Hands On: Building A Graph

1. Start the Cloudera VM and Upload the datasets to HDFS
2. Import the GraphX libraries
3. Import the Vertices
4. Import the Edges
5. Create a Graph
6. Use Spark's filter method to return Vertices in the graph

Start the Cloudera VM and Upload the datasets to HDFS

Ensure the Cloudera Quick Start virtual machine is started and that you have downloaded ExamplesOfAnalytics.zip from the provided link in the content section for this week.

Copy the ExamplesOfAnalytics.zip file to the Cloudera's Home folder before proceeding.

Open a terminal in the Cloudera Quick Start virtual machine by clicking **Applications**, **System Tools** then **Terminal**.

Type the following command in the Terminal window to extract the zip file to the Cloudera Home directory.

```
unzip ExamplesOfAnalytics.zip
```

Type the following command in the Terminal window to go into the ExamplesOfAnalytics directory.

```
cd ExamplesOfAnalytics
```

Use the hdfs command to upload the datasets in the EOADATA directory to HDFS.

```
hdfs dfs -put EOADATA
```

Start the Spark Shell will all of the libraries needed to complete the hands on exercises.

```
spark-shell --jars lib/gs-core-1.2.jar,lib/gs-ui-1.2.jar,lib/jcommon-
1.0.16.jar,lib/jfreechart-1.0.13.jar,lib/breeze_2.10-0.9.jar,lib/breeze-viz_2.10-
0.9.jar,lib/pherd-1.0.jar
```

It may take several seconds for the Spark Shell to start. Be patient and wait for the **scala>** prompt.

Import the GraphX libraries

Set log level to error, suppress info and warn messages.

```
import org.apache.log4j.Logger
import org.apache.log4j.Level

Logger.getLogger("org").setLevel(Level.ERROR)
Logger.getLogger("akka").setLevel(Level.ERROR)
```

Import the Spark's GraphX and RDD libraries along with Scala's source library.

```
import org.apache.spark.graphx._
import org.apache.spark.rdd._

import scala.io.Source
```

Import the Vertices

Before importing any datasets, let view what the files contain. Print the first 5 lines of each comma delimited text file.

input:

```
Source.fromFile("./EOADATA/metro.csv").getLines().take(5).foreach(println)
```

output:

```
#metro_id,name,population
 1,Tokyo,36923000
 2,Seoul,25620000
 3,Shanghai,24750000
 4,Guangzhou,23900000
```

input:

```
Source.fromFile("./EOADATA/country.csv").getLines().take(5).foreach(println)
```

output:

```
#country_id,name
1,Japan
2,South Korea
```

```
3,China
4,India
```

input:

```
Source.fromFile("./EOADATA/metro_country.csv").getLines().take(5).foreach(println)
```

output:

```
#metro_id,country_id
1,1
2,2
3,3
4,3
```

Create case classes for the places (metros and countries).

input:

```
class PlaceNode(val name: String) extends Serializable
```

output:

```
defined class PlaceNode
```

input:

```
case class Metro(override val name: String, population: Int) extends PlaceNode(name)
```

output:

```
defined class Metro
```

input:

```
case class Country(override val name: String) extends PlaceNode(name)
```

output:

```
defined class Country
```

Read the comma delimited text file metros.csv into an RDD of Metro vertices, ignore lines that start with # and map the columns to: id, Metro(name, population).

input:

```
val metros: RDD[(VertexId, PlaceNode)] =
  sc.textFile("./EOADATA/metro.csv").
    filter(! _.startsWith("#")).
    map {line =>
      val row = line split ','
      (0L + row(0).toInt, Metro(row(1), row(2).toInt))
    }
```

output:

```
metros: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, PlaceNode)] =
MapPartitionsRDD[18] at map at <console>:36
```

Read the comma delimited text file country.csv into an RDD of Country vertices, ignore lines that start with # and map the columns to: id, Country(name). Add 100 to the country indexes so they are unique from the metro indexes.

input:

```
val countries: RDD[(VertexId, PlaceNode)] =
  sc.textFile("./EOADATA/country.csv").
    filter(! _.startsWith("#")).
    map {line =>
      val row = line split ','
      (100L + row(0).toInt, Country(row(1)))
    }
```

output:

```
countries: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, PlaceNode)] =
MapPartitionsRDD[26] at map at <console>:36
```

Import the Vertices

Read the comma delimited text file metro_country.tsv into an RDD[Edge[Int]] collection. Remember to add 100 to the countries' vertex id.

input:

```
val mclinks: RDD[Edge[Int]] =
  sc.textFile("./EOADATA/metro_country.csv").
    filter(! _.startsWith("#")).
    map {line =>
      val row = line split ','
      Edge(0L + row(0).toInt, 100L + row(1).toInt, 1)
    }
```

output:

```
mclinks: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] =
MapPartitionsRDD[30] at map at <console>:33
```

Create a Graph

Concatenate the two sets of nodes into a single RDD.

input:

```
val nodes = metros ++ countries
```

output:

```
nodes: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, PlaceNode)] =
UnionRDD[31] at $plus$plus at <console>:39
```

Pass the concatenated RDD to the Graph() factory method along with the RDD link

input:

```
val metrosGraph = Graph(nodes, mclinks)
```

output:

```
metrosGraph: org.apache.spark.graphx.Graph[PlaceNode,Int] =
org.apache.spark.graphx.impl.GraphImpl@7b13f4ea
```

Print the first 5 vertices and edges.

input:

```
metrosGraph.vertices.take(5)
```

output:

```
res8: Array[(org.apache.spark.graphx.VertexId, PlaceNode)] = Array((34,Metro(Hong
Kong,7298600)),
(52,Metro(Ankara,5150072)), (4,Metro(Guangzhou,23900000)),
(16,Metro(Istanbul,14377018)), (28,Metro(Nagoya,9107000)))
```

input:

```
metrosGraph.edges.take(5)
```

output:

```
res9: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(1,101,1), Edge(2,102,1),
Edge(3,103,1), Edge(4,103,1), Edge(5,104,1))
```

Use Spark’s filter method to return Vertices in the graph

Filter all of the edges in metrosGraph that have a source vertex Id of 1 and create a map of destination vertex Ids.

input:

```
metrosGraph.edges.filter(_.srcId == 1).map(_.dstId).collect()
```

output:

```
res10: Array[org.apache.spark.graphx.VertexId] = Array(101)
```

Similarly, filter all of the edges in metrosGraph where the destination vertexId is 103 and create a map of all of the source Ids.

input:

```
metrosGraph.edges.filter(_.dstId == 103).map(_.srcId).collect()
```

output:

```
res11: Array[org.apache.spark.graphx.VertexId] = Array(3, 4, 7, 24, 34)
```

Hands On: Building A Degree Histogram

1. Count the number of vertices and edges
2. Define a min and max function for Spark's reduce method
3. Compute min and max degrees
4. Compute the histogram data of the degree of connectedness

Count the number of vertices and edges

Print the number of links.

input:

```
metrosGraph.numEdges
```

output:

```
res13: Long = 65
```

Print the number of nodes.

input:

```
metrosGraph.numVertices
```

output:

```
res14: Long = 93
```

Define a min and max function for Spark’s reduce method

Define a min and max function.

input:

```
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
```

output:

```
max: (a: (org.apache.spark.graphx.VertexId, Int), b:
(org.apache.spark.graphx.VertexId, Int))(org.apache.spark.graphx.VertexId, Int)
```

input:

```
def min(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 <= b._2) a else b
}
```

output:

```
min: (a: (org.apache.spark.graphx.VertexId, Int), b:
(org.apache.spark.graphx.VertexId, Int))(org.apache.spark.graphx.VertexId, Int)
```

Compute min and max degrees

Find which which VertexId and the edge count of the vertex with the most out edges. (This can be any vertex because all vertices have one out edge.)

input:

```
metrosGraph.outDegrees.reduce(max)
```

output:

```
res15: (org.apache.spark.graphx.VertexId, Int) = (5,1)
```

Print the returned vertex.

input:

```
metrosGraph.vertices.filter(_._1 == 5).collect()
```

output:

```
res16: Array[(org.apache.spark.graphx.VertexId, PlaceNode)] =
Array((5,Metro(Delhi,21753486)))
```

Find which which VertexId and the edge count of the vertex with the most in edges.

input:

```
metrosGraph.inDegrees.reduce(max)
```

output:

```
res17: (org.apache.spark.graphx.VertexId, Int) = (108,14)
```

Print the returned vertex.

input:

```
metrosGraph.vertices.filter(_._1 == 108).collect()
```

output:

```
res18: Array[(org.apache.spark.graphx.VertexId, PlaceNode)] =
Array((108,Country(United States)))
```

Find the number vertexes that have only one out edge.

input:

```
metrosGraph.outDegrees.filter(_._2 <= 1).count
```

output:

```
res19: Long = 65
```

Find the maximum and minimum degrees of the connections in the network.

input:

```
metrosGraph.degrees.reduce(max)
```

output:

```
res20: (org.apache.spark.graphx.VertexId, Int) = (108,14)
```

input:

```
metrosGraph.degrees.reduce(min)
```

output:

```
res21: (org.apache.spark.graphx.VertexId, Int) = (34,1)
```

Compute the histogram data of the degree of connectedness

Print the histogram data of the degrees for countries only.

input:

```
metrosGraph.degrees.
  filter { case (vid, count) => vid >= 100 }. // Apply filter so only VertexId < 100
(countries) are included
  map(t => (t._2,t._1)).
  groupByKey.map(t => (t._1,t._2.size)).
  sortBy(_._1).collect()
```

output:

```
res22: Array[(Int, Int)] = Array((1,18), (2,4), (3,2), (5,2), (9,1), (14,1))
```

Hands On: Plot the Degree Histogram

1. Import the BreezeViz library
2. Define a function to calculate the degree histogram
3. Calculate the probability distribution for the degree histogram
4. Graph the results

Import the BreezeViz library

```
import breeze.linalg._
import breeze.plot._
```

Define a function to calculate the degree histogram

Define a function to create a histogram of the degrees. The function is nearly identical to how the histogram was computed in the previous hands on exercise. Remember to only include countries!

input:

```
def degreeHistogram(net: Graph[PlaceNode, Int]): Array[(Int, Int)] =
  net.degrees.
    filter { case (vid, count) => vid >= 100 }.
    map(t => (t._2,t._1)).
    groupByKey.map(t => (t._1,t._2.size)).
    sortBy(_._1).collect()
```

output:

```
degreeHistogram: (net: org.apache.spark.graphx.Graph[PlaceNode,Int])Array[(Int, Int)]
```

Calculate the probability distribution for the degree histogram

Get the probability distribution (degree distribution) from the degree histogram by normalizing the node degrees by the total number of nodes, so that the degree probabilities add up to one.

input:

```
val nn = metrosGraph.vertices.filter{ case (vid, count) => vid >= 100 }.count()
```

output:

```
nn: Long = 28
```

input:

```
val metroDegreeDistribution = degreeHistogram(metrosGraph).map({case(d,n) =>
(d,n.toDouble/nn)})
```

output:

```
metroDegreeDistribution: Array[(Int, Double)] = Array((1,0.6428571428571429),
(2,0.14285714285714285),
(3,0.07142857142857142), (5,0.07142857142857142), (9,0.03571428571428571),
(14,0.03571428571428571))
```

Graph the results

Plot degree distribution and the histogram of vertex degrees.

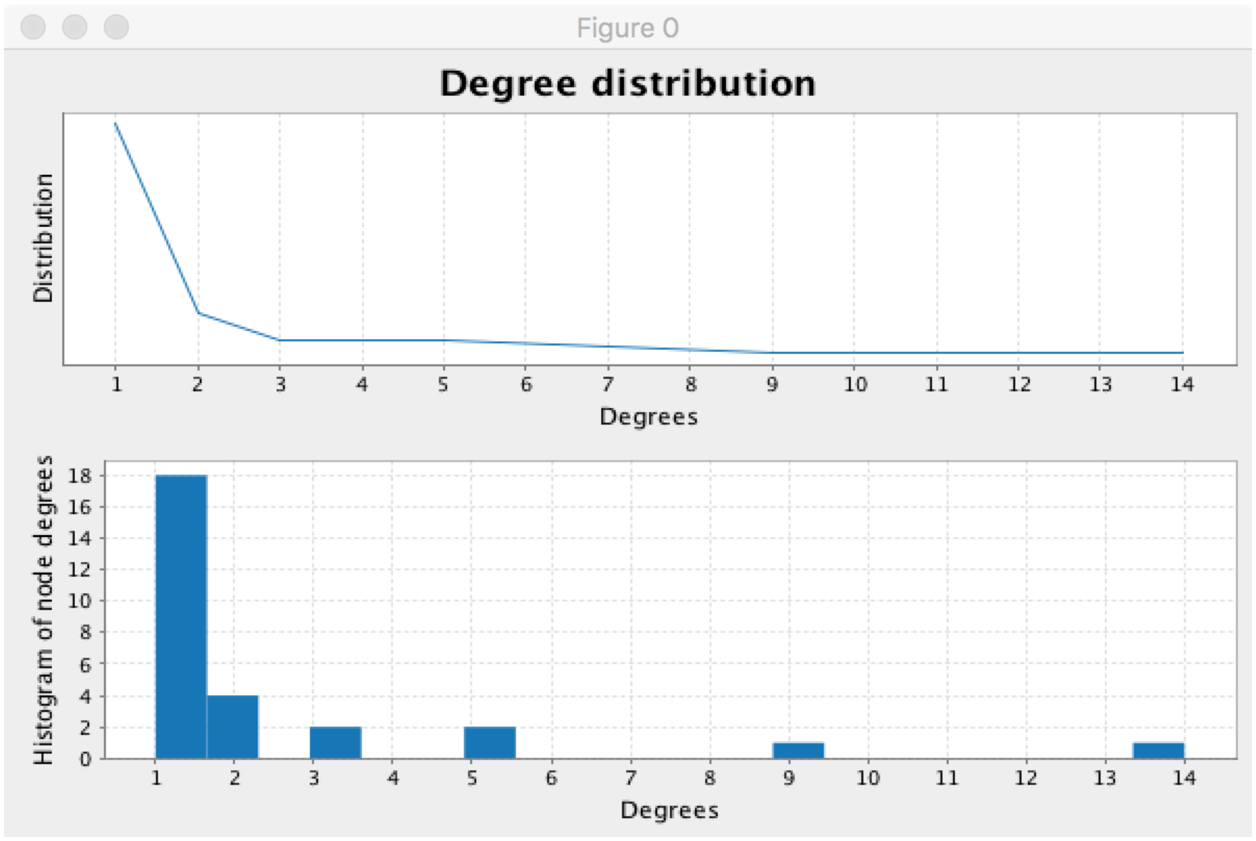
input:

```
val f = Figure()
val p1 = f.subplot(2,1,0)
val x = new DenseVector(metroDegreeDistribution map (_._1.toDouble))
val y = new DenseVector(metroDegreeDistribution map (_._2))

p1.xlabel = "Degrees"
p1.ylabel = "Distribution"
p1 += plot(x, y)
p1.title = "Degree distribution"


val p2 = f.subplot(2,1,1)
val metrosDegrees = metrosGraph.degrees.filter { case (vid, count) => vid >= 100
}.map(_._2).collect()

p2.xlabel = "Degrees"
p2.ylabel = "Histogram of node degrees"
p2 += hist(metrosDegrees, 20)
```
output:



Hands On: Network Connectedness and Clustering Components

1. Create a new graph by adding the Continents dataset
2. Import the GraphStream library
3. Import countriesGraph into a GraphStream SingleGraph
4. Visualize countriesGraph
5. Visualize Facebook graph

Create a new graph by adding the Continents dataset

To make the graph more interesting, create a new graph and add the continents.

input:

```
Source.fromFile("./EOADATA/continent.csv").getLines().take(5).foreach(println)
```

output:

```
#continent_id,name
1,Asia
2,Africa
3,North America
4,South America
```

input:

```
Source.fromFile("./EOADATA/country_continent.csv").getLines().take(5).foreach(println
)
```

output:

```
#country_id,continent_id
1,1
2,1
3,1
4,1
```

input:

```
case class Continent(override val name: String) extends PlaceNode(name)
```

output:

```
defined class Continent
```

input:

```
val continents: RDD[(VertexId, PlaceNode)] =
  sc.textFile("./EOADATA/continent.csv").
    filter(! _.startsWith("#")).
    map {line =>
      val row = line split ','
      (200L + row(0).toInt, Continent(row(1))) // Add 200 to the VertexId to keep the
indexes unique
    }
```

output:

```
continents: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, PlaceNode)] =
MapPartitionsRDD[106] at map at <console>:42
```

input:

```
val cclinks: RDD[Edge[Int]] =
  sc.textFile("./EOADATA/country_continent.csv").
    filter(! _.startsWith("#")).
    map {line =>
      val row = line split ','
      Edge(100L + row(0).toInt, 200L + row(1).toInt, 1)
    }
```

output:

```
cclinks: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] =
MapPartitionsRDD[110] at map at <console>:39
```

Concatenate the three sets of nodes into a single RDD.

input:

```
val cnodes = metros ++ countries ++ continents
```

output:

```
cnodes: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, PlaceNode)] =
UnionRDD[112] at $plus$plus at <console>:49
```

Concatenate the two sets of edges

input:

```
val clinks = mclinks ++ cclinks
```

output:

```
clinks: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] = UnionRDD[113]
at $plus$plus at <console>:40
```

input:

```
val countriesGraph = Graph(cnodes, clinks)
```

output:

```
countriesGraph: org.apache.spark.graphx.Graph[PlaceNode,Int] =
org.apache.spark.graphx.impl.GraphImpl@c5afb2f
```

Import the GraphStream library

Import the GraphStream library

```
import org.graphstream.graph.implementations._
```

Create a new instance of GraphStream's SingleGraph class using the countriesGraph.

```
val graph: SingleGraph = new SingleGraph("countriesGraph")
```

Set up the visual attributes for graph visualization.

```
graph.addAttribute("ui.stylesheet","url(file:.//style/stylesheet)")
graph.addAttribute("ui.quality")
graph.addAttribute("ui.antialias")
```

Load the graphX vertices into GraphStream nodes.

```
for ((id:VertexId, place:PlaceNode) <- countriesGraph.vertices.collect())
{
  val node = graph.addNode(id.toString).asInstanceOf[SingleNode]
  node.addAttribute("name", place.name)
  node.addAttribute("ui.label", place.name)

  if (place.isInstanceOf[Metro])
    node.addAttribute("ui.class", "metro")
  else if(place.isInstanceOf[Country])
    node.addAttribute("ui.class", "country")
  else if(place.isInstanceOf[Continent])
    node.addAttribute("ui.class", "continent")
}
```

Load the graphX edges into GraphStream edges.

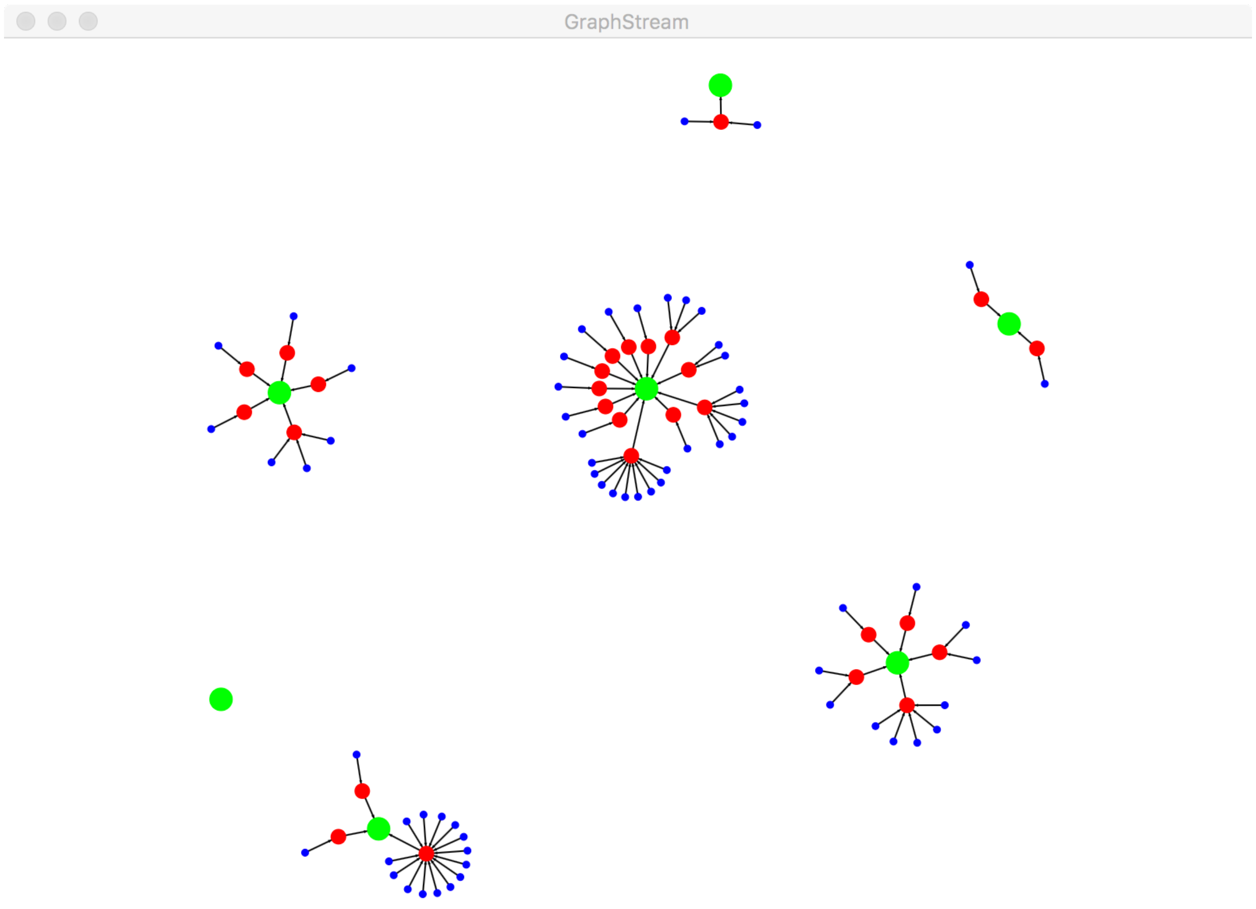
```
for (Edge(x,y,_) <- countriesGraph.edges.collect()) {
  graph.addEdge(x.toString ++ y.toString, x.toString, y.toString,
true).asInstanceOf[AbstractEdge]
}
```

Display the graph. The metros are the small blue dots, the countries are the medium red dots and the continents are the large green dots.

input:

```
graph.display()
```

output:



Visualize Facebook graph

Open a terminal in the Cloudera Quick Start virtual machine by clicking **Applications**, **System Tools** then **Terminal.**

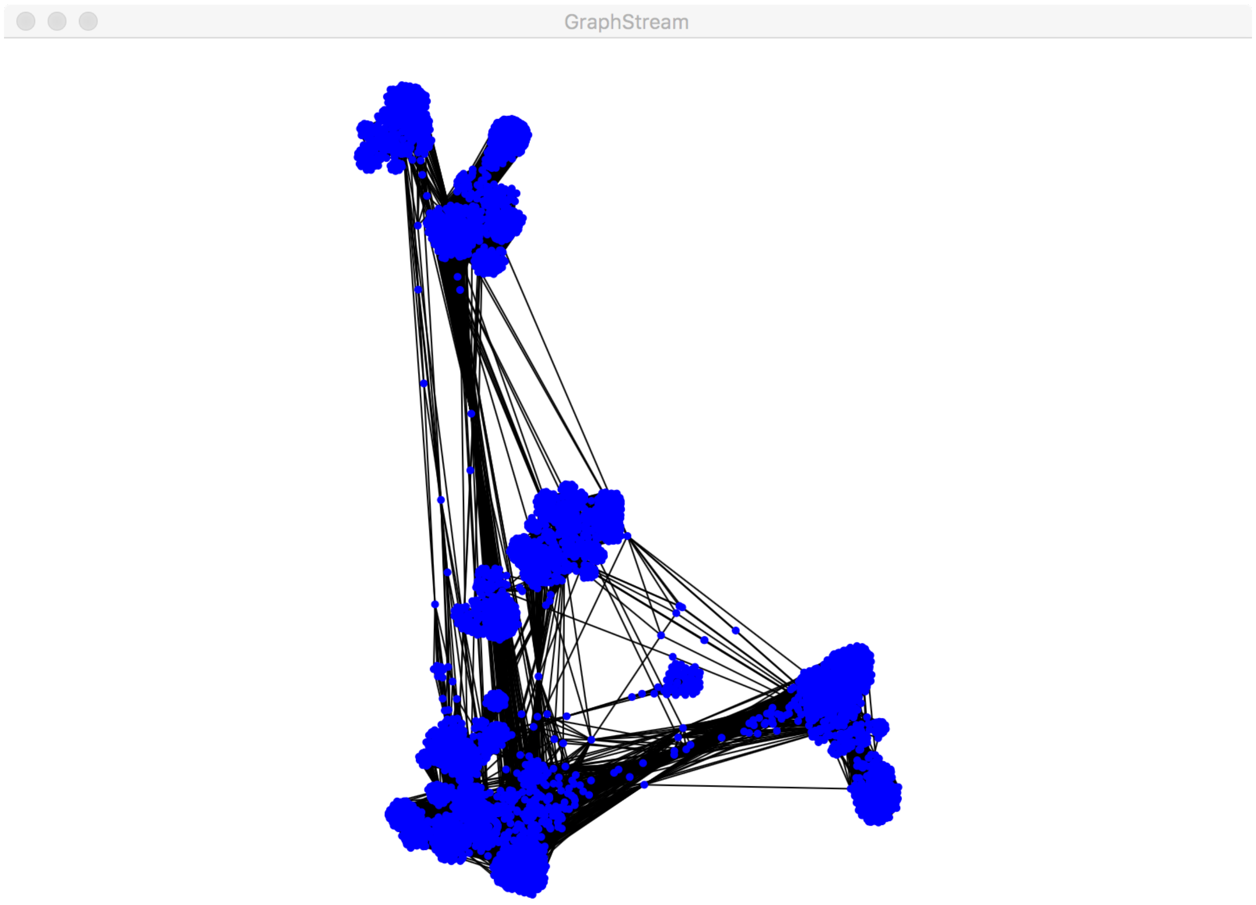
Type the following command in the Terminal window to go into the ExamplesOfAnalytics directory.

```
cd ExamplesOfAnalytics
```

Run the spark-shell with the Facebook.scala file to visualize the Facebook dataset.

```
spark-shell --jars lib/gs-core-1.2.jar,lib/gs-ui-1.2.jar,lib/jcommon-
1.0.16.jar,lib/jfreechart-1.0.13.jar,lib/breeze_2.10-0.9.jar,lib/breeze-viz_2.10-
0.9.jar,lib/pherd-1.0.jar -i Facebook.scala
```

output:



Hands On: Joining Graph Datasets

1. Create a new dataset
2. Join two datasets with JoinVertices
3. Join two datasets with outerJoinVertices
4. Create a new return type for the joined vertices

Run the Spark Shell

Open a terminal in the Cloudera Quick Start virtual machine by clicking **Applications**, **System Tools** then **Terminal.**

Once the terminal is open, start the Spark Shell.

```
spark-shell
```

It may take several seconds for the Spark Shell to start. Be patient and wait for the **scala>** prompt.

Create a new dataset

Set log level to error in order to suppress the info and warn messages so the output is easier to read.

```
import org.apache.log4j.Logger
import org.apache.log4j.Level

Logger.getLogger("org").setLevel(Level.ERROR)
Logger.getLogger("akka").setLevel(Level.ERROR)
```

Import the GraphX and RDD libraries.

```
import org.apache.spark.graphx._
import org.apache.spark.rdd._
```

Define a simple list of vertices containing five international airports.

input:

```
val airports: RDD[(VertexId, String)] = sc.parallelize(
    List((1L, "Los Angeles International Airport"),
      (2L, "Narita International Airport"),
      (3L, "Singapore Changi Airport"),
      (4L, "Charles de Gaulle Airport"),
      (5L, "Toronto Pearson International Airport")))
```

output:

```
airports: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, String)] =
ParallelCollectionRDD[0] at parallelize at <console>:29
```

Two airports are connected in this graph if there is a flight between them. Define a list of edges that will make up the flights for this hands on exercise. We will assign a made up flight number to each flight.

input:

```
val flights: RDD[Edge[String]] = sc.parallelize(
  List(Edge(1L,4L,"AA1123"),
    Edge(2L, 4L, "JL5427"),
    Edge(3L, 5L, "SQ9338"),
    Edge(1L, 5L, "AA6653"),
    Edge(3L, 4L, "SQ4521")))
```

output:

```
flights: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[1] at parallelize at <console>:29
```

Define the flightGraph graph from the airports vertices and the flights edges.

input:

```
val flightGraph = Graph(airports, flights)
```

output:

```
flightGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@5051d760
```

Each triplet in the flightGraph graph represents a flight between two airports. Print the departing and arrival airport and the flight number for each triplet in the flightGraph graph.

input:

```
flightGraph.triplets.foreach(t => println("Departs from: " + t.srcAttr + " - Arrives
at: " + t.dstAttr + " - Flight Number: " + t.attr))
```

output:

```
Departs from: Los Angeles International Airport - Arrives at: Charles de Gaulle
Airport - Flight Number: AA1123
Departs from: Los Angeles International Airport - Arrives at: Toronto Pearson
International Airport - Flight Number: AA6653
Departs from: Narita International Airport - Arrives at: Charles de Gaulle Airport -
Flight Number: JL5427
Departs from: Singapore Changi Airport - Arrives at: Charles de Gaulle Airport -
Flight Number: SQ4521
Departs from: Singapore Changi Airport - Arrives at: Toronto Pearson International
Airport - Flight Number: SQ9338
```

Lets define a dataset with airport information so we can practice joining the airport information dataset with the datasets that we have already defined.

Define an AirportInformation class to store the airport city and code.

input:

```
case class AirportInformation(city: String, code: String)
```

output:

```
defined class AirportInformation
```

Define the list of airport information vertices.

Note: We do not have airport information defined for each airport in **flightGraph** graph and we have airport information for airports not in **flightGraph** graph.

input:

```
val airportInformation: RDD[(VertexId, AirportInformation)] = sc.parallelize(
  List((2L, AirportInformation("Tokyo", "NRT")),
    (3L, AirportInformation("Singapore", "SIN")),
    (4L, AirportInformation("Paris", "CDG")),
    (5L, AirportInformation("Toronto", "YYZ")),
    (6L, AirportInformation("London", "LHR")),
    (7L, AirportInformation("Hong Kong", "HKG"))))
```

output:

```
airportInformation: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
AirportInformation)] = ParallelCollectionRDD[19] at parallelize at <console>:31
```

Join two datasets with JoinVertices

In this first example we are going to use joinVertices to join the airport information **flightGraph** graph.

Create a mapping function that appends the city name to the name of the airport. The mapping function should return a string since that is the vertex attribute type of the flightsGraph graph.

input:

```
def appendAirportInformation(id: VertexId, name: String, airportInformation:
AirportInformation): String = name + ":"+ airportInformation.city
```

output:

```
appendAirportInformation: (id: org.apache.spark.graphx.VertexId, name: String,
airportInformation: AirportInformation)String
```

Use joinVertices on flightGraph to join the airportInformation vertices to a new graph called flightJoinedGraph using the appendAirportInformation mapping function.

input:

```
val flightJoinedGraph =
flightGraph.joinVertices(airportInformation)(appendAirportInformation)
```

```
flightJoinedGraph.vertices.foreach(println)
```

output:

```
(4,Charles de Gaulle Airport:Paris)
(1,Los Angeles International Airport)
(3,Singapore Changi Airport:Singapore)
(5,Toronto Pearson International Airport:Toronto)
(2,Narita International Airport:Tokyo)
```

Join two datasets with outerJoinVertices

Use outerJoinVertices on flightGraph to join the airportInformation vertices with additional airportInformation such as city and code, to a new graph called flightOuterJoinedGraph using
the => operator which is just syntactic sugar for creating instances of functions.

input:

```
val flightOuterJoinedGraph =
flightGraph.outerJoinVertices(airportInformation)((_,name, airportInformation) =>
(name, airportInformation))
flightOuterJoinedGraph.vertices.foreach(println)
```

output:

```
(4,(Charles de Gaulle Airport,Some(AirportInformation(Paris,CDG))))
(1,(Los Angeles International Airport,None))
(3,(Singapore Changi Airport,Some(AirportInformation(Singapore,SIN))))
(5,(Toronto Pearson International Airport,Some(AirportInformation(Toronto,YYZ))))
(2,(Narita International Airport,Some(AirportInformation(Tokyo,NRT))))
```

Use outerJoinVertices on flightGraph to join the airportInformation vertices with additional airportInformation such as city and code, to a new graph called flightOuterJoinedGraphTwo
but this time printing 'NA' if there is no additional information.

input:

```
val flightOuterJoinedGraphTwo = flightGraph.outerJoinVertices(airportInformation)((_,
name, airportInformation) => (name,
airportInformation.getOrElse(AirportInformation("NA","NA"))))
flightOuterJoinedGraphTwo.vertices.foreach(println)
```

output:

```
(4,(Charles de Gaulle Airport,AirportInformation(Paris,CDG)))
(1,(Los Angeles International Airport,AirportInformation(NA,NA)))
(3,(Singapore Changi Airport,AirportInformation(Singapore,SIN)))
```

```
(5,(Toronto Pearson International Airport,AirportInformation(Toronto,YYZ)))
(2,(Narita International Airport,AirportInformation(Tokyo,NRT)))
```

Create a new return type for the joined vertices

Create a case class called Airport to store the information for the name, city, and code of the airport.

input:

```
case class Airport(name: String, city: String, code: String)
```

output:

```
defined class Airport
```

Print the airportInformation with the name, city, and code within each other.

input:

```
  val flightOuterJoinedGraphThree =
flightGraph.outerJoinVertices(airportInformation)((_, name, b) => b match {
  case Some(airportInformation) => Airport(name, airportInformation.city,
airportInformation.code)
  case None => Airport(name, "", "")
})
flightOuterJoinedGraphThree.vertices.foreach(println)
```

output:

```
(4,Airport(Charles de Gaulle Airport,Paris,CDG))
(1,Airport(Los Angeles International Airport,,))
(3,Airport(Singapore Changi Airport,Singapore,SIN))
(5,Airport(Toronto Pearson International Airport,Toronto,YYZ))
(2,Airport(Narita International Airport,Tokyo,NRT))
```