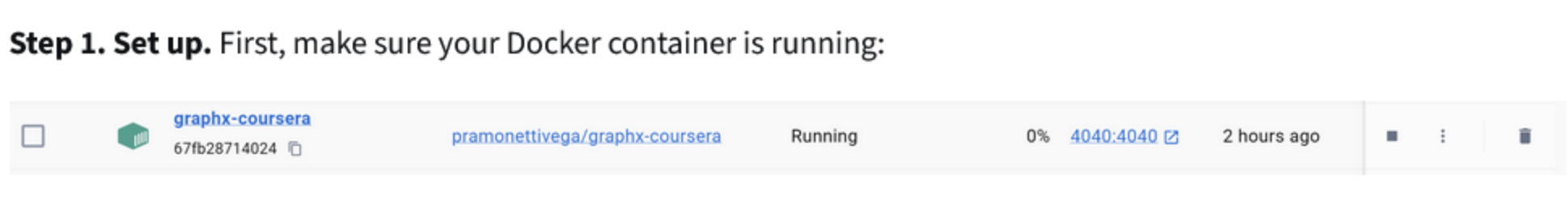
Hands On: Joining Graph Datasets

In this activity, we are going to:

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

Step 1. Set up. First, make sure your Docker container is running:



Once you verify the Docker container is running, run the following command to access the container's shell:

```
docker exec -it graphx-coursera /bin/sh
```

Once the container's shell 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.

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._
```

Step 2. Create a new dataset.

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)] = ParallelCo
```

Define a list of edges that will make up the flights.

Two airports are connected in this graph if there is a flight between them. 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]] = ParallelCollectio
```

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.Gr
```

Print the departing and arrival airport and the flight number for each triplet in the flightGraph graph.

Each triplet in the *flightGraph* graph represents a flight between two airports.

Input:

```
flightGraph.triplets.foreach(t => println("Departs from: " + t.srcAttr + " - Arrives at: "
```

Output:

```
Departs from: Los Angeles International Airport - Arrives at: Charles de Gaulle Airport - F
Departs from: Narita International Airport - Arrives at: Charles de Gaulle Airport - Flight
Departs from: Singapore Changi Airport - Arrives at: Toronto Pearson International Airport
Departs from: Los Angeles International Airport - Arrives at: Toronto Pearson International
Departs from: Singapore Changi Airport - Arrives at: Charles de Gaulle Airport - Flight Num
```

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

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

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, AirportInfo
```

Step 3. 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: AirportInforma
```

Output:

```
appendAirportInformation: (id: org.apache.spark.graphx.VertexId, name: String, airportInfor
```

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

Output:

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

Step 4. 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, air
flightOuterJoinedGraph.vertices.foreach(println)
```

Output:

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

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,
flightOuterJoinedGraphTwo.vertices.foreach(println)
```

Output:

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

Step 5. 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:

```
```

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

Input:

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

Output:

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

Go to next item ✓ Completed

Like Dislike Report an issue