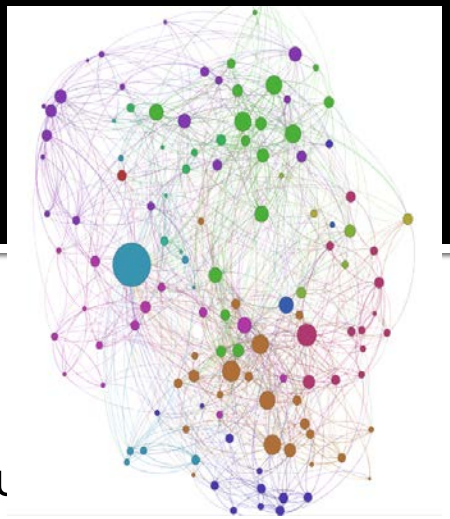
Large-scale Graph Processing in the Big Data World

Assoc. Prof. Dr. Buğra Gedik
Dept. of Computer Engineering,
Bilkent University, Ankara, Turkey

Bilkent Üniversitesi

Motivation



- Graph data is everywhere
  - *Relationships* between people, systems, and the natu
  - *Interactions* between people, systems, and the nature
- Relationship graphs
  - Social media: Twitter follower-followee graph, Facebook friendship graph, etc.
  - The web: The link graph of web pages
  - Transportation networks, biological networks, etc.
- Interaction graphs
  - Social media: Mention graph of twitter
  - Telecommunications: Call Detail Records (CDRs)
- Interaction graphs can be summarized to form relationship graphs

Applications

- Finding influence for ranking
  - Pages that are influential within the web graph (PageRank)
  - Users that are influential within the Twitter graph (TweetRank)
- Community detection for recommendations, churn prediction
  - If X is in the same community with Y, they may have similar interests
  - If X has churned, Y might be likely to churn as well
- Diffusion for targeted advertising
  - Start with known users in the graph with known interests, diffuse to others
- Regular path queries and graph matching for surveillance

Big Data

- The increase in the **V**olume, **V**elocity, and **V**ariety of data has passed a threshold such that existing data management and mining technologies are insufficient in managing and extracting actionable insight from this data.
- Big Data technologies are new technologies that represent a paradigm shift, in the areas of platform, analytics, and applications.
- Key features

Scalability in managing and mining of data

Analysis and mining *with low-latency and high throughout*

Non-traditional data, including semi-structured and unstructured

Graphs: Processing vs. Management

- Graphs pose challenges in processing and management
- RDBMS are inadequate for graph analytics
  - Traditional graph algorithms require *traversals* (e.g., BFS, DFS)
  - Traversals require recursive SQL: difficult to write, costly to execute
- Large-scale graphs require distributed systems for scalability
- Management vs Processing
  - Management: CRUD operations (Create, Read, Update, Delete)
  - Processing: Graph analytics (BFS, Connected Components, Community Detection. Clustering Coefficient, PageRank, etc.)
- Systems may support one or both
  - This talk focus on *graph processing systems*, with a focus on distributed ones

Distributed Graph Processing Systems

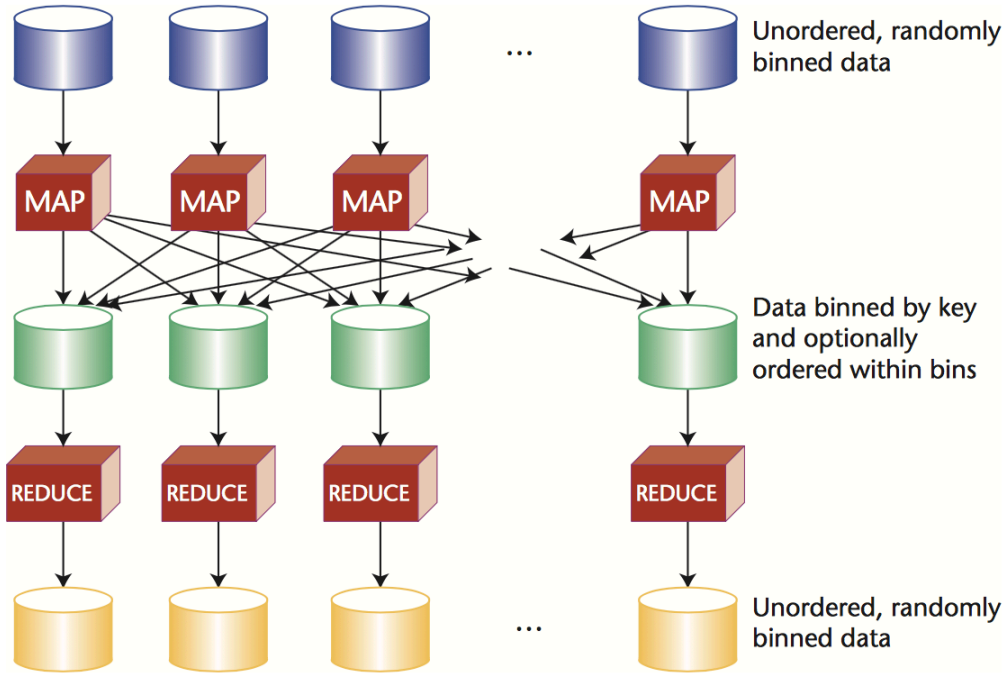
- Graph data stays on the disk, typically in a distributed file system
  - E.g., graph data is on HDFS, in the form of *list of edges*
- To perform a graph analytic, the graph is loaded from the disk to the memory of a set of processing nodes
- The graph analytic is performed in-memory, using multiple nodes, typically requiring communication between them
- The graph could be potentially morphed during the processing
- The results (which could be a graph as well) are written back to disk
- Overall, it is a batch process
  - E.g., Compute the PageRank over the current snapshot of the web graph
- Advantages: Fast due to in-memory processing, scalable with increasing number of processing nodes

Some approaches

- Apache Hadoop & Map/Reduce
  - Use Map/Reduce framework for executing graph analytics
- Vertex Programming
  - BSP-inspired distributed processing specifically designed for graphs
  - Synchronous model
    - Foundational work: Pregel from Google
    - Pregel clones: Giraph, HAMA (more general), Spark GraphX
  - Asynchronous model
    - GraphLab, PowerGraph
    - Disk-based variety: GraphChi

HADOOP & MAP/REDUCE

- M/R is a scalable and fault-tolerant distributed data processing framework
- Why not use it for graph processing?
- Can we represent useful graph operations as M/R steps?
- How easy and efficient is it to implement graph analytics as a series of M/R steps?

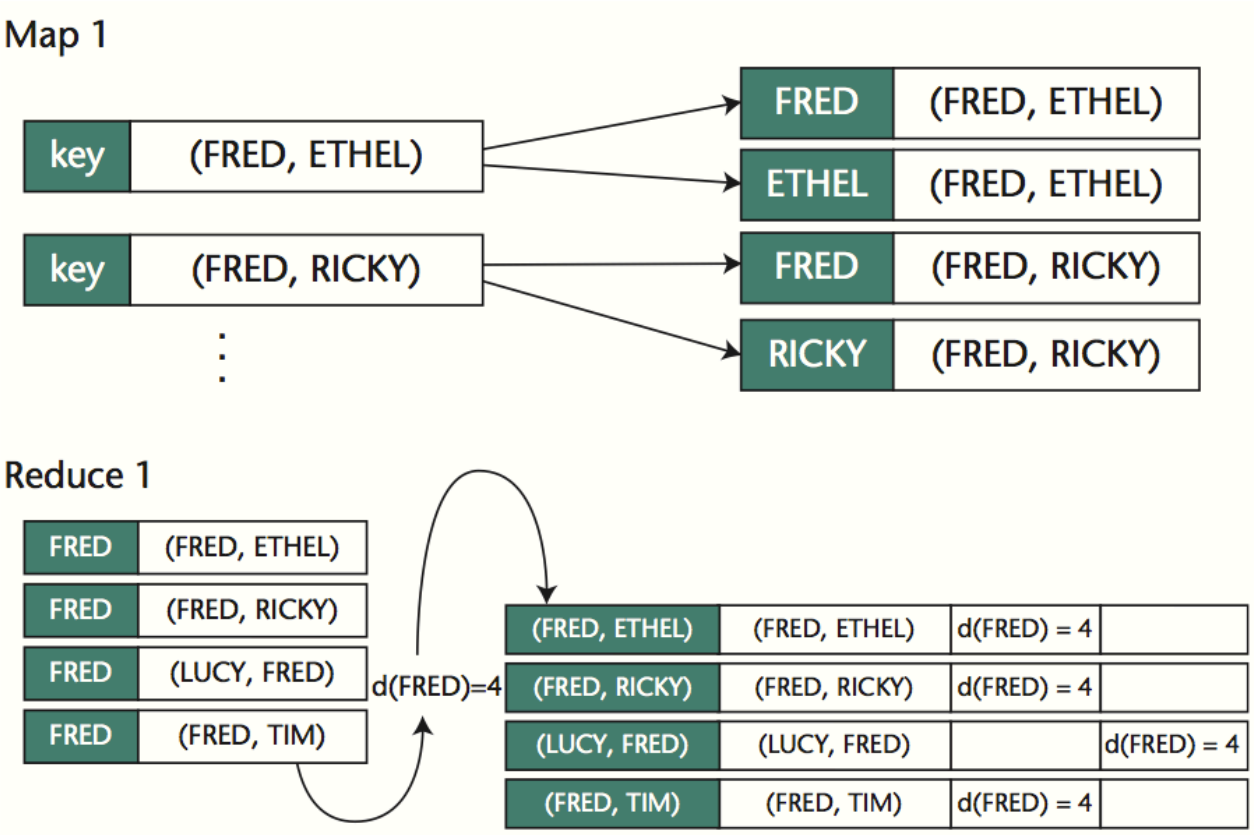


- Jonathan Cohen: Graph Twiddling in a MapReduce World. Computing in Science and Engineering 11(4): 29-41 (2009)

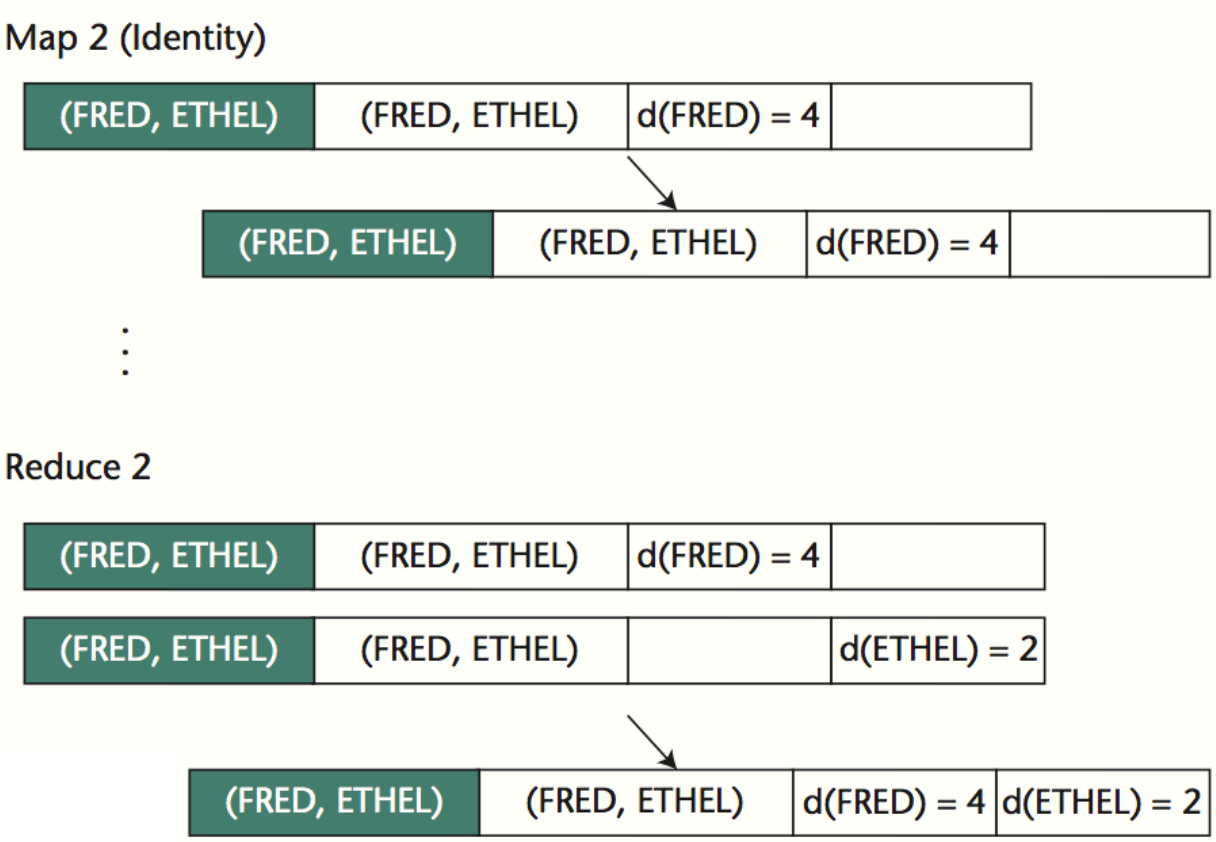
Example: Degree Computation

- Out-degree computation
  - Source Data: (from_vertex, to_vertex)
  - Mapper: (from_vertex, to_vertex) => *key*: from_vertex, *value*: 1
  - Reducer: *key*: vertex, *values*: [1, 1, …] => (vertex, vertex_degree)
- Degree computation
  - Source Data: (from_vertex, to_vertex)
  - **Mapper**: (from_vertex, to_vertex) => *key*: from_vertex, *value*: 1
key: to_vertex, *value*: 1
  - Reducer: *key*: vertex, *values*: [1, 1, …] => (vertex, vertex_degree)
- What if you want to augment each edge with the degrees of the vertices involved: (u, v) => (u, v, d(u), d(v))
  - We can add a second job to add the d(u), and a third to add d(v)
  - Can we do this using less number of jobs?

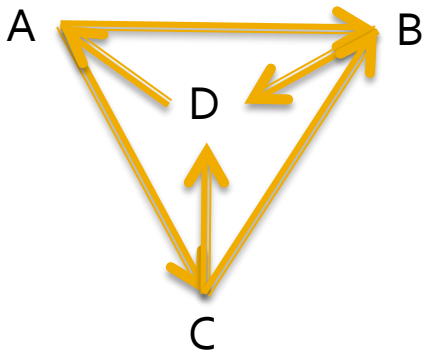
Example: Degree Augmentation (1)



ExAMPLE: Degree AUGMENTATION (1I)



Complete Example



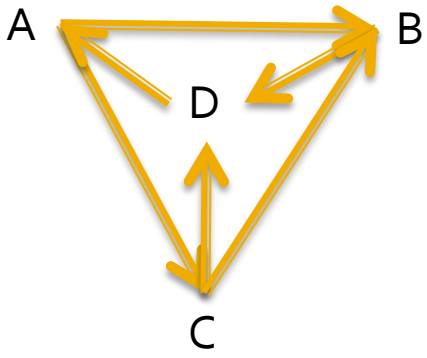
Edges:

(A, B)
(A, C)
(B, D)
(C, B)
(C, D)
(D, A)

Result of map 1:

A | **(A, B)**
B | **(A, B)**
A | (A, C)
C | (A, C)
B | **(B, D)**
D | **(B, D)**
C | (C, B)
B | (C, B)
C | **(C, D)**
D | **(C, D)**
D | (D, A)
A | (D, A)

Complete Example



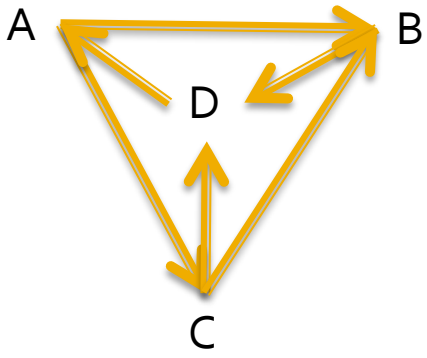
Result of map 1 (sorted by key):

A | (A, B)
A | (A, C)
A | (D, A)
B | (A, B)
B | (B, D)
B | (C, B)
C | (A, C)
C | (C, B)
C | (C, D)
D | (B, D)
D | (C, D)
D | (D, A)

Result of reduce 1

(A, B) | (A, B, 3, -)
(A, C) | (A, C, 3, -)
(D, A) | (D, A, -, 3)
(A, B) | (A, B, -, 3)
(B, D) | (B, D, 3, -)
(C, B) | (C, B, -, 3)
(A, C) | (A, C, -, 3)
(C, B) | (C, B, 3, -)
(C, D) | (C, D, 3, -)
(B, D) | (B, D, -, 3)
(C, D) | (C, D, -, 3)
(D, A) | (D, A, 3, -)

Complete Example



Result of map 2
(sorted by key)

(A, B) | (A, B, 3, -)
(A, B) | (A, B, -, 3)
(A, C) | (A, C, 3, -)
(A, C) | (A, C, -, 3)
(B, D) | (B, D, -, 3)
(B, D) | (B, D, 3, -)
(C, B) | (C, B, -, 3)
(C, B) | (C, B, 3, -)
(C, D) | (C, D, 3, -)
(C, D) | (C, D, -, 3)
(D, A) | (D, A, 3, -)
(D, A) | (D, A, -, 3)

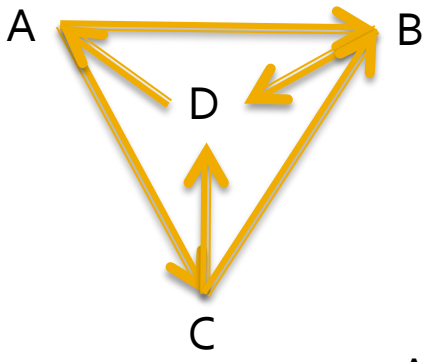
Result of reduce 2
(sorted by key)

(A, B) | (A, B, 3, 3)
(A, C) | (A, C, 3, 3)
(B, D) | (B, D, 3, 3)
(C, B) | (C, B, 3, 3)
(C, D) | (C, D, 3, 3)
(D, A) | (D, A, 3, 3)

Example: PageRank

- Probability of a web surfer being at a particular page under the *random surfer model*
- Random surfer model:
  - The surfer starts from a random page
  - With probability *d*, she continues surfing by following one of the outlinks on the page at random
  - With probability (1-d), she jumps to random page
- Let p_i be the PageRank of page *i*, *N* be the total number of pages, *M*(*i*) be the pages that link to page *i*, and *L*(*i*) be the out-degree of page *i* , we have:
  - $p_i = (1-d) / N + d * \Sigma_{j \in M(i)} p_j / L(j)$
- Iterative implementation
  - Start with all pages having a PageRank of 1/*N*
  - Apply the formula above to update each page's PageRank using the values from the last step
  - Repeat fixed number of times or until convergence
  - Note: pages with no outgoing links need special handling (assumed as if they link to all other pages)

PageRank M/R Style



v | PR(*v*), Neig(*v*)

A | PR(A), [B, C]
B | PR(B), [D]
C | PR(C), [B, D]
D | PR(D), [A]

source data

map

A | PR(A), [B, C]
B | PR(A)/2
C | PR(A)/2

B | PR(B), [D]
D | PR(B)

C | PR(C), [B, D]
B | PR(C)/2
D | PR(C)/2

D | PR(D), [A]
A | PR(D)

reducer

A | PR(A), [B, C]
A | PR(D)

...

B | PR(B), [D]
B | PR(A)/2
B | PR(C)/2

B | (1-*d*)/*N* +
d * (PR(A)/2+PR(C)/2), [D]

C | PR(C), [B, D]
C | PR(A)/2

...

D | PR(D), [A]
D | PR(B)
D | PR(C)/2

...

Note: an additional step is needed to handle nodes with no outgoing links

Vertex Programming (sync.)

- Graph analytics are written from the perspective of a single vertex
  - You are programming a single vertex
  - The vertex program is executed for all of the vertices
- There a few basic principles governing the execution
  - Each vertex maintains its own data
  - The execution proceeds in *supersteps*
  - At each superstep
    - the vertex program is executed for all vertices
  - Between two supersteps
    - Messages sent during the previous superstep are delivered

Supersteps

- During a superstep, the vertex program can do the following:
  - Access the list of messages sent to it during the last superstep
  - Update the state of the vertex
  - Send messages to other vertices (these will be delivered in the next superstep)
  - Vote to halt, if done
- Each vertex has access to vertex ids of its neighbors
- Vertex ids are used for addressing messages
- Messages can be sent to neighbor vertices or any other vertex (as long as the vertex id is learnt by some means, such as through messages exchanged earlier)
- The execution continues until no more supersteps can be performed, which happens when:
  - There are no pending messages
  - There are no unhalted vertices

BSP & Pregel

- Vertex programs can be executed in a scalable manner using the Bulk Synchronous Processing (BSP) paradigm
- Pregel system by Google (research paper, code not available) does that
- Vertices are distributed to machines using some partitioning
  - The default is a hash based partitioning (on the vertex id)
- At each superstep, each machine executes the vertex program for the vertices it hosts (keeps the state for those vertices as well)
- At the end of the superstep, messages that need to cross machine boundaries are transported
- Pregel also supports additional abstractions
  - Aggregations: Reduction functions that can be applied on vertex values
  - Combiners: Reduction functions that can applied to messages destined to the same vertex from different vertices
  - Ability to remove vertices (morphing the graph)

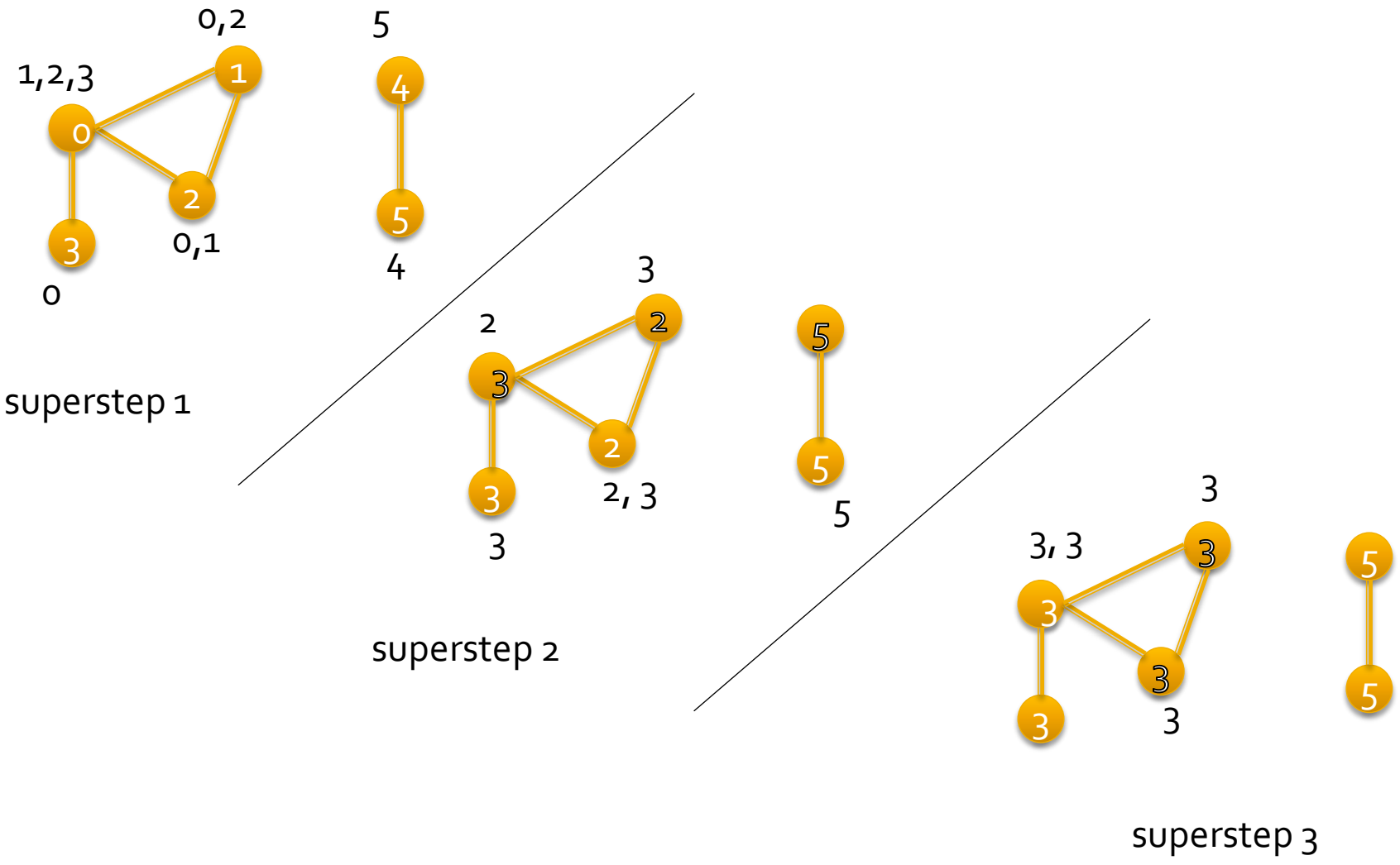
Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski: Pregel: a system for large-scale graph processing. SIGMOD Conference 2010: 135-146

Example: Connected Components

- Vertex state
  - Just an int value, representing the id of the connected component the vertex belongs
- The vertex program

```
if (superstep() == 0)
  getVertexValue() = getVertexId();
  sendMessageToAllNeighbors(getVertexId());
} else {
  int mid = getVertexValue();
  for (msg in getReceivedMessages())
    mid = max(mid, msg.getValue());
  if (mid != getVertexValue()) {
    getVertexValue() = mid;
    sendMessageToAllNeighbors(mid);
  }
  VoteToHalt();
}
```

EXAMPLE: Execution



Example: PageRank

- Vertex state
  - Just a double value, representing the PageRank
- The vertex program

```
if (superstep() >= 1) {
  double sum = 0;
  for (msg in getReceivedMessages())
    sum += msg.getValue();
  getVertexValue() = 0.15 / getNumVertices() +
                     0.85 * sum;
}
if (superstep() < 30) {
  int n = getNumOutEdges();
  sendMessageToOutNeighbors(getVertexValue()/n);
} else {
  VoteToHalt();
}
```

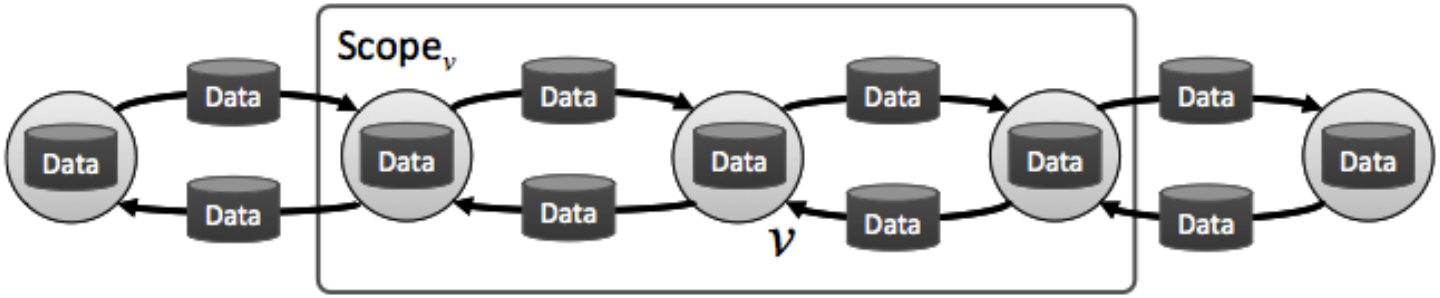
Surprisingly simple

Systems Issues

- Scalability
  - Better than M/R for most graph analytics
  - Minimizing communication is key (many papers on improved partitioning to take advantage of high locality present in graph analytics)
  - Skew could be an issue for power graphs where the are a few number of very high degree vertices (results in imbalance)
- Fault tolerance
  - Checkpointing between supersteps, say every *x* supersteps, or every *y* seconds
- Example Open Source Systems
  - Apache Giraph: Pregel-like
  - Apache HAMA: More general BSP framework for data mining & machine learning

ASYNCHRONOUS VP & GraphLab

- GraphLab targets not just graph processing, but also iterative Machine Learning and Data Mining algorithms
- Similar to Pregel but with important differences
  - It is *asynchronous* and supports *dynamic execution schedules*
- Vertex programs in GraphLab
  - Access vertex data, adjacent edge data, adjacent vertex data
    - These are called the *scope*
    - No messages as in Pregel, but similar in nature
  - Update any of the things in *scope* as a result of execution
  - Return a list of vertices, that will be scheduled for execution in the future

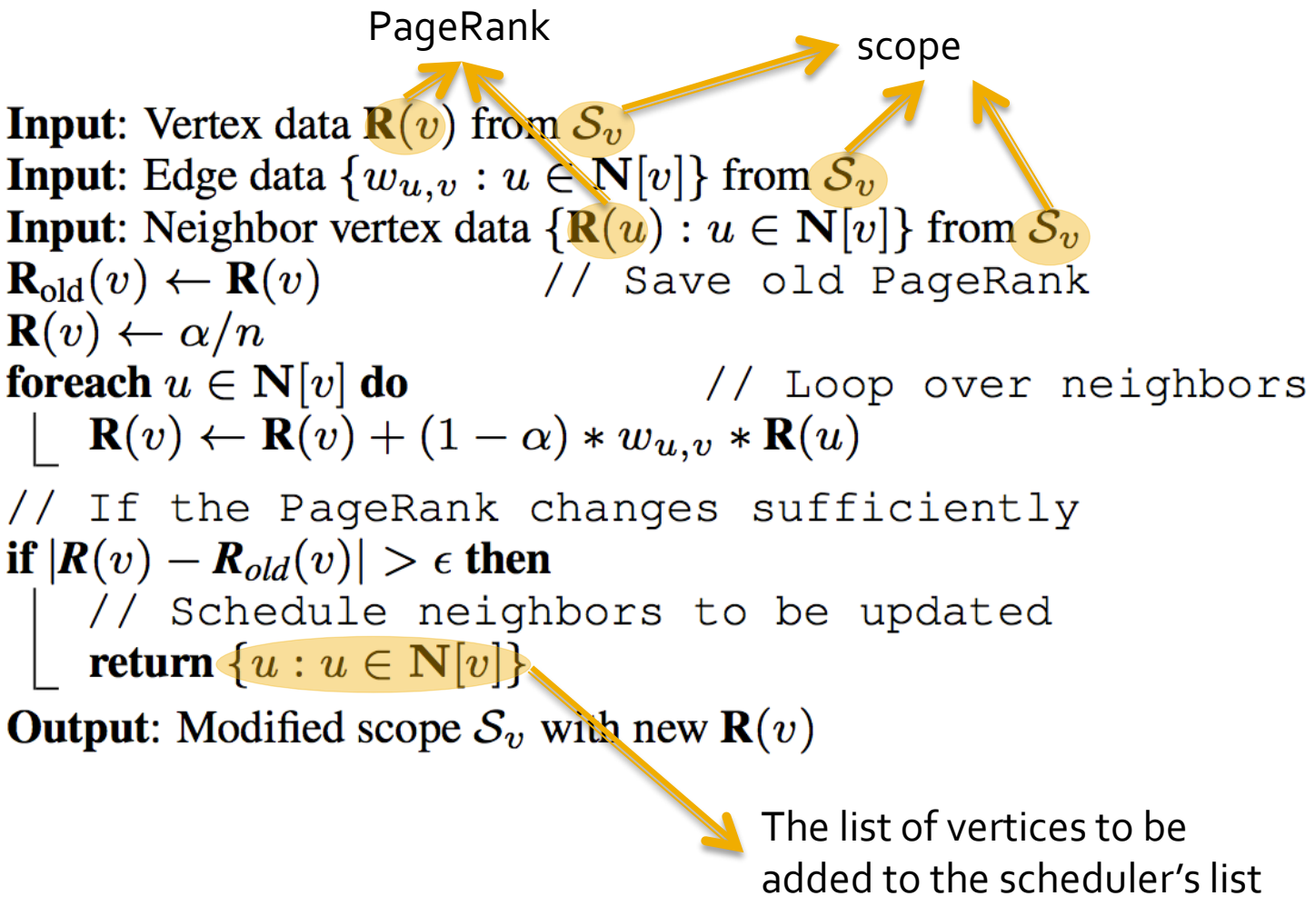


GRAPHLAB Continued

- Asynchronous
  - Pregel works in supersteps, with synchronization in-between
  - Graphlab works asynchronously
    - It is shown that this improves convergence for many iterative data mining algorithms
- Dynamic execution schedule
  - Pregel executes each vertex at each superstep
  - In GraphLab, new vertices to be executed are determined as a result of previous vertex executions
- GraphLab also supports
  - Configuring the *consistency model*: determines the extent to which computation can overlap
  - Configuring the *scheduling*: determines the order in which vertices are scheduled (synchronous, round-robin, etc.)
- GraphLab has multi-core (single machine) and distributed versions

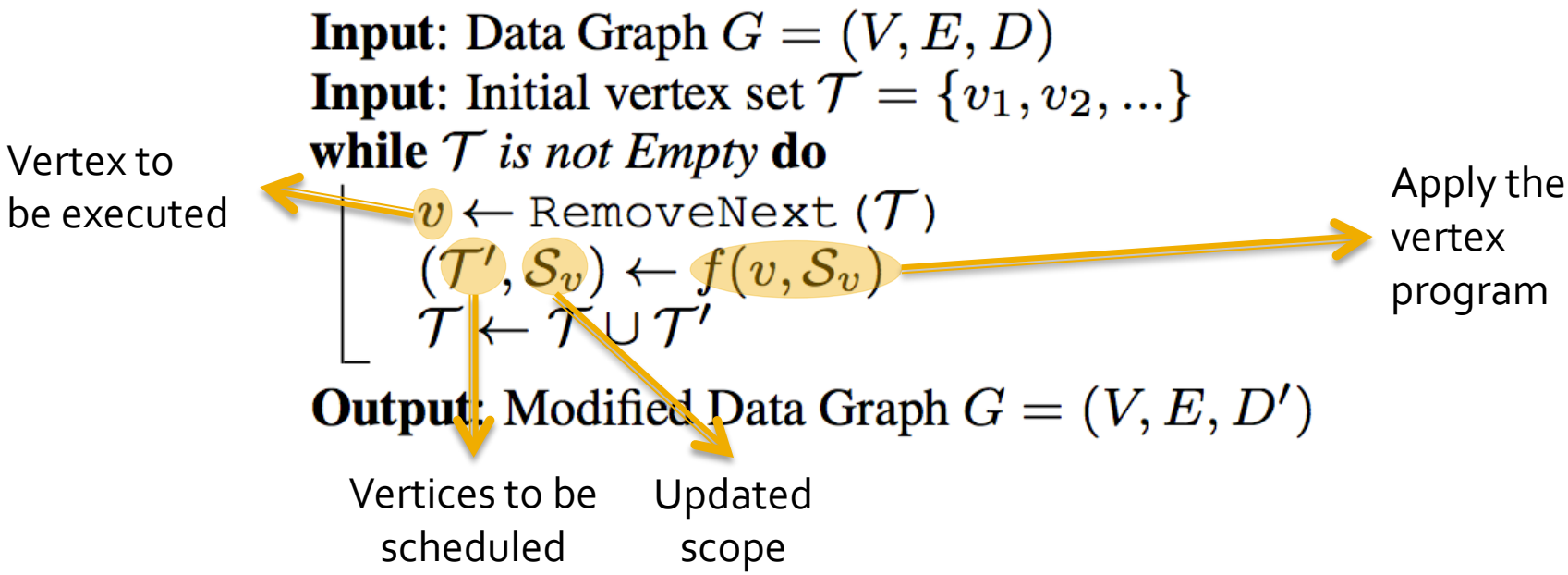
- Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein: GraphLab: A New Framework For Parallel Machine Learning. UAI 2010: 340-349
- Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein: Distributed GraphLab: A Framework for Machine Learning in the Cloud. PVLDB 5(8): 716-727 (2012)

EXAMPLE: PageRank



Understanding Dynamic Scheduling

- A very high level view of the execution model



- Consistency model adjusts how the execution is performed in parallel
- Scheduling adjusts how the RemoveNext method is implemented

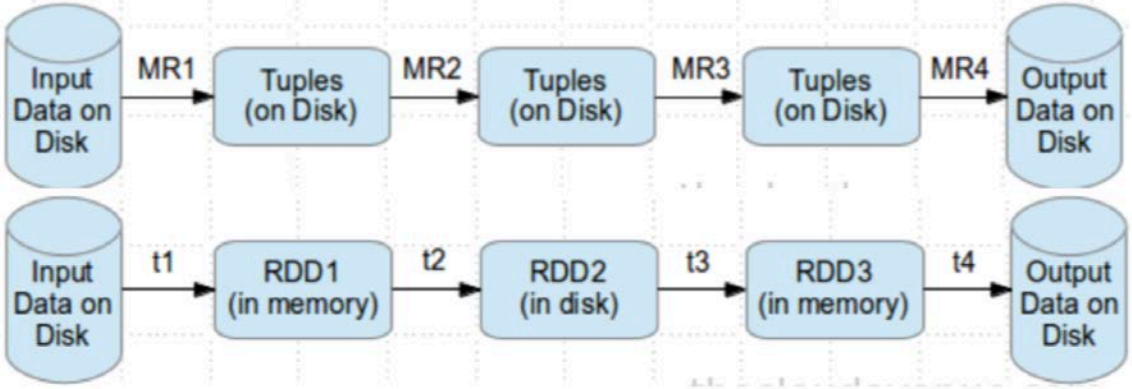
More on GraphLab

- GraphLab also supports
  - Global aggregations over vertex values, which are read-only accessible to vertex programs
  - Unlike Pregel, these are computed continuously in the background
- PowerGraph is a GraphLab variant
  - Specially designed for scale-free graphs
    - Degree distribution follows a power law
    - $P(k) \sim k^{-y}$ k: degree, y: typically in the range $2 < y < 3$
  - Main idea is to decompose the vertex program into 3 steps
    - Gather: Collect data from the scope
    - Apply: Compute the value of the central vertex
    - Scatter: Update the data on adjacent vertices
  - This way a single vertex with very high-degree can be distributed over multiple nodes (partition edges not vertices)
- GraphChi is another GraphLab variant
  - It is designed for disk-based, single machine processing
  - The main idea is a disk layout technique that can be used to execute vertex programs by doing mostly sequential I/O (potentially parallel)

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI 2012: 17-30
Aapo Kyrola, Guy E. Blelloch, Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. OSDI 2012: 31-46

Spark GraphX

- Apache Spark GraphX is a graph processing system on Apache Spark
- Apache Spark is a Big Data processing framework based on the concept of RDDs (Resilient Distributed Datasets)
- RDD: Read-only, partitioned collection of records
  - Distributed over a set of nodes, replicated for fault-tolerance
  - An RDD is either created from input data on disk, or by applying a transformation over existing RDDs
  - If the RDD fits into memory of multiple nodes, no disk processing is involved
- Comparison to M/R



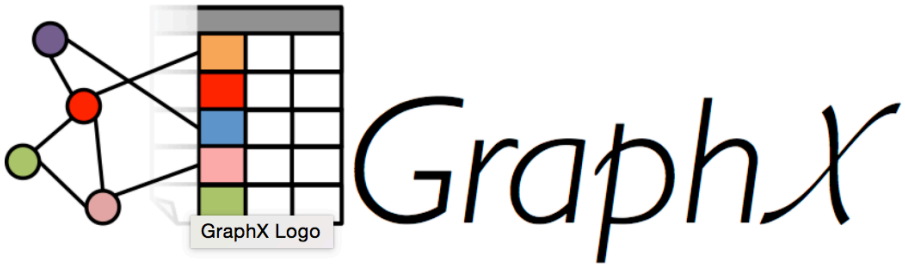
Apache Spark

- RDD transformations are typically implemented via M/R-like techniques, but without the disk being involved (as long as there is enough memory)
- Apache Spark
  - Provides interactive analysis via RDD abstraction
  - Can work in the Hadoop ecosystem: HDFS, YARN
  - Supports Scala, Python, and Java languages
- Consider the word count example:

```
file = spark.textFile("hdfs://...")
counts = file.flatMap(lambda line: line.split(" "))
              .map(lambda word: (word, 1))
              .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

Apache Spark GraphX

- Introduces a Graph abstraction in Spark
  - a directed multi-graph with properties attached to each vertex and edge
- Exposes a set of fundamental operations
- Provides an optimized version of the Pregel API
- Comes with a collection of graph algorithms
- At this time only Scala APIs are available



GraphX

- Graph: a fundamental data structure on top of Spark RDDs
  - with the following logical structure

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

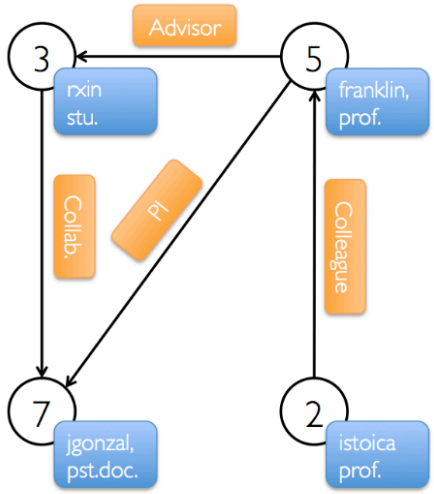
  - VertexRDD[VD] is an optimized version of RDD[(VertexID, VD)]
  - EdgeRDD[ED] is an optimized version of RDD[Edge[ED]]

Example

```
// Assume the SparkContext has already been constructed
val sc: SparkContext
// 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")))
// Build the initial Graph
val graph = Graph(users, relationships)
```

Normally you would load data from HDFS

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

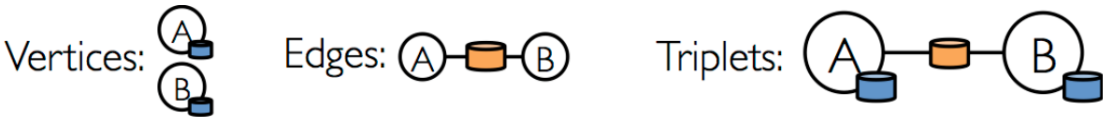
Simple Views on Graphs

- Can access vertex and edge properties easily

```
// List vertices (users) who are professors
graph.vertices.filter {
    case (id, (name, job)) => job == “professor”
}
```

- A very useful view is the *triplets* view

```
for (triplet <- graph.triplets.collect) {
 /**
  * Triplet has the following Fields:
  * triplet.srcAttr: (String, String)
  * triplet.dstAttr: (String, String)
  * triplet.attr: Int
  * triplet.srcId: VertexId
  * triplet.dstId: VertexId
  */
}
```



GRAPH OPERATIONS

- Information about the Graph

```
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]
```

- Let's compute some statistics:

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

Graph Operations

- Transform vertex and edge attributes

```
// map from vertex attributes (type VD) into new vertex attributes (type VD2)
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
// map from edge attributes (type ED) into a new edge attributes (type VD2)
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
// same as before, but have access to edge triplets
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
```

- Join graphs with RDDs

```
def joinVertices[U]
(table: RDD[(VertexID, U)])          // table to join with
(mapFunc: (VertexID, VD, U) => VD)   // vertex map function
: Graph[VD, ED]
```

- There is also an outer join version

```
def joinVertices[U]
(table: RDD[(VertexID, U)])
(mapFunc: (VertexID, VD, Option[U]) => VD)   // note the Option
: Graph[VD, ED]
```

Let's Put These into Use

- Assume we have a simple graph, where each vertex contains as data, a name-age pair
- We want to associate each vertex with a custom User object that will hold the name and age, as well as the in and out degrees of the vertex

```
// Define a class to more clearly model the user property
case class User(name: String, age: Int, inDeg: Int, outDeg: Int)
// Use vertex transformation to create a user Graph
val initialUserGraph: Graph[User, Int] =
    graph.mapVertices{ case (id, (name, age)) => User(name, age, 0, 0) }
// Use outer join to populate the degree information
val userGraph = initialUserGraph.outerJoinVertices(initialUserGraph.inDegrees) {
  case (id, u, inDegOpt) => User(u.name, u.age, inDegOpt.getOrElse(0), u.outDeg)
}.outerJoinVertices(initialUserGraph.outDegrees) {
  case (id, u, outDegOpt) => User(u.name, u.age, u.inDeg, outDegOpt.getOrElse(0))
}
```

Aggregate Neighbors

- Collect neighbor ids or their vertex attributes

```
// Collect neighbor ids
def collectNeighborIds(edgeDirection: EdgeDirection):
    VertexRDD[Array[VertexID]]
// Collect neighbors
def collectNeighbor(edgeDirection: EdgeDirection):
    VertexRDD[Array[(VertexID,VD)]]
```

- Map/Reduce triplets in the graph

```
def mapReduceTriplets[MsgType](
  // Function from an edge triplet to a collection of messages (i.e., Map)
  map: EdgeTriplet[VD, ED] => Iterator[(VertexId, MsgType)],
  // Function that combines messages to the same vertex (i.e., Reduce)
  reduce: (MsgType, MsgType) => MsgType)
: VertexRDD[MsgType]
```

Let's Do Some Map/Reduce

- Assume we wan to find the oldest follower for each user

```
val oldestFollower: VertexRDD[(String, Int)] =
    userGraph.mapReduceTriplets[(String, Int)]
            (
      // For each edge send a message to the destination
      // vertex with the attribute of the source vertex
      edge => Iterator((edge.dstId, (edge.srcAttr.name,
                                     edge.srcAttr.age))),
      // To combine messages take the message for the
      // older follower
      (a, b) => if (a._2 > b._2) a else b
    )
```

Morphing the Graph

- Operations that create a new graph

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

// create a subgraph
def subgraph(
  epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
  vpred: (VertexID, VD) => Boolean = ((v, d) => true))
: Graph[VD, ED]

// reduces edges into one
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
```

Pregel-style Computation

```
def pregel[Msg]
  (initialMsg: A, // initial message
   maxIter: Int = Int.MaxValue,
   activeDir: EdgeDirection = EdgeDirection.Out)
  (vprog: (VertexId, VD, A) => VD, // vertex program
   // message sender
   sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
   mergeMsg: (A, A) => A) // message combiner
: Graph[VD, ED]
```

Logically, what happens in a superstep is:

- Run the vertex program, where each vertex gets the **combined** messages from earlier step (initially the initial message for each vertex)
- Update the graph vertex data with the results from running the vertex program
- Run a map/reduce triplets operation, where the message sender is used as the mapper and the message combiner is used as a reducer (generates the combined message to be used in the next supersetep)

E.g.: Single-source Shortest Path

```
val sourceId: VertexId = 42 // the single source

// initialize the graph such that all vertices except the source have
// distance infinity, and the source has distance 0
val initialGraph = graph.mapVertices(
    (id, _) => if (id == sourceId) 0.0 else Double.PositiveInfinity)

// run Pregel-stype computation
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // vertex program
  triplet => {  // message sender
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // message combiner
)

println(sssp.vertices.collect.mkString("\n"))
```

Built-in 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]
```

Questions

- ???