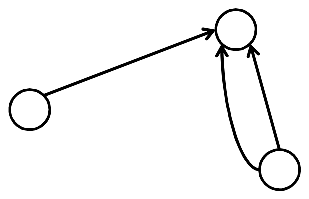
DADS 6002 / CI 7301
Big Data Analytics

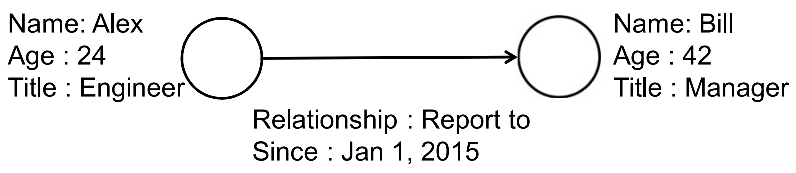
Graphs

- A graph is a data structure composed of vertices and edges.
- A graph can be directed (edges have directions) or undirected.
- A directed Multigraph is a directed graph that contains pairs of vertices connected by two or more edges.

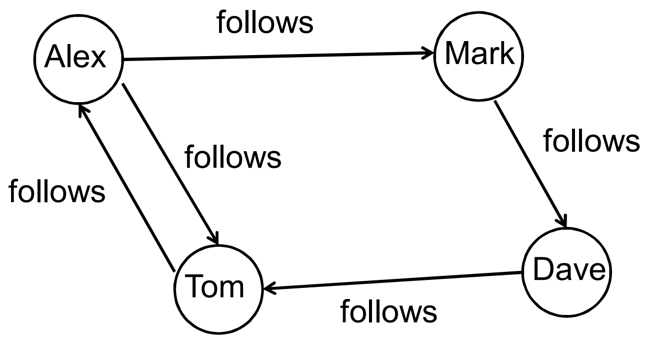


Graphs

- Property Graph is a directed multigraph that has data, e.g. attributes and labels, with the vertices and the edges.



Graphs



Social Network on Twitter

GraphX

- GraphX is a distributed graph analytics framework. It extends Spark for large-scale graph processing.
- It provides a higher-level abstraction for graph analytics than that provided by the spark core API.
- GraphX provides both fundamental graph operators and advanced operators implementing graph algorithms such as Page Rank, strongly connected components and triangle count.

GraphX

- It also provides an implementation of Google’s Pregel API. These operators simplify graph analytics tasks.
- A graph analytics pipeline generally consists of the following steps:

a) Read raw data
b) Preprocess data (e.g. cleanse data)
c) Extract vertices and edges to create a property graph.
d) Slice a subgraph.
e) Run a graph algorithm.
f) Analyse the results.
g) Repeat steps d) to f) with another slice of the graph.

GraphX API

- The graphX API provides data types for representing graph-oriented data and operators for graph analytics.

Data Abstractions

- VertexRDD – represents a distributed collection of vertices in a property graph.
- VertexRDD stores only one entry for each vertex.
- Each vertex is represented by a key-value pair, where key is a unique id and value is the data associated with a vertex.
- The data type of the key is VertexId (64 bit long Int). The value can be of any type.

GraphX API

- VertexRDD is a generic or parameterized class e.g VertexRDD[VD] where VD is the type parameter specifying the data type of value associated with a vertex. VD can be Int, long, Double, String or a user defined type.
- The Edge class abstracts a directed edge in a property graph.
- An instance of Edge class contains source vertex id, destination vertex id and edge attributes of type Int, long, Double, String and a user defined type.

Graph

- Graph is the class for representing property graphs.
- Like RDD, the instance of Graph is immutable, distributed and fault tolerant.
- GraphX partitions a Graph instance across a cluster using vertex partitioning.
- A Graph instance contains a pair of typed RDD.
- One is a partitioned collection of vertices and the other is a partitioned collection of edges.

Graph

- The Graph class provides methods to transform and analyze vertices and edges.
- GraphX provides an object named Graph, which provides factory methods for constructing graphs from RDDs, e.g. a pair of RDDs representing vertices and edges.

Graph

- Create a Graph

```
import org.apache.spark.graphx._
case class User(name:String, age:Int)
val users = List( (1L, User(“John”, 26)), (2L, User(“Bill”,
42)), (3L, User(“Carol”, 18)), (4L, User(“Tom”, 30)) )
val userRDD = sc.parallelize(users)
val follows = List( Edge(1L, 2L,1),  Edge(2L, 3L, 1),
                 Edge(3L, 4L, 1), Edge(4L, 5L, 1 ))
val followsRDD = sc.parallelize(follows)
val defaultUser = User(“NA”, 0 )
val socialGraph = Graph( userRDD, followsRDD,
                         defaultUser )
```

Graph

- The third argument for Graph object e.g. defaultUser represents the default set of properties which is assigned to vertices that have not been explicitly assigned any properties.
- Graph Properties

Number of edges – numEdges

`val numEdges = socialGraph.numEdges`

Number of vertices - numVertices

`val numVertices = socialGraph.numVertices`

Graph

# of incoming edges - inDegrees

val inDegrees = socialGraph.inDegrees

The inDegrees is an RDD containing pairs of vertexId and its inDegree

# of outgoing edges – outDegrees

val outDegrees = socialGraph.outDegrees

The outDegrees is an RDD containing pairs of vertexId and its outDegree

Graph Operators

- Property Transformation
- Structure Transformation
- Join
- Aggregation
- Graph-parallel computation
- Graph Algorithm

Property Transformation Operator

- These operators allow us to change the properties of edges or vertices in a graph. It returns a new graph with the modified properties.
- MapVertices – the method applies a user-defined transformation to each vertices in a property graph. It takes a function as an argument and returns a new graph. The function should take a pair of vertex id and vertex attributes as arguments and returns new vertex attributes.

```
val updateAges = socialGraph.mapVertices((vertexId, user)
                          => User(user.name, user.age + 1 ))
```

Property Transformation Operator

- mapEdges – the method applies a user defined transformation to each edge in a property graph. It takes a function as an argument and returns a new graph. The function should take the attributes of an edge as argument and returns a new edge atrributes.

```
val followGraph = socialGraph.mapEdges( (n) =>
                          "follows" )
```

Property Transformation Operator

- mapTriplets – the method applies a user-specified transformation to each triplet in a property graph by modifying the properties of each edge in the source graph. It takes a function as an argument and returns a new graph. The function should take an edgeTriplet as argument and return a new property of the edge in the edgeTriplet.

```
val weightGraph = socialGraph.mapTriplets { t =>
                                   if ( t.srcAttr.age >= 30 )
                                          2 else 1 }
```

Structure Transformation Operators

- reverse – the method reverses the direction of all the edges in a property graph. It returns a new property graph.

```
val reverseGraph = socialGraph.reverse
```

- Subgraph – the method applies a user-defined filter to each vertex and edge. It returns a subgraph of the source graph, that contains only the vertices and edges that satisfy the filtering conditions.

Structure Transformation Operators

- The method takes two predicates (functions returning Boolean results). The first predicate takes an EdgeTriplet as argument and the second predicate takes a pair of vertexId and the vertex properties.

```
val subgraph = weightedGraph.subgraph (
        edgeTriplet => edgeTriplet.attr > 1,
        (vertexId, vertexProperty) => true )
```

Structure Transformation Operators

- Mask – The method takes a graph as argument or input graph and returns a subgraph of the source graph containing all vertices and edges in the input graph. The vertices and edges of the subgraph exist in both the input graph and the source graph, however, the vertex and edge properties in the returned graph are from the source graph.

Structure Transformation Operators

```
val femaleConnections = List( Edge( 2L, 3L, 0 ), Edge(3L,
1L, 0), Edge(3L, 4L, 0 ) )
val femaleConnectionRDD =
        sc.parallelize( femaleConnections )
val femaleGraphMask = Graph.fromEdges(
femaleConnectionRDD, defaultUser )
val femaleGraph = socialGraph.mask( femaleGraphMask )
```

Structure Transformation Operators

- groupEdges – the method merges parallel edges in a property graph.
- It takes a function as argument. The function takes a pair of edges properties as arguments, merge them and returns a new property.

Structure Transformation Operators

```
val multiEdges = (Edge(1L,2L,100), Edge(1L,2L,200),
Edge(2L,3L,300), Edge(2L,3L,400), Edge(3L,1L,200),
Edge(3L,1L,300))
val multiEdgeRDD = sc.parallelize(multiEdges)
val defaultVertexPropery = 1
val multiEdgeGraph = Graph.fromEdges(multiEdgeRDD,
                        defaultVertexProperty)
val singleEdgeGraph =
        multiEdgeGraph.groupEdges((edge1, edge2) =>
                                edge1 + edge2 )
```

Join Operators

- Join Operators updates existing properties or add a new property to the vertices in a graph. Many join operators are provided in GraphX.
- joinVertices is the method updates the vertices in the source graph with a collection of vertices provided to it as input.
- It takes two arguments. The first is an RDD of pairs between vertex id and vertex data (properties). The second is a user-specified function that updates a vertex in the source graph using its current properties and new input data. Vertices that do not have entries in the input RDD retain their current properties.

Join Operators

```
val correctAges = sc.parallelize ( List((3L,28),
(4L,26))
val correctGraph =
socialGraph.joinVertices(correctAges)((id, user,
correctAge) => User(user.name, correctAge))
```

Join Operators

- OuterJoinVertices - the method adds new properties to the vertices in the source graph.
- It takes two arguments. The first is an RDD of pairs of vertex id and vertex properties. The second is a user-specified function that takes as argument a triplet containing a vertex id, current vertex attributes, and optional attributes in the input RDD for the same vertex.
- The function returns new attributes for each vertex.

Join Operators

- The input RDD should contain at most one entry for each vertex in the source graph. If the input RDD does not have an update entry for a vertex in the source graph, the user-defined function receives None as the third argument when the function is called on that vertex.
- The outerJoinVertices can update current attributes or add new attributes to the vertices in the source graph.

Join Operators

- Suppose we want to add a new property called city to each user in the source graph.

```
case class UserWithCity ( name : string, age : int, city : string )
val userCities = sc.parallelize(List((1L,”Boston”), (3L,”New
York”), (5L,”London”), (7L, “Bombay”), (9L, “Tokyo”), (10L, “Palo
Alto”)))
val socialGraphwithCity =
socialGraph.outerJoinVertices(userCities)((id, user, cityOpt) =>
cityOpt match
{ case Some(city) => UserWithCity( user.name, user.age, city )
case None => UserWithCity( user.name, user.age, “NA” )))
```

Aggregation Operators

- Aggregation Operators
  - aggregateMessages aggregates values for each vertex from neighboring vertices and connecting edges. It uses two user-defined functions to do the aggregations. These functions are called for every triplet in a property graph.
  - The first function takes as argument an EdgeContext. The edgeContext class provides not only access to the properties of the source and destination vertices in a triplet as well as the edge connecting them but also functions to send messages to the source and destination vertices in a triplet.

Aggregation Operators

- The second function takes two messages as arguments and merge/combine the two messages into one and return the single message.
- Finally, the method returns an RDD of pairs of vertex id and aggregated messages.
- The following code calculate the number of followers of each user in a social network graph using aggregateMessages method.

Aggregation Operators

```
val followers = socialGraph.aggregateMessages[Int]
( edgeContext => edgeContext.sendToDst(1), (x,y)
=> (x + y))
```

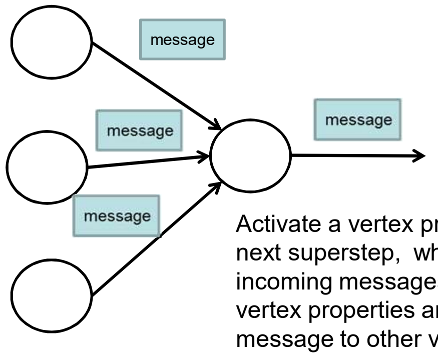
The above code is just an example to calculate the number of followers. A simpler approach to calculate the number of followers is the follows:

```
val followers = socialGraph.inDegrees
```

Graph Parallel Operators

- The operators allows us to implement custom iterative graph algorithms for processing large scale graph-oriented data.
- Pregel – the method implement a variant of Google's Pregel algorithm which is based on the Bulk Synchronous Parallel (BSP) algorithm.
- In Pregel, a graph computation consists of a sequence of iterations, also known as supersteps.
- During a superstep, a vertex is activated by messages sent to it from the previous superstep, and a user-defined function, also known as vertex program, is invoked on the vertex in parallel with others.

Graph Parallel Operators



Activate a vertex program at the next superstep, which combines all incoming messages, update the vertex properties and send out a message to other vertices

Graph Parallel Operators

- A vertex program processes the aggregate of the messages sent to a vertex in the previous superstep, updates the properties of a vertex and its outgoing edges, and sends messages to other vertices.
- If no messages are sent to a vertex in a superstep, the vertex program will not be invoked on it in the next superstep.

Graph Parallel Operators

- At the current superstep, all the vertex programs are allowed to finish processing the messages from the previous superstep before the next superstep can start.
- These iterations are repeated until there are no messages in transit and all the vertices are inactive (no further work to do).

Graph Parallel Operators

- The GraphX implements pregel method for processing large-scale graph data.
- The method takes two sets of arguments and returns a graph.
- The first set contains three arguments:

1. The initial message that each vertex will receive in the first superstep.
2. The maximum number of iterations.
3. The direction in which messages will be sent.

Graph Parallel Operators

- The second set of arguments also contains three arguments :

1. A user-defined update vertex function that computes the new vertex properties.
2. A user-defined send function that computes optional messages for the neighboring vertices.
3. A user-defined aggregate function that merges or combines messages sent to a vertex in the previous superstep.

Graph Parallel Operators

- The user-defined update program takes three arguments:

1. ID of a vertex,
2. Current properties of the vertex,
3. An aggregation of all messages sent to it in the previous superstep (the aggregation is computed by the user-defined aggregate function).

Graph Parallel Operators

```
val firstMessage = 0.0
val iterations = 20
val edgeDirection = EdgeDirection.out
val influenceGraph =
        inputGraph.pregel(firstMessage, iterations,
        edgeDirection)
        ((a,b,c) => ….,    //  updateVertex
        edgeTriplet => ….,   //  sendMsg (compute a
message value for each edgeTriplet )
        (x,y) => …. )    //  aggregateMsgs
```

Finding Shortest Paths from a vertex to others using Pregel

```
import org.apache.spark.graphx._

val inf = Double.PositiveInfinity

val vertices = sc.parallelize(Seq((1L, 0.0), (2L, inf), (3L, inf),
(4L, inf)))
val edges = sc.parallelize(Seq(Edge(1L, 2L, 5.0), Edge(1L,
3L, 7.0), Edge(2L, 4L, 12.0), Edge(3L, 4L, 8.0)))

val graph = Graph(vertices, edges)
```

Finding Shortest Paths from a vertex to others using Pregel

```
val sssp = graph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b) // Merge Message
)
```

Graph Algorithm

- The aggregateMessages and pregel operators are powerful tools for implementing custom graph algorithms.
- GraphX also provides built-in implementations for a few common graph algorithms.
- These algorithms can be invoked as method calls on the Graph class.

PageRank

- Assume page A has pages B, C, D ..., which point to it. The parameter d is a damping factor which can be set between 0 and 1. Usually set d to 0.85. The PageRank of a page A is given as follows:

$$PR(A) = 1 - d + d\left(\frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \cdots\right)$$

where L(w) is the out degree of page w

PageRank

- PageRank of a web page (or document) can indicate the importance of the web page.The PageRank of a web page is based on the PageRank of the web pages linking to it (the higher the more contribution) as well as the number of outgoing links of the web pages (the lower the more contribution).

PageRank

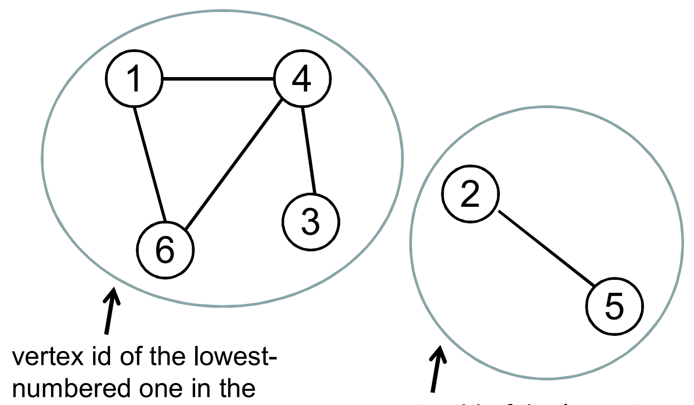
- PageRank of a user in a social network can indicate the influence rank of the user. The influence rank of a user is based on the influence rank of the followers (the higher the more contribution) and the number of people followed by each follower (number of outgoing links of the follower, the lower the more contribution).

Connected Components

- The connectedComponents method implements the connected components algortihm on an undirected graph (identify subgraphs that are not connected to others)
- It computes the connected component membership of each vertex, which is the vertex id of the lowest-numbered vertex in the connected component.
- It returns a graph, where each vertex's property is the vertex id of the lowest-numbered vertex in the connected component.

```
val connectedComponentsGraph =
                inputGraph.connectedComponents()
```

Connected Components

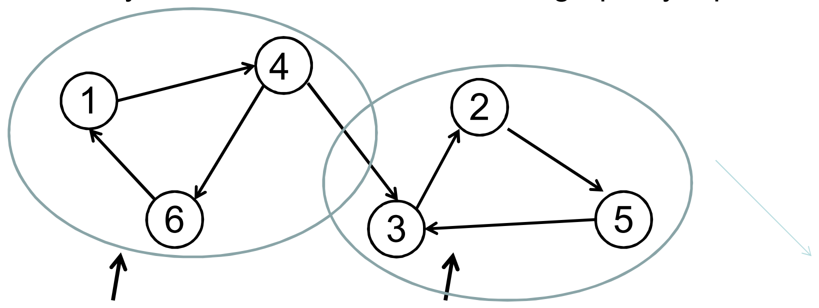


vertex id of the lowest-numbered one in the connected component = 1

vertex id of the lowest-numbered one in the connected component = 2

Strongly Connected Component

- A strongly connected component (SCC) of a directed graph is a subgraph containing vertices that are reachable from every other vertex in the same subgraph by a path.



vertex id of the lowest-numbered one in the connected component = 1

vertex id of the lowest-numbered one in the connected component = 2

Strongly Connected Component

- The stronglyConnectedComponents method finds SCC for each vertex and returns a graph where a vertex's property is the lowest vertex id in the SCC containing the vertex.

```
val sccGraph =
        inputGraph.stronglyConnectedComponents()
```

Applications of SCC

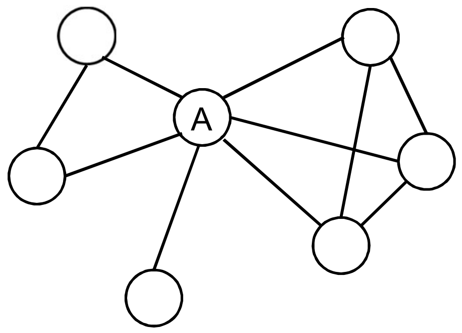
- SCC can be used to compute the connectivity of different network configurations when measuring routing performance in multihop wireless networks.
- The SCC algorithms can be used in social media to find groups of users that are strongly connected and may share common interests, and so we may suggest the commonly liked pages or videos to the people in the group who have not yet liked those pages or videos.

TriangleCount

- The triangleCount method computes for each vertex the number of trangles containing it. A vertex is a part of a triangle if it has two neighboring vertices with an edge between them. The triangleCount method returns a graph, where a vertex's property is the number of triangles containing it.

```
val triangleCountGraph =
                              inputGraph.triangleCount()
```

TraingleCount



TriangleCount for vertex A= 4

Triangle Counts

- Triangle counts can be used to detect communities in a social media and measure the cohesiveness of those communities.
- It can also be used to determine the stability of a graph.

TraingleCount

- TriangleCount of a vertex can be used to compute a clustering coefficient for the vertex, as well as the whole graph (the average of the clustering coefficients) which can determine the strong communities in a social media network.

$$Clustering\ Coefficent = \frac{Triangle\ Count}{\binom{d}{2}}$$

Where d is the degree or # of neighboring vertices of the vertex