Big Data Final Project

Analysing Flight Data using Apache Spark GraphX and GraphFrames Using Scala On Databricks

Jenny Savani : 0675557
Shreya Walia : 0633359

Introduction

Motivation of the project:

Graphs are present in our everyday lives without us realizing it. A public transit map, like a metro map or map on the GPS navigation system are examples of graphs wherein the graphs represent some geographical locations, the buses or metro stops or villages are the area and the connections between them are bus routes or the roads. So the bus stops or other locations act as vertices and the routes act as the edges. Similar is for flight data where different airports act as stops and there is a route between them. There are other graphs such as the social network graphs wherein the social graphs comprise of the people we have interaction with such as messaging, following, viewing posts. Similarly when we make or receive calls we are again forming graphs with records. Bank transactions involving sending and receiving money can be viewed as a graph as well. Therefore, a lot of data in real life can be used to form graphs and it becomes necessary to be able to work with this form of graphical data to derive better insights from it.

Through this project we aim at analyzing flight data which will be represented in graphical form i.e. in form of vertices and edges. Our motivation is to be able to work on this data and use data analysis techniques to draw meaning information from it.

Research Problem:

There are hundreds of thousands of flights that run around the world each day and 36.8 million fly each year(according to IATA-2017). To be able to work properly and efficiently with the data being generated from the daily, it needs to be represented in the form of a graph because a graph database can make it easier to work with data and express certain kinds of queries.

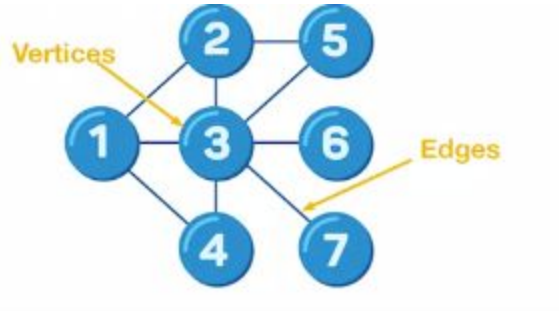
The research problem for our project is to be able to perform analysis on graphical flight data and use two approaches to do so. We are using GraphX and GraphFrame Apache Spark libraries to understand working with graphical data and being able to compare the two approaches.

Dataset:

The data under review is obtained from "Bureau of Transportation Statistics" which is the official site of the US government containing air transit data. We are using data for January 2019. Our file is approximately 56 MB and contains 583985 observations and 21 data fields. We are using useful fields as per our requirements for queries and analysis.

What is a graph?

In mathematical terms, a graph can be defined as "a structure relating to a set of objects in which some pair of objects are in some sense related." The objects correspond to the vertices and the lines between each of the related pairs of vertices are the edges which can be either undirected or directed. Below is a graph of 7 vertices and 9 edges.



Graph databases represent the data using graph structures, which include vertices, edges, and properties. The graph relates the stored data items to a collection of vertices and edges, the edges representing the relationships between the vertices**.** The relationships allow data to be linked together directly and, in many cases, retrieved with one operation. Graph databases hold the relationships between data as a priority. Performing queries in graph databases becomes easy, fast, and convenient; because they are incessantly stored within the database itself. Relationships can be intuitively visualized using graph databases, making them useful for heavily interconnected data.

Flight data in terms of graph:

In graphical form the airports can be looked at as being the vertices and the flight route from source to destination between these airports will lead to creation of the edge between the airport vertices.

Approach

Platform and libraries used:

This project has been built on databricks using Apache Spark framework and Scala language using 2 graph processing libraries namely GraphX and GraphFrames.

Databricks:

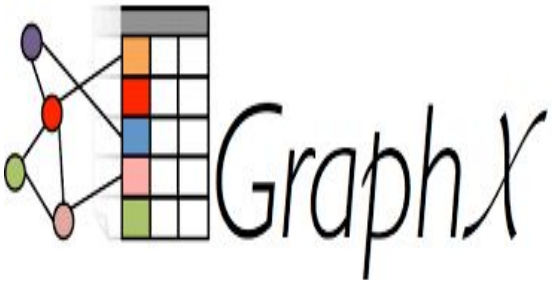
Databricks is an open source platform for unified data analytics. It is available on cloud and needs no installation. Databricks provides the user with a web-based platform for working with Spark. It provides automated cluster management and databrick notebooks to write code in. We use the cluster with spark version 2.4 scala 2.11, DBR 6.4, 15.25 GB memory and 2 cores.

Apache Spark:

Apache Spark is an open source tool for performing distributed computations and analyzing a huge amount of data. Data and different tasks are distributed across the cluster and processed parallely. Hence, it functions similar to MapReduce in Hadoop. It possesses a variety of data repositories like NoSQL databases, Hadoop Distributed File System (HDFS), relational data stores, such as Apache Hive, et cetera. In order to overcome the limitation in the MapReduce cluster computing paradigm, which has purely disk-based processing and uses persistent storage, Apache Spark offers RDD (Resilient Distributed Datasets) as its basic data type. RDDs in Apache Spark work as a set for distributed programs and have distributed shared memory in restricted form. Due to this, it offers better performance for the data on the dedicated clusters. Apache Spark supports Java, Python, Scala, and R programming languages.

GraphX:

GraphX is an API of Apache Spark. It is used for graphs and its related distributed computations. It consists of various graph iterative algorithms and builders which enables graph analysis. GraphX is like the Spark in-memory version of Apache Giraph. But it is faster than Apache Giraph and slightly slower than GraphLab.



GraphX consists of the Spark RDD with a resilient distributed property graph. This property graph is like a directed multigraph with multiple edges in parallel. Edges and vertices in the graph can have associated user defined properties, along with multiple relationships between the vertices. GraphX performs exploratory analysis, transformation of data and computing to get results. It allows viewing data as both graphs and collections.

GraphFrames:

GraphFrames is a graph processing library for Apache spark compatible with versions of spark above 1.4. GraphFrames benefits from scalability and high performance of data frames and provides uniform api for graph processes. It is available from scala, java and python. It supports general graph processing algorithms and GraphFrames is built on top of spark data frames which gives some key advantages.

- We can use GraphFrames to phrase queries in the familiar API of spark SQL and spark data frames.
- Fully support data frame data sources which allows writing and reading data from many formats like json, csv.
- We can use motif finding for pattern matching on data frames.

Solution Frameworks

Using GraphX to work with flight data:

(This part of the project has been worked upon by Jenny Savani:)

In this project we are using the Scala programming language and dataset in the form of graph structure. For this we need to convert our regular data (with rows and columns) into graph structure.

Transforming and loading the data:

For loading the dataset in the Spark Apache environment, we define the schema corresponding to the dataset that we have using variable names and data types.

Then we define a parsing function, which parses lines from the input data present in the csv file according to the schema that we have defined earlier. Once the parsing function for input data is defined, we finally load our data from the csv file in a RDD and call the parsing function. In order to create a property graph for this data, we create a Vertex RDD, Edge RDD, and a default vertex. For this project, we have used airports as vertices and routes as edges.

Once the graph structure for our flight data is created, we carry out desired queries using Scala, which has been discussed briefly in the next section. Apart from these queries, we have also used PageRank algorithm and Pregel algorithm.

PageRank:

Distributed processing of graph databases offers the iterative nature of many algorithms unlike MapReduce method. Google based PageRank algorithm is present as a predefined function in GraphX.

PageRank algorithm measures the importance of each vertex (attributes) in the graph. It determines the vertices that have the most number of edges with the other vertices. We are using PageRank algorithm to determine the most important (busiest) airports (vertex) from our data, depending upon the maximum number of connections with the other airports.

Pregel:

Distributed processing of graph databases offers the iterative nature of many algorithms unlike MapReduce method. Pregel is one kind of the vertex-centric graph processing iterative algorithm supported by Apache Spark GraphX and it comes along with its own predefined implementation.

The Pregel algorithm is used to traverse the graph and it is called upon the property graph as a method. This method includes following steps:

- Each vertex sends messages to the other neighboring vertices.
- The graph is processed in iterations, which is called superstep.
- Each superstep does the following:
    - Messages from the previous iterations are received at each vertex.
    - This vertex runs the function and transforms itself accordingly.
    - Now this vertex will send messages to the other neighboring vertices.

This method upon the end of traversing will return a newly transformed graph with the same structure and data type. Edges remain the same, but the attributes of the vertices change from one superstep to the next superstep.

Iteration terminates once the threshold or maximum number of iterations is achieved. At the end of each iteration, a reduce function is applied, which computes the new value of each vertex in iteration (mostly it returns the minimum value). If the value is found to be changed, then this new value (message) is transferred to the neighboring vertices in the next superstep.

The Pregel method in GraphX allows the exchanged messages to access attributes of both, source and destination vertices.

We are using the Pregel method to determine which airports (or routes between the pair of source and destination airports) have the lowest flight fare and how much.

Steps/Implementation:

- Importing necessary packages for GraphX:

```
// importing packages required for GraphX
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators

```

- Defining schema for the flight data, Data_flight using case class :

```
//Creating case class shcema for our data
case class Data_flight(trip_id:Int, day_of_month:String, day_of_week:String, carrier_code:String, tail_num:String, flight_num:String, origin_id:Long,
origin:String, dest_id:Long, dest:String, dest_state:String, scheduled_dept_time:String, actual_dept_time:String, dep_delay_mins:String,
scheduled_arr_time:String, actual_arr_time:String, arr_delay_mins:String,elapsed_time_mins:String, total_flights:Long, distance:Int, city:String)
```

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
defined class Data_flight

Command took 0.54 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:48:18 AM on new_project
```

- Creating a function to parse the input flight data, parse_flight_data:

```
Cmd 2

// Creating parse_flight_data function for parsing the file, whenever we load it
def parse_flight_data(str: String): Data_flight = {
  val line = str.split(",")
  Data_flight(line(0).toInt, line(1), line(2), line(3), line(4), line(5), line(6).toLong, line(7), line(8).toLong, line(9), line(10), line(11), line(12),
line(13), line(14), line(15), line(16), line(17), line(18).toLong, line(19).toInt, line(20).toString)
}

parse_flight_data: (str: String)Data_flight

Command took 0.19 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:20:56 PM on new_cluster
```

- Loading flight data into a RDD, data_RDD and calling the parsing function to parse the input data and saving parsed data as flights_RDD:

Cmd 3

```
//loading csv file of our data in data_RDD
val data_RDD = sc.textFile("/FileStore/tables/data3.csv")
//View of our data
data_RDD.toDF().show(10,false)

//parsing the input file from data_RDD using parse_flight_data function and saving it in flights_RDD
val flights_RDD = data_RDD.map(parse_flight_data).cache()
```

▸ (1) Spark Jobs

```
+---------------------------------------------------------------------------------------------------------+
|value                                                                                                    |
+---------------------------------------------------------------------------------------------------------+
|222,1,2,9E,N8688C,3280,11953,GNV,10397,ATL,GA,600,601,1,723,722,-1,83,1,300,"Atlanta, GA"                |
|223,1,2,9E,N348PQ,3281,13487,MSP,11193,CVG,KY,1404,1359,-5,1709,1633,-36,125,1,596,"Cincinnati, OH"      |
|224,1,2,9E,N8896A,3282,11433,DTW,11193,CVG,KY,1220,1215,-5,1345,1329,-16,85,1,229,"Cincinnati, OH"       |
|225,1,2,9E,N8886A,3283,15249,TLH,10397,ATL,GA,1527,1521,-6,1639,1625,-14,72,1,223,"Atlanta, GA"          |
|226,1,2,9E,N8974C,3284,10397,ATL,11778,FSM,AR,1902,1847,-15,2005,1940,-25,123,1,579,"Fort Smith, AR"     |
|227,1,2,9E,N927EV,3285,11267,DAY,13487,MSP,MN,900,853,-7,1012,953,-19,132,1,574,"Minneapolis, MN"        |
|228,1,2,9E,N915XJ,3286,12448,JAN,10397,ATL,GA,1558,1553,-5,1823,1832,9,85,1,341,"Atlanta, GA"            |
|229,1,2,9E,N295PQ,3287,12953,LGA,11193,CVG,KY,1555,1551,-4,1821,1824,3,146,1,585,"Cincinnati, OH"        |
|230,1,2,9E,N337PQ,3288,12451,JAX,12953,LGA,NY,1045,1037,-8,1301,1239,-22,136,1,833,"New York, NY"        |
|231,1,2,9E,N311PQ,3289,10397,ATL,10685,BMI,IL,1245,1245,0,1332,1318,-14,107,1,533,"Bloomington/Normal, IL"|
+---------------------------------------------------------------------------------------------------------+
only showing top 10 rows

data_RDD: org.apache.spark.rdd.RDD[String] = /FileStore/tables/data3.csv MapPartitionsRDD[2182] at textFile at command-1875235421595032:2
flights_RDD: org.apache.spark.rdd.RDD[Data_flight] = MapPartitionsRDD[2185] at map at command-1875235421595032:7
```

Command took 1.77 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:48:32 AM on new_project

- View of our data.

Cmd 4

```
//View of how our data looks like
flights_RDD.toDF().show(10)
```

▾ (1) Spark Jobs
▸ Job 210 View (Stages: 1/1)

```
+-------+------------+-----------+------------+--------+----------+---------+------+-------+----+----------+-------------------+-----------------+------------
--+------------------+---------------+--------------+-----------------+-------------+--------+-------------------+
|trip_id|day_of_month|day_of_week|carrier_code|tail_num|flight_num|origin_id|origin|dest_id|dest|dest_state|scheduled_dept_time|actual_dept_time|dep_delay_mi
ns|scheduled_arr_time|actual_arr_time|arr_delay_mins|elapsed_time_mins|total_flights|distance|               city|
+-------+------------+-----------+------------+--------+----------+---------+------+-------+----+----------+-------------------+-----------------+------------
--+------------------+---------------+--------------+-----------------+-------------+--------+-------------------+
|    222|           1|          2|          9E|  N8688C|      3280|    11953|   GNV|  10397| ATL|        GA|                600|              601|
1|               723|            722|            -1|               83|            1|     300|           "Atlanta|
|    223|           1|          2|          9E|  N348PQ|      3281|    13487|   MSP|  11193| CVG|        KY|               1404|             1359|
-5|              1709|           1633|           -36|              125|            1|     596|        "Cincinnati|
|    224|           1|          2|          9E|  N8896A|      3282|    11433|   DTW|  11193| CVG|        KY|               1220|             1215|
-5|              1345|           1329|           -16|               85|            1|     229|        "Cincinnati|
|    225|           1|          2|          9E|  N8886A|      3283|    15249|   TLH|  10397| ATL|        GA|               1527|             1521|
-6|              1639|           1625|           -14|               72|            1|     223|           "Atlanta|
|    226|           1|          2|          9E|  N8974C|      3284|    10397|   ATL|  11778| FSM|        AR|               1902|             1847|           -
15|              2005|           1940|           -25|              123|            1|     579|        "Fort Smith|
|    227|           1|          2|          9E|  N927EV|      3285|    11267|   DAY|  13487| MSP|        MN|                900|              853|
-7|              1012|            953|           -19|              132|            1|     574|       "Minneapolis|
|    228|           1|          2|          9E|  N915XJ|      3286|    12448|   JAN|  10397| ATL|        GA|               1558|             1553|
-5|              1823|           1832|             9|               85|            1|     341|           "Atlanta|
|    229|           1|          2|          9E|  N295PQ|      3287|    12953|   LGA|  11193| CVG|        KY|               1555|             1551|
```

Command took 3.23 minutes -- by jennysavani@trentu.ca at 4/17/2020, 4:48:40 AM on new_project

- Defining vertices and a default vertex for the graph:

We are using origin airports as vertex RDD - vertices_airports for our graph. It contains ID of origin airports, along with name of origin airports as its property.

Cmd 5

```
//Creating RDD for airports as vertices, using origin airport ID and airports
val vertices_airports = flights_RDD.map(flight => (flight.origin_id, flight.origin)).distinct
    vertices_airports.take(5)
```

(1) Spark Jobs

```
vertices_airports: org.apache.spark.rdd.RDD[(Long, String)] = MapPartitionsRDD[4525] at distinct at command-1929007126660343:2
res110: Array[(Long, String)] = Array((12917,LCK), (14057,PDX), (11013,CIU), (12177,HOB), (14262,PSP))
```

Command took 3.46 minutes -- by jennysavani@trentu.ca at 4/17/2020, 4:20:56 PM on new_cluster

Cmd 6

```
// Defining a default vertex called default_vertex
val default_vertex = "default_vertex"

```

- Defining edges RDD for the graph:

We are using the routes between the vertices (airports) as edge RDD. Here, route_RDD has a source, a destination, and distance as property.

```
//Creating RDD, routes_RDD for origin_id, destination_id of airports, and distance
val routes_RDD = flights_RDD.map(flight => ((flight.origin_id.toLong, flight.dest_id.toLong), flight.distance.toInt)).distinct
routes_RDD.cache
routes_RDD.take(5)
```

(1) Spark Jobs

```
default_vertex: String = default_vertex
routes_RDD: org.apache.spark.rdd.RDD[((Long, Long), Int)] = MapPartitionsRDD[102] at distinct at command-1875235421595038:5
res119: Array[((Long, Long), Int)] = Array(((10721,12889),2381), ((14869,14683),1087), ((13204,11996),449), ((14683,14771),1482), ((12982,14831),2466))
```

Command took 1.17 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:44:17 AM on new_project

Creating airport_map function to map numerical origin ID of airport to the code of corresponding airport. Later, creating RDD of edges from routes_RDD.

Cmd 7

```
// Since origin_id is numerical - Mapping origin_id to the 3-letter airport code
val airport_map = vertices_airports.map { case ((origin_id), name) => (origin_id -> name) }.collect.toList.toMap.take(15)
```

(1) Spark Jobs

```
airport_map: scala.collection.immutable.Map[Long,String] = Map(10785 -> BTV, 10577 -> BGM, 14574 -> ROA, 14057 -> PDX, 13933 -> ORH, 11898 -> GFK, 14709 -> SCC, 15380 -> TVC, 11308 -> DHN, 10299 -> ANC, 13244 -> MEM, 12094 -> HDN, 12129 -> HIB, 10431 -> AVL, 11603 -> EUG)
```

Command took 0.43 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:21:54 AM on new_project

Cmd 8

```
// Defining the routes_RDD as edges for our graph
val edges = routes_RDD.map { case ((origin_id, dest_id), distance) => Edge(origin_id.toLong, dest_id.toLong, distance) }

edges.take(5)
```

(1) Spark Jobs

```
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] = MapPartitionsRDD[105] at map at command-357071919253792:2
res22: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(10721,12889,2381), Edge(14869,14683,1087), Edge(13204,11996,449), Edge(14683,14771,1482), Edge(12982,14831,2466))
```

Command took 0.39 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:48:15 AM on new_project

- Creating graph of our data:

We are creating the graph structure using the vertex RDD, edge RDD, and default vertex. Default vertex is used in case any edge is pointing to any undefined vertex.

Following image also shows how vertex RDD and edges RDD looks in the graph data.

Performing queries to answer our concerned questions:

1) Find out the total number of airports in the data.

Total vertices are equal to total airports.

2)Find out the total number of routes in the data.

Total edges are equal to routes between the pair of airports.

3)Find the airport ID with the distance greater than 4500 miles between source and destination.

Array of - Edge(Source airport ID, destination airport ID, distance)

```
1 // origin_id, dest_id, and distance of the routes greater than 4500 miles distance?
2 graph.edges.filter { case ( Edge(origin_id, dest_id, distance))=> distance > 4500}.take(5)
```

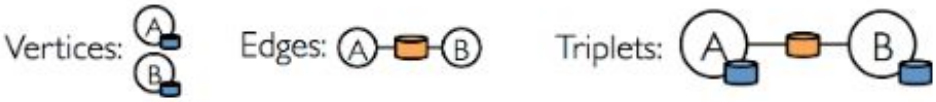
▸ (2) Spark Jobs

```
res24: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(10397,12173,4502), Edge(11618,12173,4962), Edge(12173,11618,4962), Edge(12478,12173,4983), Edge(12173,10397,4502))
```

Command took 0.35 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:48:33 AM on new_project

4)Determine top 5 airports with code having the longest distance between source and destination airports.

Using Triplet view to show a collection of strings describing relationships between vertices and edges. Triplet is extended from Edge class by determining scrAttr and dstAttr, which contains properties of origin airport code and destination airport code respectively.



Triplet shows airport ID and code of origin and destination airports along with distance between the both.

Cmd 14

```
1 //The edge class with the source and destination properties, respectively.
2 graph.triplets.take(5).foreach(println)
```

▸ (1) Spark Jobs

```
((10135,ABE),(10397,ATL),692)
((10135,ABE),(13930,ORD),654)
((10135,ABE),(14082,PGD),1018)
((10135,ABE),(14761,SFB),882)
((10140,ABQ),(10397,ATL),1269)
```

Command took 0.49 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:48:33 AM on new_project

Output of query (in the image below):

```
1 //Longest routes (distance) in our data with source and destination airports. Also sorting it in descending order
2 graph.triplets.sortBy(_.attr, ascending=false).map(triplet =>
3      "Total distance is " + triplet.attr.toString + " from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").take(5).foreach(println)
```

▸ (2) Spark Jobs

```
Total distance is 4983 from JFK to HNL.
Total distance is 4983 from HNL to JFK.
Total distance is 4962 from EWR to HNL.
Total distance is 4962 from HNL to EWR.
Total distance is 4817 from HNL to IAD.
```

Command took 0.88 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:48:33 AM on new_project

5)Find Airport ID which has maximum incoming and outgoing flights separately, with total number.

Defining maximum_function for carrying out reduce operation on vertex of the graph, airports in our case. It returns the maximum of both the inputs.

Using maximum_incoming function with reduce operation on maximum_function and predefined function inDegrees to find maximum incoming flights and airport ID :
Output:(Airport ID, Total incoming) : ((11298,162))

Using maximum_outgoing function with reduce operation on maximum_function and predefined function outDegrees to find maximum outgoing flights and Airport ID:
Output:(Airport ID, Total outgoing) : ((11298,162))

Cmd 16

```
1 // Computing the highest degree vertex by using a reduce operation
2 def maximum_function(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
3  if (a._2 > b._2) a else b
4 }
5
6 //Fetching airport_id with maximum incoming edges or flights
7 val maximum_incoming: (VertexId, Int) = graph.inDegrees.reduce(maximum_function)
```

▸ (1) Spark Jobs

```
maximum_function: (a: (org.apache.spark.graphx.VertexId, Int), b: (org.apache.spark.graphx.VertexId, Int))(org.apache.spark.graphx.VertexId, Int)
maximum_incoming: (org.apache.spark.graphx.VertexId, Int) = (11298,162)
```

Command took 0.45 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:53:31 AM on new_project

Cmd 17

```
1 //Fetching airport_id with maximum outcoming edges or flights
2 val maximum_outgoing: (VertexId, Int) = graph.outDegrees.reduce(maximum_function)
```

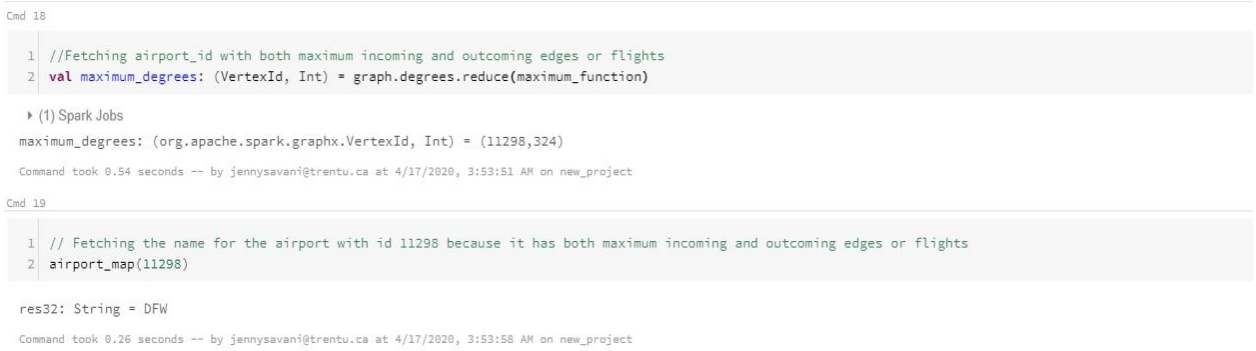
▸ (1) Spark Jobs

```
maximum_outgoing: (org.apache.spark.graphx.VertexId, Int) = (11298,162)
```

Command took 0.44 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:53:46 AM on new_project

6)Find airport code with maximum total of incoming and outgoing flights.

Using maximum_degrees function with reduce operation on maximum_function and predefined function degrees to find maximum total of outgoing and incoming flights and Airport ID :
Output:(Airport ID, Total) : ((11298,324))
Output: DFW (Airport code for 11298 ID)



```
Cmd 18

1 //Fetching airport_id with both maximum incoming and outcoming edges or flights
2 val maximum_degrees: (VertexId, Int) = graph.degrees.reduce(maximum_function)

▸ (1) Spark Jobs
maximum_degrees: (org.apache.spark.graphx.VertexId, Int) = (11298,324)
Command took 0.54 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:53:51 AM on new_project

Cmd 19

1 // Fetching the name for the airport with id 11298 because it has both maximum incoming and outcoming edges or flights
2 airport_map(11298)

res32: String = DFW
Command took 0.26 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:53:58 AM on new_project
```

7)Find top 5 airports with maximum incoming flights.

Again using inDegrees function. Sorting airport codes based on descending order of total incoming flights.

```
Cmd 20

1 // Fetching top 5 airport codes with maximum incoming flights:
2 val max_incoming_flights = graph.inDegrees.collect.sortWith(_._2 > _._2).map(x => (airport_map(x._1), x._2)).take(5)
3
4 max_incoming_flights.foreach(println)

▸ (1) Spark Jobs
(DFW,162)
(ATL,158)
(ORD,153)
(DEN,145)
(CLT,128)
max_incoming_flights: Array[(String, Int)] = Array((DFW,162), (ATL,158), (ORD,153), (DEN,145), (CLT,128))
Command took 0.56 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:55:41 AM on new_project
```

8)Find top 5 airports ID and code with maximum outgoing flights.

Again using outDegrees function. Sorting airport codes based on descending order of total outgoing flights.

Cmd 21

```
1 //Fetching top 5 airport codes (vertices) with maximum outgoing flights:
2 val max_outcoming_flights= graph.outDegrees.join(vertices_airports).sortBy(_._2._1, ascending=false).take(5)
3
4 max_outcoming_flights.foreach(println)
```

▸ (2) Spark Jobs

```
(11298,(162,DFW))
(10397,(158,ATL))
(13930,(154,ORD))
(11292,(143,DEN))
(11057,(128,CLT))
max_outcoming_flights: Array[(org.apache.spark.graphx.VertexId, (Int, String))] = Array((11298,(162,DFW)), (10397,(158,ATL)), (13930,(154,ORD)), (11292,(143,
DEN)), (11057,(128,CLT)))
```

9) Determine most busiest/Important airport in the data.

Using the PageRank algorithm for this. Explanation is done in the Approach part of the document. GraphX has a predefined pageRank function for this purpose, which assigns rank to the vertices (airports) based upon the most number of connections with the other vertices. Giving tolerance value as 0.001.

```
1 // using pageRank function with 0.001 tolerance to rank airports based on maximum number of vertices (routes)
2 val define_rank = graph.pageRank(0.001).vertices
```

▸ (1) Spark Jobs

```
define_rank: org.apache.spark.graphx.VertexRDD[Double] = VertexRDDImpl[1867] at RDD at VertexRDD.scala:57
```

Command took 11.50 seconds -- by jennysavani@trentu.ca at 4/17/2020, 3:57:06 AM on new_project

Joining the airport ranks for airport ID obtained in the previous step with codes of airports, using the join function on the vertex RDD of our graph data.

Cmd 23

```
// joining the ranks of airports with the map of airport id to name
val airport_rank= define_rank.join(vertices_airports)

airport_rank.take(10).foreach(println)
```

(1) Spark Jobs

```
(14696,(0.7779248831113741,SBN))
(12264,(3.611344018379724,IAD))
(15024,(0.7001690762087346,STT))
(12278,(0.7156375777266548,ICT))
(11292,(9.404110300786398,DEN))
(11996,(1.1500578841289073,GSP))
(15074,(0.21126192893616919,SWO))
(10868,(0.4892371246408856,CAE))
(10620,(0.4781091714130192,BIL))
(11146,(0.38684891985499187,CRW))
airport_rank: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, (Double, String))] = MapPartitionsRDD[1894] at join at command-1875235421595212:2
```

Command took 0.73 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:01:27 AM on new_project

Sorting the output by the ranks in descending order, using sortBy().
Next, using important_10_airports on sorted output in the previous step, fetching only top 10 airport codes with highest ranks.

This gives an array as output shown in image below:

DFW being the first element in the array, has the highest rank. Hence, DFW is the busiest/most important airport of all.

Cmd 24

```
// sorting the airports by ranks
val sort_airport_rank = airport_rank.sortBy(_._2._1, false)

sort_airport_rank.take(10)
```

(1) Spark Jobs

```
sort_airport_rank: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, (Double, String))] = MapPartitionsRDD[1899] at sortBy at command-1875235421595
211:2
res48: Array[(org.apache.spark.graphx.VertexId, (Double, String))] = Array((11298,(11.157003294778962,DFW)), (13930,(9.877113185343603,ORD)), (10397,(9.59578
8110997207,ATL)), (11292,(9.404110300786398,DEN)), (13487,(7.414538173031479,MSP)), (11057,(7.3959613106977375,CLT)), (12889,(6.19347341727337,LAS)), (11433,
(6.180326574430039,DTW)), (12266,(6.135883396901176,IAH)), (14107,(5.452483216841405,PHX)))
```

Command took 0.96 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:01:33 AM on new_project

Cmd 25

```
// Getting the airport code of the top 10 airports with highest rank
val important_10_airports =sort_airport_rank.map(_._2._2)
important_10_airports.take(10)
```

(1) Spark Jobs

```
important_10_airports: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1901] at map at command-1875235421595213:2
res50: Array[String] = Array(DFW, ORD, ATL, DEN, MSP, CLT, LAS, DTW, IAH, PHX)
```

Command took 0.44 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:02:45 AM on new_project

10) Find flights with minimum airfare (flight price).

Using the Pregel algorithm for this query. The working of the Pregel algorithm is given in the Approach part of this document.

We can use any airport ID as a source ID. We are using source ID as 11298, because it is the busiest/important airport. Transforming the distance which is edge in our graph data, to price using mapEdges function and creating new graph as new_graph.

Initializing this new_graph using mapVertices function and transforming all the distances in neighboring edges to airfare. Next, calling the predefined pregel method in GraphX. Pregel's vertex program preserves the minimum distance/airfare from the new airfare and actual airfare. Using a triplet, messages are sent off to the neighboring edges using incrementation for each edge between every pair of source vertex and destination vertex. Iteration continues until there are no more messages left to send.

Finally, the reduce operation (a,b) => math.min(a,b) at the end makes sure to select minimum airfare if multiple messages (values) are received for the same vertex.

Cmd 26

```
//Pregel to find flights with cheapest airfare from DFW, 11298 airport
// defining starting vertex ID as 11298 which is for 'DFW' airport
val source_Id: VertexId = 11298
// Transforming edges (distance) into flight fare to compute the cost and defining it in another graph
val new_graph = graph.mapEdges(e => 50.toDouble + e.attr.toDouble/11 )

// initializing new graph. Setting distance of all vertices except source, infinity
val initialize_graph = new_graph.mapVertices((id, _) => if (id == source_Id) 0.0 else Double.PositiveInfinity)

// calling pregel on initialize_graph
val call_pregel = initialize_graph.pregel(Double.PositiveInfinity)(
 // Defining vertex program to preserve the minimum distance of actual distance (airfare cost) and new distance (airfare cost)
 (id, dist, newDist) => math.min(dist, newDist),
  //Triplet sends off message to neighboring vertices and increment occurs
 triplet => {
  if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
   Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
  } else {
   Iterator.empty
  }
 },
  //Merge messages
 // Defining reduce operation to preserve minimum distance (airfare cost) incase multiple messages are received for the same vertex
 (a,b) => math.min(a,b)
)
```

▸ (5) Spark Jobs

Calling our user defined call_pregel function on edges of our data. It returns the origin airport ID, destination airport ID, with airfare between them.

```
Cmd 27

1 //Shows airport_ids (source, destination) of routes with lowest airfare price
2 println(call_pregel.edges.take(5).mkString("\n"))

▸ (1) Spark Jobs
Edge(10135,10397,112.9090909090909)
Edge(10135,13930,109.45454545454545)
Edge(10135,14082,142.54545454545456)
Edge(10135,14761,130.1818181818182)
Edge(10140,10397,165.36363636363637)

Command took 0.33 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:14:29 AM on new_project
```

- Finding 5 source and destination airports codes with minimum airfare.

Output is arranged in ascending order and call_pregel is called upon the edges. Flights between PSG and WRG and vice versa have the lowest airfare.

```
1 //Shows airport codes (source, destination) of routes with lowest airfare price in sorted ascending order
2 call_pregel.edges.map{ case ( Edge(origin_id, dest_id, airfare))=> ( (airport_map(origin_id), airport_map(dest_id), airfare)) }.takeOrdered(5)
  (Ordering.by(_._3)).mkString("\n")

▸ (1) Spark Jobs
res68: String =
(PSG,WRG,52.81818181818182)
(WRG,PSG,52.81818181818182)
(PAH,CGI,54.09090909090909)
(CGI,PAH,54.09090909090909)
(STS,SFO,56.0)

Command took 0.48 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:14:29 AM on new_project
```

Calling our user defined call_pregel function on vertices of our data. It returns the airport ID and airfare with respect to our source_ID 11298 which is DFW (defined in very first step of Pregel implementation).

```
Cmd 29

1 // Airports id with lowest airfare price
2 println(call_pregel.vertices.take(5).mkString("\n"))

▸ (1) Spark Jobs
(10208,179.45454545454544)
(12264,156.54545454545456)
(14828,379.3636363636364)
(14698,225.1818181818182)
(12278,79.81818181818181)

Command took 0.29 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:14:29 AM on new_project
```

- Finding 10 Airports with minimum airfare from DFW.

Calling call_pregel function on vertices (airports) this time. Output is sorted in ascending order.

First line of output (DFW, 0) has airfare 0 because it is the source airport. // DFW = 11298

```
Cmd 30

1 // Airports code with lowest airfare price; DFW = 0.0 because that is our start vertex.
2 call_pregel.vertices.collect.map(x => (airport_map(x._1), x._2)).sortWith(_._2 < _._2).take(10).mkString("\n")

▸ (1) Spark Jobs
res70: String =
(DFW,0.0)
(ACT,58.09090909090909)
(TYR,59.27272727272727)
(SPS,60.27272727272727)
(GRK,62.18181818181818)
(GGG,62.72727272727273)
(LAW,62.72727272727273)
(ABI,64.36363636363636)
(CLL,64.9090909090909)
(OKC,65.9090909090909)

Command took 0.53 seconds -- by jennysavani@trentu.ca at 4/17/2020, 4:14:29 AM on new_project
```

Using GraphFrames to work with data:

(This part has been performed by Shreya Walia.)

About:
GraphFrames ia a package from Apache Spark that lets us make data frame based graphs. It supports working with writing and reading data from many formats like json, csv. Moreover performing queries is easier as it allows queries in the familiar API of spark SQL and spark data frames.

For creating a GraphFrame we need to use two DataFrames, vertex and edge .

- Vertex DataFrame: A vertex DataFrame is used to specify unique IDs for each vertex in the graph and should contain a special column named id.
- Edge DataFrame: An edge DataFrame is used to specify edges between the vertices and should contain two special columns: src (source vertex ID of edge) and dst (destination vertex ID of edge).

In addition to the mentioned columns these Data Frames can have other columns also for representing the attributes of the vertex and edge.

As GraphX works with RDDs and thus it limits the functionality for programmers from other backgrounds while GraphFrames is built on DataFrames which makes it easy to use the functionality of conventional data frames like performing queries such as filtering, grouping, ordering, sorting etc. Also it gives the opportunity to use Motif finding, a useful feature which is not supported in GraphX. We can perform and run different pre-built algorithms like PageRank, Breadth-First, Shortest-Path etc.

Steps/Implementation:

- Loading GraphFrame library on our Databricks cluster:
  We need to install the GraphFrame library onto our cluster as it does not pre-exist on our cluster. For doing so we need to open our cluster and go to the library section. Here click import library and select Maven and add the name of the library (GraphFrames) and choose the suitable version of the library as per the scala and spark as per the cluster being run.

- Importing packages:
  Import all the required packages to run the project:

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
import org.graphframes._
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
```

- View of all columns of complete dataset:

```
+-----------------------------------------------------------------------------------------------------------------+
|value                                                                                                            |
+-----------------------------------------------------------------------------------------------------------------+
|222,1,2,9E,N8688C,3280,11953,GNV,10397,ATL,GA,600,601,1,723,722,-1,83,1,300,"Atlanta, GA"                        |
|223,1,2,9E,N348PQ,3281,13487,MSP,11193,CVG,KY,1404,1359,-5,1709,1633,-36,125,1,596,"Cincinnati, OH"             |
|224,1,2,9E,N8896A,3282,11433,DTW,11193,CVG,KY,1220,1215,-5,1345,1329,-16,85,1,229,"Cincinnati, OH"              |
|225,1,2,9E,N8886A,3283,15249,TLH,10397,ATL,GA,1527,1521,-6,1639,1625,-14,72,1,223,"Atlanta, GA"                 |
|226,1,2,9E,N8974C,3284,10397,ATL,11778,FSM,AR,1902,1847,-15,2005,1940,-25,123,1,579,"Fort Smith, AR"            |
|227,1,2,9E,N927EV,3285,11267,DAY,13487,MSP,MN,900,853,-7,1012,953,-19,132,1,574,"Minneapolis, MN"               |
|228,1,2,9E,N915XJ,3286,12448,JAN,10397,ATL,GA,1558,1553,-5,1823,1832,9,85,1,341,"Atlanta, GA"                   |
|229,1,2,9E,N295PQ,3287,12953,LGA,11193,CVG,KY,1555,1551,-4,1821,1824,3,146,1,585,"Cincinnati, OH"               |
|230,1,2,9E,N337PQ,3288,12451,JAX,12953,LGA,NY,1045,1037,-8,1301,1239,-22,136,1,833,"New York, NY"               |
|231,1,2,9E,N311PQ,3289,10397,ATL,10685,BMI,IL,1245,1245,0,1332,1318,-14,107,1,533,"Bloomington/Normal, IL"     |
+-----------------------------------------------------------------------------------------------------------------+
```

```
df: org.apache.spark.sql.DataFrame
    trip_id: integer
    day_of_month: string
    day_of_week: string
    carrier_code: string
    tail_num: string
    flight_num: string
    origin_id: long
    origin: string
    dest_id: long
    dest: string
    dest_state: string
    scheduled_dept_time: string
    actual_dept_ime: string
    dep_delay_mins: string
    scheduled_arr_time: string
    actual_arr_time: string
    arr_delay_mins: string
    elapsed_time_mins: string
    total_flights: long
    distance: integer
    city: string
```

- Preprocessing the data to make new Dataframes:
  We create two new data frames city_data and trip_data that contain the columns we need to work on

```
4
5 val city_data = df.select("origin").distinct()
6 val trip_data = df.select("origin", "dest", "dep_delay_mins", "distance", "dest_state", "scheduled_dept_time")
7
```

```
df: org.apache.spark.sql.DataFrame = [trip_id: integer, day_of_month: string ... 19 more fields]
city_data: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
    origin: string
trip_data: org.apache.spark.sql.DataFrame
    origin: string
    dest: string
    dep_delay_mins: string
    distance: integer
    dest_state: string
    scheduled_dept_time: string
```

We make two new data frames after changing the column names origin in city_data to “id” for vertex DataFrame and origin to “src” and dest to “dst” in trip_data for edges dataframe.

```
4  val trip_vertices = city_data
5    .withColumnRenamed("origin", "id")
6
7  val trip_edges = trip_data
8    .withColumnRenamed("origin", "src")
9    .withColumnRenamed("dest", "dst")
```

```
▾ trip_vertices: org.apache.spark.sql.DataFrame
    id: string
▾ trip_edges: org.apache.spark.sql.DataFrame
    src: string
    dst: string
    dep_delay_mins: string
    distance: integer
    dest_state: string
    scheduled_dept_time: string
```

- View of trip_vertices and trip_edges dataframe:

```
▸ (3) Spark Jobs
+---+
| id|
+---+
|BGM|
|PSE|
|INL|
|MSY|
|PPG|
+---+
only showing top 5 rows

+---+---+--------------+--------+----------+--------------------+
|src|dst|dep_delay_mins|distance|dest_state|scheduled_dept_time|
+---+---+--------------+--------+----------+--------------------+
|GNV|ATL|             1|     300|        GA|                 600|
|MSP|CVG|            -5|     596|        KY|                1404|
|DTW|CVG|            -5|     229|        KY|                1220|
|TLH|ATL|            -6|     223|        GA|                1527|
|ATL|FSM|           -15|     579|        AR|                1902|
+---+---+--------------+--------+----------+--------------------+
only showing top 5 rows
```

- Creating the GraphFrame

  Below, we create a GraphFrame by supplying a vertex DataFrame "trip_vertices" and an edge DataFrame "trip_edges" to create GraphFrame named "trip_graph".

```
1 val trip_graph = GraphFrame(trip_vertices, trip_edges)
2
3 trip_edges.cache()
4 trip_vertices.cache()
```

```
trip_graph: org.graphframes.GraphFrame = GraphFrame(v:[id: string], e:[src: string, dst: string ... 4 more fields])
res3: trip_vertices.type = [id: string]
```

```
Command took 0.65 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:35:31 PM on new2
```

Performing Queries:

1. Finding the total number of airports using vertices of GraphFrame:

```
1 // Total number of airports:
2 trip_graph.vertices.count
3
```

```
▸ (1) Spark Jobs
res5: Long = 346
```

```
Command took 4.42 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:35:31 PM on new2
```

2. Finding the total number of flights using edges of GraphFrame:

```
1 // Total number of flights:
2 trip_graph.edges.count
```

```
▸ (1) Spark Jobs
res6: Long = 583985
```

```
Command took 3.06 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:35:31 PM on new2
```

3. Finding the flights that cover more than 800 miles of distances by filtering on the distance column of edges of GraphFrame.

```
1  // flights covering more than 800 miles distance
2  trip_graph.edges.filter("distance > 800").show
3
```

▸ (1) Spark Jobs

```
+---+---+--------------+--------+----------+-------------------+
|src|dst|dep_delay_mins|distance|dest_state|scheduled_dept_time|
+---+---+--------------+--------+----------+-------------------+
|JAX|LGA|            -8|     833|        NY|               1045|
|BOS|BNA|            -6|     942|        TN|               1745|
|ATL|LNK|            -4|     841|        NE|               2115|
|LGA|JAX|            -3|     833|        FL|               1630|
|BNA|BOS|           -11|     942|        MA|               2045|
|SAT|MSP|            -1|    1097|        MN|                745|
|JAX|LGA|            -6|     833|        NY|               1500|
|MSP|GTF|            -9|     887|        MT|               1945|
|MSP|RDU|            -4|     980|        NC|               1120|
|SAT|DTW|           -10|    1214|        MI|               1500|
|BDL|MSP|           -17|    1050|        MN|               1630|
|DFW|DTW|            -1|     986|        MI|               1125|
|MSP|RIC|            -5|     970|        VA|               1749|
|IAD|MSP|           -14|     908|        MN|               1713|
|MSP|IAD|            23|     908|        VA|               1245|
|RDU|MSP|            -4|     980|        MN|                600|
|IAD|MSP|            15|     908|        MN|               1435|
|MSP|IAD|            58|     908|        VA|               1000|
```

Command took 1.02 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:35:31 PM on new2

4. Finding all the flights having destination state as California “CA” by filtering on edges of the GraphFrame.

```
1 trip_graph.edges.filter("dest_state ='CA'").show
```

▸ (1) Spark Jobs

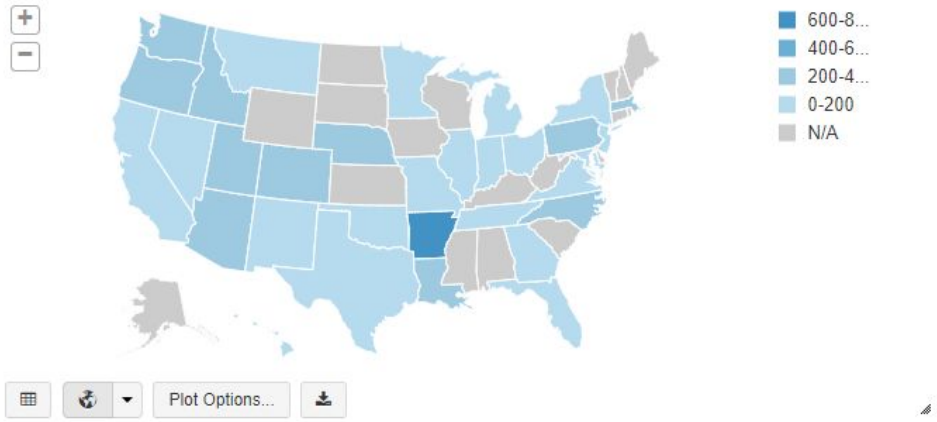
src	dst	dep_delay_mins	distance	dest_state	scheduled_dept_time
BNA	LAX	73	1797	CA	1901
DFW	SJC	-2	1438	CA	1840
MIA	SFO	-3	2585	CA	730
PHX	LAX	45	370	CA	1506
CLT	SFO	1	2296	CA	2020
JFK	LAX	-1	2475	CA	800
JFK	LAX	-7	2475	CA	1230
OGG	LAX	29	2486	CA	2302
JFK	LAX	-4	2475	CA	700
KOA	LAX	-11	2504	CA	1350
DFW	LAX	27	1235	CA	2215
JFK	LAX	-4	2475	CA	1500
DFW	SAN	-2	1171	CA	933
JFK	LAX	23	2475	CA	1700
DFW	SFO	-4	1464	CA	1258
HNL	LAX	-9	2556	CA	2321
MSY	LAX	2	1670	CA	700
EGE	LAX	344	748	CA	1444

Command took 0.50 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:40:56 PM on new2

5. Finding all the flights originating from “SFO” with delay more than 100 mins and visualizing them on the US map as per the average delay per state of the flights at the airports in that state.

```
1 display(trip_graph.edges.filter("src = 'SFO' and dep_delay_mins > 100"))
2
```

▸ (2) Spark Jobs



Command took 1.67 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:35:31 PM on new2

6. Finding the longest route between source and destination airports based on the distance column and arranging it in descending order.

```
1  //longest routes
2  trip_graph.edges
3    .groupBy("src", "dst")
4    .max("distance")
5    .sort(desc("max(distance)")).show
```

▸ (1) Spark Jobs

```
+---+---+-------------+
|src|dst|max(distance)|
+---+---+-------------+
|HNL|JFK|         4983|
|JFK|HNL|         4983|
|HNL|EWR|         4962|
|EWR|HNL|         4962|
|HNL|IAD|         4817|
|IAD|HNL|         4817|
|ATL|HNL|         4502|
|HNL|ATL|         4502|
|HNL|ORD|         4243|
|ORD|HNL|         4243|
|ORD|OGG|         4184|
|OGG|ORD|         4184|
|HNL|MSP|         3972|
|MSP|HNL|         3972|
|HNL|IAH|         3904|
|IAH|HNL|         3904|
|GUM|HNL|         3801|
|HNL|GUM|         3801|
```

7. Finding the flights from DTW airport covering more than 1000 miles of distance arranged as per the departure delay.

```
// distance is greater than 1000 miles for DTW airport arranged by delay
trip_graph.edges.filter("distance > 1000 and src == 'DTW'")
.orderBy(desc("dep_delay_mins")).show(5)

```

▸ (1) Spark Jobs

```
+---+---+--------------+--------+----------+-------------------+
|src|dst|dep_delay_mins|distance|dest_state|scheduled_dept_time|
+---+---+--------------+--------+----------+-------------------+
|DTW|RSW|           992|    1084|        FL|               2015|
|DTW|FLL|           990|    1127|        FL|                640|
|DTW|LAX|            98|    1979|        CA|               1220|
|DTW|FLL|            97|    1127|        FL|               1210|
|DTW|IAH|            97|    1075|        TX|                833|
+---+---+--------------+--------+----------+-------------------+
only showing top 5 rows
```

Command took 0.81 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:47:33 PM on new2

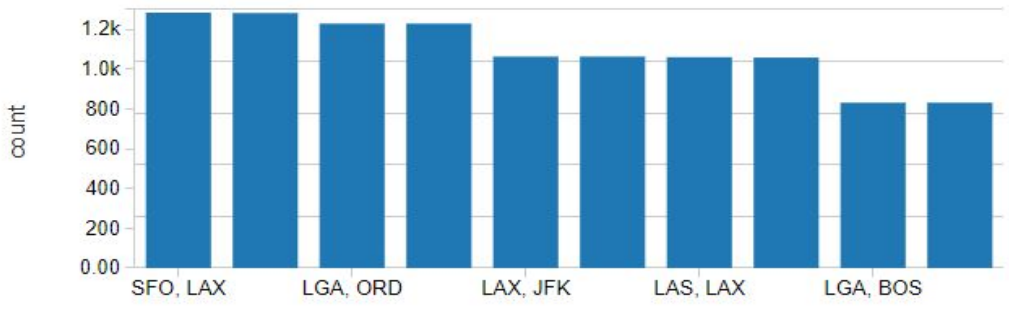
8. Finding the source and destination airports having maximum number of flights between them (busiest route) and viewing them using bar plot.

```
//busiest flight routes as per trip count
val most_trips = trip_graph
  .edges
  .groupBy("src", "dst")
  .count()
  .orderBy(desc("count"))
  .limit(10)

display(most_trips)
```

▸ (1) Spark Jobs

▸ most_trips: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [src: string, dst: string ... 1 more fields]

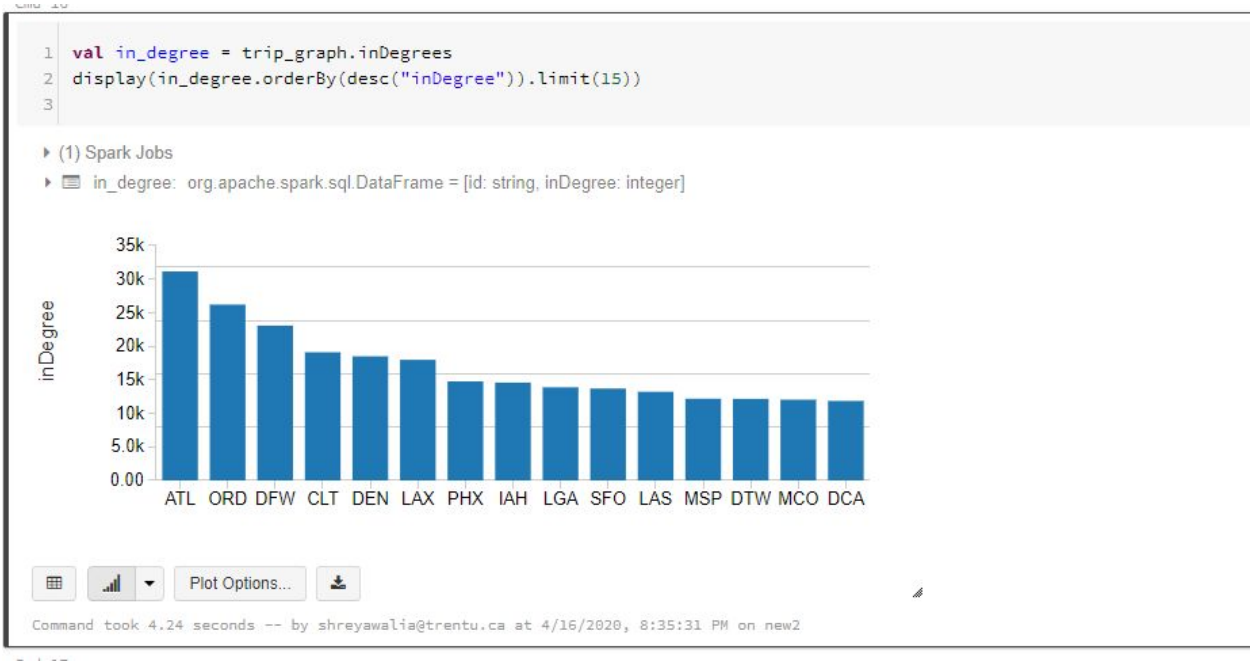


Plot Options...

Calculating In and Out Degrees:

Degrees for a vertex can be defined as the number of edges that touch the vertex. In GraphFrame, we can find the inDegree, outDegree and ration between them for a vertex. Degree of a vertex provides us information about the incoming and outgoing flights for an airport.

1. In degree: It is the number of edges that enter a vertex and thus represents the number of incoming flights to an airport.
   We can find the airport having most incoming flights. Here it is ATL.

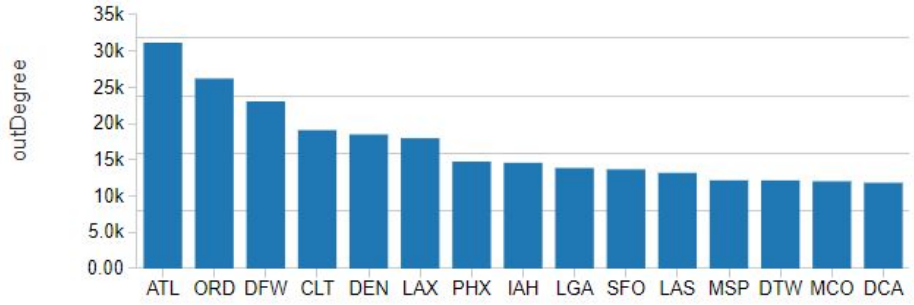


2. Out degree: It is the number of edges that exit a vertex and thus represents the number of outgoing flights to an airport.
   We can find the airport having most outgoing flights. Here it is ATL.

```
val out_degree = trip_graph.outDegrees
display(out_degree.orderBy(desc("outDegree")).limit(15))
```

(1) Spark Jobs

out_degree: org.apache.spark.sql.DataFrame = [id: string, outDegree: integer]



Plot Options...

Command took 3.42 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 8:35:31 PM on new2

3. Ratio between Degrees: We can calculate the ratio between the inDegree and outDegree. It can be used to find the airports which have high incoming flights but low outgoing flights or vice versa which can be interpreted through the abnormality in the ratio.

```
val degree_ratio = in_degree.join(out_degree, in_degree.col("id") === out_degree.col("id"))
  .drop(out_degree.col("id"))
  .selectExpr("id", "double(inDegree)/double(outDegree) as degree_ratio")

degree_ratio.cache()

display(degree_ratio.orderBy(desc("degree_ratio")).limit(10))
```

(1) Spark Jobs

degree_ratio: org.apache.spark.sql.DataFrame
id: string
degree_ratio: double

id	degree_ratio
LYH	1.0208333333333333
BFF	1.0188679245283019
LAR	1.0188679245283019
APN	1.0188679245283019
EAR	1.0188679245283019
LBF	1.0188679245283019
LBL	1.0188679245283019
UIN	1.0172413793103448
CCC	1.0172413793103448

Motif Find for Graph Pattern Queries:

Motif finding refers to searching for structural patterns in a graph. Graph frames motif finding uses a simple domain specific language(DSL) for expressing structural queries. When we specify (a)-[e] ->(b) , it expresses an edge from vertex a to vertex b. Patterns can be joined by semicolons.

1. Finding airports that have flights leaving them to a destination airport but not coming back into the same airport from there.

```
1 val res = trip_graph
2  .find("(a)-[]->(b); !(b)-[]->(a)")
3
4 display(res)
```

▸ (5) Spark Jobs
▸ res: org.apache.spark.sql.DataFrame = [a: struct, b: struct]

a	b
{"id":"DFW"}	{"id":"PIB"}
{"id":"SCC"}	{"id":"BRW"}
{"id":"ORD"}	{"id":"BIS"}
{"id":"BTR"}	{"id":"DCA"}
{"id":"TWF"}	{"id":"LAX"}
{"id":"TWF"}	{"id":"SFO"}
{"id":"BRW"}	{"id":"FAI"}

2. Complex motif find queries can be expressed by applying filters to result dataframes.
   Here we are finding flights that have connection flights between LGA and BOS. We can see that there are some flights going through MSY between LGA and BOS.

```
//using motif with dataframe operation
val motif =trip_graph.find("(a)-[ab]->(b); (b)-[bc]->(c)")
.filter("a.id = 'LGA'")
.filter("c.id = 'BOS'").limit(5)

display(motif)
```

▸ (2) Spark Jobs

▸ motif: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [a: struct, ab: struct ... 3 more fields]

a	ab	b	bc
{"id":"LGA"}	{"src":"LGA","dst":"MSY","dep_delay_mins":"21","distance":1183,"dest_state":"LA","scheduled_dept_time":"1825"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368
{"id":"LGA"}	{"src":"LGA","dst":"MSY","dep_delay_mins":"-1","distance":1183,"dest_state":"LA","scheduled_dept_time":"1605"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368
{"id":"LGA"}	{"src":"LGA","dst":"MSY","dep_delay_mins":"-4","distance":1183,"dest_state":"LA","scheduled_dept_time":"950"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368
{"id":"LGA"}	{"src":"LGA","dst":"MSY","dep_delay_mins":"-1","distance":1183,"dest_state":"LA","scheduled_dept_time":"1825"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368
{"id":"LGA"}	{"src":"LGA","dst":"MSY","dep_delay_mins":"-8","distance":1183,"dest_state":"LA","scheduled_dept_time":"1355"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368

Command took 6.59 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 9:12:37 PM on new2


```
//using motif with dataframe operation
val motif =trip_graph.find("(a)-[ab]->(b); (b)-[bc]->(c)")
.filter("a.id = 'LGA'")
.filter("c.id = 'BOS'").limit(5)

display(motif)
```

▸ (2) Spark Jobs

▸ motif: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [a: struct, ab: struct ... 3 more fields]

	b	bc	c
"distance":1183,"dest_state":"LA","scheduled_dept_time":"1825"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368,"dest_state":"MA","scheduled_dept_time":"1437"}	{"id":"BOS"}
distance":1183,"dest_state":"LA","scheduled_dept_time":"1605"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368,"dest_state":"MA","scheduled_dept_time":"1437"}	{"id":"BOS"}
distance":1183,"dest_state":"LA","scheduled_dept_time":"950"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368,"dest_state":"MA","scheduled_dept_time":"1437"}	{"id":"BOS"}
distance":1183,"dest_state":"LA","scheduled_dept_time":"1825"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368,"dest_state":"MA","scheduled_dept_time":"1437"}	{"id":"BOS"}
distance":1183,"dest_state":"LA","scheduled_dept_time":"1355"}	{"id":"MSY"}	{"src":"MSY","dst":"BOS","dep_delay_mins":"-3","distance":1368,"dest_state":"MA","scheduled_dept_time":"1437"}	{"id":"BOS"}

Command took 6.59 seconds -- by shreyawalia@trentu.ca at 4/16/2020, 9:12:37 PM on new2

Algorithms:

Shortest Path Graph Algorithm:

Shortest path algorithm is used for finding the airports having the shortest path as per the number of connecting flights from each vertex to the given landmark vertices.

1. Finding the shortest path from each airport to ORD.
   We can see that there are no direct flights from airports BGM, CLL, BLI, PSE and INL to ORD as the distances greater than 1 whereas there are direct flights from BNA, CGI and CLT as distance is equal to 1.

```
1 //shortest-path
2 val shortest_path = trip_graph.shortestPaths.landmarks(Seq("ORD")).run()
3 display(shortest_path)
```

▸ (8) Spark Jobs
▸ shortest_path: org.apache.spark.sql.DataFrame = [id: string, distances: map]

id	distances
BNA	▸ {"ORD":1}
BGM	▸ {"ORD":2}
CGI	▸ {"ORD":1}
CLL	▸ {"ORD":2}
BLI	▸ {"ORD":2}
INL	▸ {"ORD":2}
PSE	▸ {"ORD":2}
CLT	▸ {"ORD":1}
DSC	▸ {"ORD":2}

Breadth First Search Graph Algorithm

Breadth-first search (BFS) algorithm is used to traverse the graph to find the shortest path based on hop count between the desired beginning and end vertices. It gives us the flight between these vertices as per the maxPathLength specified.

1. Finding flights from JFK airport to ORD airport that are direct using BFS.

```
1 //breadth-first algorithm
2 val breadth_first = trip_graph.bfs
3   .fromExpr((col("id") === "JFK"))
4   .toExpr(col("id") === "ORD")
5   .maxPathLength(1).run()
6
7 display(breadth_first)
```

▸ (18) Spark Jobs

▸ breadth_first: org.apache.spark.sql.DataFrame = [from: struct, e0: struct ... 1 more fields]

from	e0	to
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"6","distance":740,"dest_state":"IL","scheduled_dept_time":"1542"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"-2","distance":740,"dest_state":"IL","scheduled_dept_time":"758"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"-6","distance":740,"dest_state":"IL","scheduled_dept_time":"759"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"-1","distance":740,"dest_state":"IL","scheduled_dept_time":"1300"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"-4","distance":740,"dest_state":"IL","scheduled_dept_time":"1125"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"-1","distance":740,"dest_state":"IL","scheduled_dept_time":"1932"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"42","distance":740,"dest_state":"IL","scheduled_dept_time":"1708"}	{"id":"ORD"}
{"id":"JFK"}	{"src":"JFK","dst":"ORD","dep_delay_mins":"-11","distance":740,"dest_state":"IL","scheduled_dept_time":"1125"}	{"id":"ORD"}

PageRank algorithm:

PageRank algorithm is used to find the importance of a vertex based on the number of edges the vertex has with other vertices of the graph.
Here we can use PageRank algorithm to find which airports are the most important, by measuring the connection of the airport with other airports that are also well connected.

1. Running PageRank we find that ATL is the most important airport.

```
// use pageRank
val page_ranks = trip_graph.pageRank.resetProbability(0.15).maxIter(10).run()

page_ranks.vertices.orderBy($"pagerank".desc).show()

```

▸ (4) Spark Jobs

```
+---+------------------+
| id|          pagerank|
+---+------------------+
|ATL|16.441502602773404|
|ORD|14.776759457990115|
|DFW|14.402832851330512|
|DEN|10.843523223072308|
|CLT| 9.974519623207293|
|LAX|  8.53507675993154|
|MSP|  7.71583703580518|
|IAH| 7.565543317845415|
|PHX|  7.32301238287094|
|DTW|7.0752380771452215|
|SFO| 6.855419553004998|
|LAS| 6.420161348050773|
|LGA|  6.34000592772682|
|SLC| 5.889808351749144|
|MCO| 5.719265194157857|
|SEA| 5.698224174371949|
|DCA| 5.462485110589576|
|BOS|  4.98708486699475|
```

Evaluation

About working with GraphX:

GraphX takes very less time for computation, compared to the size of data. Computing large data in the graph form is really simple and easy. Predefined functions for graph preprocessing algorithms like PageRank and Pregel yields the results very fastly.

GraphX does not provide any built-in functionality for visualization of data. Although, visualization can be done using third party libraries like GraphStream, BreezeViz, and alike. However, these libraries come with certain limitations and

do not leverage any results. GraphX does not support graph queries, which makes it tedious to write the program all the time, even for simple queries

About working with GraphFrames:

Working with GraphFrames is easy for people having experience of programming with DataFrames. As the GraphFrame is built on DataFrames running queries is comparatively easier than GraphX.

As per the visualization is concerned, GraphFrame has an advantage in that context as it returns results majorly in the form of DataFrames which helps in making plots and graphs from the results.

GraphFrame provides additional functionality of Motif Finding which gives the user a handy tool to form structural queries and get interesting insights. GraphFrames also has a number of in-built functions for running algorithms which makes it easy to work with.

Comparison of 2 approaches:

GraphFrames	GraphX
Supports Visualization of data and making plots is easy as data is returned in formed of data frames.	GraphX do not provide any built-in functionality for data visualization, since the main focus is on data processing. We need third party libraries like GraphStream, BreezeViz, et cetera to create plots, however with certain limitations.
It supports several graph processing algorithms and graph queries. This collectively helps in processing hidden data and patterns inside the graph structure.	It only supports graph processing algorithms and not the graph queries.
Supports Motif finding using DSL which helps in searching for structural patterns in a graph	GraphX do not support Motif finding.
For edges,the edges in graph frames correspond to the total number of rows in the data i.e., for each flight one edge is created.	As it evaluates data in the form of directed graph, edges are formed between all the pairs of vertices and edges can be multiple and parallel as well.
Takes more time than GraphX to execute queries.	Takes less time to execute as compared to GraphFrames.

Contribution of each participant:

Graphx: Entire implementation done by Jenny Savani.
GraphFrames: Entire implementation done by Shreya Walia.