Przetwarzanie wielkich grafów: Apache Spark

Aleksander Ciepiela
Rafał Grabiański
Marcin Ziąber

What is Apache Spark?

“Apache® Spark™ is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics. It was originally developed at UC Berkeley in 2009.”

What is Apache Spark?

- Provides an application programming interface centered on a data structure called resilient distributed dataset (RDD) - a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way
- Developed in response to limitations in the MapReduce cluster computing paradigm which forces a particular linear dataflow structure on distributed programs

MapReduce

Read data from disk -> map -> reduce -> store results on disk

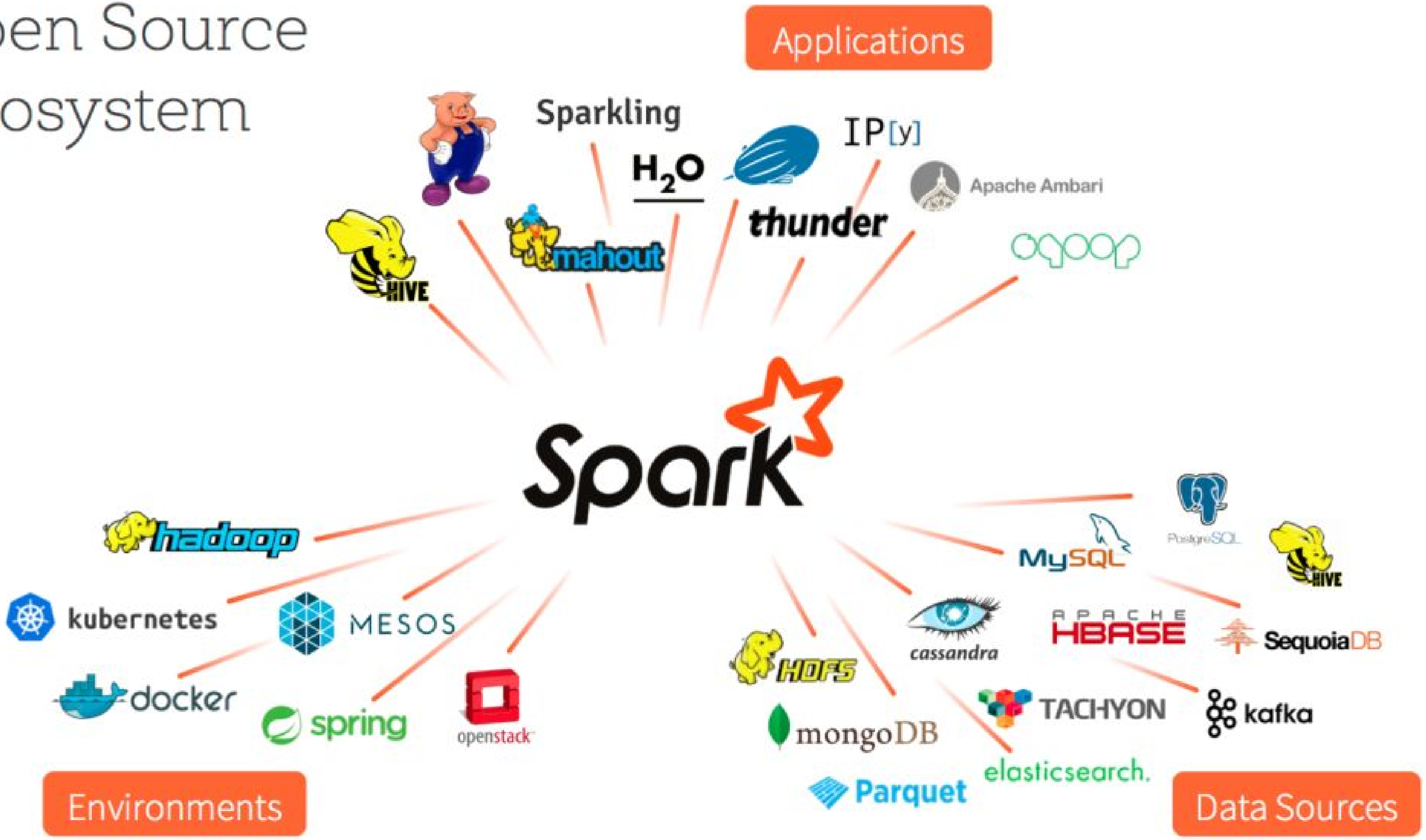
RDD

- Spark's RDDs function as a working set for distributed programs that offers a restricted form of distributed shared memory
- Facilitates the implementation of both iterative algorithms, that visit their dataset multiple times in a loop, and interactive/exploratory data analysis
- Especially useful for training algorithms for machine learning systems and graph processing algorithms

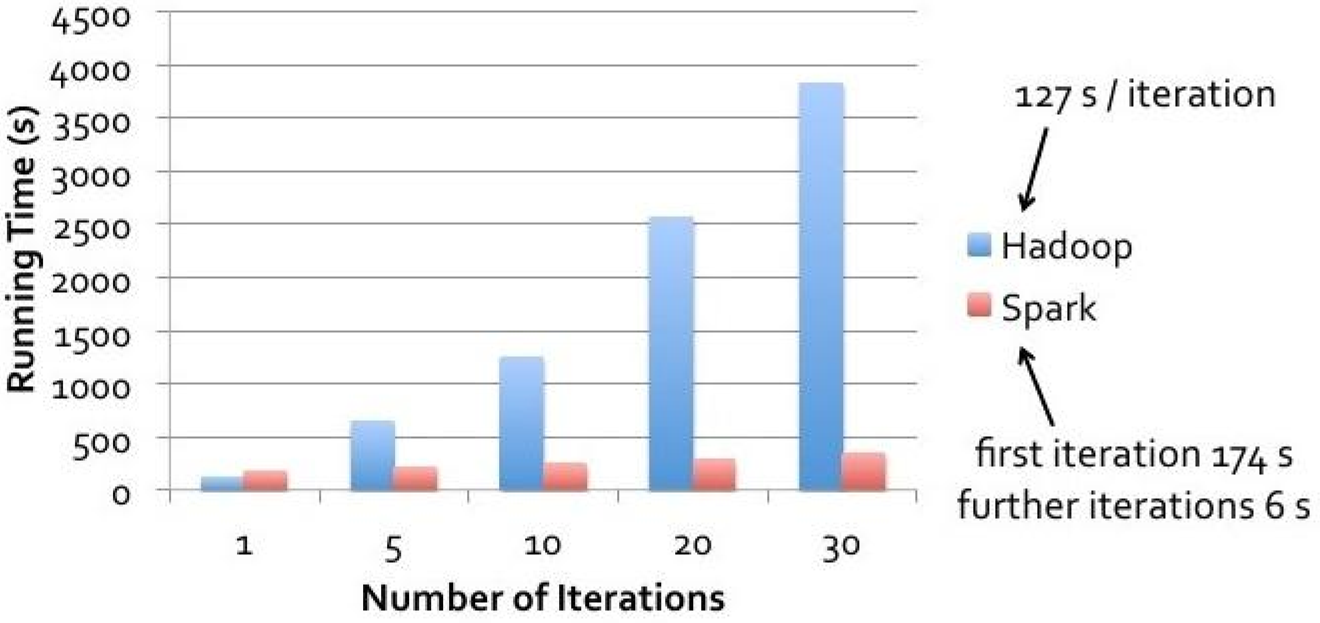
What is Apache Spark?

- Requires a cluster manager and a distributed storage system
- Cluster manager: Hadoop YARN, Apache Mesos ...
- Distributed storage system: HDFS, Cassandra, Amazon S3 …
- Supports pseudo-distributed local mode for development and testing: local file system and one executor per CPU core

Open Source Ecosystem



Logistic Regression Performance



Spark - GraySort 2014 winner

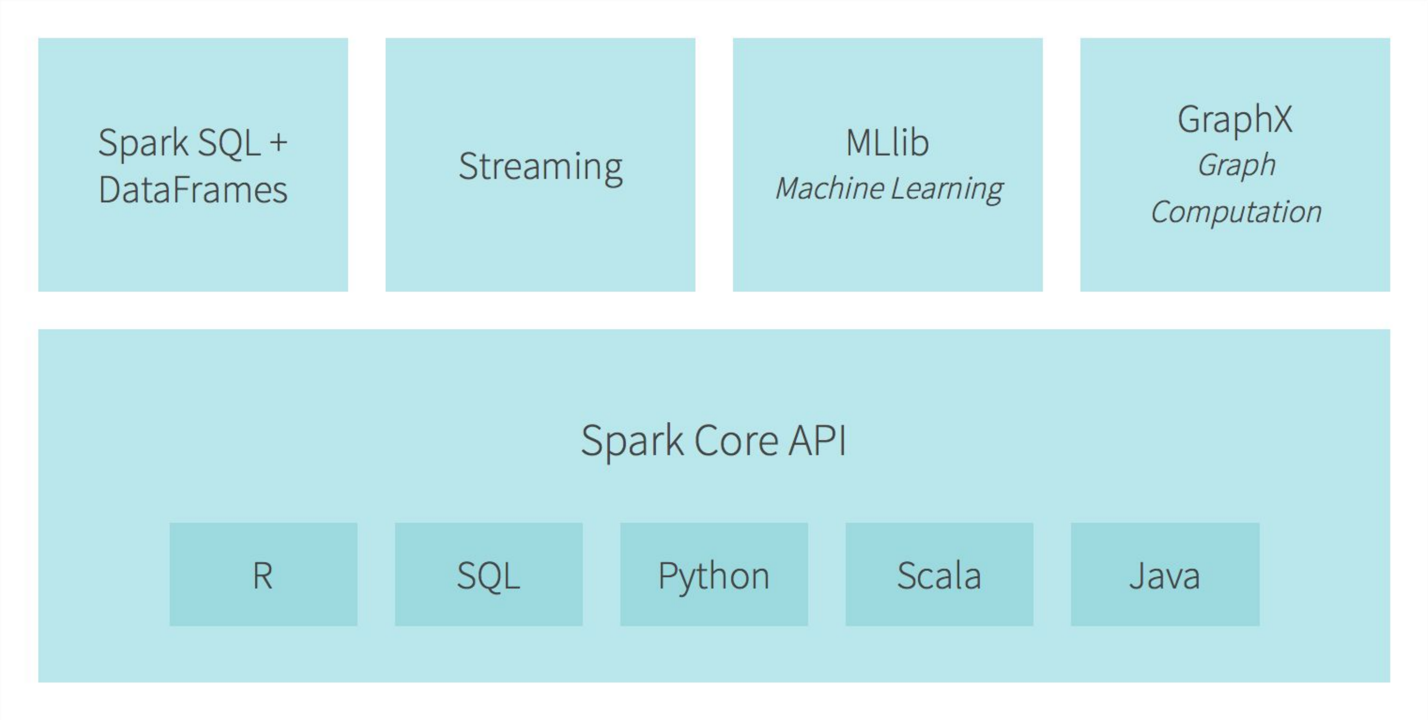
- An industry benchmark on how fast a system can sort 100 TB of data
- Previous world record: 72 minutes, Hadoop MapReduce cluster of 2100 nodes (Yahoo)
- New record: 23 minutes, 206 EC2 nodes
- 3X faster using 10X fewer machines

Spark - GraySort 2014 winner

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

* not an official sort benchmark record

Apache Spark Ecosystem



Spark Core

- Provides distributed task dispatching, scheduling, basic I/O exposed through an API
- Supports a functional/higher-order model of programming: map, filter, reduce etc.
- Spark schedules the function’s execution in parallel on the cluster
- Operations take RDDs as input and produce new RDDs - they are immutable and lazy evaluated
- Fault tolerance is achieved by keeping track of the lineage of each RDD (the sequence of operations that produced it) so that it can be reconstructed in the case of data loss

Example: Top 10 most frequent words

```scala
val conf = new SparkConf().setAppName("test")
val sc = new SparkContext(conf)
val data = sc.textFile("/path/to/somedir")
val mostFrequentWords = data.flatMap(_.split(" "))
                        .map((_, 1))
                        .reduceByKey(_ + _)
                        .sortBy(s => -s._2)
                        .map(x => (x._2, x._1))
                        .top(10)
```

Spark SQL

- Introduces a data abstraction called DataFrames which provides support for structured and semi-structured data.
- Provides domain-specific language to manipulate DataFrames
- Provides SQL language support with command-line interfaces and ODBC/JDBC server

Example: Count people by age

```scala
import org.apache.spark.sql.SQLContext

val url = "jdbc:mysql://yourIP:yourPort/test?user=yourUsername;password=yourPassword" // URL for your database server.
val sqlContext = new org.apache.spark.sql.SQLContext(sc) // Create a sql context object

val df = sqlContext
  .read
  .format("jdbc")
  .option("url", url)
  .option("dbtable", "people")
  .load()

df.printSchema() // Looks the schema of this DataFrame.
val countsByAge = df.groupBy("age").count() // Counts people by age
```

Streaming

- Leverages Spark Core’s fast scheduling capability to perform streaming analytics
- Ingests data in mini-batches and performs RDD transformations on those mini-batches of data. It enables the same set of application code written for batch analytics to bes used in streaming analytics (lambda architecture)
- Penalty of latency equal to the mini-batch duration
- Consumes data from Kafka, Flume, Twitter, Kinesis, TCP/IP sockets etc.

Streaming


input data
stream
Spark
Streaming
batches of
input data
Spark
Engine
batches of
processed data

MLib - machine learning library

- Distributed machine learning framework on top of Spark Core
- Due to its distributed memory-based architecture it is up to nine times faster than disk-based implementation used by Apache Mahout
- Many common machine learning and statistical algorithms have been implemented and are shipped with MLib which simplifies large scale machine learning pipelines

MLib - some included algorithms

- Summary statistics, correlations, random data generation
- Classification and regression: support vector machines, linear regression, decision trees
- Cluster analysis methods: k-means, Latent Dirichlet Allocation (LDA)
- Dimensionality reduction techniques: singular value decomposition (SVD), principal component analysis (PCA)
- Optimization algorithms: stochastic gradient descent, limited-memory BFGS (L-BFGS)

GraphX

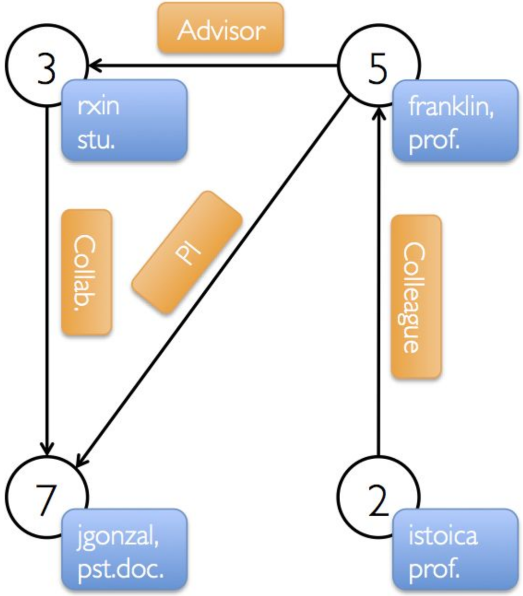
- "GraphX is a new component (alpha) in Spark for graphs and graph-parallel computation."
- extends the Spark RDD by introducing a new Graph abstraction: a directed multigraph with properties attached to each vertex and edge.
- Introduces:
  - set of fundamental graph operators (e.g., subgraph, joinVertices, and aggregateMessages)
  - optimized variant of the Pregel API

Property Graph

- Directed multigraph with user defined objects attached to each vertex and edge
- Like RDDs, property graphs are immutable, fault-tolerant and distributed
- GraphX optimizes the representation of edge and vertex types when they are primitive data types by using specialized arrays (memory footprint reduction)
- Each Vertex is keyed by a unique 64-bit long id

Example Property Graph

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

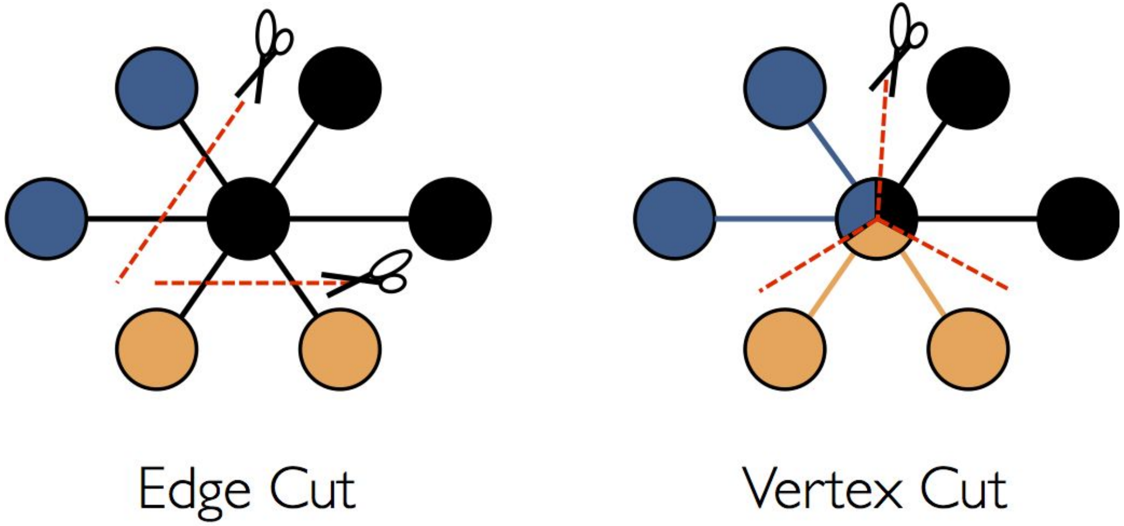
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

```
val userGraph: Graph[(String, String), String]

// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L,
("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L,
("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L,
"advisor"),  Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with
missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```

Graphs distribution

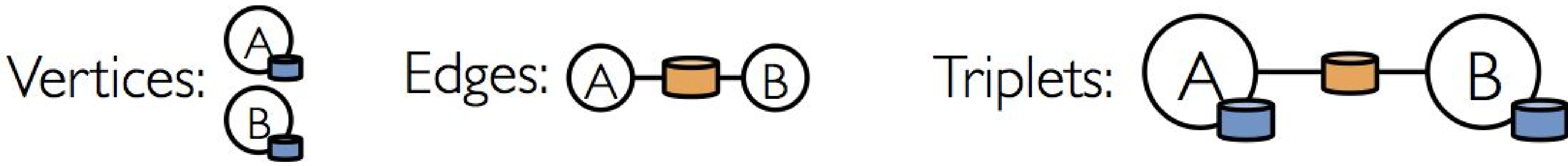
In order to optimize graphs parallel processing they are distributed across the executors using vertex-cut approach.



The exact method of assigning edges depends on PartitionStrategy. User can choose between different heuristics.

Graph views

- Edge view:
  - `graph.edges`
- Vertex view:
  - `graph.vertices`
- Triplet view:
  - `graph.triplet`



Graph operators

```scala
// Information about the Graph
 val numEdges: Long
 val numVertices: Long
 val inDegrees: VertexRDD[Int]
 val outDegrees: VertexRDD[Int]
 val degrees: VertexRDD[Int]
```

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

```scala
// Functions for caching graphs (by default they are not cached in
memory)
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY):
Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
```

```scala
// 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]
```

```scala
// Basic graph algorithms
  def pageRank(tol: Double, resetProb: Double =
0.15): Graph[Double, Double]


  def connectedComponents(): Graph[VertexID, ED]


  def triangleCount(): Graph[Int, ED]


  def stronglyConnectedComponents(numIter: Int):
Graph[VertexID, ED]
```

What’s more...

GraphX supports more complex operations like:

- Iterative like graph-parallel computation
- Joining RDDs with graph
- Aggregation of informations about adjacent triplets

Supported graph algorithms

- PageRank
- ConnectedComponents
- StronglyConnectedComponents
- TriangleCounting

PageRank

- PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.
- PageRank is a link analysis algorithm and it assigns a numerical weighting to each element of a hyperlinked set of documents, such as the WWW, with the purpose of "measuring" its relative importance within the set.

PageRank

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

- PR(X) - PageRank value for page (vertex) X
- d - damping factor - probability that the surfer will continue
- N - number of pages (vertices)
- L(X) - number of X outbound links

PageRank in GraphX

- GraphX comes with static and dynamic implementations of PageRank
- static PageRank runs for a fixed number of iterations
- dynamic PageRank runs until the ranks converge (i.e., stop changing by more than a specified tolerance)
- May be performed using PageRank object or directly by calling methods on Graph object

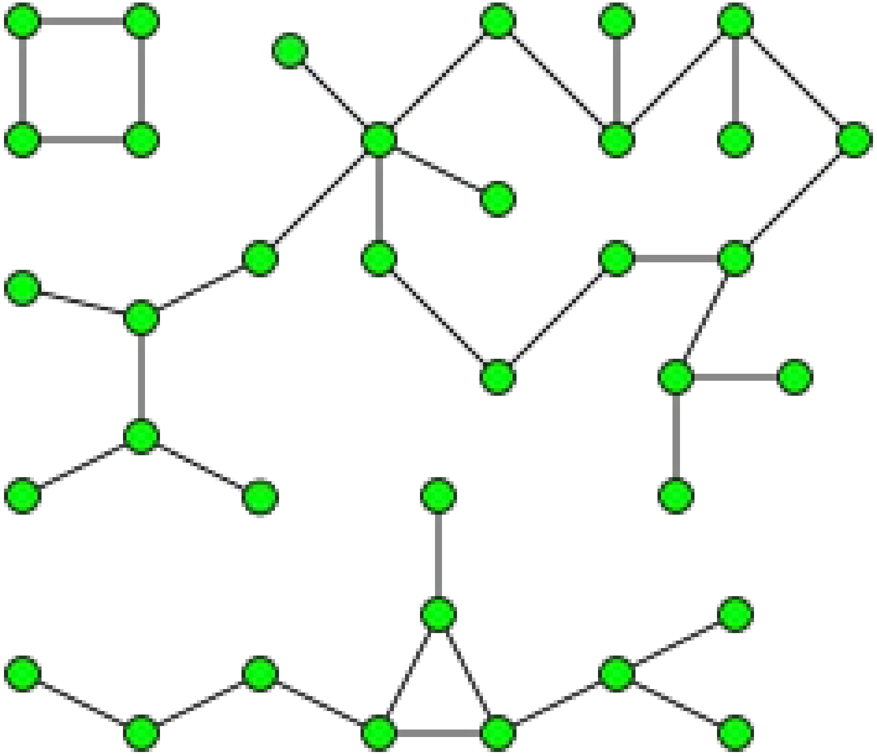
PageRank in GraphX

```scala
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
```

Connected components - definition

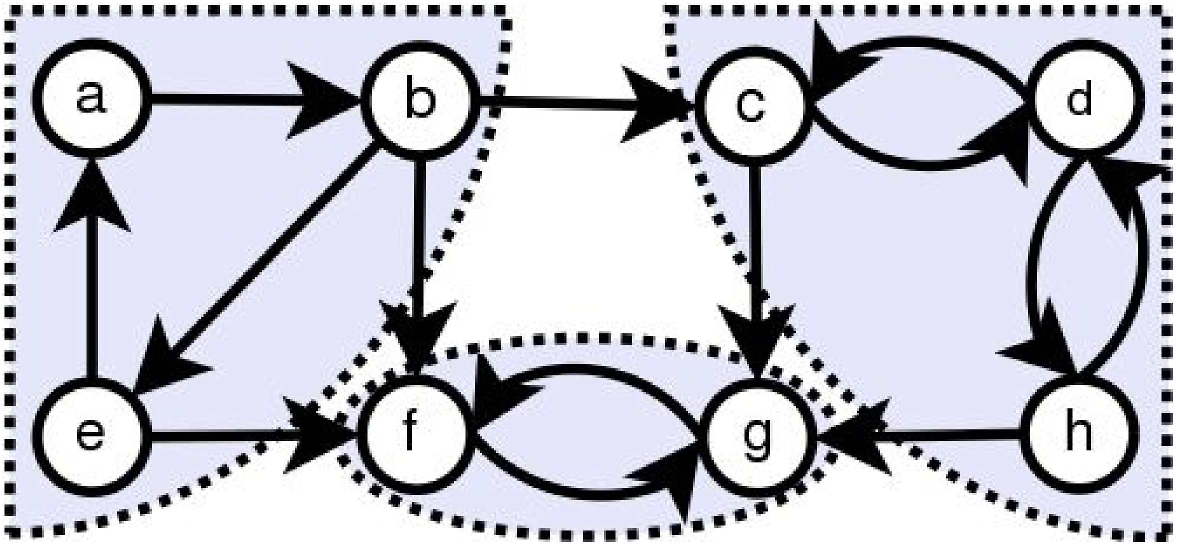
Connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Connected components



Strongly connected components - definition

In the mathematical theory of directed graphs, a graph is said to be strongly connected or disconnected if every vertex is reachable from every other vertex.



Connected components in GraphX

The connected components algorithm labels each connected component of the graph with the ID of its lowest-numbered vertex

```
// Find the connected components

val cc = graph.connectedComponents().vertices

val cc = graph.stronglyConnectedComponents(10).vertices
```

Triangle counting

The triangle is a three-node small graph, where every two nodes are connected.

GraphX requires that:

- the edges to be in canonical orientation (srcId < dstId)
- graph to be partitioned using *Graph.partitionBy*

Triangle count - algorithm

GraphX counts the triangles passing through each vertex using a straightforward algorithm:

- Compute the set of neighbors for each vertex;
- For each edge compute the intersection of the sets and send the count to both vertices
- Compute the sum at each vertex and divide by two since each triangle is counted twice

Triangle count - example

```scala
// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
```

Facebook: A comparison of state-of-the-art graph processing systems

[Giraph vs GraphX (19.10.2016)]

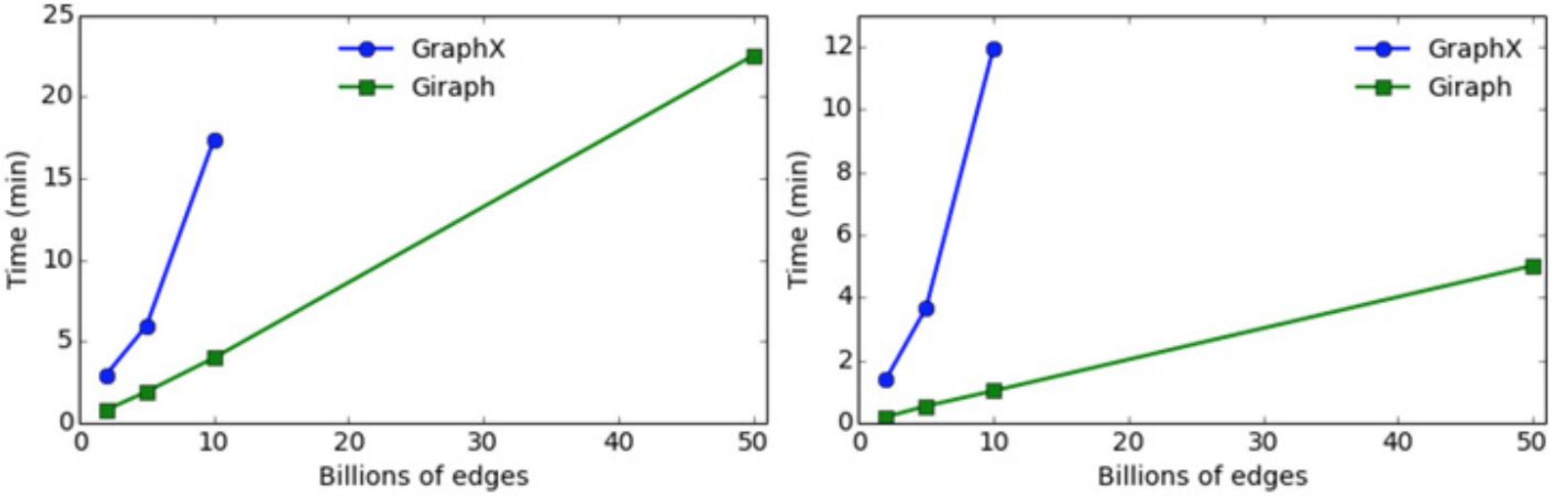


Figure 4. Running time of PageRank (left) and Connected Components (right) on a synthetic graph as the number of edges increases.

Zadania

Repozytorium z kodem: https://github.com/rafalgrm/spark

Dane do pobrania w katalogu data: Marvel-graph.txt, Marvel-names.txt

Porównanie ‘pure Spark’ vs Spark + GraphX w prostych zadaniach przetwarzania grafów

Wykonanie wbudowanego w bibliotekę GraphX algorytmu grafowego na większym datasecie (jak starczy czasu)

Zbiory danych

Marvel-names.txt:

1 "24-HOUR MAN/EMMANUEL"
2 "3-D MAN/CHARLES CHAN"
3 "4-D MAN/MERCURIO"
4 "8-BALL/"
5 "A"
6 "A'YIN"
7 "ABBOTT, JACK"
8 "ABCISSA"
9 "ABEL"

Marvel-graph.txt:

5988 748 1722 3752 4655 5743 1872 3413 5527 6368 6085 4319 4728 1636 2397 3364 4001 1614 1819 1585 732 2660 3952 2507 **3891**
2070 2239 2602 612 1352 5447 4548 1596 5488 1605 5517 11 479 2554 2043 17 865 4292 6312 473 534 1479 6375 4456
5989 4080 4264 4446 3779 2430 2297 6169 3530 3272 4282 6432 2548 4140 185 105 3878 2429 1334 4595 2767 3956 3877 4776 **4946**
3407 128 269 5775 5121 481 5516 4758 4053 1044 1602 3889 1535 6038 533 3986
5982 217 595 1194 3308 2940 1815 794 1503 5197 859 5096 6039 2664 651 2244 528 284 1449 1097 1172 1092 108 3405 5204 387 **4607**
4545 3705 4930 1805 4712 4404 247 4754 4427 1845 536 5795 5978 533 3984 6056
5983 1165 3836 4361 1282 716 4289 4646 6300 5084 2397 4454 1913 5861 5485
5980 2731 3712 1587 6084 2472 2546 6313 875 859 323 2664 1469 522 2506 2919 2423 3624 5736 5046 1787 5776 3245 3840 2399

1. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci

- Preprocessing danych - mapowanie każdej linijki inputu do krotki: (heroId, # of occurrences)
- Redukcja wyrazów po kluczu (heroId)
- Znalezienie maksymalnej wartości # of occurrences
- Znalezienie nazwy superbohatera

Notebook: MostPopularHero/MostPopularHero.ipynb
Plik źródłowy: src/exercises/MostPopularHero.scala

1. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci

```scala
/* given line extracts heroID -> number of connections tuple*/
def countCoOccurences(line:String) = {
 var elements = line.split("\\s+")
 (elements(0).toInt, elements.length-1)
}

/* given line extracts heroID -> heroName tuple */
def parseNames(line:String):Option[(Int, String)] = {
 var fields = line.split("\"")
 if (fields.length > 1) {
   Some(fields(0).trim.toInt, fields(1))
 } else {
   None
 }
}
```

```scala
// create SparkContext with use of every core on our local machine
val sc = new SparkContext("local[*]",
"MostPopularHeroContext")

// heroID -> name RDD
val namesRdd =
sc.textFile("Marvel-names.txt").flatMap(parseNames)

// heroID -> number of connections RDD
val pairings =
sc.textFile("Marvel-graph.txt").map(countCoOccurences)

// TODO --- calculating result
// make reduction of the same heroID RDDs

// extracting result

// TODO ^^^ calculating result
```

1. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci

```
// TODO --- calculating result
// make reduction of the same heroID RDDs
val totalFriendsByCharacters = pairings.reduceByKey((x,y) => x+y)
…
```

1. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci

```
// TODO --- calculating result
// make reduction of the same heroID RDDs
val totalFriendsByCharacters = pairings.reduceByKey((x,y) => x+y)
// extracting result
val flipped = totalFriendsByCharacters.map(x => (x._2, x._1))
val mostPopular = flipped.max
val mostPopularName = namesRdd.lookup(mostPopular._2)
println(mostPopularName(0))
// TODO ^^^ calculating result
```

2. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci (**GraphX**)

- Preprocessing danych:
  - Stworzenie VertexRDD (VertexId -> name of hero)
  - Zmapowanie Marvel-graph.txt do listy obiektów Edge (EdgeRDD): (List[Edge[Int]])
- Zbudowanie grafu Graph(verts, edges, default)
- Analiza grafu - znalezienie 10 bohaterów o najwyższych stopniach w grafie

Notebook: MostPopularHero/MostPopularHeroGraph.ipynb

Plik źródłowy: src/exercises/MostPopularHeroGraph.scala

2. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci (**GraphX**)

```scala
def parseNames(line:String):Option[(VertexId, String)] = {
 var fields = line.split("\"")
 if (fields.length > 1) {
   val heroId:Long = fields(0).trim.toLong
   if (heroId < 6487) {
     return Some(fields(0).trim.toLong, fields(1))
   }
 }
 None
}

def makeEdges(line:String):List[Edge[Int]] = {
 import scala.collection.mutable.ListBuffer
 var edges = new ListBuffer[Edge[Int]]()
 val fields = line.split(" ")
 val origin = fields(0)

 for (x <- 1 to (fields.length-1)) {
   edges += Edge(origin.toLong, fields(x).toLong, 0)
 }

 edges.toList
}
```

```scala
val sc = new SparkContext("local[*]", "MostPopularHeroGraphCtx")

// vertices
val names = sc.textFile("Marvel-names.txt")
val vertices = names.flatMap(parseNames)

// edges
val lines = sc.textFile("Marvel-graph.txt")
val edges = lines.flatMap(makeEdges)

// graph
val default = "Nobody"
val graph = Graph(vertices, edges, default).cache()

// get top 15 most-connected heroes
// TODO your code goes here
```

2. Wyświetl nazwę bohatera występującego z największą liczbą innych postaci (**GraphX**)

```
graph.degrees.join(vertices).sortBy(_._2._1, ascending = false).take(15).foreach(println)
```

3. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (?) implementując BFS

- Preprocessing danych:
  - Stworzenie krotki (id, connections, distance, color) (color = WHITE, GRAY, BLACK)
- Implementacja BFS:
  - Szukamy szarych nodów
  - Aktualizujemy stany sąsiadów zmieniając ich kolor na szary
  - Kolorujemy na czarno wyjściowy node
- BFS jako operacja Map-Reduce:
  - **MAP**: Każdy szary node => nowe nody dla każdego połączenia z odległością zwiększoną o 1, szarym kolorem i bez połączeń + node wejściowy zmieniony kolor na czarny
  - **REDUCE**: łączymy każdy node z tym samym heroId, zachowując najmniejszą odległość i najciemniejszy kolor oraz wszystkie połączenia
- Kiedy kończymy? Accumulator!

Notebook: MostPopularHero/DegreesOfSeparation.ipynb
Plik źródłowy: src/exercises/DegreesOfSeparation.scala

3. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (?) implementując BFS

```
/* convertion line from input file to bfs node */
def convertToBFS(line:String):BFSNode = {
 val fields = line.split("\\s+")
 val id = fields(0).toInt

 var connections:ArrayBuffer[Int] = ArrayBuffer()
 for (connection <- 1 to (fields.length-1)) {
   connections += fields(connection).toInt
 }

 var color:Color.Color = Color.WHITE
 var distance:Int = Int.MaxValue

 if (id == startCharId) {
   color = Color.GRAY
   distance = 0
 }

 (id, (connections.toArray, distance, color))
}

def createStartingRDD(sc:SparkContext): RDD[BFSNode] = {
 sc.textFile("Marvel-graph.txt").map(convertToBFS)
}
```

```
val sc = new SparkContext("local[*]",
"DegreesOfSeparationContext")
hitCounter = Some(sc.accumulator(0))

var iterationRDD = createStartingRDD(sc)

var iteration:Int = 0

for (iteration <- 1 to 10) {
 println("BFS Iteration " + iteration)

 val mapped = iterationRDD.flatMap(BFSMap)
 println("Processing " + mapped.count() + " values.")

 if (hitCounter.isDefined) {
   val hitCount = hitCounter.get.value
   if (hitCount > 0) {
     println("Hit the target counter! From " + hitCount + " different
directions")
   }
   return
 }

 // reducer work
 iterationRDD = mapped.reduceByKey(BFSReduce)
}

// TODO --- print results
```

```scala
// map function
// expands node into itself and its children
def BFSMap(node:BFSNode):Array[BFSNode] = {

 val characterId = node._1
 val data = node._2

 val connections:Array[Int] = data._1
 val distance:Int = data._2
 var color:Color.Color = data._3

 var result:ArrayBuffer[BFSNode] = ArrayBuffer()

 if (color == Color.GRAY) {
   for (conn <- connections) {
     val newCharID = conn
     val newDist = distance + 1
     val newColor = Color.GRAY

     // have we stumbled accross searched character?
     if (targetCharId == conn) {
       if (hitCounter.isDefined) hitCounter.get.add(1)
     }

     val newEntry:BFSNode = (newCharID, (Array(), newDist, newColor))
     result += newEntry
   }

   // all nodes processed here...
   color = Color.BLACK
 }

 val thisEntry:BFSNode = (characterId, (connections, distance, color))
 result += thisEntry
 result.toArray
}
```

```scala
def BFSReduce(data1:BFSData, data2:BFSData):BFSData = {
 // extracting data we are combining
 val edges1:Array[Int] = data1._1
 val edges2:Array[Int] = data2._1
 val dist1:Int = data1._2
 val dist2:Int = data2._2
 val color1:Color.Color = data1._3
 val color2:Color.Color = data2._3

 // default node values
 var dist:Int = Int.MaxValue
 var color:Color.Color = Color.WHITE
 var edges:ArrayBuffer[Int] = ArrayBuffer()

 // TODO --- TYPE YOUR CODE HERE
 // merge edges

 // preserve minimum distance

 // preserve darkest color

 // TODO ^^^ TYPE YOUR CODE HERE


 // return result of reduction
 (edges.toArray, dist, color)
}
```

3. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (?) implementując BFS

```
// TODO --- TYPE YOUR CODE HERE
// merge edges
if (edges1.length > 0) {
 edges ++= edges1
}

if (edges2.length > 0) {
 edges ++= edges2
}

// preserve minimum distance
dist = math.min(dist1, dist2)

// preserve darkest color
...
// TODO ^^^ TYPE YOUR CODE HERE
```

3. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (?) implementując BFS

```
// TODO --- TYPE YOUR CODE HERE
// merge edges
if (edges1.length > 0) {
 edges ++= edges1
}

if (edges2.length > 0) {
 edges ++= edges2
}

// preserve minimum distance
dist = math.min(dist1, dist2)

// preserve darkest color
if (color1 == Color.WHITE) color = color2
else if (color1 == Color.GRAY) {
 if (color2 == Color.BLACK) color = color2
 else color = color1
}
else color = color1
// TODO ^^^ TYPE YOUR CODE HERE

// TODO --- print results
println(iterationRDD.lookup(targetCharId)(0)._2)
```

4. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (**GraphX + Pregel API**)

- Preprocessing danych dokładnie taki sam jak w poprzednim przykładzie z GraphX
- Użycie modelu Pregel
  - Wierzchołki grafu wysyłają wiadomości do swoich sąsiadów
  - Graf jest procesowany w iteracjach zwanymi supersteps
  - W każdym superstepie:
    - Wiadomości z poprzednich iteracji są odbierane przez wierzchołek
    - Każdy wierzchołek przetwarza siebie na bazie tych wiadomości
    - Każdy wierzchołek wysyła wiadomości do innych wierzchołków

Notebook: MostPopularHero/DegreesOfSeparationGraph.ipynb

Plik źródłowy: src/exercises/DegreesOfSeparationGraph.scala

4. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (**GraphX + Pregel API**)

```scala
def parseNames(line:String):Option[(VertexId, String)] = {
 var fields = line.split("\"")
 if (fields.length > 1) {
   val heroId:Long = fields(0).trim.toLong
   if (heroId < 6487) {
     return Some(fields(0).trim.toLong, fields(1))
   }
 }
 None
}

def makeEdges(line:String):List[Edge[Int]] = {
 import scala.collection.mutable.ListBuffer
 var edges = new ListBuffer[Edge[Int]]()
 val fields = line.split(" ")
 val origin = fields(0)

 for (x <- 1 to (fields.length-1)) {
   edges += Edge(origin.toLong, fields(x).toLong, 0)
 }

 edges.toList
}
```

```scala
def main(args: Array[String]): Unit = {
 val sc = new SparkContext("local[*]",
"MostPopularHeroGraphCtx")

 // vertices
 val names = sc.textFile("Marvel-names.txt")
 val vertices = names.flatMap(parseNames)

 // edges
 val lines = sc.textFile("Marvel-graph.txt")
 val edges = lines.flatMap(makeEdges)

 // graph
 val default = "Nobody"
 val graph = Graph(vertices, edges, default).cache()

...
```

4. Policz stopień oddalenia (odległość) pomiędzy SpiderManem i ADAM-em (**GraphX + Pregel API**)

```
val initialGraph = graph.mapVertices((id, _) => if (id == root) 0.0 else Double.PositiveInfinity)

// pregel algorithm
// pregel sends initial message of PositiveInfinity to every vertex and we set up 10 iterations
// TODO --- correct pregel arguments
val bfs = initialGraph.pregel(Double.PositiveInfinity, 10) (
 // program for vertex - it has to preserve the shortest distance between incoming message and current attribute
 (id, attr, msg) => attr,

 // send message function - propagates out to all neighbours every iteration
 triplet => Iterator.empty,

 // reduce operation - preserving minimum of messages received by vertex if it received more than one in each iteration
 (a, b) => a
)
// TODO ^^^ correct pregel arguments


  // get top 10 results
  bfs.vertices.join(vertices).take(10).foreach(println)
  // like in previous exercise SpiderMan to Adam
  println("\n\nDegrees from SpiderMan to ADAM")
  bfs.vertices.filter(x => x._1 == 14).collect.foreach(println)
```

Przykłady funkcji specjalizowanych z biblioteki GraphX

Użyjemy do naszego grafu dwóch funkcji:
pageRank() oraz triangleCount()

```scala
val sc = new SparkContext("local[*]", "MostPopularHeroGraphCtx")

// vertices
val names = sc.textFile("Marvel-names.txt")
val vertices = names.flatMap(parseNames)

// edges
val lines = sc.textFile("Marvel-graph.txt")
val edges = lines.flatMap(makeEdges)

// graph
val default = "Nobody"
val graph = Graph(vertices, edges, default).cache()

// calculating PageRank
// TODO your code goes here

// calcularing triangle count
// TODO your code goes here
```

PageRank in GraphX

```scala
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
```

Triangle count - example

```scala
// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
```

Przykłady funkcji specjalizowanych z biblioteki GraphX

```
// calculating PageRank

val ranks = graph.pageRank(0.001).vertices

ranks.join(vertices).sortBy(_._2._1, ascending = false).take(20).foreach(println)



// calcularing triangle count

val graphPartiotioned = graph.partitionBy(PartitionStrategy.RandomVertexCut)

val triCounts = graph.triangleCount().vertices

triCounts.join(vertices).sortBy(_._2._1, ascending = false).take(20).foreach(println)
```