

Edges are the routes between airports. An edge must have a source, a destination, and can have properties. In our example, an edge consists of :

* Edge origin id 🡪 src (Long)
* Edge destination id 🡪 dest (Long)
* Edge Property distance 🡪 distance (Long)

Edges Table for Routes

|  |  |  |
| --- | --- | --- |
| srcid | destid | Property(E) |
| 14869 | 14683 | 1087 |

We define an RDD with the above properties that is then used for the Edges . The edge RDD has the form (src id, dest id, distance ).

|  |
| --- |
| // create routes RDD with srcid, destid , distance  val routes = flightsRDD.map(flight => ((flight.org\_id, flight.dest\_id), flight.dist)).distinctdistinct  routes.take(2)  // Array(((14869,14683),1087), ((14683,14771),1482))  // create edges RDD with srcid, destid , distance  val edges = routes.map {  case ((org\_id, dest\_id), distance) =>Edge(org\_id.toLong, dest\_id.toLong, distance) }  edges.take(1)  //Array(Edge(10299,10926,160)) |

## Create Property Graph

To create a graph, you need to have a Vertex RDD, Edge RDD and a Default vertex.

Create a property graph called graph.

|  |
| --- |
| // define the graph  val graph = Graph(airports, edges, nowhere)  // graph vertices  graph.vertices.take(2)  Array((10208,AGS), (10268,ALO))  // graph edges  graph.edges.take(2)  Array(Edge(10135,10397,692), Edge(10135,13930,654)) |

1. How many airports are there?

|  |
| --- |
| // How many airports?  val numairports = graph.numVertices  // Long = 301 |

1. How many routes are there?

|  |
| --- |
| // How many airports?  val numroutes = graph.numEdges  // Long = 4090 |

1. which routes > 1000 miles distance?

|  |
| --- |
| // routes > 1000 miles distance?  graph.edges.filter { case ( Edge(org\_id, dest\_id,distance))=> distance > 1000}.take(3)  // Array(Edge(10140,10397,1269), Edge(10140,10821,1670), Edge(10140,12264,1628)) |

1. The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.

|  |
| --- |
| // triplets  graph.triplets.take(3).foreach(println)  ((10135,ABE),(10397,ATL),692)  ((10135,ABE),(13930,ORD),654)  ((10140,ABQ),(10397,ATL),1269) |

1. compute the highest degree vertex

|  |
| --- |
| // 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  }  val maxInDegree: (VertexId, Int) = graph.inDegrees.reduce(max)  maxInDegree: (org.apache.spark.graphx.VertexId, Int) = (10397,152)  val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)  maxOutDegree: (org.apache.spark.graphx.VertexId, Int) = (10397,153)  val maxDegrees: (VertexId, Int) = graph.degrees.reduce(max)  val maxDegrees: (VertexId, Int) = graph.degrees.reduce(max)  airportMap(10397)  res70: String = ATL |

1. which airport has the most incoming flights?

|  |
| --- |
| // get top 3  val maxIncoming = graph.inDegrees.collect.sortWith(\_.\_2 > \_.\_2).map(x => (airportMap(x.\_1), x.\_2)).take(3)  maxIncoming.foreach(println)  (ATL,152)  (ORD,145)  (DFW,143)  **// which airport has the most outgoing flights?**  val maxout= graph.outDegrees.join(airports).sortBy(\_.\_2.\_1, ascending=false).take(3)  maxout.foreach(println)  (10397,(153,ATL))  (13930,(146,ORD))  (11298,(143,DFW)) |

1. What are the top 10 flights from airport to airport?

|  |
| --- |
| // get top 10 flights airport to airport  graph.triplets.sortBy(\_.attr, ascending=false).map(triplet => "There were " + triplet.attr.toString + " flights from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").take(3) .foreach(println)  There were 4983 flights from JFK to HNL  There were 4983 flights from HNL to JFK.  There were 4963 flights from EWR to HNL |

# PageRank

PageRank measures the importance of each vertex in a graph, by determining which vertexes have the most edges with other vertexes. In our example we can use PageRank to determine which airports are the most important , by measuring which airports have the most connections to other airports.

Another operator is PageRank. It is based on the Google PageRank algorithm. It can be used to measure the importance of the vertices. We have to specify the tolerance, which is the measure of convergence.

PageRank on a graph will return a graph with vertex attributes containing the normalized edge weight.

In our example, this would be to answer the question – Which are the most important airports?

1. What are the most important airports according to PageRank?

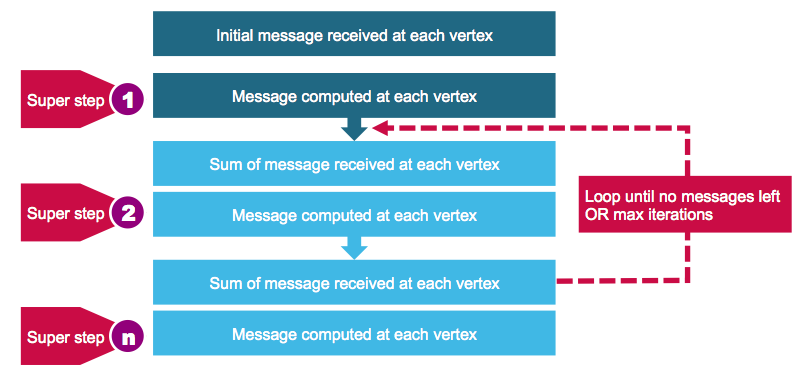
|  |
| --- |
| // use pageRank  val ranks = graph.pageRank(0.1).vertices  val ranksAndAirports = ranks.join(airports).sortBy(\_.\_2.\_1, false).map(\_.\_2.\_2)  ranksAndAirports.take(4)  //res6: Array[String] = Array(ATL, ORD, DFW, DEN) |

# Pregel

GraphX exposes a variant of the Pregel API. The Pregel API is useful for iterative computing as it unpersists intermediate results. The Pregel operator in GraphX is defined as a bulk-synchronous messaging abstraction.

In GraphX, properties of vertices depend on properties of their neighbors, which depend on properties of *their* neighbors. This means that the Pregel implementation in GraphX is such that vertices in GraphX can only send messages to neighboring vertices.

The Pregel operator is executed in a series of super steps. The initial message is received at each vertex and computes.



In each super step, the vertices receive the sum of their inbound messages from the previous super step. A new value for the vertex property is computed and messages are sent to the neighboring vertices in the next super step. When there are no more messages remaining, the Pregel operator will end the iteration and the final graph is returned.

Assume that you have the following formula to compute airfare.

50 + distance / 20

Use the code below to compute the cheapest airfare.

|  |
| --- |
| val sourceId: VertexId = 13024  // airfare formula  val gg = graph.mapEdges(e => 50.toDouble + e.attr.toDouble/20 )  val initialGraph = gg.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.take(4).mkString("\n"))  (10208,277.79)  (10268,260.7)  (14828,261.65)  (14698,125.25)  println(sssp.edges.take(4).mkString("\n"))  Edge(10135,10397,84.6)  Edge(10135,13930,82.7)  Edge(10140,10397,113.45)  Edge(10140,10821,133.5)  sssp.vertices.collect.map(x => (airportMap(x.\_1), x.\_2)).sortWith(\_.\_2 < \_.\_2)  res21: Array[(String, Double)] = Array(PDX,62.05), (SFO,65.75), (EUG,117.35) |

**Want to learn more?**

* http://spark.apache.org/docs/latest/graphx-programming-guide.html
* http://ampcamp.berkeley.edu/big-data-mini-course/graph-analytics-with-graphx.html