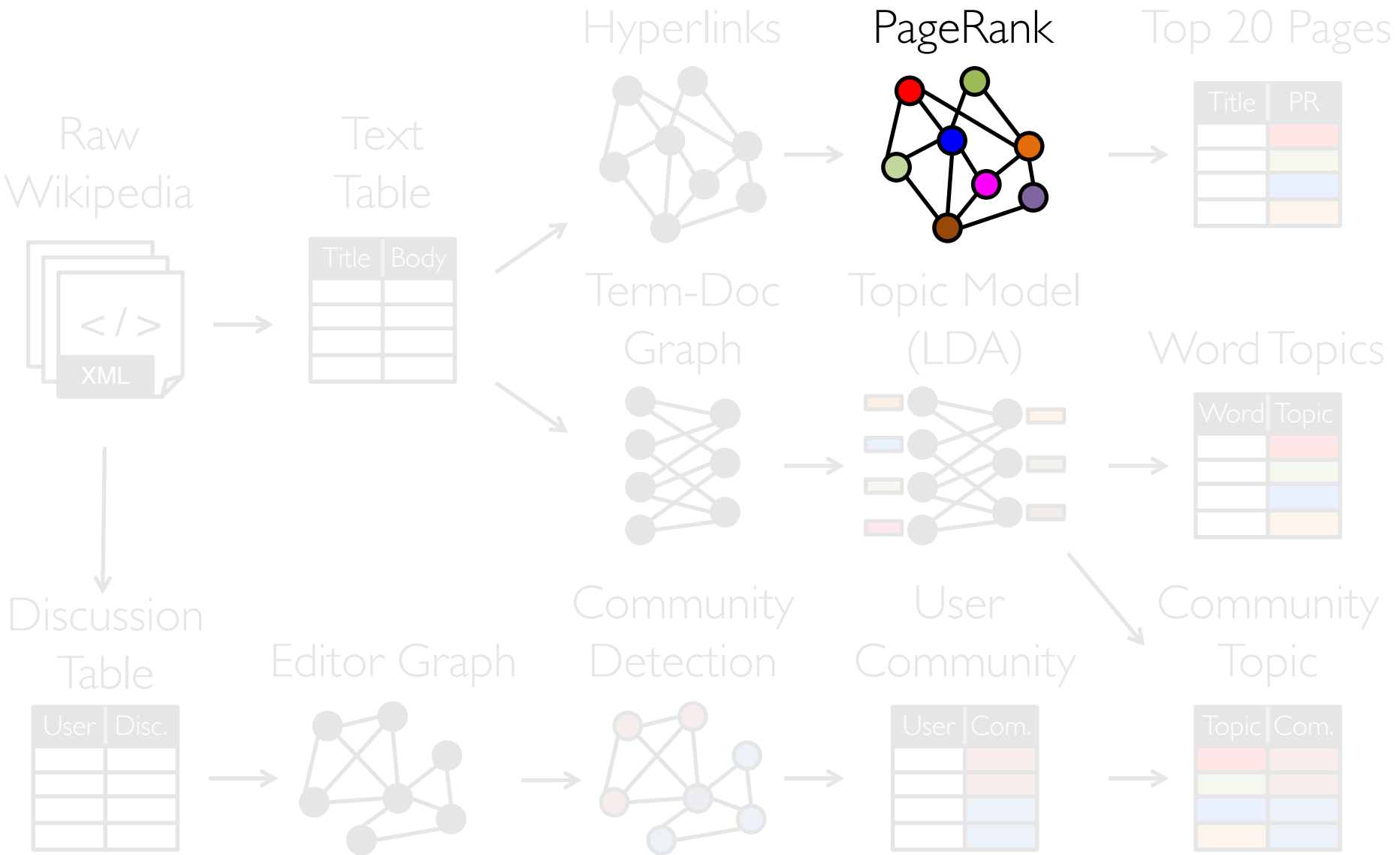
GraphX:
Unifying Data-Parallel and Graph-Parallel Analytics

Courtesy of Joseph Gonzalez

Graphs are Central to Analytics



PageRank: Identifying Leaders

$$R[i] = 0.15 + \sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]$$

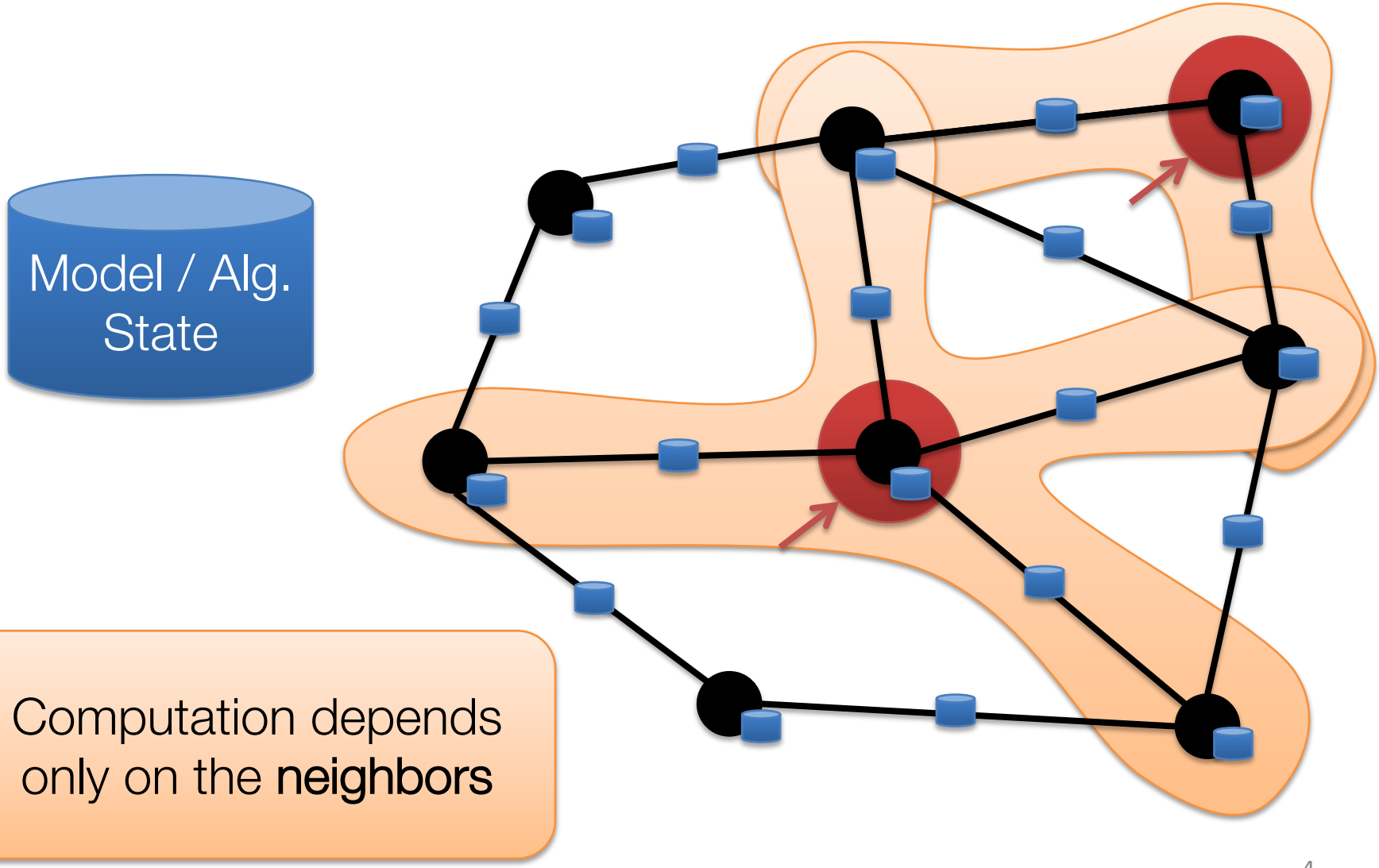
Rank of user *i*

Weighted sum of neighbors' ranks

Update ranks in parallel

Iterate until convergence

The Graph-Parallel Pattern



Many Graph-Parallel Algorithms

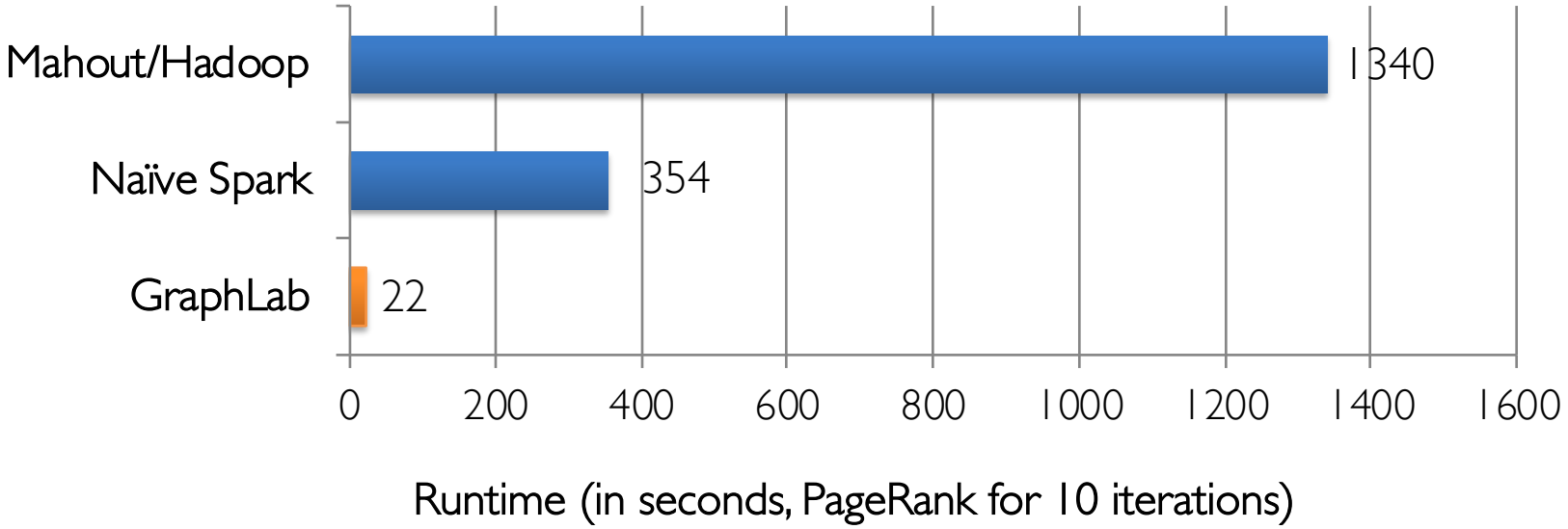
- Collaborative Filtering
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - Tensor Factorization
- Structured Prediction
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- Semi-supervised ML
  - Graph SSL
  - CoEM
- Community Detection
  - Triangle-Counting
  - K-core Decomposition
  - K-Truss
- Graph Analytics
  - PageRank
  - Personalized PageRank
  - Shortest Path
  - Graph Coloring
- Classification
  - Neural Networks

Graph-Parallel Systems

Expose specialized APIs to simplify graph programming.

Exploit graph structure to achieve orders-of-magnitude performance gains over more general data-parallel systems.

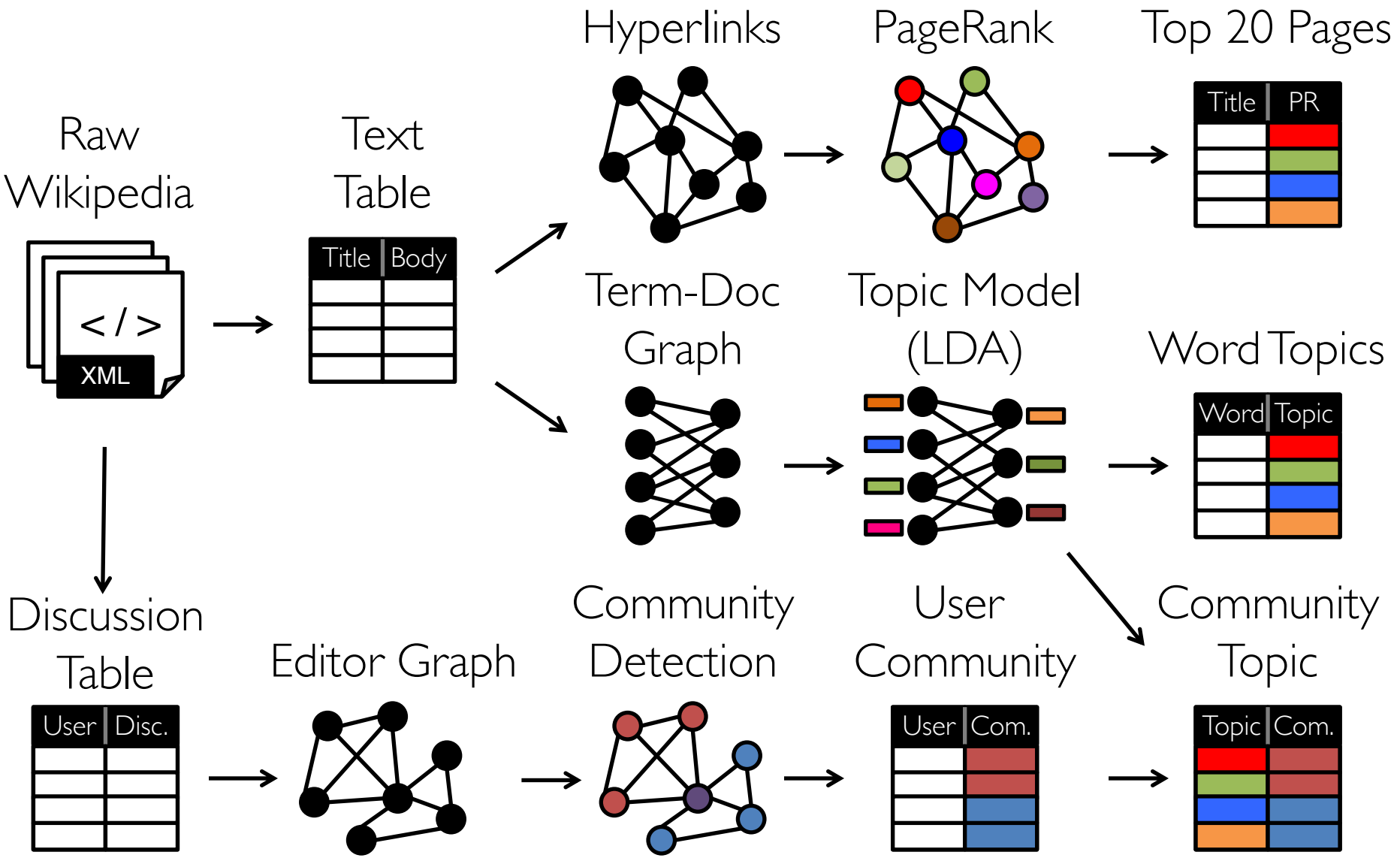
PageRank on the Live-Journal Graph



GraphLab is *60x faster* than Hadoop

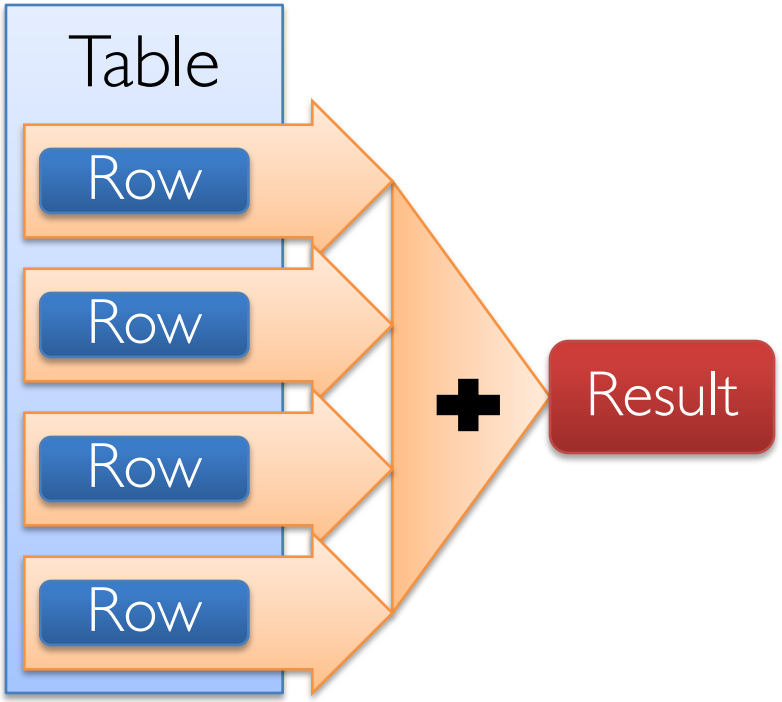
GraphLab is *16x faster* than Spark

Graphs are Central to Analytics

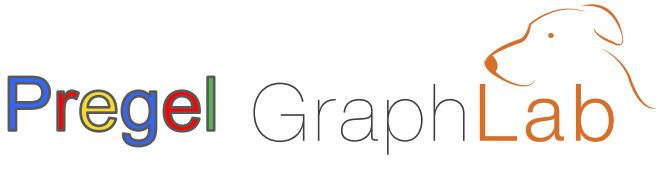


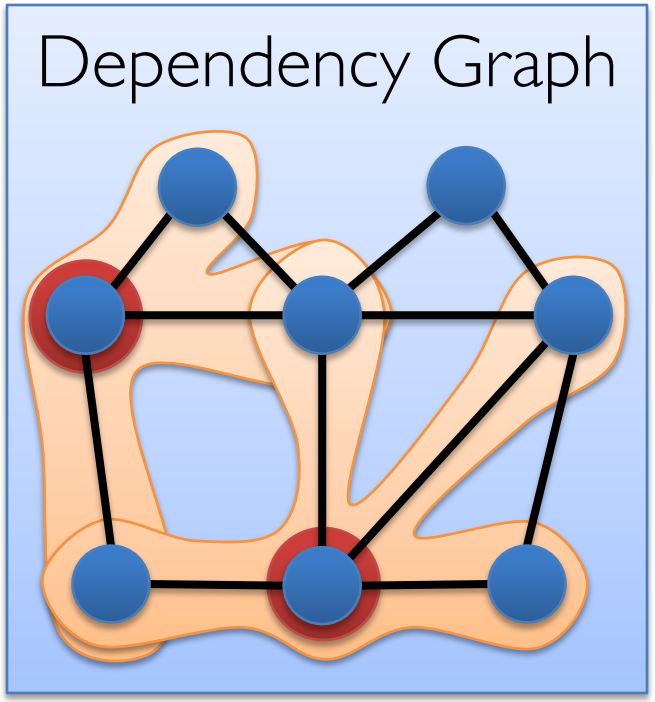
Separate Systems to Support Each View

Table View



Graph View

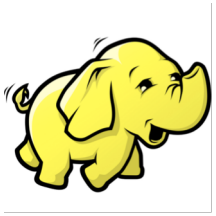




Having separate systems for each view is difficult to use and inefficient

Difficult to Program and Use

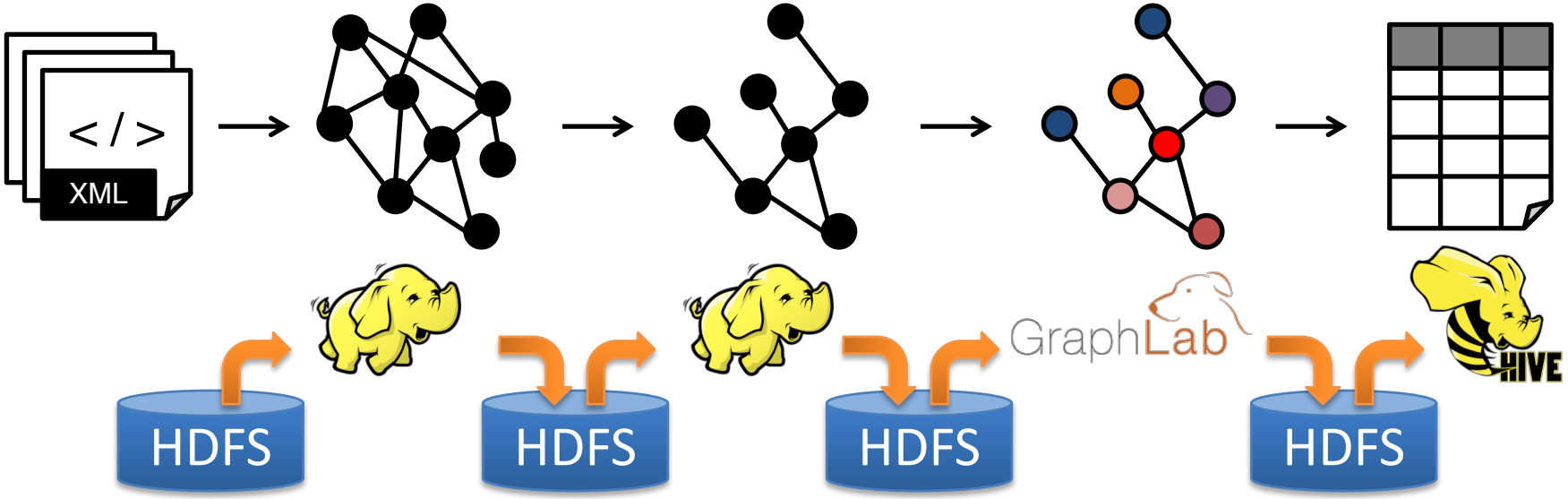
Users must *Learn*, *Deploy*, and *Manage* multiple systems




Leads to brittle and often complex interfaces

Inefficient

Extensive data movement and duplication across the network and file system

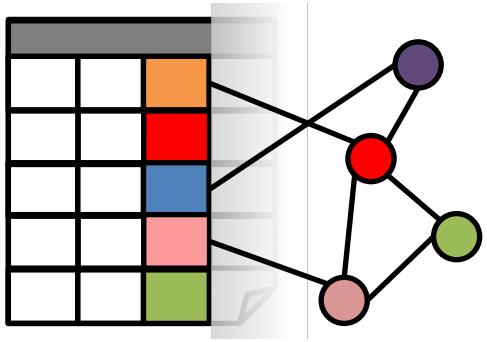


Limited reuse internal data-structures across stages

Solution: The GraphX Unified Approach

New API

Blurs the distinction between Tables and Graphs

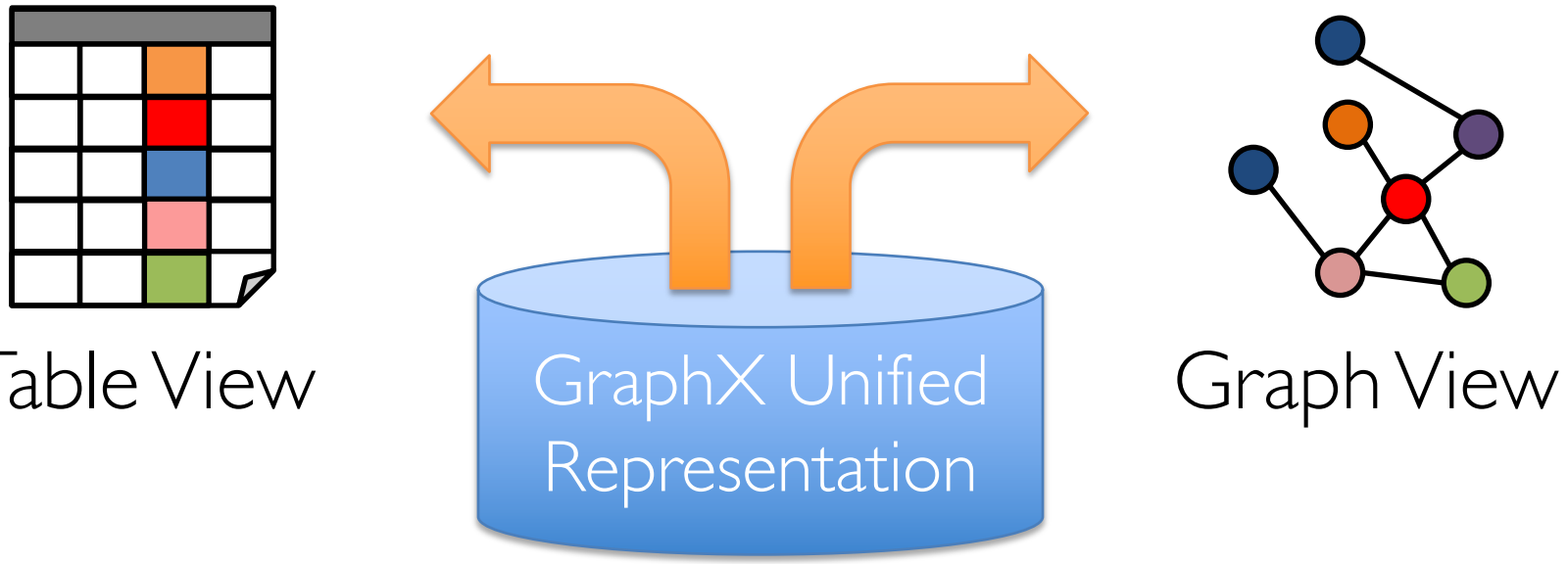


New System

Combines Data-Parallel Graph-Parallel Systems

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

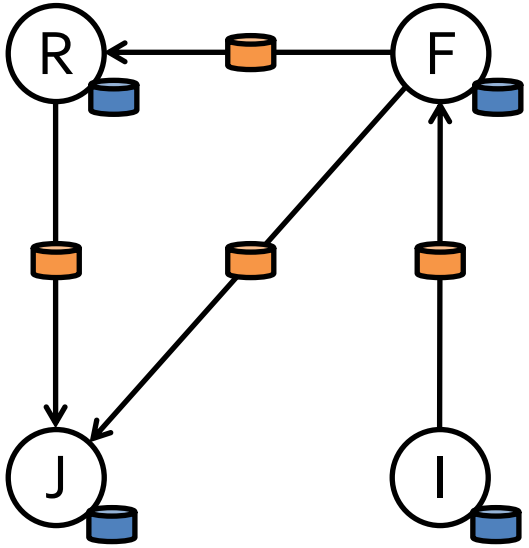
Tables and Graphs are composable *views* of the *same physical* data



Each view has its own operators that exploit the semantics of the view to achieve efficient execution

View a Graph as a Table

Property Graph



Vertex Property Table

Id	Property (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Edge Property Table

SrcId	DstId	Property (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Table Operators

Table (RDD) operators are inherited from Spark:

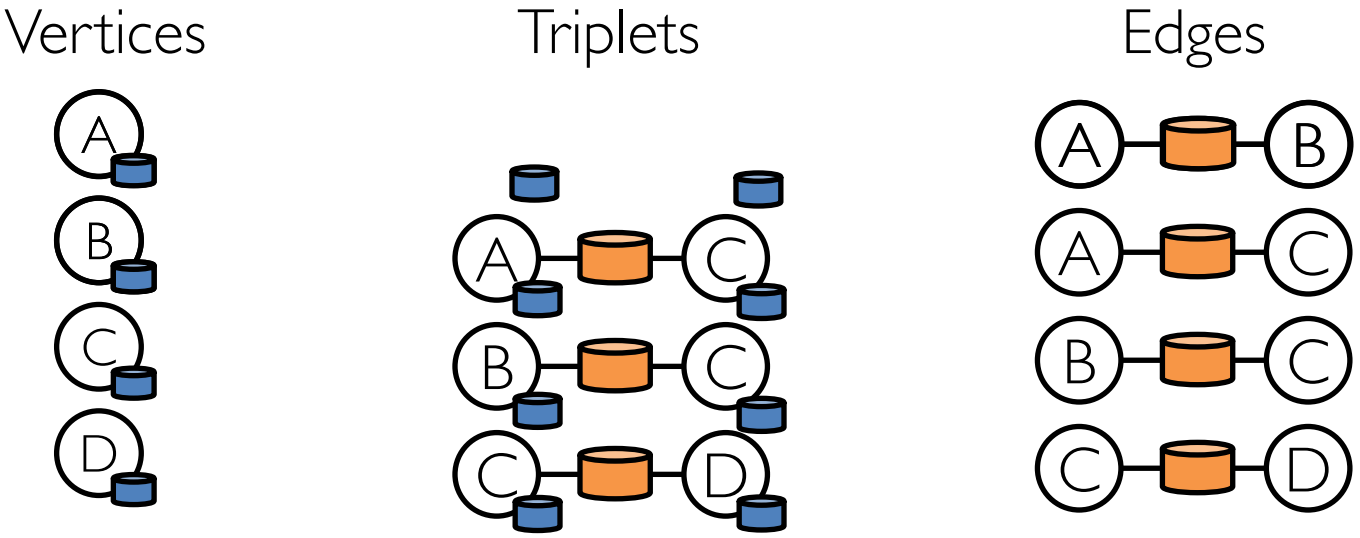
`map`	`reduce`	`sample`
`filter`	`count`	`take`
`groupBy`	`fold`	`first`
`sort`	`reduceByKey`	`partitionBy`
`union`	`groupByKey`	`mapWith`
`join`	`cogroup`	`pipe`
`leftOuterJoin`	`cross`	`save`
`rightOuterJoin`	`zip`	`...`

Graph Operators

```
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) => Boolean,
               pE: Edge[V,E] => Boolean): Graph[V,E]
  def mapV(m: (Id, V) => T ): Graph[T,E]
  def mapE(m: Edge[V,E] => 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]) => List[(Id, T)],
                 reduceF: (T, T) => T): Graph[T, E]
}
```

Triplets Join Vertices and Edges

The *triplets* operator joins vertices and edges:

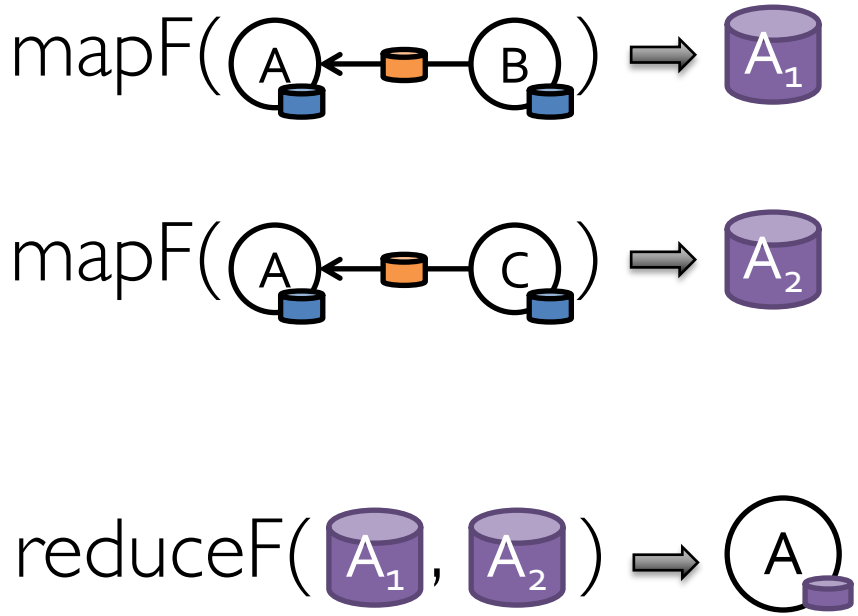


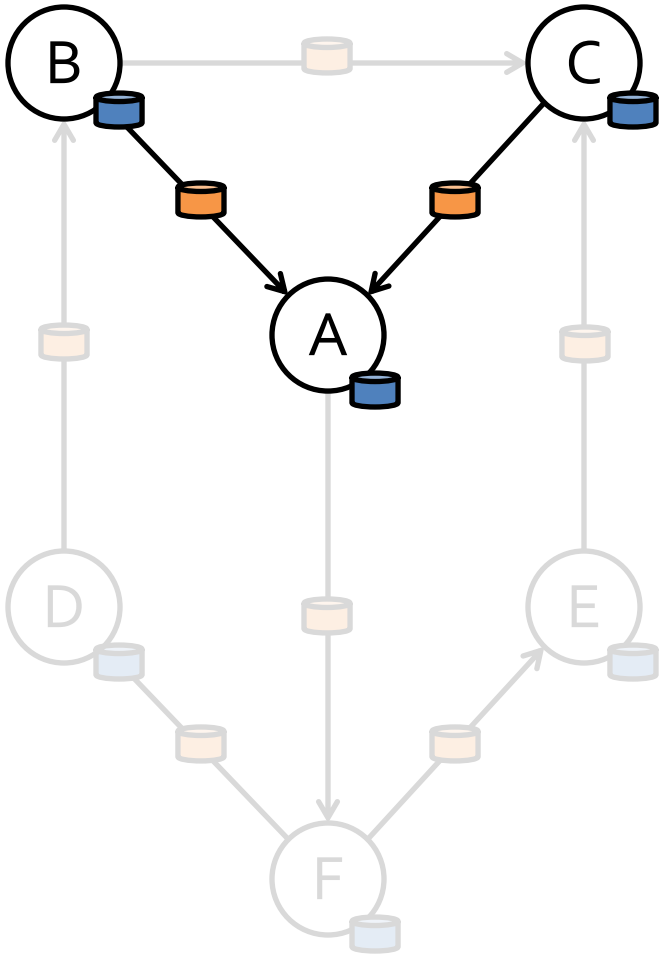
The *mrTriplets* operator sums adjacent triplets.

SELECT t.dstId, *reduceUDF*(*mapUDF*(t)) **AS** sum
FROM triplets **AS** t **GROUPBY** t.dstId

Map Reduce Triplets

Map-Reduce for each vertex

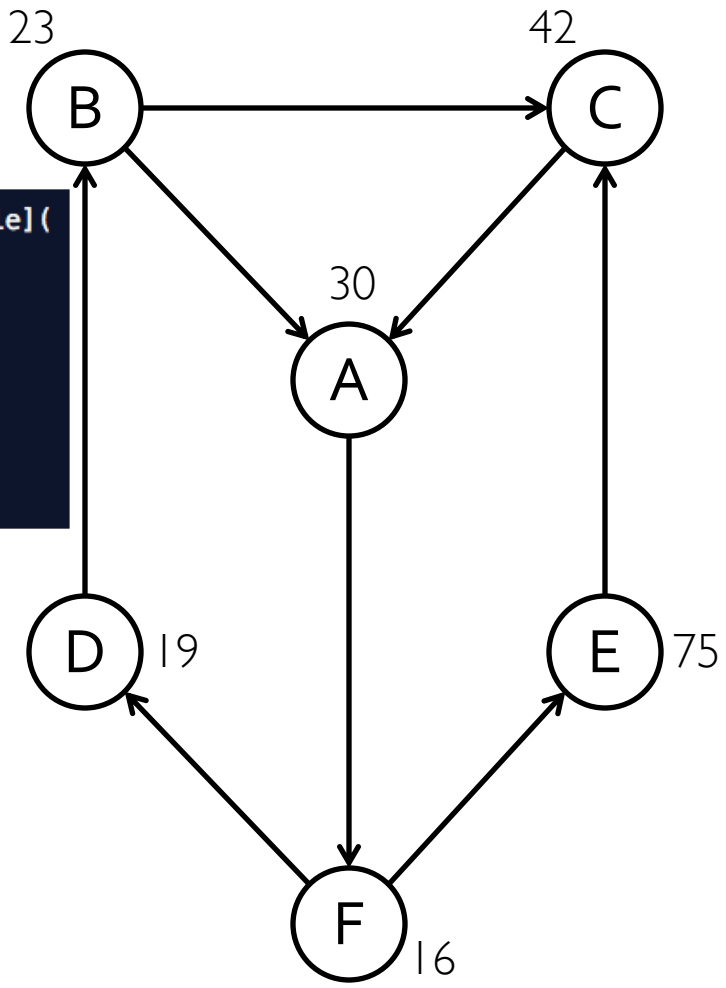




Example: Oldest Follower

What is the age of the oldest follower for each user?

```
val oldestFollowers: VertexRDD[Double] = graph.aggregateMessages[Double](
   triplet => { // Map Function
       // Send message to destination vertex containing age
       triplet.sendToDst(triplet.srcAttr._2)
   },
   // Find max age
   (a, b) => Math.max(a,b) // Reduce Function
 )
```



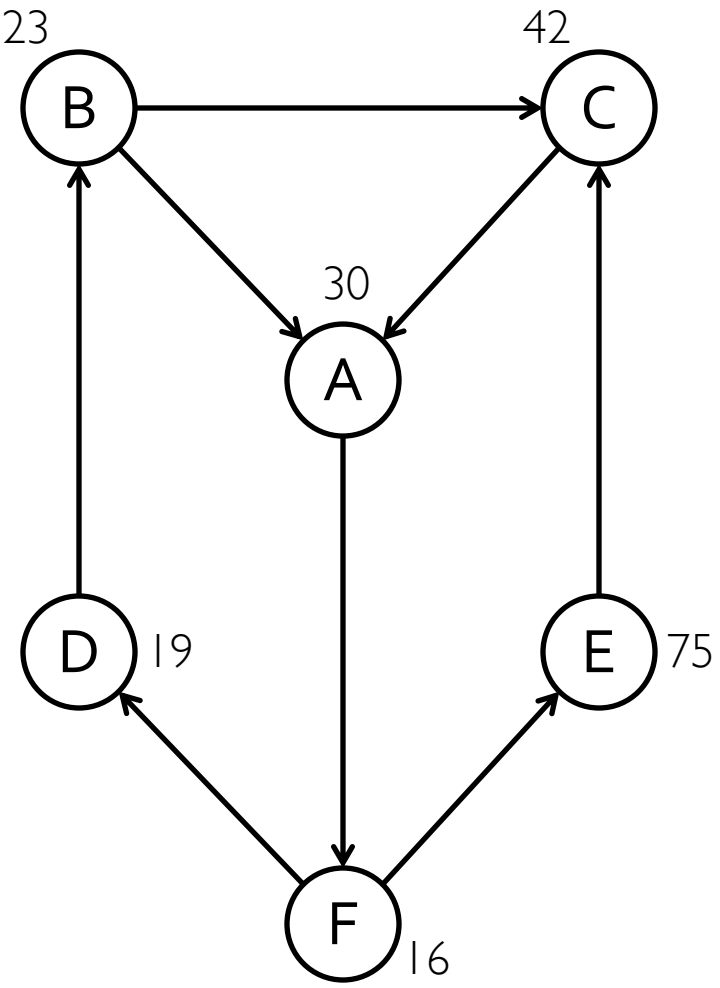
Example: Oldest Follower Code

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

val vertexArray = Array(
  (1L, ("A", 30)),
  (2L, ("B", 23)),
  (3L, ("C", 42)),
  (4L, ("D", 19)),
  (5L, ("E", 75)),
  (6L, ("F", 16))
  )
val edgeArray = Array(
  Edge(2L, 1L, 1),
  Edge(3L, 1L, 1),
  Edge(3L, 2L, 1),
  Edge(4L, 2L, 1),
  Edge(6L, 4L, 1),
  Edge(1L, 6L, 1),
  Edge(6L, 5L, 1),
  Edge(5L, 3L, 1)
  )

 val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
 val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
 val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)

 val oldestFollowers: VertexRDD[Double] = graph.aggregateMessages[Double](
    triplet => { // Map Function
        // Send message to destination vertex containing age
        triplet.sendToDst(triplet.srcAttr._2)
    },
    // Find max age
    (a, b) => Math.max(a,b) // Reduce Function
  )
```



We express the Pregel and GraphLab abstractions using the GraphX operators in less than 50 lines of code!

By composing these operators we can construct entire graph-analytics pipelines.