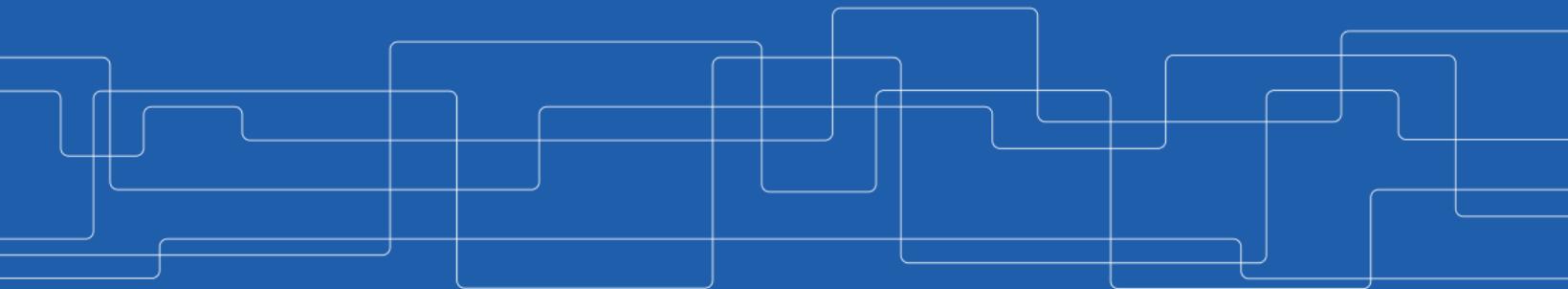




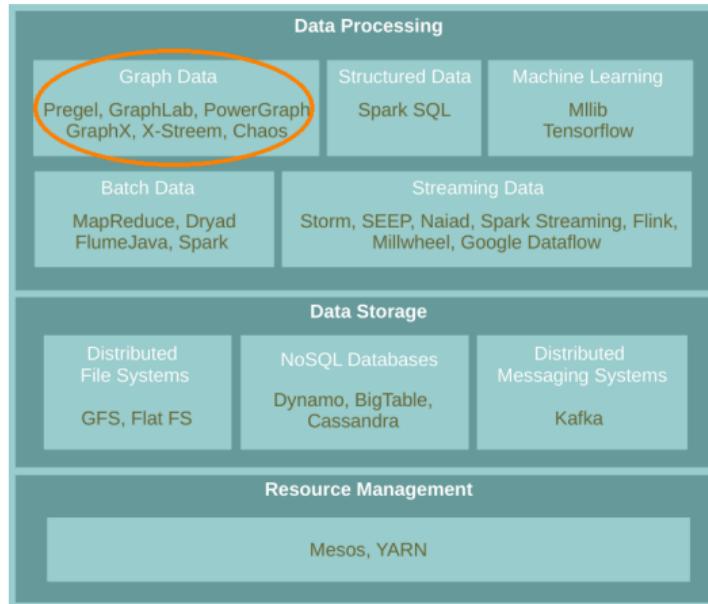
# Large Scale Graph Processing

Amir H. Payberah  
[payberah@kth.se](mailto:payberah@kth.se)  
2025-09-22





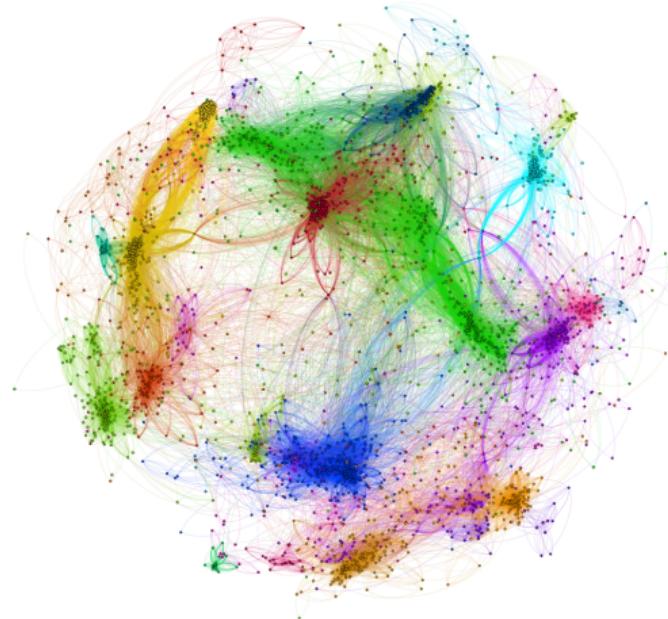
# Where Are We?



- ▶ 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.

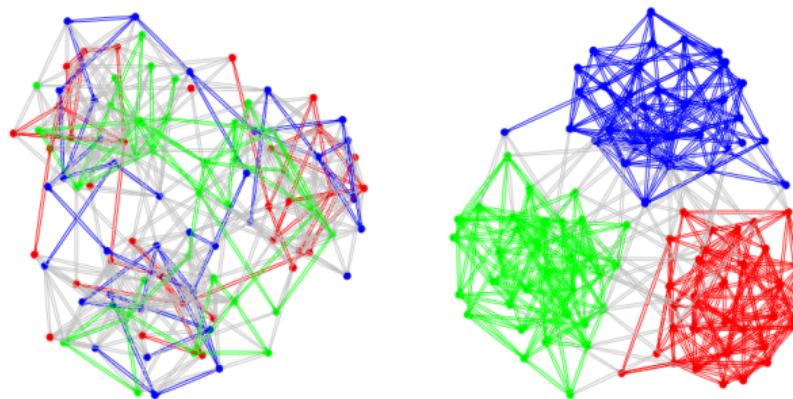


# Graph Algorithms Challenges

- ▶ 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.

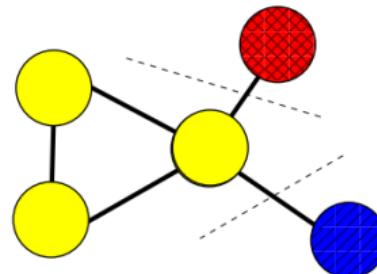
# Graph Partitioning

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



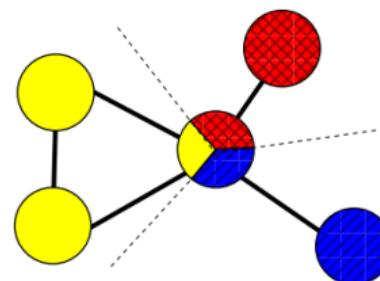
# Edge-Cut Graph Partitioning

- ▶ 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.



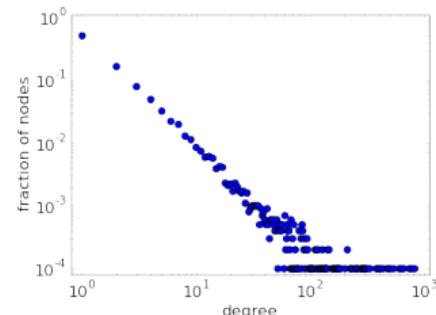
# Vertex-Cut Graph Partitioning

- ▶ 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.

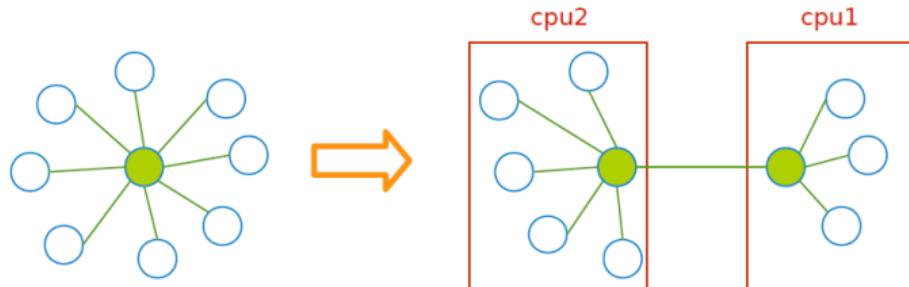
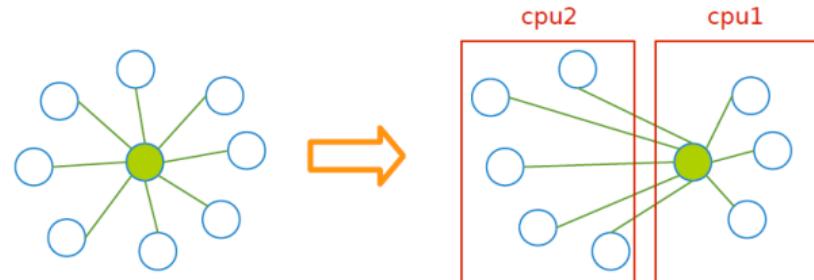


# Edge-Cut vs. Vertex-Cut Graph Partitioning (1/2)

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



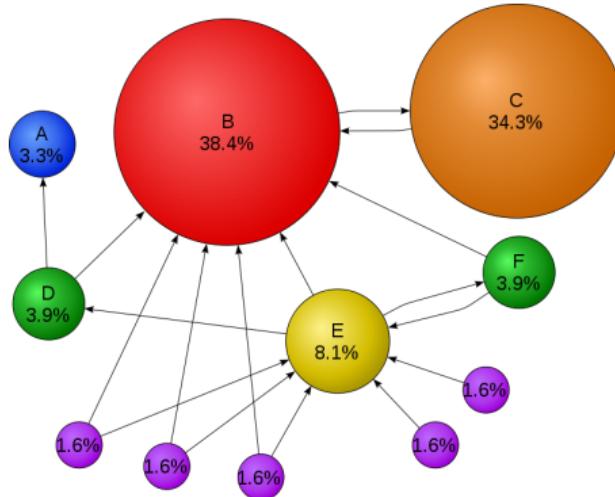
## Edge-Cut vs. Vertex-Cut Graph Partitioning (2/2)





# PageRank with MapReduce

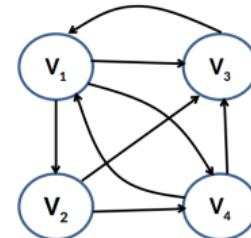
# PageRank



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

## PageRank Example (1/2)

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



## PageRank Example (1/2)

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

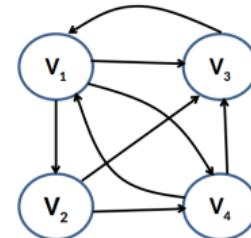
► Input

V1: [0.25, V2, V3, V4]

V2: [0.25, V3, V4]

V3: [0.25, V1]

V4: [0.25, V1, V3]



## PageRank Example (1/2)

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

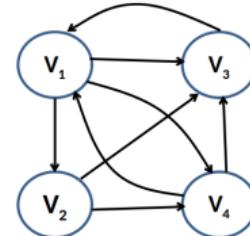
► Input

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)

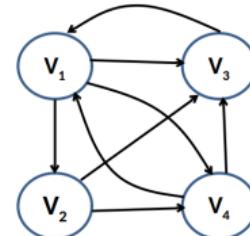
V2: (V3, 0.25/2), (V4, 0.25/2)

V3: (V1, 0.25/1)

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

## PageRank Example (2/2)

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



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

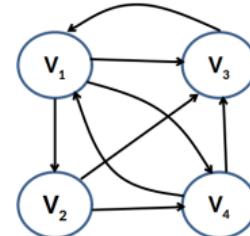
V2: (V3, 0.25/2), (V4, 0.25/2)

V3: (V1, 0.25/1)

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

## PageRank Example (2/2)

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



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]

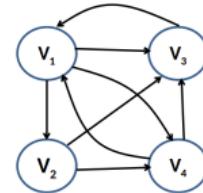
V3: [0.33, V1]

V4: [0.20, V1, V3]



# PageRank in MapReduce - Map (1/2)

- ▶ Map function

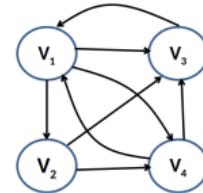


```
map(key: [url, pagerank], value: outlink_list)
    for each outlink in outlink_list:
        emit(key: outlink, value: pagerank / size(outlink_list))

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

# 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))

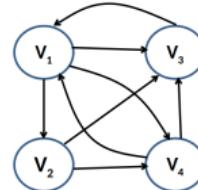
    emit(key: url, value: outlink_list)
```

- ▶ 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

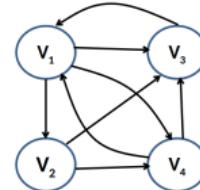


```
map(key: [url, pagerank], value: outlink_list)
    for each outlink in outlink_list:
        emit(key: outlink, value: pagerank / size(outlink_list))

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

## 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])
```

# 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)
```



# 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)
```



## 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



# Problems with MapReduce for Graph Analytics

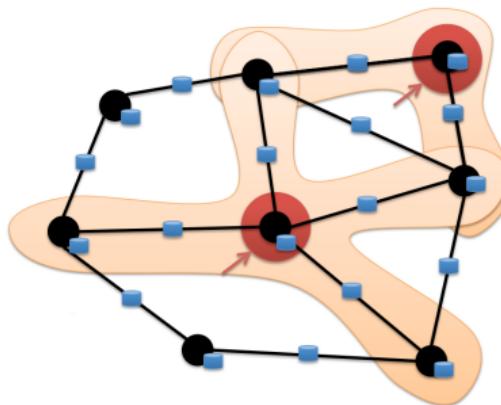
- ▶ MapReduce does **not directly support iterative** algorithms.
  - Invariant graph-topology-data **re-loaded** and **re-processed** at each iteration is **wasting** I/O, network bandwidth, and CPU
- ▶ Materializations of intermediate results at every MapReduce iteration **harm performance**.



# Think Like a Vertex

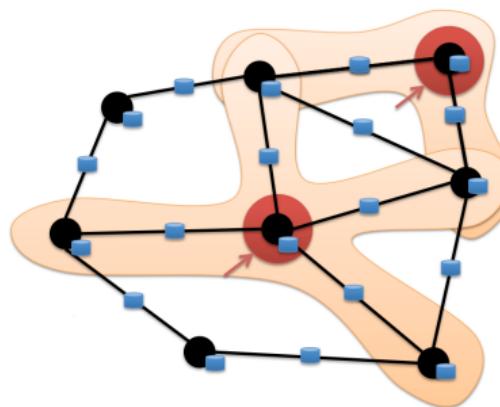
# Think Like a Vertex

- ▶ Each vertex computes **individually** its value (in **parallel**).
- ▶ Computation typically depends on the **neighbors**.

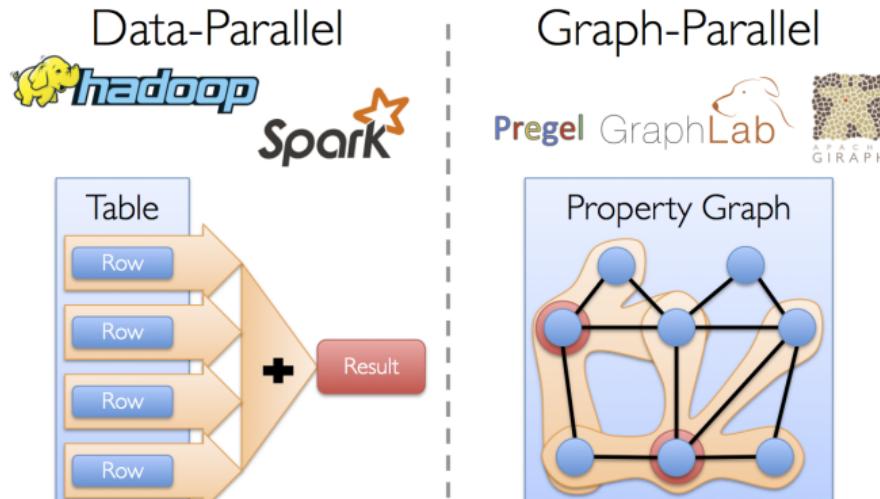


# Think Like a Vertex

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



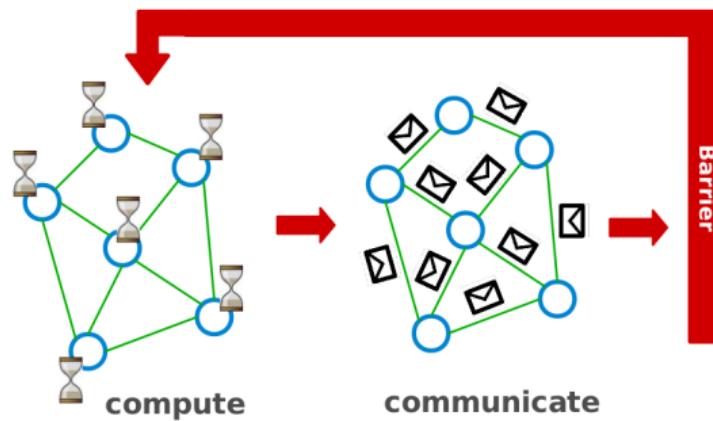
# Data-Parallel vs. Graph-Parallel Computation





# 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**.



## 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.

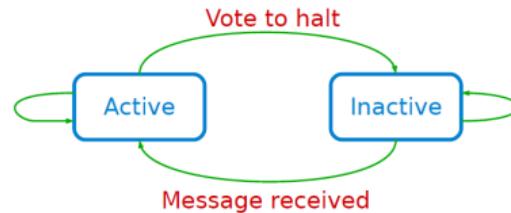


## Execution Model (1/2)

- ▶ 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**.

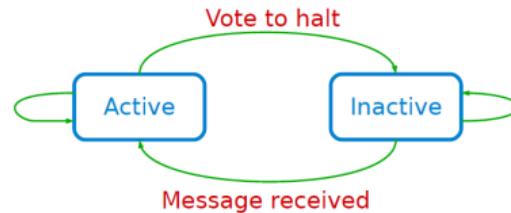
## Execution Model (2/2)

- ▶ Superstep 0: all vertices are in the **active** state.



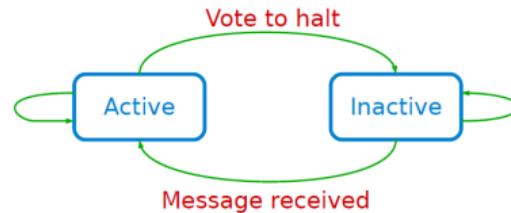
## Execution Model (2/2)

- ▶ Superstep 0: all vertices are in the **active** state.
- ▶ A vertex **deactivates** itself by voting to **halt**: no further work to do.



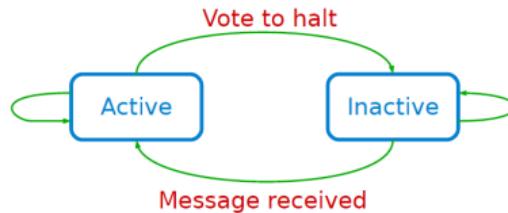
## Execution Model (2/2)

- ▶ 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**.



## 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

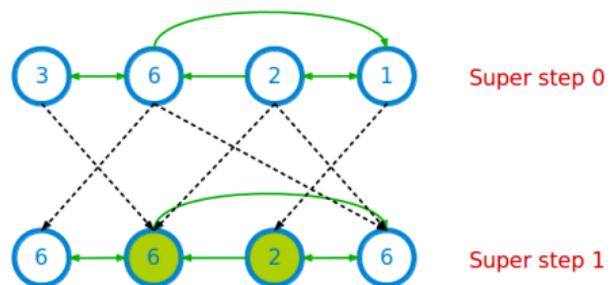
if i_val == val then
    vote_to_halt
else
    for each neighbor v
        send_message(v, val)
```



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)
```



Super step 0

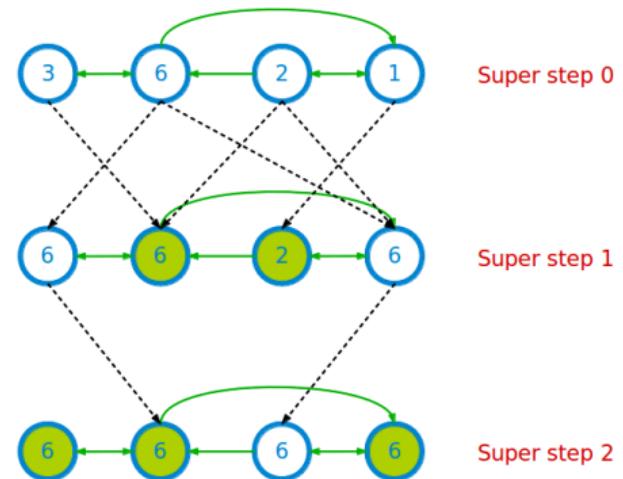
Super step 1

## Example: Max Value (3/4)

```
i_val := val

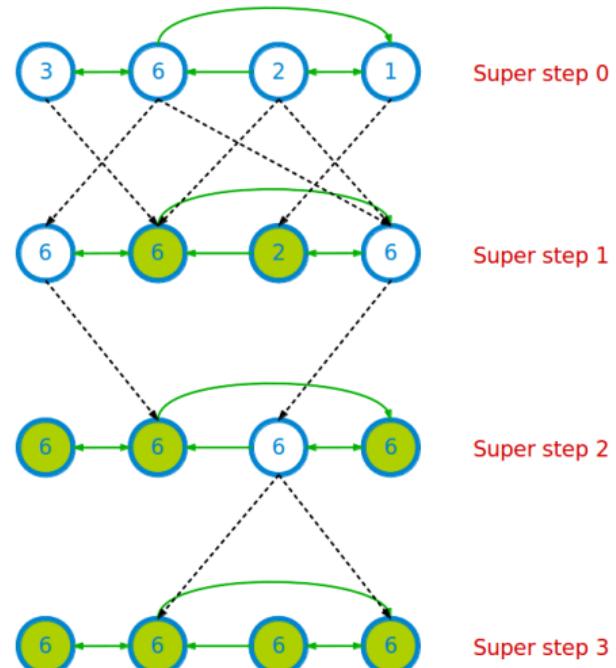
for each message m
    if m > val then val := m

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

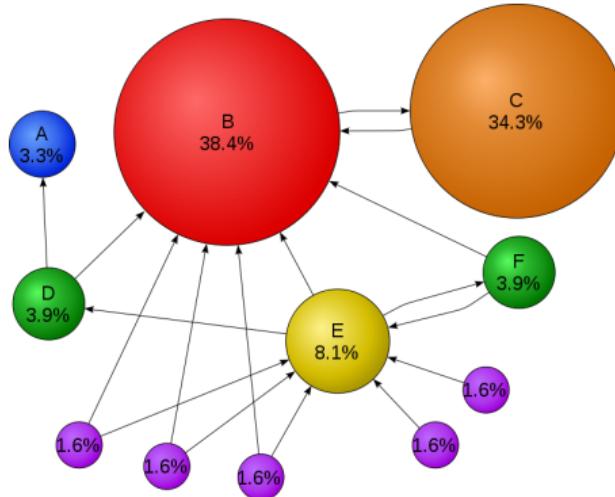


## Example: Max Value (4/4)

```
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
```

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



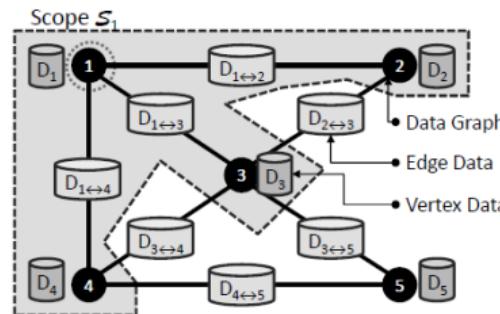
# GraphLab/Turi



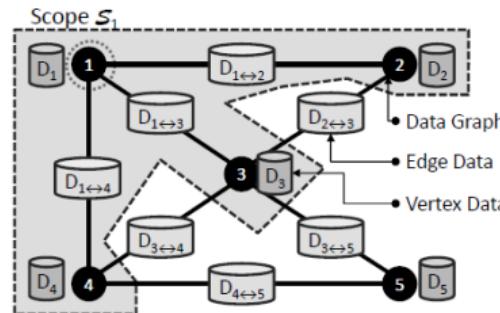
# GraphLab

- ▶ GraphLab allows **asynchronous** iterative computation.

- ▶ GraphLab allows **asynchronous** iterative computation.
- ▶ **Vertex scope** of vertex  $v$ : the data stored in  $v$ , and in all **adjacent vertices and edges**.



- ▶ 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
```

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



## Gather-Apply-Scatter (GAS)

- ▶ Factorizes the local vertices functions into the **Gather**, **Apply** and **Scatter** phases.
- ▶ **Gather**: accumulate information from neighborhood.
- ▶ **Apply**: apply the accumulated value to center vertex.
- ▶ **Scatter**: update adjacent edges and vertices.



## Example: PageRank (GraphLab - GAS)

```
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

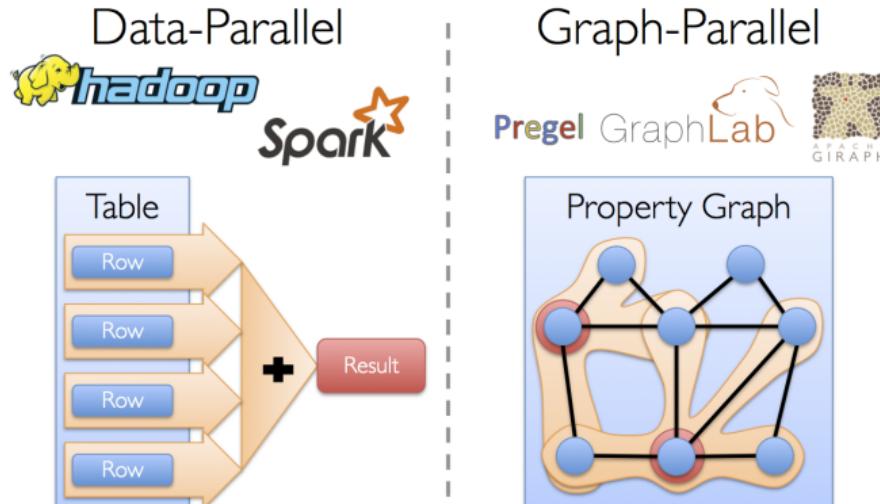
    Scatter(i -> j):
        if R[i] changed then activate(j)
```

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



# Think Like a Table

# Data-Parallel vs. Graph-Parallel Computation



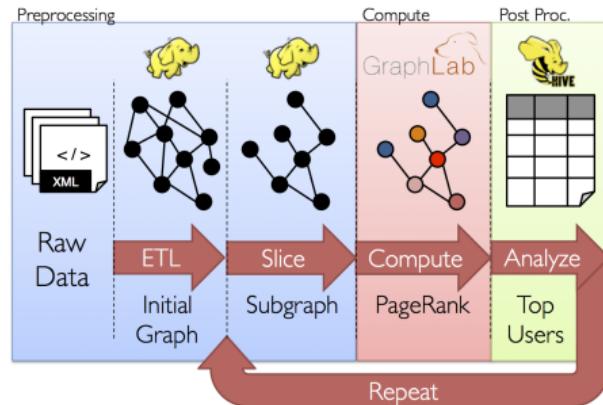


## Motivation (2/3)

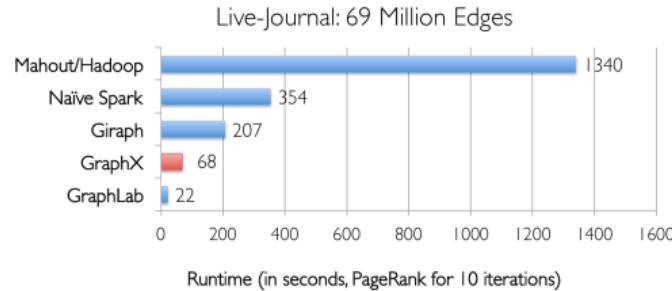
- ▶ Graph-parallel computation: **restricting** the types of computation to achieve **performance**.

## Motivation (2/3)

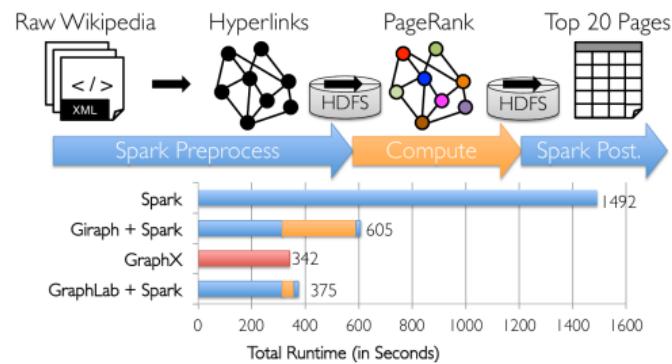
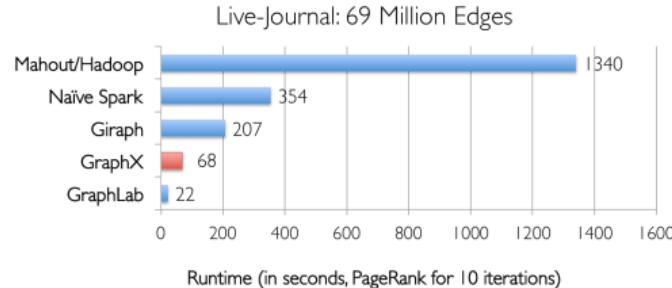
- ▶ Graph-parallel computation: **restricting** the types of computation to achieve **performance**.
- ▶ The same restrictions make it **difficult** and **inefficient** to express many stages in a typical graph-analytics **pipeline**.



## Motivation (3/3)

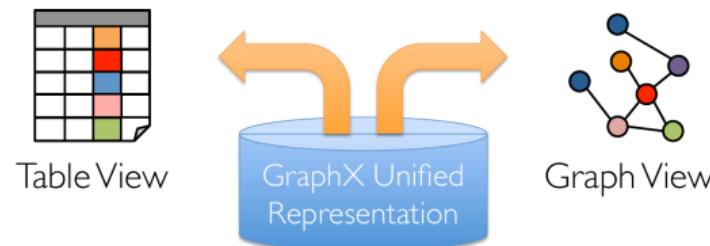


## Motivation (3/3)



# Think Like a Table

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



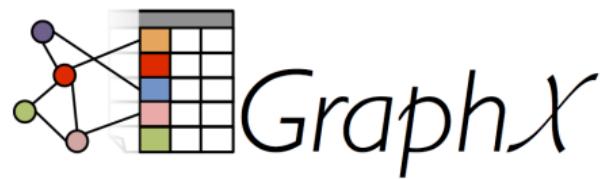


# GraphX



# GraphX

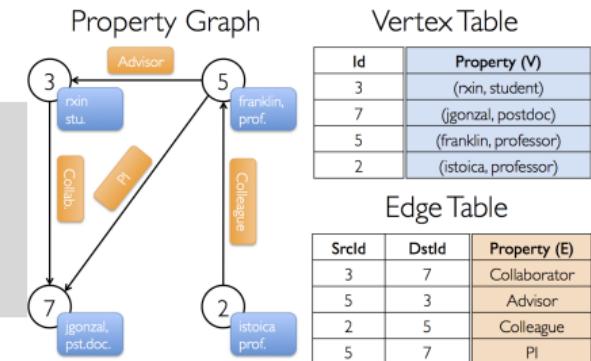
- ▶ GraphX is the library to perform graph-parallel processing in Spark.



# 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**

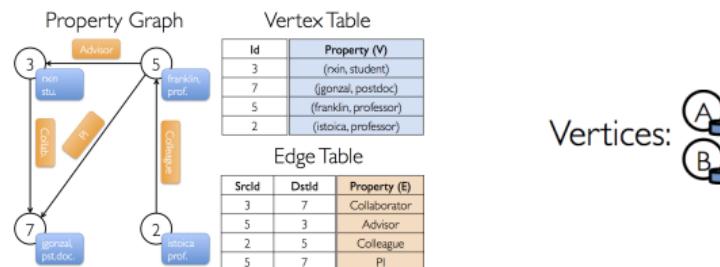
```
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
    val vertices: VertexRDD[VD]
    val edges: EdgeRDD[ED]
}
```



# The Vertex Collection

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

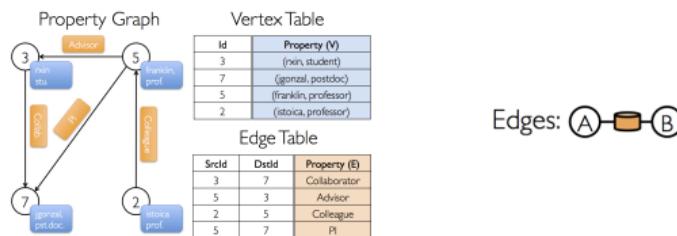
```
class Graph[VD, ED] {  
    val vertices: VertexRDD[VD]  
    val edges: EdgeRDD[ED]  
}  
  
// VD: the type of the vertex attribute  
abstract class VertexRDD[VD] extends RDD[(VertexId, VD)]
```



# 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]]
```



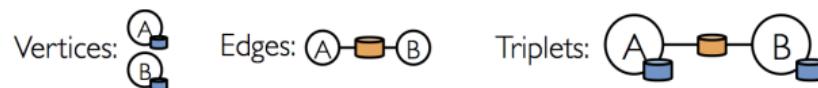
# The Triplet Collection

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



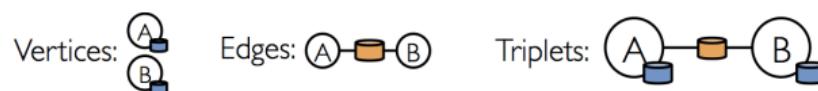
# The Triplet Collection

- ▶ 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]]`.

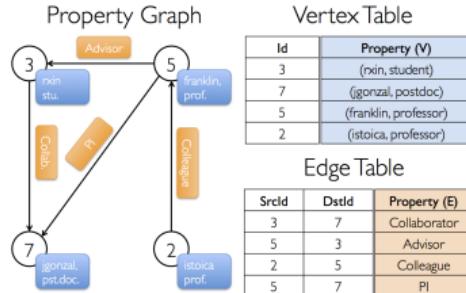


# The Triplet Collection

- ▶ 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.

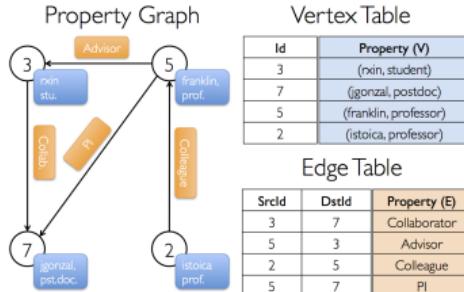


# Building a Property Graph



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

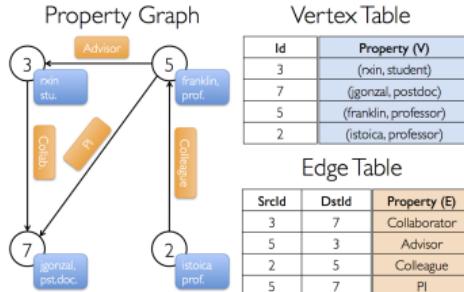
# Building a Property Graph



```
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

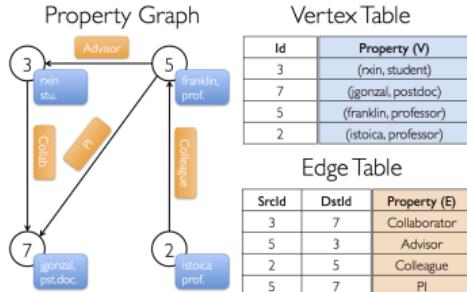


```
val users: RDD[(VertexId, (String, String))] = sc.parallelize(Array((3L, ("rxin", "student")),  
    (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
```

```
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
- ▶ ...





## Questions

- ▶ Chris Anderson's "The End of Theory" claims that with big data, numbers can "[speak for themselves](#)", removing the need for theory or context. [PageRank](#) ranks nodes by link counts in this spirit. What gets [lost](#) when importance is defined only by data, without social or cultural context?



## Possible Answers

- ▶ PageRank treats **popularity** (links) as importance: ignores **meaning**, **quality**, or **intent** behind connections.



## Possible Answers

- ▶ PageRank treats **popularity** (links) as importance: ignores **meaning**, **quality**, or **intent** behind connections.
- ▶ Marginalized or less-connected voices risk **invisibility**, even if socially significant.



## Possible Answers

- ▶ PageRank treats **popularity** (links) as importance: ignores **meaning**, **quality**, or **intent** behind connections.
- ▶ Marginalized or less-connected voices risk **invisibility**, even if socially significant.
- ▶ **Context** (history, culture, power) is erased in favor of raw link counts.



## Possible Answers

- ▶ PageRank treats **popularity** (links) as importance: ignores **meaning**, **quality**, or **intent** behind connections.
- ▶ Marginalized or less-connected voices risk **invisibility**, even if socially significant.
- ▶ **Context** (history, culture, power) is erased in favor of raw link counts.
- ▶ **Danger:** reinforces existing hierarchies under the guise of neutrality.



## Questions

- ▶ When we **partition** graphs for scalability, what social ties might be **lost**?



## Possible Answers

- ▶ Algorithms typically optimize for **balance** and **cut-size**, not **social meaning**.



## Possible Answers

- ▶ Algorithms typically optimize for **balance** and **cut-size**, not **social meaning**.
- ▶ Peripheral or minority groups may be **split across partitions**.



## Possible Answers

- ▶ Algorithms typically optimize for **balance** and **cut-size**, not **social meaning**.
- ▶ Peripheral or minority groups may be **split across partitions**.
- ▶ Cross-community ties (weak ties) can be broken: **erasing nuance**.



## Possible Answers

- ▶ Algorithms typically optimize for **balance** and **cut-size**, not **social meaning**.
- ▶ Peripheral or minority groups may be **split across partitions**.
- ▶ **Cross-community ties** (weak ties) can be broken: **erasing nuance**.
- ▶ **Result:** efficiency prioritized over representation of real relationships.



# Summary



# Summary

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