School of Computer Science

COMP47470

Lab 7
Spark GraphX

Teaching Assistant:	Patrick Cormac English
Coordinator:	Dr Anthony Ventresque
Date:	Wednesday 7th April, 2021
Total Number of Pages:	7

1 Introduction

In this lab, we will be using GraphX, a Spark library for graph representation and manipulation. First, we will build a small graph manually to understand the different components of a graph and the different attributes we have access to on a graph, and then we will build a similar, bigger graph from data in a file.

You can find more information about GraphX in the GraphX programming guide and in the GraphX ScalaDoc.

You can restart the docker containers you used in the previous lab using the following commands:

```
$ docker start spark-master
$ docker start spark-worker-1
```

Then, in the spark-master container:

```
$ spark-shell
```

2 Small Graph

In this section we will create and manipulate a small graph representing airports.

We define the vertices - but first we import the GraphX packages.

```
scala> import org.apache.spark._
scala> import org.apache.spark.rdd.RDD
// import classes required for using GraphX
scala> import org.apache.spark.graphx._
```

We define airports as **vertices**. Vertices have an Id and can have properties or attributes associated with them. In our example, each vertex consists of :

- Vertex id → Id (Long)
- Vertex Property → name (String)

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

```
// create vertices RDD with ID and Name
scala> val vertices=Array((1L, ("SFO")),(2L, ("ORD")),(3L,("DFW")))
scala> val vRDD= sc.parallelize(vertices)
scala> vRDD.take(1)
// Array((1,SFO))

// Defining a default vertex called nowhere
scala> val nowhere = "nowhere"
```

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)

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
scala> val edges = Array(Edge(1L,2L,1800),Edge(2L,3L,800),Edge(3L,1L,1400))
scala> val eRDD= sc.parallelize(edges)

scala> eRDD.take(2)
// Array(Edge(1,2,1800), Edge(2,3,800))
```

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

```
// define the graph
scala> val graph = Graph(vRDD,eRDD, nowhere)
// graph vertices
scala> graph.vertices.collect.foreach(println)
// (2,ORD)
// (1,SFO)
// (3,DFW)

// graph edges
scala> graph.edges.collect.foreach(println)

// Edge(1,2,1800)
// Edge(2,3,800)
// Edge(3,1,1400)
```

Let's explore our dataset:

1. How many airports are there?

   ```
   scala> val numairports = graph.numVertices
   // Long = 3
   ```

2. How many routes are there?

   ```
   scala> val numroutes = graph.numEdges
   // Long = 3
   ```

3. which routes are >1000 miles of distance?

   ```
   scala> graph.edges.filter { case Edge(src, dst, prop) => prop > 1000
   ↪  }.collect.foreach(println)
   // Edge(1,2,1800)
   // Edge(3,1,1400)
   ```

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

```
scala> graph.triplets.take(3).foreach(println)
((1,SFO),(2,ORD),1800)
((2,ORD),(3,DFW),800)
((3,DFW),(1,SFO),1400)
```

5. Which are the longuest routes?

```
scala> graph.triplets.sortBy(_.attr, ascending=false).map(triplet =>
     "Distance " + triplet.attr.toString + " from " + triplet.srcAttr
     ↪  + " to " + triplet.dstAttr + ".").collect.foreach(println)
```

3 Real Flight Data

Download the dataset here. A smaller dataset is available here if you have problems with the big dataset (e.g. spark crashing). This data comes from this website. We are using flight information for November 2019. For each flight, we have the information listed in Table 1.

Field name and type	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

Table 1: Flight info

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

First we will import the GraphX packages.

```
scala> import org.apache.spark._
scala> import org.apache.spark.rdd.RDD
scala> import org.apache.spark.util.IntParam
// import classes required for using GraphX
```

```
scala> import org.apache.spark.graphx._
scala> import org.apache.spark.graphx.util.GraphGenerators
```

Below we use Scala case classes to define the flight schema corresponding to the csv data file.

```
// define the Flight Schema
scala> 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
scala> 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). RDDs can have transformations and actions, the first() action returns the first element in the RDD.

```
// load the data into a RDD variable
scala> var textRDD = sc.textFile("/path/to/the/file/data_lab_6.csv")
// MapPartitionsRDD[1] at textFile

//get the header of the CSV file into a value (can not be modified)
scala> val header = textRDD.first()
//header: String =
 ↪  DayofMonth,DayOfWeek,Reporting_Airline,Tail_Number,Flight_Number_Reporting_Airline,Orig

//filter out the header (that we can not parse to a Flight)
scala> textRDD = textRDD.filter(row => row != header)
//MapPartitionsRDD[2] at filter

// parse the RDD of csv lines into an RDD of flight classes
scala> val flightsRDD = textRDD.map(parseFlight).cache()
//MapPartitionsRDD[3] at map
```

We define airports as vertices. Vertices can have properties or attributes associated with them. Each vertex has the Airport name (String) as a property.

We define an RDD with the above property and id for each airport, that is then used for the vertices.

```
// create airports RDD with ID and Name
scala> val airports = flightsRDD.flatMap(flight => Seq((flight.org_id,
↪  flight.origin),(flight.dest_id, flight.dest))).distinct
//RDD[(Long, String)] = MapPartitionsRDD[7] at distinct

scala> airports.take(1)
// Array((12917,LCK))

// Defining a default vertex called nowhere
scala> val nowhere = "nowhere"

// Map airport ID to the 3-letter code to use to print human-friendly
↪  messages
scala> val airportMap = airports.map { case ((org_id), name) => (org_id ->
↪  name) }.collect.toMap
// Map(10785 -> BTV, 10577 -> BGM,...)
```

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)

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
scala> val routes = flightsRDD.map(flight => ((flight.org_id,
↪  flight.dest_id), flight.dist)).distinct
// MapPartitionsRDD[12] at distinct

scala> routes.take(2)
// Array(((14100,12478),94), ((14869,14683),1087))

// create edges RDD with srcid, destid , distance as Edge objects
scala> val edges = routes.map {
 case ((org_id, dest_id), distance) =>Edge(org_id, dest_id, distance) }
// MapPartitionsRDD[13] at map

scala> edges.take(1)
//Array(Edge(14100,12478,94))
```

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
scala> val graph = Graph(airports, edges, nowhere)
```

```
// graph vertices
scala> graph.vertices.take(2)
// Array((10208,AGS), (12264,IAD))

// graph edges
scala> graph.edges.take(2)
// Array(Edge(10135,10397,692), Edge(10135,10693,685))
```

1. How many airports are there?

   ```
   scala> val numairports = graph.numVertices
   // Long = 348
   ```

2. How many routes are there?

   ```
   scala> val numroutes = graph.numEdges
   // Long = 5631
   ```

3. Which routes are >1000 miles distance of distance?

   ```
   scala>  graph.edges.filter { case ( Edge(org_id, dest_id,distance))=>
   ↪  distance > 1000}.take(3)
   // Array(Edge(10135,14082,1018), Edge(10140,10397,1269),
   ↪  Edge(10140,10821,1670))
   ```

4. The EdgeTriplet class extends the edge class by adding the srcAttr and dstAttr members which contain the source and destination properties, respectively.

   ```
   scala> graph.triplets.take(3).foreach(println)
   //((10135,ABE),(10397,ATL),692)
   //((10135,ABE),(10693,BNA),685)
   //((10135,ABE),(13577,MYR),518)
   ```

5. Sort and print out the longest distance routes

   ```
   scala> graph.triplets.sortBy(_.attr, ascending=false).map(triplet =>
        "Distance " + triplet.attr.toString + " from " + triplet.srcAttr
        ↪  + " to " + triplet.dstAttr + ".").take(10).foreach(println)

   //Distance 5095 from BOS to HNL.
   //Distance 5095 from HNL to BOS.
   //Distance 4983 from JFK to HNL.
   //Distance 4983 from HNL to JFK.
   //Distance 4962 from EWR to HNL.
   //Distance 4962 from HNL to EWR.
   //Distance 4817 from HNL to IAD.
   //Distance 4817 from IAD to HNL.
   //Distance 4502 from ATL to HNL.
   //Distance 4502 from HNL to ATL.
   ```

6. Compute the highest degree vertex

```
// Define a reduce operation to compute the highest degree vertex
scala> def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId,
↪  Int) = {
 if (a._2 > b._2) a else b
}

scala> val maxInDegree: (VertexId, Int) = graph.inDegrees.reduce(max)
//maxInDegree: (org.apache.spark.graphx.VertexId, Int) = (11298,175)

scala> val maxOutDegree: (VertexId, Int) =
↪  graph.outDegrees.reduce(max)
//maxOutDegree: (org.apache.spark.graphx.VertexId, Int) = (11298,175)

scala> val maxDegrees: (VertexId, Int) = graph.degrees.reduce(max)
//maxDegrees: (org.apache.spark.graphx.VertexId, Int) = (11298,350)

// Get the name for the airport with id 10397
scala> airportMap(11298)
//res70: String = DFW
```

7. Which airport has the most incoming flights?

```
// get top 3
scala> val maxIncoming = graph.inDegrees.collect.sortWith(_._2 >
↪  _._2).map(x => (airportMap(x._1), x._2)).take(3)

scala> maxIncoming.foreach(println)
//(DFW,175)
//(ORD,163)
//(DEN,159)

// which airport has the most outgoing flights?
scala> val maxout= graph.outDegrees.join(airports).sortBy(_._2._1,
↪  ascending=false).take(3)

scala> maxout.foreach(println)
//(11298,(175,DFW))
//(13930,(163,ORD))
//(11292,(158,DEN))
```