GraphX

What is Spark GraphX?

- *GraphX* is the Spark API for graphs and graph-parallel computation. It includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

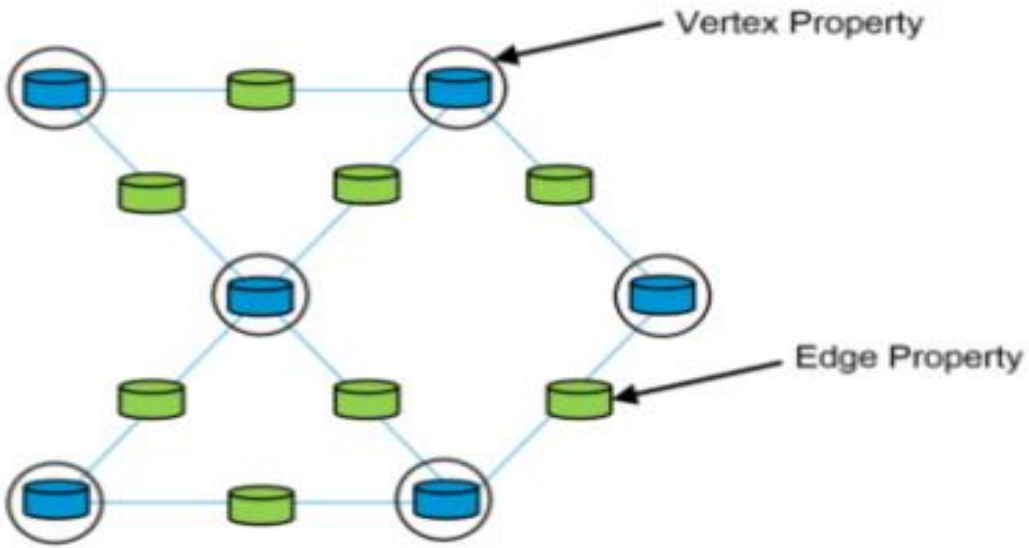


Figure: *Property Graph*

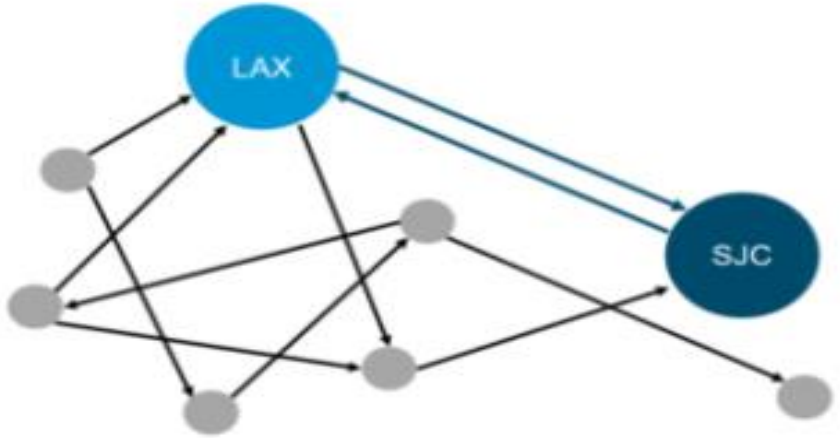


Figure: *An example of property graph*

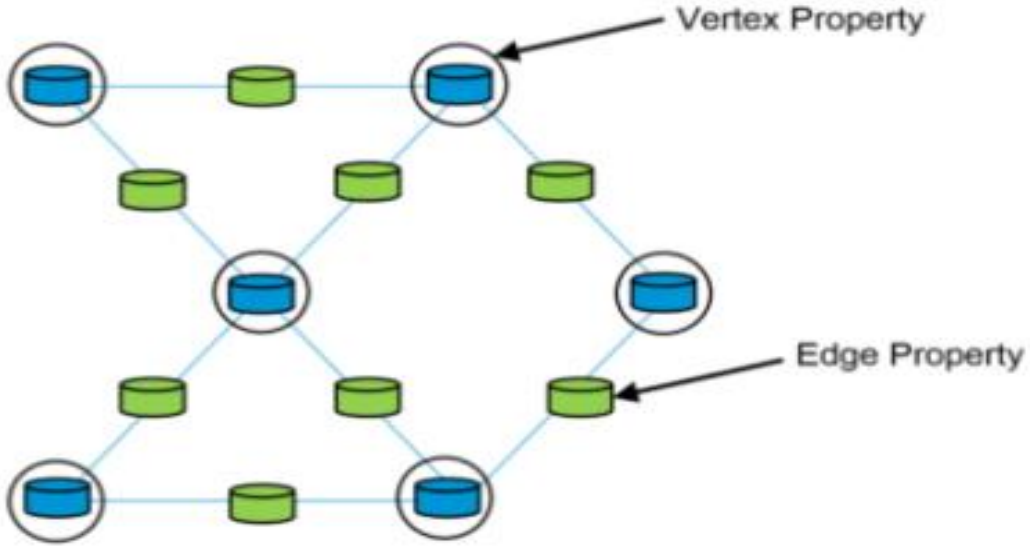


Figure: *Property Graph*

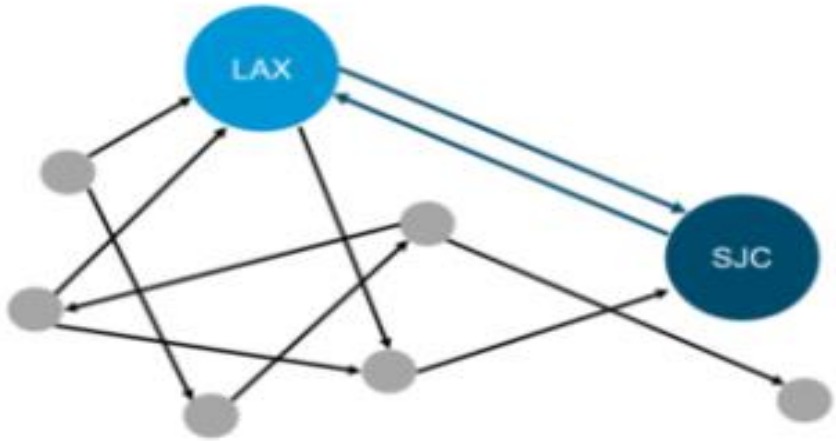


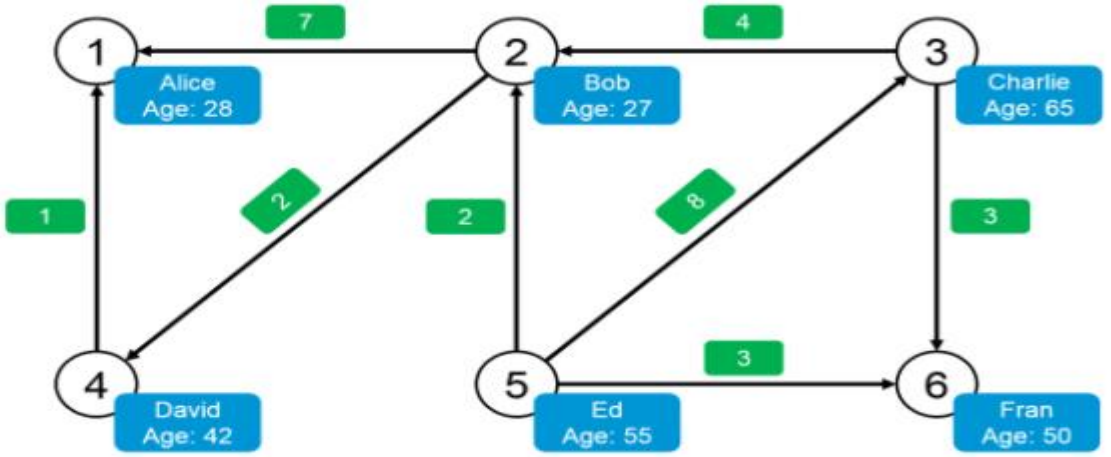
Figure: *An example of property graph*

- GraphX extends the Spark RDD with a Resilient Distributed Property Graph.
- The property graph is a directed multigraph which can have multiple edges in parallel.
- Every edge and vertex has user-defined properties associated with it.
- The parallel edges allow multiple relationships between the same vertices.

- Spark GraphX Features
- The following are the features of Spark GraphX:
- **Flexibility**:
- Spark GraphX works with both graphs and computations. GraphX unifies ETL (Extract, Transform & Load), exploratory analysis, and iterative graph computation within a single system. We can view the same data as both graphs and collections, transform and join graphs with RDDs efficiently and write custom iterative graph algorithms using the Pregel API.
- **Speed**:
- Spark GraphX provides comparable performance to the fastest specialized graph processing systems. It is comparable with the fastest graph systems while retaining Spark's flexibility, fault tolerance, and ease of use.
- **Growing Algorithm Library**:
- We can choose from a growing library of graph algorithms that Spark GraphX has to offer. Some of the popular algorithms are page rank, connected components, label propagation, SVD++, strongly connected components and triangle count.

GraphX with Examples

Let us consider a simple graph as shown in the image below.



- Looking at the graph, we can extract information about the people (vertices) and the relations between them (edges). The graph here represents the Twitter users and whom they follow on Twitter. For e.g., Bob follows Davide and Alice on Twitter.

Displaying Vertices: Further, we will now display all the names and ages of the users (vertices).

```
Displaying Vertices: Further, we will now display all the names and ages of the users (ve
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
graph.vertices.filter { case (id, (name, age)) => age > 30 }
.collect.foreach { case (id, (name, age)) => println(s"$name is $age")}
```

```
David is 42
Fran is 50
Ed is 55
Charlie is 65
```

Displaying Edges: Let us look at which person likes whom on Twitter.

```
for (triplet <- graph.triplets.collect)
{
println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")
}
```

The output for the above code is as below:

```
Bob likes Alice
Bob likes David
Charlie likes Bob
Charlie likes Fran
David likes Alice
Ed likes Bob
Ed likes Charlie
Ed likes Fran
```

Number of followers: Every user in our graph has a different number of followers. Let us look at all the followers for every user.

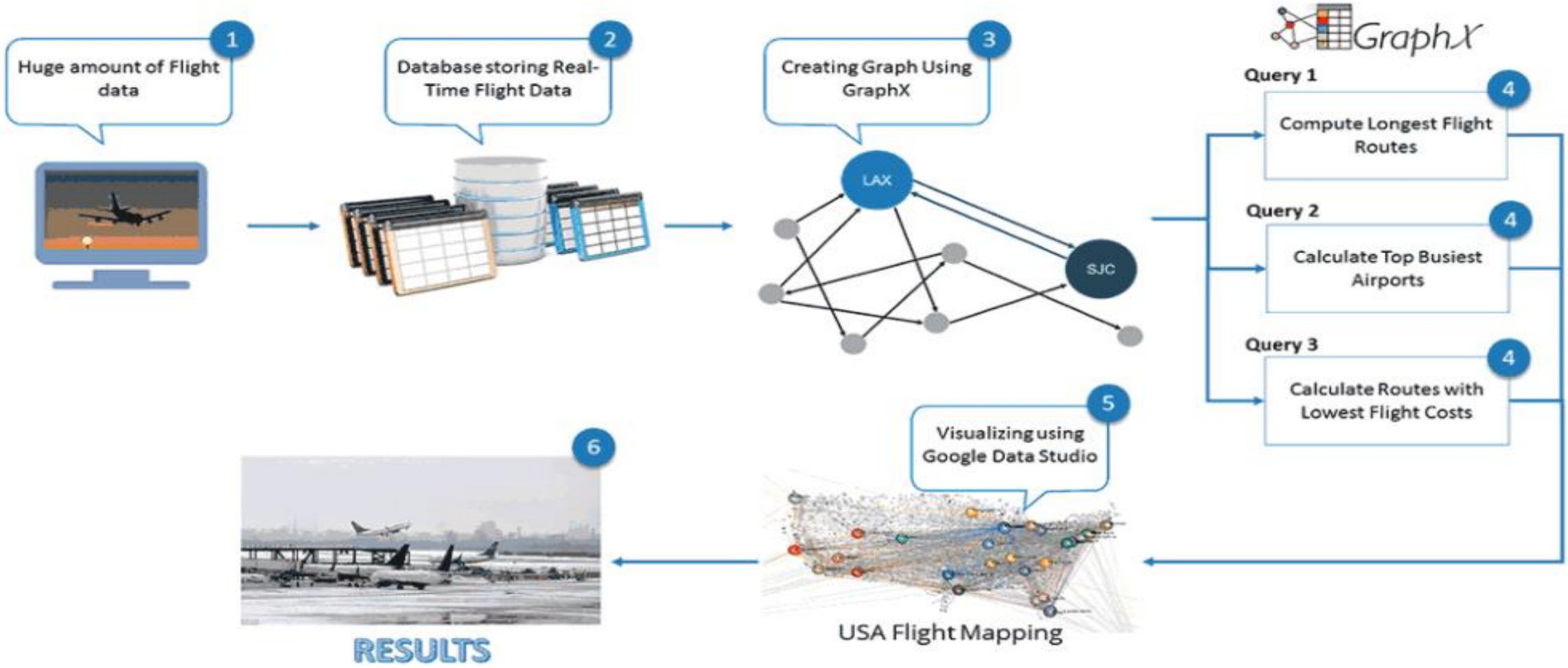
```
// Defining a class to more clearly model the user property
case class User(name: String, age: Int, inDeg: Int, outDeg: Int)
// Creating a user Graph
val initialUserGraph: Graph[User, Int] = graph.mapVertices{ case (id, (name,
age)) => User(name, age, 0, 0) }

// Filling in the degree information
val userGraph = initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {
case (id, u, inDegOpt) => User(u.name, u.age, inDegOpt.getOrElse(0), u.outDeg)
}.outerJoinVertices(initialUserGraph.outDegrees) {
case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg,
outDegOpt.getOrElse(0))
}
for ((id, property) <- userGraph.vertices.collect) {
println(s"User $id is called ${property.name} and is liked by ${property.inDeg}
people.")
```

Use Case — Dataset:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	dOfM	dOfW	carrier	tailNum	flNum	origin_id	origin	dest_id	dest	crsdeptime	deptime	depdelaymins	crsarrtime	arrtime	arrdelaymins	crselapsedtime	dist
2	1	3	AA	N338AA	1	12478	JFK	12892	LAX	900	914	14	1225	1238	13	385	2475
3	2	4	AA	N338AA	1	12478	JFK	12892	LAX	900	857	0	1225	1226	1	385	2475
4	4	6	AA	N327AA	1	12478	JFK	12892	LAX	900	1005	65	1225	1324	59	385	2475
5	5	7	AA	N323AA	1	12478	JFK	12892	LAX	900	1050	110	1225	1415	110	385	2475
6	6	1	AA	N319AA	1	12478	JFK	12892	LAX	900	917	17	1225	1217	0	385	2475
7	7	2	AA	N328AA	1	12478	JFK	12892	LAX	900	910	10	1225	1212	0	385	2475
8	8	3	AA	N323AA	1	12478	JFK	12892	LAX	900	923	23	1225	1215	0	385	2475
9	9	4	AA	N339AA	1	12478	JFK	12892	LAX	900	859	0	1225	1204	0	385	2475
10	10	5	AA	N319AA	1	12478	JFK	12892	LAX	900	929	29	1225	1245	20	385	2475

- Use Case: Flight Data Analysis using Spark GraphX
- Now that we have understood the core concepts of Spark GraphX, let us solve a real-life problem using GraphX. This will help give us the confidence to work on any Spark projects in the future.
- **Problem Statement**:
- *To analyze Real-Time Flight data using Spark GraphX, provide near real-time computation results, and visualize the results using Google Data Studio.*
- **Use Case — Computations to be done**:
- Compute the total number of flight routes
- Compute and sort the longest flight routes
- Display the airport with the highest degree vertex
- List the most important airports according to PageRank
- List the routes with the lowest flight costs

The following illustration clearly explains all the steps involved in our Flight Data Analysis.



- object airport {

  def main(args: Array[String]){

  //Creating a Case Class Flight
  case class Flight(dofM:String, dofW:String, ... ,dist:Int)

  //Defining a Parse String function to parse input into Flight class
  def parseFlight(str: String): Flight = {
  val line = str.split(",")
  Flight(line(0), line(1), ... , line(16).toInt)
  }
  val conf = new SparkConf().setAppName("airport").setMaster("local[2]")
  val sc = new SparkContext(conf)
  //Load the data into a RDD

  val textRDD = sc.textFile("/home/edureka/usecases/airport/airportdataset.csv")

  //Parse the RDD of CSV lines into an RDD of flight classes
  val flightsRDD = Map ParseFlight to Text RDD

  //Create airports RDD with ID and Name
  val airports = Map Flight OriginID and Origin
  airports.take(1)

  //Defining a default vertex called nowhere and mapping Airport ID for printlns
  val nowhere = "nowhere"
  val airportMap = Use Map Function .collect.toList.toMap

  //Create routes RDD with sourceID, destinationID and distance
  val routes = flightsRDD. Use Map Function .distinct
  routes.take(2)

  //Create edges RDD with sourceID, destinationID and distance
  val edges = routes.map{(Map OriginID and DestinationID) => Edge(org_id.toLong, dest_id.toLong, distance)}
  edges.take(1)

  //Define the graph and display some vertices and edges
  val graph = Graph(Airports, Edges and Nowhere)
  graph.vertices.take(2)
  graph.edges.take(2)

- //Query 1 - Find the total number of airports
  val numairports = Vertices Number

  //Query 2 - Calculate the total number of routes?
  val numroutes = Number Of Edges

  //Query 3 - Calculate those routes with distances more than 1000 miles
  graph.edges.filter { Get the edge distance)=> distance > 1000}.take(3)

  //Similarly write Scala code for the below queries
  //Query 4 - Sort and print the longest routes
  //Query 5 - Display highest degree vertices for incoming and outgoing flights of airports
  //Query 6 - Get the airport name with IDs 10397 and 12478
  //Query 7 - Find the airport with the highest incoming flights
  //Query 8 - Find the airport with the highest outgoing flights
  //Query 9 - Find the most important airports according to PageRank
  //Query 10 - Sort the airports by ranking
  //Query 11 - Display the most important airports
  //Query 12 - Find the Routes with the lowest flight costs
  //Query 13 - Find airports and their lowest flight costs
  //Query 14 - Display airport codes along with sorted lowest flight costs