UMD DATA605 - Big Data Systems

12.3: Graph Data Processing

- **Instructor**: Dr. GP Saggese, gsaggese@umd.edu

UNIVERSITY OF MARYLAND

SCIENCE ACADEMY

Options for Processing Graph Data

- Write your own programs
  - Extract relevant data, construct in-memory graph
  - Different storage options help to varying degrees
- Write queries in a declarative language
  - Suitable for some graph queries/tasks
  - E.g., Cypher for Neo4j
- Use a general-purpose distributed programming framework
  - E.g., Hadoop or Spark
  - Difficult for many graph analysis tasks
- Use a graph-specific programming framework
  - Simplifies writing graph analysis tasks, scales to large volumes
  - E.g., Giraph or GraphX

Option 2: Declarative Interfaces

- No consensus on declarative, high-level languages for querying or analysis
  - Variety in query/analysis tasks
  - Hard to find and exploit commonalities
- Limited, useful solutions:
  - XQuery for XML
    - Limited to tree-structured data
  - SPARQL for RDF
    - Standardized query language, limited functionality
  - Cypher by Neo4j
  - Datalog-based frameworks for analysis tasks
    - Many prototypes, task-specific

Option 3: MapReduce

- Popular option for processing large datasets
  - Hadoop or Spark
- Key advantages:
  - Scalability without scheduling, distributed execution, fault tolerance concerns
  - Simple programming framework
- Disadvantages:
  - Difficult for graph analysis tasks
  - Each traversal requires a new map-reduce phase
    - Hadoop not ideal for many phases, Spark is better
- Much work on graph analysis tasks using MapReduce

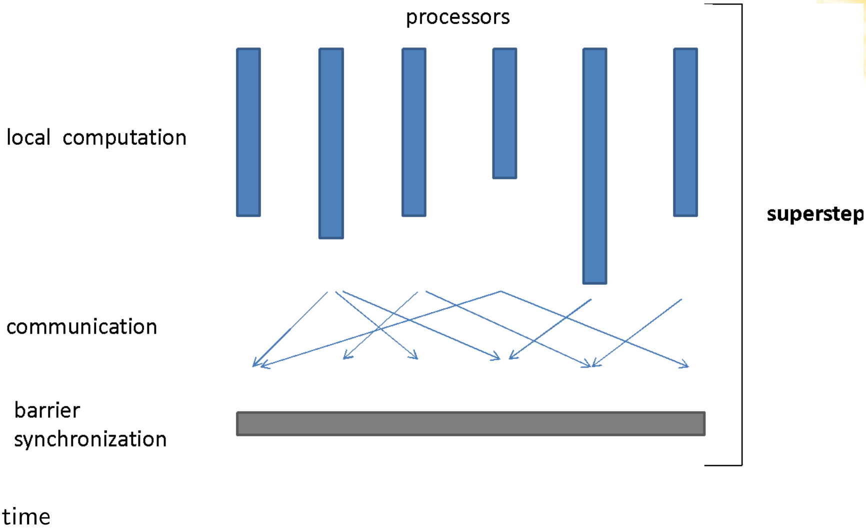
Option 3: MapReduce

- Disadvantages:
  - Difficult for graph analysis
  - Each traversal requires a new map-reduce phase
    - Each job executes N times
  - Hadoop not ideal for many phases (even with YARN)
  - Inefficient – redundant work
    - Mappers send PR values and graph structure
  - In PageRank: repeated reading and parsing each iteration
    - Extensive I/O at input, shuffle/sort, output

Option 4: Graph Programming Frameworks

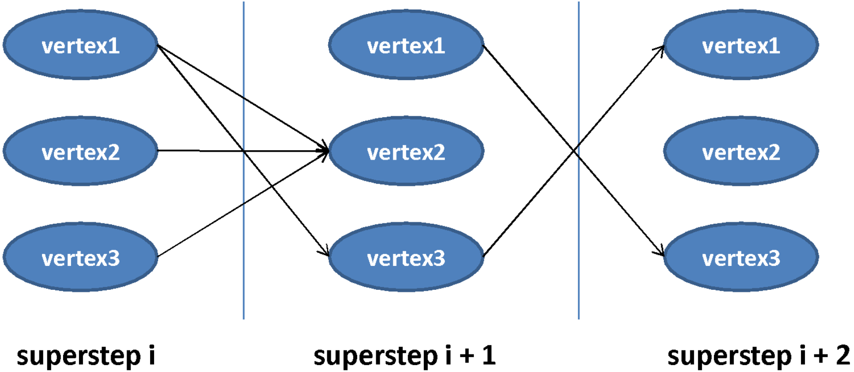
- Frameworks (analogous to MapReduce) for analyzing large graph data
  - Address MapReduce limitations
  - Most are *vertex-centric*
    - Programs from a vertex perspective
  - Based on message passing between nodes
- Pregel: original framework by Google
  - Based on "Bulk Synchronous Parallel" (BSP) model
- Giraph: open-source Pregel on Hadoop
- GraphLab: asynchronous execution
- GraphX: built on Spark

Bulk Synchronous Parallel (BSP)



Vertex-centric BSP

- Each vertex has an id, value, list of adjacent vertex ids, and edge values
- Each vertex invoked in each superstep, recomputes value, sends messages to other vertices, delivered over superstep barriers
- Advanced features: termination votes, combiners, aggregators, topology mutations



Think like a vertex

- I know my local state
- I know my neighbours
- I can send messages to vertices
- I can declare that I am done
- I can mutate graph topology

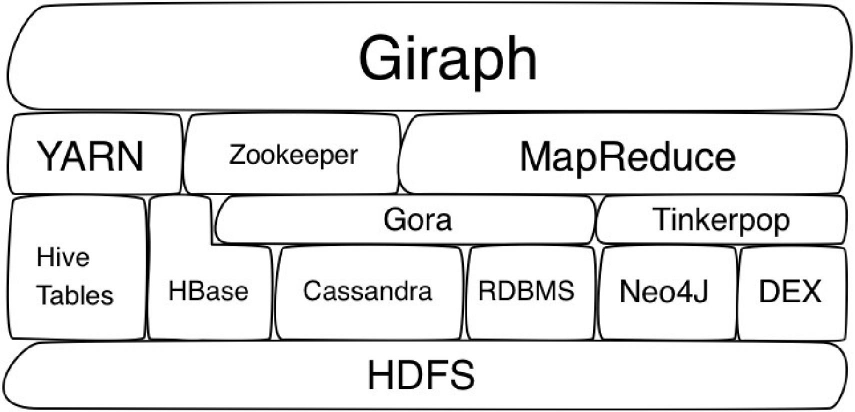
Option 4: Pregel

- Programmers write one program: `compute()`
- Typical structure of `compute()`:
  - *Inputs*: current values of the node
  - *Inputs*: messages from neighboring nodes
  - Modify current values (if desired)
  - *Outputs*: send messages to neighbors
- Execution framework:
  - Execute *compute()* for all nodes in parallel
  - Synchronize (wait for all messages)
  - Repeat

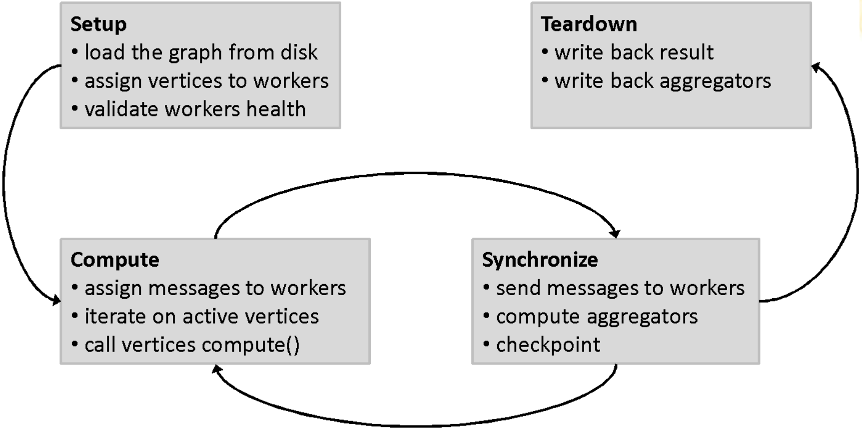
Apache Giraph

- Pregel is proprietary, but:
  - **Apache Giraph**: open source implementation
  - Runs on standard **Hadoop** infrastructure
  - Computation executed in memory
  - Can be a job in a pipeline (**MapReduce, Hive**)
  - Uses **Apache ZooKeeper** for synchronization
  - Graph partition via hashing
  - Fault tolerance via checkpointing

Plays well with Hadoop



Giraph Execution



Which part is doing what?

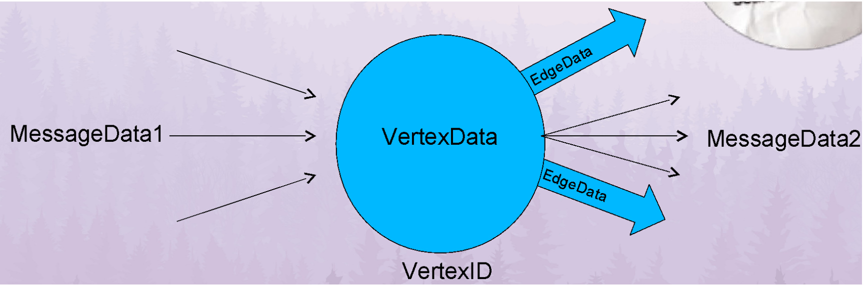
- **ZooKeeper**: responsible for computation state
  - partition/worker mapping
  - global state: #superstep
  - checkpoint paths, aggregator values, statistics
- **Master**: responsible for coordination
  - assigns partitions to workers
  - coordinates synchronization
  - requests checkpoints
  - aggregates aggregator values
  - collects health statuses
- **Worker**: responsible for vertices
  - invokes active vertices compute() function
  - sends, receives, assigns messages
  - computes local aggregation values

What do you have to implement?

- Your algorithm as a **Vertex**
  - Subclass existing implementations: `BasicVertex`, `MutableVertex`, `EdgeListVertex`, `HashMapVertex`, `LongDoubleFloatDoubleVertex`
- A `VertexInputFormat` to read your graph
  - e.g., from a text file with adjacency lists like ...
- A `VertexOutputFormat` to write back the result
  - e.g.,

SCIENCE ACADEMY

A vertex view



Designed for iterations

- Stateful (in-memory)
  - Keep all data in memory if possible
- Only intermediate values (messages) sent
  - Communicate with other vertices
- Hits disk at input, output, checkpoint
- Can go out-of-core
  - If data doesn't fit into memory

Graph modeling in Giraph

- BasicComputation$<$ I extends WritableComparable, // VertexID – vertex ref V extends Writable, // VertexData – a vertex datum E extends Writable, // EdgeData – an edge label M extends Writable$>$ // MessageData-– message payload

Giraph "Hello World"

```
public class GiraphHelloWorld extends
BasicComputation<IntWritable, IntWritable, NullWritable, NullWrita

public void compute(Vertex<IntWritable, IntWritable,     NullWrita
   System.out.println("Hello world from the: " + vertex.getId() +
      System.out.println(" " + e.getTargetVertexId());     }
   System.out.println("");
   vertex.voteToHalt();
}
}
```

SCIENCE ACADEMY

Example: Ping neighbors

```
public void compute(Vertex<Text, DoubleWritable, DoubleWritable> v
   if (getSuperstep() == 0) {         sendMessageToAllEdges(vertex,
   } else {         for (Text m : ms) {
          if (vertex.getEdgeValue(m) == null) {                vert
   }      vertex.voteToHalt();
}
```

SCIENCE ACADEMY

Giraph PageRank Example

```
public class PageRankComputation      extends BasicComputation<Int
  //Number of supersteps
  public static final String SUPERSTEP_COUNT =              "girap
```

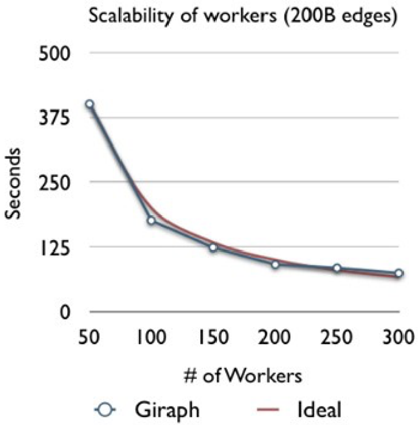
Giraph PageRank Example

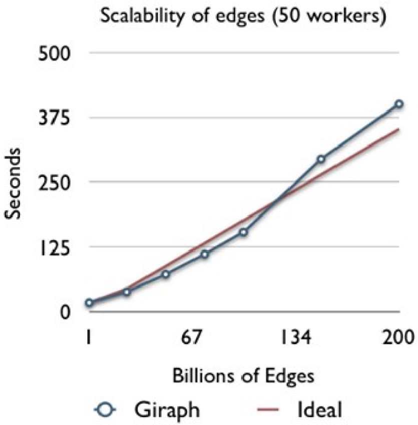
```
public void compute(Vertex<IntWritable, FloatWritable, NullWritabl
    if (getSuperstep() >= 1) {
      float sum = 0;
      for (FloatWritable message : messages) {
        sum += message.get();
      }
      vertex.getValue().set((0.15f / getTotalNumVertices()) +
    }
    if (getSuperstep()<getConf().getInt(SUPERSTEP_COUNT, 0)) {
      sendMessageToAllEdges(vertex,
          new FloatWritable(vertex.getValue().get() /
    } else {
      vertex.voteToHalt();
    }
  }
}
```

Additional functionality

- Combiners
  - To minimize messages
- Aggregators
  - global aggregations across vertices
- MasterCompute
  - computation executed on master
- WorkerContext
  - executed per worker task
- PartitionContext
  - executed per partition

Giraph scales

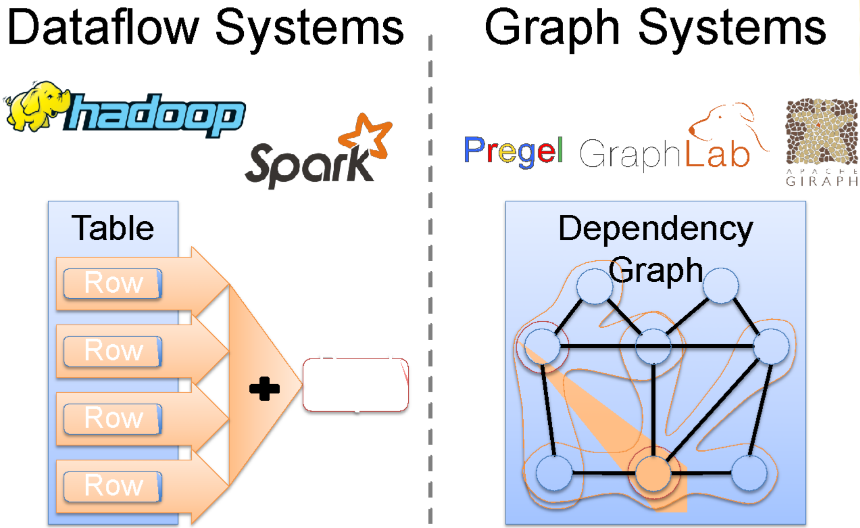




https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-

Graphx

GraphX Motivation



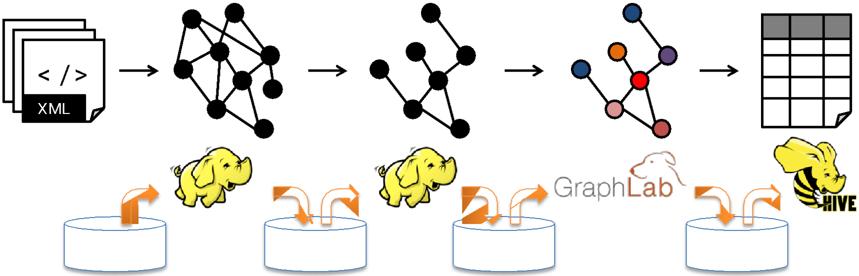
GraphX Motivation

- Difficult to Program and Use
- Users must ***Learn, Deploy, and Manage*** multiple systems

- Leads to brittle and often complex interfaces

And Inefficient

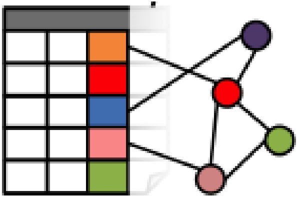
- Extensive **data movement** and duplication across the network and file system



- Limited reuse of internal data-structures across stages

The GraphX Unified Approach

New API Blurs the distinction between *Tables* and *Graphs*

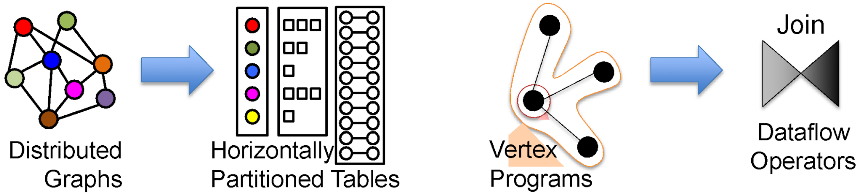


New System Combines Data-Parallel Graph-Parallel Systems

APACHE GIRAPH · Spark · GraphLab

Enables users to easily and efficiently express the entire graph analytics pipeline

Representation



- Plus optimizations:
  - Distributed join optimization
  - Materialized view maintenance

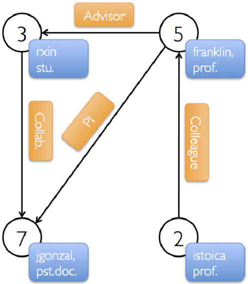
Graph modeling in GraphX

- The property graph is parameterized over the vertex (VD) and edge (ED) types

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

- Graph[(String, String), String]

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

Creating a Graph (Scala)

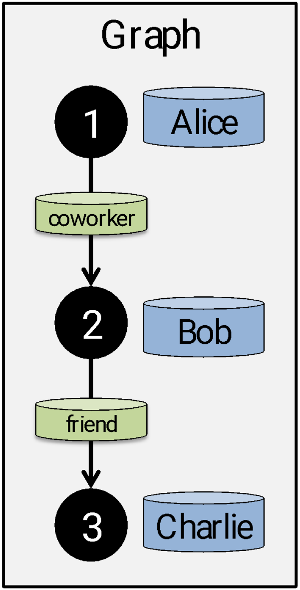
```
type VertexId = Long

val vertices: RDD[(VertexId, String)] =
sc.parallelize(List(
(1L,"Alice"),
(2L, "Bob"),
(3L, "Charlie")))

class Edge[ED](
val srcId: VertexId,
val dstId: VertexId,
val attr: ED)

val edges: RDD[Edge[String]] =
sc.parallelize(List(
Edge(1L, 2L, "coworker"),
Edge(2L, 3L, "friend")))

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



Hello world in GraphX

```
$ spark*/bin/spark-shell
scala> val inputFile = sc.textFile("hdfs:///tmp/graph/1.txt")
scala> val edges = inputFile.flatMap(s $\implies$ {
val l = s.split("\t");
l.drop(1).map(x $\implies$ (l.head.toLong, x.toLong))
})
scala> val graph = Graph.fromEdgeTuples(edges, "")
scala> val result = graph.collectNeighborIds(EdgeDirection.Out).ma
println("Hello world from the: " + x._1 + " : " + x._2.mkString("
scala> result.collect() // don't try this @home

Hello world from the: 1 :
Hello world from the: 2 : 1 3
Hello world from the: 3 : 1 2
```

SCIENCE ACADEMY

Spark Table Operators

- GraphX **Table** (RDD) operators are inherited from Spark:

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin

- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip

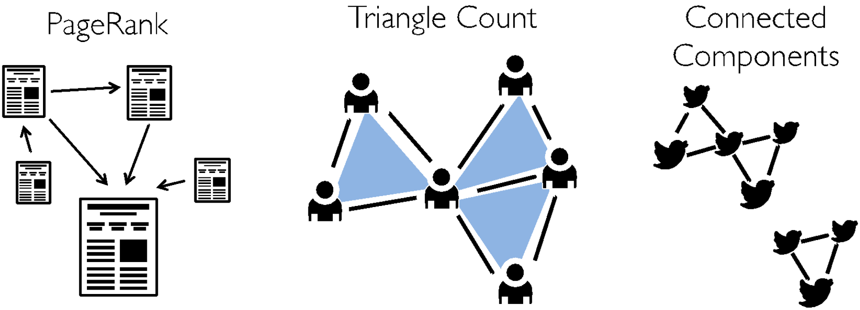
- sample
- take
- first
- partitionBy
- mapWith
- pipe
- save
- ...

Graph Operators (Scala)

```
class Graph [ V, E ] {
   def Graph(vertices: Table[ (Id, V) ],
                     edges: Table[ (Id, Id, E) ])
   // Table Views -----------------
   def vertices: Table[ (Id, V) ]
   def edges: Table[ (Id, Id, E) ]
   def triplets: Table [ ((Id, V), (Id, V), E) ]
   // Transformations ------------------------------
   def reverse: Graph[V, E]
   def subgraph(pV: (Id, V) $\implies$ Boolean,
                          pE: Edge[V,E] $\implies$ Boolean): Graph[V,E]
   def mapV(m: (Id, V) $\implies$ T ): Graph[T,E]
   def mapE(m: Edge[V,E] $\implies$ T ): Graph[V,T]
   // Joins ----------------------------------------
   def joinV(tbl: Table [(Id, T)]): Graph[(V, T), E ]
   def joinE(tbl: Table [(Id, Id, T)]): Graph[V, (E, T)]
   // Computation ----------------------------------
   def mrTriplets(mapF: (Edge[V,E]) $\implies$ List[(Id, T)],
                           reduceF: (T, T) $\implies$ T): Graph[T, E]
}
```

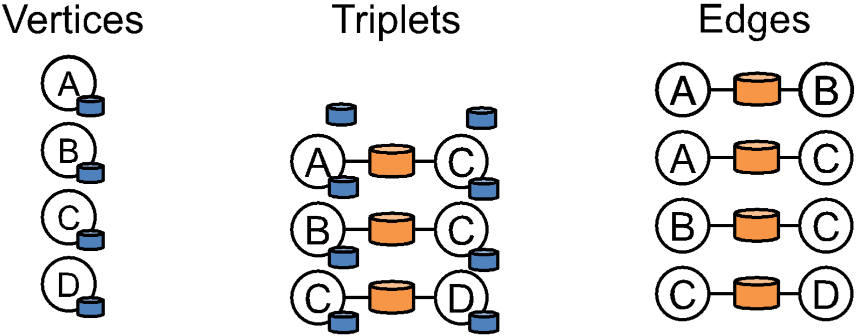
Built-in Algorithms (Scala)

```
def pageRank(tol: Double): Graph[Double, Double]
def triangleCount(): Graph[Int, ED]
def connectedComponents(): Graph[VertexId, ED]
def stronglyConnectedComponents(numIter:Int):Graph[VertexID,ED]
```



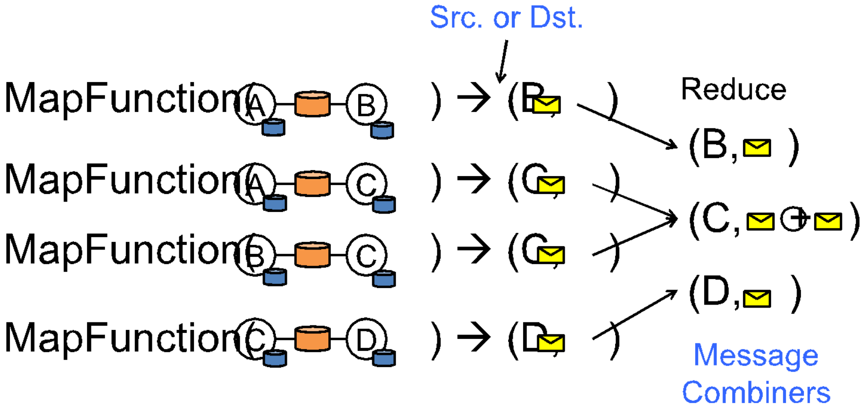
Triplets Join Vertices and Edges

- Triplets capture Gather-Scatter pattern from specialized graph processing systems (like Giraph)
- **Triplets** operator joins vertices and edges

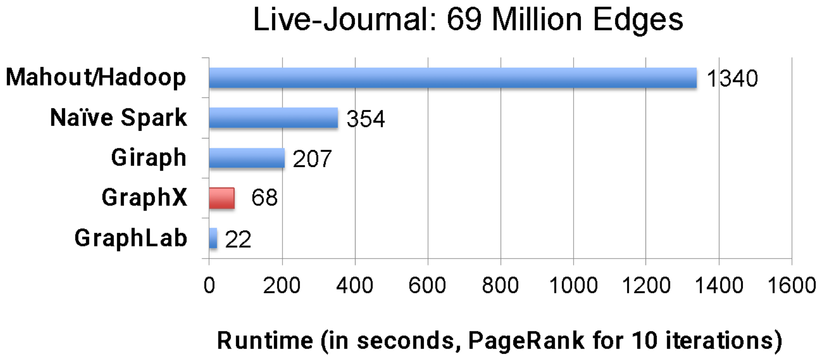


MapReduce Triplets

Map-Reduce triplets collect information about the neighborhood of each vertex:

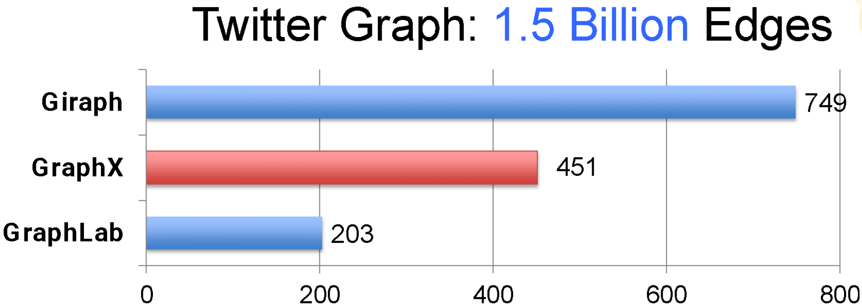


Performance Comparisons



GraphX is roughly 3x slower than GraphLab

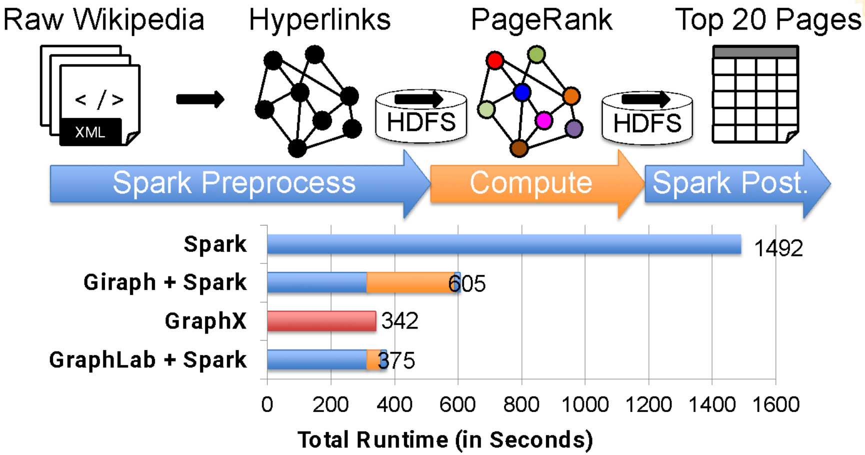
GraphX scales to larger graphs



- GraphX is roughly 2x slower than GraphLab - Scala + Java overhead: Lambdas, GC time, ... - No shared memory parallelism: 2x increase in communication

But, a Small Pipeline in GraphX

Timed end-to-end GraphX is faster than GraphLab (and Giraph)



Giraph vs. GraphX

- **Giraph**
  - An unconstrained BSP framework
  - Specialized fully mutable, dynamically balanced in-memory graph representation
  - Procedural, vertex-centric programming model
  - Part of Hadoop ecosystem

- **GraphX**
  - An RDD framework
  - Graphs are "views" on RDDs and thus immutable
  - Functional-like, "declarative" programming model
  - Genuine part of Spark ecosystem