CPSC 436C
Cloud Computing for Data Science

Graph Processing

Maryam R.Aliabadi
mraiyata@cs.ubc.ca

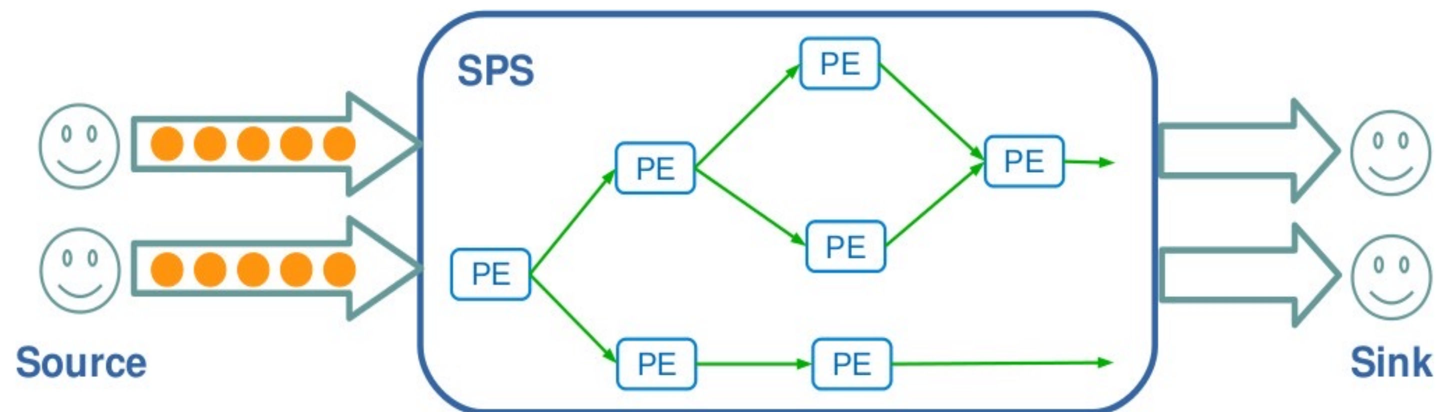
Spring 2024

Last Class's Review

- Spark Streaming
  - Run a streaming computation as a series of very small, deterministic batch jobs.
  - Discretized Stream (D-Stream): sequence of RDDs
  - Operators: Transformations (stateless, stateful, and window) and output operations

Streaming Data Processing

- The tuples are processed by the application's operators or processing element (PE).
- A PE is the basic functional unit in an application.
  - A PE processes input tuples, applies a function, and outputs tuples.
  - A set of PEs and stream connections, organized into a data flow graph.



Parallelization

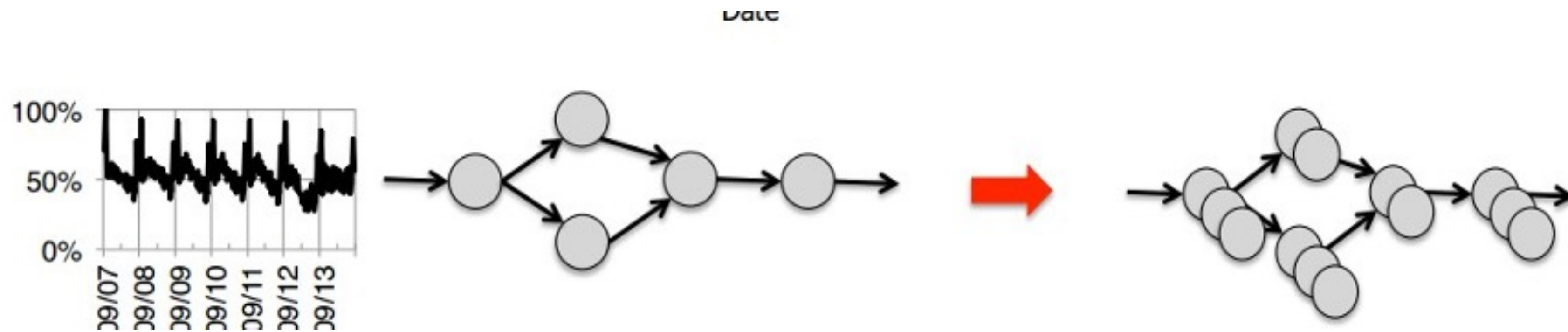
- How to scale with increasing the number queries and the rate of incoming events?
- Three forms of parallelisms.
  - Pipelined parallelism
  - Task parallelism
  - Data parallelism

Data Parallelism

- Users of big data applications expect fresh results.
- New stream processing systems are designed to scale to large numbers of cloud-hosted machines.
- Clouds provide virtually infinite pools of resources.
- Fast and cheap access to new machines (VMs) for operators.
- How do you decide on the optimal number of VMs?
  - Over-provisioning system is expensive.
  - Too few nodes leads to poor performance.

Challenges

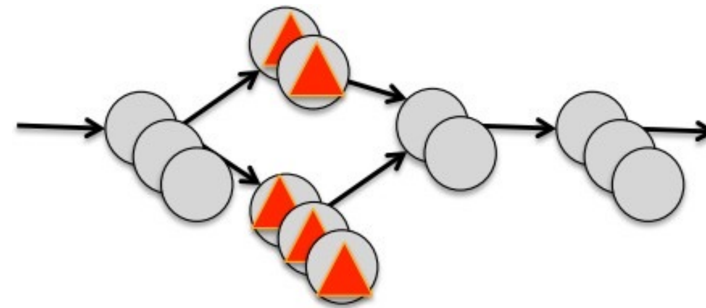
- Elastic data-parallel processing



- Fault-tolerant processing

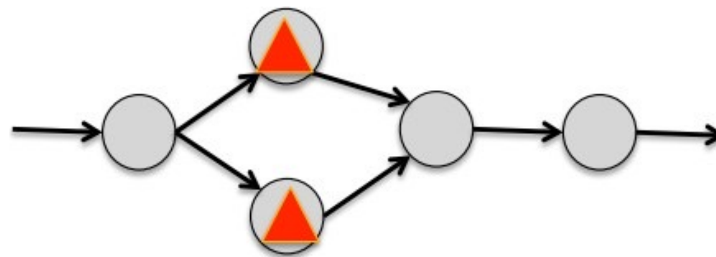
State Compliance Thing

- Dynamic scale out impacts state.



Partitioning of state

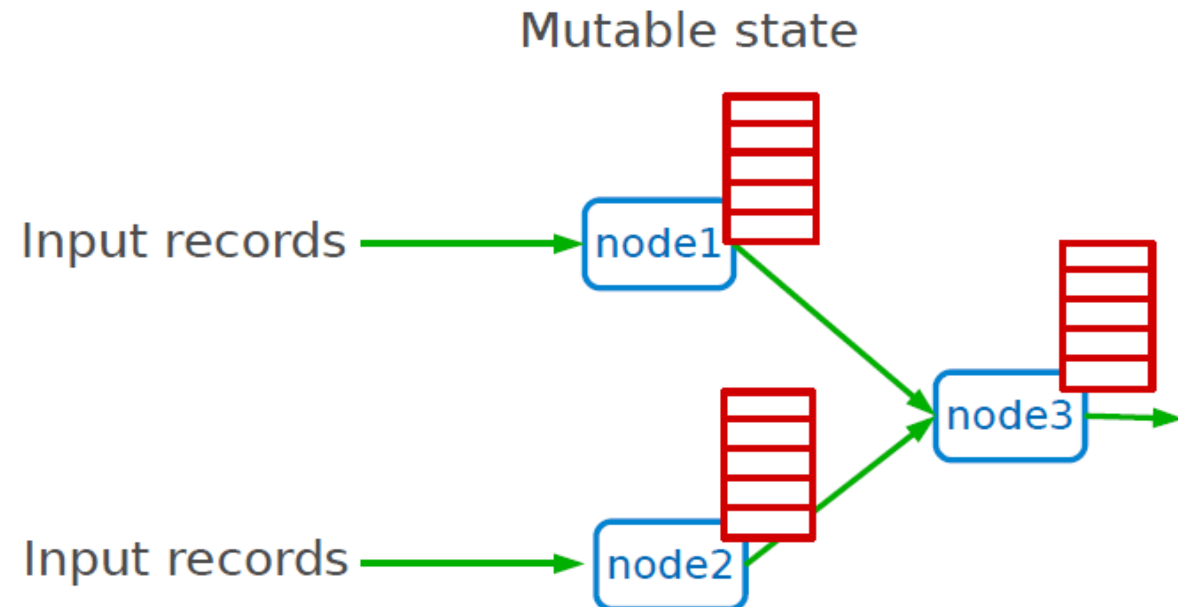
- Recovery from failures.



Loss of state after node failure

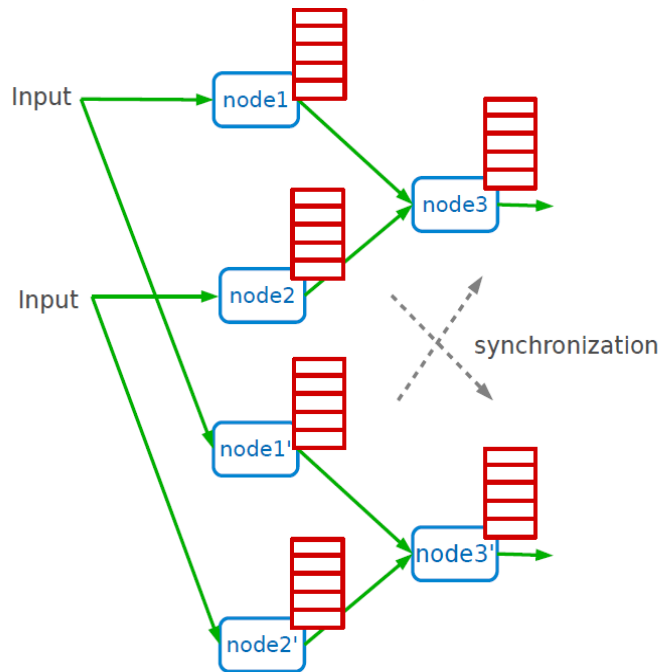
Traditional Streaming Systems

- Record-at-a-time processing model:
  - Each node has mutable state.
  - For each record, updates state and sends new records.
  - State is lost if node dies.

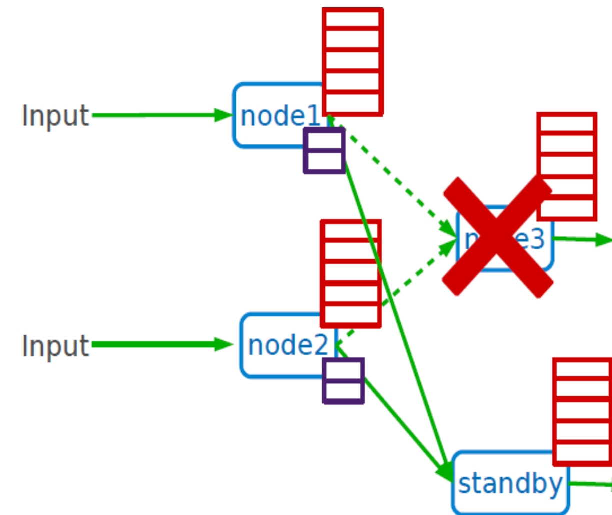


Traditional streaming systems

- Fault tolerance via replication or upstream backup.



Fast recovery, but 2x hardware cost



Only need one standby, but slow to recover

Proposed Solution : Spark Streaming

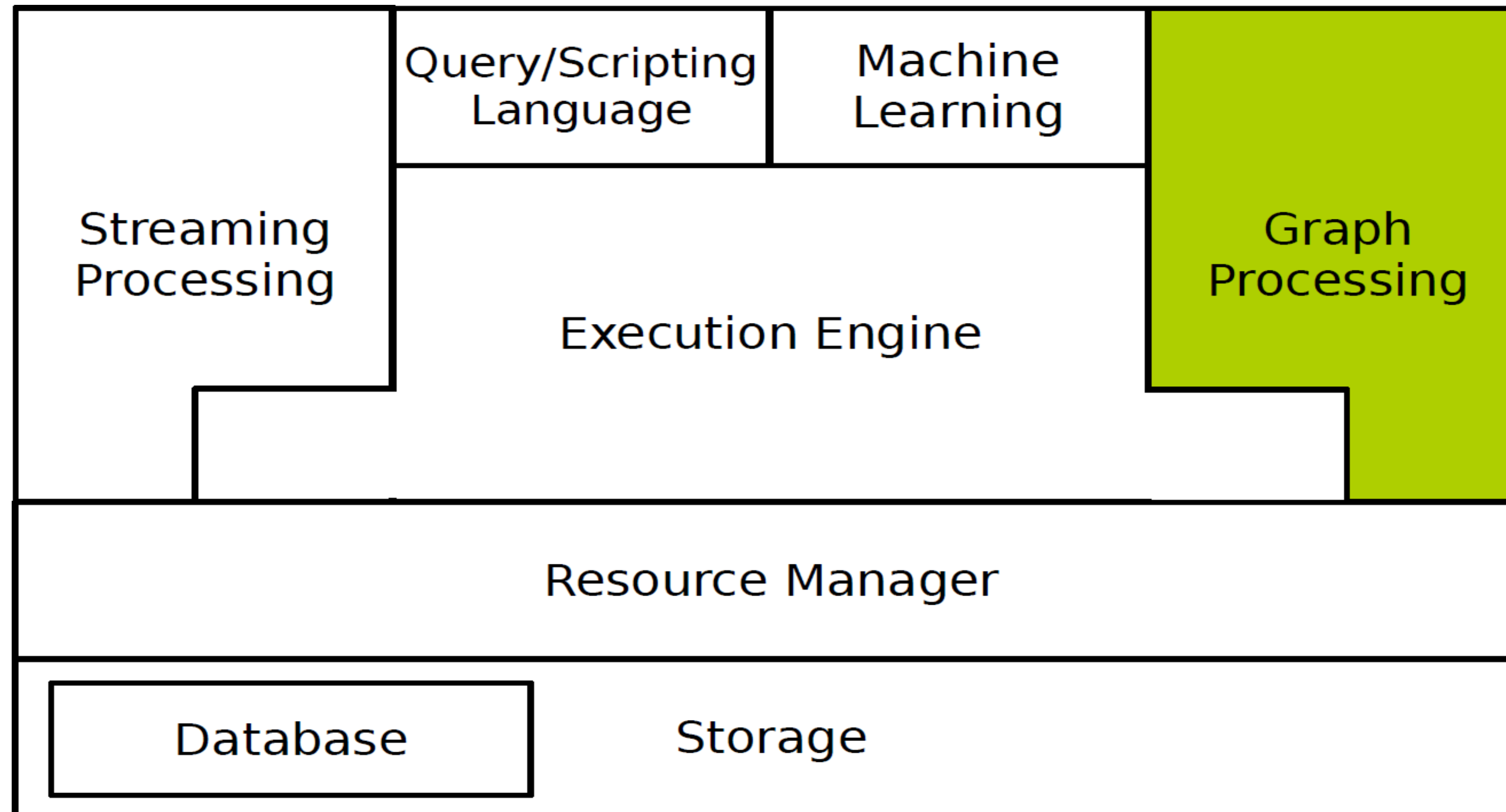
- Latency (interval granularity)
  - Resilient Distributed Dataset (RDD)
  - Keep data in memory
  - No replication
- Recovering quickly from faults and stragglers
  - Storing the lineage graph
  - Using the determinism of D-Streams (Discretized- Stream)
  - Parallel recovery of a lost node’s state

Spark Streaming

- Core idea: Run a streaming computation as a series of very small, deterministic batch jobs.
  - Chop up the live stream into batches of X seconds.
  - Spark treats each batch of data as RDDs and processes them using RDD operations.
  - Finally, the processed results of the RDD operations are returned in batches.
  - Discretized Stream (D-Stream) Processing

Today's Topics

- Graph Processing



SOUNDCLOUD

f

g+

Introduction

- Graphs provide a flexible abstraction for describing relationships between discrete objects.

- Many problems can be modeled by graphs and solved with appropriate graph algorithms.

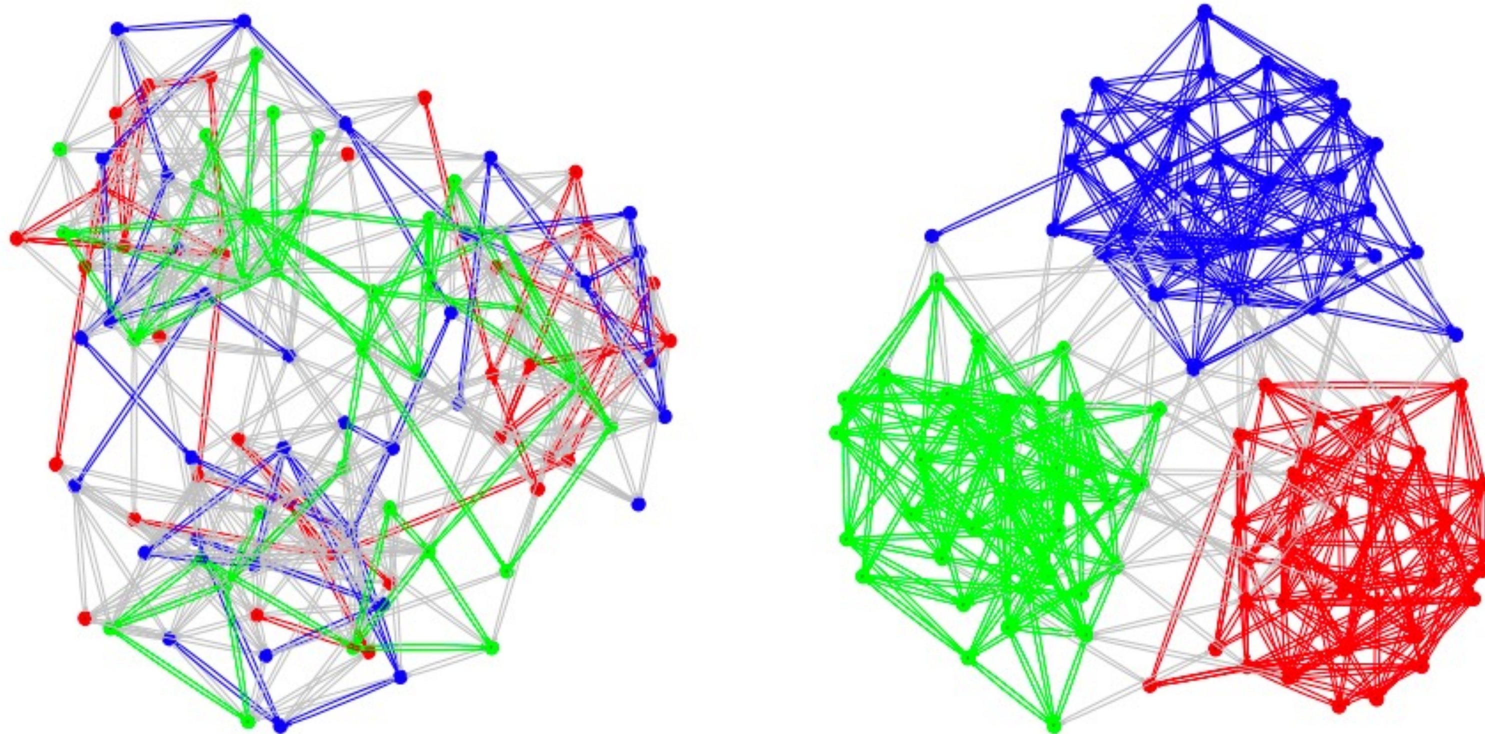
Large Graphs

Graph Algorithm 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.

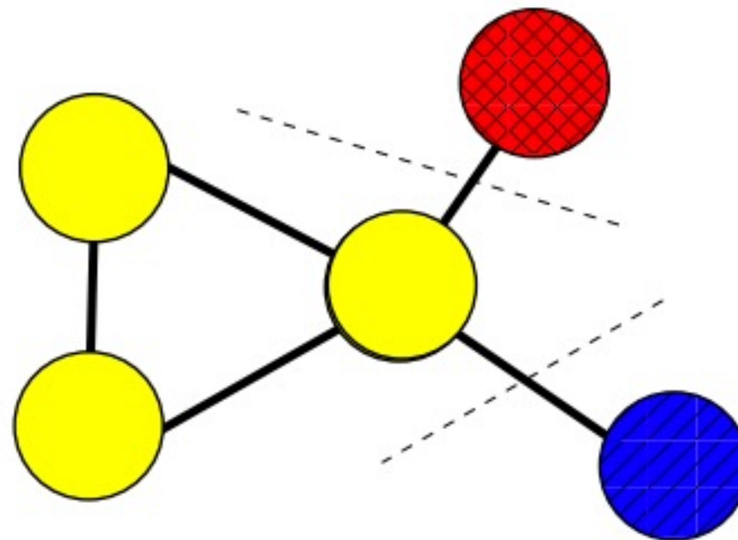
Graph Partitioning

- Partition large scale graphs and distribute to hosts.



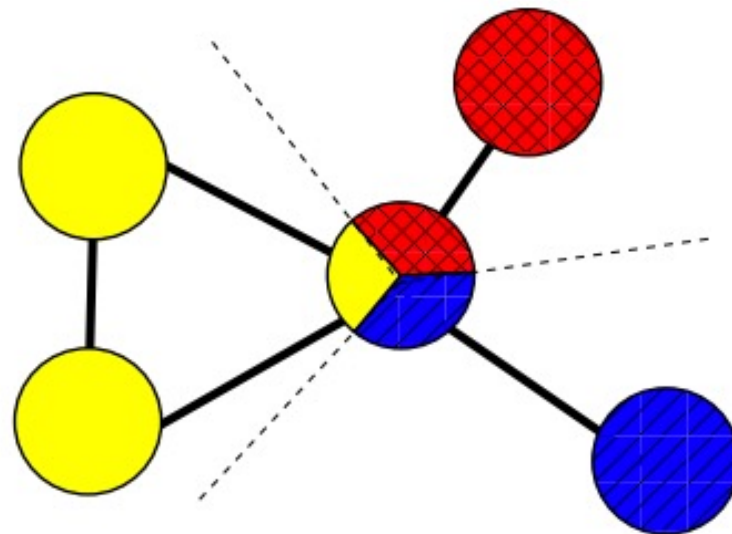
Edge-Cut Graph Processing

- 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.



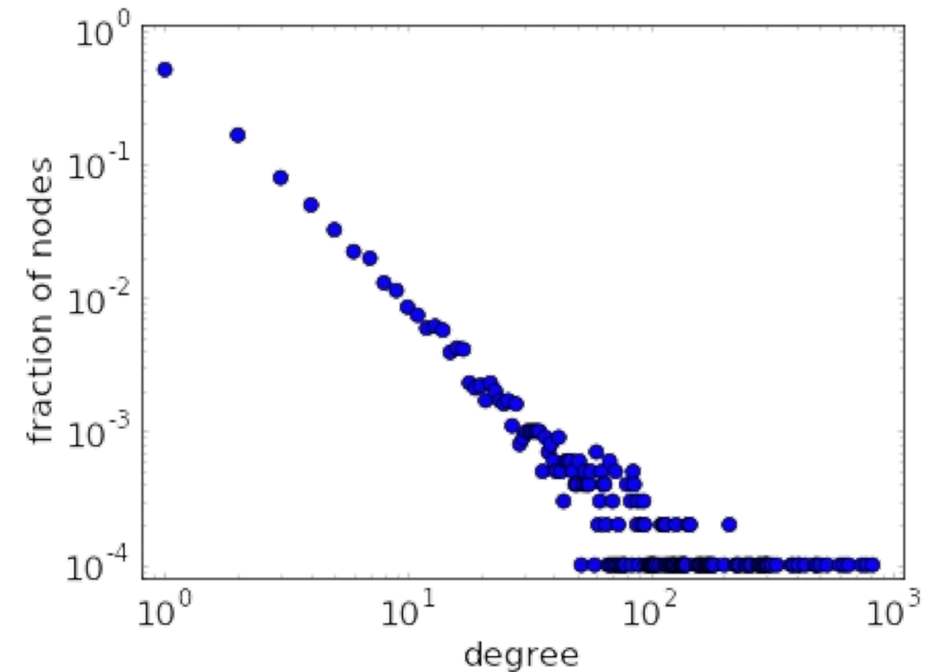
Vertex-Cut Graph Processing

- 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.

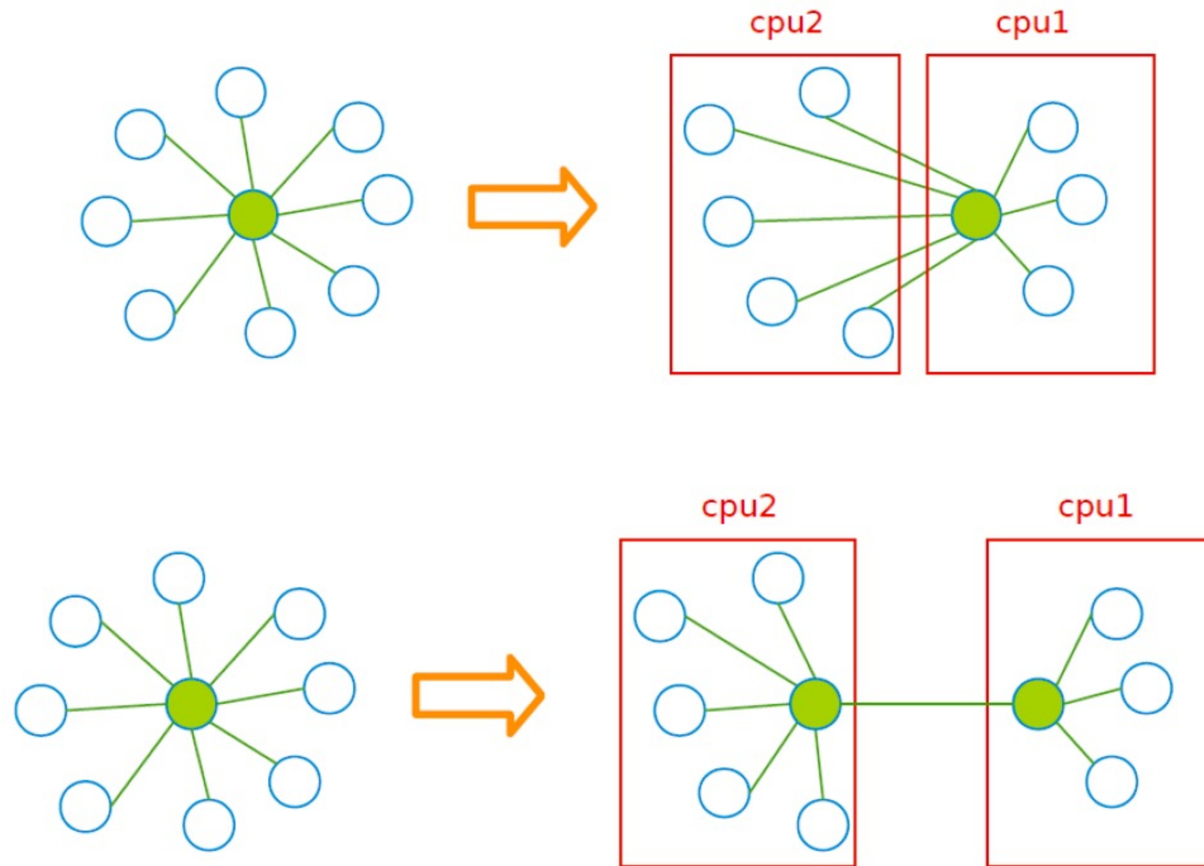


Edge-Cut Vs. Vertex-Cut Graph Partitioning

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



Edge-Cut Vs. Vertex-Cut Graph Partitioning



Large-Scale Graph Processing

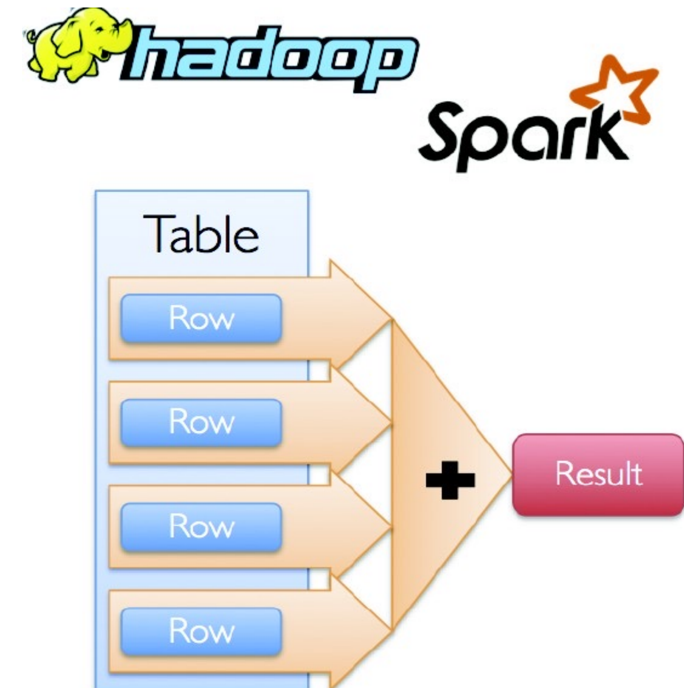
- Large graphs need large-scale processing.
- A large graph either cannot fit into memory of single computer or it fits with huge cost.

Large-Scale Graph Processing

Can we use execution engines like Spark, which is based on **Data-Parallel** model, for large-scale graph processing?

Large-Scale Graph Processing

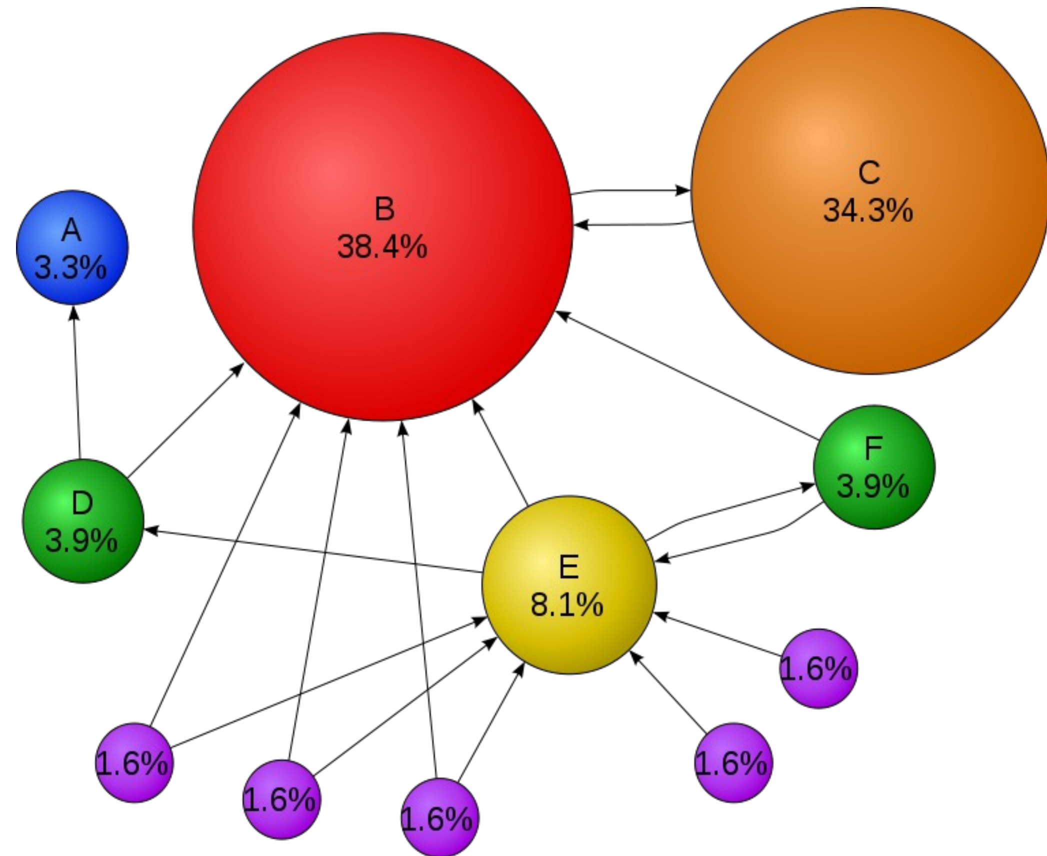
- The platforms that have worked well for developing parallel applications are not necessarily effective for large-scale graph problems.
- Why?



PageRank with MapReduce

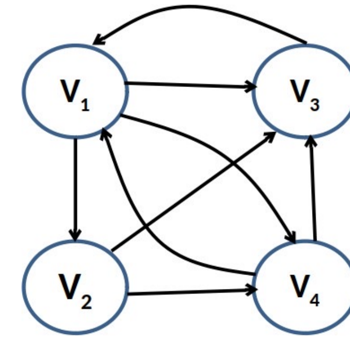
PageRank

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



PageRank Example

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

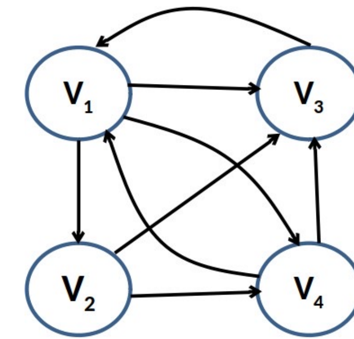



```
V1: [0.25, V2, V3, V4]
V2: [0.25, V3, V4]
V3: [0.25, V1]
V4: [0.25, V1, V3]
```

Number of vertices: ***n***
Initial weight: ***1/n***

PageRank Example

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



- Input

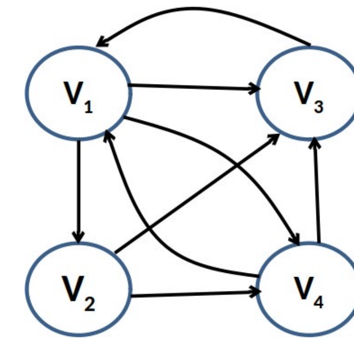
```
V1: [0.25, V2, V3, V4]
V2: [0.25, V3, V4]
V3: [0.25, V1]
V4: [0.25, V1, V3]
```

- 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

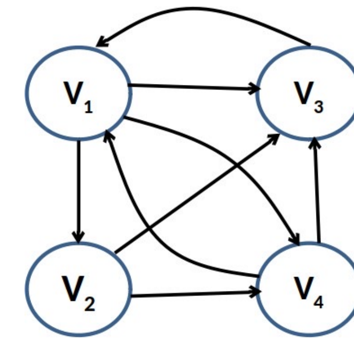
- $R[i] = \sum_{j \in Nbrs(i)} w_{ji} R[j]$




```
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

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



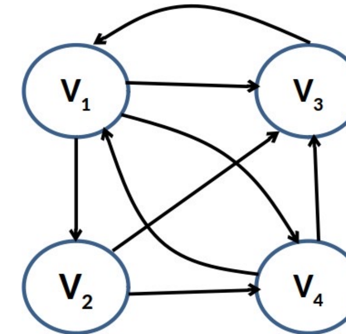

```
V1: (V2, 0.25/3), (V3, 0.25/3), (V4, 0.25/3)
V2: (V3, 0.25/2), (V4, 0.25/2)
V3: (V1, 0.25/1)
V4: (V1, 0.25/2), (V3, 0.25/2)
```

- Output after one iteration

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

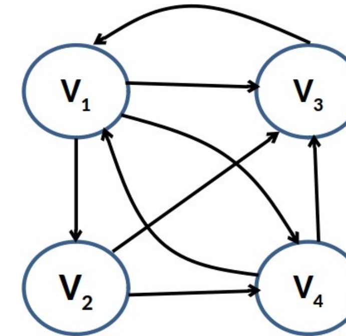
PageRank in MapReduce

- Map 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 function

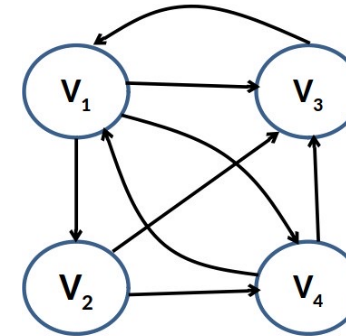
```
map(key: [url, pagerank], value: outlink_list)
  for each outlink in outlink_list:
        emit(key: outlink, value: pagerank / size(outlink_list))

  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])
```

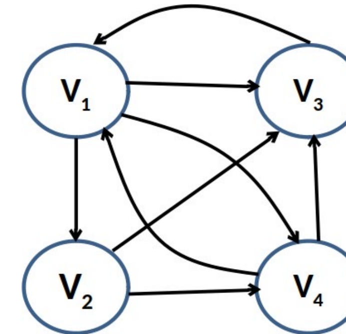
PageRank in MapReduce

- Map 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 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

- 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

- 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

- 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

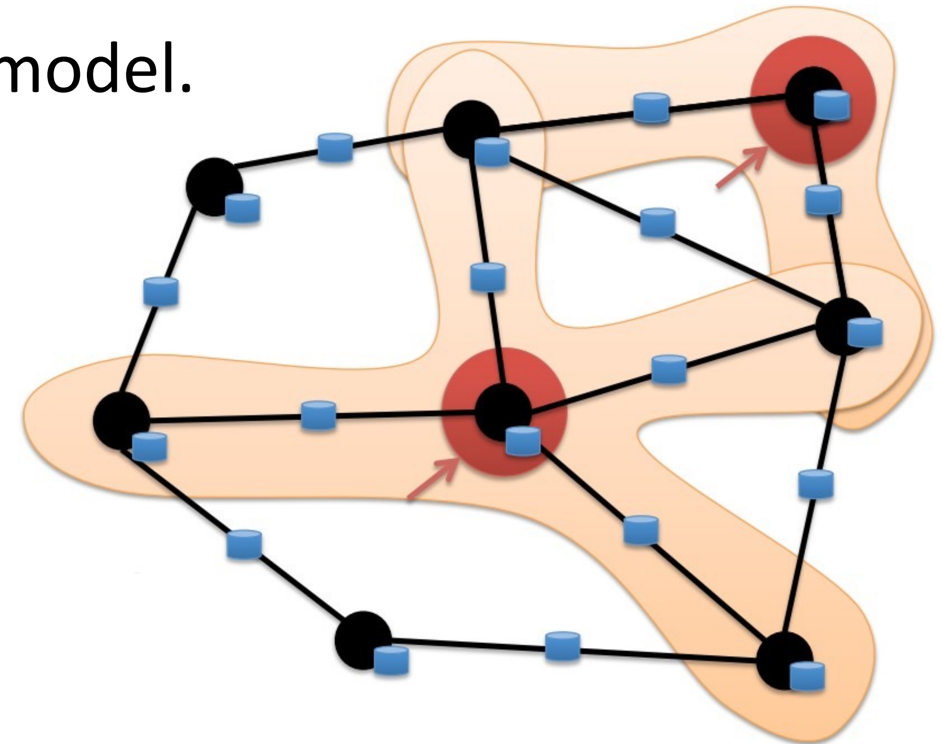
- MapReduce does not directly support iterative algorithms.
  - Invariant graph-topology-data re-loaded and re-processed at each iteration is wasting I/O, network bandwidth, and CPU

- Materializations of intermediate results at every MapReduce iteration harm performance.

Think Like a Vertex

Think like a Vertex

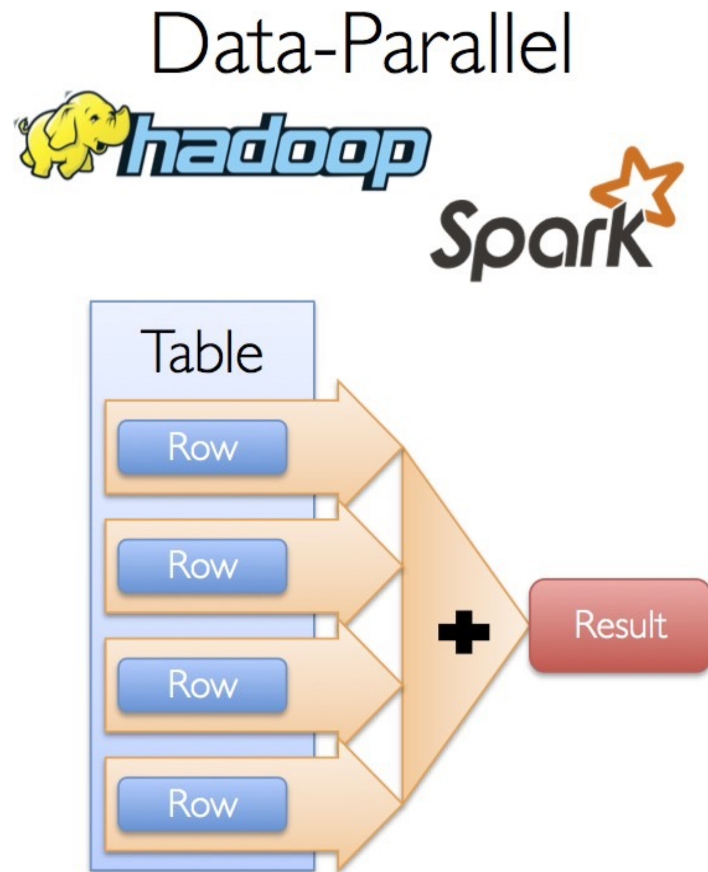
- Each vertex computes individually (in parallel)
- Computation typically depends on the neighbors
- Also know as graph-parallel processing model.

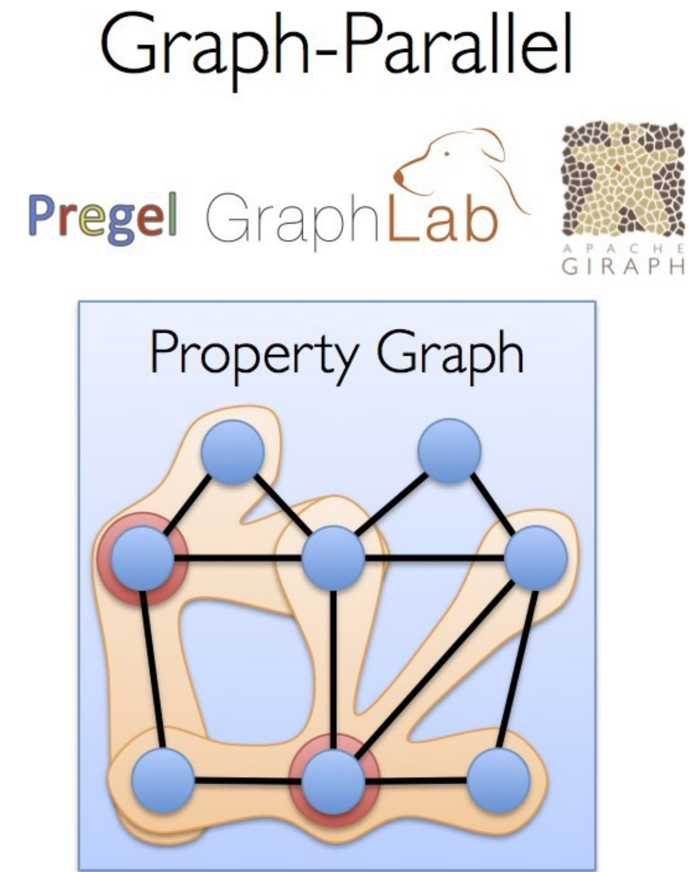


Graph-Parallel Processing

- Restricts the types of computation.
- New techniques to partition and distribute graphs.
- Exploit graph structure.
- Executes graph algorithms orders-of-magnitude faster than more general data-parallel systems.

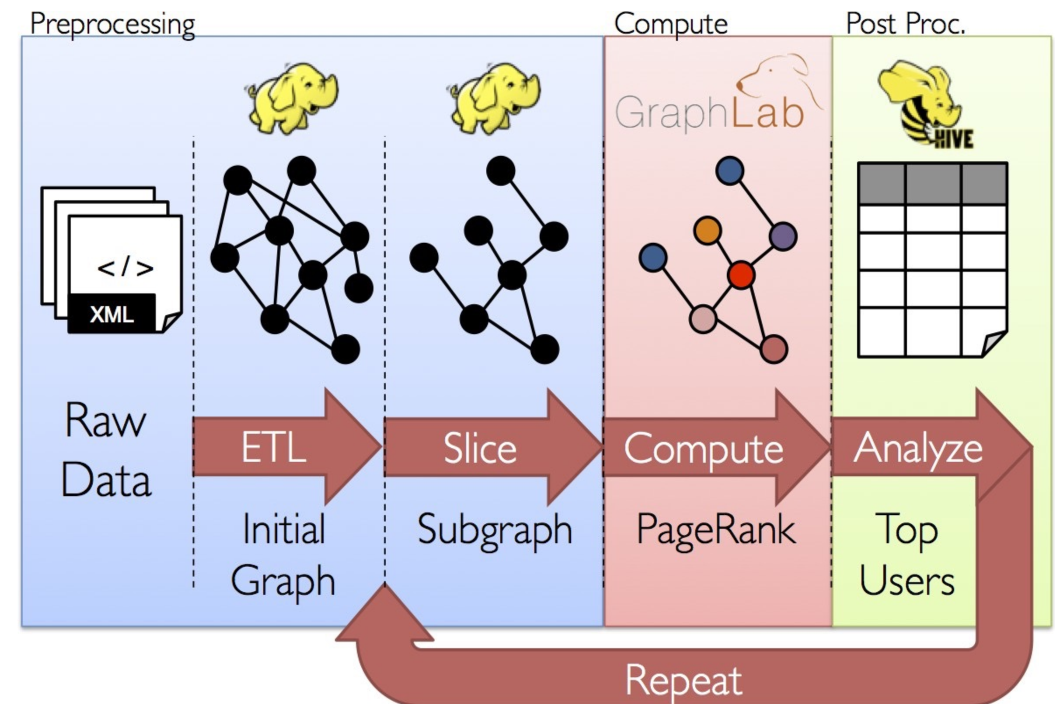
Data-Parallel vs. Graph-Parallel Computation



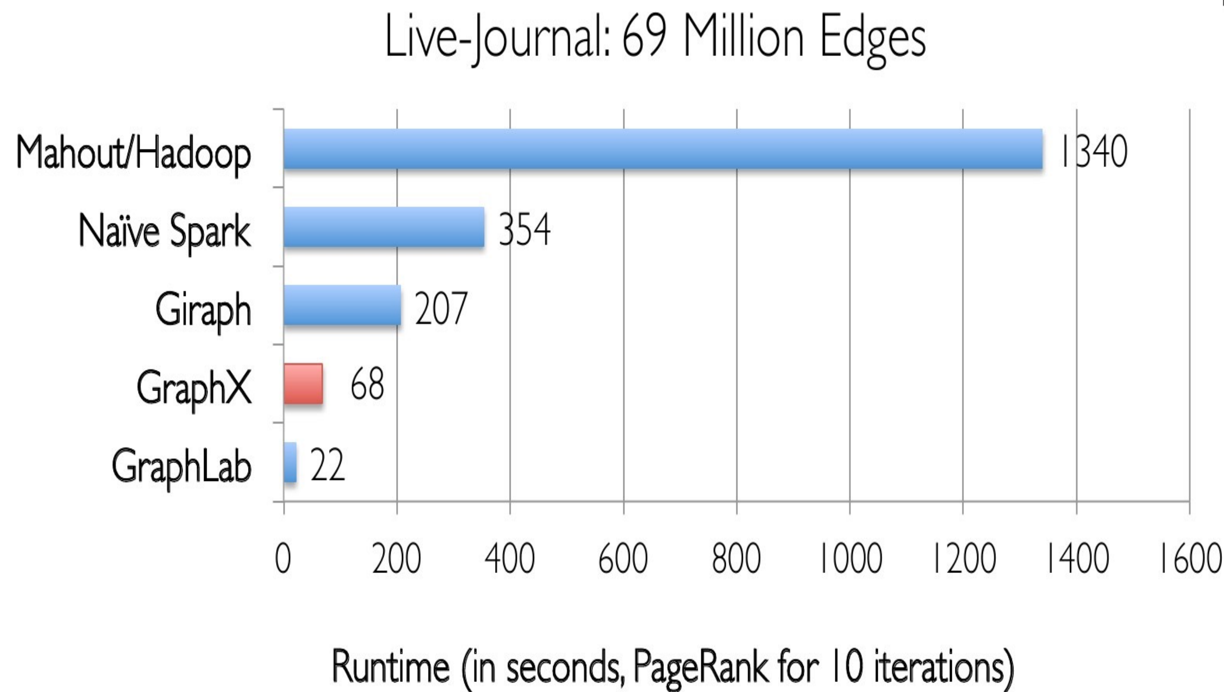


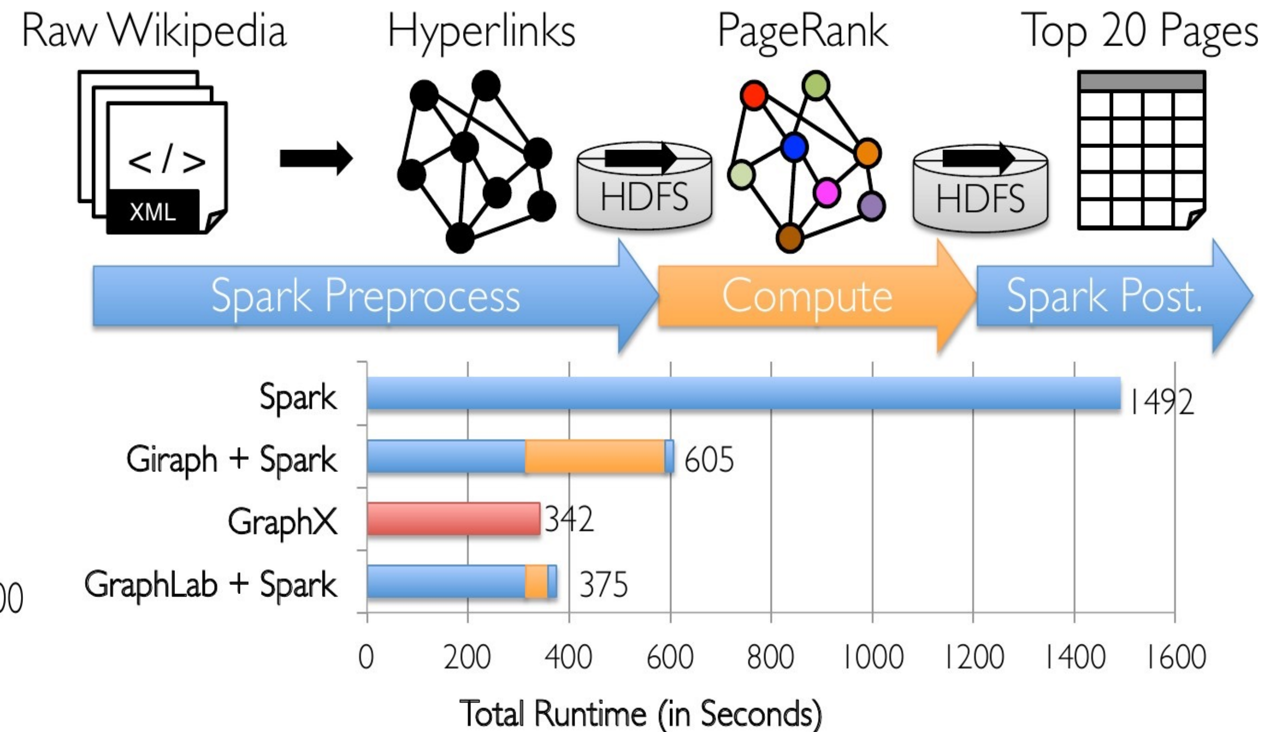
Data-Parallel vs. Graph-Parallel Computation

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



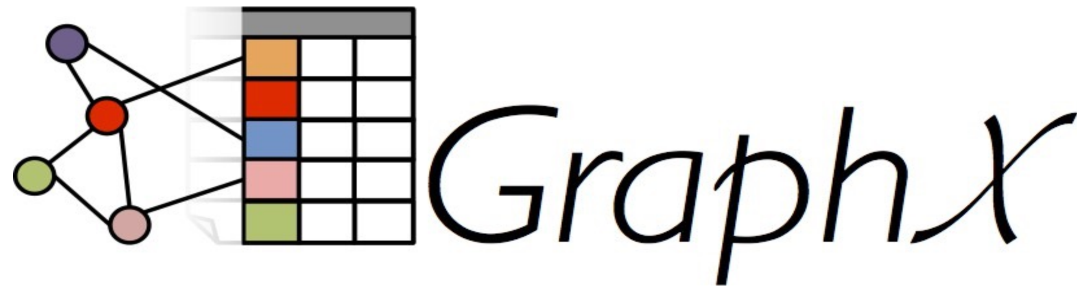
Graph-Parallel Systems





GraphX

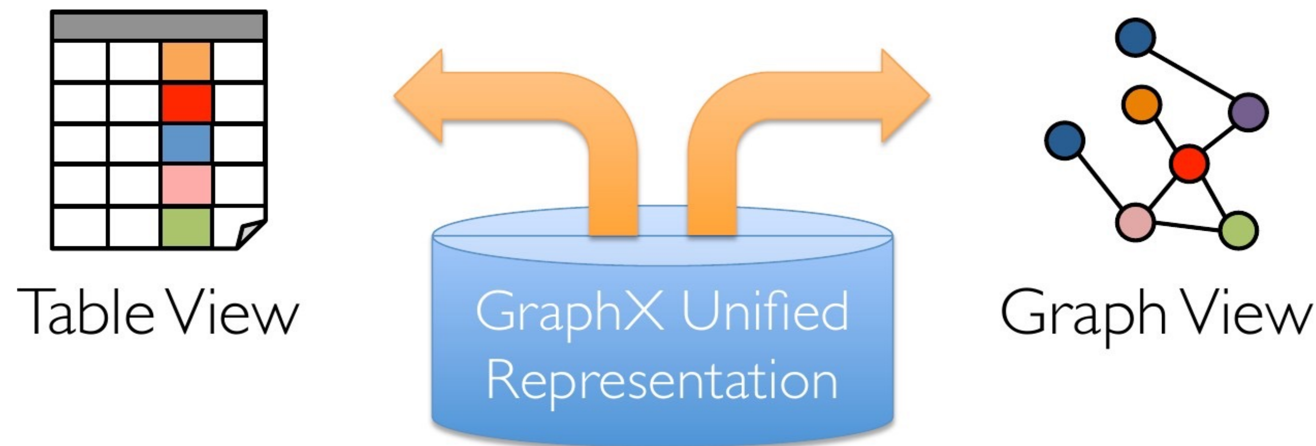
► GraphX is the library to perform graph-parallel processing in Spark.



Think Like a Table

Think like a Table

- Unifying Data-Parallel and Graph-Parallel Analytics
- Tables and Graphs are composable views of the same physical data.
- Each view has its own operators that exploit the semantics of the view to achieve efficient execution.

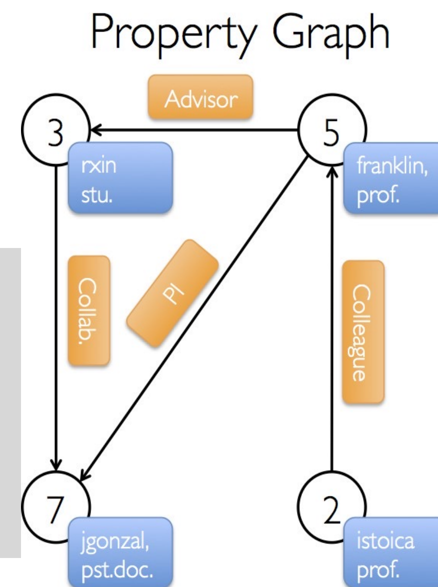


The Property Graph Data Model

- Spark represent graph structured data as a property graph.
- Property Graph: represented using two Spark RDDs:
  - Vertex collection: VertexRDD
  - Edge collection: 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, VD]
}
```

Property Graph



Vertex Table

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

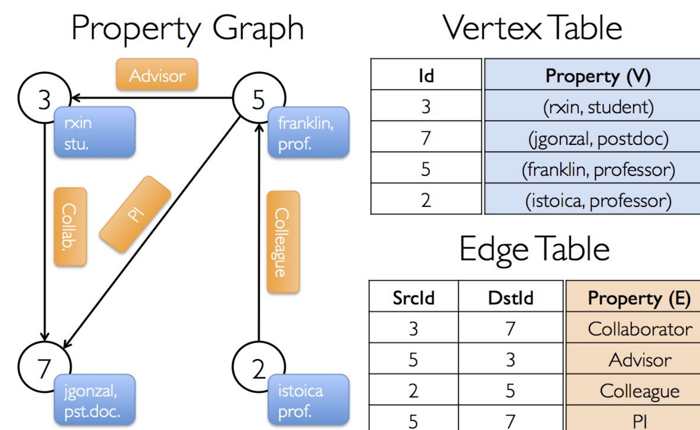
Edge Table

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

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)]
```



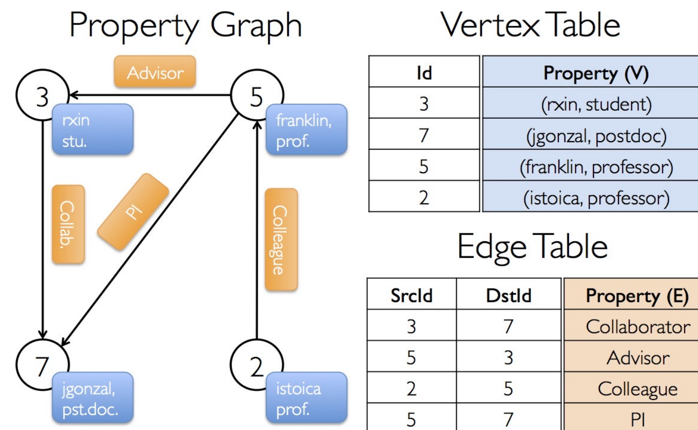
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

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

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]]
```



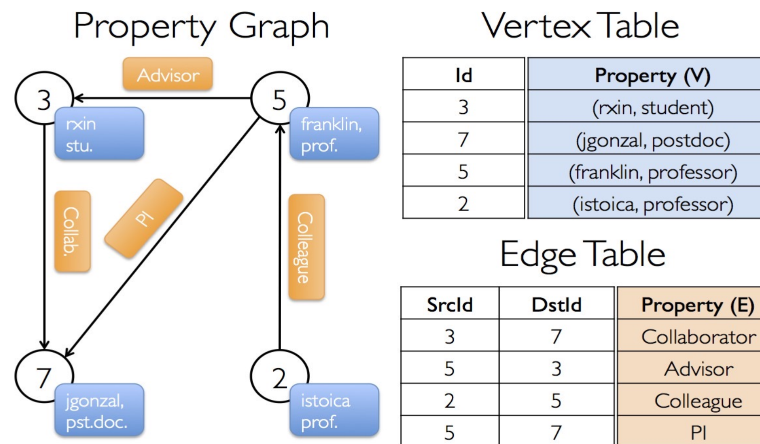
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

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

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



Vertex Table

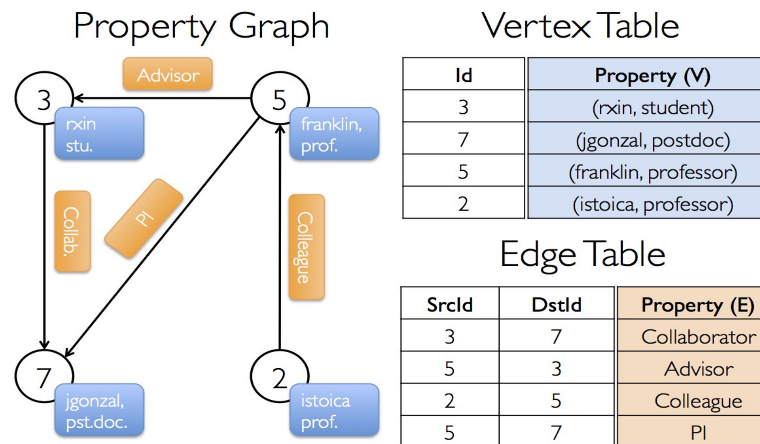
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

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

```
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



Vertex Table

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

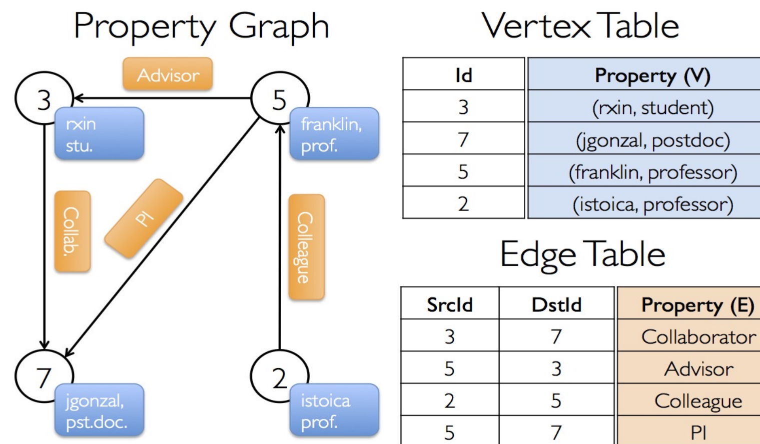
Edge Table

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

```
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



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

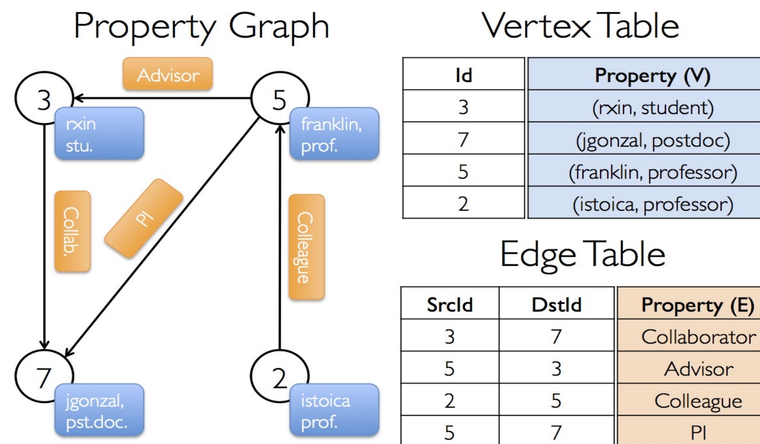
SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

```
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



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

```
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

- 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)

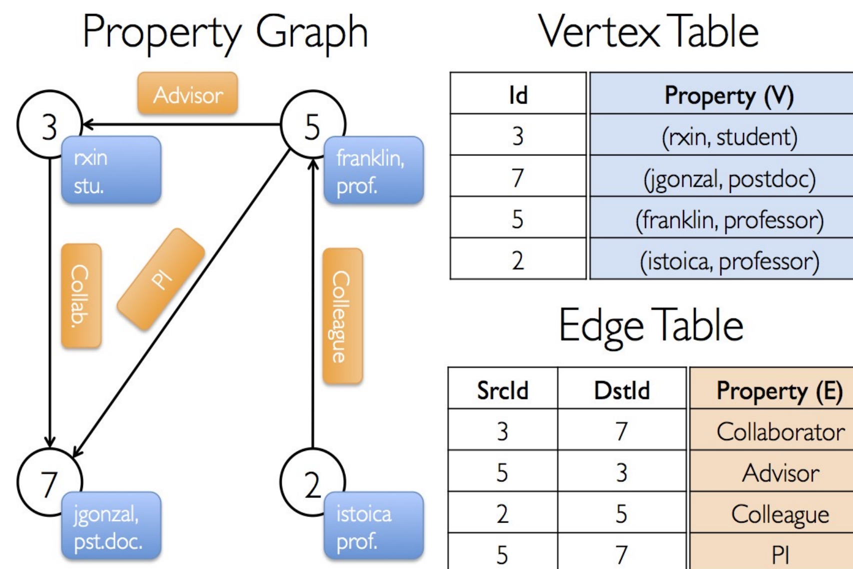
- Information about the graph

```
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)



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

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

```
// 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 Operator

- 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 Operator

- 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 Operator

- 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

- 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

- 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]
```

```
// 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)
```

Recap

- Large–scale graph processing
- Think like a vertex
  - Pregel, GraphLab, PowerGraph
- Think like a table
  - Graphx: unifies data-parallel and graph-parallel systems.

Next Class

Resource Management and Scheduling