Big Data Hadoop and Spark Developer

Spark GraphX

Learning Objectives

By the end of this lesson, you will be able to:

- Recognize Spark GraphX
- Work with different algorithms of Spark GraphX
- Identify Spark GraphFrames
- Examine the PageRank algorithm with social media data

DATA AND
ARTIFICIAL INTELLIGENCE

Introduction to Graphs

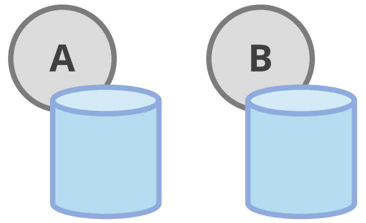
simplilearn

Graph

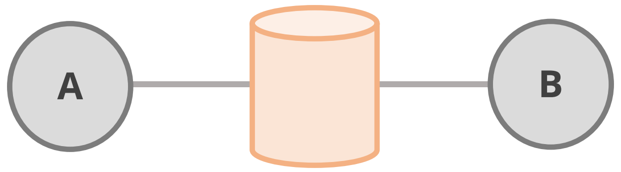
- A graph is a set of points that are interconnected by lines.
- The set of points are called vertices and the interconnecting lines are called edges.

Graph: Example

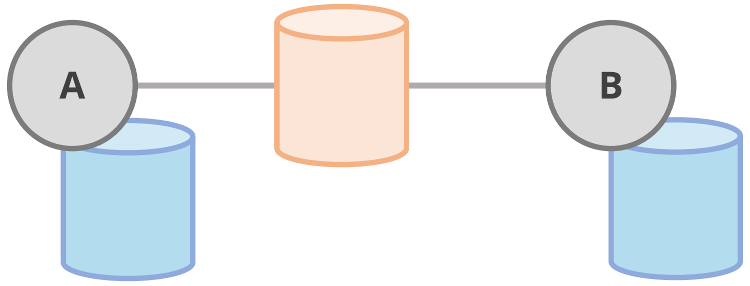
The components of a graph are explained with an example below:



Vertices: The two nodes are called vertices.



Edges: The lines that connect the two vertices are called edges.



Triplets: A triplet contains information about both the vertices and the edges.

Use Cases of GraphX

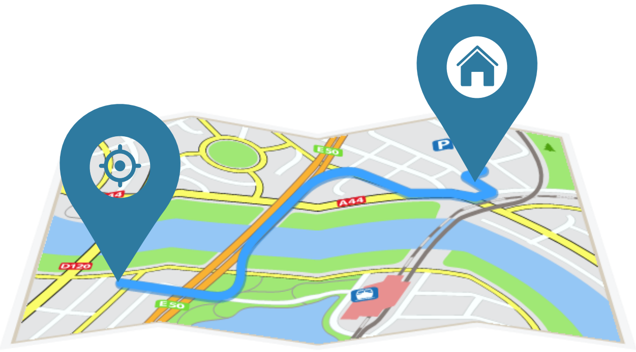
GraphX: Use Case

Fraud detection system

Page rank

Disaster detection system

Business analysis



Geographic information system

Google pregel

Use Case of GraphX

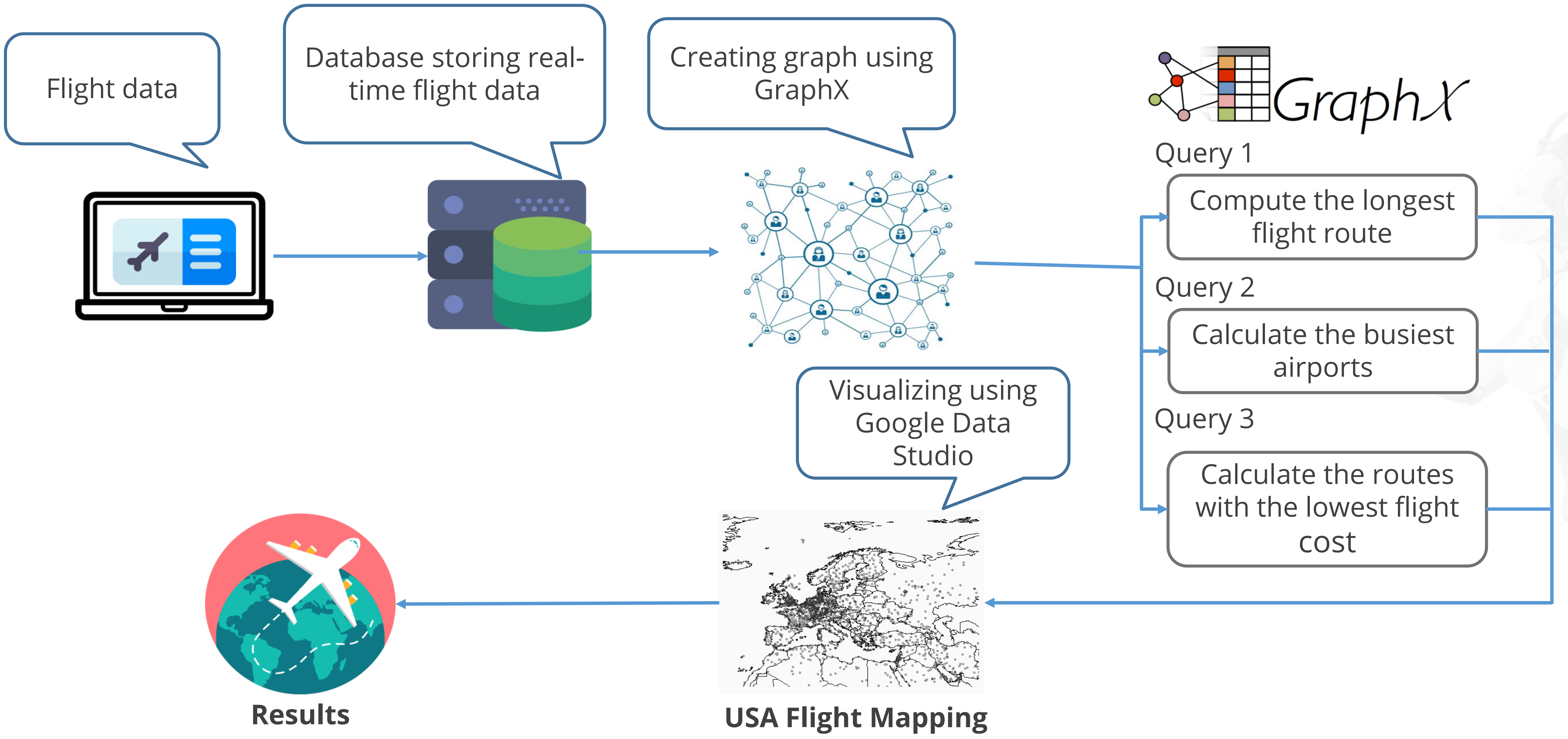
Problem

Flight data analysis using Spark

A data analyst wants to analyze the real-time data of flight using Spark GraphX to provide computation results and visualize them.

Use Case of GraphX

The following diagram illustrates the use of GraphX in fetching flight details:



Types of Graph

There are eight types of graphs:

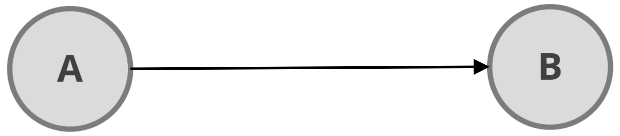
- 01 Undirected graph
- 02 Directed graph
- 03 Vertex labeled graph
- 04 Edge labeled graph
- 05 Cyclic graph
- 06 Weighted graph
- 07 Directed acyclic graph
- 08 Disconnected graph

Types of Graph

01

Undirected graph:

- The edges of an undirected graph are bidirectional and have no orientation.
- The graph can be traversed from node A to node B and vice versa.



02

Directed graph:

- A directed graph is made up of a set of vertices (nodes) connected by edges, each with its direction.
- The graph can be traversed from vertex A to vertex B, but not the other way around.

Types of Graph

03

Vertex labeled graph:

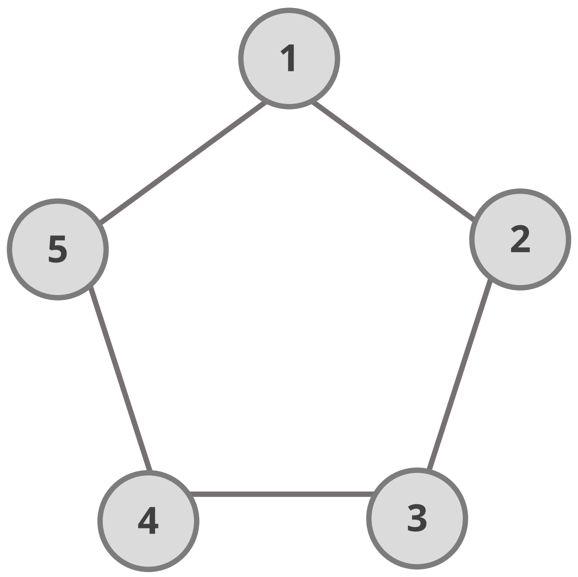
- Vertex labeling is a function that is applied to a graph such a function is known as a vertex labeled graph.
- The vertices are labeled.

04

Edge labeled graph:

- Edge labeling is a function that is applied to a graph such a function is known as an edge labeled graph.
- The edges are labeled.

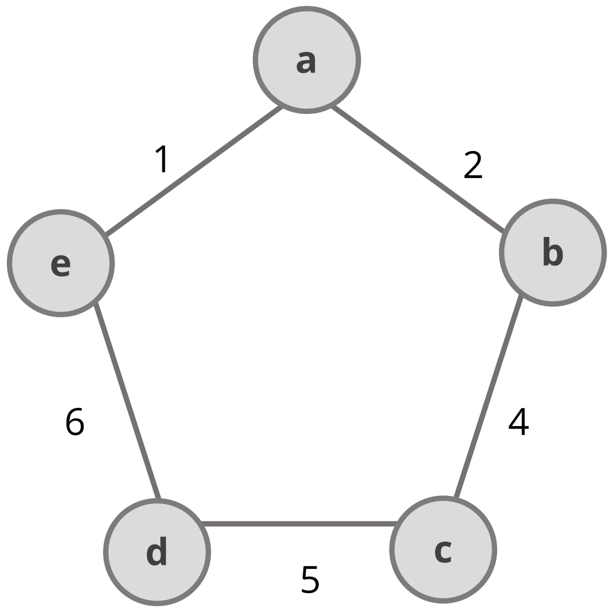
Types of Graph



05

Cyclic graph:

A cyclic graph contains a cycle.

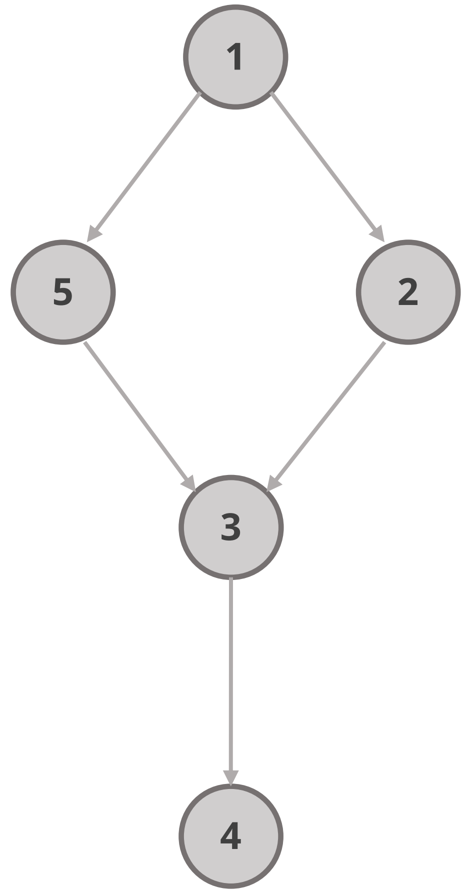


06

Weighted graph:

A weighted graph is a graph in which each branch is given a numerical weight.

Types of Graph



07

Directed acyclic graph:

It is made up of vertices and edges with each edge pointing from one vertex to the next in such a way the directions would never result in a closed loop.

08

Disconnected graph:

A graph is considered unconnected if at least two of its vertices are not connected by a path.

DATA AND
ARTIFICIAL INTELLIGENCE

Introduction to Spark GraphX

Spark GraphX

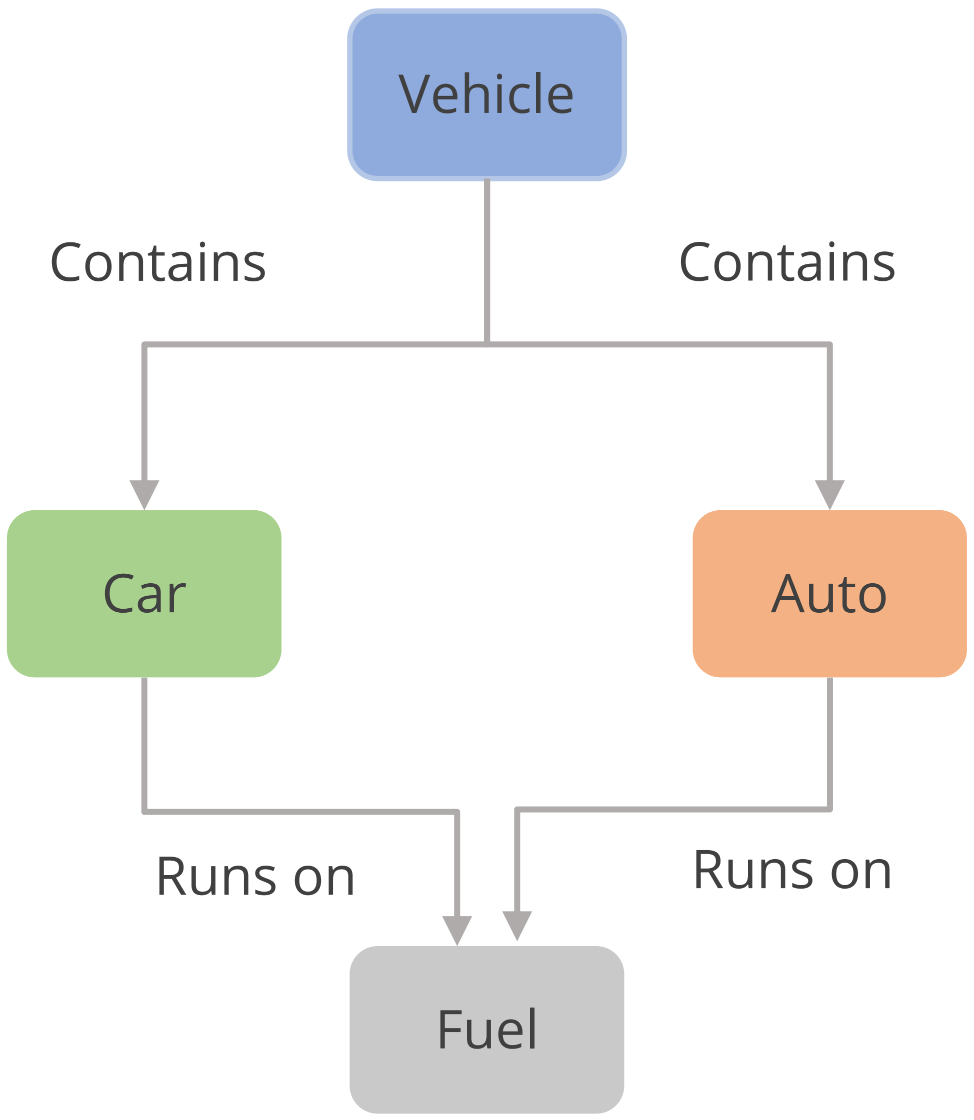
- Spark GraphX is a new component in Spark for graphs and graph-parallel computation.
- It is a graph computation system that runs on a data-parallel system framework.
- It extends the Spark RDD by introducing a new graph abstraction: a directed multigraph with properties attached to each vertex and edge.

Features of Spark GraphX

GraphX provides users with the following features:

- GraphX is a real-time processing framework.
- GraphX extends the RDD abstraction and introduces RDG.
- GraphX simplifies the graph ETL and analysis process substantially.

Property Graph



- A property graph is a directed graph with potentially multiple parallel edges sharing the same source and destination vertex.
- It is a type of graph model where relationships not only are connections but also carry a name (type) and some properties.

Property Graph

The following are the characteristics of the property graph:

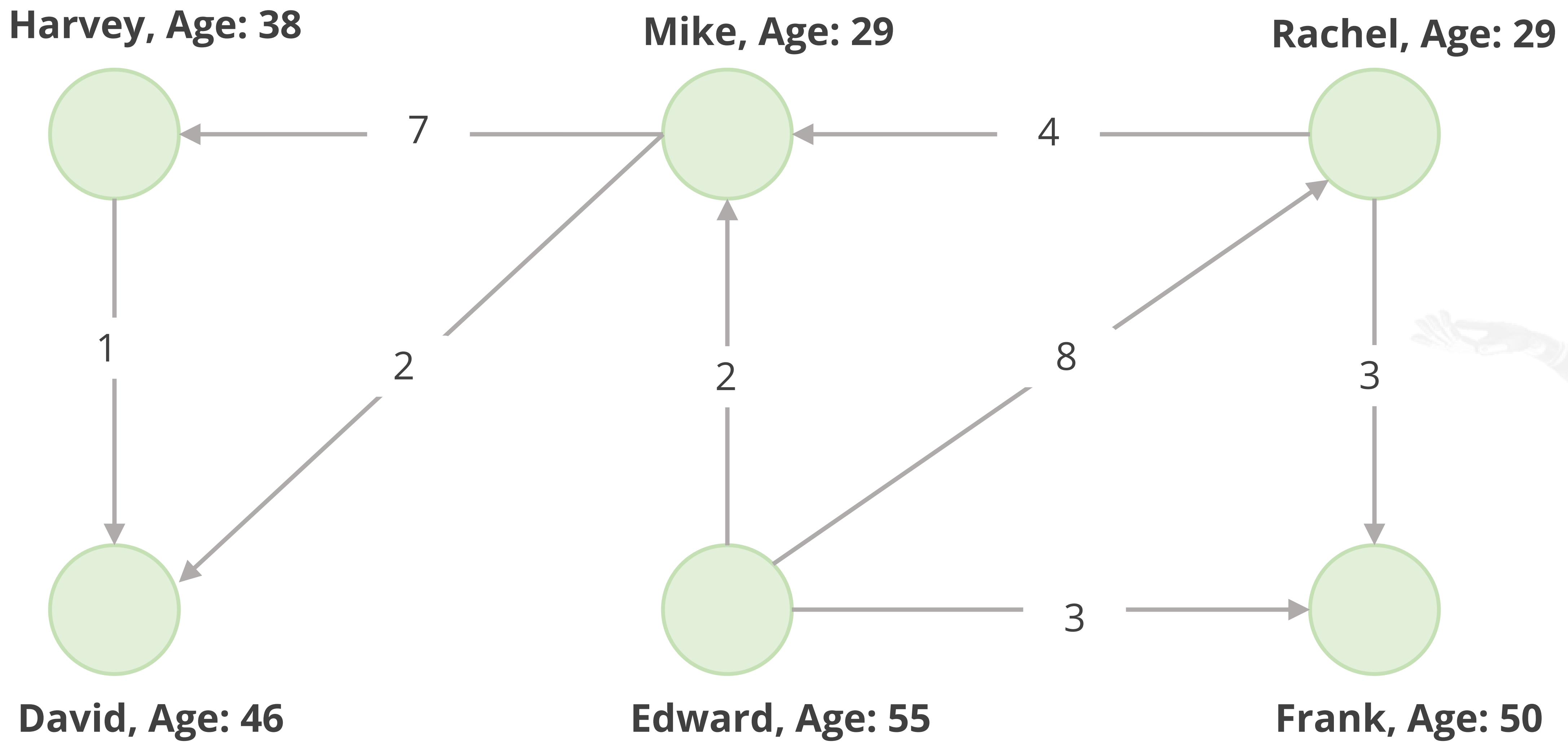
01 Immutable

02 Distributed

03 Fault-tolerant

GraphX: Example

The following graph represents the age of people who are connected with one another:



Implementation of GraphX

Step 1: Import the necessary libraries after logging in to the spark environment

Import libraries

```
//Log in to the spark environment
Command:
spark3-shell

//import the dependencies
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
```

Implementation of GraphX

Step 2: Create a vertex array that contains the ID, name of a person, and age

Vertex Array Creation

```
val vertexArray = Array((1L, ("Harvey", 38)),(2L, ("Mike", 29)),(3L, ("Rachel", 25)),(4L,
("David", 46)),(5L, ("Edward", 55)),(6L, ("Frank",50)))

Output:

vertexArray: Array[(Long, (String, Int))] = Array((1,(Harvey,38)), (2,(Mike,29)),
(3,(Rachel,25)), (4,(David,46)), (5,(Edward,55)), (6,(Frank
,50)))
```

Implementation of GraphX

Step 3: Convert the vertex array to RDD

Vertex Array Creation

```
val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)

Output:

vertexRDD: org.apache.spark.rdd.RDD[(Long, (String, Int))] = ParallelCollectionRDD[16] at
parallelize at <console>:35
```

Implementation of GraphX

Step 4: Create an edge array

Edge Array Creation

```
val edgeArray = Array(Edge(2L, 1L, 7),Edge(2L, 4L, 2),Edge(3L, 2L, 4),Edge(3L, 6L, 3),Edge(4L,
1L, 1),Edge(5L, 2L, 2),Edge(5L, 3L, 8),Edge(5L, 6L, 3))

Output:

edgeArray: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(2,1,7), Edge(2,4,2),
Edge(3,2,4), Edge(3,6,3), Edge(4,1,1), Edge(5,2,2), Edge(5,3,8), Edge(5,6,3))
```

Implementation of GraphX

Step 5: Convert the edge array to RDD

Edge Array Creation

```
val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)

Output:

edgeRDD: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] =
ParallelCollectionRDD[17] at parallelize at <console>:35
```

Implementation of GraphX

Step 6: Create a graph that contains vertices whose age is above 30

Edge Array Creation

```
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")}


Output:
David is 46
Frank is 50
Harvey is 38
Edward is 55
```

Assisted Practice 20.1: Implementation of a Simple GraphX

Duration: 15 mins

Problem Scenario: Create a graph object with five friends from different age groups who are connected through social media

Objective: In this demonstration, you will learn to implement graphX.

Steps to perform:

1. Import the necessary libraries after logging in to the spark environment
2. Create a vertex array that contains the ID, name of the friend, and age
3. Create an edge array
4. Create a graph that contains vertices whose age is below 35
5. Display the data

Note: The solution to this assisted practice is provided under the course resources section.

DATA AND
ARTIFICIAL INTELLIGENCE

GraphX Operators

simplilearn

GraphX Operators

Property graphs are graph models that contain a collection of basic operators. These operators are called GraphX operators. These operators take user-defined functions as input and produce new graphs.

Example: indegree calculation

```
val inDegrees: VertexRDD[Int] = graph.inDegrees

Output:
inDegrees: org.apache.spark.graphx.VertexRDD[Int] = VertexRDDImpl[35] at RDD at
VertexRDD.scala:57
```

GraphX Operators

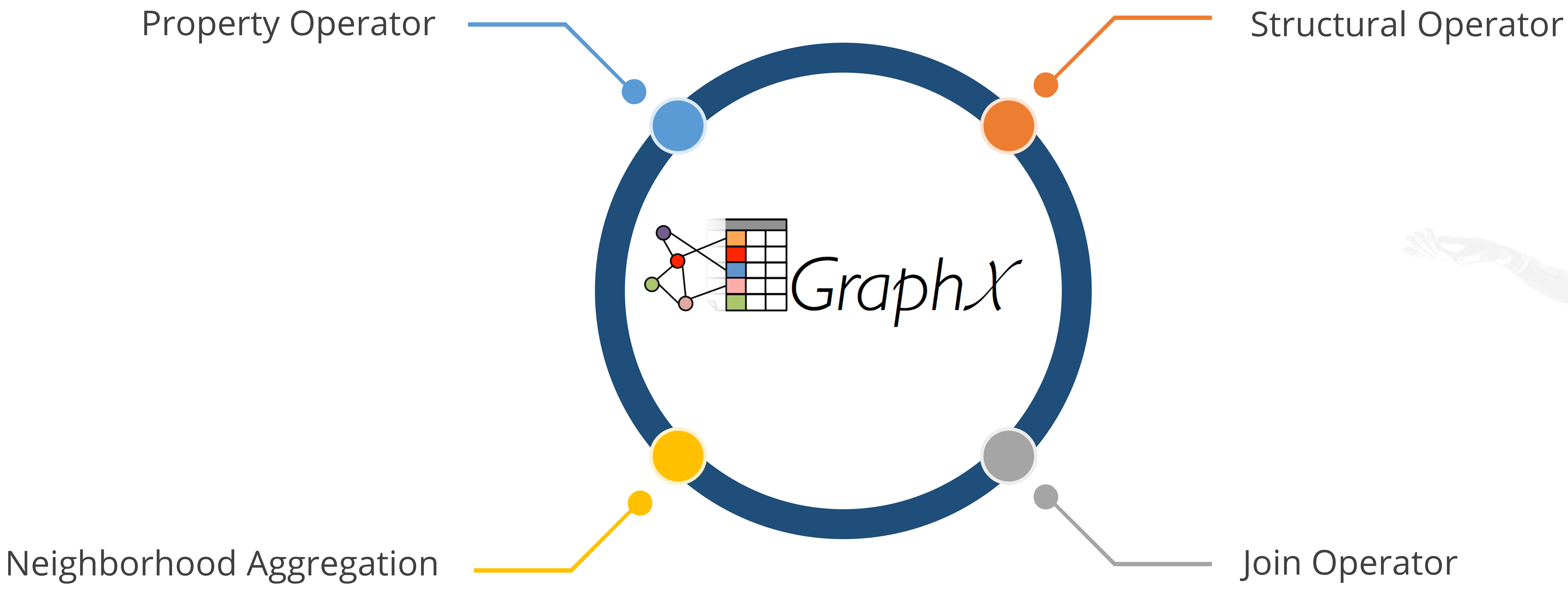
Property graphs have a collection of basic operators. These operators take user-defined functions as the input and produce new graphs.

Example: Property operator

```
class Graph[VD, ED] {
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

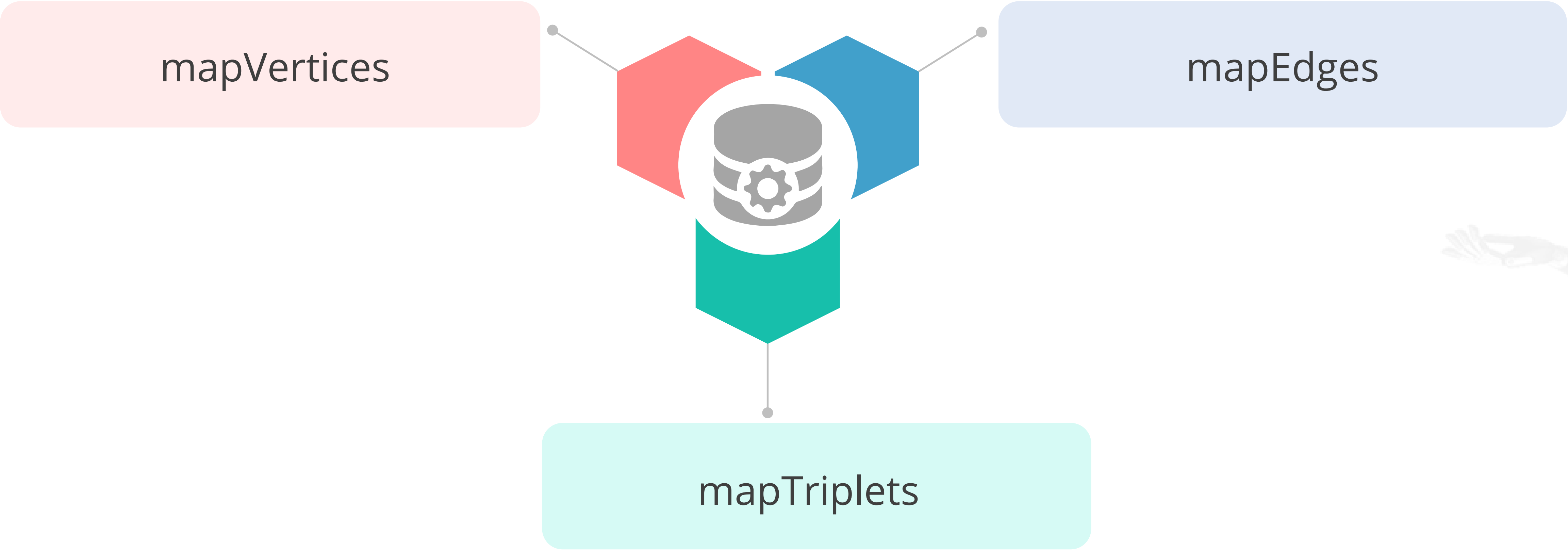
Types of GraphX Operators

The types of GraphX operators are given below:



Property Operator

The property operator contains the following operations:



Property Operator

The following is the syntax of property operators:

Syntax of property operator

```
class Graph[VD, ED]
{
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
}
```

Structural Operators

The following are a few basic structural operators:

Types

01 reverse

02 subgraph

03 mask

04 groupEdges

Structural Operators

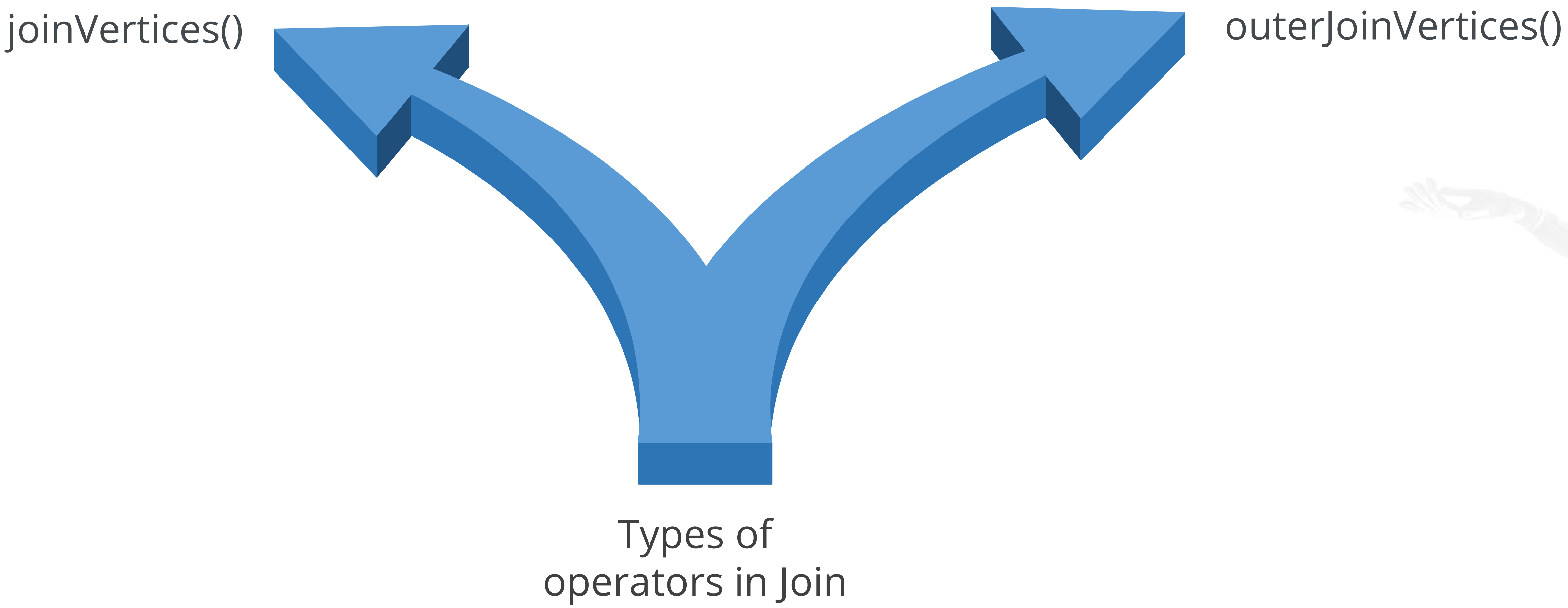
The following is the syntax of structural operators:

Syntax of structural operator

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

Join Operators

The join operators join data from external collections (RDDs) with a graph.



joinVertices Operator

The joinVertices is an operator that joins the vertices with the input RDD and returns a new graph with the vertex properties.

Syntax of joinVertices:

```
val nonUniqueCosts: RDD[(VertexId, Double)]
val uniqueCosts: VertexRDD[Double] =  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a
+ b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

outerJoinVertices Operator

In the outerJoinVertices operator, the user-defined map function is applied to all vertices and can change the vertex property type.

Syntax of outerJoin operator:

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

Neighborhood Aggregation

Neighborhood aggregation is the key task in graph analytics which includes aggregating information about the neighborhood of each vertex.

graph.mapReduceTriplets → graph.AggregateMessages

aggregateMessages is the core aggregation operation in GraphX which applies a user-defined sendMsg function to each edge triplet in the graph.

Neighborhood Aggregation

The following is the syntax of aggregateMessage operator:

Syntax of aggregateMessage operator :

```
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

GraphX: Example

The following steps illustrates the creation of GraphX with an example:

Step1: Import the required packages

Import packages

```
import org.apache.spark.SparkContext
import org.apache.spark.graphx.{Edge, Graph}
import org.apache.spark.sql.SparkSession
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
```

GraphX: Example

Step 2: Create a vertex array that contains the city and population

Vertex array:

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

Vertex array:

```
Output:

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)))
```

GraphX: Example

Step 3: Create an edge array where the first and the second arguments indicate the source and the destination vertices respectively

Edge array:

```
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))
```

GraphX: Example

The output after the creation of the array will be as shown here:

Edge array

```
Output:

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))
```

GraphX: Example

Step 4: Create a spark context

Spark Context:

```
val sc =
SparkSession.builder().master("local[2]").getOrCreate().spar
kContext;
```

Spark Context:

```
Output:

22/05/01 10:30:34 WARN lineage.LineageWriter: Lineage
directory /var/log/spark/lineage doesn't exist or is not
writable. Lineage for this application will be
disabled.22/05/01 10:30:34 WARN sql.SparkSession$Builder:
Using an existing SparkSession; some configuration may not
take effect.sc: org.apache.spark.SparkContext =
org.apache.spark.SparkContext@19cee7ed
```

GraphX: Example

Step 5: Convert the array to RDD

Spark RDD:

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

Spark RDD:

```
Output:
verRDD: org.apache.spark.rdd.RDD[(Long, (String, Int))] =
ParallelCollectionRDD[0] at parallelize at <console>:41

org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]]
= ParallelCollectionRDD[1] at parallelize at <console>:41
```

GraphX: Example

Step 6: Create a property graph which contains RDD of vertices and RDD of edges

Property Graph:

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

Property Graph:

```
Output:
graph: org.apache.spark.graphx.Graph[(String, Int),Int] =
org.apache.spark.graphx.impl.GraphImpl@7f7b15d4
```

GraphX: Example

Step 7: Find the cities with a population of more than 50000
To implement this, use the filter operator

Property Graph:

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

Property Graph:

```
Output:

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

GraphX: Example

Step 8: Calculate the distance between two cities using triplets

Property Graph:

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

Property Graph:

```
Output:
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 Philadelphia and New York is 130 kilometers
The distance between Philadelphia and Scranton is 168 kilometers
The distance between Harrisburg and Scranton is 162 kilometers
The distance between Wilmington and Philadelphia is 39 kilometers
The distance between New York and Scranton is 159 kilometers
```

GraphX: Example

Step 9: Perform filtration based on the edges

Property Graph:

```
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")

  }
```

Property Graph:

```
Output:
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 1 and 5 is 130
The distance between 4 and 1 is 39
```

GraphX: Example

Step 10: Calculate the total population of the neighboring cities

Reversed property graph:

```
val undirectedEdgeRDD =
graph.reverse.edges.union(graph.edges)
val graph1 = Graph(verRDD, undirectedEdgeRDD)
```

Note

The current GraphX in this example deals only with directed graphs. But in this case, consider edges in both directions and add the reverse directions to the graph.

GraphX: Example

Reversed property graph:

```
val neighbors = graph1.aggregateMessages[Int](ectx =>
ectx.sendToSrc(ectx.dstAttr._2), _ + _)

neighbors.foreach(println(_))
```

Step 11:

- The directed graph is converted to an undirected graph with all the edges and directions considered
- Perform the aggregation using the aggregate message operator

DATA AND
ARTIFICIAL INTELLIGENCE

Graph-Parallel System

Graph-Parallel System

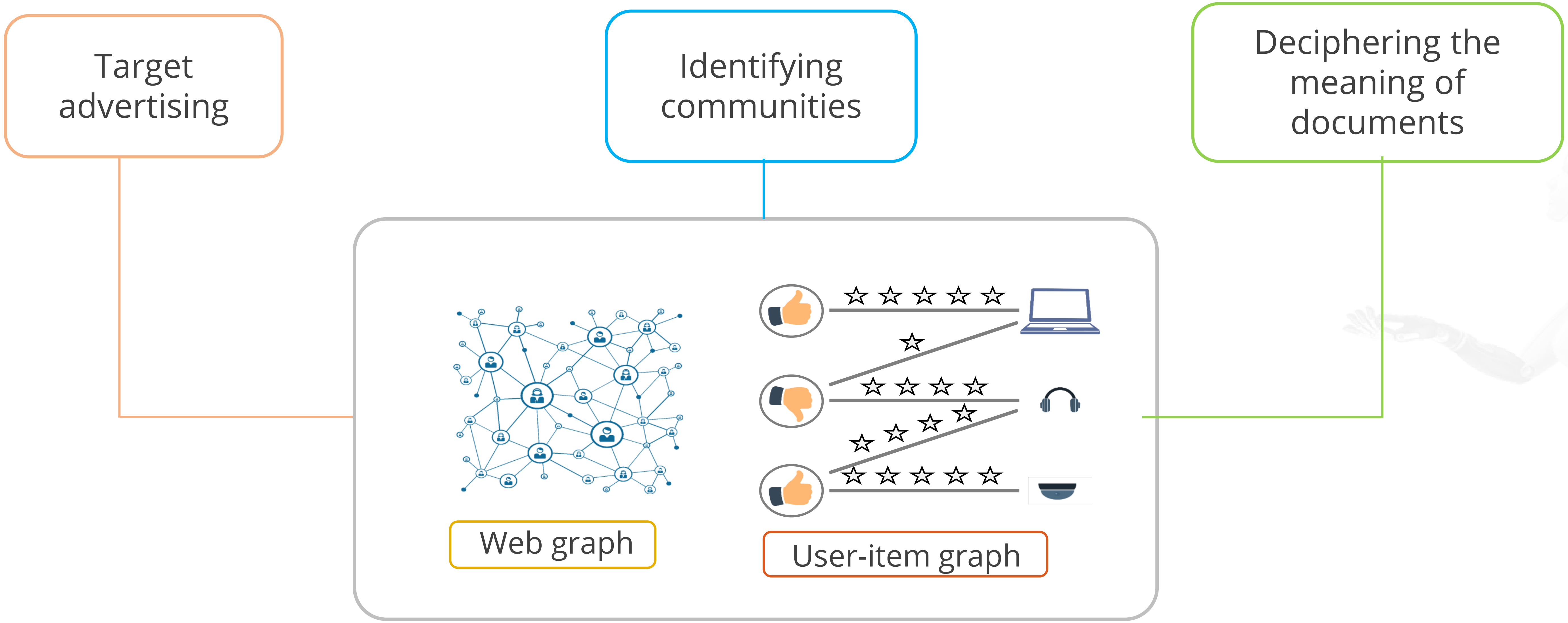
Parallel graph processing refers to the use of multiple cores to process a graph.

Web graph

User-item graph

Data Exploding Using Graphs

The various graphs can be used to extract meaningful information from data.



Limitations of Graph-Parallel System

1. Each graph-parallel system framework represents a different graph computation.
2. These frameworks depend on different runtimes.
3. These frameworks cannot resolve the data ETL and cannot decipher process issues.

DATA AND
ARTIFICIAL INTELLIGENCE

Algorithms in Spark

PageRank Algorithm

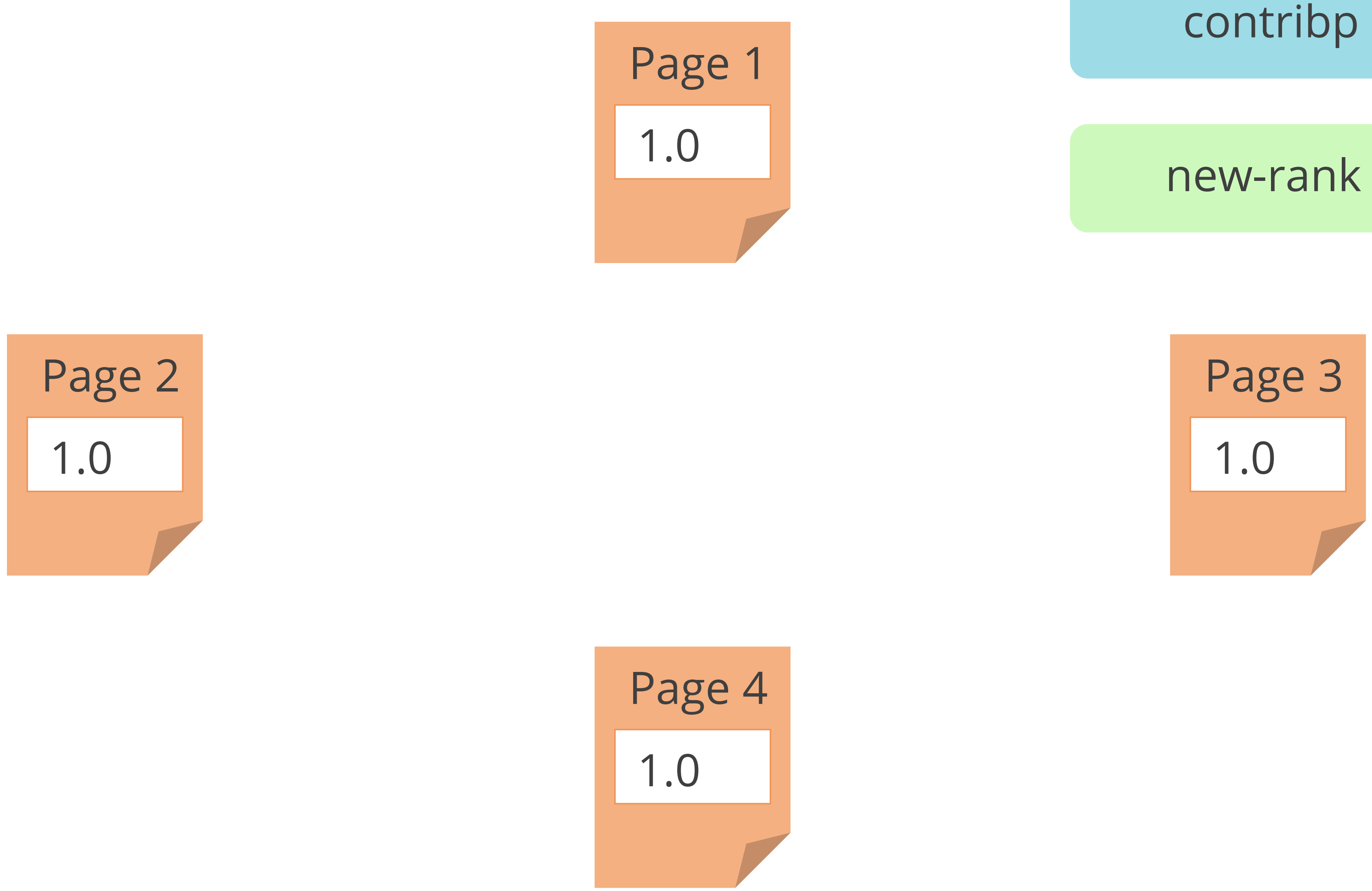
- It is an iterative algorithm.
- It is used to determine the relevance or importance of a webpage.
- It gives web pages a ranking score.
- It outputs a probability distribution.

PageRank Algorithm

In each iteration, a page contributes to its neighbors its rank, divided by the number of its neighbors.

contribp = rankp / neighborsp

new-rank = Σcontribs * .85 + .15



PageRank with Social Media Network

GraphX includes a social network dataset to run the PageRank algorithm.

Page rank algorithm:

```
import org.apache.spark.graphx.GraphLoader
```

Step 1: Download the dataset and upload it to the hdfs on the Simplilearn lab

Step 2: Log in to the **Web console** and enter the spark environment

Step 3: Import the necessary libraries

PageRank with Social Media Network

Step 4: Load the graph from an edge list formatted file where each line contains two integers

.

Page rank algorithm:

```
val graph = GraphLoader.edgeListFile(sc,
“/user/simplilearnuser/data/followers.txt")
```

Step 5: Run the pageRank

Page rank algorithm:

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

PageRank with Social Media Network

Step 6: Join the ranks with the usernames

Page rank algorithm:

```
val users = sc.textFile(" user/simplilearnuser/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}

val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
```

PageRank with Social Media Network

Step 7: Print the result

Page rank algorithm:

```
println(ranksByUsername.collect().mkString("\n"))

Output:
(justinbieber,0.15007622780470478)
(matei_zaharia,0.7017164142469724)
(ladygaga,1.3907556008752426)
(BarackObama,1.4596227918476916)
(odersky,1.2979769092759237)
(jeresig,0.9998520559494657)
```

Connected Components

The connected component is an algorithm that labels each connected component of the graph.

Connected component algorithm:

```
import org.apache.spark.graphx.GraphLoader
```

Step 1: Download the dataset and upload it to the hdfs on the Simplilearn lab

Step 2: Log in to the **Web console** and enter the spark environment

Step 3: Import the necessary libraries

Connected Components

Step 4: Load the graph from an edge list formatted file where each line contains two integers

.

Connected component algorithm:

```
val graph = GraphLoader.edgeListFile(sc,
"/user/simplilearnuser/data/followers.txt")
```

Step 5: Find the connected components.

Connected component algorithm:

```
val cc = graph.connectedComponents().vertices
```


simplilearn

Connected Components

Step 6: Join the connected components with the usernames

Connected component algorithm:

```
val users =
sc.textFile("/user/bhavanavasudevsimplilearn/data1/data/users.txt").map {
line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
```

Connected Components

Step 7: Print the result

Connected component algorithm:

```
println(ccByUsername.collect().mkString("\n"))

Output:
(justinbieber,1)
(matei_zaharia,3)
(ladygaga,1)
(BarackObama,1)
(jeresig,3)
(odersky,3)
```

Triangle Counting

Triangle counting is an algorithm that determines the number of triangles passing through each vertex, providing a measure of clustering.

Triangle counting algorithm:

```
import org.apache.spark.graphx.{GraphLoader,
PartitionStrategy}
```

Step 1: Download the dataset and upload it to the hdfs on the Simplilearn lab

Step 2: Log in to the **Web console** and enter the spark environment

Step 3: Import the necessary libraries

Triangle Counting

Step 4: Load the edges in canonical order and partition the graph for the triangle count

Triangle counting algorithm:

```
val graph = GraphLoader.edgeListFile(sc,
“/data/simplilearnuser/data/followers.txt",
true).partitionBy(PartitionStrategy.RandomVertexCut)
```

Step 5: Find the triangle count for each vertex

Triangle counting algorithm:

```
val triCounts = graph.triangleCount().vertices
```

Triangle Counting

Step 6: Join the triangle counts with the usernames

Connected component algorithm:

```
val users =
sc.textFile("/user/simplilearnuserdata/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map {case (id,
(username, tc)) =>
  (username, tc)
}
```

Triangle Counting

Step 7: Print the result

Triangle counting algorithm:

```
println(triCountByUsername.collect().mkString("\n"))

Output:
((justinbieber,0)
(matei_zaharia,1)
(ladygaga,0)
(BarackObama,0)
(odersky,1)
(jeresig,1)
```

DATA AND
ARTIFICIAL INTELLIGENCE

Pregel API

simplilearn

Pregel API

Pregel API is used for developing any vertex-centric algorithm.

Vertex program	It takes a message list as input and has access to the current state of the vertex attribute and vertex id.
Send message program	It takes the triplet view as the input with all the attributes materialized.
Merge message program	It takes two messages meant for the same vertex and combines them into one message.

Pregel API

Pregel API requires the following parameters:

Initial Message

The initial message to start the computation

Max Iteration

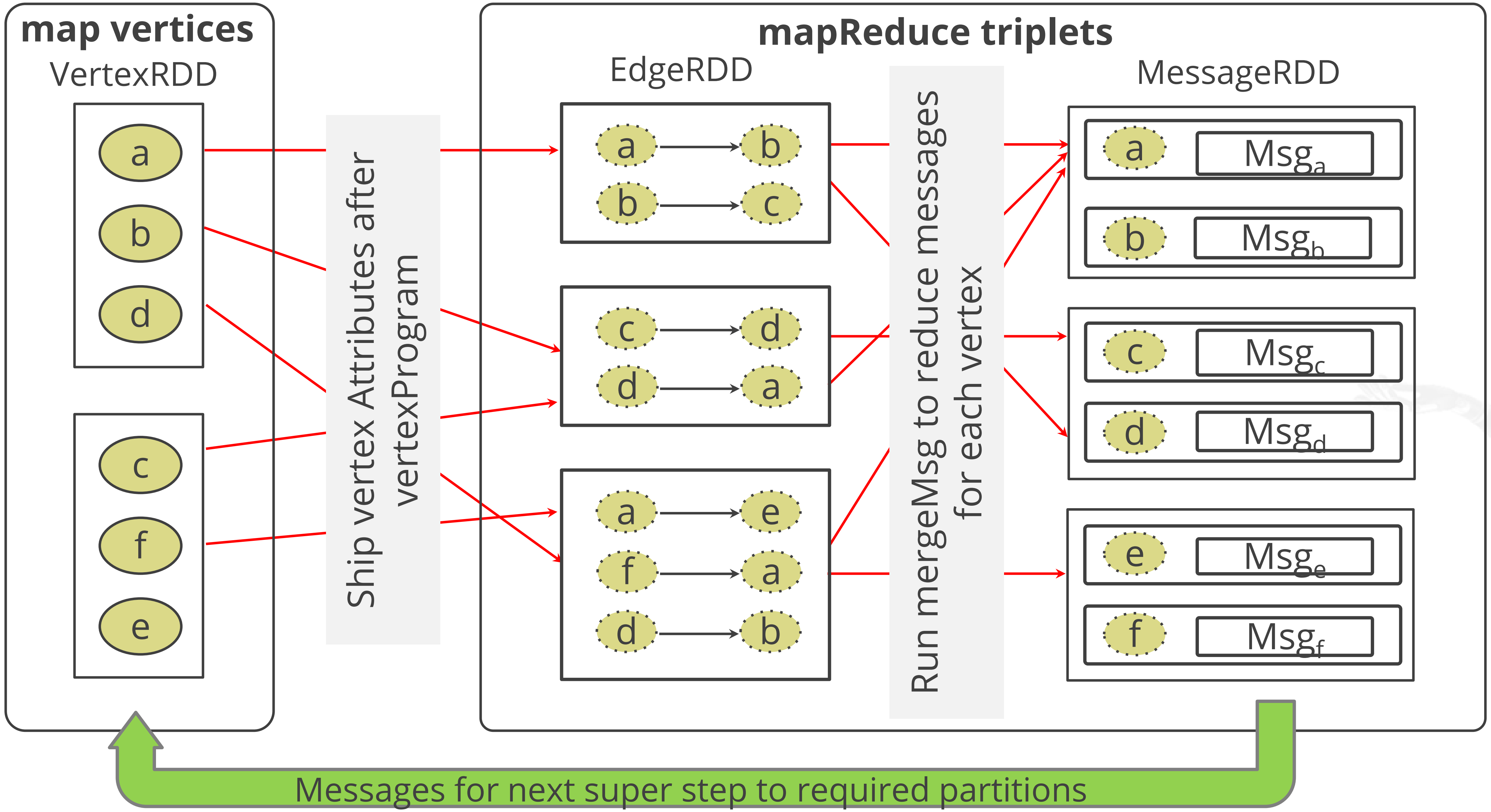
The max number of super steps for the Pregel API

Edge Direction

To filter the edges on which send message function will run

Pregel API

The architecture of Pregel API is shown below:



DATA AND
ARTIFICIAL INTELLIGENCE

GraphFrames

GraphFrames

- Databricks released GraphFrames which is a graph processing library for Apache Spark.
- It is a built-in collaboration with UC Berkeley and MIT
- Graph library is based on DataFrames.
- GraphFrames provides scalability and very high performance.
- It provides a uniform API for graph processing in Scala, Java, and Python.

GraphFrames: Advantages

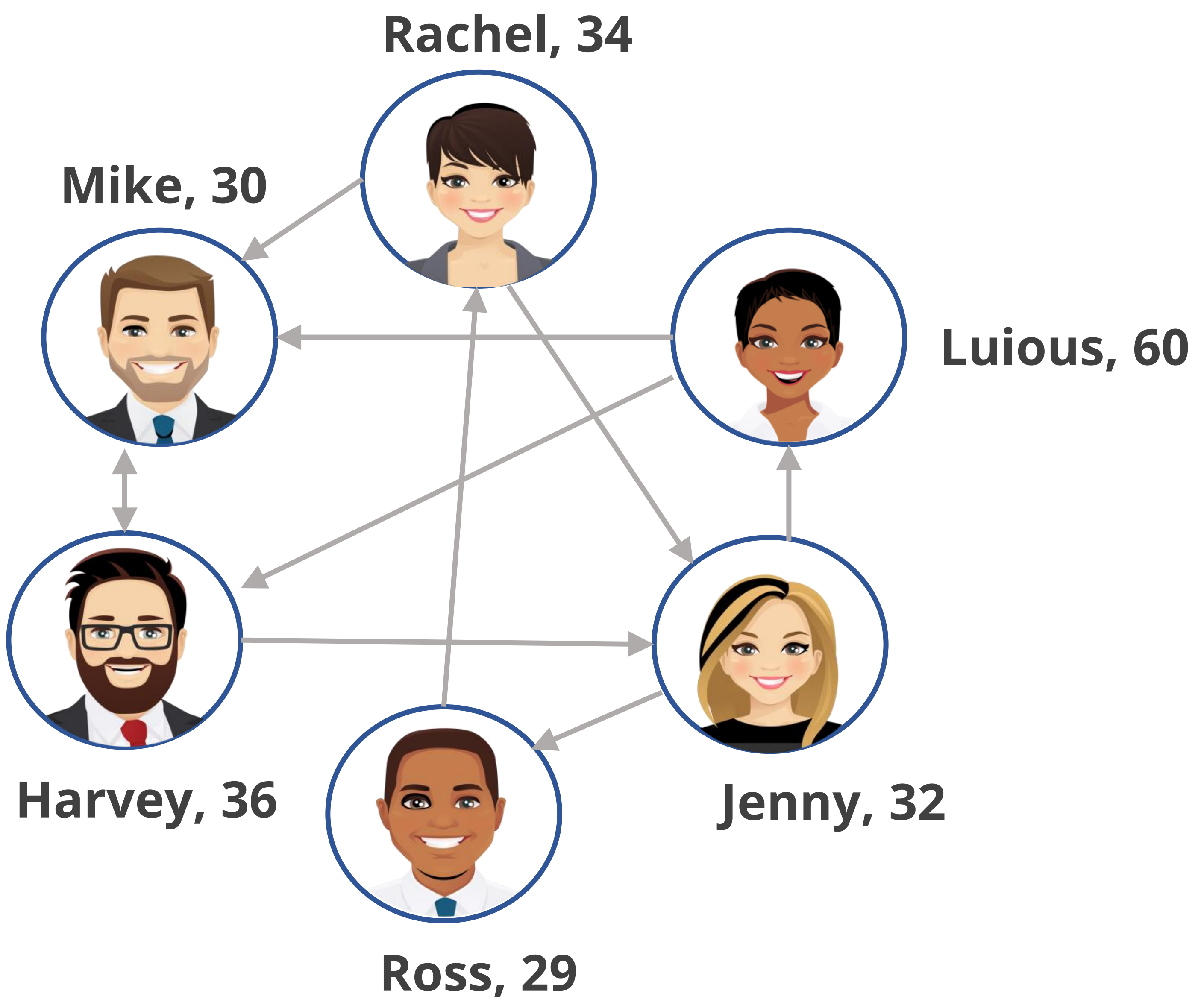
GraphFrames support general graph processing, similar to Apache Spark's GraphX library. GraphFrames are built on DataFrames and have some key advantages:

Advantages

1. It provides uniform APIs for three languages: Java, Python, and Scala.
2. It fully supports the writing and reading of graphs using formats such as Parquet, JSON, and CSV.
3. Users can query in GraphFrames using the familiar APIs of Spark SQL and DataFrames.
4. Its vertices and edges are represented as DataFrames.

GraphFrames: Example

The network is represented as a graph, which contains a set of vertices (users) and edges (connections between users.)



Implementation of GraphFrames

Step 1: Import the necessary libraries after logging in to the spark environment

Libraries

```
Command:
pyspark --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11
```

Implementation of GraphFrames

Step 2: Create a vertices data frame

Vertices Dataframe

```
v = sqlContext.createDataFrame([
  ("a", "Rachel", 34),
  ("b", "Harvey", 36),
  ("c", "Mike", 30),
  ("d", "Ross", 29),
  ("e", "Jenny", 32),
  ("f", "Luious", 60),
], ["id", "name", "age"])
```

Implementation of GraphFrames

Step 3: Create edges DataFrame

Edges Dataframe

```
e = sqlContext.createDataFrame([
  ("a", "b", "friend"),
  ("b", "c", "follow"),
  ("c", "b", "follow"),
  ("f", "c", "follow"),
  ("e", "f", "follow"),
  ("e", "d", "friend"),
  ("d", "a", "friend"),
], ["src", "dst", "relationship"])
```

Implementation of GraphFrames

Step 4: Create a GraphFrame

Edges Dataframe

```
g = GraphFrame(v, e)
```

Implementation of GraphFrames

Step 5: Calculate how many users in the social network have an “age” > 35

Edges Dataframe

```
g.vertices.filter("age > 35")
```

Implementation of GraphFrames

Step 6: Calculate how many users have at least 2 followers?

Edges Dataframe

```
g.inDegrees.filter("inDegree >= 2")
```

Assisted Practice 20.2: GraphX

Duration: 15 mins

Problem Scenario: Create a graph object to calculate the distance between different cities using GraphX

Objective: In this demonstration, Sam uses “GraphX” to solve a real-world social media problem by calculating the distance between the places.

Tasks to perform:

1. Open the Spark shell in “Webconsole” by typing a command and start importing packages.
2. By specifying the paths to the files that were uploaded, you should read the “vertices” and “edges” data.
3. Create a graph object from the “vertices” and “edges” array and calculate the triplets and the distance between the cities and then display the output

Assisted Practice 20.2: GraphX

Duration: 15 mins

Steps to Perform:

Step 1: Download the text files with the names “vertices” and ” edges” from the course resources section

Step 2: Log in to your practice labs on the LMS account

Step 3: Click on “Hue” and click on the “Auth Url” to upload the files and copy the “Username” and the “Password” provided to log in to the “Hue”

Step 4: Click on the “HDFS” icon and click on the “+” symbol to upload the dataset

Step 5: In “Web console,” type the command below to open the “spark-shell”

Step 6: Import required packages

Step 7: By specifying the paths to the files that were uploaded, you should read the “vertices” and “edges” data

Step 8: Create a graph object from the “vertices” and “edges” array and calculate the triplets and the distance between the cities and then display the output

Note: The solution to this assisted practice is provided under the course resources section.

Key Takeaways

- A graph is a set of points that are interconnected by lines.
- The set of points are called vertices and the interconnecting lines are called edges.
- GraphX is a graph computation system that runs on a data-parallel system framework.
- A property graph is a type of graph model where relationships are not only connections but also carry a name (type) and some properties.

DATA AND ARTIFICIAL INTELLIGENCE

Knowledge Check

Knowledge Check
1

Which of the following is a part of a graph?

A. Edges

B. Vertices

C. Triplets

D. All of the above

Knowledge Check
1

Which of the following is a part of a graph?

A. Edges

B. Vertices

C. Triplets

D. All of the above

The correct answer is **D.**

Edges, vertices, and triplets are parts of a graph.

simplilearn

Knowledge Check

2

Which of the following operators joins the vertices with the input RDD and returns a new graph with the vertex properties?

A. joinVertices()

B. outerJoinVertices()

C. Both a and b

D. None of the above

Knowledge Check 2

Which of the following operators joins the vertices with the input RDD and returns a new graph with the vertex properties?

A. joinVertices()

B. outerJoinVertices()

C. Both A and B

D. None of the above

The correct answer is **A.**

joinVertices() joins the vertices with the input RDD and returns a new graph with the vertex properties.

Knowledge Check

3

Which of the following structural operator constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph?

A. reverse

B. subgraph

C. groupEdges

D. mask

Knowledge Check 3

Which of the following structural operator constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph?

A. reverse

B. subgraph

C. groupEdges

D. mask

The correct answer is **D.**

mask operator constructs a subgraph by returning a graph that contains the vertices and edges that are also found in the input graph.

Thank You