



Large Scale Graph Processing - GraphX, Giraph++, and Pegasus

Amir H. Payberah
payberah@kth.se
09/10/2018

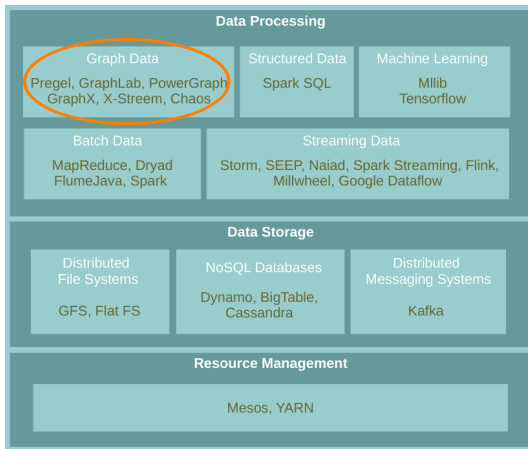




The Course Web Page

<https://id2221kth.github.io>

Where Are We?







Graph Algorithms Challenges

- ▶ Difficult to extract parallelism based on partitioning of the data.
- ▶ Difficult to express parallelism based on partitioning of computation.

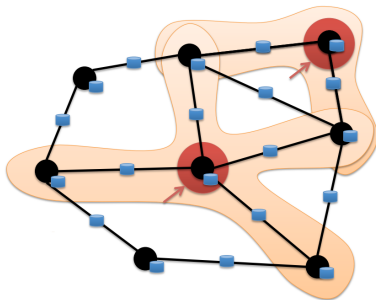


Different Approached to Process Large Scale Graphs

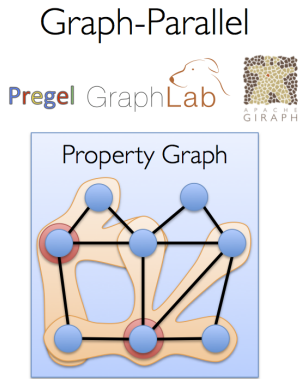
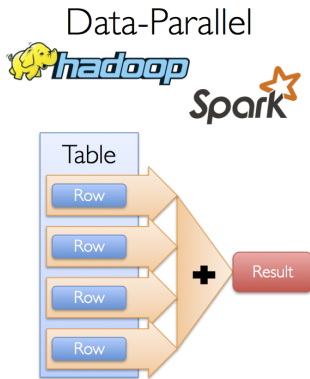
- ▶ Think like a **vertex**
- ▶ Think like an **edge**
- ▶ Think like a **table**
- ▶ Think like a **graph**
- ▶ Think like a **matrix**

Think Like a Table

Graph-Parallel Processing Model



Data-Parallel vs. Graph-Parallel Computation



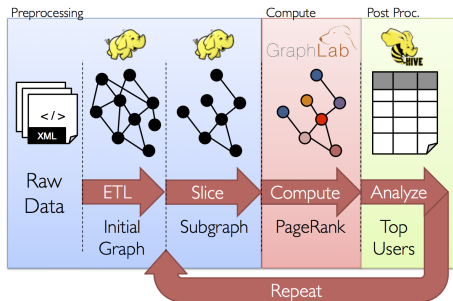


Motivation (2/3)

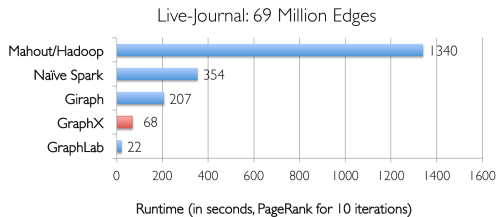
- ▶ **Graph-parallel** computation: **restricting** the types of computation to achieve **performance**.

Motivation (2/3)

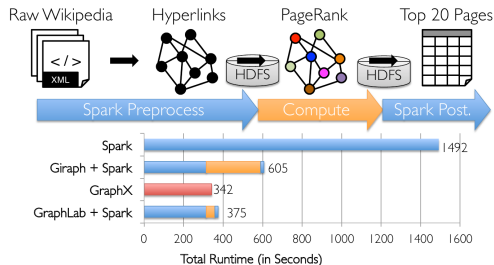
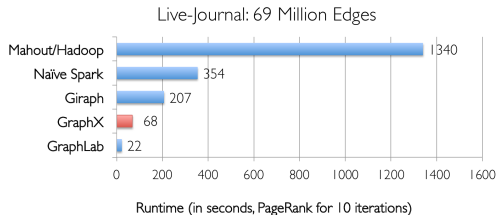
- ▶ **Graph-parallel** computation: **restricting** the types of computation to achieve **performance**.
- ▶ The same restrictions make it **difficult** and **inefficient** to express many stages in a typical graph-analytics **pipeline**.



Motivation (3/3)

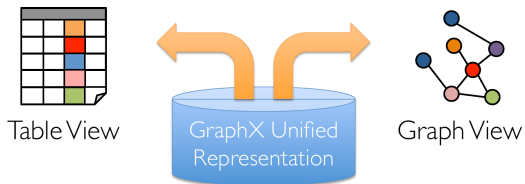


Motivation (3/3)



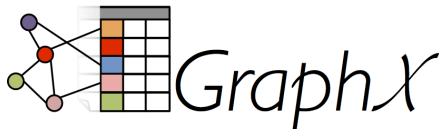
Think Like a Table

- ▶ Unifies **data-parallel** and **graph-parallel** systems.
- ▶ **Tables** and **Graphs** are **composable views** of the **same physical data**.



GraphX

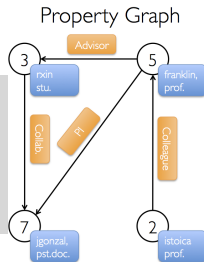
- ▶ **GraphX** is the library to perform **graph-parallel** processing in **Spark**.
- ▶ **In-memory** caching.
- ▶ **Lineage-based** fault tolerance.



The Property Graph Data Model

- ▶ Spark represent **graph** structured data as a **property graph**.
- ▶ It is logically represented as a pair of **vertex** and **edge property collections**.
 - **VertexRDD** and **EdgeRDD**

```
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```



Vertex Table

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

Edge Table

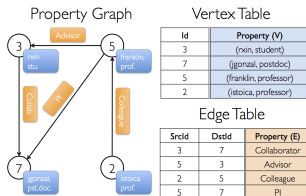
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI


The Vertex Collection

- **VertexRDD**: contains the vertex properties **keyed by the vertex ID**.

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

// VD: the type of the vertex attribute
abstract class VertexRDD[VD] extends RDD[(VertexId, VD)]
```



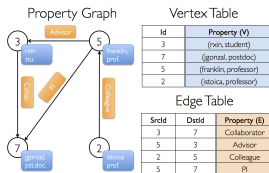
Vertices: 

The Edge Collection

- **EdgeRDD**: contains the edge properties **keyed by the source and destination vertex IDs**.

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

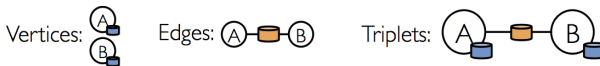
// ED: the type of the edge attribute
case class Edge[ED](srcId: VertexId, dstId: VertexId, attr: ED)
abstract class EdgeRDD[ED] extends RDD[Edge[ED]]
```



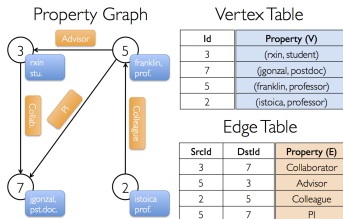
Edges: A  B

The Triplet Collection

- ▶ The **triplets collection** consists of each **edge** and its **corresponding source and destination vertex** properties.
- ▶ It logically **joins the vertex and edge properties**: `RDD[EdgeTriplet[VD, ED]]`.
- ▶ The `EdgeTriplet` class extends the `Edge` class by adding the `srcAttr` and `dstAttr` members, which contain the **source and destination properties** respectively.



Building a Property Graph



```

val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
    (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

val relationships: RDD[Edge[String]] = sc.parallelize(Array(Edge(3L, 7L, "collab"),
    Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"), Edge(5L, 1L, "-")))

val defaultUser = ("John Doe", "Missing")

val graph: Graph[(String, String), String] = Graph(users, relationships, defaultUser)
  
```



Graph Operators

- ▶ Information about the graph
- ▶ Property operators
- ▶ Structural operators
- ▶ Joins
- ▶ Aggregation
- ▶ Iterative computation
- ▶ ...



Information About The Graph (1/2)

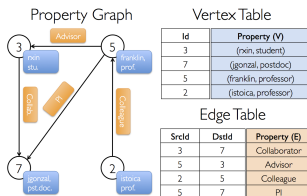
► Information about the graph

```
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]]
```

Information About The Graph (2/2)



```
// Constructed from above
val graph: Graph[(String, String), String]

// Count all users which are postdocs
graph.vertices.filter { case (id, (name, pos)) => pos == "postdoc" }.count

// Count all the edges where src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```




Property Operators

- ▶ Transform vertex and edge attributes
- ▶ Each of these operators yields a new graph with the vertex or edge properties modified by the user defined map function.

```
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]
```



Property Operators

- ▶ Transform vertex and edge attributes
- ▶ Each of these operators yields a new graph with the vertex or edge properties modified by the user defined map function.

```
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]
```

```
val relations: RDD[String] = graph.triplets.map(triplet =>  
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)  
relations.collect.foreach(println)
```



Property Operators

- ▶ Transform vertex and edge attributes
- ▶ Each of these operators yields a new graph with the vertex or edge properties modified by the user defined map function.

```
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]
```

```
val relations: RDD[String] = graph.triplets.map(triplet =>  
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)  
relations.collect.foreach(println)
```

```
val newGraph = graph.mapTriplets(triplet =>  
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1)  
newGraph.edges.collect.foreach(println)
```



Structural Operators

- ▶ `reverse` returns a new graph with all the edge directions reversed.

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

Structural Operators

- ▶ **reverse** returns a new graph with all the edge directions reversed.
- ▶ **subgraph** takes vertex/edge predicates and returns the graph containing only the vertices/edges that satisfy the given predicate.

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

```
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
graph.vertices.collect.foreach(println)
validGraph.vertices.collect.foreach(println)

// Restrict the answer to the valid subgraph
val validUserGraph = graph.mask(validGraph)
```

Structural Operators

- ▶ `reverse` returns a new graph with all the edge directions reversed.
- ▶ `subgraph` takes vertex/edge predicates and returns the graph containing only the vertices/edges that satisfy the given predicate.
- ▶ `mask` constructs a `subgraph` of the input graph.

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

```
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
graph.vertices.collect.foreach(println)
validGraph.vertices.collect.foreach(println)

// Restrict the answer to the valid subgraph
val validUserGraph = graph.mask(validGraph)
```



Join Operators

- ▶ `joinVertices` joins the `vertices` with the `input RDD`.
 - Returns a new graph with the vertex properties obtained by applying the user defined `map` function to the `result of the joined vertices`.
 - Vertices without a matching value in the RDD retain their `original value`.

```
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD): Graph[VD, ED]
```

Join Operators

- ▶ `joinVertices` joins the **vertices** with the **input RDD**.
 - Returns a new graph with the vertex properties obtained by applying the user defined **map** function to the **result of the joined vertices**.
 - Vertices without a matching value in the RDD retain their **original value**.

```
def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD): Graph[VD, ED]
```

```
val rdd: RDD[(VertexId, String)] = sc.parallelize(Array((3L, "phd")))
val joinedGraph = graph.joinVertices(rdd)((id, user, role) => (user._1, role + " " + user._2))
joinedGraph.vertices.collect.foreach(println)
```




Aggregation (1/2)

- `aggregateMessages` applies a user defined `sendMsg` function to each `edge triplet` in the graph and then uses the `mergeMsg` function to aggregate those messages at `their destination vertex`.

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

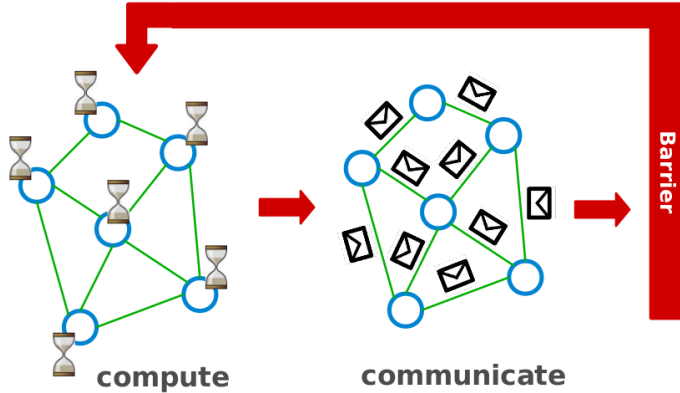


Aggregation (2/2)

```
// count and list the name of friends of each user
val profs: VertexRDD[(Int, String)] = validUserGraph.aggregateMessages[(Int, String)](
  // map
  triplet => {
    triplet.sendToDst((1, triplet.srcAttr._1))
  },
  // reduce
  (a, b) => (a._1 + b._1, a._2 + " " + b._2)
)

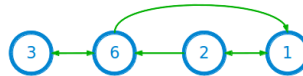
profs.collect.foreach(println)
```

Iterative Computation (1/9)



Iterative Computation (2/9)

```
i_val := val  
  
for each message m  
  if m > val then val := m  
  
if i_val == val then  
  vote_to_halt  
else  
  for each neighbor v  
    send_message(v, val)
```



Super step 0

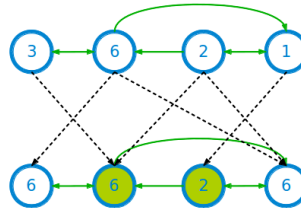
Iterative Computation (3/9)

```

i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
  
```



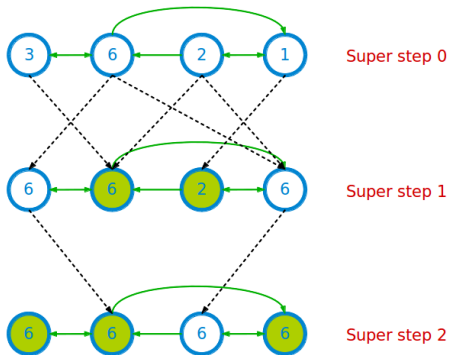
Iterative Computation (4/9)

```

i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
  
```



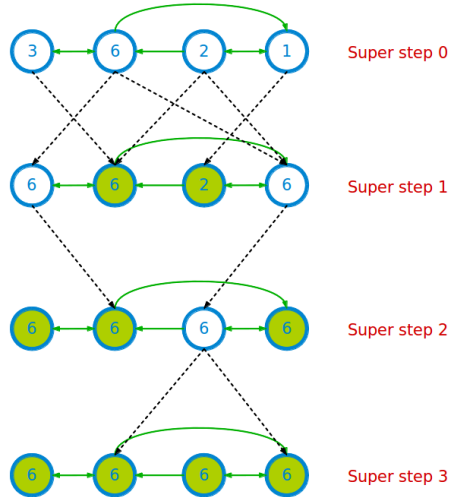
Iterative Computation (5/9)

```

i_val := val

for each message m
  if m > val then val := m

if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
  
```





Iterative Computation (6/9)

- `pregel` takes two argument lists: `graph.pregel(list1)(list2)`.

```
def pregel[A]  
  (initialMsg: A, maxIter: Int = Int.MaxValue, activeDir: EdgeDirection = EdgeDirection.Out)  
  (vprog: (VertexId, VD, A) => VD, sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],  
   mergeMsg: (A, A) => A):  
  Graph[VD, ED]
```


Iterative Computation (6/9)

- ▶ `pregel` takes two argument lists: `graph.pregel(list1)(list2)`.
- ▶ The `first list` contains `configuration parameters`
 - The initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).

```
def pregel[A]  
  (initialMsg: A, maxIter: Int = Int.MaxValue, activeDir: EdgeDirection = EdgeDirection.Out)  
  (vprog: (VertexId, VD, A) => VD, sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],  
   mergeMsg: (A, A) => A):  
  Graph[VD, ED]
```

Iterative Computation (6/9)

- ▶ `pregel` takes two argument lists: `graph.pregel(list1)(list2)`.
- ▶ The **first list** contains **configuration parameters**
 - The initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).
- ▶ The **second list** contains the **user defined functions**.
 - For receiving messages (the vertex program `vprog`), computing messages (`sendMsg`), and combining messages (`mergeMsg`).

```
def pregel[A]  
  (initialMsg: A, maxIter: Int = Int.MaxValue, activeDir: EdgeDirection = EdgeDirection.Out)  
  (vprog: (VertexId, VD, A) => VD, sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],  
   mergeMsg: (A, A) => A):  
  Graph[VD, ED]
```

Iterative Computation (7/9)

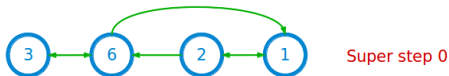
```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD

val initialMsg = -9999

val vertices: RDD[(VertexId, (Int, Int))] = sc.parallelize(Array((1L, (1, -1)),
    (2L, (2, -1)), (3L, (3, -1)), (6L, (6, -1))))

val relationships: RDD[Edge[Boolean]] = sc.parallelize(Array(Edge(1L, 2L, true),
    Edge(2L, 1L, true), Edge(2L, 6L, true), Edge(3L, 6L, true), Edge(6L, 1L, true),
    Edge(6L, 3L, true)))

val graph = Graph(vertices, relationships)
```



Iterative Computation (8/9)

```
// the function for receiving messages
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) = {
  if (message == initialMsg)
    value
  else
    (math.max(message, value._1), value._1)
}
```

Iterative Computation (8/9)

```
// the function for receiving messages
```

```
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) = {  
  if (message == initialMsg)  
    value  
  else  
    (math.max(message, value._1), value._1)  
}
```

```
// the function for computing messages
```

```
def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]): Iterator[(VertexId, Int)] = {  
  val sourceVertex = triplet.srcAttr  
  if (sourceVertex._1 == sourceVertex._2)  
    Iterator.empty  
  else  
    Iterator((triplet.dstId, sourceVertex._1))  
}
```



```
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) = {
  if (message == initialMsg)
    value
  else
    (math.max(message, value._1), value._1)
}
```

```
def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]): Iterator[(VertexId, Int)] = {
  val sourceVertex = triplet.srcAttr
  if (sourceVertex._1 == sourceVertex._2)
    Iterator.empty
  else
    Iterator((triplet.dstId, sourceVertex._1))
}
```

```
def mergeMsg(msg1: Int, msg2: Int): Int = math.max(msg1, msg2)
```



Iterative Computation (9/9)

```
val minGraph = graph.pregel(initialMsg,
                             Int.MaxValue,
                             EdgeDirection.Out)(
    vprog, // the function for receiving messages
    sendMsg, // the function for computing messages
    mergeMsg) // the function for combining messages

minGraph.vertices.collect.foreach{
  case (vertexId, (value, original_value)) => println(value)
}
```



GraphFrames

- ▶ **GraphFrames** extends **GraphX** to provide a **DataFrame** API.
- ▶ To build a GraphFrame we need to define the **vertices and edges** as **DataFrames**.
- ▶ `spark-shell --packages graphframes:graphframes:0.6.0-spark2.3-s_2.11`
 - You may need to delete `.ivy2` from your home folder.



Querying the GraphFrames

```
import org.graphframes._
import org.apache.spark.sql.SQLContext

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val userDF = sqlContext.createDataFrame(Array(("rxin", "student"), ("jgonzal", "postdoc"),
      ("franklin", "prof"), ("istoica", "prof"))).toDF("id", "role")

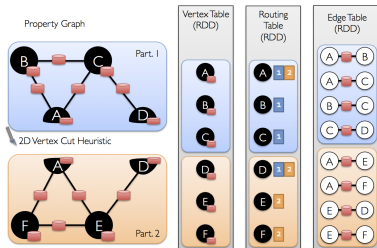
val relationshipsDF = sqlContext.createDataFrame(Array(("rxin", "jgonzal", "collab"),
      ("franklin", "rxin", "advisor"), ("istoica", "franklin", "colleague"),
      ("franklin", "franklin", "pi"))).toDF("src", "dst", "relationship")

val graphDF = GraphFrame(userDF, relationshipsDF)

graphDF.edges.where("src = 'franklin'").groupBy("src", "dst").count().show
```

Graph Representation

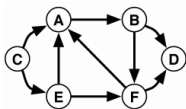
- ▶ **Vertex-cut** partitioning
- ▶ Representing graphs using **two RDDs**: **edge-collection** and **vertex-collection**
- ▶ **Routing table**: a **logical map** from a vertex id to the set of edge partitions that contains adjacent edges.



Think Like a Graph

Motivation (1/2)

- ▶ **Vertex-centric** programming model.
 - Operate on a **vertex** and its edges.
 - Communication to other vertices, via **message passing** (Pregel), or **shared memory** (GraphLab).
- ▶ Divide input graphs into **partitions**.



Partition	Vertex	Edge List
P1	(A)	B
	(B)	D F

P2	(C)	A E
	(D)	

P3	(E)	A F
	(F)	A D

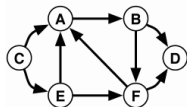


Motivation (2/2)

- ▶ In the **vertex-centric** model, a vertex is very **short sighted**.
 - A vertex has information about **its immediate neighbors**.
 - Information is propagated through graphs slowly, **one hop at a time**.
- ▶ **Graph-centric** programming paradigm is proposed to overcome this **limitation**.

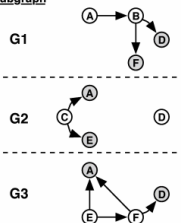
Think Like a Graph

	Think Like a Vertex	Think Like a Graph
Partition	A collection of vertices	A proper subgraph
Computation	A vertex and its edges	A subgraph
Communication	1-hop at a time, e.g., $A \rightarrow B \rightarrow D$	Multiple-hops at a time, e.g., $A \rightarrow D$



Partition	Vertex	Edge List
P1	A	B
	B	D F
P2	C	A E
	D	
P3	E	A F
	F	A D

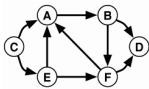
Subgraph



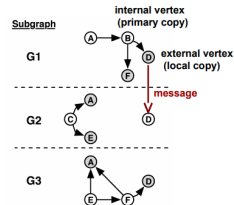
Giraph++

Giraph++

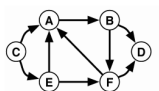
- ▶ Expose **subgraphs** to programmers.
- ▶ **Internal** vertices vs. **boundary** vertices.



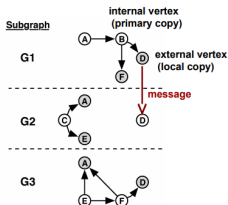
Partition	Vertex	Edge List
P1	A	B
	B	D F
P2	C	A E
	D	
P3	E	A F
	F	A D



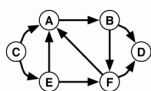
- ▶ Expose **subgraphs** to programmers.
- ▶ **Internal** vertices vs. **boundary** vertices.
 - Information **exchange between internal vertices** of a partition is **immediate**.
 - Messages are only sent from **boundary vertices** of a partition to internal vertices of a **different partition**.



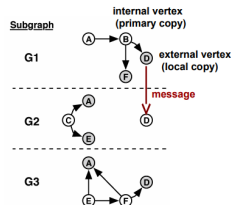
Partition	Vertex	Edge List
P1	A	B
	B	D F
P2	C	A E
	D	
P3	E	A F
	F	A D



- ▶ Expose **subgraphs** to programmers.
- ▶ **Internal** vertices vs. **boundary** vertices.
 - Information **exchange between internal vertices** of a partition is **immediate**.
 - Messages are only sent from **boundary vertices** of a partition to internal vertices of a **different partition**.
- ▶ A vertex is an **internal** vertex in **exactly one** subgraph, but it can be a **boundary** vertex in **zero or more** subgraphs.



Partition	Vertex	Edge List
P1	A	B
	B	D F
P2	C	A E
	D	
P3	E	A F
	F	A D





Execution Model (1/2)

- ▶ A program is executed in **sequence of supersteps**.
- ▶ **Supersteps** are separated by **global synchronization barriers**.



Execution Model (1/2)

- ▶ A program is executed in **sequence of supersteps**.
- ▶ **Supersteps** are separated by **global synchronization barriers**.
- ▶ In each superstep, the computation is performed on the **whole subgraph in a partition**.



Execution Model (1/2)

- ▶ A program is executed in **sequence of supersteps**.
- ▶ **Supersteps** are separated by **global synchronization barriers**.
- ▶ In each superstep, the computation is performed on the **whole subgraph in a partition**.
- ▶ Like in Pregel, each **internal vertex** of a partition has **two states**: **active or inactive**.
- ▶ A **boundary vertex** does not have any state.



Execution Model (2/2)

- ▶ Differentiate **internal messages** and **external messages**.



Execution Model (2/2)

- ▶ Differentiate **internal messages** and **external messages**.
- ▶ What messages can be used in **local computation**?
 - **External** messages from **previous superstep** (global **synchronous** computation).
 - **Internal** messages from **previous + current superstep** (local **asynchronous** computation).



Execution Model (2/2)

- ▶ Differentiate **internal messages** and **external messages**.
- ▶ What messages can be used in **local computation**?
 - **External** messages from **previous superstep** (global **synchronous** computation).
 - **Internal** messages from **previous + current superstep** (local **asynchronous** computation).
- ▶ This is called **hybrid execution model**.

Think Like a Matrix



Graphs and Matrices (1/2)

- ▶ A **graph** can be represented by an **adjacency matrix**.
- ▶ Operations on graphs can be performed by **algebraic operations** on matrices.
- ▶ **Linear algebra** and **matrix theory** can be applied to solve graph problems.

Graphs and Matrices (2/2)

- ▶ Given a graph $G = (V, E)$
- ▶ **Adjacency matrix** $A(G)$, a $|V| \times |V|$ matrix

$$A[i][j] = \begin{cases} 1 & \text{if } i \neq j \text{ and } (v_i, v_j) \in E \\ 0 & \text{if } i \neq j \text{ and } (v_i, v_j) \notin E \\ 0 & \text{if } i = j \end{cases}$$



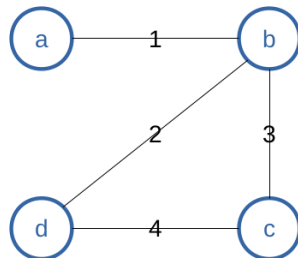
Adjacency Matrix Example

- ▶ Produce a vector representing the neighbors of a vertex v_i .

Adjacency Matrix Example

- ▶ Produce a vector representing the **neighbors of a vertex v_i** .
- ▶ By computing $\mathbf{A} \cdot \mathbf{x}_{v_i}$
 - $\mathbf{x}_{v_i}[i] = 1$ and all other elements of \mathbf{x}_{v_i} are 0.
- ▶ For example, to find the neighbors of vertex **b**

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = [1 \ 0 \ 1 \ 1]$$



Pegasus

Generalized Iterated Matrix-Vector (GIM-V)

- ▶ Targets at **iterative** graph algorithms.
- ▶ **Generalized Iterated Matrix-Vector** multiplication (**GIM-V**)
 - **Matrix-vector** multiplication
 - Assume **M** is a $n \times n$ matrix, **v** is a vector of size n , and $m_{i,j}$ denotes the (i, j) element of **M**.
 - $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$, where $v_i \leftarrow \sum_{j=1}^n m_{i,j} v_j$.

Generalized Iterated Matrix-Vector (GIM-V)

- ▶ Targets at **iterative** graph algorithms.
- ▶ **Generalized Iterated Matrix-Vector** multiplication (**GIM-V**)
 - **Matrix-vector** multiplication
 - Assume **M** is a $n \times n$ matrix, **v** is a vector of size n , and $m_{i,j}$ denotes the (i, j) element of **M**.
 - $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$, where $v_i \leftarrow \sum_{j=1}^n m_{i,j} v_j$.
- ▶ **Pegasus** models each **iteration of the graph computation** by a **GIM-V operation**
 - It is repeated until the vertex values in the vector converge.



GIM-V Operators

► $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$, where $v_i \leftarrow \sum_{j=1}^n m_{i,j} v_j$.



GIM-V Operators

- ▶ $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$, where $v_i \leftarrow \sum_{j=1}^n m_{i,j} v_j$.
- ▶ `combine2(i, j)`: to combine $m_{i,j}$ and v_j into a value.

- ▶ $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$, where $v_i \leftarrow \sum_{j=1}^n m_{i,j} v_j$.
- ▶ `combine2(i, j)`: to combine $m_{i,j}$ and v_j into a value.
- ▶ `combineAll(i)`: for each v_i , to combine all the n intermediate results produced by `combine2` into a single value.



GIM-V Operators

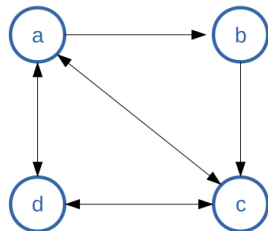
- ▶ $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$, where $v_i \leftarrow \sum_{j=1}^n m_{i,j} v_j$.
- ▶ `combine2(i, j)`: to combine $m_{i,j}$ and v_j into a value.
- ▶ `combineAll(i)`: for each v_i , to combine all the n intermediate results produced by `combine2` into a single value.
- ▶ `assign`: to overwrite the old value of v_i with the new value.



GIM-V Example: PageRank (2/3)

► PageRank formula: $\mathbf{v} \leftarrow (0.85\mathbf{E}^T + 0.15\mathbf{U}) \cdot \mathbf{v}$.

- \mathbf{v} is a **column vector** with n elements.
- \mathbf{E} is a **row-normalized adjacency matrix**.
- \mathbf{U} is a $n \times n$ matrix, with all elements set to $\frac{1}{n}$.



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad \mathbf{E} = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 0 & 0 & 0 & 1 \\ 1/2 & 0 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{bmatrix} \quad \mathbf{v}_{\text{init}} = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

► If $\mathbf{M} = 0.85\mathbf{E}^T + 0.15\mathbf{U}$, then we can write the PageRank as $\mathbf{v} \leftarrow \mathbf{M} \cdot \mathbf{v}$.

GIM-V Example: PageRank (3/3)

- ▶ PageRank formula: $\mathbf{v} \leftarrow (0.85\mathbf{E}^T + 0.15\mathbf{U}) \cdot \mathbf{v}$.
- ▶ $\text{combine2}(i, j) = 0.85 \times m_{i,j} \times v_j$
- ▶ $\text{combineAll}(i) = \frac{0.15}{n} + \sum_{j=1}^n \text{combine2}(i, j)$
- ▶ assign: $v_i \leftarrow \text{combineAll}(i)$

Summary



Summary

- ▶ Think like a table
 - Graphx: unifies data-parallel and graph-parallel systems.

- ▶ Think like a graph
 - Giraph++: exposes subgraphs to programmers

- ▶ Think like a matrix
 - Pegasus: linear algebra and matrix theory to solve graph problems.

- ▶ J. Gonzalez et al., “GraphX: Graph Processing in a Distributed Dataflow Framework”, OSDI 2014
- ▶ Y. Tian et al., “From think like a vertex to think like a graph”, VLDB 2013
- ▶ U. Kang et al., “PEGASUS: mining peta-scale graphs”, Knowledge and information systems 2011

Questions?