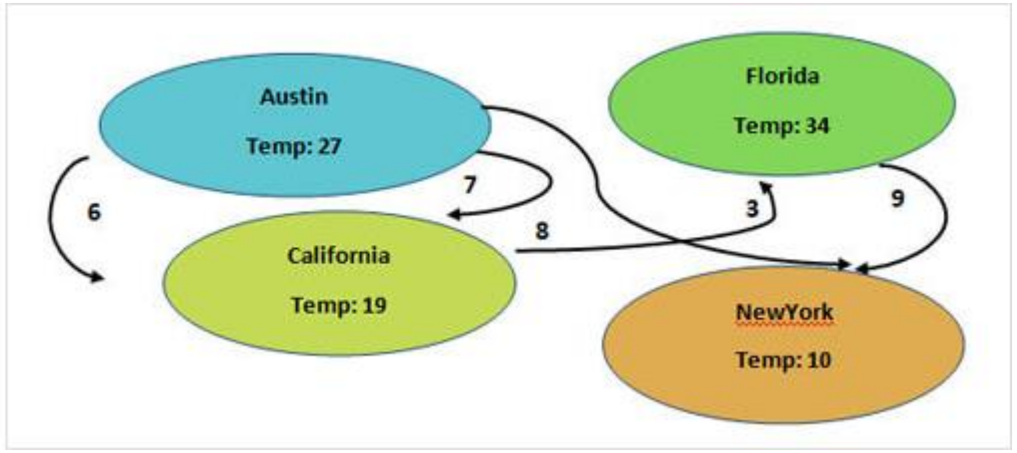
SPARK Graphx

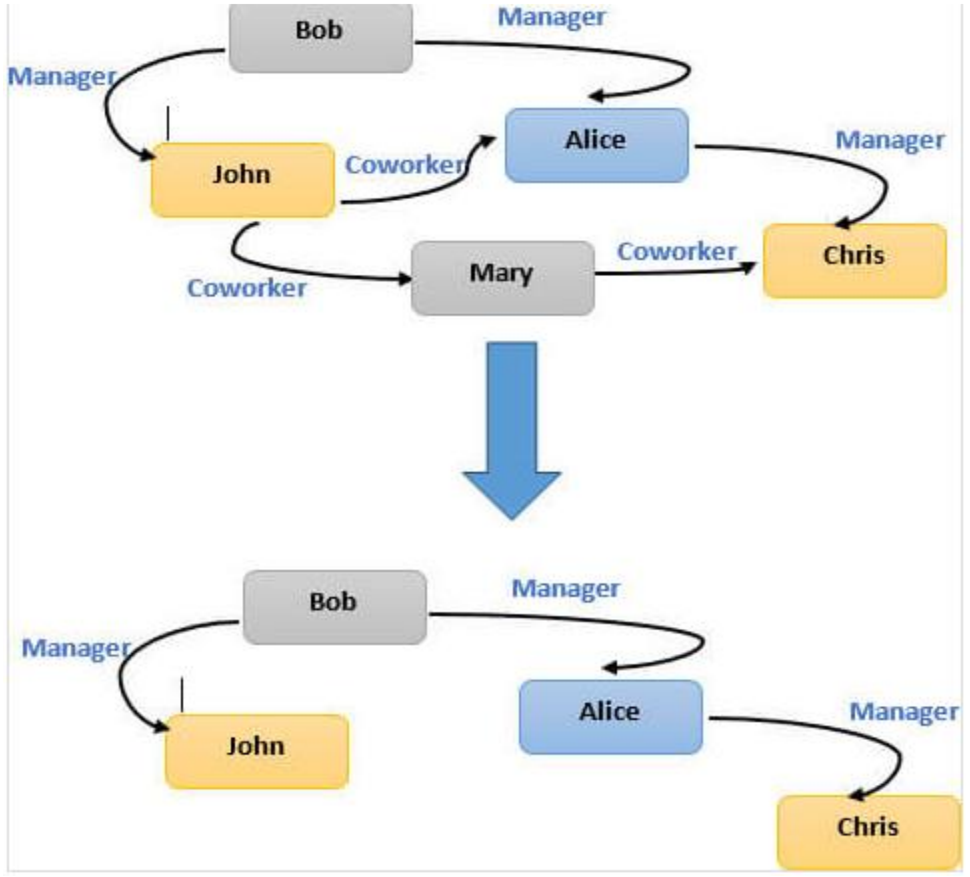
- Relationships between data can be seen everywhere in the real world, from social networks to traffic routes, from DNA structure to commercial system, in machine learning algorithms, to predict customer purchase trends and so on. Graph analytics can make sense of these connections to derive some incredible outputs and to provide more insight that cannot see with naked eyes. It widely used for recommendation engines, fraud detection, route optimization, social network analysis, page ranking, and many more. In this blog, we are going to discuss in detail one of the most popular graph analytics computing framework- Spark GraphX.
- **Different graph computing engines**
- There are many graph analytics frameworks in the market - Apache Girafe, Pregel, GraphLab, Spark GraphX.These graph engines provide an optimized abstraction over various graph algorithms, which can be run in a distributed environment thus processing large graphs in parallel and fault-tolerant manner.

- **Spark Graphx**
- Spark GraphX is a distributed graph computing engine that extends Spark RDD. How it is different from other graph-processing framework is that it can perform both graph analytics and ETL and can do graph analysis on data that is not in graph form. Spark Graphx provides an implementation of various graph algorithms such as PageRank, Connected Components, and Triangle Counting.
- GraphX supports property multigraph, which is a directed graph with multiple parallel edges to represent more than one relationship between the same source node & destination node. For example, a person can stay & work from the same address, which means the same address can use as a home address as well as an office address for that, the person thus representing multiple relationships.

- Property graphs are immutable just like RDD, which means once we create the graph it cannot be modified but, we can transform it to create new graphs.
- Property graphs distributed on multiple machines (executors) for parallel processing.
- Property graphs are fault-tolerant, which means it can recreate in case of any failures.
- In Spark GraphX, nodes and relationships are represented as dataframes or RDDS. Node dataframe must have a unique id along with other properties and the relationship dataframe must have a source and destination id along with other attributes.
- For example, we have a graph where nodes represent the cities name of the United States along with the average temperature, and edges represent much flight between these cities.



- **Subgraphs**
- Using graph operators, we can also create subgraphs out of a graph using some predicates on vertexes or edges. Let us say we have a graph that represents the people working in their organization and their relationship based on their designation. Using the subgraph operator, we will create a subgraph that will only have a manager as the relationship.



- GraphX Algorithms
- PageRank
- GraphX provides an implementation of various graph algorithms, such as PageRank. PageRank measures the importance of each vertex in a graph.

- For example, in-flight example, if a city has a higher number of incoming & outgoing flights then, that city might have higher importance than other cities. Graphx provides dynamic, as well as the static implementation of PageRank. In a static implementation, the algorithm will run a fixed number of times, whereas in dynamic implementation; Pagerank will run until the ranks converge.

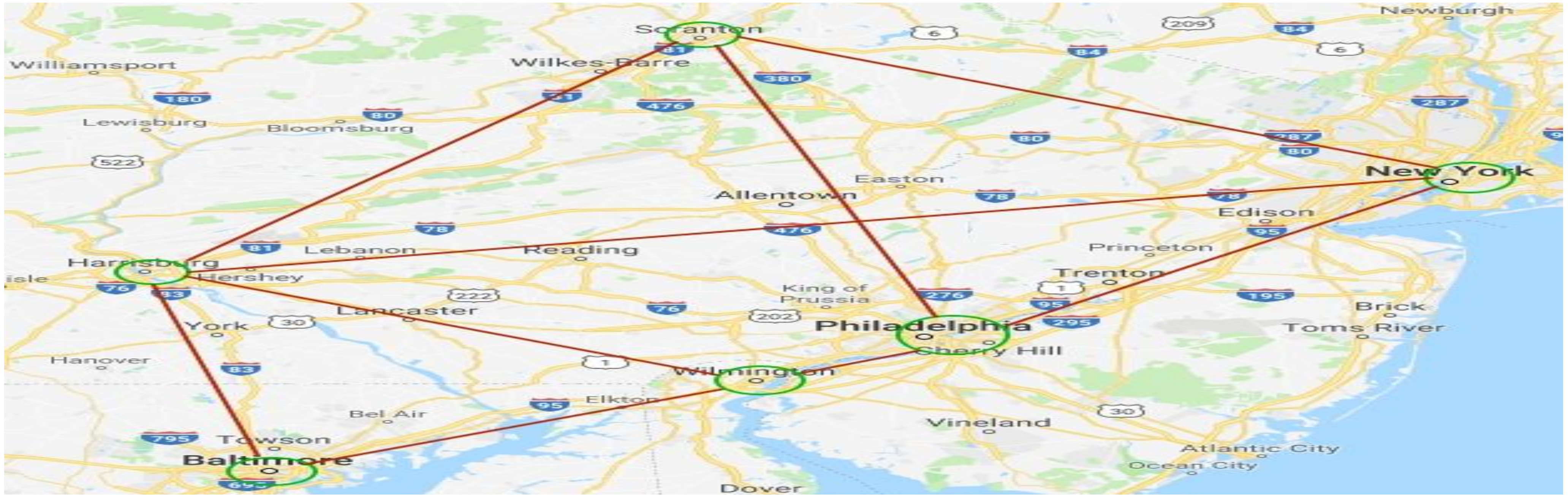
- val ranks = graph.pageRank(0.0001).vertices

- Connected Components
- A Connected Component is a connected subgraph where two vertices connected by an edge, and there are no additional vertices in the main graph. This algorithm is also used to label each connected component of the graph with the ID of its lowest-numbered vertex.

- valconnectedComponents= graph.connectedComponents().vertices
- Triangle Counting
- Triangle counting is used to detect community in a graph by determining the number of triangles passing through each vertex. This algorithm heavily used, in social network analysis spam detection and link recommendations.

- val triCounts = graph.triangleCount().vertices

- Example of the graph: the cities are the vertices and the distances between them are the edges. You can see the Google Maps illustration of this structure in the figure below.



Creating the property graph

- To create property graph we should firstly create an array of vertices and an array of edges. For vertices array, attributes of the vertices mean the city name and population, respectively

```
val verArray = Array(
(1L, ("Philadelphia", 1580863)),
(2L, ("Baltimore", 620961)),
(3L, ("Harrisburg", 49528)),
(4L, ("Wilmington", 70851)),
(5L, ("New York", 8175133)),
(6L, ("Scranton", 76089)))
```

```
verArray: Array[(Long, (String, Int))] = Array((1,(Philadelphia,1580863)), (2,(Baltimore,620961)), (3,(Harrisburg,49528)), (4,(Wilmington,70851)), (5,(New York,8175133)), (6,(Scranton,76089)))
```

To create edges array

- The first and the second arguments indicate the source and the destination vertices identifiers and the third argument means the edge property which, in our case, is the distance between corresponding cities in kilometers.

```
val edgeArray = Array(
Edge(2L, 3L, 113),
Edge(2L, 4L, 106),
Edge(3L, 4L, 128),
Edge(3L, 5L, 248),
Edge(3L, 6L, 162),
Edge(4L, 1L, 39),
Edge(1L, 6L, 168),
Edge(1L, 5L, 130),
Edge(5L, 6L, 159))
```

```
1.edgeArray: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(2,3,113), Edge(2,4,106), Edge(3,4,128), Edge(3,5,248), Edge(3,6,162), Edge(4,1,39), Edge(1,6,168), Edge(1,5,130), Edge(5,6,159))
```

- we will create RDDs from the vertices and edges arrays

```
val verRDD = sc.parallelize(verArray)
val edgeRDD = sc.parallelize(edgeArray)
```

- We are ready to build a property graph. The basic property graph constructor takes an RDD of vertices and an RDD of edges and builds a graph.

```
val graph = Graph(verRDD, edgeRDD)
```

- Now we have our property graph, and it is time to consider basic operations which can be performed with graphs such as filtration by vertices, filtration by edges, operations with triplets and aggregation.

Filtration by vertices

- To illustrate the filtration by vertices let’s find the cities with population more than 50000.

```scala
graph.vertices.filter {
   case (id, (city, population)) => population > 50000
}.collect.foreach {
    case (id, (city, population)) =>
    println(s"The population of $city is $population")
}
```

```
The population of Scranton is 76089
The population of Wilmington is 70851
The population of Philadelphia is 1580863
The population of New York is 8175133
The population of Baltimore is 620961
```

- One of the core functionalities of GraphX is exposed through the triplets RDD. There is one triplet for each edge which contains information about both the vertices and the edge information. we will find the distances between the connected cities

```scala
for (triplet <- graph.triplets.collect) {
  println(s"""The distance between ${triplet.srcAttr._1} and
${triplet.dstAttr._1} is ${triplet.attr} kilometers""")
}
```

```
The distance between Baltimore and Harrisburg is 113 kilometers
The distance between Baltimore and Wilmington is 106 kilometers
The distance between Harrisburg and Wilmington is 128 kilometers
The distance between Harrisburg and New York is 248 kilometers
The distance between Harrisburg and Scranton is 162 kilometers
The distance between Wilmington and Philadelphia is 39 kilometers
The distance between Philadelphia and New York is 130 kilometers
The distance between Philadelphia and Scranton is 168 kilometers
The distance between New York and Scranton is 159 kilometers
```

Filtration by edges

- Now, let’s consider another type of filtration, namely filtration by edges. For this purpose, we want to find the cities, the distance between which is less than 150 kilometers.

```
graph.edges.filter {
  case Edge(city1, city2, distance) => distance < 150
}.collect.foreach {
  case Edge(city1, city2, distance) =>
  println(s"The distance between $city1 and $city2 is $distance")
}
```

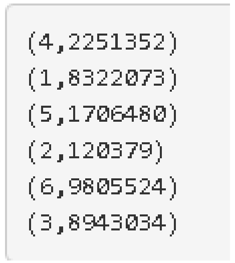
```
The distance between 2 and 3 is 113
The distance between 2 and 4 is 106
The distance between 3 and 4 is 128
The distance between 4 and 1 is 39
The distance between 1 and 5 is 130
```

Aggregation

Another interesting task which can be considered here is aggregation. We will find total population of the neighboring cities. But before we start, we should change our graph a little. The reason for this is that GraphX deals only with directed graphs. But to take into account edges in both directions, we should add the reverse directions to the graph. Let's take a union of reversed edges and original ones.

```
val undirectedEdgeRDD = graph.reverse.edges.union(graph.edges)
val graph = Graph(verRDD, undirectedEdgeRDD)
```

- Now we have an undirected graph with all the edges and directions taken into account, so we can perform the aggregation using aggregateMessages operator:
- val neighbors = graph.aggregateMessages[Int](ectx => ectx.**sendToSrc**(ectx.dstAttr._2), _ + _)
- neighbors.**foreach**(**println**(_))



```
(4,2251352)
(1,8322073)
(5,1706480)
(2,120379)
(6,9805524)
(3,8943034)
```

Algorithm

- We’ll use the following three algorithms in our sample application:
- PageRank on YouTube
- Connected Components on LiveJournal
- Triangle Counting on Facebook