Ahsanullah University of Science & Technology

Department of Computer Science and Engineering

Assignment-4

Prepared for:

Mr. Md. Zahid Hossain
Lecturer Grade-II, CSE

Course Number & Name:

CSE 4262 - Data Analytics Lab

Prepared by:

Shuvashis Sarker
20200104116
Lab group: ***DAGr-B1***

Date: **10.07.2024**

CSE 4262 Data Analytics Lab

Lab 4: GraphX: Graph-processing capabilities of Spark

Outcomes: After this lab students will be able to:

- Understand the graph-processing frameworks of Spark
- Understand the implementation of Graph Algorithms including Page Rank, Connected Component, Triangle Counting, and others using GraphX.

What are Graphs:

A Graph is a mathematical structure amounting to objects in which some pairs of objects are related in some sense. These relations can be represented using edges and vertices forming a graph. The vertices represent the objects and the edges show the various relationships between those objects.

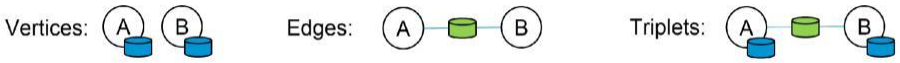
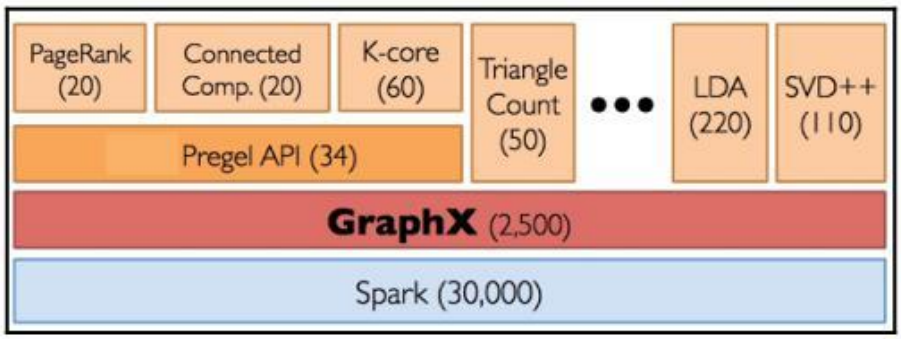


Figure: *Spark GraphX Tutorial – Vertices, edges, and triplets in graphs*

In computer science, a graph is an abstract data type meant to implement the undirected graph and directed graph concepts from mathematics, specifically the field of graph theory. A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

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. GraphX is a layer over Spark, thus it leverages all the interesting things about Spark distributed processing, the algorithms, the versioned computation graph, and so forth. Interestingly, a couple of ML algorithms are written using GraphX APIs. This diagram shows the layers and the relationships between GraphX, Spark, and the algorithms.



GraphX extends the Spark RDD with a Resilient Distributed Property Graph. The property graph is a directed multigraph that 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. GraphX has a rich yet simple computational model:

- It consists of vertices connected by edges.
- It is a property graph, which means the vertices and edges can have arbitrary objects as properties, and most importantly, these properties are visible to the APIs.

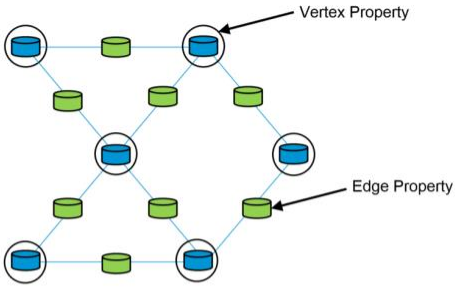


Figure: *Property Graph*

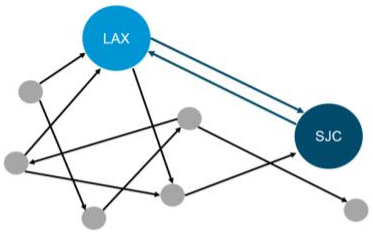
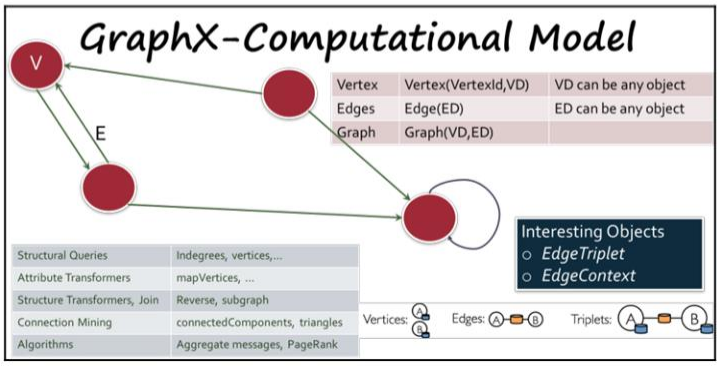
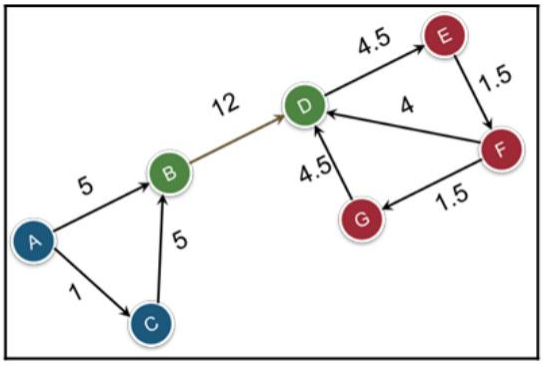


Figure: *An example of property graph*



A vertex consists of a vertexID(64-bit long int) and a property object. The property object Is required, but you can always have an arbitrary value if you are not using the property. An edge consists of a source vertexID, a destination vertexID, and a property object. The APIs, have the vertex and edge parameterized over object types.



This is a graph from the lecture given by Prof. Jeffrey D. Ullman at Stanford University on graphs and social Networks. It is a giraffe graph with two strong cliques connected by a weak edge.

vertexID	Property-Name	Property-Age
1	Alice	18
2	Bernie	17
3	Cruz	15
4	Donald	12
5	Ed	15
6	Fran	10
7	Genghis	854

Source ID	Destination ID	Property
1	2	5
1	3	1
3	2	5
2	4	12
4	5	4
5	6	2
6	7	2
7	4	5
6	4	4

Graph Processing with Apache Spark: A Breakdown

GraphX unifies ETL (Extract, Transform, Load), exploratory analysis, and iterative graph computation within a single, cohesive system. Consequently, you can view the same data as both graphs and collections, manipulate and merge it in various ways, and iteratively apply graph algorithms. Below, we have an example usage of graph processing with GraphX, implemented using PySpark:

Graph Representation in GraphX: GraphX represents a graph as a collection of vertices and edges. Vertices and edges can store associated properties or attributes, making it flexible for a wide range of applications. PySpark code snippet for creating a graph:

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext,
SparkSession from graphframes import *

conf = SparkConf().setAppName('graph_processing').set('spark.jars.packages',
'graphframes:graphframes:0.8.1-spark3.0-s_2.12') sc = SparkContext(conf=conf)

sqlContext = SQLContext(sc)
spark = SparkSession(sc)

# Vertices DataFrame
v = spark.createDataFrame([("1", "John", 28),("2", "Mike", 30),("3", "Sara", 25)],
["id", "name", "age"])

# Edges DataFrame
e = spark.createDataFrame([("1", "2", "friend"),("2", "3", "follows") ,("3",
"1", "friend")], ["src", "dst", "relationship"])

# Create a GraphFrame
g = GraphFrame(v, e)
```

Graph Operations and Transformations: GraphX provides a wide range of operations and transformations to perform graph computations efficiently. PySpark code snippets for common graph operations:

- **Vertex and Edge Operations:**

```
g.inDegrees.show()
g.outDegrees.show()
g.degrees.show()
# Query: Count the number of "follow" connections in the graph.
```

```
g.edges.filter("relationship = 'follows'").count()


#Filtering vertices and edges

filtered_vertices = graph.vertices.filter("id > 1")
filtered_edges = graph.edges.filter("relationship == 'friends'")
```

- **Graph Algorithms:**

```
# Run PageRank algorithm, and show results.
results = g.pageRank(resetProbability=0.01, maxIter=20)
results.vertices.select("id", "pagerank").show()

# Connected Components algorithm
connected_components = graph.connectedComponents()
```

PageRank is an algorithm that determines the importance/popularity of a web page based on the number of links pointing to it. So, the PageRank of a page being pointed to by 100 web pages would normally be higher than one that is pointed to by only 10 pages. It also considers the popularity of the web pages that are pointing to it. For example, if you have a page that is linked by Google, the PageRank would be much higher. It has three flavors: first, the **PageRank** algorithm, which is a dynamic version that takes a tolerance and runs until it converges to that tolerance. Second, there is a **staticPageRank** algorithm that takes several iterations and runs that many times, rather than seeking convergence. Finally, there is **personalizedPageRank**, which calculates PageRank concerning a given vertex, probably a good algorithm to implement the "people you may know" or "find popular people in a specified subcommunity" functions.

Task 1: Constructing a Graph.

- Use the information of the above giraffe graph and write the code to create a graph using `g = GraphFrame(v, e)` and find the degrees of the graph(g).
- Show all the nodes in g graph using the following commands:

```
nodes = [(r.id) for r in
g.vertices.select("id").collect()] nodes
```

- Show all the edges in g graph using the following commands:

```
edges = [(r.src, r.dst) for r in g.edges.select("src", "dst")
.collect()] edges
```

- Find out the **connected components** of the graph.

  The connected components algorithm has a subgraph in which each vertex is accessible to each other vertex by following edges. Every connected graph component in this algorithm is tagged with an ID from its lower vertex.

Task 2: Implementation of the LPA and Triangle Count Algorithms

- Create a Graph with the following vertices and edge information

```
# Vertices DataFrame
v = spark.createDataFrame([("A", "ARON" ,350 ),("B", "BILL" ,360 ),("C",
"CLAIR" ,195 ),("D", "DANIEL",90),("E", "ERIC" ,90),("F", "FRANK" ,215 ),("G",
"GRAHAM",30 ),("H", "HENRY" ,25 ),("I", "INNA" ,25 ),("J", "JEN" ,20 )], ["id",
"name", "total_seconds"])

# Edges DataFrame
```

```
e = spark.createDataFrame([("A", "B", 60),("B", "A", 50),("A", "C", 50),("C",
"A", 100),("A", "D", 90),("C", "I", 25),("C", "J", 20),("B", "F", 50),("F",
"B", 110),("F", "G", 30),("F", "H", 25),("B", "E",
90)],["src","dst","relationship"])
```

- Detect the community of the graph using **Label Propagation Algorithm (LPA).** Use the following commands to get the results of the LPA.

```
result = g.labelPropagation(maxIter=5)
result.sort(['label'],ascending=[0]).show()
```

Each node in the network is initially assigned to its own community. At every superstep, nodes send their community affiliation to all neighbors and update their state to the mode community affiliation of incoming messages.

LPA is a standard community detection algorithm for graphs. It is very inexpensive computationally, although (1) convergence is not guaranteed and (2) one can end up with trivial solutions (all nodes are identified into a single community).

- Implement the Triangle Count algorithm and find the number of triangles passing through each vertex. Using the following command.

```
g.triangleCount().show()
```

The Triangle Counting Algorithm provides the two nearby vertices and an edge in between them that presents a vertex.

Task 3: Implementation of Motifs

Finding motifs helps to execute queries to discover structural patterns in graphs. Network motifs are patterns that occur repeatedly in the graph and represent the relationships between the vertices. GraphFrames motif finding uses a declarative Domain Specific Language (DSL) for expressing structural queries. For example, graph.find("(a)-[e]->(b); (b)-[e2]->(a)") will search for pairs of vertices a,b connected by edges in both directions. It will return a DataFrame of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. In this case, the returned columns will be “a, b, e, e2.” The query can be invoked by using the find function, where the motif (in quotation marks) is expressed as the first parameter of the function. The following example will search for pairs of vertices a,b connected by edge e and pairs of vertices b, and c connected by edge e2. It will return a DataFrame of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif.

- `g.find("(a)-[e]->(b); (b)-[e2]->(a)").show()`
- ```
  mutualFriends = (g1
                   .find("(a)-[]->(b); (b)-[]->(c); (c)-[]->(b); (b)-[]->(a)")
                   .dropDuplicates()
                        )
  mutualFriends.show(100, False)
  ```
- `mutualFriends.filter('a.id == 2 and c.id == 3').show(10, False)`
  ```

Assignment

You have two Text files, users.txt and relationships.txt, which contain information about the users and their relationships in a social network. Create a graph to represent these data. Perform each of the following tasks-

1. Find all triadic closures in the graph (i.e., sets of three users who are all connected to each other. These users may be friends with each other or follow each other).
2. Identify the patterns where a user follows another user, who in turn follows a third user (i.e., chain patterns).
3. Identify the patterns where a user follows another user, but is not followed back.
4. Determine the user(s) with the highest number of followers.
5. Determine the user(s) with the lowest number of friends.
6. Detect communities using the Label Propagation Algorithm (LPA) and display the community ID for each vertex.

Ans:

1.

```
!pip install pyspark

!pip install graphframes

from pyspark import SparkConf, SparkContext

from pyspark.sql import SQLContext, SparkSession

from graphframes import GraphFrame


conf=SparkConf().setAppName('graph_processing').set('spark.jars.packages','graphframes:graphframes:0.8.1-
spark 3.0-s_2.12')

sc = SparkContext(conf=conf)

sqlContext = SQLContext(sc)

spark = SparkSession(sc)

user_data = spark.read.csv('/content/user.txt', header=True, inferSchema=True)

relationship_data = spark.read.csv('/content/relationships.txt', header=True, inferSchema=True)

vertices = user_data.withColumnRenamed('id', 'id').withColumnRenamed('name', 'name')

edges=relationship_data.withColumnRenamed('src','src').withColumnRenamed('dst','dst').withColumnRenamed

('relationship', 'relationship')

g = GraphFrame(vertices, edges)

vertices.show()

edges.show()
```

```
triangles = g.find("(a)-[e1]->(b); (b)-[e2]->(c); (c)-[e3]->(a)").dropDuplicates()

triangles.show(truncate=False)
```

output:

```
+---+-----+
| id| name|
+---+-----+
|  1| John|
|  2| Mary|
|  3|Steve|
|  4|Karen|
|  5| Paul|
|  6|Alice|
|  7|  Bob|
|  8|  Eve|
|  9| Mike|
+---+-----+
```

```
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  1|  2|     friends|
|  2|  3|     friends|
|  3|  4|     friends|
|  4|  5|     friends|
|  5|  1|     friends|
|  1|  6|     friends|
|  6|  7|     friends|
|  7|  8|     friends|
|  8|  1|     friends|
|  9|  6|     friends|
```

```
|  1|  3|      follows|
|  2|  4|      follows|
|  3|  5|      follows|
|  4|  1|      follows|
|  3|  1|      follows|
|  5|  2|      follows|
|  5|  1|      follows|
|  6|  8|      follows|
|  7|  9|      follows|
|  8|  1|      follows|
+---  +---+------------           +
only showing top 20 rows
```

```
+-------------------+----------+---------------+----------+--------+
|a                  |e1        |b              |e2        |c       |e3
|
+-------------------+----------+---------------+----------+--------+
|{5, Paul}  |{5, 1, friends}|{1, John}  |{1, 3, follows}|{3, Steve}|{3, 5,
follows}|
|{5, Paul}  |{5, 1, follows}|{1, John}  |{1, 3, follows}|{3, Steve}|{3, 5,
follows}|
|{9, Mike}  |{9, 6, friends}|{6, Alice}|{6, 7, friends}|{7, Bob}  |{7, 9,
follows}|
|{3, Steve}|{3, 5, follows}|{5, Paul}  |{5, 2, follows}|{2, Mary}  |{2, 3,
friends}|
|{6, Alice}|{6, 8, follows}|{8, Eve}  |{8, 1, follows}|{1, John}  |{1, 6,
friends}|
|{2, Mary}  |{2, 4, follows}|{4, Karen}|{4, 5, friends}|{5, Paul}  |{5, 2,
follows}|
|{2, Mary}  |{2, 3, friends}|{3, Steve}|{3, 1, follows}|{1, John}  |{1, 2,
friends}|
|{8, Eve}  |{8, 1, follows}|{1, John}  |{1, 6, friends}|{6, Alice}|{6, 8,
follows}|
```

```
|{3, Steve}|{3, 5, follows}|{5, Paul} |{5, 1, follows}|{1, John} |{1, 3, follows}|

|{6, Alice}|{6, 7, friends}|{7, Bob} |{7, 9, follows}|{9, Mike} |{9, 6, friends}|

|{1, John} |{1, 6, friends}|{6, Alice}|{6, 8, follows}|{8, Eve} |{8, 1, friends}|

|{1, John} |{1, 3, follows}|{3, Steve}|{3, 4, friends}|{4, Karen}|{4, 1, follows}|

|{7, Bob} |{7, 9, follows}|{9, Mike} |{9, 6, friends}|{6, Alice}|{6, 7, friends}|

|{1, John} |{1, 3, follows}|{3, Steve}|{3, 5, follows}|{5, Paul} |{5, 1, friends}|

|{1, John} |{1, 2, friends}|{2, Mary} |{2, 3, friends}|{3, Steve}|{3, 1, follows}|

|{1, John} |{1, 6, friends}|{6, Alice}|{6, 8, follows}|{8, Eve} |{8, 1, follows}|

|{8, Eve} |{8, 1, friends}|{1, John} |{1, 6, friends}|{6, Alice}|{6, 8, follows}|

|{3, Steve}|{3, 4, friends}|{4, Karen}|{4, 1, follows}|{1, John} |{1, 3, follows}|

|{3, Steve}|{3, 1, follows}|{1, John} |{1, 2, friends}|{2, Mary} |{2, 3, friends}|

|{3, Steve}|{3, 5, follows}|{5, Paul} |{5, 1, friends}|{1, John} |{1, 3, follows}|

+----------+---------------+----------+---------------+----------+--------
_       +
```

2.

```
chain_patterns = g.find("(a)-[e1]->(b); (b)-[e2]->(c)")\
   .filter("e1.relationship = 'follows' and e2.relationship = 'follows'")\
   .dropDuplicates()


chain_patterns.show(truncate=False)
```

output:

```
+----------+---------------+----------+---------------+----------+
|a         |e1             |b         |e2             |c         |
+----------+---------------+----------+---------------+----------+
|{5, Paul} |{5, 1, follows}|{1, John} |{1, 3, follows}|{3, Steve}|
|{9, Mike} |{9, 7, follows}|{7, Bob}  |{7, 9, follows}|{9, Mike} |
|{4, Karen}|{4, 1, follows}|{1, John} |{1, 3, follows}|{3, Steve}|
|{6, Alice}|{6, 8, follows}|{8, Eve}  |{8, 1, follows}|{1, John} |
|{3, Steve}|{3, 1, follows}|{1, John} |{1, 3, follows}|{3, Steve}|
|{1, John} |{1, 3, follows}|{3, Steve}|{3, 1, follows}|{1, John} |
|{8, Eve}  |{8, 1, follows}|{1, John} |{1, 3, follows}|{3, Steve}|
|{2, Mary} |{2, 4, follows}|{4, Karen}|{4, 1, follows}|{1, John} |
|{3, Steve}|{3, 5, follows}|{5, Paul} |{5, 1, follows}|{1, John} |
|{3, Steve}|{3, 5, follows}|{5, Paul} |{5, 2, follows}|{2, Mary} |
|{7, Bob}  |{7, 9, follows}|{9, Mike} |{9, 7, follows}|{7, Bob}  |
|{1, John} |{1, 3, follows}|{3, Steve}|{3, 5, follows}|{5, Paul} |
|{5, Paul} |{5, 2, follows}|{2, Mary} |{2, 4, follows}|{4, Karen}|
+----------+---------------+----------+---------------+----------+
```

3.

```
from pyspark.sql.functions import col

follows_edges = edges.filter(col('relationship') == 'follows')

one_way_follow = follows_edges.alias('e1').join(
    follows_edges.alias('e2'),
    (col('e1.src') == col('e2.dst')) & (col('e1.dst') == col('e2.src')) & (col('e2.relationship') ==
    'follows'), 'left_anti'
).select('e1.src', 'e1.dst', 'e1.relationship')

one_way_follow.show(truncate=False)
```

output:

```
+---+---+------------+
|src|dst|relationship|
+---+---+------------ +
|2  |4  |follows     |
|3  |5  |follows     |
|4  |1  |follows     |
|5  |2  |follows     |
|5  |1  |follows     |
|6  |8  |follows     |
|8  |1  |follows     |
+---+---+------------ +
```

4.

```
from pyspark.sql.functions import col, desc

follows_edges = edges.filter(col('relationship') == 'follows')

followers_count = follows_edges.groupBy('dst').count()

max_followers = followers_count.agg({'count': 'max'}).collect()[0][0]

users_with_max_followers = followers_count.filter(col('count') == max_followers)

users_with_max_followers.show(truncate=False)
```

output:

```
+---+-----+
|dst|count|
+---+----- +
|1  |4    |
+---+----- +
```

5.

```
from pyspark.sql.functions import col

friends_edges = edges.filter(col('relationship') == 'friends')

friends_count_src = friends_edges.groupBy('src').count().withColumnRenamed('count', 'friends_count')

friends_count_dst = friends_edges.groupBy('dst').count().withColumnRenamed('count', 'friends_count')

friends_count = friends_count_src.union(friends_count_dst)\
   .groupBy('src').agg({'friends_count': 'sum'}).withColumnRenamed('sum(friends_count)', 'total_friends')\
   .withColumnRenamed('src', 'id')

min_friends = friends_count.agg({'total_friends': 'min'}).collect()[0][0]

users_with_min_friends = friends_count.filter(col('total_friends') == min_friends)

users_with_min_friends.show(truncate=False)
```

output:

```
+---+-------------+
|id |total_friends|
+---+-------------+
|9  |1            |
+---+-------------+
```

6.

```
result = g.labelPropagation(maxIter=5)

result.select('id', 'label').show(truncate=False)
```

```
+---+-----+
|id |label|
+---+-----+
|4  |8    |
|1  |8    |
|6  |8    |
|3  |8    |
|7  |8    |
|9  |7    |
|8  |1    |
|5  |8    |
|2  |8    |
+---+-----+
```