



# Large Scale Graph Processing - X-Stream and GraphX

Amir H. Payberah  
payberah@kth.se  
2020-09-29



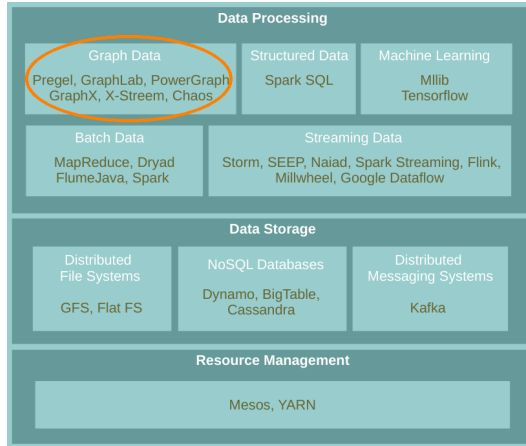


## The Course Web Page

`https://id2221kth.github.io`

`https://tinyurl.com/y4qph82u`

# 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**.



# Think Like an Edge



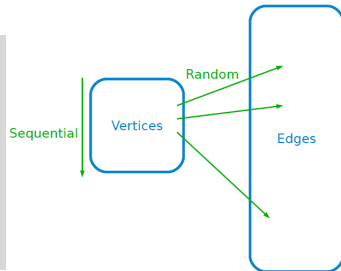
# Motivation

Could we compute **big graphs** on a **single machine**?



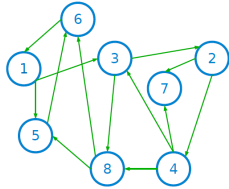
- ▶ **Vertex-centric** gather-scatter: iterates over **vertices**

```
Until convergence {  
  // the scatter phase  
  for all vertices v that need to scatter updates  
    send updates over outgoing edges of v  
  
  // the gather phase  
  for all vertices v that have updates  
    apply updates from inbound edges of v  
}
```





# Vertex-Centric Breadth First Search (1/5)



vertices

v
1
2
3
4
5
6
7
8

edges

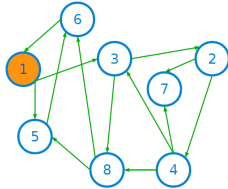
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Vertex-Centric Breadth First Search (2/5)



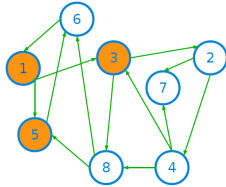
		edges	
		src	dest
vertices	v	1	3
	1	1	5
	2	2	7
	3	2	4
	4	3	2
	5	4	3
	6	4	7
	7	4	8
8	5	6	
		6	1
		8	5
		8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Vertex-Centric Breadth First Search (3/5)



vertices

v
1
2
3
4
5
6
7
8

edges

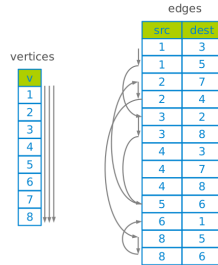
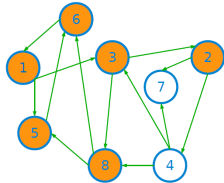
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Vertex-Centric Breadth First Search (4/5)

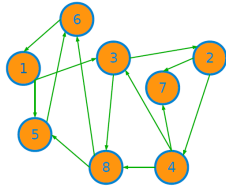


```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
  
```

# Vertex-Centric Breadth First Search (5/5)



vertices



edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# X-Stream



## X-Stream

- ▶ Could we process massive graphs on a single machine?



## X-Stream

- ▶ Could we process **massive graphs** on a **single machine**?
- ▶ **X-Stream** makes graph edges accesses **sequential**.





## X-Stream

- ▶ Could we process massive graphs on a single machine?
- ▶ X-Stream makes graph edges accesses sequential.
- ▶ Edge-centric scatter-gather model.

- ▶ Disk-based processing
  - Graph traversal = random access
  - Random access is inefficient for storage

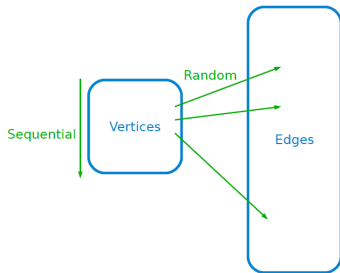
Medium	Read (MB/s)		Write (MB/s)	
	Random	Sequential	Random	Sequential
RAM	567	2605	1057	2248
SSD	22.64	355	49.16	298
Disk	0.61	174	1.27	170

Note: 64 byte cachelines, 4K blocks (disk random), 16M chunks (disk sequential)

Eiko Y., and Roy A., "Scale-up Graph Processing: A Storage-centric View", 2013.

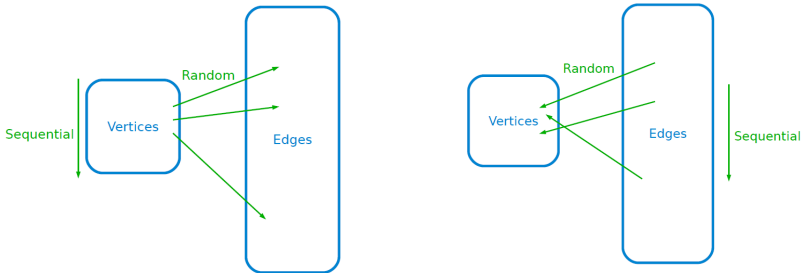
## Vertex-Centric vs. Edge-Centric Programming Model (1/2)

- ▶ **Vertex-centric** gather-scatter: **iterates** over **vertices**



## Vertex-Centric vs. Edge-Centric Programming Model (1/2)

- ▶ **Vertex-centric** gather-scatter: iterates over vertices
- ▶ **Edge-centric** gather-scatter: iterates over edges





## Vertex-Centric vs. Edge-Centric Programming Model (2/2)

```
Until convergence {  
  // the scatter phase  
  for all vertices v that need to scatter updates  
    send updates over outgoing edges of v  
  
  // the gather phase  
  for all vertices v that have updates  
    apply updates from inbound edges of v  
}
```

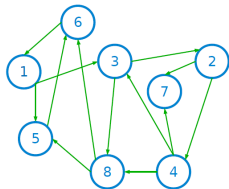


## Vertex-Centric vs. Edge-Centric Programming Model (2/2)

```
Until convergence {  
  // the scatter phase  
  for all vertices v that need to scatter updates  
    send updates over outgoing edges of v  
  
  // the gather phase  
  for all vertices v that have updates  
    apply updates from inbound edges of v  
}
```

```
Until convergence {  
  // the scatter phase  
  for all edges e  
    send update over e  
  
  // the gather phase  
  for all edges e that have updates  
    apply update to e.destination  
}
```

# Vertex-Centric Breadth First Search (1/5)



vertices

v
1
2
3
4
5
6
7
8

edges

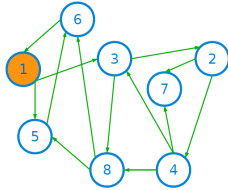
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Vertex-Centric Breadth First Search (2/5)



		edges	
		src	dest
vertices	v	1	3
	1	1	5
	2	2	7
	3	2	4
	4	3	2
	5	4	3
	6	4	7
	7	4	8
8	5	6	
		6	1
		8	5
		8	6

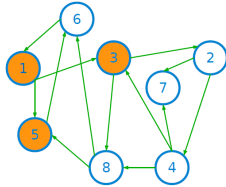
```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```



# Vertex-Centric Breadth First Search (3/5)



vertices

v
1
2
3
4
5
6
7
8

edges

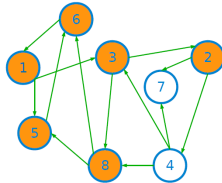
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Vertex-Centric Breadth First Search (4/5)



vertices

v
1
2
3
4
5
6
7
8

edges

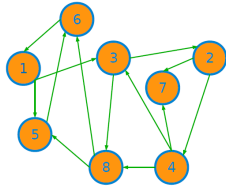
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Vertex-Centric Breadth First Search (5/5)



vertices

v
1
2
3
4
5
6
7
8

edges

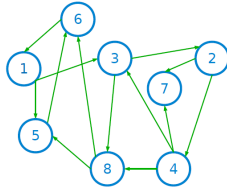
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all vertices v that need to scatter updates
    send updates over outgoing edges of v

  // the gather phase
  for all vertices v that have updates
    apply updates from inbound edges of v
}
    
```

# Edge-Centric Breadth First Search (1/5)



vertices

v
1
2
3
4
5
6
7
8

edges

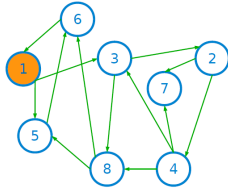
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
    
```

# Edge-Centric Breadth First Search (2/5)



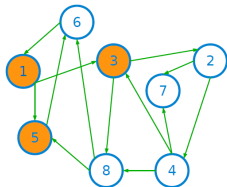
vertices		edges	
v		src	dest
1		1	3
2		1	5
3		2	7
4		2	4
5		3	2
6		3	8
7		4	3
8		4	7
		4	8
		5	6
		6	1
		8	5
		8	6

```

Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edges e that have updates
    apply update to e.destination
}
    
```

# Edge-Centric Breadth First Search (3/5)



vertices

v
1
2
3
4
5
6
7
8

edges

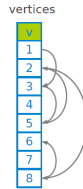
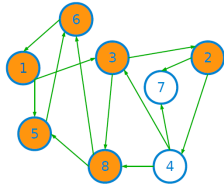
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edges e that have updates
    apply update to e.destination
}
    
```

# Edge-Centric Breadth First Search (4/5)



edges

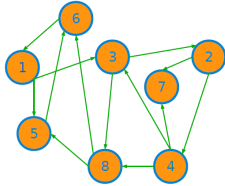
src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edges e that have updates
    apply update to e.destination
}
    
```

# Edge-Centric Breadth First Search (5/5)



edges

src	dest
1	3
1	5
2	7
2	4
3	2
3	8
4	3
4	7
4	8
5	6
6	1
8	5
8	6

```

Until convergence {
  // the scatter phase
  for all edges e
    send update over e

  // the gather phase
  for all edgaes e that have updates
    apply update to e.destination
}
    
```





## Vertex-Centric vs. Edge-Centric Tradeoff

- ▶ Vertex-centric scatter-gather:  $\frac{\text{EdgeData}}{\text{RandomAccessBandwidth}}$



## Vertex-Centric vs. Edge-Centric Tradeoff

- ▶ Vertex-centric scatter-gather:  $\frac{\text{EdgeData}}{\text{RandomAccessBandwidth}}$
- ▶ Edge-centric scatter-gather:  $\frac{\text{Scatters} \times \text{EdgeData}}{\text{SequentialAccessBandwidth}}$



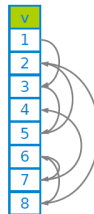
## Vertex-Centric vs. Edge-Centric Tradeoff

- ▶ Vertex-centric scatter-gather:  $\frac{\text{EdgeData}}{\text{RandomAccessBandwidth}}$
- ▶ Edge-centric scatter-gather:  $\frac{\text{Scatters} \times \text{EdgeData}}{\text{SequentialAccessBandwidth}}$
- ▶ Sequential Access Bandwidth  $\gg$  Random Access Bandwidth.
- ▶ Few scatter gather iterations for real world graphs.

# Streaming Partitions (1/4)

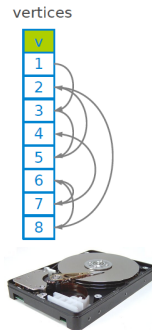
- ▶ **Problem:** still have **random** access to **vertex set**.

vertices



# Streaming Partitions (1/4)

- ▶ **Problem:** still have **random** access to **vertex set**.



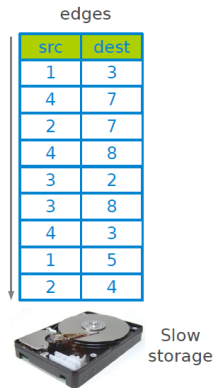
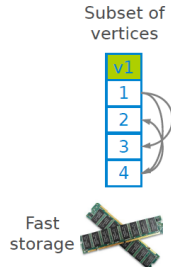
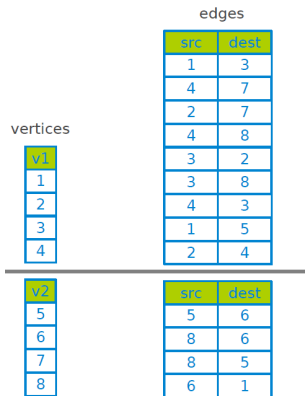
## Solution

Partition the graph into **streaming partitions**.

# Streaming Partitions (2/4)

vertices		edges	
v1		src	dest
1		1	3
2		4	7
3		2	7
4		4	8
		3	2
		3	8
		4	3
		1	5
		2	4
<hr/>			
v2		src	dest
5		5	6
6		8	6
7		8	5
8		6	1

# Streaming Partitions (2/4)





## Streaming Partitions (3/4)

- ▶ A **streaming partition** consists of: a **vertex set**, an **edge list**, and an **update list**.





## Streaming Partitions (3/4)

- ▶ A **streaming partition** consists of: a **vertex set**, an **edge list**, and an **update list**.
- ▶ The **vertex set**: a **subset of the vertex set** of the graph that fits into the **memory**.
  - Vertex sets are **mutually disjoint**.
  - Their **union** equals the vertex set of the **entire graph**.



## Streaming Partitions (3/4)

- ▶ A **streaming partition** consists of: a **vertex set**, an **edge list**, and an **update list**.
- ▶ The **vertex set**: a **subset of the vertex set** of the graph that fits into the **memory**.
  - Vertex sets are **mutually disjoint**.
  - Their **union** equals the vertex set of the **entire graph**.
- ▶ The **edge list**: all edges whose **source vertex** is in the **partition's vertex set**.



## Streaming Partitions (3/4)

- ▶ A **streaming partition** consists of: a **vertex set**, an **edge list**, and an **update list**.
- ▶ The **vertex set**: a **subset of the vertex set** of the graph that fits into the **memory**.
  - Vertex sets are **mutually disjoint**.
  - Their **union** equals the vertex set of the **entire graph**.
- ▶ The **edge list**: all edges whose **source vertex** is in the **partition's vertex set**.
- ▶ The **update list**: all updates whose **destination vertex** is in the **partition's vertex set**.



## Streaming Partitions (4/4)

```
// Scatter phase:  
for each streaming_partition p  
  read in vertex set of p  
  for each edge e in edge list of p  
    append update to Uout
```



## Streaming Partitions (4/4)

```
// Scatter phase:
for each streaming_partition p
  read in vertex set of p
  for each edge e in edge list of p
    append update to Uout

// shuffle phase:
for each update u in Uout
  p = partition containing target of u
  append u to Uin(p)
destroy Uout
```



## Streaming Partitions (4/4)

```
// Scatter phase:
```

```
for each streaming_partition p
  read in vertex set of p
  for each edge e in edge list of p
    append update to Uout
```

```
// shuffle phase:
```

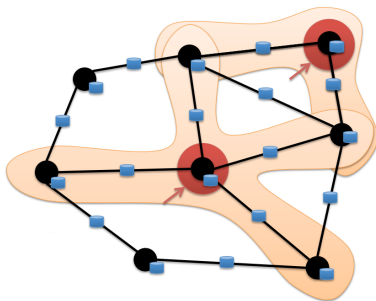
```
for each update u in Uout
  p = partition containing target of u
  append u to Uin(p)
destroy Uout
```

```
//gather phase:
```

```
for each streaming_partition p
  read in vertex set of p
  for each update u in Uin(p)
    edge_gather(u)
destroy Uin(p)
```

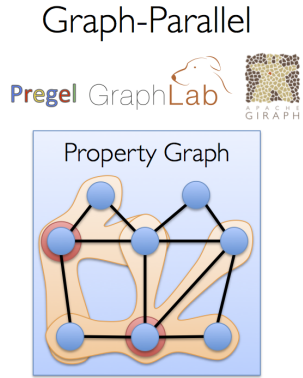
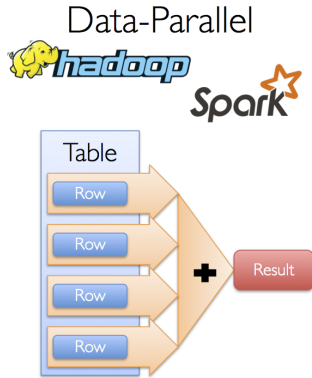
# Think Like a Table

# Graph-Parallel Processing Model





# Data-Parallel vs. Graph-Parallel Computation



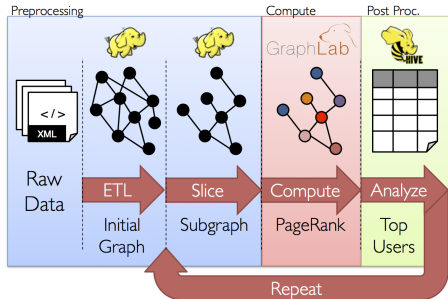


## Motivation (2/3)

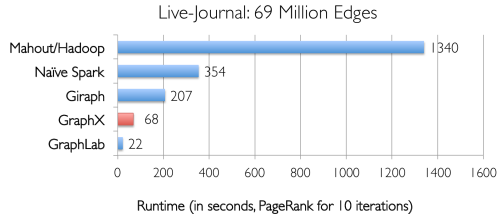
- ▶ **Graph-parallel** computation: **restricting** the types of computation to achieve **performance**.

## Motivation (2/3)

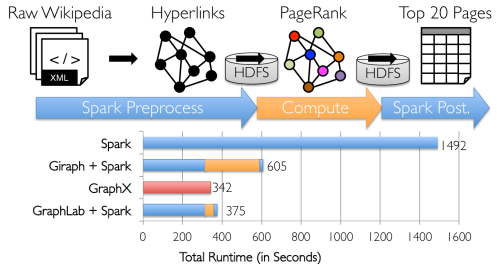
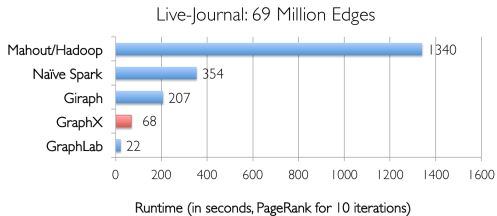
- ▶ **Graph-parallel** computation: **restricting** the types of computation to achieve **performance**.
- ▶ The same restrictions make it **difficult** and **inefficient** to express many stages in a typical graph-analytics **pipeline**.



## Motivation (3/3)

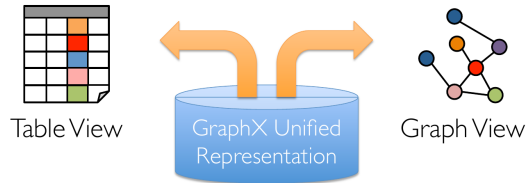


# Motivation (3/3)



# Think Like a Table

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

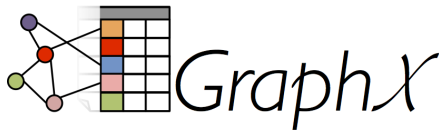


# GraphX



# GraphX

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



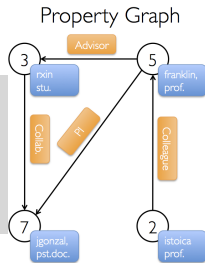


# The Property Graph Data Model

- ▶ 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

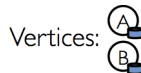
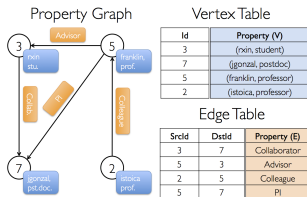
# The Vertex Collection

- ▶ **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)]
```

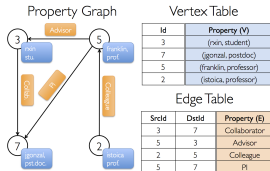


# The Edge Collection

- **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]]
```

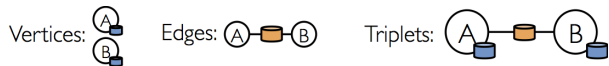


Edges: 



# The Triplet Collection

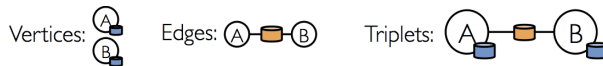
- ▶ The **triplets collection** consists of each **edge** and its **corresponding source and destination vertex** properties.





# The Triplet Collection

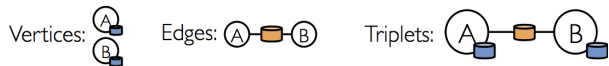
- ▶ The **triplets collection** consists of each **edge** and its **corresponding source and destination vertex** properties.
- ▶ It logically **joins the vertex and edge properties**: `RDD[EdgeTriplet[VD, ED]]`.



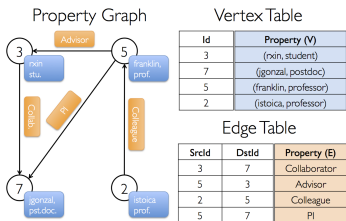


# The Triplet Collection

- ▶ 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.

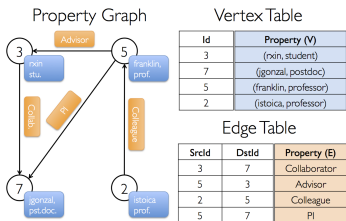


# Building a Property Graph



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

# Building a Property Graph

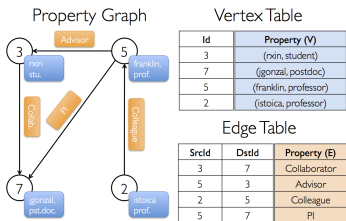


```
val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
    (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
```

```
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, "-")))
```



# Building a Property Graph

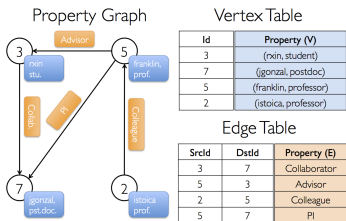


```
val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),
    (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
```

```
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")
```

# 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]
```



## Information About The Graph (1/2)

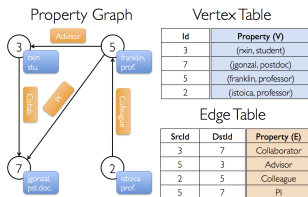
- ▶ Information about the graph

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

- ▶ Views of the graph as collections

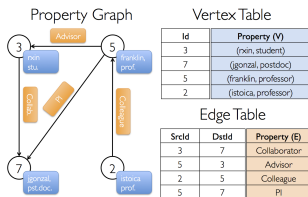
```
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]
```

## Information About The Graph (2/2)



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

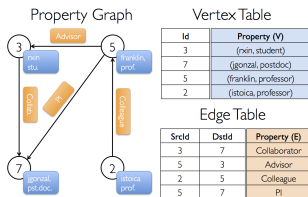
## Information About The Graph (2/2)



```
// 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
```

## 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.
- ▶ `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]
```



# 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]
```

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



## Join Operators

- ▶ `joinVertices` joins the `vertices` with the `input RDD`.

```
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]
```



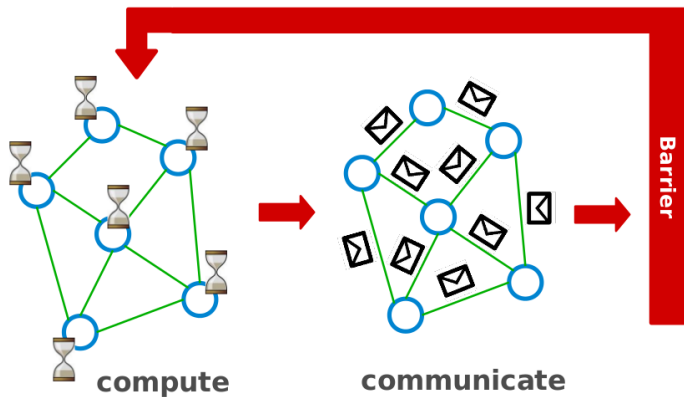


## Aggregation (2/2)

```
// 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)
)

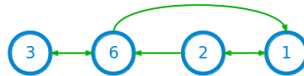
profs.collect.foreach(println)
```

# 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

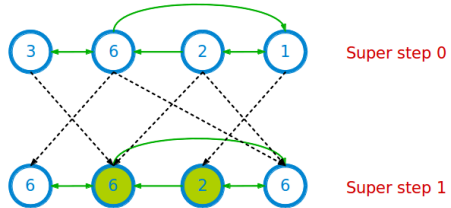
# Iterative Computation (3/9)

```

i_val := val

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

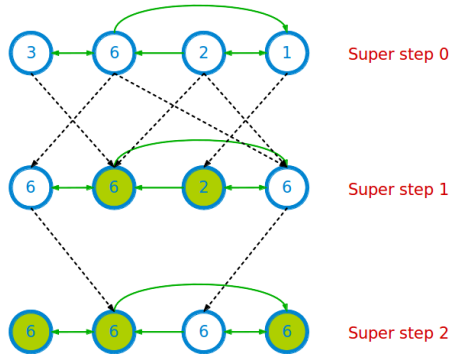
if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
  
```



# 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)
  
```



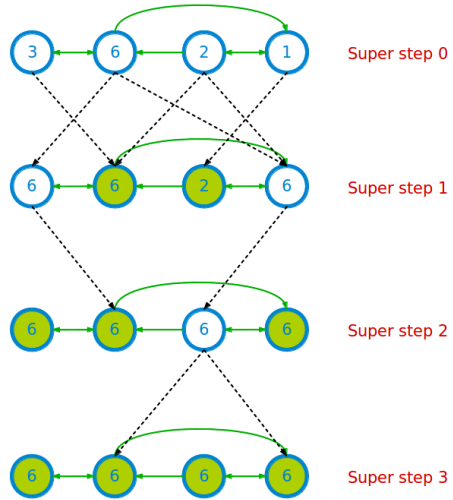
# Iterative Computation (5/9)

```

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.

```
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.
- ▶ The **second list** contains the **user defined functions**.
  - Gather: `mergeMsg`, Apply: `vprog`, Scatter: `sendMsg`

```
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)

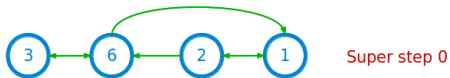


```
import org.apache.spark._  
import org.apache.spark.graphx._  
import org.apache.spark.rdd.RDD
```

```
val initialMsg = -9999
```



# Iterative Computation (7/9)



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

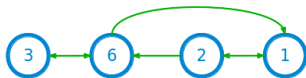
```
val initialMsg = -9999
```

```
// (vertexID, (new vertex value, old vertex value))
```

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



## Iterative Computation (7/9)



Super step 0

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

```
val initialMsg = -9999
```

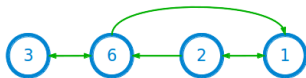
```
// (vertexID, (new vertex value, old vertex value))
```

```
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)))
```



## Iterative Computation (7/9)



Super step 0

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

```
val initialMsg = -9999
```

```
// (vertexID, (new vertex value, old vertex value))
```

```
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)

```
// Gather: the function for combining messages  
def mergeMsg(msg1: Int, msg2: Int): Int = math.max(msg1, msg2)
```



## Iterative Computation (8/9)

```
// Gather: the function for combining messages
```

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

```
// Apply: the function for receiving messages
```

```
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) = {
```

```
  if (message == initialMsg) // superstep 0
```

```
    value
```

```
  else // superstep > 0
```

```
    (math.max(message, value._1), value._1) // return (newValue, oldValue)
```

```
}
```



## Iterative Computation (8/9)

```
// Gather: the function for combining messages
```

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

```
// Apply: the function for receiving messages
```

```
def vprog(vertexId: VertexId, value: (Int, Int), message: Int): (Int, Int) = {  
  if (message == initialMsg) // superstep 0  
    value  
  else // superstep > 0  
    (math.max(message, value._1), value._1) // return (newValue, oldValue)  
}
```

```
// Scatter: the function for computing messages
```

```
def sendMsg(triplet: EdgeTriplet[(Int, Int), Boolean]): Iterator[(VertexId, Int)] = {  
  val sourceVertex = triplet.srcAttr  
  if (sourceVertex._1 == sourceVertex._2) // newValue == oldValue for source vertex?  
    Iterator.empty // do nothing  
  else  
    // propagate new (updated) value to the destination vertex  
    Iterator((triplet.dstId, sourceVertex._1))  
}
```





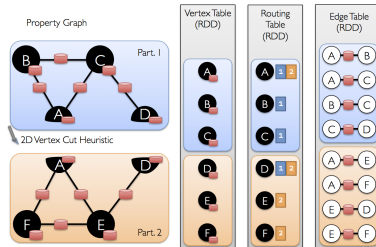
## Iterative Computation (9/9)

```
val minGraph = graph.pregel(initialMsg,
                             Int.MaxValue,
                             EdgeDirection.Out)(
    vprog, // apply
    sendMsg, // scatter
    mergeMsg) // gather

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

# 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.



# Summary



## Summary

- ▶ Think like an edge
  - XStream: edge-centric GAS, streaming partition
  
- ▶ Think like a table
  - Graphx: unifies data-parallel and graph-parallel systems.



## References

- ▶ A. Roy et al., “X-stream: Edge-centric graph processing using streaming partitions”, ACM SOSP 2013.
- ▶ J. Gonzalez et al., “GraphX: Graph Processing in a Distributed Dataflow Framework”, OSDI 2014

Questions?