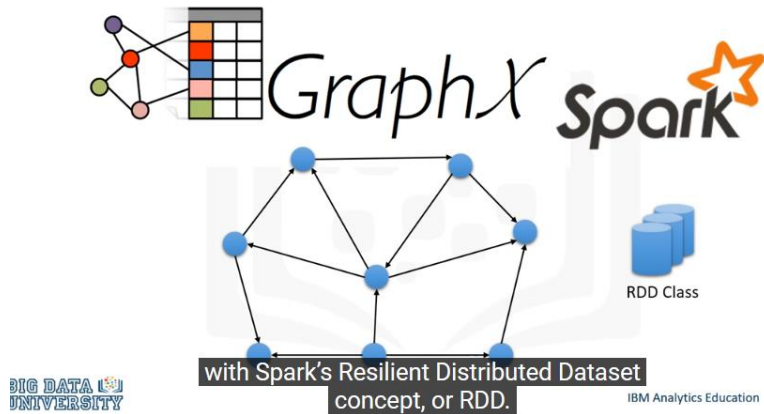
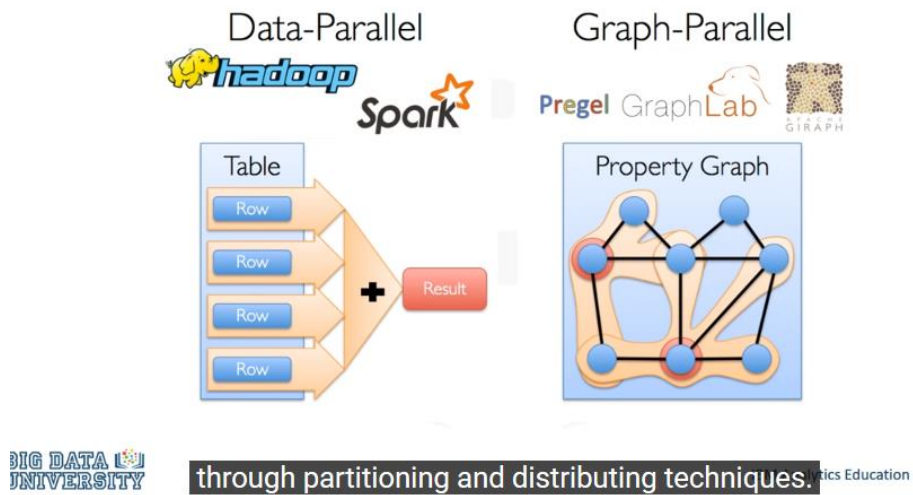


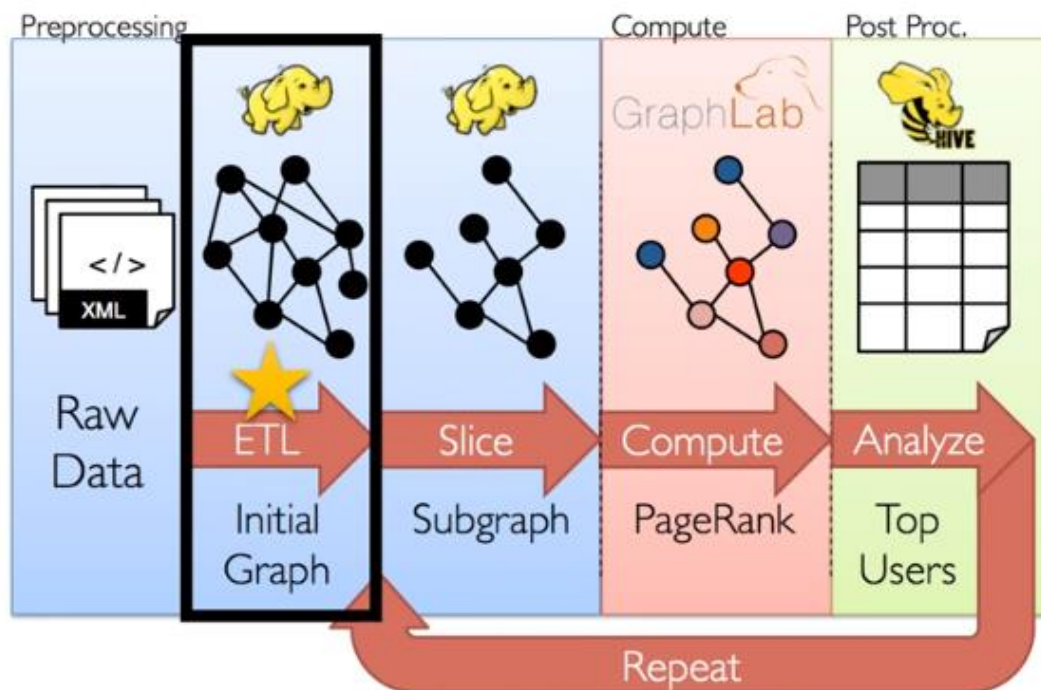
## MODULE 1 INTRODUCTION TO GRAPH-PARALLEL

### What is GraphX?

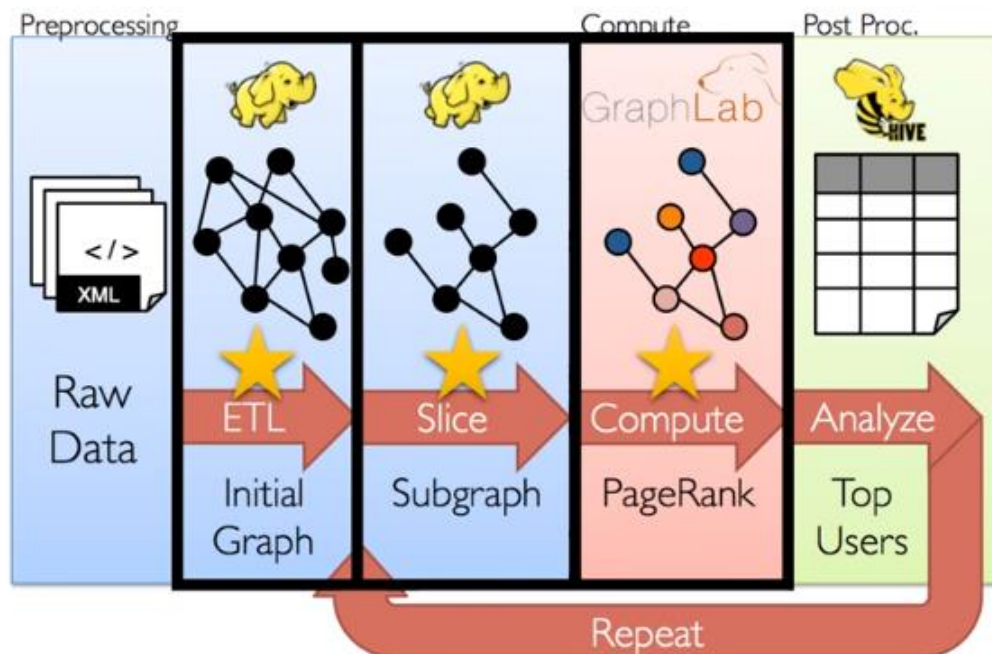


### Data-Parallel vs Graph-Parallel

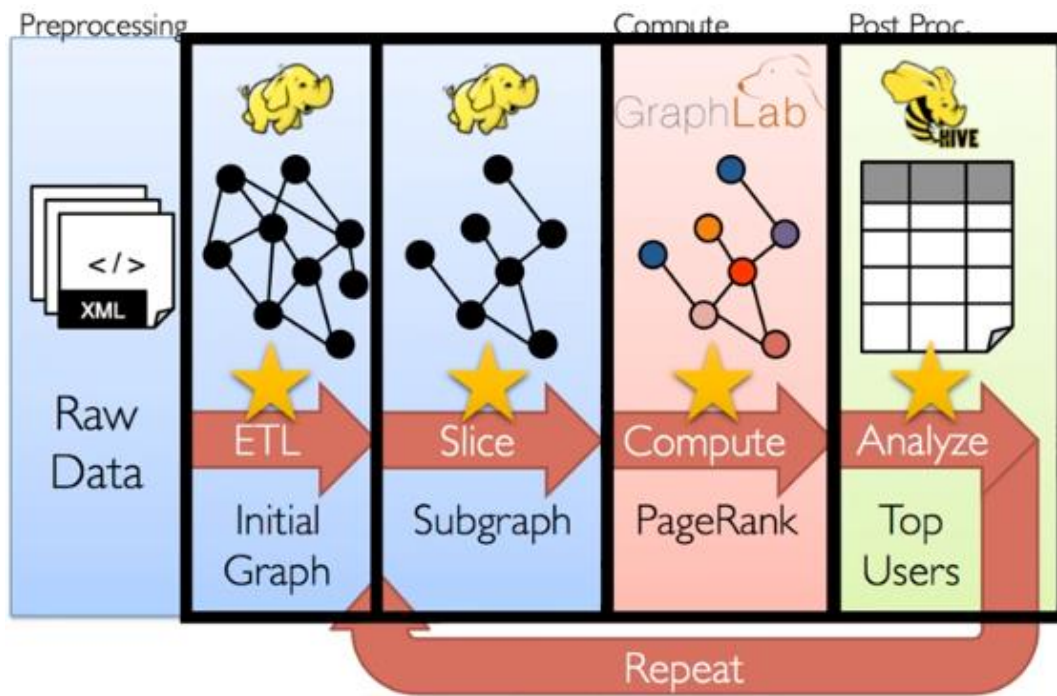




In this generic programming model, Hadoop extracts, transforms, and loads raw data as



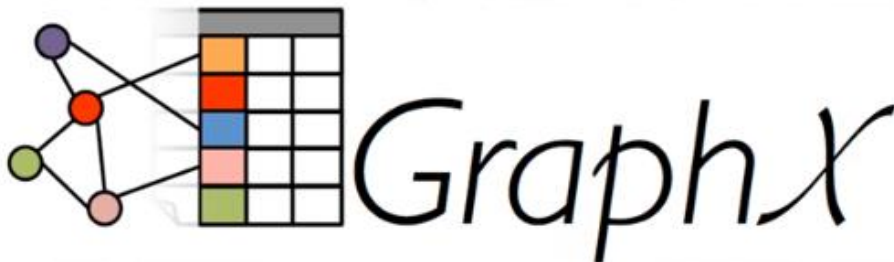
GraphLab then uses an algorithm such as PageRank for computation.



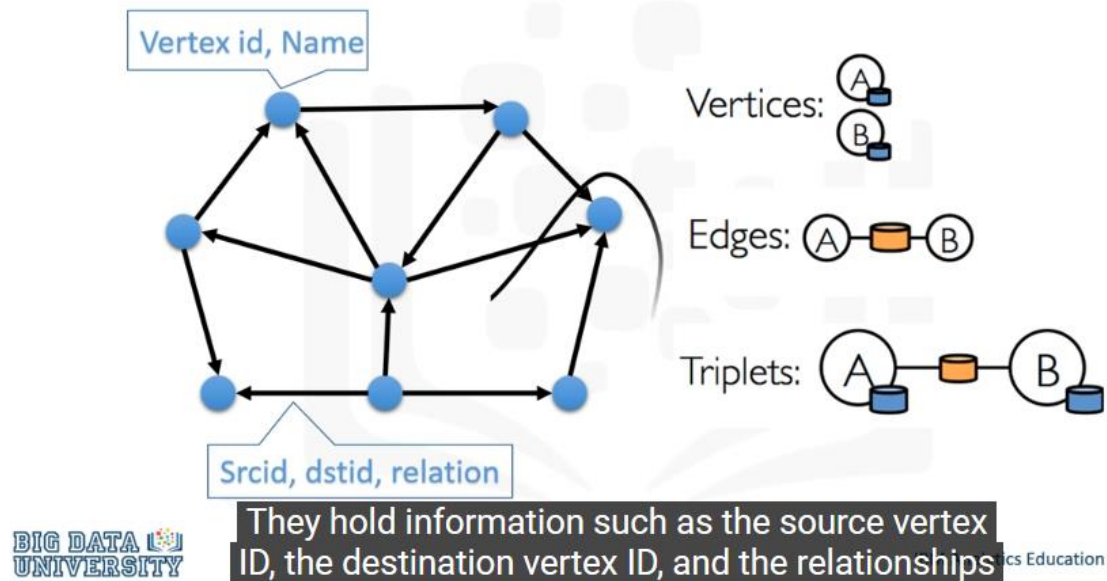
But the process of changing systems and repetition may lead to accuracy problems and inefficiencies.

Data-Parallel

Graph-Parallel



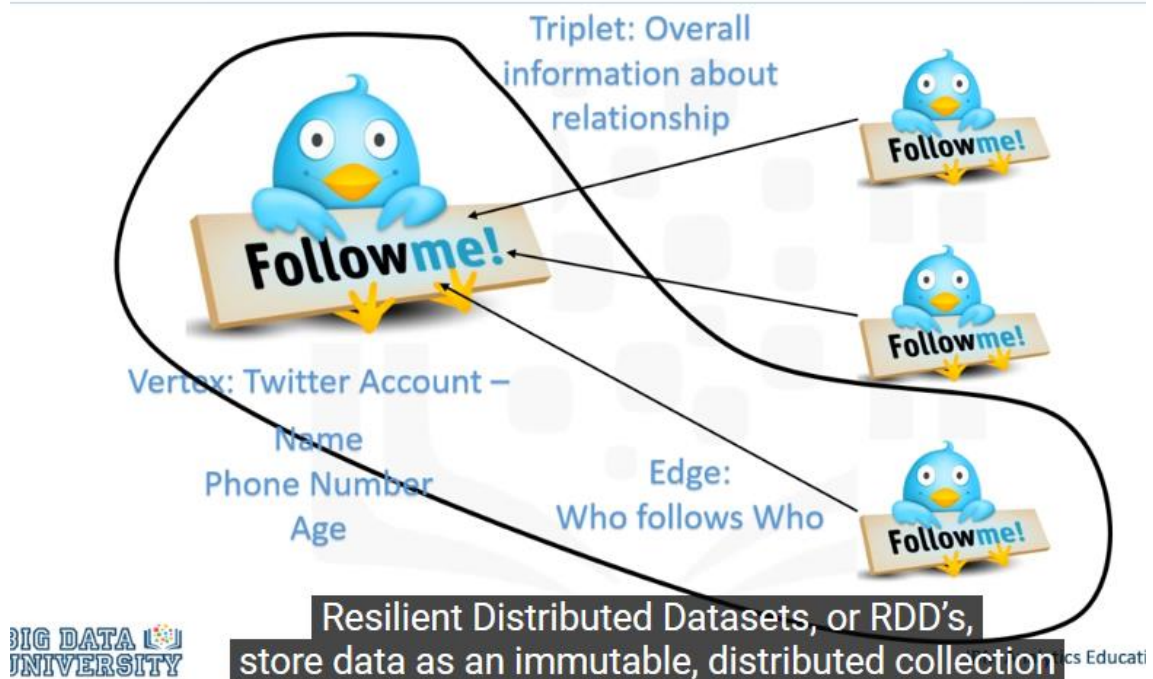
## Components of GraphX (Property Graph)



BIG DATA UNIVERSITY

ics Education

## Components of GraphX

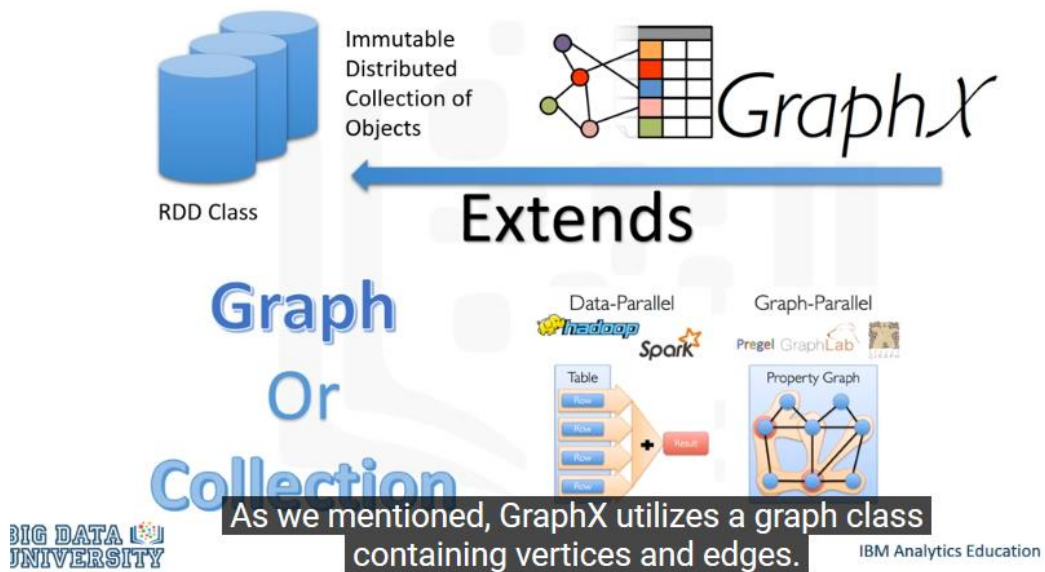


BIG DATA UNIVERSITY

ics Educati



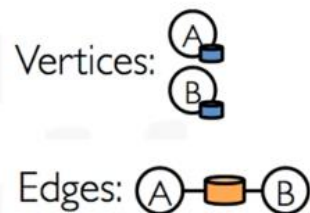
## Review of RDDs



## Constructing a Graph

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

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



## 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

BIG DATA UNIVERSITY

An Edge class contains the source id, destination  
id, and edge attribute.

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"))

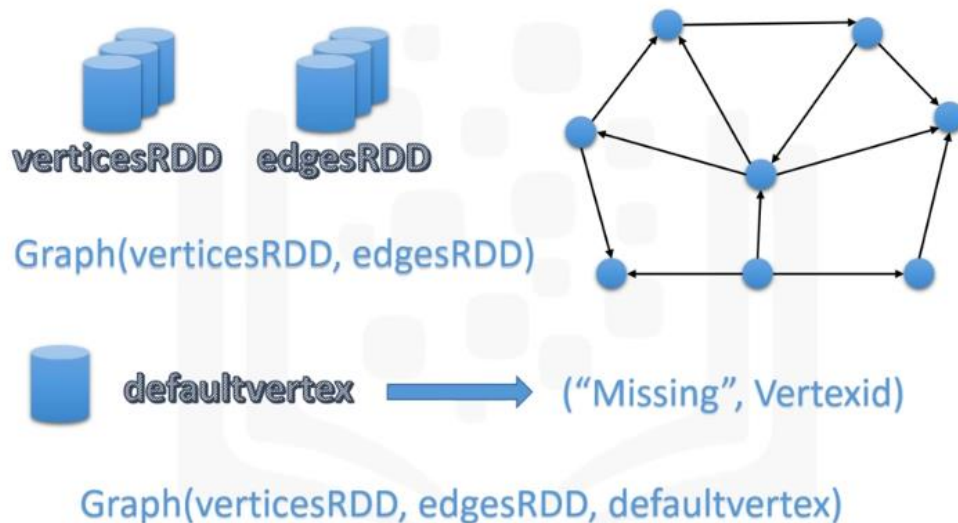
SparkContext.parallelize(Array(edges))

BIG DATA UNIVERSITY

We'll call them verticesRDD and edgesRDD.

Analytics Education

## How to construct a Graph



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.

Default tvertex does not need to be an RDD.



## Spark GraphX (Cognitive Class)

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 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 America Fan Page", "Page")))

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 defaultly connected to any edges that lead to a non-existent Vertex. Let’s call it “Self” - since you can be friends with “Yourself” and have a page that follows “Itself.”

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

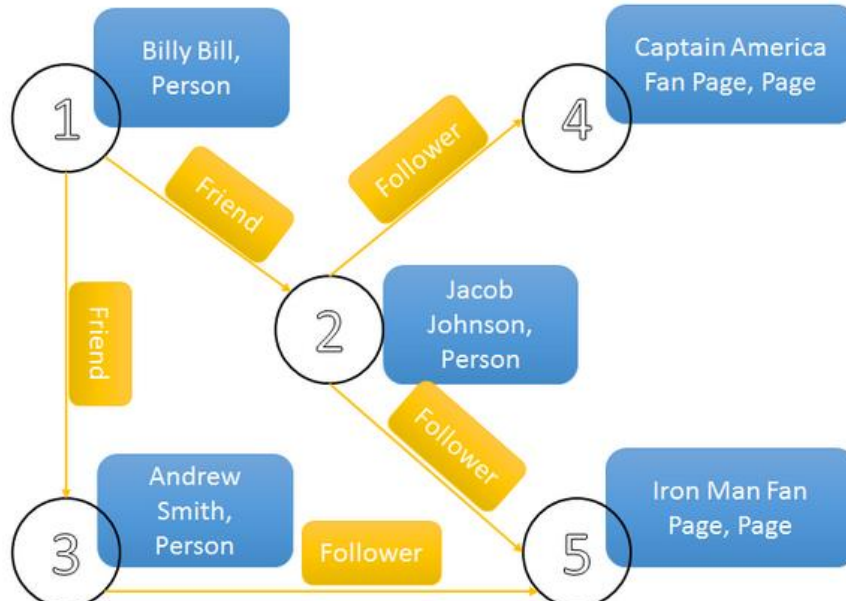
defaultVertex = (Self,Missing)
```

## Spark GraphX (Cognitive Class)

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@54d63ea
: org.apache.spark.graphx.impl.GraphImpl@54d63ea
```

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 == 4 && relation.attr == "Follower" }.count
2
```

## Spark GraphX (Cognitive Class)

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 ")
  print(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?

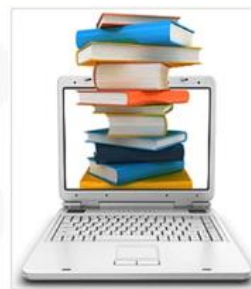


Unfortunately the GraphX library doesn't include built-in visualization tools.

IBM Analytics Education

BIG DATA UNIVERSITY

### GraphX Visualization Alternatives



Gephi

GraphLab

```
# 1 2 3 4 5 6 7 8 9 10 11 12
# 10 11 12 13 14 15 16 17 18 19 20 21
# 20 21 22 23 24 25 26 27 28 29 30 31
# 30 31 32 33 34 35 36 37 38 39 40 41
# 40 41 42 43 44 45 46 47 48 49 50 51
# 50 51 52 53 54 55 56 57 58 59 60 61
# 60 61 62 63 64 65 66 67 68 69 70 71
# 70 71 72 73 74 75 76 77 78 79 80 81
# 80 81 82 83 84 85 86 87 88 89 90 91
# 90 91 92 93 94 95 96 97 98 99 100 101
# 100 101 102 103 104 105 106 107 108 109 110 111
# 110 111 112 113 114 115 116 117 118 119 120 121
# 120 121 122 123 124 125 126 127 128 129 130 131
# 130 131 132 133 134 135 136 137 138 139 140 141
# 140 141 142 143 144 145 146 147 148 149 150 151
# 150 151 152 153 154 155 156 157 158 159 160 161
# 160 161 162 163 164 165 166 167 168 169 170 171
# 170 171 172 173 174 175 176 177 178 179 180 181
# 180 181 182 183 184 185 186 187 188 189 190 191
# 190 191 192 193 194 195 196 197 198 199 200 201
# 200 201 202 203 204 205 206 207 208 209 210 211
# 210 211 212 213 214 215 216 217 218 219 220 221
# 220 221 222 223 224 225 226 227 228 229 230 231
# 230 231 232 233 234 235 236 237 238 239 240 241
# 240 241 242 243 244 245 246 247 248 249 250 251
# 250 251 252 253 254 255 256 257 258 259 260 261
# 260 261 262 263 264 265 266 267 268 269 270 271
# 270 271 272 273 274 275 276 277 278 279 280 281
# 280 281 282 283 284 285 286 287 288 289 290 291
# 290 291 292 293 294 295 296 297 298 299 300 301
# 300 301 302 303 304 305 306 307 308 309 310 311
# 310 311 312 313 314 315 316 317 318 319 320 321
# 320 321 322 323 324 325 326 327 328 329 330 331
# 330 331 332 333 334 335 336 337 338 339 340 341
# 340 341 342 343 344 345 346 347 348 349 350 351
# 350 351 352 353 354 355 356 357 358 359 360 361
# 360 361 362 363 364 365 366 367 368 369 370 371
# 370 371 372 373 374 375 376 377 378 379 380 381
# 380 381 382 383 384 385 386 387 388 389 390 391
# 390 391 392 393 394 395 396 397 398 399 400 401
# 400 401 402 403 404 405 406 407 408 409 410 411
# 410 411 412 413 414 415 416 417 418 419 420 421
# 420 421 422 423 424 425 426 427 428 429 430 431
# 430 431 432 433 434 435 436 437 438 439 440 441
# 440 441 442 443 444 445 446 447 448 449 450 451
# 450 451 452 453 454 455 456 457 458 459 460 461
# 460 461 462 463 464 465 466 467 468 469 470 471
# 470 471 472 473 474 475 476 477 478 479 480 481
# 480 481 482 483 484 485 486 487 488 489 490 491
# 490 491 492 493 494 495 496 497 498 499 500 501
# 500 501 502 503 504 505 506 507 508 509 510 511
# 510 511 512 513 514 515 516 517 518 519 520 521
# 520 521 522 523 524 525 526 527 528 529 530 531
# 530 531 532 533 534 535 536 537 538 539 540 541
# 540 541 542 543 544 545 546 547 548 549 550 551
# 550 551 552 553 554 555 556 557 558 559 560 561
# 560 561 562 563 564 565 566 567 568 569 570 571
# 570 571 572 573 574 575 576 577 578 579 580 581
# 580 581 582 583 584 585 586 587 588 589 590 591
# 590 591 592 593 594 595 596 597 598 599 600 601
# 600 601 602 603 604 605 606 607 608 609 610 611
# 610 611 612 613 614 615 616 617 618 619 620 621
# 620 621 622 623 624 625 626 627 628 629 630 631
# 630 631 632 633 634 635 636 637 638 639 640 641
# 640 641 642 643 644 645 646 647 648 649 650 651
# 650 651 652 653 654 655 656 657 658 659 660 661
# 660 661 662 663 664 665 666 667 668 669 670 671
# 670 671 672 673 674 675 676 677 678 679 680 681
# 680 681 682 683 684 685 686 687 688 689 690 691
# 690 691 692 693 694 695 696 697 698 699 700 701
# 700 701 702 703 704 705 706 707 708 709 710 711
# 710 711 712 713 714 715 716 717 718 719 720 721
# 720 721 722 723 724 725 726 727 728 729 730 731
# 730 731 732 733 734 735 736 737 738 739 740 741
# 740 741 742 743 744 745 746 747 748 749 750 751
# 750 751 752 753 754 755 756 757 758 759 760 761
# 760 761 762 763 764 765 766 767 768 769 770 771
# 770 771 772 773 774 775 776 777 778 779 780 781
# 780 781 782 783 784 785 786 787 788 789 790 791
# 790 791 792 793 794 795 796 797 798 799 800 801
# 800 801 802 803 804 805 806 807 808 809 810 811
# 810 811 812 813 814 815 816 817 818 819 820 821
# 820 821 822 823 824 825 826 827 828 829 830 831
# 830 831 832 833 834 835 836 837 838 839 840 841
# 840 841 842 843 844 845 846 847 848 849 850 851
# 850 851 852 853 854 855 856 857 858 859 860 861
# 860 861 862 863 864 865 866 867 868 869 870 871
# 870 871 872 873 874 875 876 877 878 879 880 881
# 880 881 882 883 884 885 886 887 888 889 890 891
# 890 891 892 893 894 895 896 897 898 899 900 901
# 900 901 902 903 904 905 906 907 908 909 910 911
# 910 911 912 913 914 915 916 917 918 919 920 921
# 920 921 922 923 924 925 926 927 928 929 930 931
# 930 931 932 933 934 935 936 937 938 939 940 941
# 940 941 942 943 944 945 946 947 948 949 950 951
# 950 951 952 953 954 955 956 957 958 959 960 961
# 960 961 962 963 964 965 966 967 968 969 970 971
# 970 971 972 973 974 975 976 977 978 979 980 981
# 980 981 982 983 984 985 986 987 988 989 990 991
# 990 991 992 993 994 995 996 997 998 999 1000 1001
# 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011
# 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021
# 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031
# 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041
# 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051
# 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061
# 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071
# 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081
# 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091
# 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101
# 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111
# 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121
# 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131
# 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141
# 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151
# 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161
# 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171
# 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181
# 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191
# 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201
# 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211
# 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221
# 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231
# 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241
# 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251
# 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261
# 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271
# 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281
# 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291
# 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301
# 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311
# 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321
# 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331
# 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341
# 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351
# 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361
# 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371
# 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381
# 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391
# 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401
# 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411
# 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421
# 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431
# 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441
# 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451
# 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461
# 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471
# 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481
# 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491
# 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501
# 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511
# 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521
# 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531
# 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541
# 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551
# 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561
# 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571
# 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581
# 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591
# 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601
# 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611
# 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621
# 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631
# 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641
# 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651
# 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661
# 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671
# 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681
# 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691
# 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701
# 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711
# 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721
# 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731
# 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741
# 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751
# 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761
# 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771
# 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781
# 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791
# 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801
# 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811
# 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821
# 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831
# 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841
# 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851
# 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861
# 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871
# 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881
# 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891
# 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901
# 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911
# 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921
# 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931
# 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941
# 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951
# 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961
# 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971
# 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981
# 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991
# 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
# 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011
# 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021
# 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031
# 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041
# 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051
# 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061
# 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071
# 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081
# 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091
# 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101
# 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111
# 2110 2111 2112 2113 2114 2115 2116 2117 2118 21
```



## Next Best Option: Views

id	first_name	city
3	James	Vancouver

# Simple.



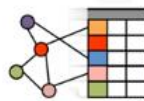
BIG DATA UNIVERSITY

We just need to format the triplet in a certain way to properly create a view.

IBM Analytics Education

## Creating a View with Triplet

graph.triplet



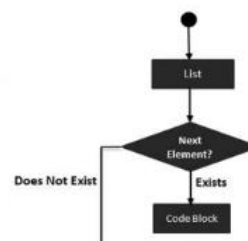
GraphX



Extends RDD Class

COLLECTION

graph.triplet.collect



BIG DATA UNIVERSITY

This will make the triplet usable in a for-loop, and hence create our view.

IBM Analytics Education

## Creating a View with Triplet

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

-----Code-----

```
}
```

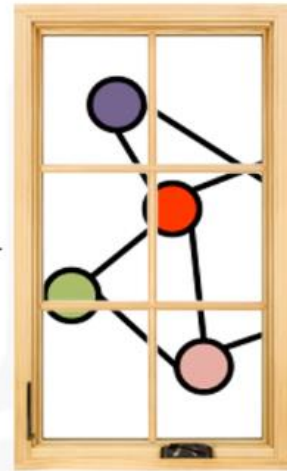
srcAttr -> **Source Attribute**

dstAttr -> **Destination Attribute**

srcId -> **Source ID**

dstId -> **Destination ID**

attr -> **Edge Attribute**



BIG DATA UNIVERSITY

## 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: (EdgeID, 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

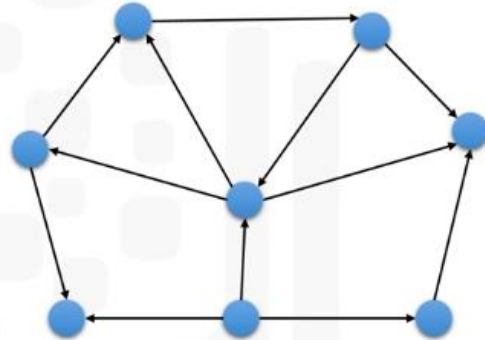
## GraphX Operators: numEdges, numVertices

graph.numEdges

**Number of Edges: 12**

graph.numVertices

**Number of Vertices: 8**



**BIG DATA UNIVERSITY**

Running numVertices delivers the number of vertices, in this case eight.

IBM Analytics Education

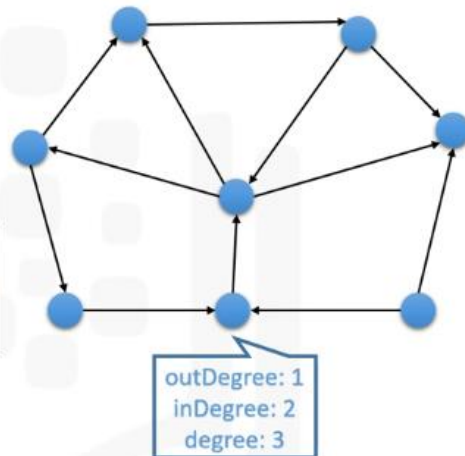
## GraphX Operators: inDegree, outDegree, degree

graph.inDegree

graph.outDegree

graph.degree

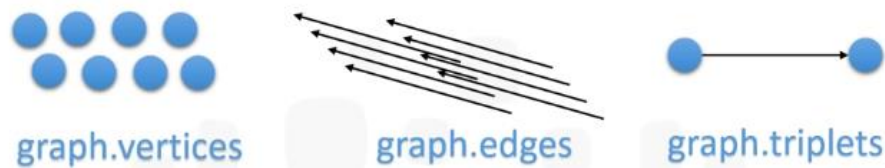
```
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]
```



**BIG DATA UNIVERSITY**

We need to treat them as collections by calling the collect function before printing the desired

## GraphX Operators: Vertices, Edges, and Triplets



```
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]
```

BIG DATA UNIVERSITY

pageRank

As with the other operators, we must run the collect function in order to print the information

Analytics Education

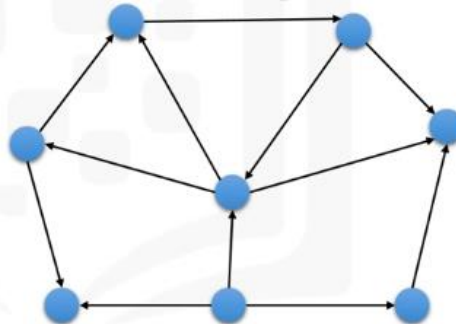
```
def pageRank(tol: Double, resetProb: Double = 0.15)
```

```
Graph[Double, Double]
```

```
graph.vertices.collect
```

```
(1, 0.25)
(2, 0.54)
(3, 0.23)
...
```

Rankings!



BIG DATA UNIVERSITY

The result is a set of Tuples.

IBM Analytics 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.

## Spark GraphX (Cognitive Class)

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 intuition!

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, 4L, "Follower"), Edge(3L, 5L, "Follower")))

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)
```

## Spark GraphX (Cognitive Class)

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

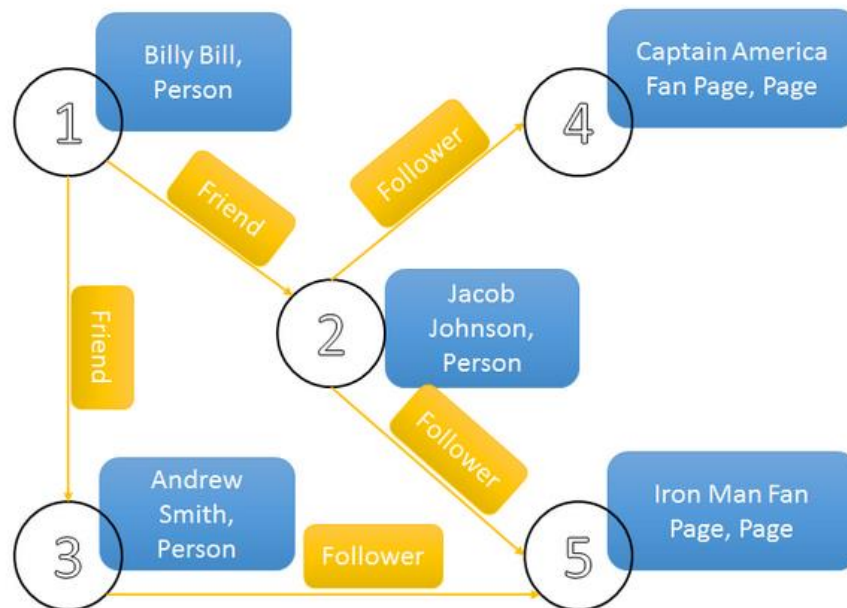
```
// Type your code here
var facebook = Graph(vertexRDD, edgeRDD, defaultVertex)

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 a 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

## Spark GraphX (Cognitive Class)

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 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 seen 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 and 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)
}
```



## Spark GraphX (Cognitive Class)

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

Immutable Distributed Collection of Objects

RDD Class

GraphX

Extends

BENEFITS

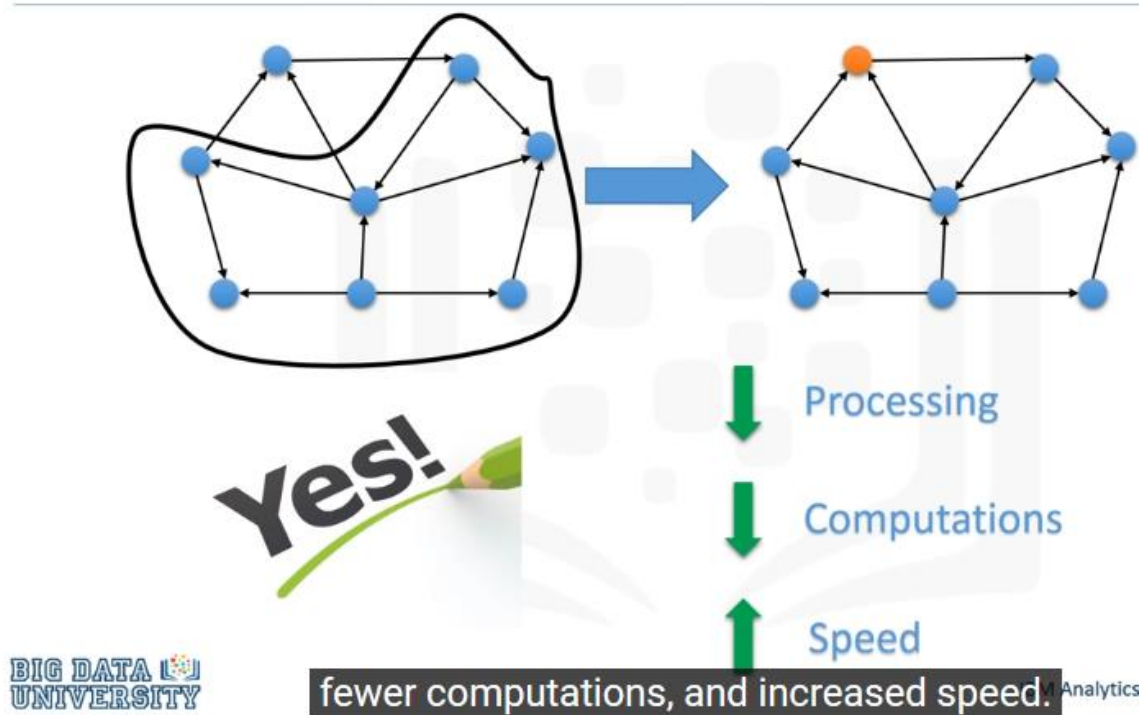
CHANGE

However, if they are immutable, can GraphX accommodate changes to a graph?

BIG DATA UNIVERSITY

IBM Analytics Education

## Modifying GraphX



## 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]
```



The mapTriplets operator runs through all of the triplets of the graph.

## 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)
```



to "Fred Flintstone".

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)
```



Our final statement will look like this.

IBM Analytics Education

## Property Operators: mapTriplets

```
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

```
graph.mapTriplets((triplet) => --if statement--)
```

srcId -> Source ID      srcAttr -> Source Attribute  
dstId -> Destination ID      dstAttr -> Destination Attribute  
attr -> Edge Attribute

```
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
```

```
if (triplet.attr == "Friend" && triplet.srcAttr == "Fred Flintstone")  
    "Worst-Friend" else edge.attr
```



Source Attribute of "Fred Flintstone"  
to change the edge relationship to "Worst-Friend".

Analytics Education

8

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”.

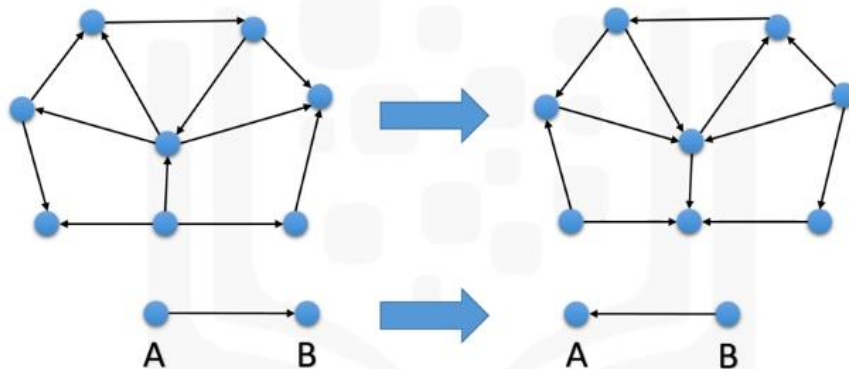
Again, notice the “else” statement at the end which keeps all other edges unchanged.

Thank you for watching this lesson!

## Structural Operators - Reverse

```
def reverse: Graph[VD, ED]
```

graph.reverse



This is particularly useful when trying to find the inverse pageRank of a graph.

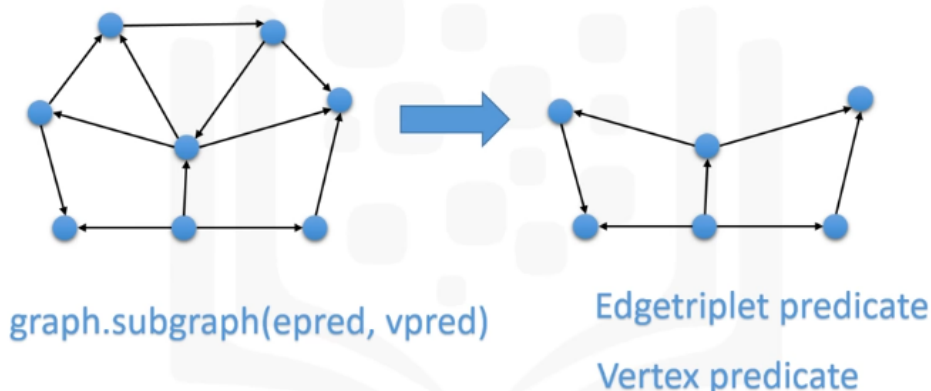
IBM Analytics Education

BIG DATA UNIVERSITY

Ver más...

## Structural Operators - SubGraph

```
def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
             vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
```



These two parameters determine what stays in the new sub-graph and what is removed.

Analytics Education

BIG DATA UNIVERSITY

## 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

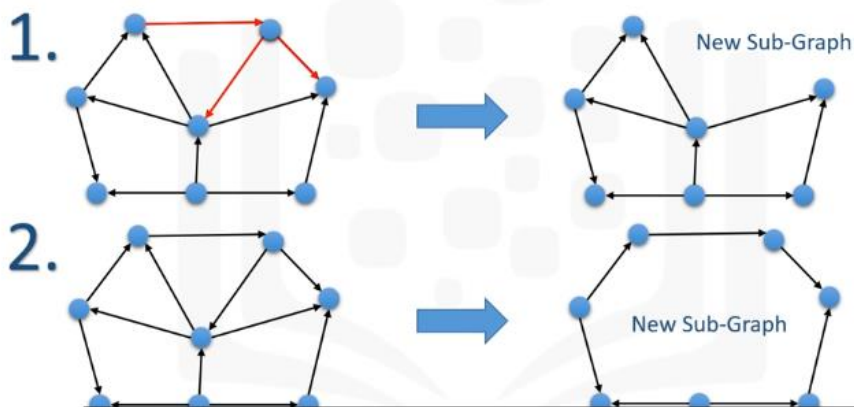
equal one, and returns a sub-graph containing only those vertices.

IBM Analytics Education

BIG DATA UNIVERSITY

## Structural Operators - SubGraph

```
def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
             vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
```



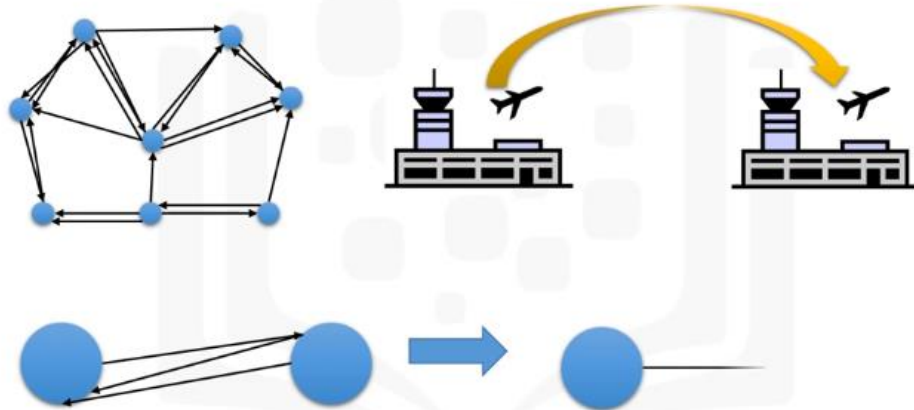
In this case, subgraph will remove the non-compliant vertex as well as any edges that were attached

Analytics Education

BIG DATA UNIVERSITY

## Structural Operators - groupEdges

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



BIG DATA UNIVERSITY

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

Analytics Education

## Structural Operators - PartitionBy

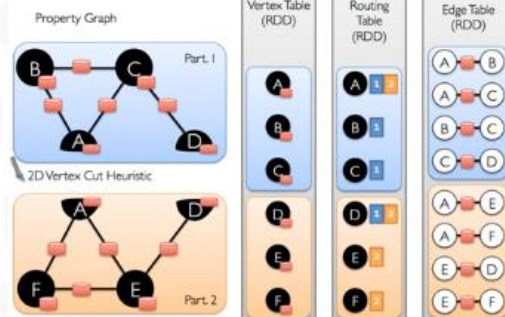
```
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
```

CanonicalRandomVertexCut

EdgePartition1D

EdgePartition2D

RandomVertexCut



```
graph.partitionBy(PartitionStrategy.EdgePartition1D)
```

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

BIG DATA UNIVERSITY

Analytics Education

## Structural Operators - groupEdges (continued)

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

```
graph.groupEdges(merge)
```

```
merge = (Edge1, Edge2) => Edge1 + Edge2
      => Edge1
      => Edge1 / Edge2
```



the other.

IBM Analyti

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 `edgetripletpredicate` and `vertexpredicate` 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 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.

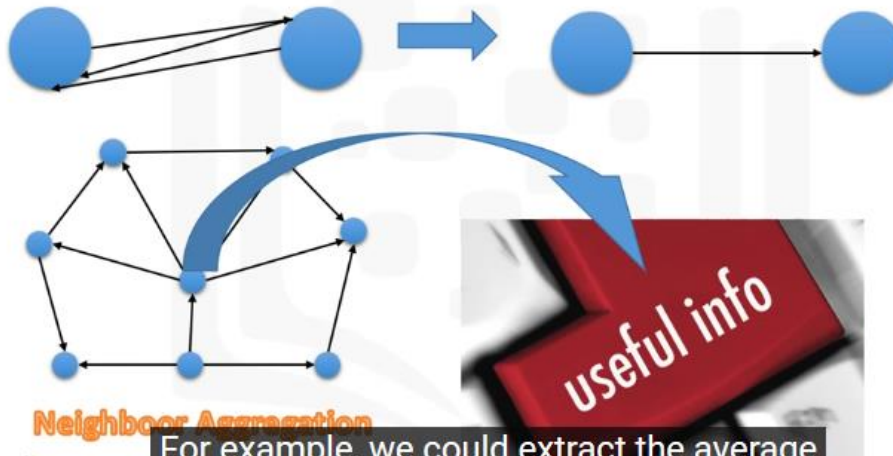
Thank you for watching this lesson!

**>>Lab:**

## MODULE 4 NEIGHBORHOOD AGGREGATION AND CACHING

### Neighbor Aggregation Functions

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



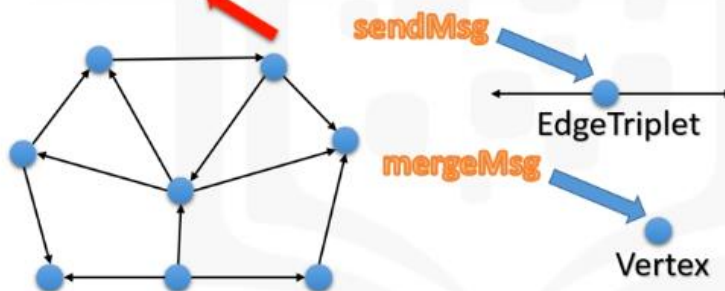
For example, we could extract the average age of friends from vertex attributes, or



Analytics Education

### Main Aggregation Function: AggregateMessages

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



Finally, AggregateMessages returns a Vertex RDD containing the destination vertices and



Analytics Education

## AggregateMessages: tripletFields Parameter

```
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

Analytics Education

## AggregateMessages: sendMsg Parameter

```
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.

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.

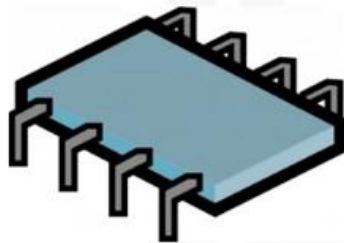
IBM Analytics Education

BIG DATA UNIVERSITY

Caching

```
def cache(): Graph[VD, ED]
```

```
graph.cache()
```



RDD Class

Immutable  
Distributed  
Collection of  
Objects

↑ Efficiency

This results in increased efficiency, speed and performance

BIG DATA UNIVERSITY

IBM Analytics Education

## 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.

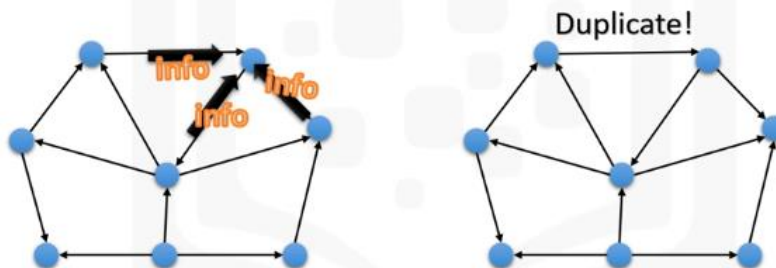
BIG DATA UNIVERSITY

IBM Analytics Education

11

## 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 aggregateMessages

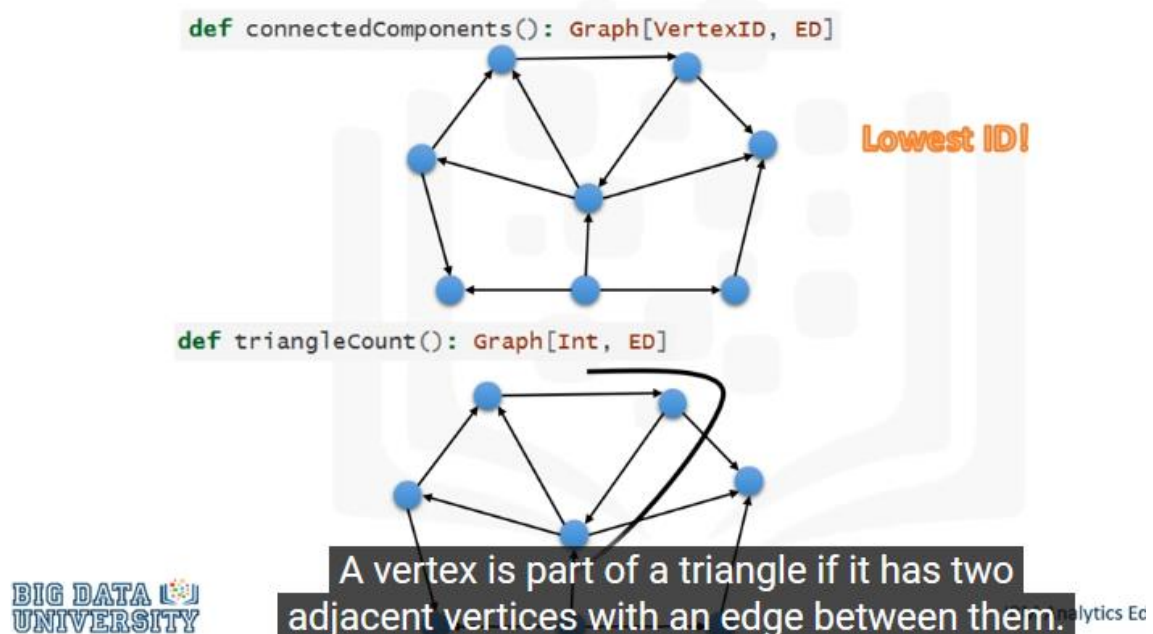
BIG DATA UNIVERSITY

Analytics Education

12



## Other Graph Algorithms



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 aggregateMessages 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:**