



Large Scale Graph Processing - Pregel, GraphLab, and GraphX

Amir H. Payberah
payberah@kth.se
2022-09-29





The Course Web Page

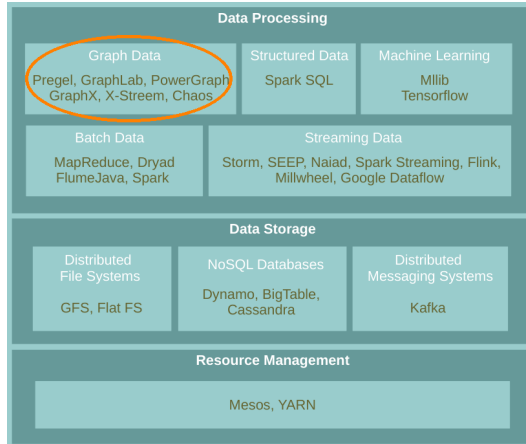
`https://id2221kth.github.io`



The Questions-Answers Page

<https://tinyurl.com/bdenpwc5>

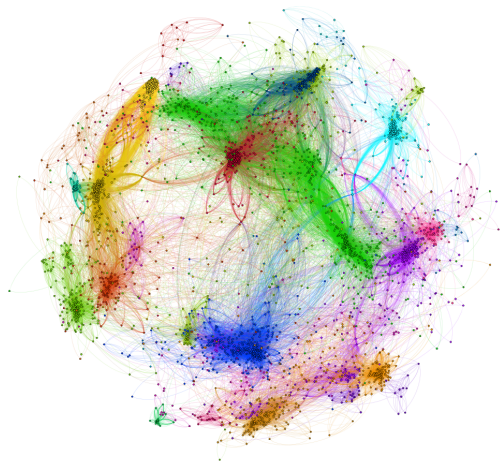
Where Are We?



- ▶ A flexible abstraction for describing relationships between discrete objects.



Large Graph



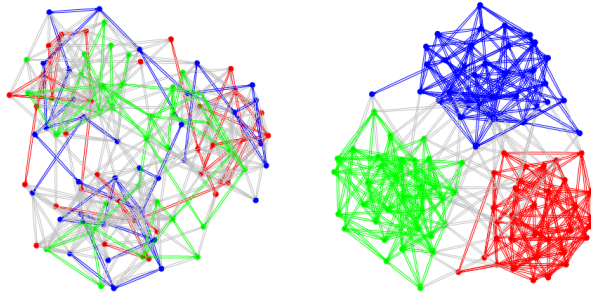


Graph Algorithms Challenges

- ▶ Difficult to extract **parallelism** based on **partitioning** of **the data**.
- ▶ Difficult to express **parallelism** based on **partitioning** of **computation**.
- ▶ **Graph partition** is a challenging problem.

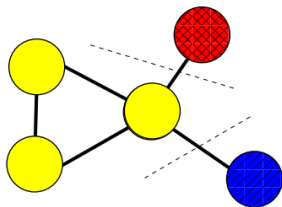
Graph Partitioning

- ▶ Partition large scale graphs and **distribut** to hosts.



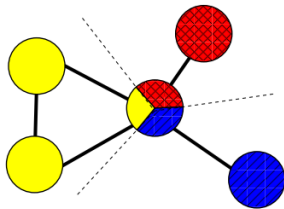
Edge-Cut Graph Partitioning

- ▶ Divide **vertices** of a graph into **disjoint clusters**.
- ▶ Nearly **equal size** (w.r.t. the number of **vertices**).
- ▶ With the **minimum number of edges** that **span separated clusters**.



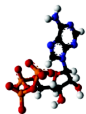
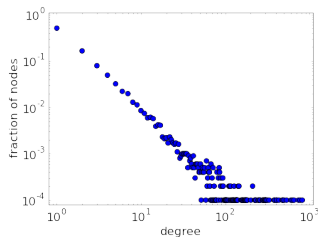
Vertex-Cut Graph Partitioning

- ▶ Divide **edges** of a graph into **disjoint clusters**.
- ▶ Nearly **equal size** (w.r.t. the number of **edges**).
- ▶ With the **minimum** number of **replicated vertices**.

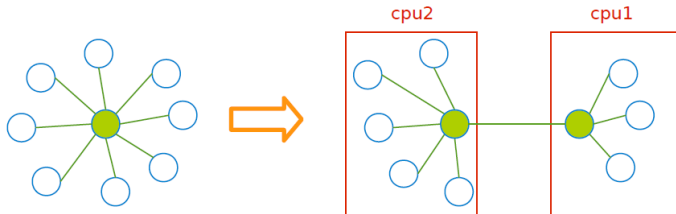
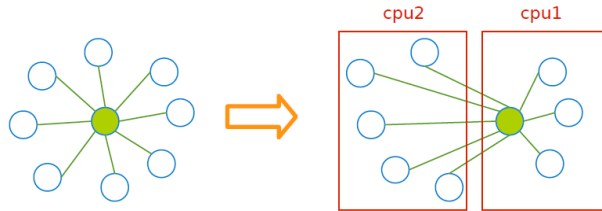


Edge-Cut vs. Vertex-Cut Graph Partitioning (1/2)

- ▶ Natural graphs: skewed **Power-Law** degree distribution.
- ▶ **Edge-cut** algorithms perform **poorly** on Power-Law Graphs.

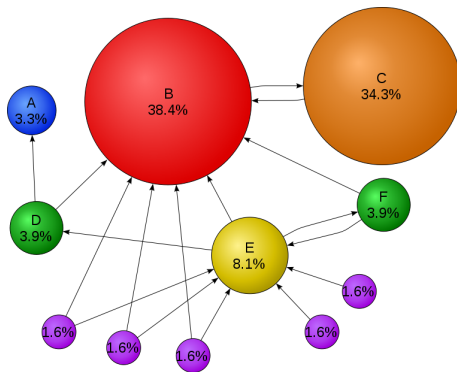


Edge-Cut vs. Vertex-Cut Graph Partitioning (2/2)





PageRank with MapReduce

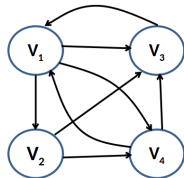


$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

PageRank Example (1/2)

▶
$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

▶ Input



V1: [0.25, V2, V3, V4]

V2: [0.25, V3, V4]

V3: [0.25, V1]

V4: [0.25, V1, V3]

▶ Share the rank among all outgoing links

V1: (V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3)

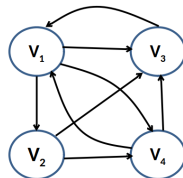
V2: (V3, 0.25/2), (V4, 0.25/2)

V3: (V1, 0.25/1)

V4: (V1, 0.25/2), (V3, 0.25/2)

PageRank Example (2/2)

$$\blacktriangleright R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$



V1: (V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3)

V2: (V3, 0.25/2), (V4, 0.25/2)

V3: (V1, 0.25/1)

V4: (V1, 0.25/2), (V3, 0.25/2)

► Output after one iteration

V1: [0.37, V2, V3, V4]

V2: [0.08, V3, V4]

V3: [0.33, V1]

V4: [0.20, V1, V3]

PageRank in MapReduce - Map (1/2)

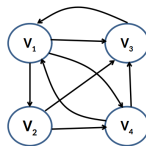
► Map function

```
map(key: [url, pagerank], value: outlink_list)
  for each outlink in outlink_list:
    emit(key: outlink, value: pagerank / size(outlink_list))

emit(key: url, value: outlink_list)
```

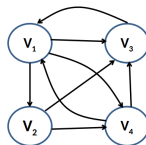
► Input (key, value)

```
((V1, 0.25), [V2, V3, V4])
((V2, 0.25), [V3, V4])
((V3, 0.25), [V1])
((V4, 0.25), [V1, V3])
```



PageRank in MapReduce - Map (2/2)

► Map function



```
map(key: [url, pagerank], value: outlink_list)
  for each outlink in outlink_list:
    emit(key: outlink, value: pagerank / size(outlink_list))

emit(key: url, value: outlink_list)
```

► Intermediate (key, value)

```
(V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3), (V3, 0.25/2), (V4, 0.25/2), (V1, 0.25/1),
(V1, 0.25/2), (V3, 0.25/2)
(V1, [V2, V3, V4])
(V2, [V3, V4])
(V3, [V1])
(V4, [V1, V3])
```



PageRank in MapReduce - Shuffle

► Intermediate (key, value)

```
(V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3), (V3, 0.25/2), (V4, 0.25/2), (V1, 0.25/1),  
(V1, 0.25/2), (V3, 0.25/2)  
(V1, [V2, V3, V4])  
(V2, [V3, V4])  
(V3, [V1])  
(V4, [V1, V3])
```

► After shuffling

```
(V1, 0.25/1), (V1, 0.25/2), (V1, [V2, V3, V4])  
(V2, 0.25/3), (V2, [V3, V4])  
(V3, 0.25/3), (V3, 0.25/2), (V3, 0.25/2), (V3, [V1])  
(V4, 0.25/3), (V4, 0.25/2), (V4, [V1, V3])
```



PageRank in MapReduce - Reduce (1/2)

▶ Reduce function

```
reducer(key: url, value: list_pr_or_urls)
  outlink_list = []
  pagerank = 0

  for each pr_or_urls in list_pr_or_urls:
    if is_list(pr_or_urls):
      outlink_list = pr_or_urls
    else
      pagerank += pr_or_urls

  emit(key: [url, pagerank], value: outlink_list)
```

▶ Input of the Reduce function

```
(V1, 0.25/1), (V1, 0.25/2), (V1, [V2, V3, V4])
(V2, 0.25/3), (V2, [V3, V4])
(V3, 0.25/3), (V3, 0.25/2), (V3, 0.25/2), (V3, [V1])
(V4, 0.25/3), (V4, 0.25/2), (V4, [V1, V3])
```



PageRank in MapReduce - Reduce (2/2)

▶ Reduce function

```
reducer(key: url, value: list_pr_or_urls)
  outlink_list = []
  pagerank = 0

  for each pr_or_urls in list_pr_or_urls:
    if is_list(pr_or_urls):
      outlink_list = pr_or_urls
    else
      pagerank += pr_or_urls

  emit(key: [url, pagerank], value: outlink_list)
```

▶ Output

```
((V1, 0.37), [V2, V3, V4])
((V2, 0.08), [V3, V4])
((V3, 0.33), [V1])
((V4, 0.20), [V1, V3])
```



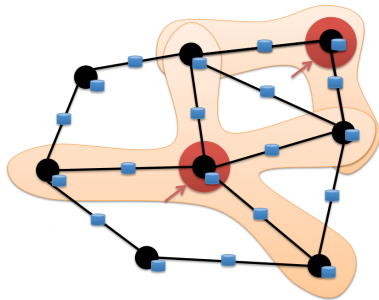
Problems with MapReduce for Graph Analytics

- ▶ MapReduce does **not directly support iterative** algorithms.
 - Invariant graph-topology-data **re-loaded** and **re-processed** at each iteration is **wasting** I/O, network bandwidth, and CPU
- ▶ **Materializations** of intermediate results at every MapReduce iteration **harm performance**.

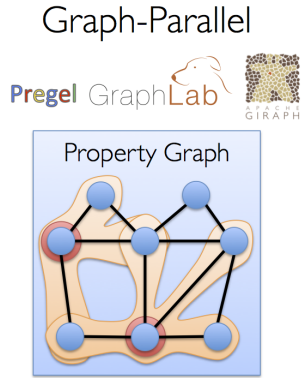
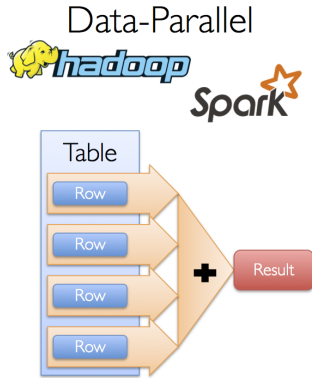
Think Like a Vertex

Think Like a Vertex

- ▶ Each vertex computes **individually** its value (in **parallel**).
- ▶ Computation typically depends on the **neighbors**.
- ▶ Also know as **graph-parallel** processing model.

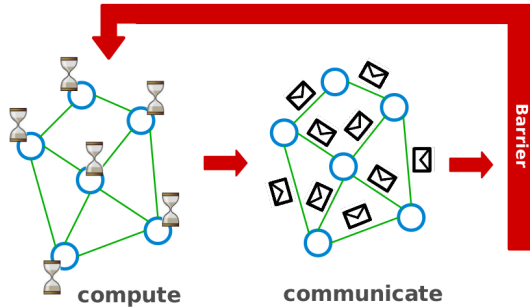


Data-Parallel vs. Graph-Parallel Computation



Pregel

- ▶ Large-scale **graph-parallel** processing platform developed at Google.
- ▶ Inspired by **bulk synchronous parallel (BSP)** model.



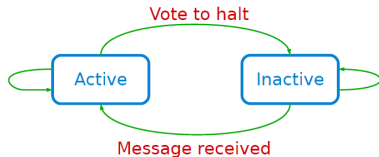


Execution Model (1/2)

- ▶ Applications run in sequence of **iterations**, called **supersteps**.
- ▶ A vertex in superstep **S** can:
 - **reads** messages sent to it in superstep **S-1**.
 - **sends** messages to other vertices: receiving at superstep **S+1**.
 - **modifies** its state.
- ▶ Vertices communicate directly with one another by **sending messages**.

Execution Model (2/2)

- ▶ Superstep 0: all vertices are in the active state.
- ▶ A vertex deactivates itself by voting to halt: no further work to do.
- ▶ A halted vertex can be active if it receives a message.
- ▶ The whole algorithm terminates when:
 - All vertices are simultaneously inactive.
 - There are no messages in transit.

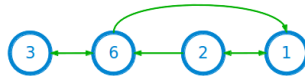


Example: Max Value (1/4)

```
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

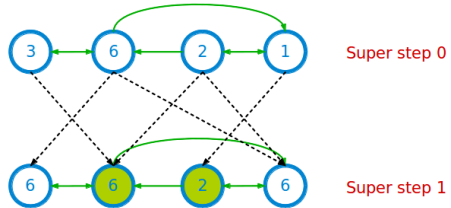
Example: Max Value (2/4)

```

i_val := val

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

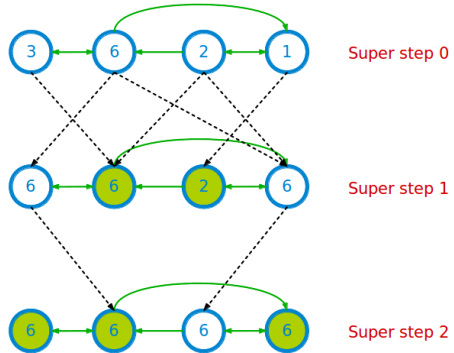
if i_val == val then
  vote_to_halt
else
  for each neighbor v
    send_message(v, val)
  
```



Example: Max Value (3/4)

```

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)
  
```



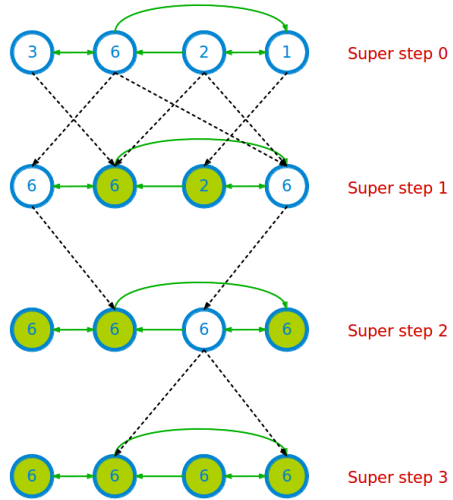
Example: Max Value (4/4)

```

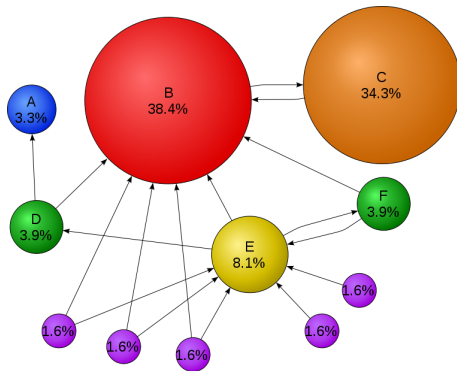
i_val := val

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)
  
```



Example: PageRank



$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$



Example: PageRank

```
Pregel_PageRank(i, messages):  
  // receive all the messages  
  total = 0  
  foreach(msg in messages):  
    total = total + msg  
  
  // update the rank of this vertex  
  R[i] = total  
  
  // send new messages to neighbors  
  foreach(j in out_neighbors[i]):  
    sendmsg(R[i] * wij) to vertex j
```

$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$



System Model

- ▶ **Master-worker** model.
- ▶ The **master**
 - **Coordinates** workers.
 - Assigns one or more **partitions** to each **worker**.
 - Instructs each worker to perform a **superstep**.
- ▶ Each **worker**
 - Executes the **local computation** method on its **vertices**.
 - Maintains the **state** of its **partitions**.
 - Manages **messages** to and from other workers.

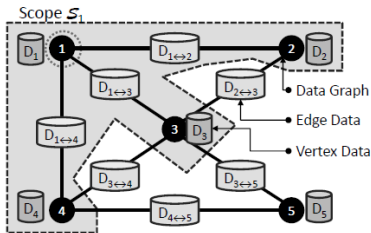


Pregel Limitations

- ▶ **Inefficient** if different regions of the graph converge at **different speed**.
- ▶ Runtime of each phase is determined by the **slowest** machine.

GraphLab/Turi

- ▶ GraphLab allows **asynchronous** iterative computation.
- ▶ **Vertex scope** of **vertex v** : the data stored in v , and in all **adjacent vertices and edges**.
- ▶ A vertex can **read** and **modify** any of the data in its **scope** (**shared memory**).





Example: PageRank (GraphLab)

```
GraphLab_PageRank(i)
  // compute sum over neighbors
  total = 0
  foreach(j in in_neighbors(i)):
    total = total + R[j] * wji

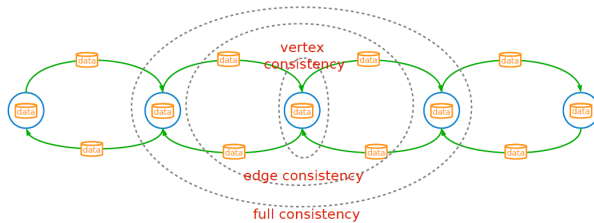
  // update the PageRank
  R[i] = total

  // trigger neighbors to run again
  foreach(j in out_neighbors(i)):
    signal vertex-program on j
```

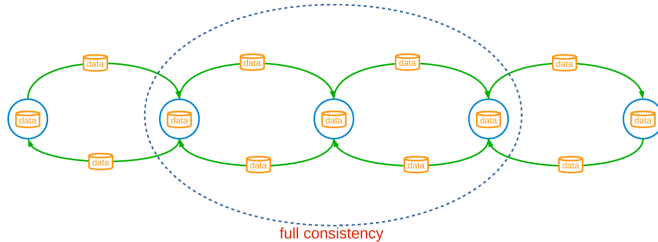
$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

Consistency (1/5)

- ▶ **Overlapped scopes:** **race-condition** in simultaneous execution of **two update functions**.

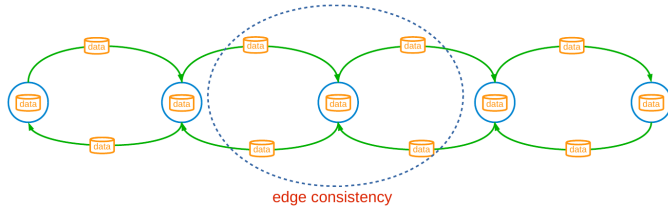


Consistency (2/5)



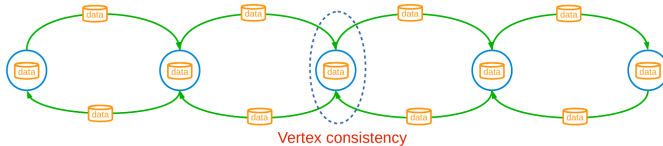
- ▶ **Full consistency:** during the execution $f(v)$, no other function reads or modifies data within the v scope.

Consistency (3/5)



- ▶ **Edge consistency:** during the execution $f(v)$, no other function reads or modifies any of the data on v or any of the edges adjacent to v .

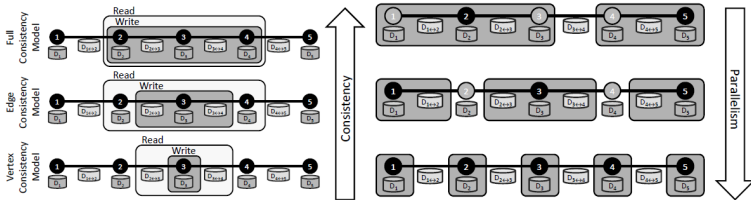
Consistency (4/5)



- ▶ **Vertex consistency**: during the execution $f(v)$, no other function will be applied to v .

Consistency (5/5)

Consistency vs. Parallelism



[Low, Y., GraphLab: A Distributed Abstraction for Large Scale Machine Learning (Doctoral dissertation, University of California), 2013.]



GraphLab2/Turi (PowerGraph)



PowerGraph

- ▶ Factorizes the local vertices functions into the Gather, Apply and Scatter phases.



Programming Model

- ▶ Gather-Apply-Scatter (GAS)
- ▶ **Gather**: **accumulate** information from neighborhood.
- ▶ **Apply**: **apply** the accumulated value to center vertex.
- ▶ **Scatter**: **update** adjacent edges and vertices.



Execution Model (1/2)

- ▶ Initially **all vertices** are **active**.
- ▶ It executes the **vertex-program** on the **active vertices** until none remain.
 - Once a vertex-program completes the **scatter** phase it becomes **inactive** until it is reactivated.
 - Vertices can activate **themselves** and **neighboring** vertices.
- ▶ PowerGraph can execute both **synchronously** and **asynchronously**.



Execution Model (2/2)

- ▶ **Synchronous** scheduling like **Pregel**.
 - Executing the **gather**, **apply**, and **scatter in order**.
 - Changes made to the vertex/edge data are committed at the **end** of each step.

- ▶ **Asynchronous** scheduling like **GraphLab**.
 - Changes made to the vertex/edge data during the **apply and scatter** functions are **immediately** committed to the graph.
 - **Visible** to subsequent computation on neighboring vertices.



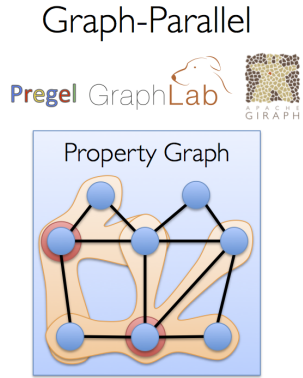
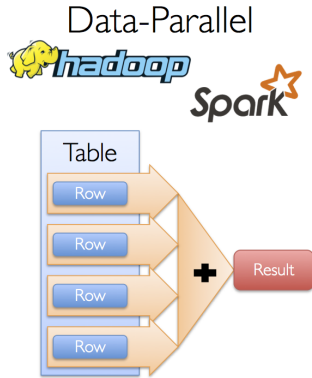
Example: PageRank (PowerGraph)

```
PowerGraph_PageRank(i):  
  Gather(j -> i):  
    return wji * R[j]  
  
  sum(a, b):  
    return a + b  
  
  // total: Gather and sum  
  Apply(i, total):  
    R[i] = total  
  
  Scatter(i -> j):  
    if R[i] changed then activate(j)
```

$$R[i] = \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

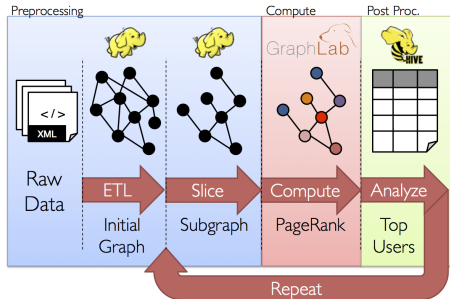
Think Like a Table

Data-Parallel vs. Graph-Parallel Computation

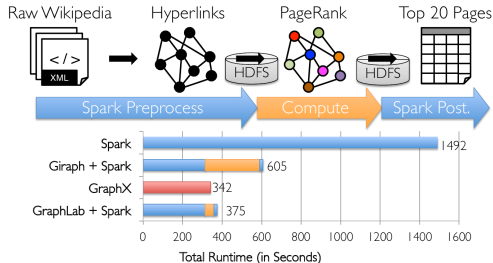
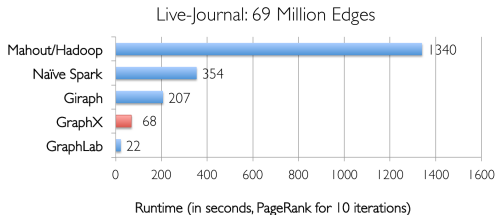


Motivation (2/3)

- ▶ **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**.

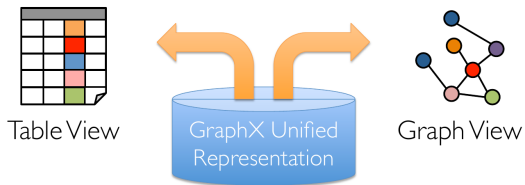


Motivation (3/3)



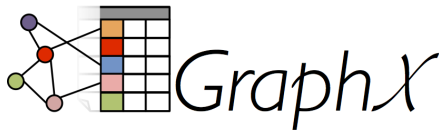
Think Like a Table

- ▶ 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**.

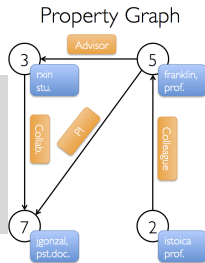


The Property Graph Data Model

- ▶ 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

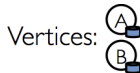
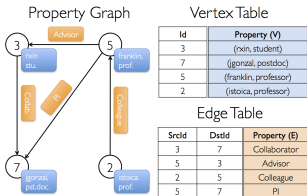
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

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)]
```

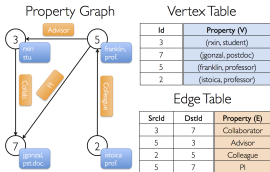


The Edge Collection

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

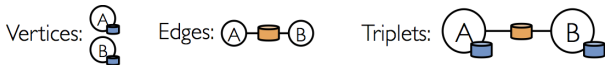
```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}

// 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]]
```

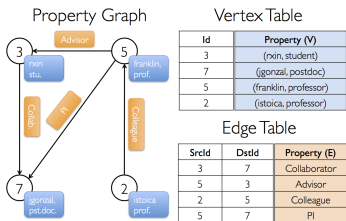


The Triplet Collection

- ▶ 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)

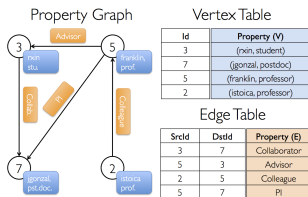
- ▶ Information about the graph

```
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]
```

```
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]
```

```
// 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`.
 - 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]
```

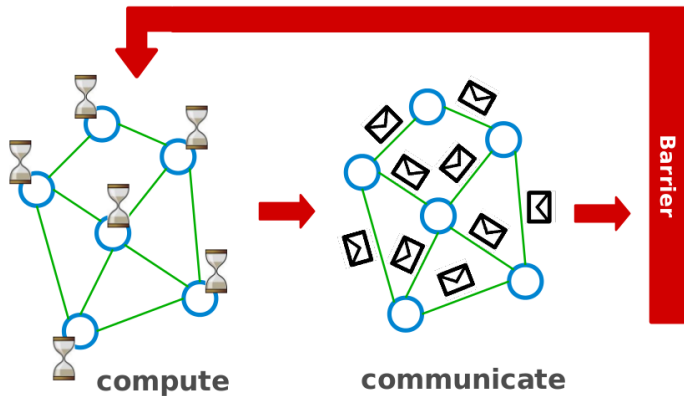


Aggregation (2/2)

```
// count and list the name of friends of each user
val profs: VertexRDD[(Int, String)] = validGraph.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 (1/6)



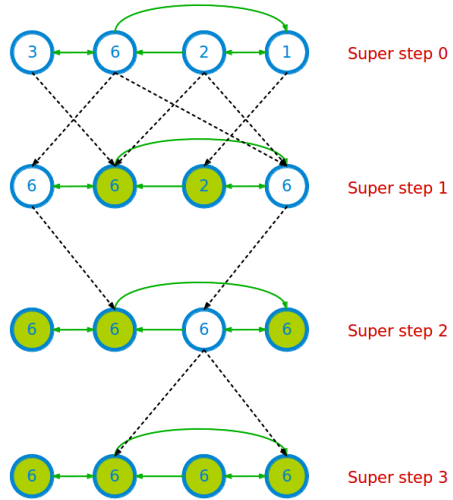
Iterative Computation (2/6)

```

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)
  
```





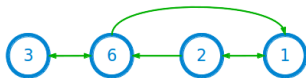
Iterative Computation (3/6)

- ▶ `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 (4/6)



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 (5/6)

```
// 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 (6/6)

```
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)
}
```

Summary



Summary

- ▶ Think like a vertex
 - Pregel: BSP, synchronous parallel model, message passing
 - GraphLab: asynchronous model, shared memory
 - PowerGraph: synchronous/asynchronous model, GAS

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



References

- ▶ G. Malewicz et al., “Pregel: a system for large-scale graph processing”, ACM SIGMOD 2010
- ▶ Y. Low et al., “Distributed GraphLab: a framework for machine learning and data mining in the cloud”, VLDB 2012
- ▶ J. Gonzalez et al., “Powergraph: distributed graph-parallel computation on natural graphs”, OSDI 2012
- ▶ J. Gonzalez et al., “GraphX: Graph Processing in a Distributed Dataflow Framework”, OSDI 2014

Questions?