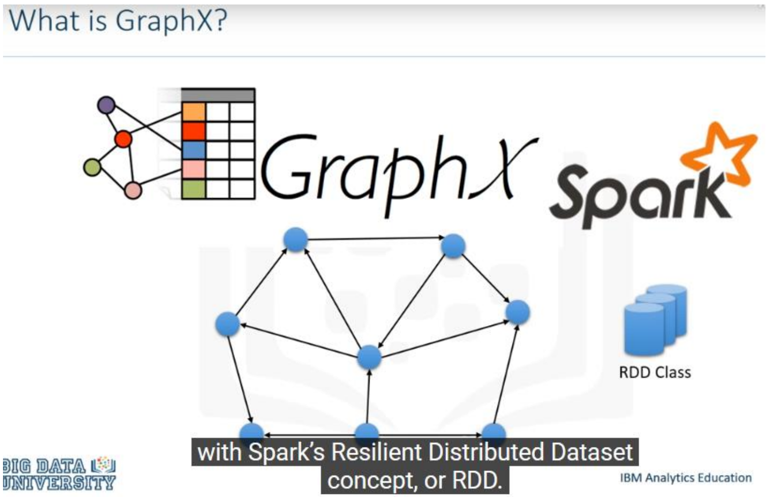
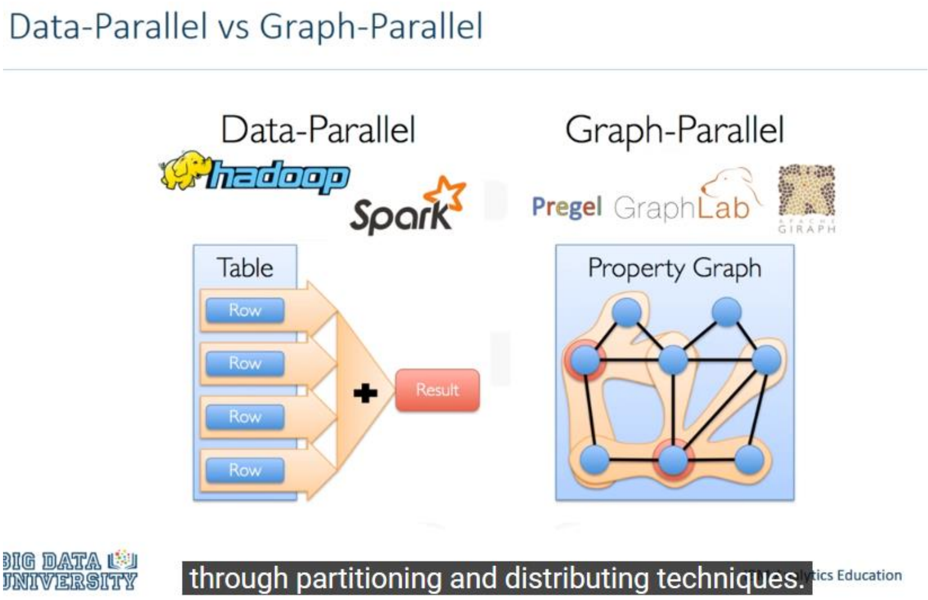
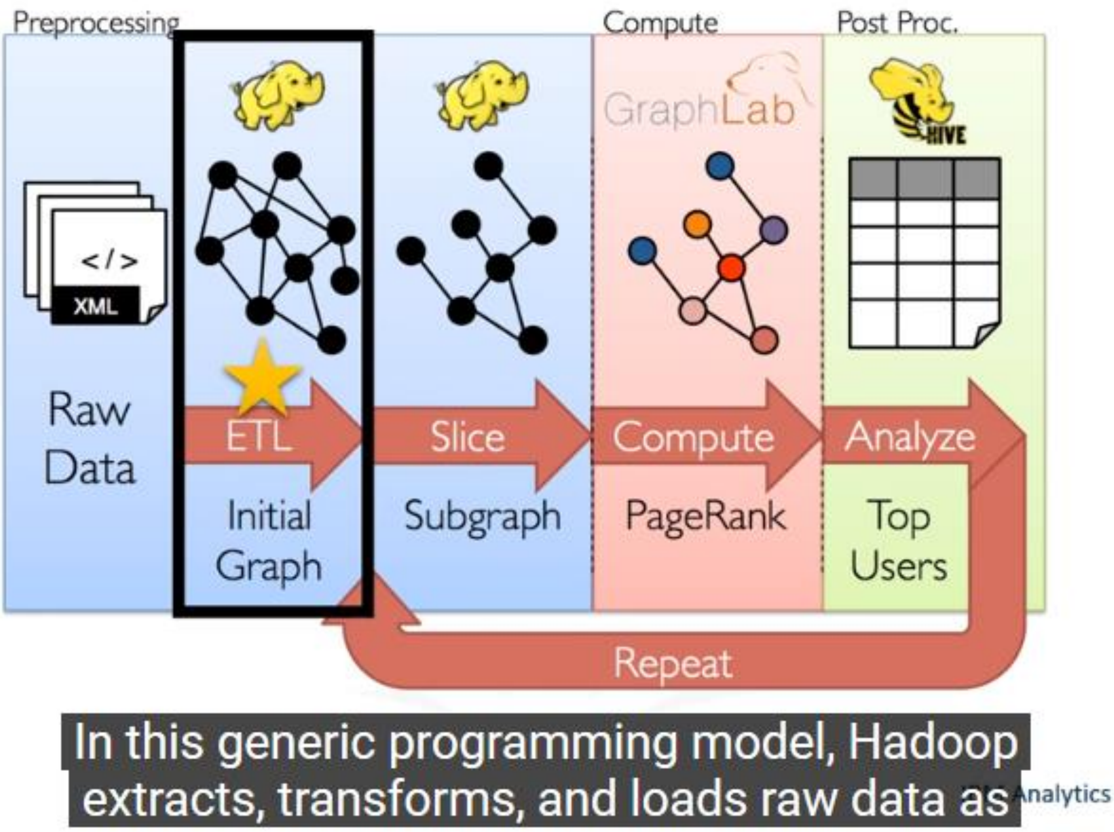
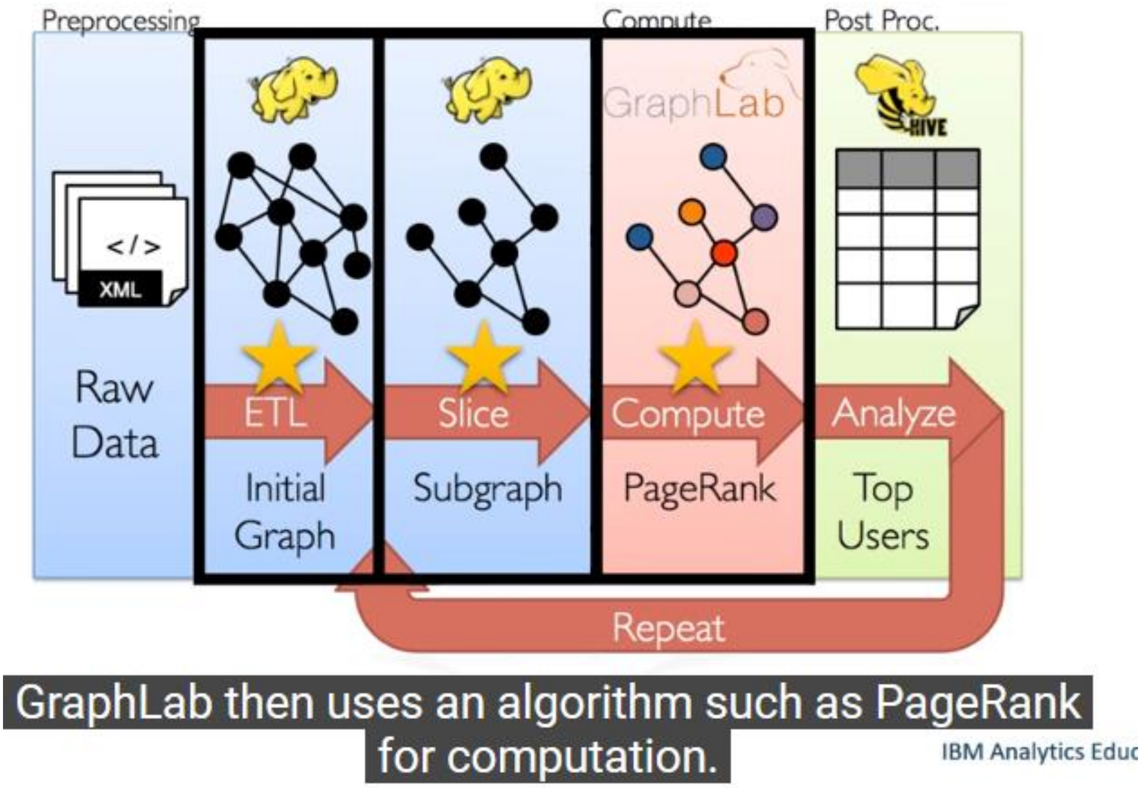
MODULE 1 INTRODUCTION TO GRAPH-PARALLEL



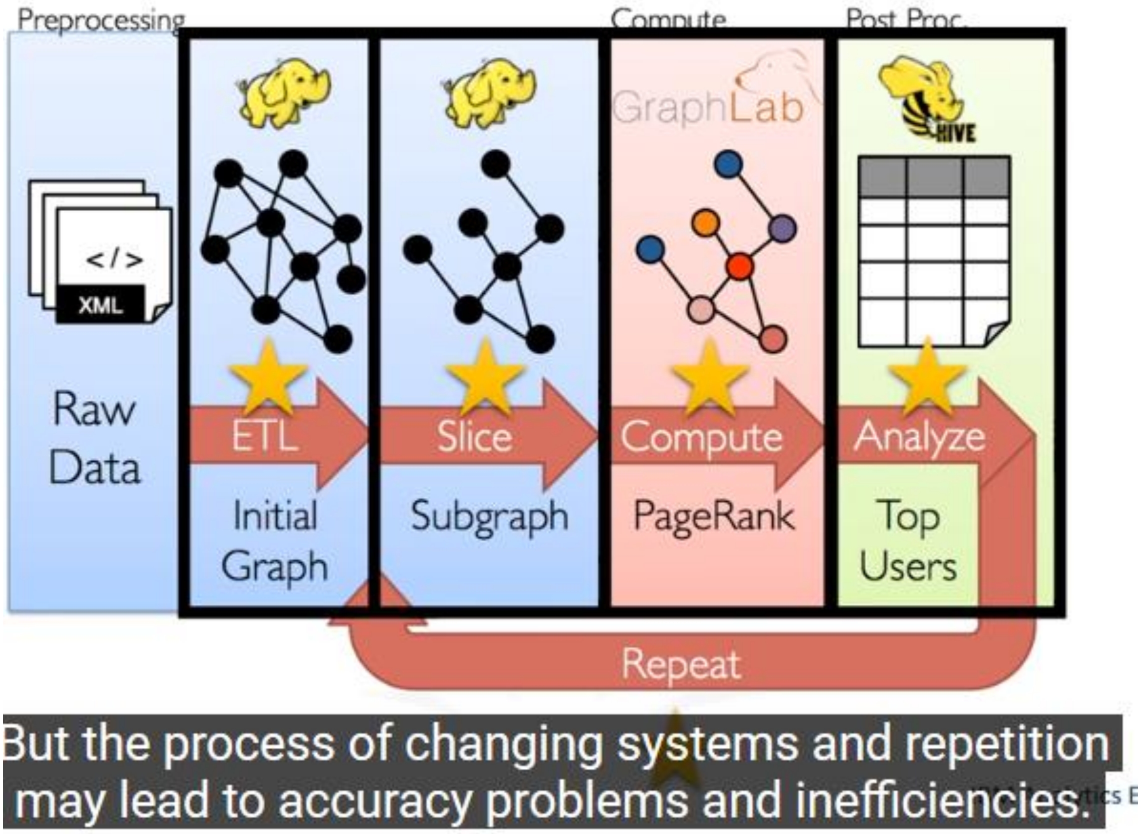


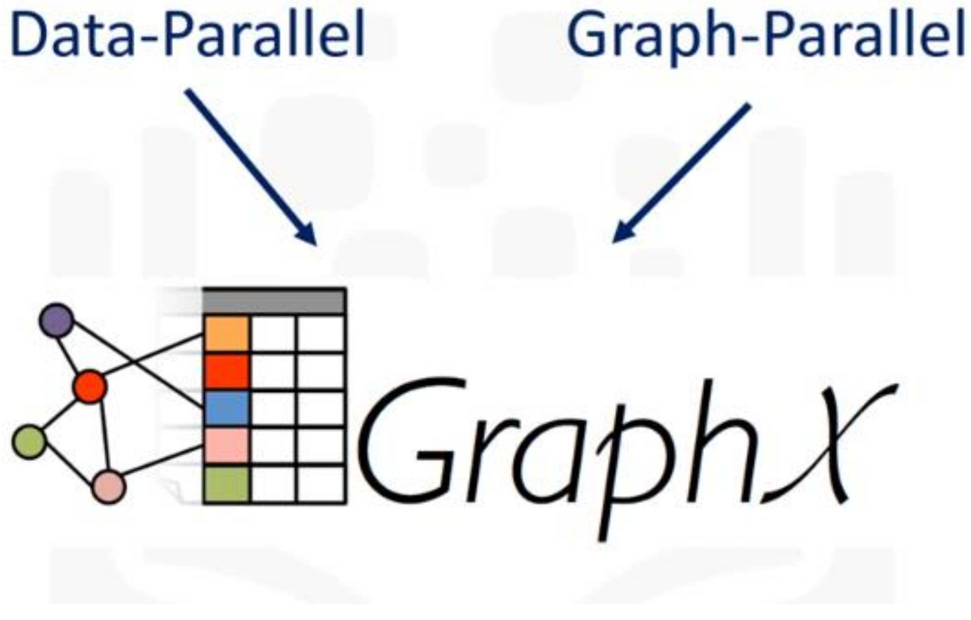


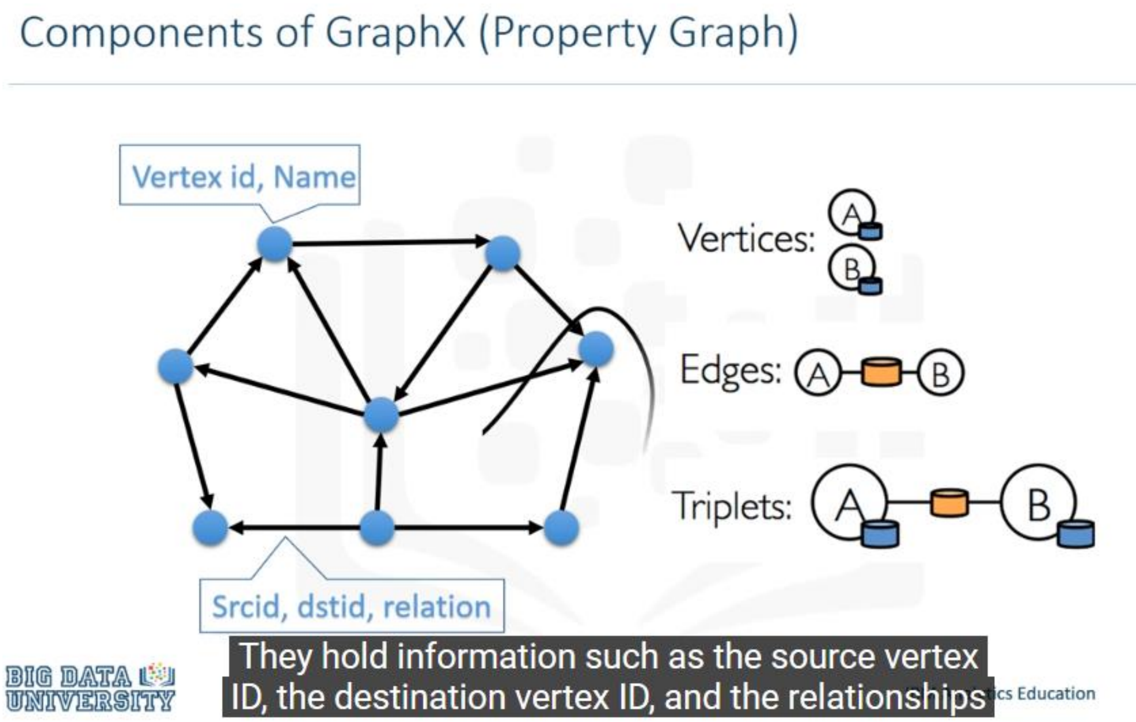
Preprocessing
Compute
Post Proc.
GraphLab
HIVE
</>
XML
Raw Data
ETL
Slice
Compute
Analyze
Initial Graph
Subgraph
PageRank
Top Users
Repeat
In this generic programming model, Hadoop extracts, transforms, and loads raw data as

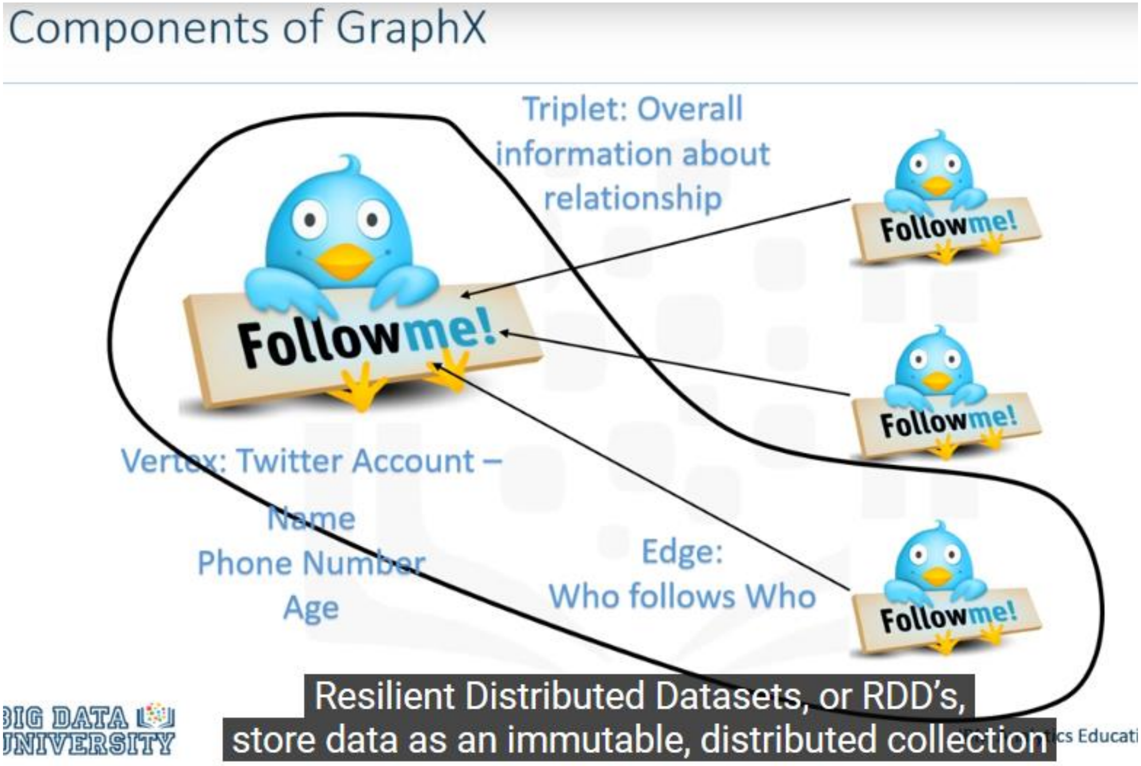


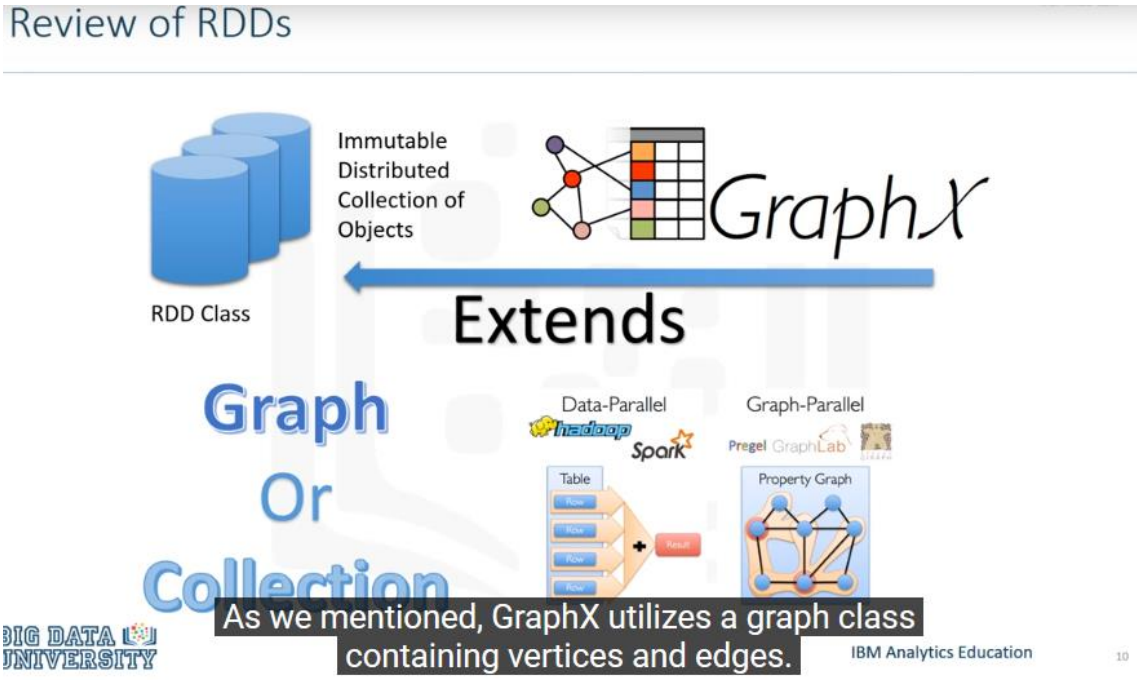
Preprocessing
Compute
Post Proc.
GraphLab
HIVE
</>
XML
Raw Data
ETL
Slice
Compute
Analyze
Initial Graph
Subgraph
PageRank
Top Users
Repeat
GraphLab then uses an algorithm such as PageRank for computation.

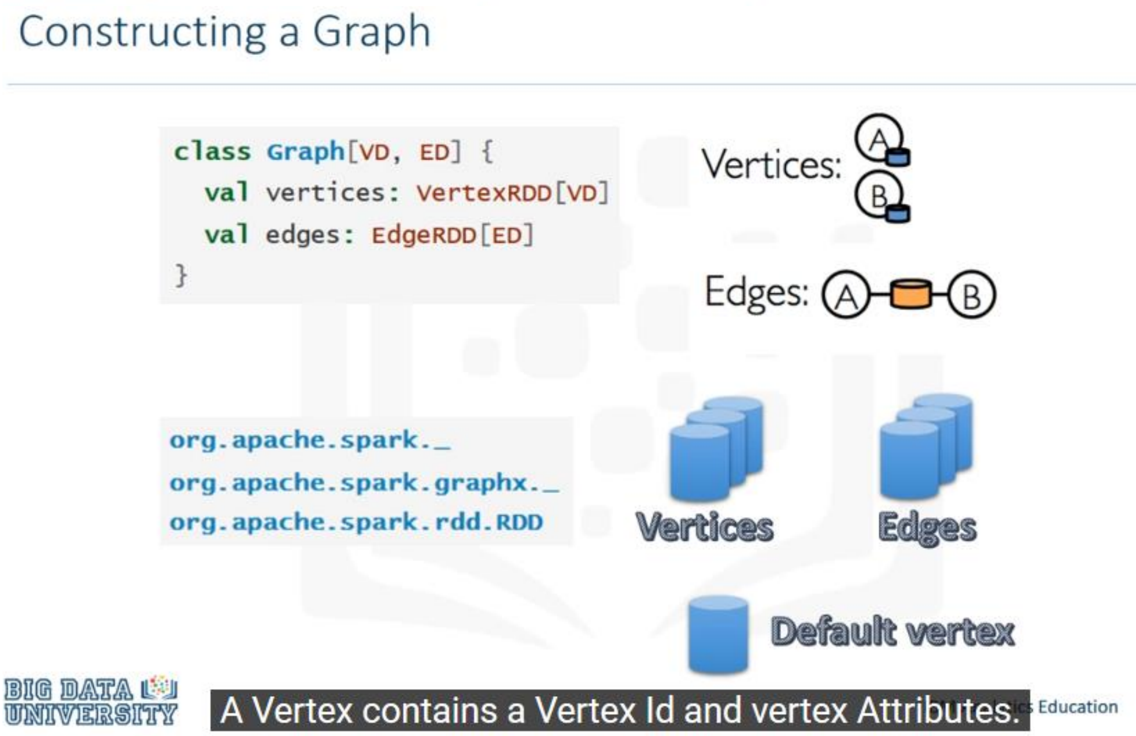

Preprocessing
Compute
Post Proc.
GraphLab
HIVE
XML
Raw
Data
ETL
Initial
Graph
Slice
Subgraph
Compute
PageRank
Analyze
Top
Users
Repeat
But the process of changing systems and repetition
may lead to accuracy problems and inefficiencies.

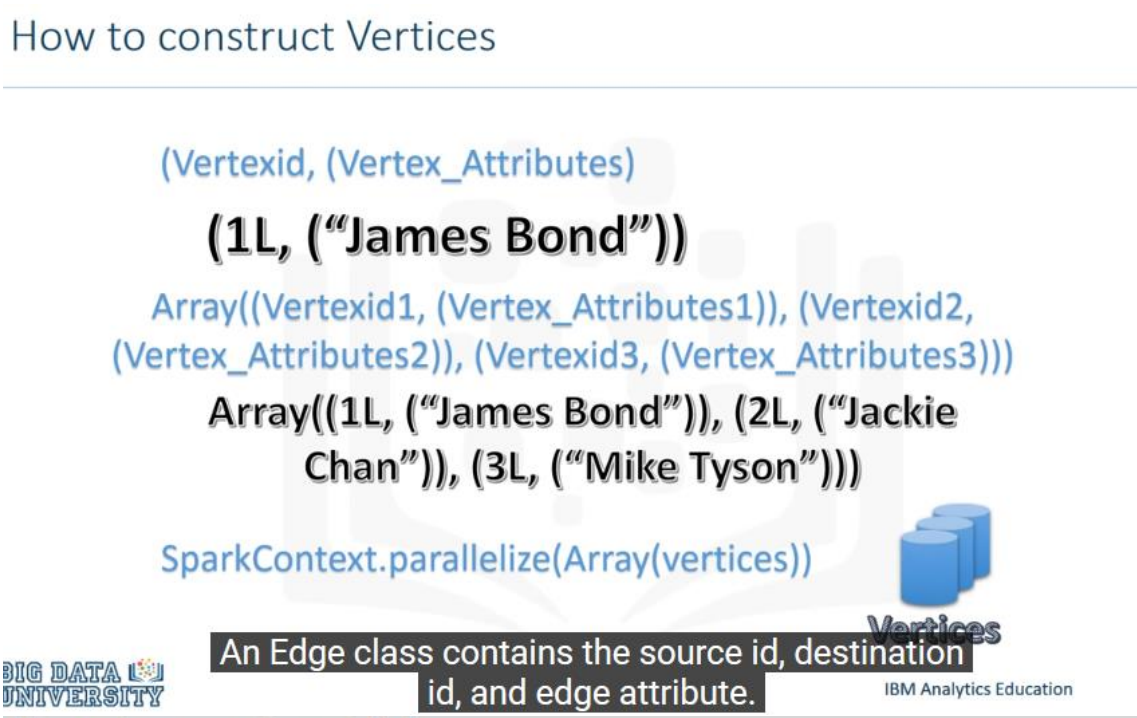

Data-Parallel
Graph-Parallel
GraphX


Components of GraphX (Property Graph)
Vertex id, Name
Vertices:
Edges:
Triplets:
Srcid, dstid, relation
They hold information such as the source vertex ID, the destination vertex ID, and the relationships
BIG DATA UNIVERSITY

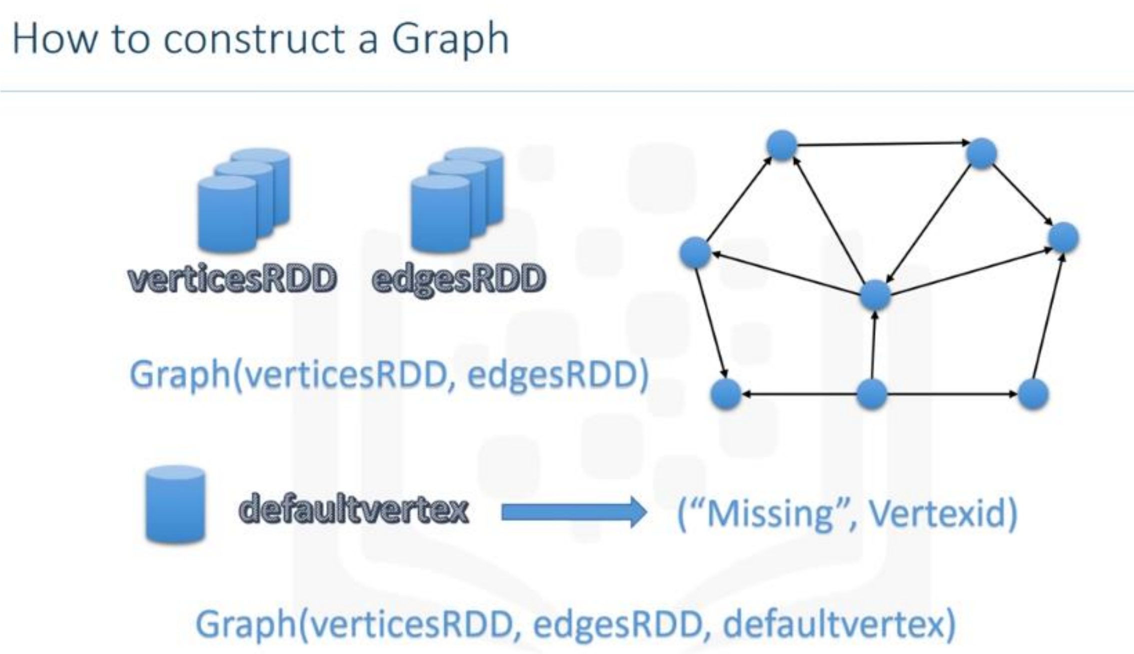

Components of GraphX
Triplet: Overall information about relationship
Followme!
Vertex: Twitter Account –
Name
Phone Number
Age
Edge:
Who follows Who
Resilient Distributed Datasets, or RDD's, store data as an immutable, distributed collection
BIG DATA UNIVERSITY


Review of RDDs
Immutable
Distributed
Collection of
Objects
GraphX
RDD Class
Extends
Graph
Or
Collection
Data-Parallel
Spark
Graph-Parallel
Pregel GraphLab
Table
Property Graph
As we mentioned, GraphX utilizes a graph class
containing vertices and edges.
BIG DATA UNIVERSITY
IBM Analytics Education
10


Constructing a Graph
class Graph[VD, ED] {
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
}
Vertices:
Edges:
org.apache.spark._
org.apache.spark.graphx._
org.apache.spark.rdd.RDD
Vertices
Edges
Default vertex
BIG DATA UNIVERSITY
A Vertex contains a Vertex Id and vertex Attributes.
Education


How to construct Vertices
(Vertexid, (Vertex_Attributes)
(1L, ("James Bond"))
Array((Vertexid1, (Vertex_Attributes1)), (Vertexid2, (Vertex_Attributes2)), (Vertexid3, (Vertex_Attributes3)))
Array((1L, ("James Bond")), (2L, ("Jackie Chan")), (3L, ("Mike Tyson")))
SparkContext.parallelize(Array(vertices))
Vertices
An Edge class contains the source id, destination id, and edge attribute.
BIG DATA UNIVERSITY
IBM Analytics Education


How to construct Edges
Edge(Srcid, Dstid, attr)
Edge(1L, 2L, "Friends")
Array(Edge(Srcid1, Dstid1, attr), Edge(Srcid2, Dstid2, attr), Edge(Srcid3, Dstid3, attr))
Array(Edge(1L, 2L, "Friends"), Edge(2L, 3L, "Friends"))
SparkContext.parallelize(Array(edges))
BIG DATA UNIVERSITY
We'll call them verticesRDD and edgesRDD.
Analytics Education



Hello, and welcome.
My name is Deborah, and this lesson is an Introduction to Graph-Parallel.
GraphX is a graph processing library built into Apache Spark.
GraphX uses the concept of the Property Graph, which is a directed multigraph, in combination
with Spark's Resilient Distributed Dataset concept, or RDD.
This yields a hybrid computational methodology.
Data-Parallel analysis systems, such as Hadoop and Spark, focus on distributing data across multiple nodes and systems.
Here, processing is handled in parallel.
Graph-Parallel systems, such as Pregel, GraphLab and Giraph, efficiently execute graph algorithms
through partitioning and distributing techniques.
However, due to the way that each formats and processes data, both system approaches can suffer from excessive data movement and processing inefficiencies.
In this generic programming model, Hadoop extracts, transforms, and loads raw data as an initial graph.
From here, Hadoop slices the initial graph into subgraphs as a required input to GraphLab.
GraphLab then uses an algorithm such as PageRank for computation.
Finally, the data is analyzed by Hive.
This process may repeat.
But the process of changing systems and repetition may lead to accuracy problems and inefficiencies.
GraphX addresses these problems by unifying data-parallel and graph-parallel methods into one unified library.
When it comes to providing a unified library, there are few, if any, alternatives to GraphX.
And while GraphX may not offer all of the capabilities that we need, we can always utilize Pregel, GraphLab or Apache Giraph for such features as Visualization.
Let's look at the Property Graph.
These are multi-directional graphs complete with properties attached.
Property Graphs are constructed using vertices, edges, and triplets.
Vertices are the nodes that we see here.

Vertices contain an identifier, or vertex ID, as well as a number of attributes, such as “Name” or “Label”.
The arrows between vertices are called edges.
These represent the relationships between vertices.
They hold information such as the source vertex ID, the destination vertex ID, and the relationships
between those two.
Finally, Triplets combine edges and vertices.
Triplets hold the information contained in an edge and a vertex combined.
For example, an edge has access to the vertex ID, but not to its attributes.
Triplets contain the edge relationship, and the vertex attributes and ID.
Triplets are created by GraphX when the graph is initialized.
A Twitter account is a good example of a vertex.
It contains information about you, and provides an address unique to you.
The same goes for other Twitter account holders.
One Twitter user can follow the activities of another Twitter user.
This follower relationship is a good example of an edge.
A Triplet contains the follower’s and your account information, as well as the followed-follower
relationship between you.
Resilient Distributed Datasets, or RDD’s, store data as an immutable, distributed collection of objects.
By extending RDD’s, GraphX can perform as either a graph or a collection tool.
This is the outcome of unifying Data-Parallel and Graph-Parallel systems.
As we mentioned, GraphX utilizes a graph class containing vertices and edges.
To build a graph, we will first need the libraries shown here.
Then, to instantiate the graph, we will need one RDD containing Vertices information, and another containing Edges information.
Finally, we need an optional, default vertex parameter.
This does not need to be an RDD.
A Vertex contains a Vertex Id and vertex Attributes.
This is represented in a Tuple.
Note that a vertex ID can be a long.
For example, this vertex has an ID of “1L”, where L stands for long.
Its attribute is “James Bond”.
We place multiple Tuples like this into an Array.
Finally we use the parallelize function of SparkContext to make an RDD of vertices
To construct an edge, we must construct an edge class.
An Edge class contains the source id, destination id, and edge attribute.
For example, we can use vertex 1L as the source ID, vertex 2L as the destination ID, and “Friends”
as the edge attribute.
We then place multiple edges into an Array.
Finally, we use the parallelize function of SparkContext to make an RDD of edges
So far we’ve built vertices and edges RDD’s.
We’ll call them verticesRDD and edgesRDD.
To create a graph, we call the “Graph” function and use verticesRDD as the first parameter, and edgesRDD as the second parameter.
Our Graph is constructed!
If needed, we can also input a third parameter called Default vertex.
If an edge cannot find the vertex it needs, it will link to this value as a default.
Defaul tvertex does not need to be an RDD.

It is a simple tuple as shown.
Using a default vertex, our "Graph" function is now written as shown here.
Thank you!

>>Lab:

If you are interested in more keyboard shortcuts, go to Help -> Keyboard Shortcuts

Hello! First before we start creating our graph, we will need to the import the following libraries:

- org.apache.spark._
- org.apache.spark.graphx._
- org.apache.spark.rdd.RDD

```
// Type your code here
import org.apache.spark._;
import org.apache.spark.graphx._;
import org.apache.spark.rdd.RDD;
```

Double-click **here** for the solution.

Now to begin, as a reminder, we have a SparkContext called `sc`.

Now next we will create the "Vertices" of our graph. Let's try to make it a simple, easy-to-relate graph. Let's use "Facebook" as an example. We will create an Array called `facebook_vertices` that consists of 3 people and 2 pages.

Now next we will create the "Vertices" of our graph. Let's try to make it a simple, easy-to-relate graph. Let's use "Facebook" as an example. We will create an Array called `facebook_vertices` that consists of 3 people and 2 pages.

```
val facebook_vertices = Array((1L, ("Billy Bill", "Person")), (2L, ("Jacob Johnson", "Person")), (3L, ("Andrew Smith", "Person")), (4L, ("Iron Man Fan Page", "Page")), (5L, ("Captain Am
```

```
facebook_vertices = Array((1,(Billy Bill,Person)), (2,(Jacob Johnson,Person)), (3,(Andrew Smith,Person)), (4,(Iron Man Fan Page,Page)), (5,(Captain America Fan Page,Page)))
Array((1,(Billy Bill,Person)), (2,(Jacob Johnson,Person)), (3,(Andrew Smith,Person)), (4,(Iron Man Fan Page,Page)), (5,(Captain America Fan Page,Page)))
```

Here, we are just making a simple array that has 3 People:

- Billy Bill
- Jacob Johnson
- Andrew Smith

and 2 Pages:

- Iron Man Fan Page
- Captain America Fan Page

These will become our vertices later on. Vertices carry an identifier (1L, 2L, 3L, ...) and user-defined attributes such as "Person" or "Page".

Next, we will create the relationships of each one of them. The variable relationships will become the "Edges" of our graph.

```
val relationships = Array(Edge(1L, 2L, "Friends"), Edge(1L, 3L, "Friends"), Edge(2L, 4L, "Follower"), Edge(2L, 5L, "Follower"), Edge(3L, 5L, "Follower"))
```

```
relationships = Array(Edge(1,2,Friends), Edge(1,3,Friends), Edge(2,4,Follower), Edge(2,5,Follower), Edge(3,5,Follower))
Array(Edge(1,2,Friends), Edge(1,3,Friends), Edge(2,4,Follower), Edge(2,5,Follower), Edge(3,5,Follower))
```

Now we have created another Array called relationships that are Edges, with attributes of the srcId (Source ID), dstId (Destination ID). These are the following relationships that we created:

- Billy is Friends with Jacob
- Billy is Friends with Andrew
- Jacob is a Follower of the Iron Man Fan Page
- Jacob is a Follower of the the Captain America Fan Page
- Andrew is a Follower of the the Captain America Fan Page

Now we have our Vertices `facebook_vertices` and Edges `relationships`. However, they are just Arrays. When we create our Graph, these variables need to be RDDs. To create RDDs, we will use the parallelize function of SparkContext `sc`. We will also have to make sure that the correct types are labeled in type format.

```
val vertexRDD: RDD[(Long, (String, String))] = sc.parallelize(facebook_vertices)
val edgeRDD: RDD[Edge[String]] = sc.parallelize(relationships)
```

```
vertexRDD = ParallelCollectionRDD[0] at parallelize at <console>:44
edgeRDD = ParallelCollectionRDD[1] at parallelize at <console>:45
ParallelCollectionRDD[1] at parallelize at <console>:45
```

Now we have our Vertices and Edges in proper format, but before we define our graph we just need to define one user - which will be "fallback" user. This user will be defaulty connected to any edges that lead to a non-existant Vertex. Let's called it "Self" - since you can be friends with "Yourself" and have a page that follows "Itself."

```
val defaultvertex = ("Self", "Missing")
```

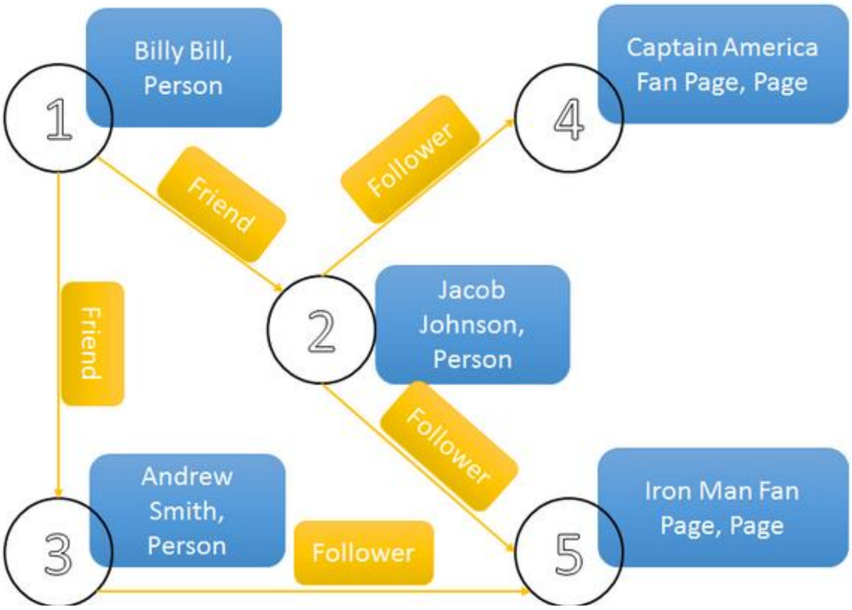
```
defaultvertex = (Self,Missing)
```

This variable is just a tuple. Now we can move onto creating our Graph. We will create a variable called facebook which will be our instantiate of Graph with 3 variables - `vertexRDD`, `edgeRDD`, and `defaultvertex`.

```
val facebook = Graph(vertexRDD, edgeRDD, defaultvertex)
```

```
facebook = org.apache.spark.graphx.impl.GraphImpl@540d63ea
org.apache.spark.graphx.impl.GraphImpl@540d63ea
```

Here's a visual representation created by me to show what the graph should look like:



We did it! We made facebook! (multi-directional graph representing facebook :)). Now the Graph we created has some interesting components that it has made from our parameters. Let's try printing out the vertices component of facebook.

```
[19]: // Type your code here
print(facebook.vertices)
```

```
VertexRDDImpl[11] at RDD at VertexRDD.scala:57
```

This is the vertices of our of graph. You can do the same for Edges by using the edges components. Try printing it out!

```
// Type your code here
print(facebook.edges)
```

```
EdgeRDDImpl[13] at RDD at EdgeRDD.scala:41
```

Now, what's so important about these two components? You can use them to create views of their respective components of the graph! However, they are slightly different from each other, so we will take a look at vertices first!

So right now vertices is called as a whole, so we will need to separate the results we want using the filter function. Then we will make cases for each attribute and then define a condition to be met.

```
facebook.vertices.filter { case (id, (name, user_type)) => user_type == "Person" }.count
```

As you can see, we used the filter function, defined the attributes of the vertex then made a condition that only selects a "Person" in our graph. We counted this to produce a result of 3, which matches the 3 vertices (people) in our graph. However, we could have easily have replaced the count function with a collect and have dealt with it as a tuple and used for loops to print out a each person.

Now let's try the same with Edges except it only has one defined case variable, which is the edge itself. However, the Edge class has attributes such as srcId (sourceID), dstId (destinationID), and attr (Attribute) which stores the edge property.

Let's see if you are able to use the filter function on facebook.edges to find how many people follow the "Captain America Fan Page"

Hint: The destination will be the Captain America Fan Page's ID and the relationship has to be Follower.

```
// Type your code here
facebook.edges.filter { case (relation) => relation.dstId == 5L && relation.attr == "Follower"}.count
```

```
2
```

The answer should be 2! So now that you have gotten some insight into Vertices and Edges of the graph, you may think be thinking how can I visualize GraphX? Unfortunately, GraphX does not have any visualizations built-in, it is mainly a parallel graph processing library. The closest options we have to visualize the data is through views as we did above with Vertices and Edges.

However, there is an easier way to create views, and that is with the EdgeTriplet class. This class contains information about the Edge and Vertex because of it logical join. We will discuss more later on, however here is a little taste of what EdgeTriplets can do.

```
val selected = facebook.triplets.filter { case (triplet) => triplet.srcAttr._1 == "Billy Bill"}.collect
```

```
selected = Array(((1,(Billy Bill,Person)),(2,(Jacob Johnson,Person)),Friends), ((1,(Billy Bill,Person)),(3,(Andrew Smith,Person)),Friends))
Array(((1,(Billy Bill,Person)),(2,(Jacob Johnson,Person)),Friends), ((1,(Billy Bill,Person)),(3,(Andrew Smith,Person)),Friends))
```

```
for (person <- selected) {
    print(person.srcAttr._1)
    print(" is ")
    print(person.attr)
    print(" with ")
    println(person.dstAttr._1)
```

First we created a variable called selected which contained the collection of the information for Billy Bill. Then we cycled through a for loop of that collection and outputted Billy Bill's relationships and with whom. You are able to do much more with the EdgeTriplet class, but that will be discussed later.

Note: You can access the "selected" variables by using the () and putting an index in between the brackets.

Can you think of the possibilities of the EdgeTriplet class?

Now with that lingering question in your mind, let's see if you can create another graph with the knowledge you have gained!

This time, we will make a little more different, and it will just model "real" relationships between people. Let's pick some popular Simpson characters:

- Homer Simpson -> VertexId = 1
- Bart Simpson -> VertexId = 2
- Marge Simpson -> VertexId = 3
- Milhouse Houten -> VertexId = 4

However, we are going to try to create an RDD vertex called characters all in one step! Let's see if you can combine the two steps we learned earlier!

```
// Type your code here
val characters: RDD[(VertexId, (String, String))] = sc.parallelize(Array((1L, ("Homer Simpson", "Person")), (2L, ("Bart Simpson", "Person")), (3L, ("Marge Simpson", "Person")), (4L, ("
```

Awesome! Now let's model some of their relationships (For simplicity sake we will only model a few):

- Homer Simpson is the Father of Bart Simpson
- Marge Simpson is the Wife of Homer Simpson
- Bart Simpson is the Friend of Milhouse Houten

We can also create an EdgeRDD variable called `simpson_relationships` in one step too! It is done similarly as the previous step, so if your stuck, take a look there!

```
// Type your code here
val simpson_relationships : RDD[Edge[String]] = sc.parallelize(Array(Edge(1L, 2L, "Father"), Edge(3L, 1L, "Wife"), Edge(2L, 4L, "Friends")))
```

```
simpson_relationships = ParallelCollectionRDD[21] at parallelize at <console>:41
ParallelCollectionRDD[21] at parallelize at <console>:41
```

Double-click **here** for the solution.

Now we will just reuse the defaultvertex variable as our "fallback" user. If you don't have this variable instantiated, then go ahead and scroll up to do so.

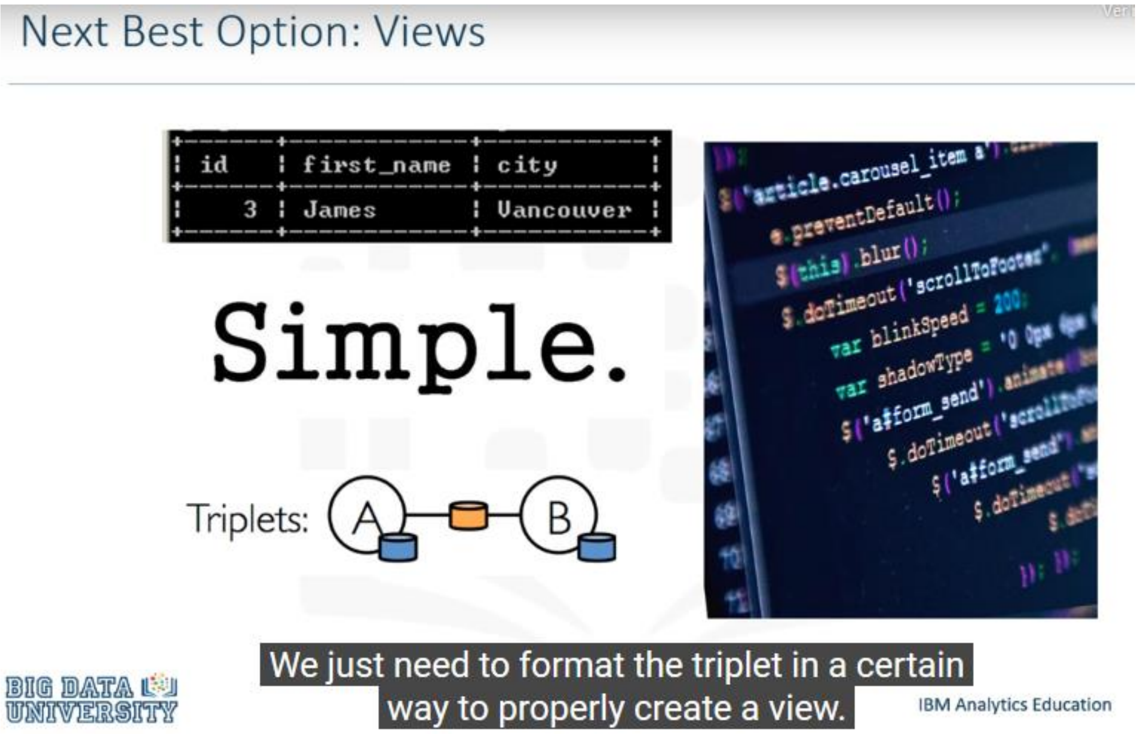
Now let's create our graph with our Vertices (`characters`), Edges (`simpson_relationships`), and defaultvertex called `the_simpsons` .

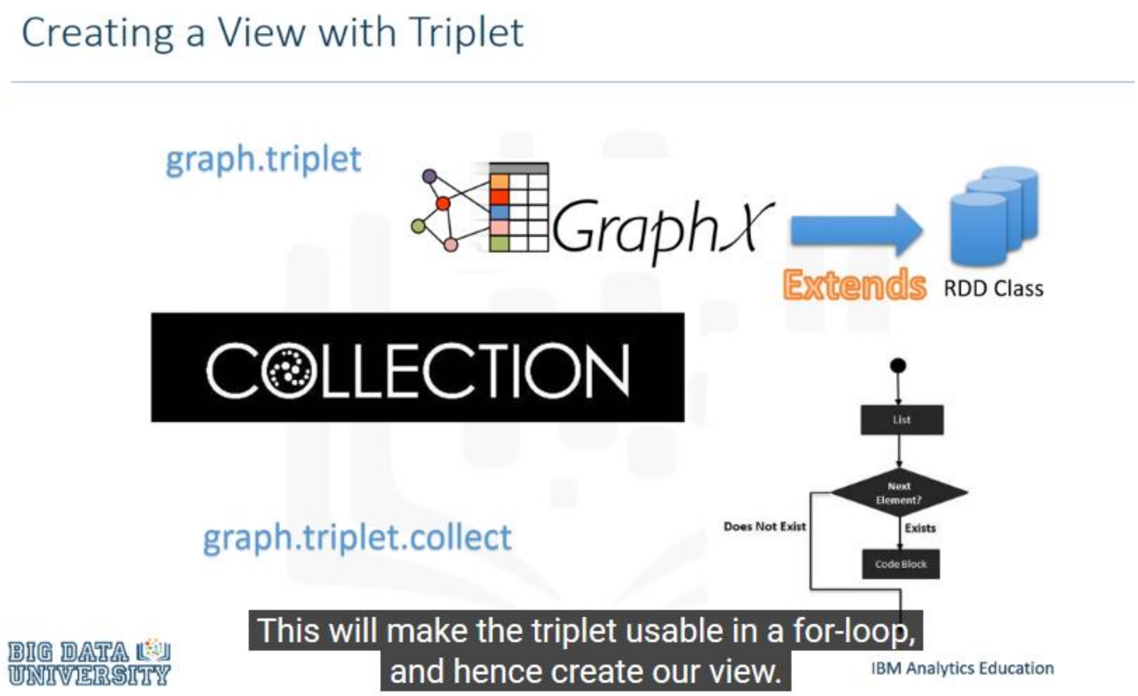
```
// Type your code here
val the_simpsons = Graph(characters, simpson_relationships, defaultvertex)
```

MODULE 2 VISUALIZING GRAPHX AND EXPORING GRAPH OPERATORS


How to Visualize with GraphX?
PROCESSING
Unfortunately,
Unfortunately the GraphX library doesn't
include built-in visualization tools.
BIG DATA UNIVERSITY
IBM Analytics Education


GraphX Visualization Alternatives
However...
Gephi
GraphLab
GraphX has done the processing.
BIG DATA UNIVERSITY
IBM Analytics Education


Next Best Option: Views
id
first_name
city
3
James
Vancouver
Simple.
Triplets:
A
B
We just need to format the triplet in a certain way to properly create a view.
BIG DATA UNIVERSITY
IBM Analytics Education


Creating a View with Triplet
graph.triplet
GraphX
Extends
RDD Class
COLLECTION
graph.triplet.collect
List
Next Element?
Does Not Exist
Exists
Code Block
This will make the triplet usable in a for-loop, and hence create our view.
BIG DATA UNIVERSITY
IBM Analytics Education

Creating a View with Triplet

```
for (triplet <- graph.triplet.collect) {
--------Code---------
}
```

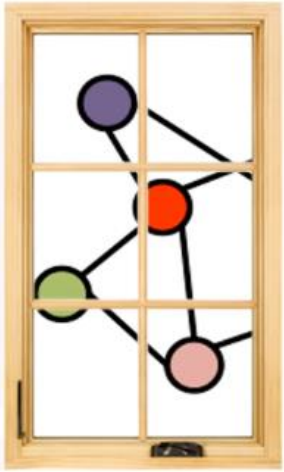
srcAttr -> Source Attribute

dstAttr -> Destination Attribute

srcId -> Source ID

dstId -> Destination ID

attr -> Edge Attribute



BIG DATA

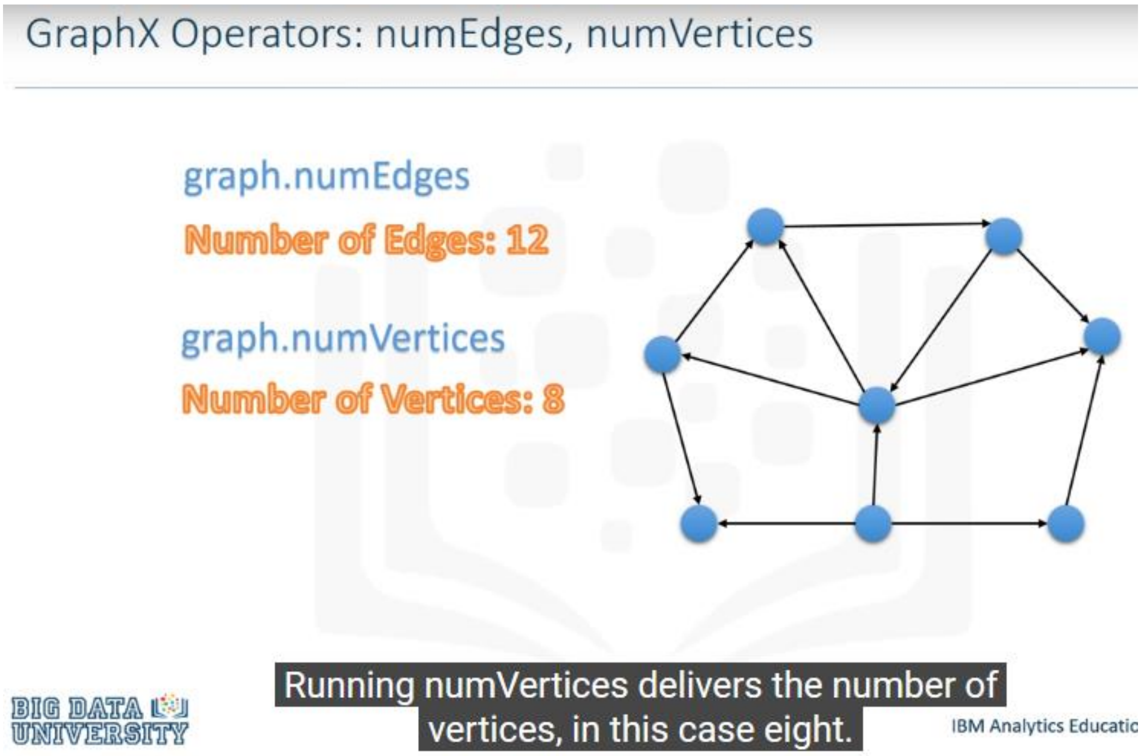
GraphX Operators

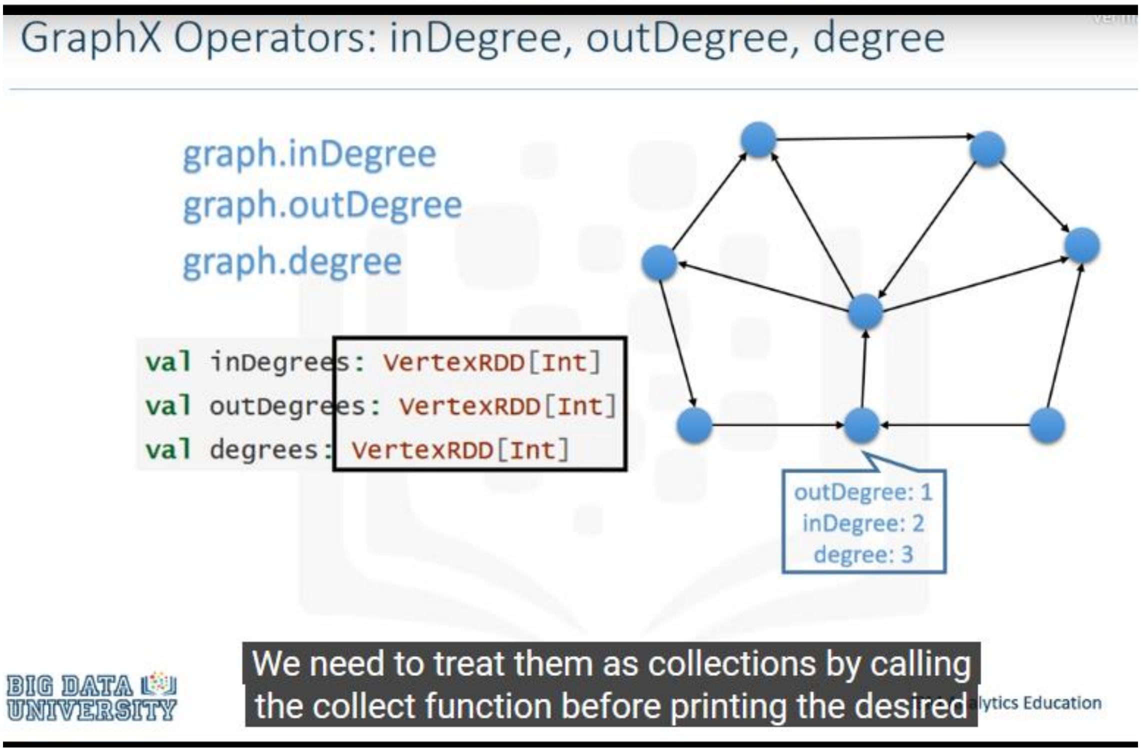
```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // Information about the Graph ===================================================
  val numEdges: Long
  val numVertices: Long
  val inDegrees: VertexRDD[Int]
  val outDegrees: VertexRDD[Int]
  val degrees: VertexRDD[Int]
  // Views of the graph as collections =============================================
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
  val triplets: RDD[EdgeTriplet[VD, ED]]
  // Functions for caching graphs ==================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // Change the partitioning heuristic  ============================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
  // Transform vertex and edge attributes ==========================================
  def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
    : Graph[VD, ED2]
  // Modify the graph structure ====================================================
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexID, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
```

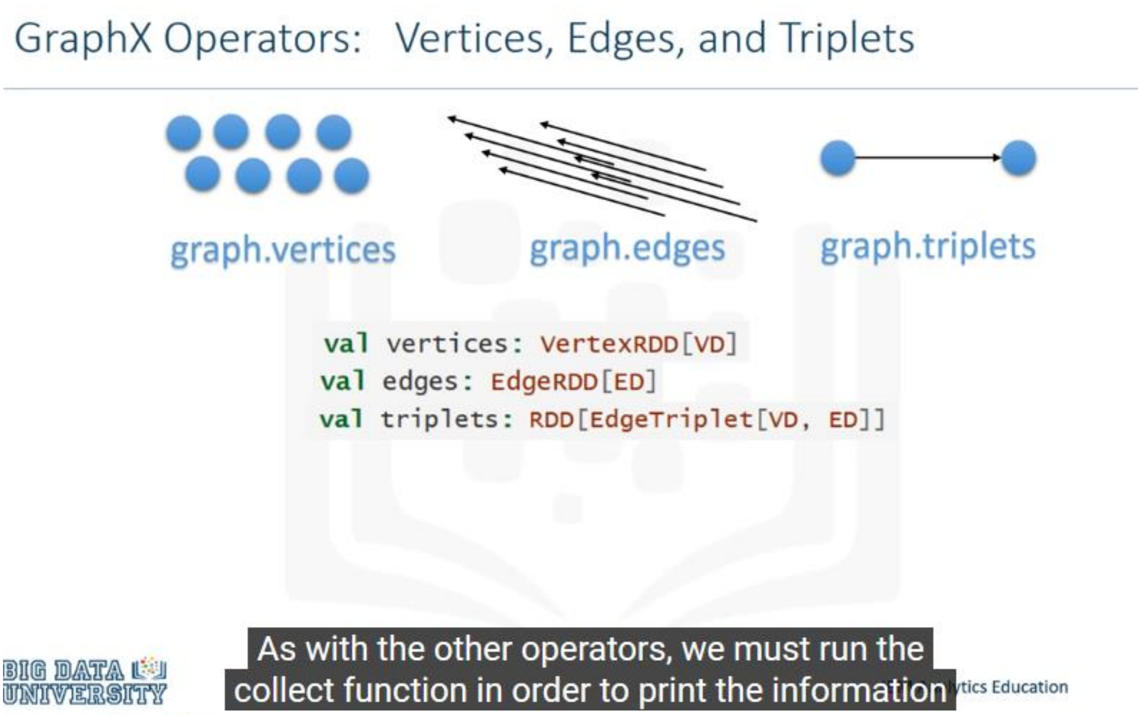
BIG DATA UNIVERSITY

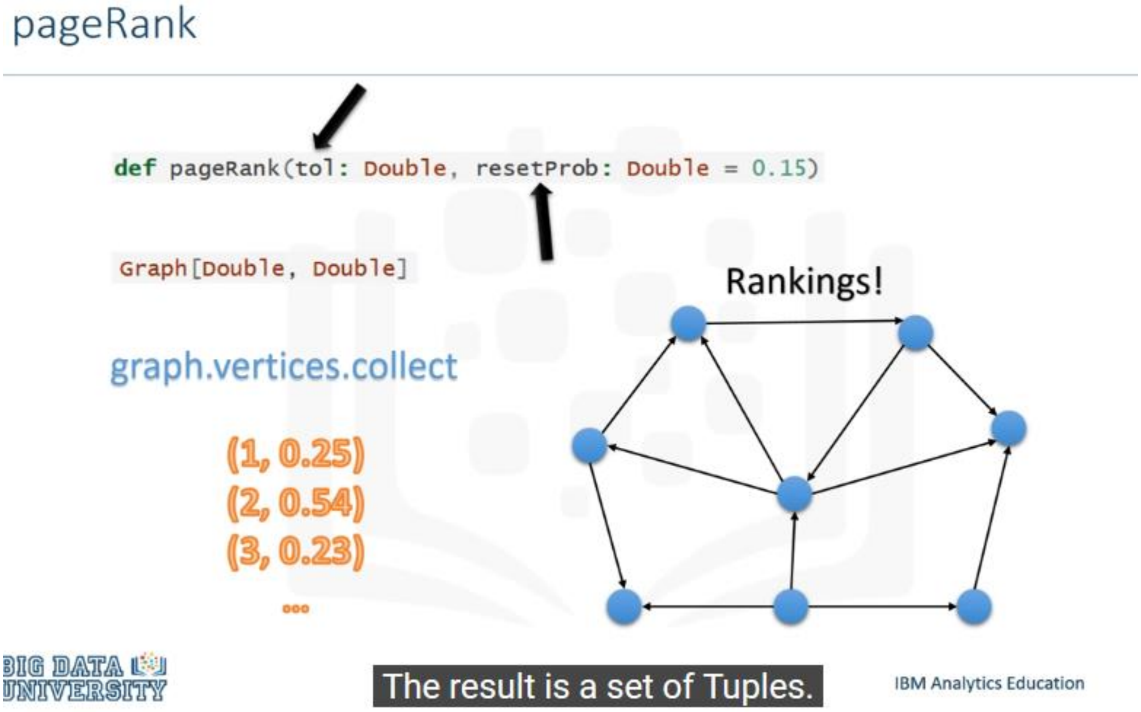
Let's look at some of the key operators.

IBM Analytics Education


GraphX Operators: numEdges, numVertices
graph.numEdges
Number of Edges: 12
graph.numVertices
Number of Vertices: 8
Running numVertices delivers the number of
vertices, in this case eight.
BIG DATA UNIVERSITY
IBM Analytics Educatic


GraphX Operators: inDegree, outDegree, degree
graph.inDegree
graph.outDegree
graph.degree
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]
outDegree: 1
inDegree: 2
degree: 3
We need to treat them as collections by calling
the collect function before printing the desired
BIG DATA UNIVERSITY
lytics Education





Hello and welcome.
My name is Deborah.
In this lesson we will learn about visualizing GraphX and exploring graph operators.
GraphX is primarily a graph processing library.
However, we frequently need to visualize our data.
Unfortunately the GraphX library doesn't include built-in visualization tools.
But that doesn't mean we can't use other libraries along with GraphX.
Libraries such as Gephi or GraphLab are available to us to help provide visualization after GraphX has done the processing.
We can also create views for visualizing data in GraphX.
Views provide a simple approach to visualization.

We can use the Triplet class to create a view since each triplet contains all the information we need to represent a data relationship.
We just need to format the triplet in a certain way to properly create a view.
Let's see how this works.
The graph.triplet function allows us to call upon the triplets of our graph.
However, we won't be able to print from just this triplet class.
But since GraphX extends the RDD Class, we can treat the triplet like a collection, and then run the collect function on it.
This will make the triplet usable in a for-loop, and hence create our view.
If we are using Scala, the for-loop would look like this.
All we need to do then is code how the attributes should be ordered when they are printed.
We can call the following functions on each triplet to acquire the following information:
The SrcAttr function delivers the source attribute.
The dstAttr function delivers the destination attribute.
SrcId delivers the Source ID, and so on.
With these functions in place, we only need to organize them so that we print what we want in the appropriate order.
We have now created a view of our graph.
GraphX includes a lot of functionality.
Here is just a partial list of the many functions available.
Let's look at some of the key operators.
Let's start with numEdges and numVertices.
Running numEdges delivers a value that represents the number of edges in the graph, in this case twelve.
Running numVertices delivers the number of vertices, in this case eight.
Both of these operators return long values.
Next let's look at three operators together: inDegree, outDegree, and degree.
A degree in a property graph provides information about the number of ingoing, outgoing, or total edges associated with a particular vertex.
In this example, there is one edge leading away from the vertex.
Therefore the outDegree for this vertex returns the value one.
Likewise there are two edges leading toward the vertex, so inDegree returns the value two.
There are three total edges associated with this vertex, so the degree operator returns the value three.
Each of these functions returns a Vertex RDD containing the information in the call function, and takes the form VertexRDD[Int].
But similar to calling a triplet, these functions are not in a suitable format for printing.
We need to treat them as collections by calling the collect function before printing the desired information in a for-loop.
The vertices, edges, and triplets functions return a Vertex RDD, an Edge RDD, or an Edge Triplet, respectively, of all the vertices, edges, and triplets in the graph.
As with the other operators, we must run the collect function in order to print the information in these operators.
Now we will look at a basic graph algorithm function called pageRank.
This function helps to determine the importance or popularity of vertices in the graph.
As the name implies, the algorithm ranks the vertices by correlating their relation to edges, both in terms of quality and quantity.
The tolerance of the algorithm is determined by a Double as shown here.
The reset probably is a default parameter set to 0.15.
However, this can be changed.
pageRank returns a Graph of the type Double, Double.

We can print out the ranking of each vertex by running the graph.vertices.collect function on the returned graph.
The result is a set of Tuples.
The first element of each tuple is the vertex ID.
The second element is the vertex ranking.
Thank you!

>> Lab:

If you are interested in more keyboard shortcuts, go to Help -> Keyboard Shortcuts

So in the last exercise, you looked at creating our simple recreation of "facebook". You were given most of the code, so let's go ahead and recreate the same graph with a little less help and a bit more intitution!

First we will import the following libraries:

- org.apache.spark._
- org.apache.spark.graphx._
- org.apache.spark.rdd.RDD

```
// Type your code here
import org.apache.spark._;
import org.apache.spark.graphx._;
import org.apache.spark.rdd.RDD;
```

In our "facebook" graph we created we had the following People:

- Billy Bill -> VertexId = 1
- Jacob Johnson -> VertexId = 2
- Andrew Smith -> VertexId = 3

and 2 Pages:

- Iron Man Fan Page -> VertexId = 4
- Captain America Fan Page -> VertexId = 5

And we are going to create the vertices in one step! This will be tied to the variable called vertexRDD

Hint: The type is RDD[(Long, (String, String))]

```
// Type your code here
val vertexRDD: RDD[(Long, (String, String))] = sc.parallelize(Array((1L, ("Billy Bill", "Person")), (2L, ("Jacob Johnson", "Person")), (3L, ("Andrew Smith", "Person"
```

```
vertexRDD = ParallelCollectionRDD[0] at parallelize at <console>:35
ParallelCollectionRDD[0] at parallelize at <console>:35
```

Hint: The Type is RDD[Edge[String]]

```
// Type your code here
val edgeRDD: RDD[Edge[String]] = sc.parallelize(Array(Edge(1L, 2L, "Friends"), Edge(1L, 3L, "Friends"), Edge(2L, 4L, "Follower"), Edge(2L, 5L, "Follower"), Edge(3L,
```

```
edgeRDD = ParallelCollectionRDD[1] at parallelize at <console>:35
ParallelCollectionRDD[1] at parallelize at <console>:35
```

Double-click **here** for the solution.

Now let's create the a variable called `defaultvertex` which will be the "fallback" for any edges that cannot connect to a vertex. It is only a tuple which contains "Self" and "Missing"

```
// Type your code here
var defaultvertex = ("Self", "Missing")
```

```
defaultvertex = (Self,Missing)
(Self,Missing)
```

Alright, now let's go ahead and construct the Graph! We will name it facebook again!

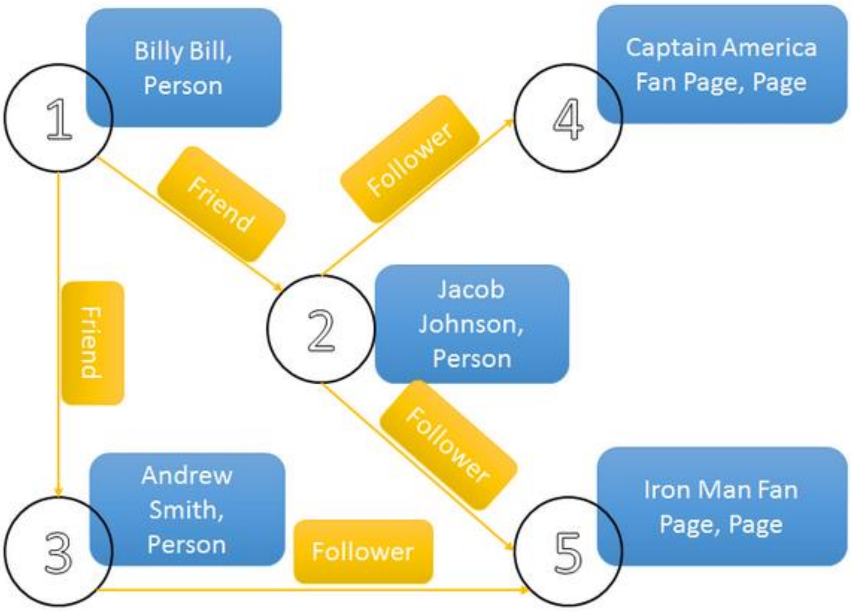
```
// Type your code here
var facebook = Graph(vertexRDD, edgeRDD, defaultvertex)
```

```
facebook = org.apache.spark.graphx.impl.GraphImpl@693d72a2
org.apache.spark.graphx.impl.GraphImpl@693d72a2
```

Double-click **here** for the solution.

Perfect! Here's a reminder of the visualized Graph:

Perfect! Here's a reminder of the visualized Graph:



Alright so now we will take a look at the few of the Graph Operators! These Graph Operators are called by using the Graph "facebook" variable we created. You use them by calling them on the Graph variable or "facebook" in our case. Let's try to extract how many vertices there are in this graph by using `numVertices` function.

```
// Type your code here
facebook.numVertices
```

```
5
```

Double-click **here** for the solution.

Sweet! Now let's find out the number of edges using `numEdges` function.

```
// Type your code here
facebook.numEdges
```

```
5
```

Ironically, they are both the same. So make sure you didn't just use the same function both times! Haha.

Now the next Operator we will look at involve degrees. In this case we are talking about degrees as the number of edges a vertex touches! The Edges in a multi-directional graph have a direction. As you can see, sometimes it can be mutual such as:

-> Billy is a Friend of Andrew.

-> Andrew is a Friend of Billy.

However there are cases where the edge or "relationship" is not mutual. This is such as:

-> Jacob is a Follower of the Captain America Fan Page.

-> Captain America Fan Page is a Follower of Jacob.

So, if we are looking at a specific vertex, we can determine the edges that point "out" with the function `outDegrees`. However, the question is... How do we find a specific vertex? We use the filter function like we did in the last exercise!

We can use the filter function on the `outDegrees` function of facebook and select the case where the id is the number or numbers we want.

Let's find Billy's outDegrees information by filtering it with a id of 1 and using the collect function afterwards. Let's save it as `Billy_outDegree`.

Note: The case we will need is case(id, outdegree), as the id of the person is the first parameter and the outdegree number is the second parameter.

```
// Type your code here
var Billy_outDegree = facebook.outDegrees.filter{ case(id, outdegree) => id == 1}.collect
```

```
Billy_outDegree = Array((1,2))
```

You got an error when you tried to print the `Billy_inDegree` didn't you? That's to be expected because since there wasn't an inDegree value for Billy's vertex, there wasn't an anything in `Billy_inDegree` variable.

Now let's take a look at the degrees operator. We will do something different than before, and go ahead and use a for loop to cycle through the total degree of each vertex (inDegree + OutDegree)

```
// Type your code here
for (degree <- facebook.degrees.collect)
{
    println(degree)}
```

```
(4,1)
(1,2)
(3,2)
(5,2)
(2,3)
lastException: Throwable = null
```

Now the next Graph Operators we are looking at is .vertices, .edges, and .triplets. As you have used, and seem them before in the last exercise. They are Graph Operators and it is important to know how to use each of their cases:

- .vertices -> Uses format of the defined Vertices of the graph.
  Ex. We defined our Vertices as (Long, (String, String)), therefore when you call a case on this, you must define variables for each such as (id, (name, user_type
- .edges -> Uses format of the defined Edges of the graph.
  Ex. We defined our Edges as Edge[String], therefore when you call a case on this, you can just define one variable such as (relation). However, this variable will have attributes such as .srcId (Source Id), .dstId (Destination Id), and .attr (Attribute).
- .triplets -> Uses the combined format of the defined Vertices and Edges.
  Ex. Follow the above example, when we call a case on this, you define one variable such as (triplet). And this variable will have attributes of both Vertices an Edges such as .srcAttr (Source Attribute), .dstAttr (Destination Attribute) from Vertices, and .srcId (Source Id), .dstId (Destination Id), and .attr (Attribute) from Edges.

So since you've dealt with .vertices and edges, we do a quick example with each then start looking at how to visualize the graph with .triplets since it a combination of .vertices and .edges.

Unfortunately, GraphX does not have any build-in visualization, so it's important to know how to create views. Let's go ahead and trying printing out all of the vertices.

Hint: Use a for loop and the collect function on .vertices

```
// Type your code here
for (degree <- facebook.degrees.collect)
{
    println(degree)
```

Awesome! Now let's do the same with edges just so we have an idea of all the vertices and edges.

```
// Type your code here
for (edge <- facebook.edges.collect) {
    println(edge)
}
```

Alright, now let's use triplets to create a view of the graph. Just like in last two examples, we will use the collect function on .triplets, however we will denote the Source Attribute (.srcAttr), the edge attribute (.attr), and the Destination Attribute (.dstAttr) all in the same `println` statement to denote each relationship.

Hint: Make sure to use the index on the Source and Destination Attribute!

```
// Type your code here
for (triplet <- facebook.triplets.collect) {
    print(triplet.srcAttr._1)
    print(" is a ")
    print(triplet.attr)
    print(" of ")
    println(triplet.dstAttr._1)
}
```

Now we will take a look at an important algorithm in GraphX: pageRank.

Pagerank is a algorithm that measures the importance of each vertex by directly correlating it's importance with edges (properties and quantity). There are two options for Pagerank, static and dynamic. Static runs for a fixed number of iterations while dynamic runs until the rank converges.

We won't worry too much as we will just introduce the concept. Now, in this case I went ahead and used the pageRank function on our graph, and collected the vertices into a variable called rank. Now go ahead and try to print it out!

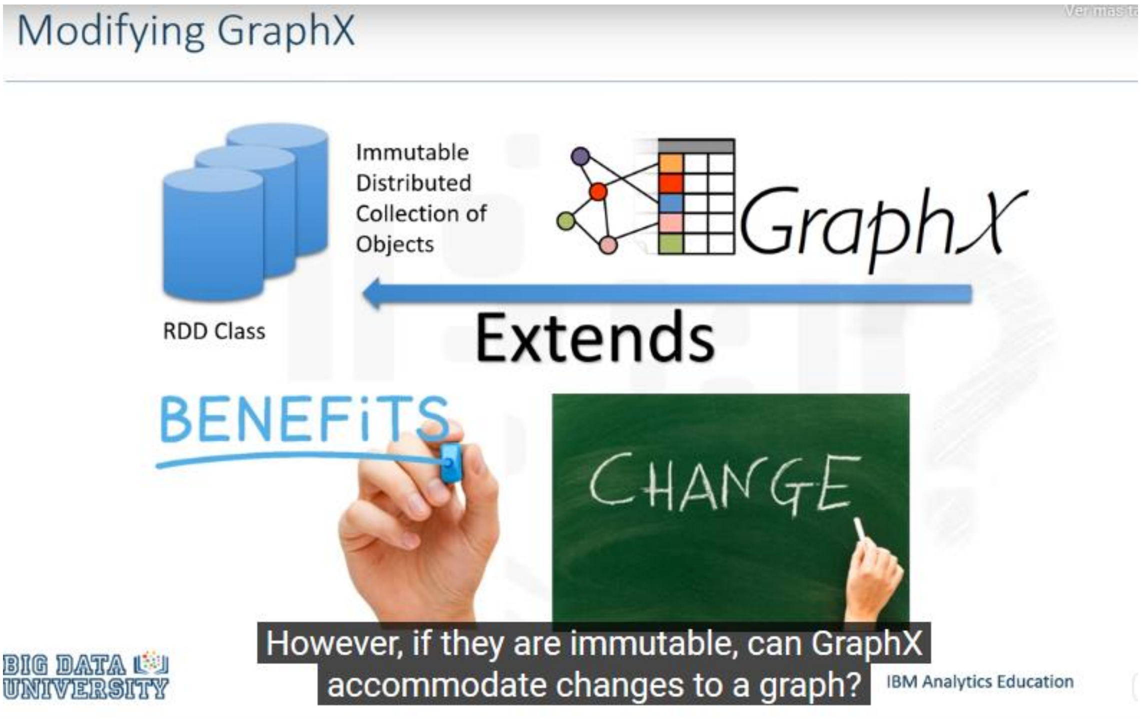
Note: rank is a collection, so you will need to use a for loop!

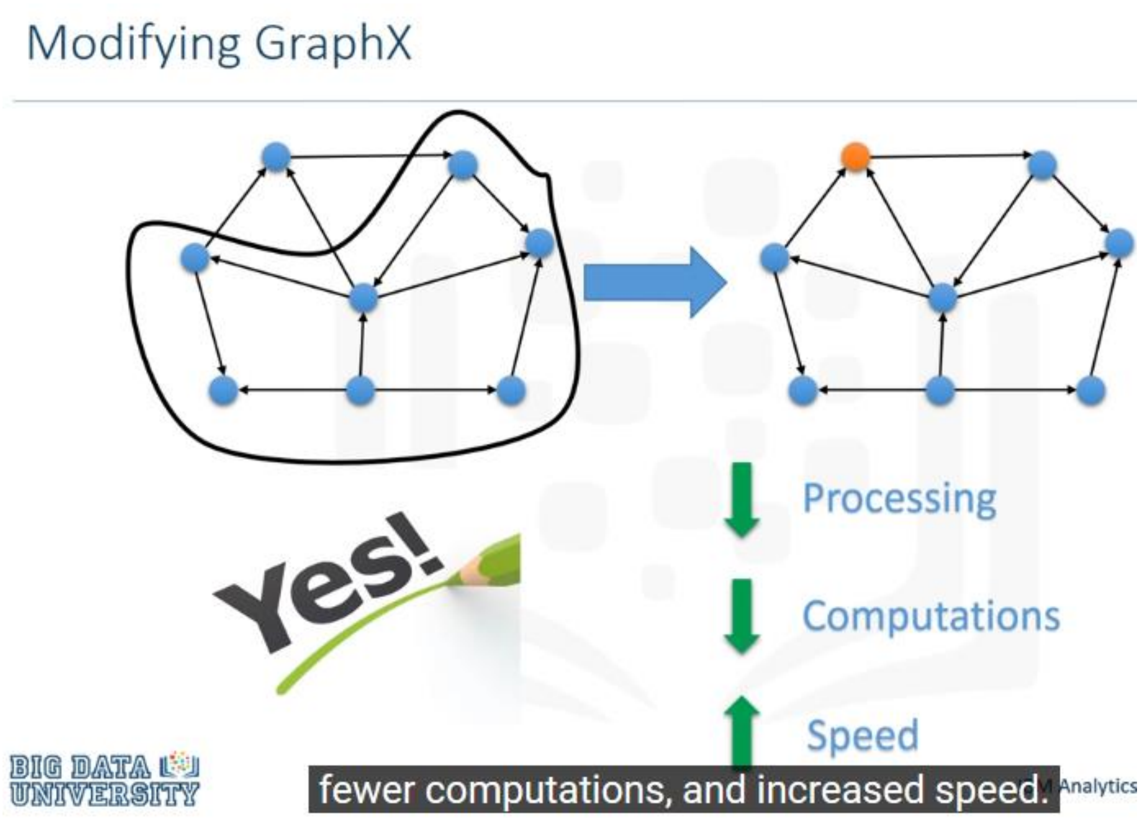
```
val rank = facebook.pageRank(0.1).vertices.collect
```

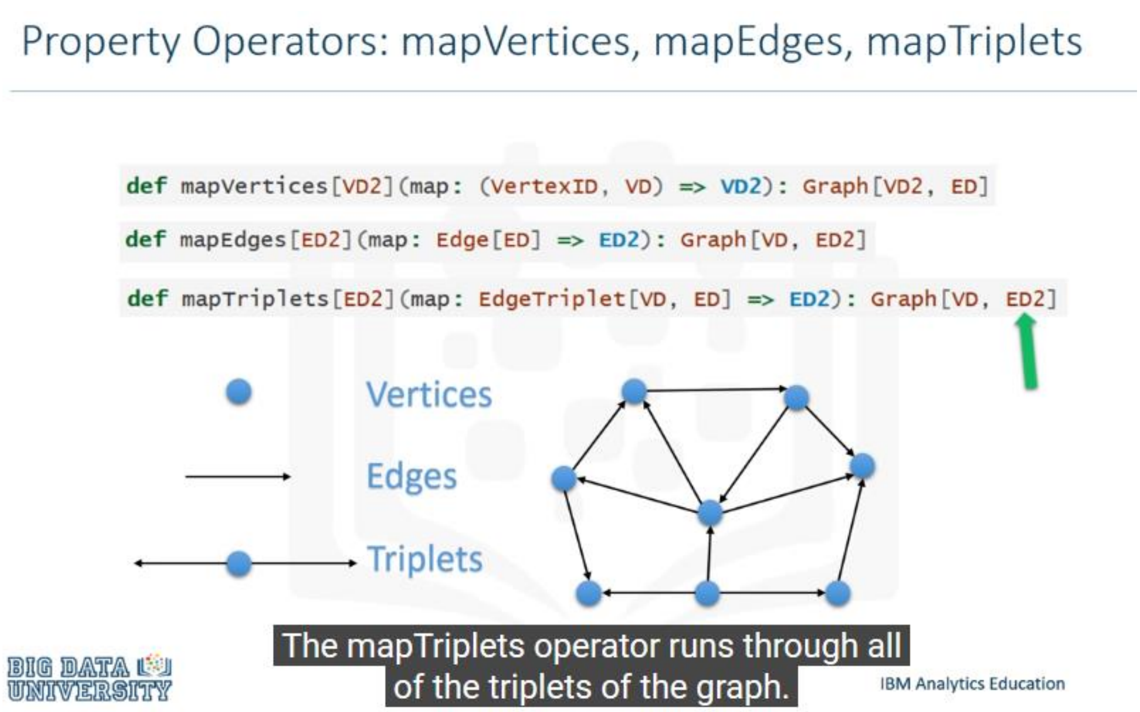
StackTrace:

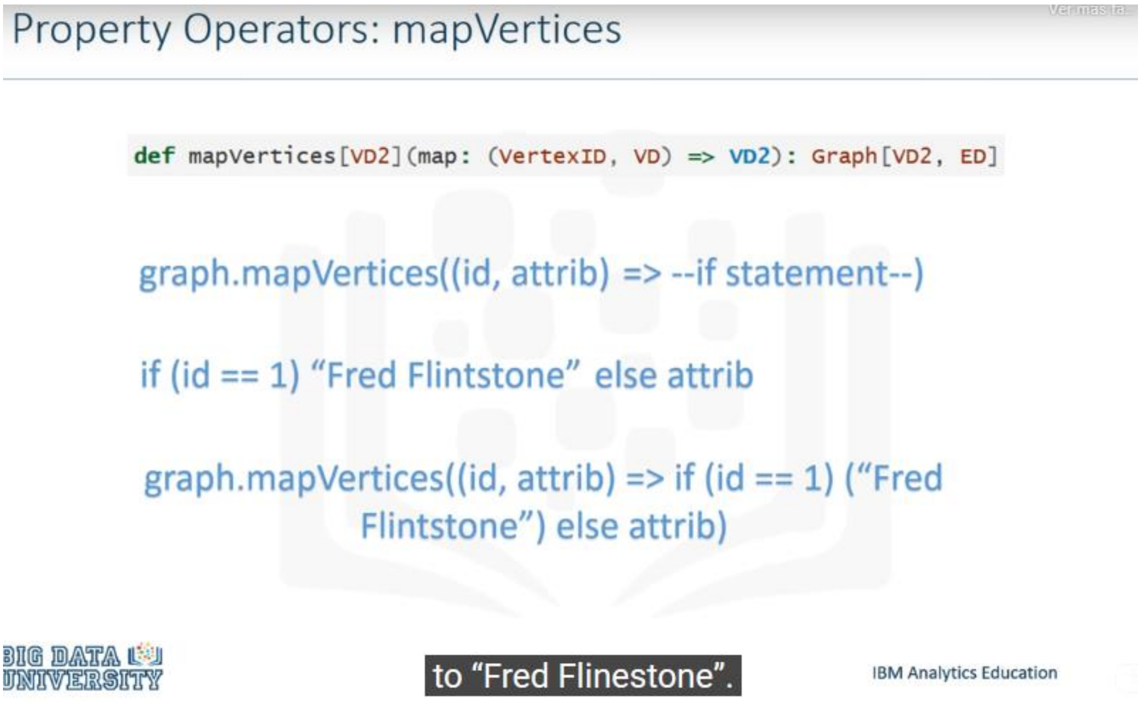
```
// Type your code here
for (rankee <- rank) {
    println(rankee)
}
```

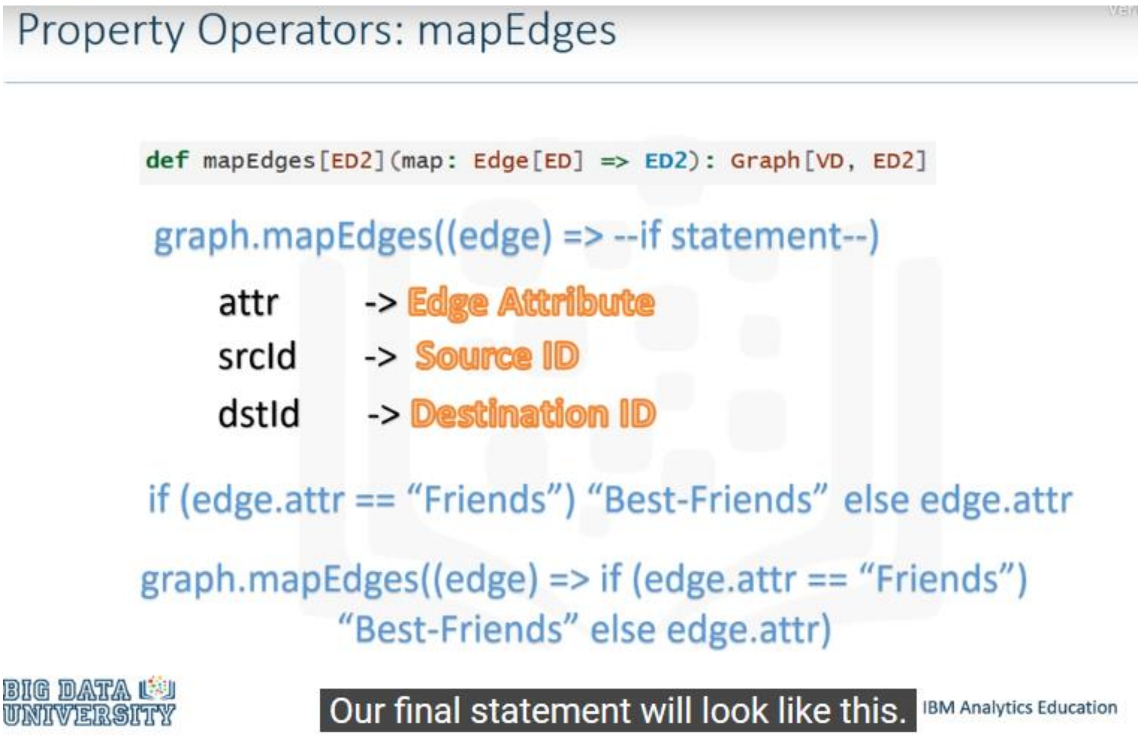
MODULE 3 MODIFYING GRAPHX

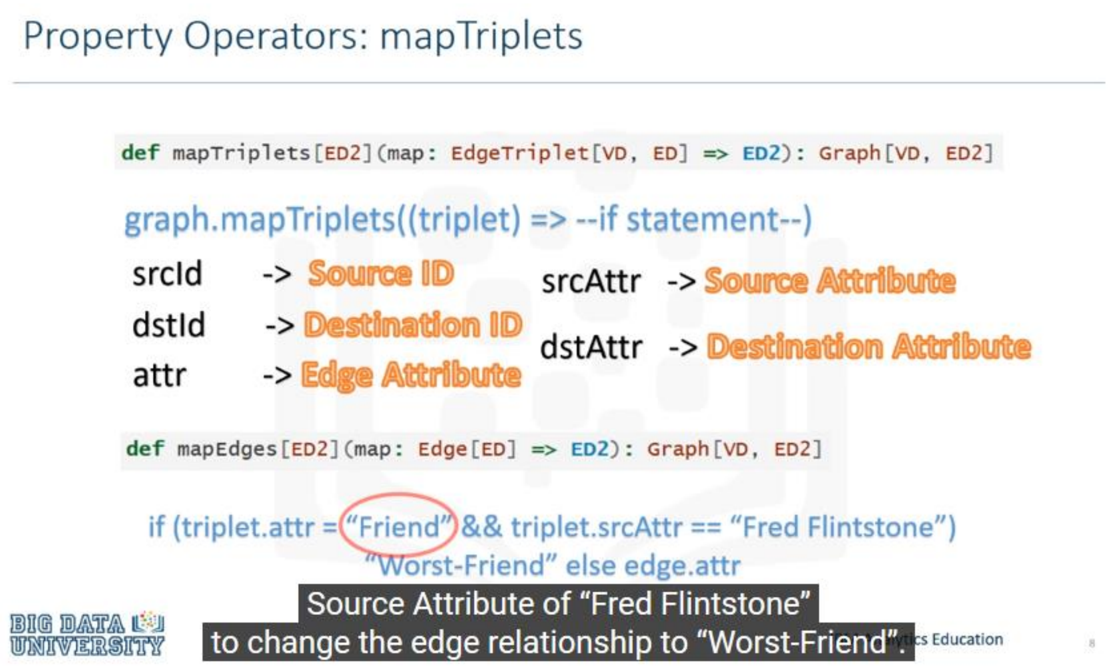



Modifying GraphX
Yes!
Processing
Computations
Speed
BIG DATA UNIVERSITY
fewer computations, and increased speed.
Analytics


Property Operators: mapVertices, mapEdges, mapTriplets
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]
Vertices
Edges
Triplets
BIG DATA UNIVERSITY
The mapTriplets operator runs through all
of the triplets of the graph.
IBM Analytics Education


Property Operators: mapVertices
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
graph.mapVertices((id, attrib) => --if statement--)
if (id == 1) “Fred Flintstone” else attrib
graph.mapVertices((id, attrib) => if (id == 1) (“Fred Flintstone”) else attrib)
BIG DATA UNIVERSITY
to “Fred Flinestone”.
IBM Analytics Education


Property Operators: mapEdges
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
graph.mapEdges((edge) => --if statement--)
attr -> Edge Attribute
srcId -> Source ID
dstId -> Destination ID
if (edge.attr == “Friends”) “Best-Friends” else edge.attr
graph.mapEdges((edge) => if (edge.attr == “Friends”) “Best-Friends” else edge.attr)
BIG DATA UNIVERSITY
Our final statement will look like this.
IBM Analytics Education



Courseware , current location
Course Info
Discussion
Wiki
Resources
Progress

About this course
Module 1 - Introduction to Graph-Parallel
Module 2 - Visualizing GraphX and Exploring Graph Operators
Module 3 - Modifying GraphX

Learning Objectives

Modifying GraphX - Property Operators (3:45) current section

Modifying GraphX - Structural Operators (4:41)

Lab

Graded Review Questions

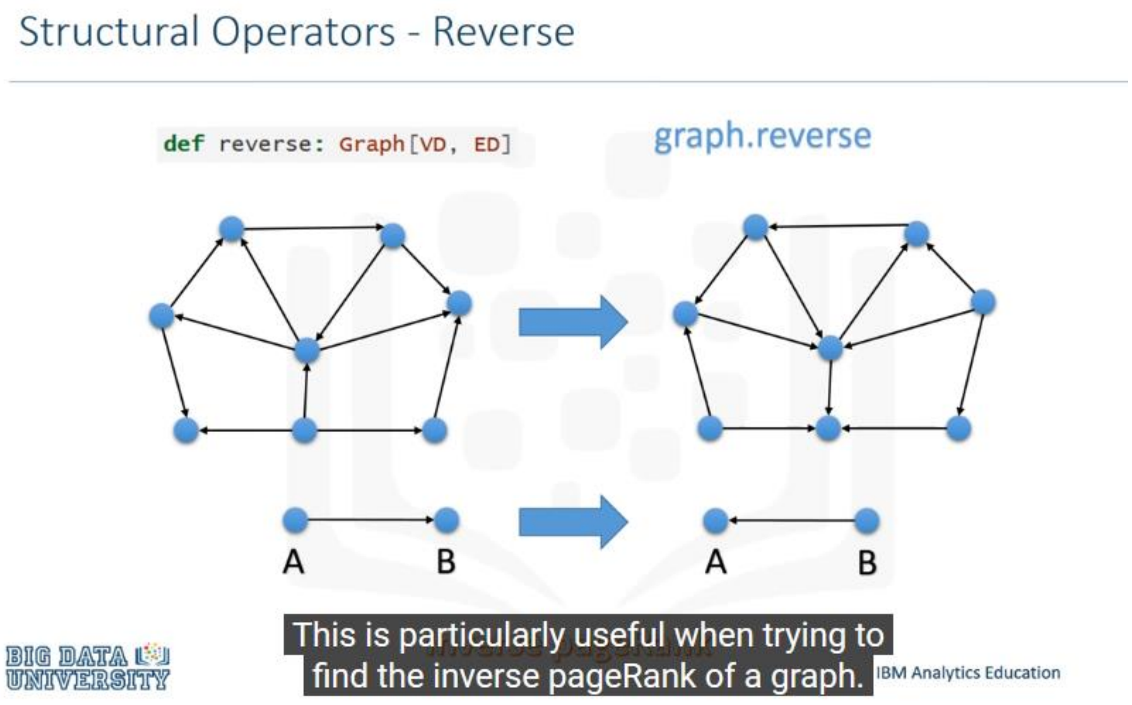
Review Questions This content is graded
Module 4 - Neighborhood Aggregation and Caching
Final Exam
Completion Certificate
Course Survey and Feedback

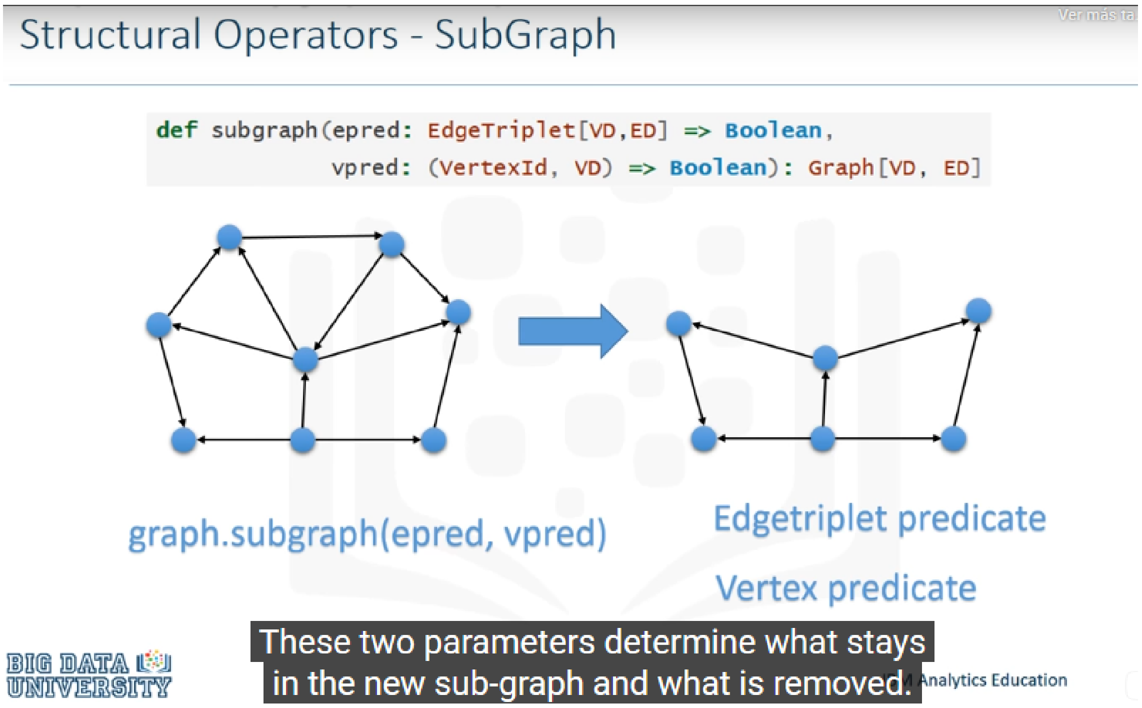
Modifying GraphX - Property Operators (3:45)
Skip to a navigable version of this video's transcript.

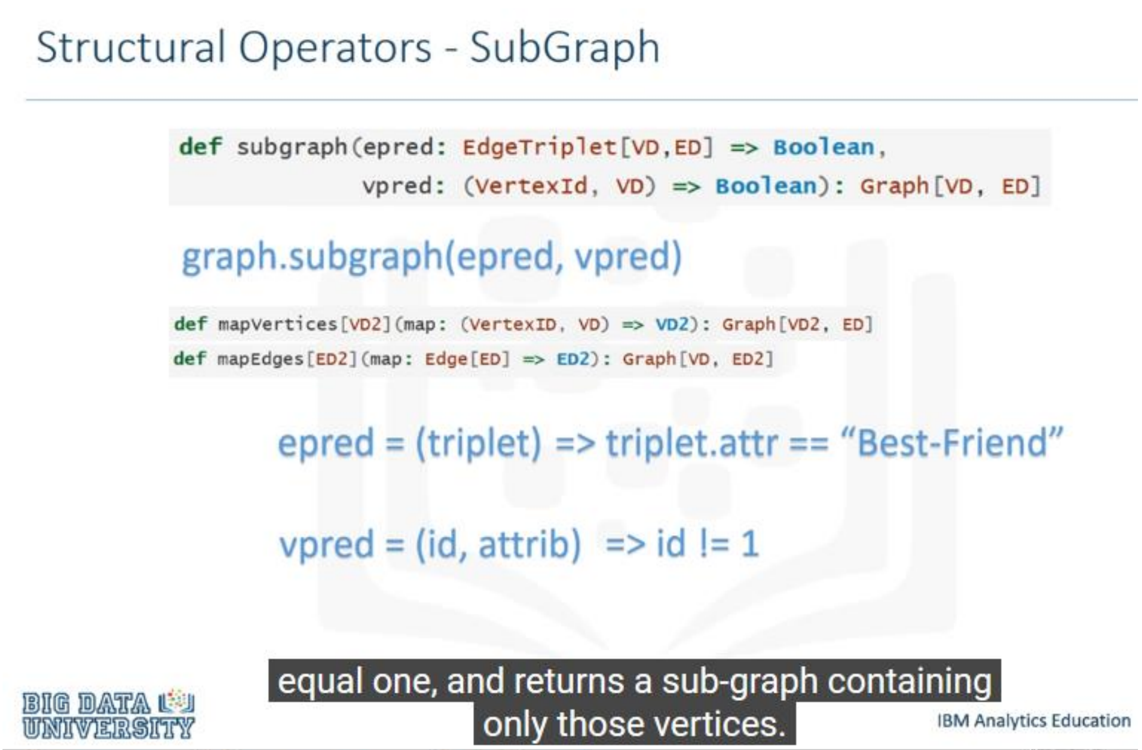
3:38 / 3:46
Maximum Volume.
Skip to end of transcript.

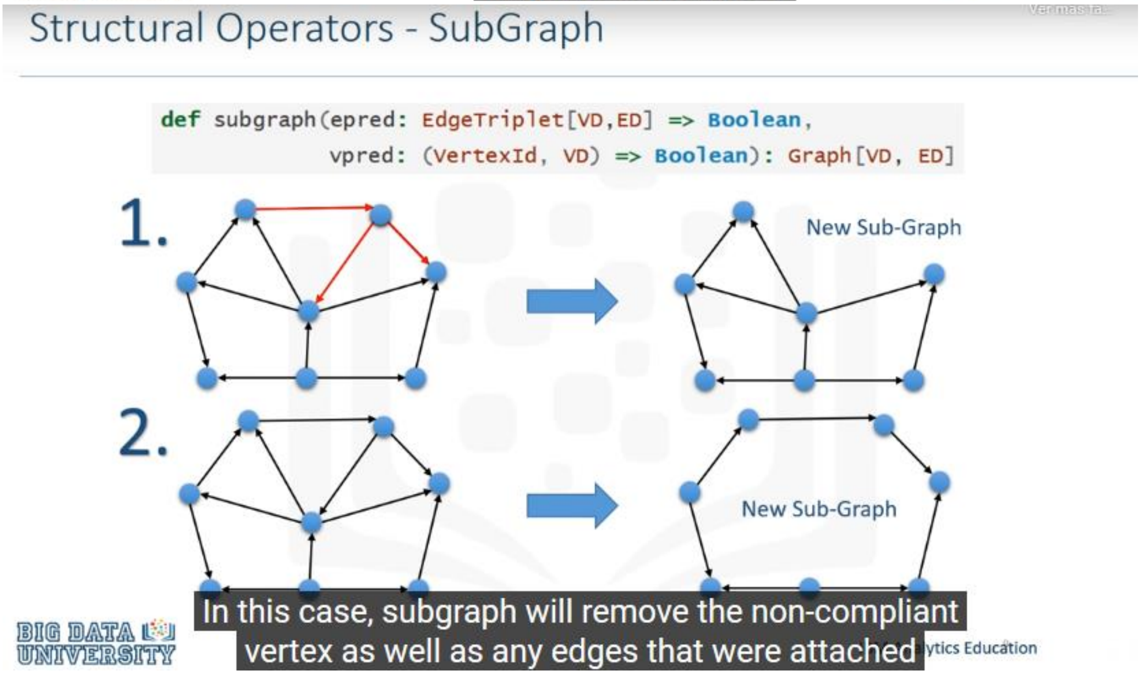
Hello, and welcome.
My name is Deborah.
In this lesson we will look at Modifying GraphX using Property Operators.
In another lesson, we noted that GraphX extends RDD's, or Resilient Distributed Datasets.
Rdd's are immutable, distributed collections of objects.
We get all the benefits of the format, functionality, and speed associated with RDD's.
However, if they are immutable, can GraphX accommodate changes to a graph?
Yes, of course!
GraphX actually creates a new graph for every change it encounters!
However, it doesn't start from scratch.
GraphX reuses structural indices that are unaffected by changes.
This allows GraphX to handle these changes efficiently, resulting in reduced processing, fewer computations, and increased speed.
Now let's look at the Property Operators, mapVertices, mapEdges, and mapTriplets.
These map functions are used to modify the properties of the graph.
For example, mapVertices runs through all of vertices in the graph and returns a new graph with modified vertex attributes.
Similarly, mapEdges runs through all of the edges in the graph and return a new graph with modified edge attributes.
The mapTriplets operator runs through all of the triplets of the graph.
While it can return a graph with modified edge attributes, it also provides access to vertex attributes.
To implement the mapVertices function, we need to define a map function.
To do this, we define a variable to represent the vertex ID and the vertex attribute.
Here we simply use "id" and "attrib".
This is followed by an "if" statement that cycles the modification.
In this "if" statement, we selected a vertex with an id of 1, and change its attribute to "Fred Flintstone".
We must follow the "if" statement with an "else" statement.
Otherwise, the attributes of all other vertices will be blank.
An "else" statement of "attrib" will leave all other vertex attributes unchanged.
The final line of code, shown here, changes the attribute of a vertex with an ID of 1 to "Fred Flinestone".
It will not change the attribute of any other vertex.
Similarly, we implement the mapEdges function by first defining a map function.
The map function requires one variable to represent the edge.
This is again followed by an "if" statement to cycle our modifications.
Keep in mind that we have access to not only the edge attribute, but also to the Source Id and the Destination Id.
In this example, we wish to select all edges with an attribute of "Friends", and then change that attribute to "Best-Friends".
We again include an "else" statement of "edge.attr" to leave the attributes of all other edges unchanged.
Our final statement will look like this.
Finally, we have the mapTriplets function, for which we also define a map function.
Like the mapEdges function, we just need one variable which represents a triplet, followed again by an "if" statement to cycle our modifications.
As with the mapEdge function, we have access to all of the attributes contained in the

triplet.
Interestingly, mapTriplets can change the edge attribute, just as mapEdges can.
However, mapTriplets also gives us access to the attributes of the vertices contained in the triplet, and allows us to specify changes to those as well.
In this example, our "if" statement finds an edge relationship of "Friend" and a Source Attribute of "Fred Flintstone" to change the edge relationship to "Worst-Friend".
Again, notice the "else" statement at the end which keeps all other edges unchanged.
Thank you for watching this lesson!



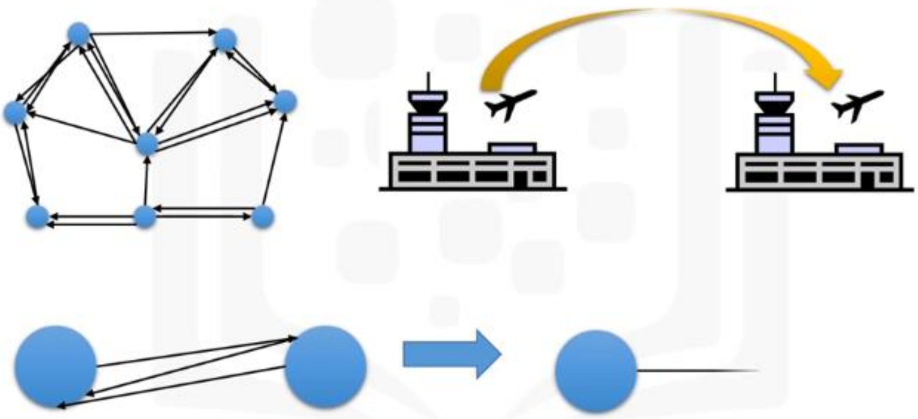



Structural Operators - SubGraph
def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
graph.subgraph(epred, vpred)
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
epred = (triplet) => triplet.attr == "Best-Friend"
vpred = (id, attrib) => id != 1
equal one, and returns a sub-graph containing
only those vertices.
BIG DATA UNIVERSITY
IBM Analytics Education


Structural Operators - SubGraph
def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
1.
New Sub-Graph
2.
New Sub-Graph
In this case, subgraph will remove the non-compliant
vertex as well as any edges that were attached
BIG DATA UNIVERSITY

Structural Operators - groupEdges

```
def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
```

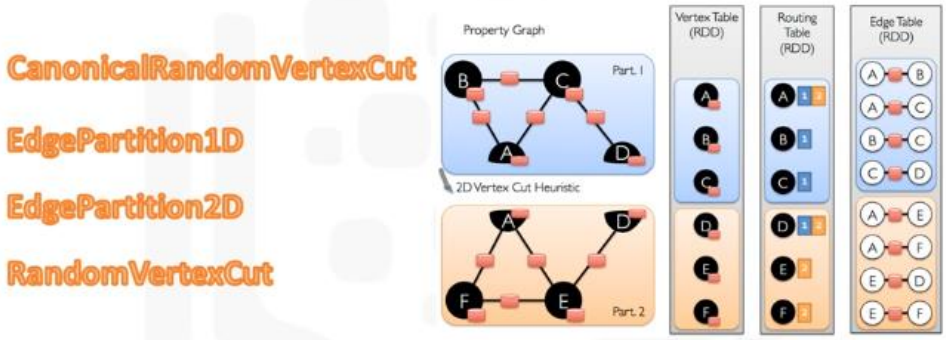


GroupEdges organizes repetitive or similar data by grouping or merging parallel edges

BIG DATA UNIVERSITY ... Analytics Education

Structural Operators - PartitionBy

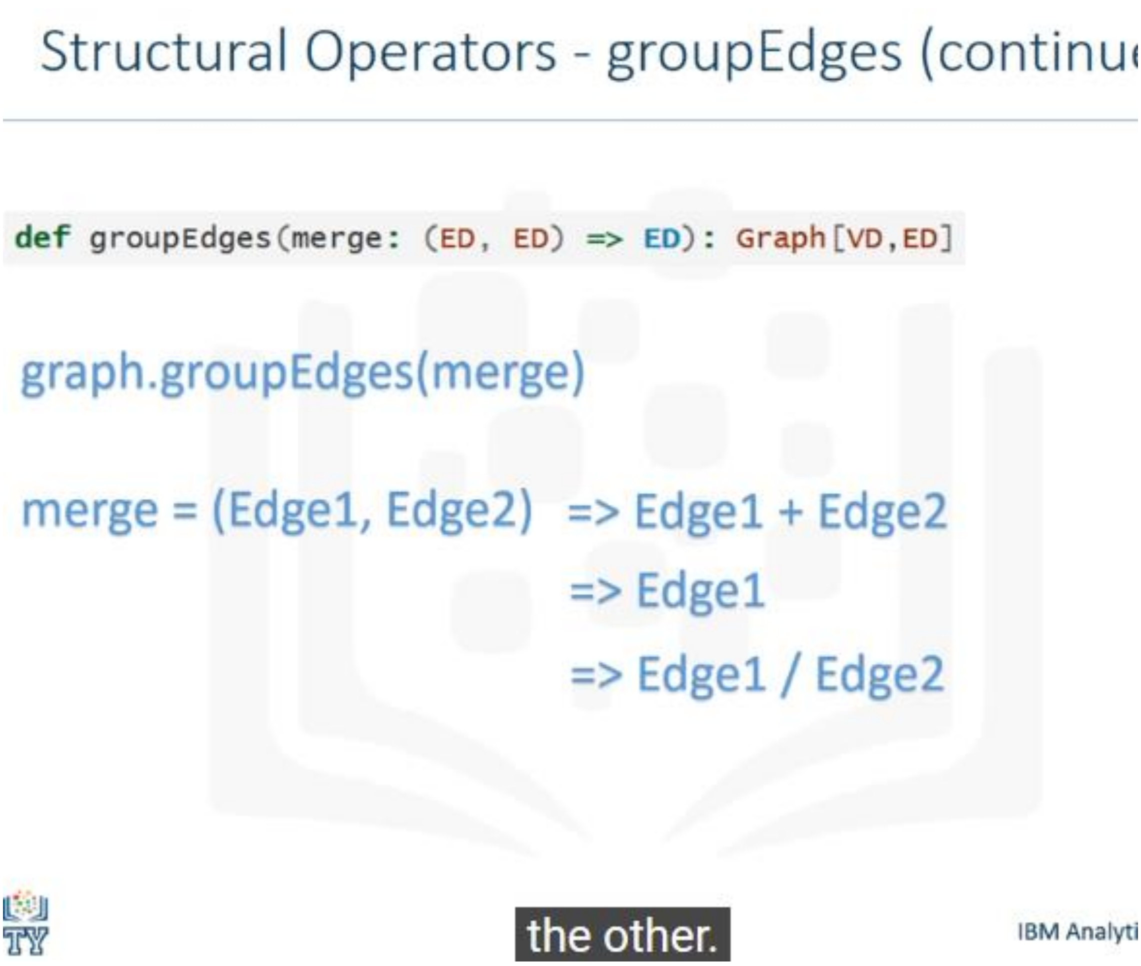
```
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
```



graph.partitionBy(PartitionStrategy.EdgePartition1D)

This line of code runs the partitionBy function with EdgePartition1D as the partition strategy.

BIG DATA UNIVERSITY ... Analytics Education

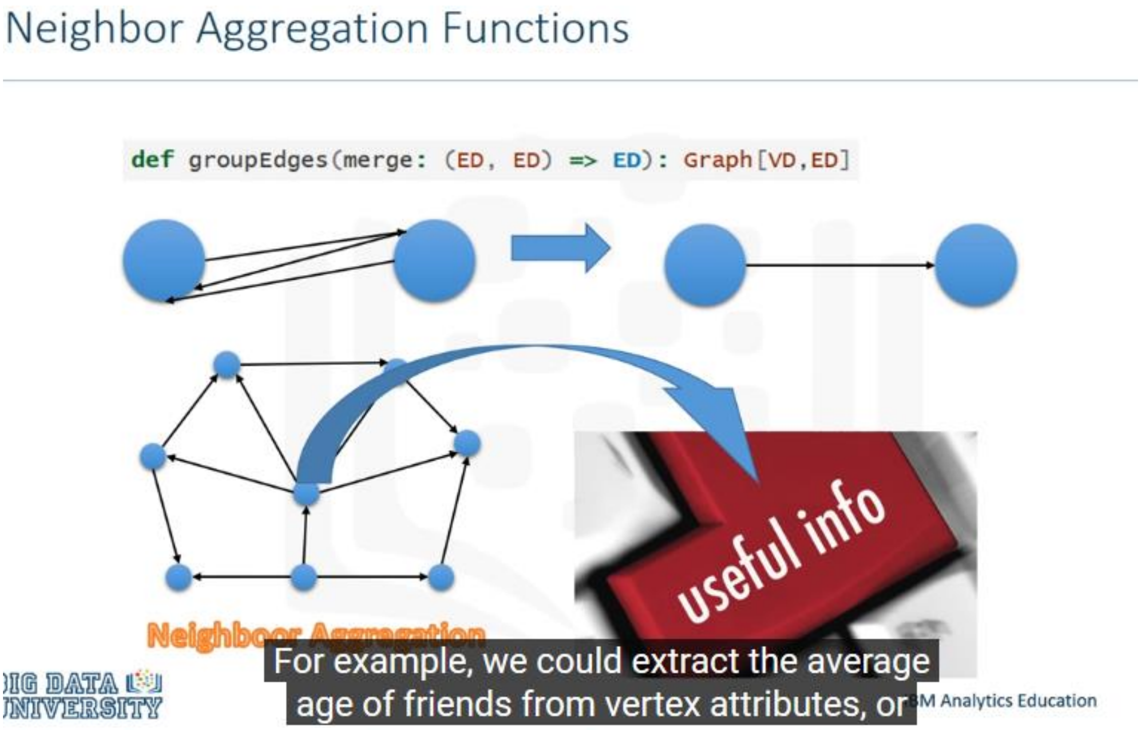


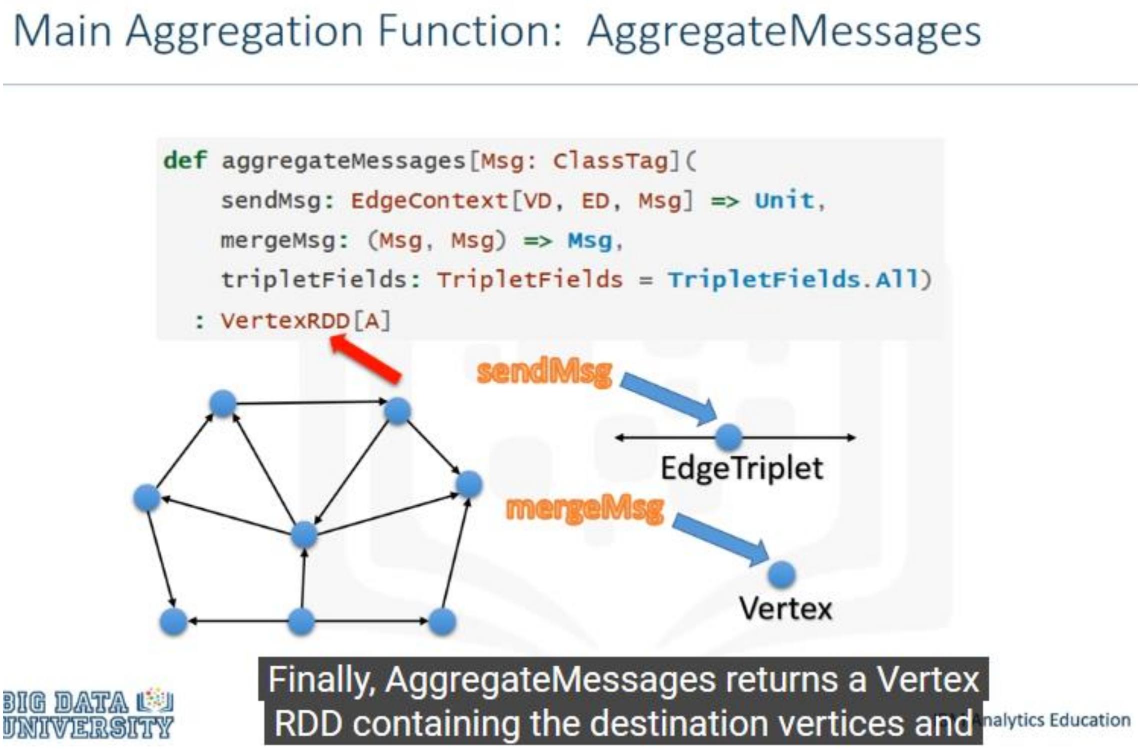
Hello, and welcome.
My name is Deborah.
In this lesson we will look at Modifying GraphX using Structural Operators.
In another lesson, we noted that GraphX extends Resilient Distributed Datasets, which are immutable, distributed collections of objects.
We get the benefits of the format, functionality, and speed of RDD’s.
But if they are immutable, can GraphX accommodate changes to a graph?
Yes!
GraphX creates a new graph for every change it encounters!
Rather than starting from scratch, GraphX reuses structural indices that are unaffected by changes.
This allows GraphX to handle changes efficiently, resulting in reduced processing, fewer computations,
and increased speed.
Property Operators change the attributes of the graph.
In this lesson, we will focus on Structural Operators which alter the structure of a graph.
The first Structural Operator we will look at is the reverse operator.
Reverse returns a new graph in which all of the edge directions are reversed.
For example if an edge in the original graph pointed from vertex A to vertex B, the same edge in the new graph now points from B to A.
This is particularly useful when trying to find the inverse pageRank of a graph.
Next let’s look at the subgraph function.
This function returns a portion of the original graph as a new subgraph.

Subgraph utilizes the parameters epred and vpred.
These are short for edgetriplet predicate and vertex predicate respectively.
These two parameters determine what stays in the new sub-graph and what is removed.
When defining the EdgeTriplet and Vertex predicates, we define variables for the respective class
variables, then give a Boolean expression for the edges or vertices you want to keep.
This criteria is similar to that for mapEdges and mapVertices.
When defining epred, we set a variable to represent the edge triplet.
Here we use the variable “triplet”.
We then define the Boolean expression that will filter out edge triplets in the graph.
In this example, triplet.attr is set to “Best-Friend”.
The new sub-graph will only retain edge triplets with the attribute “Best-Friend”.
Similarly, we define vpred by setting at least two variables.
These represent the vertex ID and a vertex attribute.
Here we name these “ID” and “attrib”.
Our example Boolean expression filters out all vertices in the graph whose ID does not
equal one, and returns a sub-graph containing only those vertices.
There are a two special scenarios that may arise when using the subgraph function.
Suppose all of the vertices in the graph pass the vertex predicate criteria.
However, all of the edges connected to a particular vertex did not pass.
In this case, subgraph will remove the three edges.
Since the vertex is not attached to any edge, it is removed as well.
In the second scenario, all the edges in the graph pass the edge triplet predicate, but
one vertex, the middle vertex in this example, did not pass.
In this case, subgraph will remove the non-compliant vertex as well as any edges that were attached
to it.
Now let’s look at the groupEdges operator.
A Property graph may have multiple, parallel edges between vertices.
Imagine multiple airlines flying between airports at different times.
GroupEdges organizes repetitive or similar data by grouping or merging parallel edges
into one edge to improve clarity and efficiency.
To better understand this operator, we must first look at the partitionBy function.
PartitionBy returns a graph that is partitioned based on the partitionStrategy it is given.
Popular partitionStrategy choices include: CanonicalRandomVertexCut
EdgePartition1D EdgePartition2D, and
RandomVertexCut.
The partitionBy function must be run on our graph before running the groupEdges function.
If not, we may receive incorrect results.
This line of code runs the partitionBy function with EdgePartition1D as the partition strategy.
After running the partitionBy function on our graph, we can now run groupEdges.
GroupEdges accepts a merge parameter.
To define this parameter, we define a variable to represent each of the parallel edges, in
this case, two.
We then define how the edges will merge.
Options include taking the sum, overwriting one edge or the other, or dividing one by
the other.
The final line of code looks like this.
Thank you for watching this lesson!

>>Lab:

MODULE 4 NEIGHBORHOOD AGGREGATION AND CACHING





AggregateMessages: tripletFields Parameter

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

- TripletFields.All — Default
- TripletFields.Dst
- TripletFields.Src
- TripletFields.EdgeOnly
- TripletFields.None

However, we can improve performance of this function by using the parameter that only

BIG DATA UNIVERSITY — IBM Analytics Education

AggregateMessages: sendMsg Parameter

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

srcId -> Source ID

dstId -> Destination ID

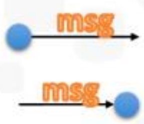
attr -> Edge Attribute

srcAttr -> Source Attribute

dstAttr -> Destination Attribute

sendToDst(msg)

sendToSrc(msg)

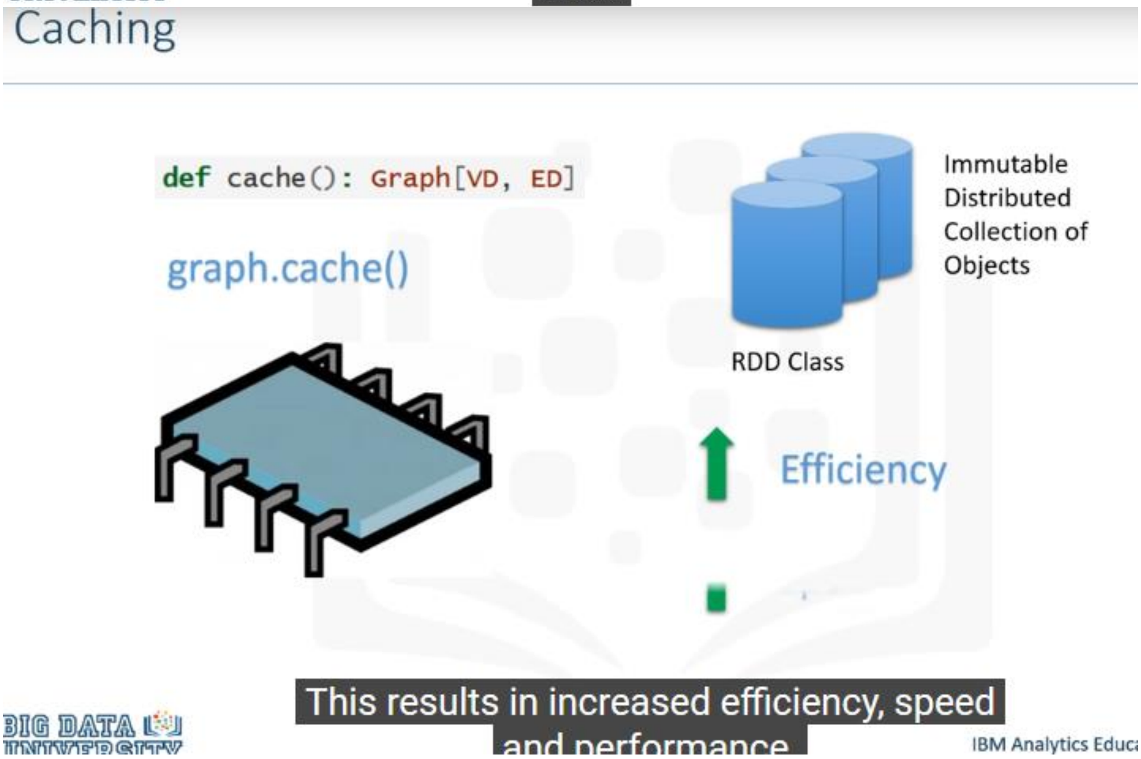


sendMsg = edge_context => edge_context.sendToDst(100)

In this example, edge context sends 100 to every destination vertex.

BIG DATA UNIVERSITY — IBM Analytics Education


AggregateMessages: Bringing it Together
def aggregateMessages[Msg: ClassTag](
sendMsg: EdgeContext[VD, ED, Msg] => Unit,
mergeMsg: (Msg, Msg) => Msg,
tripletFields: TripletFields = TripletFields.All)
: VertexRDD[A]
sendMsg = edge_context => edge_context.sendToDst(100)
mergeMsg = (Msg1, Msg2) => Msg1 + Msg2
graph.aggregateMessages[Int](edge_context => edge_context.sendToDst(100), (Msg1, Msg2) = Msg1 +Msg2, TripletFields.Dst)
Now let's combine them into one line of code.
BIG DATA UNIVERSITY
IBM Analytics Education


Caching
def cache(): Graph[VD, ED]
graph.cache()
Immutable Distributed Collection of Objects
RDD Class
Efficiency
This results in increased efficiency, speed
BIG DATA
IBM Analytics Educ

MapReduceTriplets

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

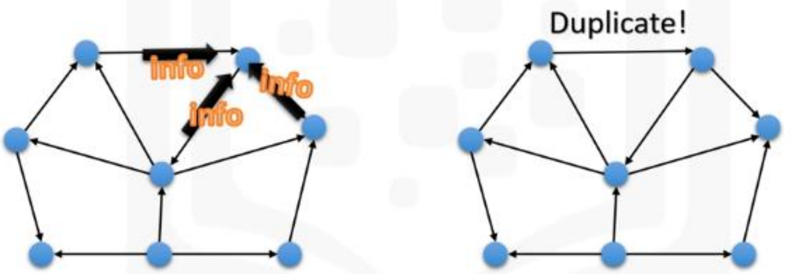
```
def mapReduceTriplets[Msg](
    map: EdgeTriplet[VD, ED] => Iterator[(VertexId, Msg)],
    reduce: (Msg, Msg) => Msg)
  : VertexRDD[Msg]
```

In the end, however, tripletFields allow aggregateMessages to run faster.

BIG DATA UNIVERSITY IBM Analytics Education

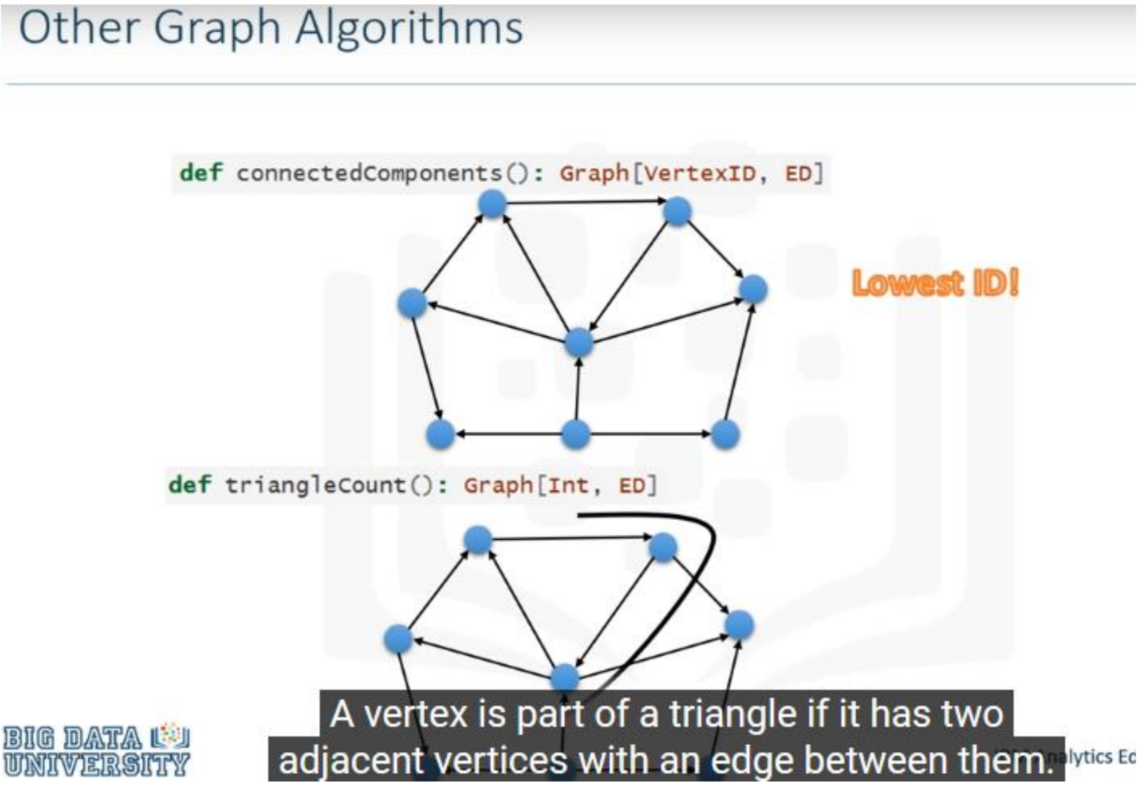
CollectNeighbors

```
class GraphOps[VD, ED] {
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[ Array[(VertexId, VD)] ]
}
```



We will not review them here, but remember that they exist, and to use aggregatesMessages

BIG DATA UNIVERSITY IBM Analytics Education



Hello, and welcome.
My name is Deborah.
In this lesson we will look at Neighborhood Aggregation and Caching in GraphX.
In another lesson we explored simple aggregation methods like the groupEdges function, which
merges parallel edges.
In this lesson we focus on Neighborhood Aggregation.
These methods allow us to pull useful information from a graph.
For example, we could extract the average age of friends from vertex attributes, or
the percentage of friends that are in a certain age group.
PageRank and degree functions use neighborhood aggregation.
The main aggregation function in GraphX is AggregateMessages.
AggregateMessage utilizes many parameters, making it a bit complex, but very customizable and useful.
AggregateMessages works on a graph by applying a user-defined send-message function to each
edge triplet in the graph.
Then, a user-defined merge message function aggregates or collects the messages from the destination vertices.
Finally, AggregateMessages returns a Vertex RDD containing the destination vertices and their merged messages.
Any vertices that did not receive a message are not included in the RDD.
Within AggregateMessages is a parameter called tripletFields.
This parameter indicates what fields of a Triplet are to be accessed.
For example: TripletFields.All accesses all the fields
TripletFields.Dst accesses just the destination and edge fields
TripletFields.Src accesses just the source field
TripletFields.EdgeOnly accesses only the edge field
TripletFields.None accesses none of the fields If not specified, tripletField will default

to TripletFields.All.
However, we can improve performance of this function by using the parameter that only returns what we need, since GraphX will be able to select an optimized join strategy.
To define the sendMsg parameter, we must first define a variable for the EdgeContext class.
An EdgeContext is similar to the EdgeTriplet that we studied in another lesson.
We can therefore use its attributes as show here.
In addition, we will introduce two new functions in just a minute.
For the variable we'll pick edge_context.
Now we define the message to be sent.
We use EdgeContext's two new functions for this.
They are sendToDst and sendToSrc.
These functions accept a message parameter which is sent to the destination or source vertex respectively.
In this example, edge context sends 100 to every destination vertex.
The message is then converged by using mergeMsg, which we'll see next.
We saw a parameter called groupEdges in another lesson.
The mergeMsg parameter is defined in a similar way.
We first define variables that represent the messages we wish to merge.
For example, let's use Msg1 and Msg2 as our variables.
Next we define how we want the messages to merge.
As with groupEdges, we can define the merge in several ways as shown here.
Now let's put it all together.
We created the sendMsg and mergeMsg parameters.
Now let's combine them into one line of code.
Note how we defined the ClassTag of Msg in square brackets, and how we only denoted the destination field under the tripletFields parameter.
AggregateMessages is quite a complex function!
Let's lower the complexity a bit by exploring the Cache function.
We call this important function before using our graph to cache our graph into memory.
Since by default RDD's do not persist in memory, utilizing memory allows GraphX to avoid re-computation.
This results in increased efficiency, speed and performance.
Prior to aggregateMessages, mapReduceTriplets dominated GraphX's neighborhood aggregation.
The two are similar in function, but with some name differences.
We see iterator, and EdgeContext versus EdgeTriplet in the map function.
Iterator was abandoned because it inhibited our ability to apply additional optimizations.
In the end, however, tripletFields allow aggregateMessages to run faster.
An alternative to aggregateMessages are functions called collectNeighborsIds and collectNeighbors
from GraphOps.
These collect neighboring vertices and the attributes at each vertex.
However, these functions require substantial communication and tend to duplicate information,
making them rather costly to run.
We will not review them here, but remember that they exist, and to use aggregatesMessages
whenever possible.
Two other functions used by GraphX are connectedComponents and triangleCount.
connectedComponents labels each connected component of the graph with the ID of its lowest-number vertex.
This can be used to approximate clusters in graphs like social networks.

triangleCount finds the number of triangles passing through each vertex.
A vertex is part of a triangle if it has two adjacent vertices with an edge between them.
We will not examine these here, however you are encouraged to explore them on your own.
Thank you for watching!

>>Lab: