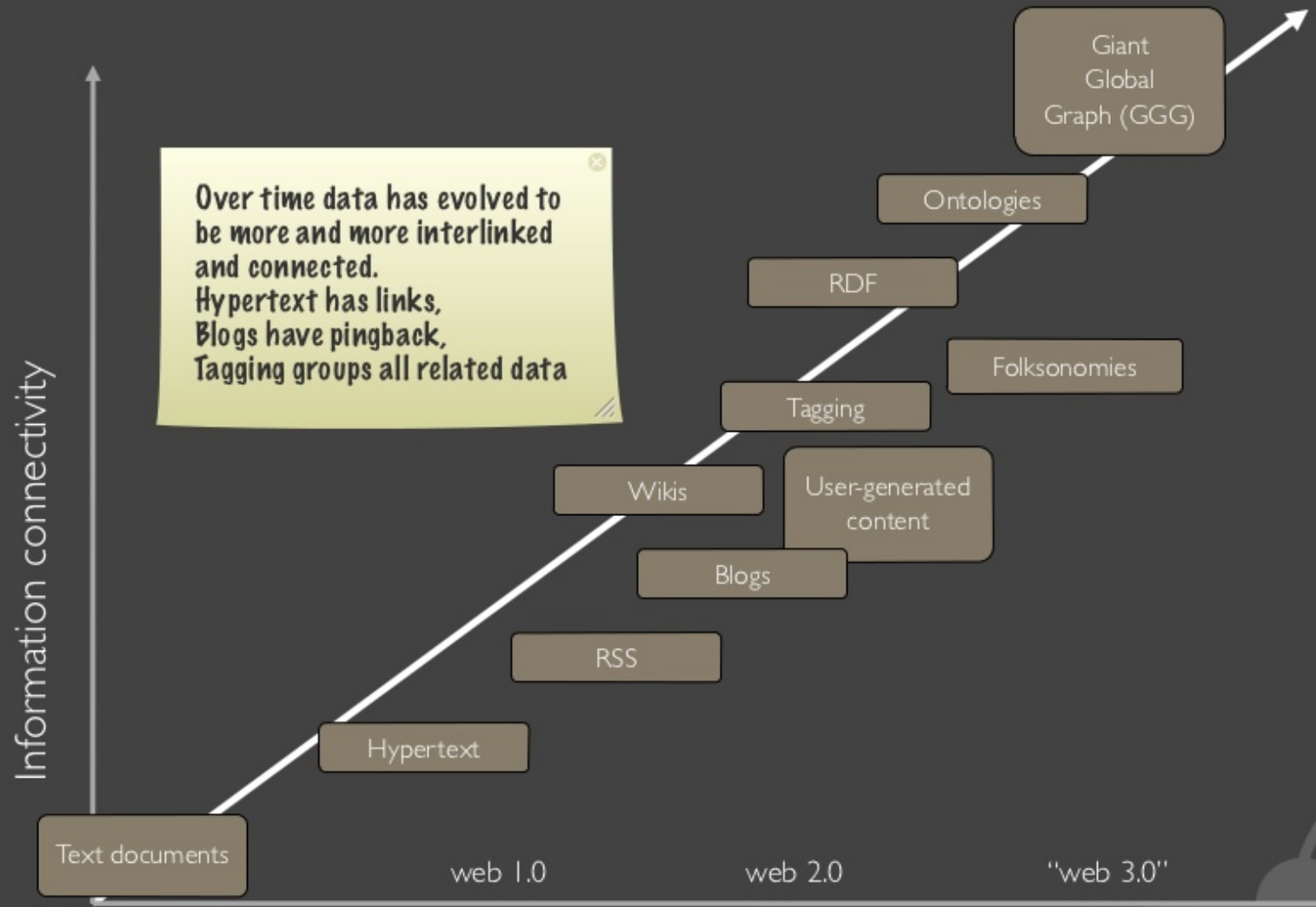
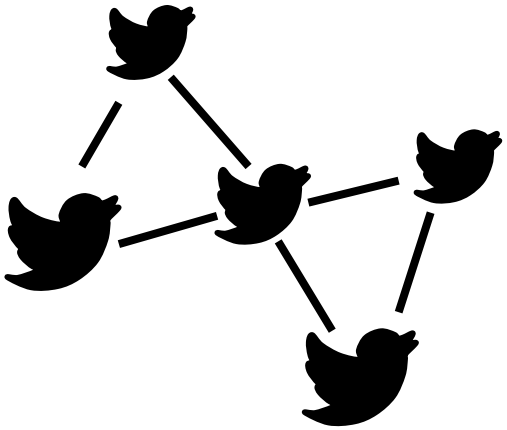
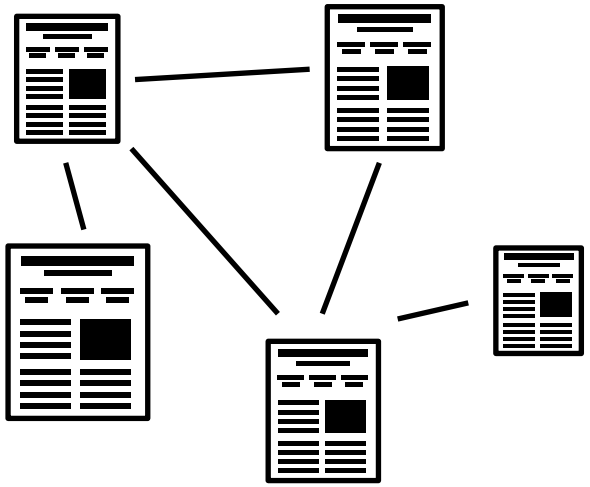
GraphDB: Neo4j

Trend 2: Connectedness

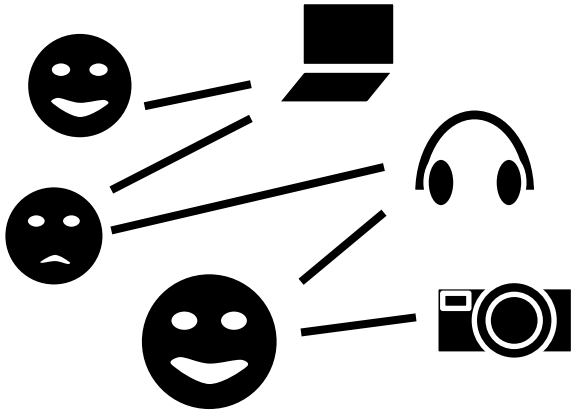


Over time data has evolved to be more and more interlinked and connected.
Hypertext has links,
Blogs have pingback,
Tagging groups all related data

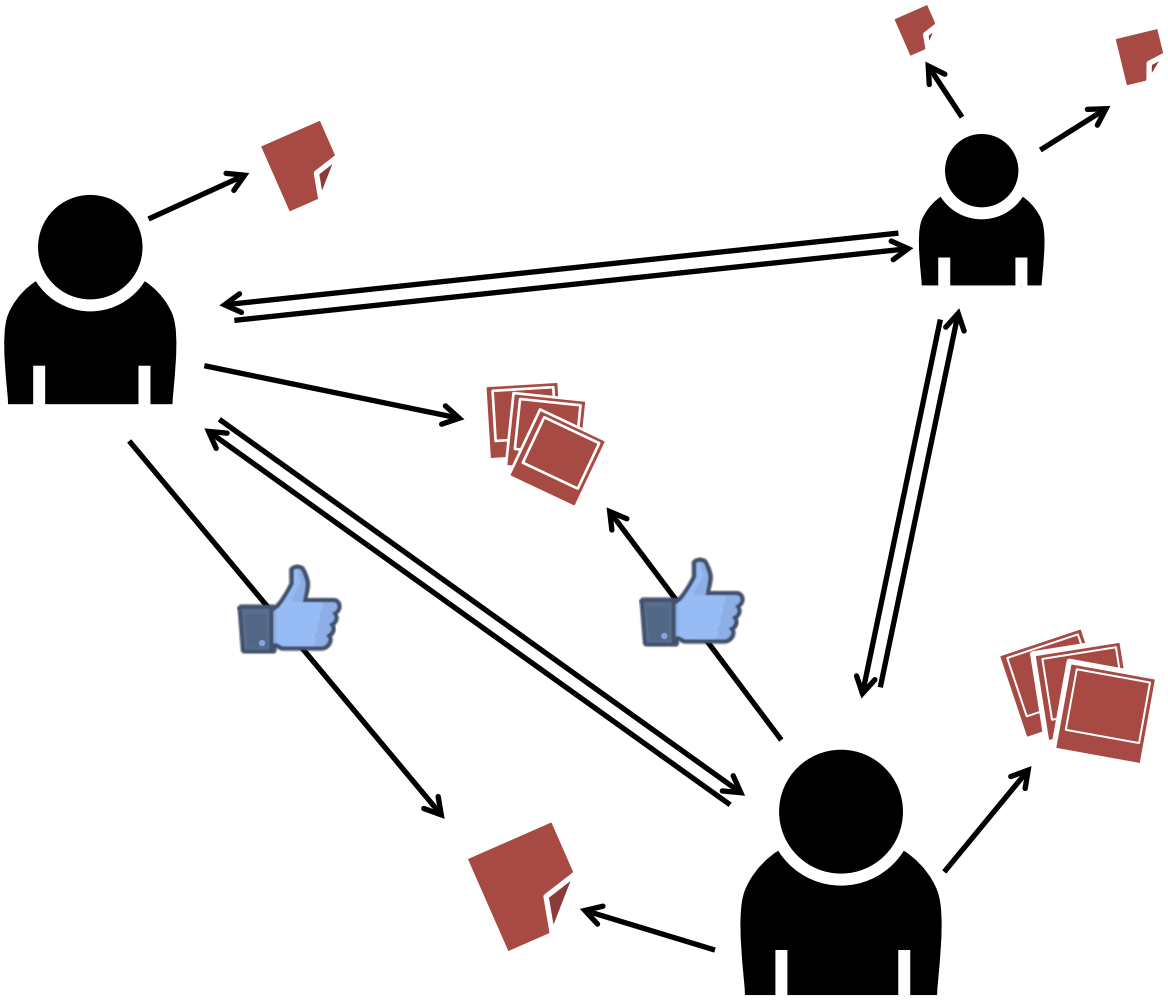




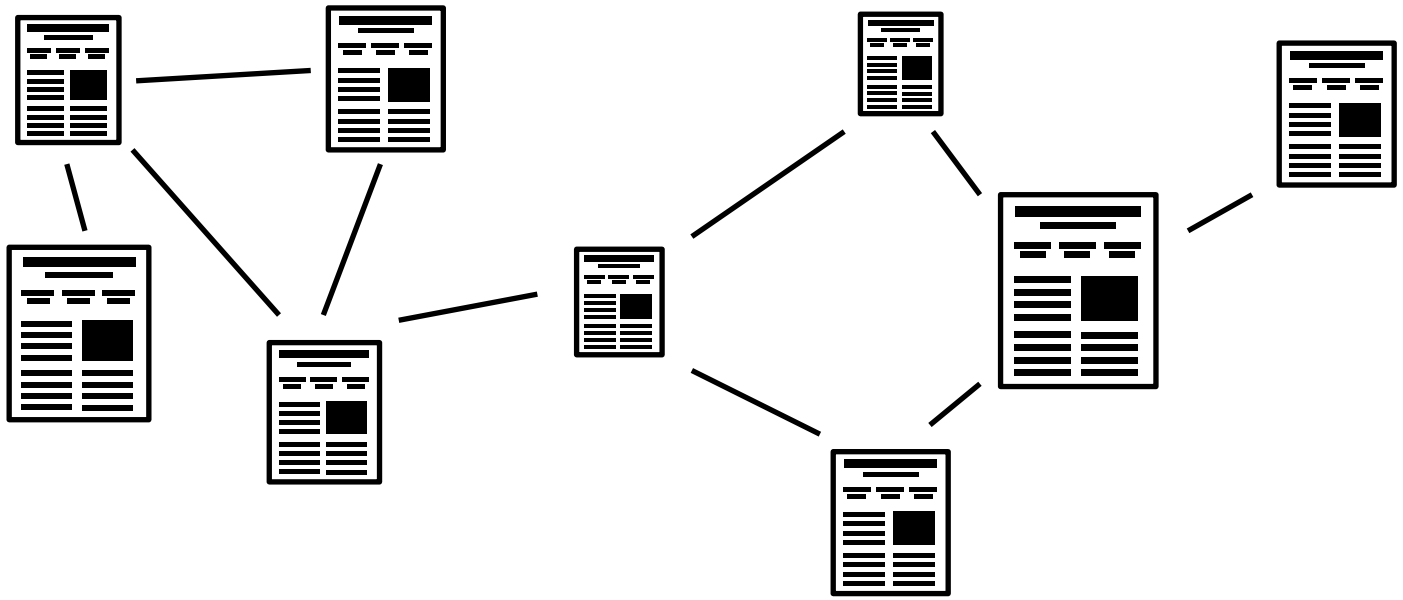
Graphs are Everywhere



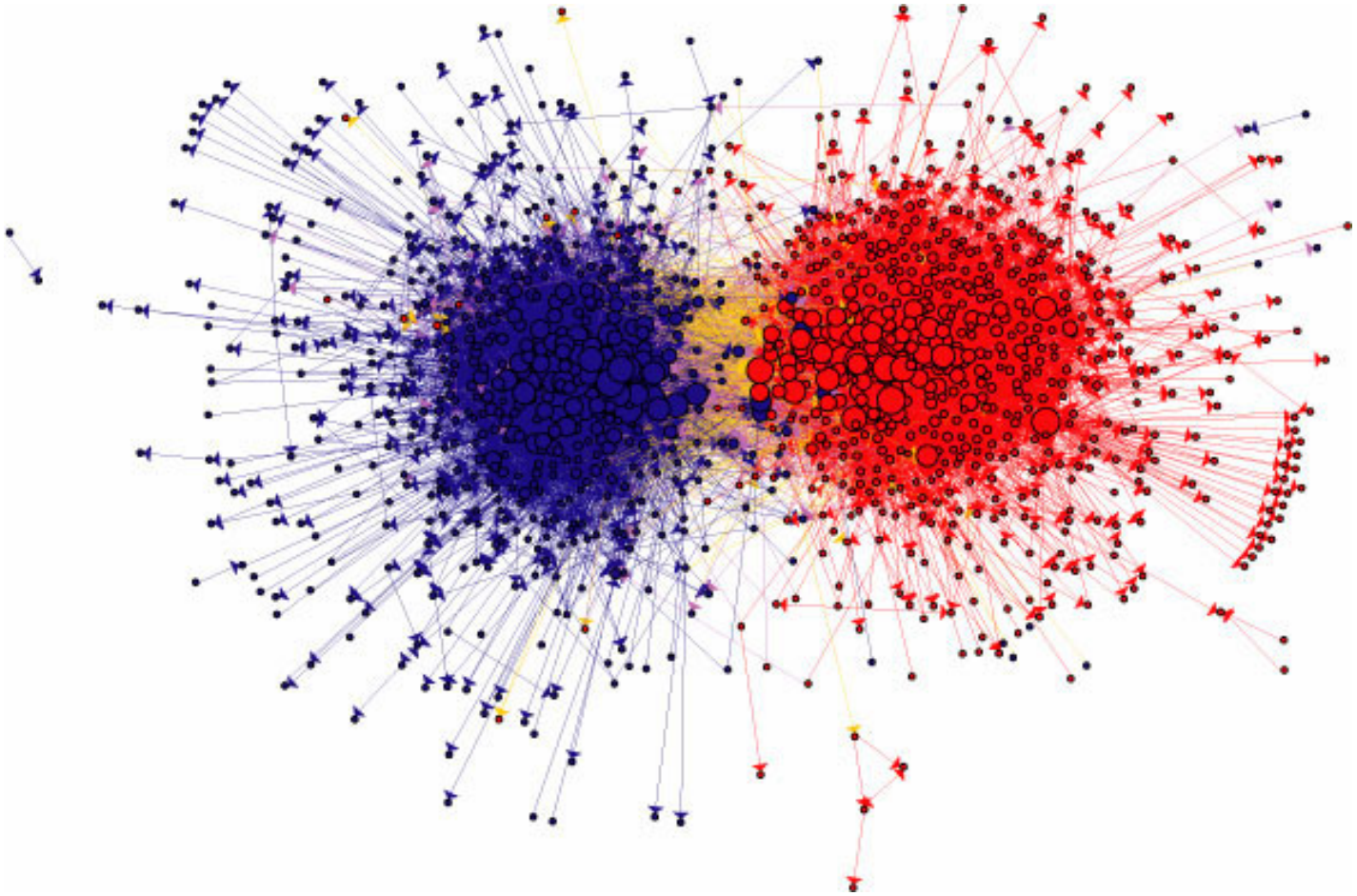
Social Networks



Web Graphs

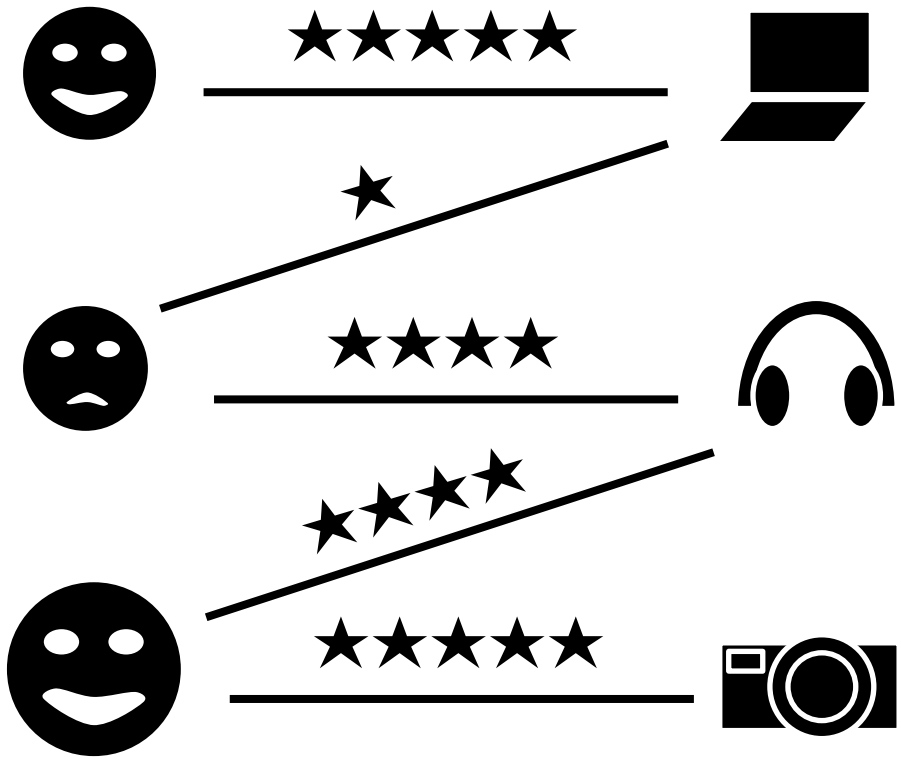


Web Graphs

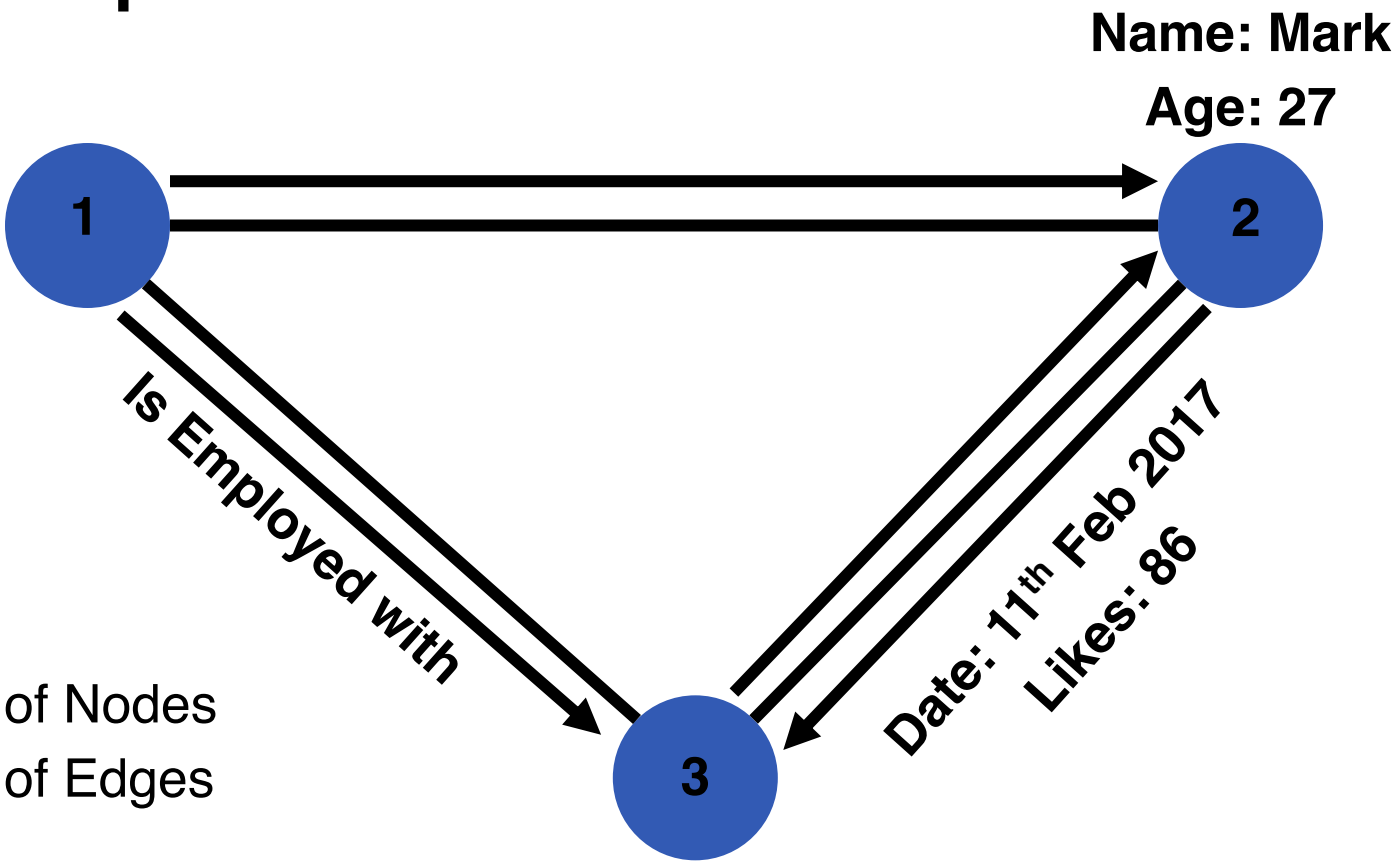


"The political blogosphere and the 2004 U.S. election: divided they blog." Adamic and Glance, 2005.

User-Item Graphs



Graph Representation



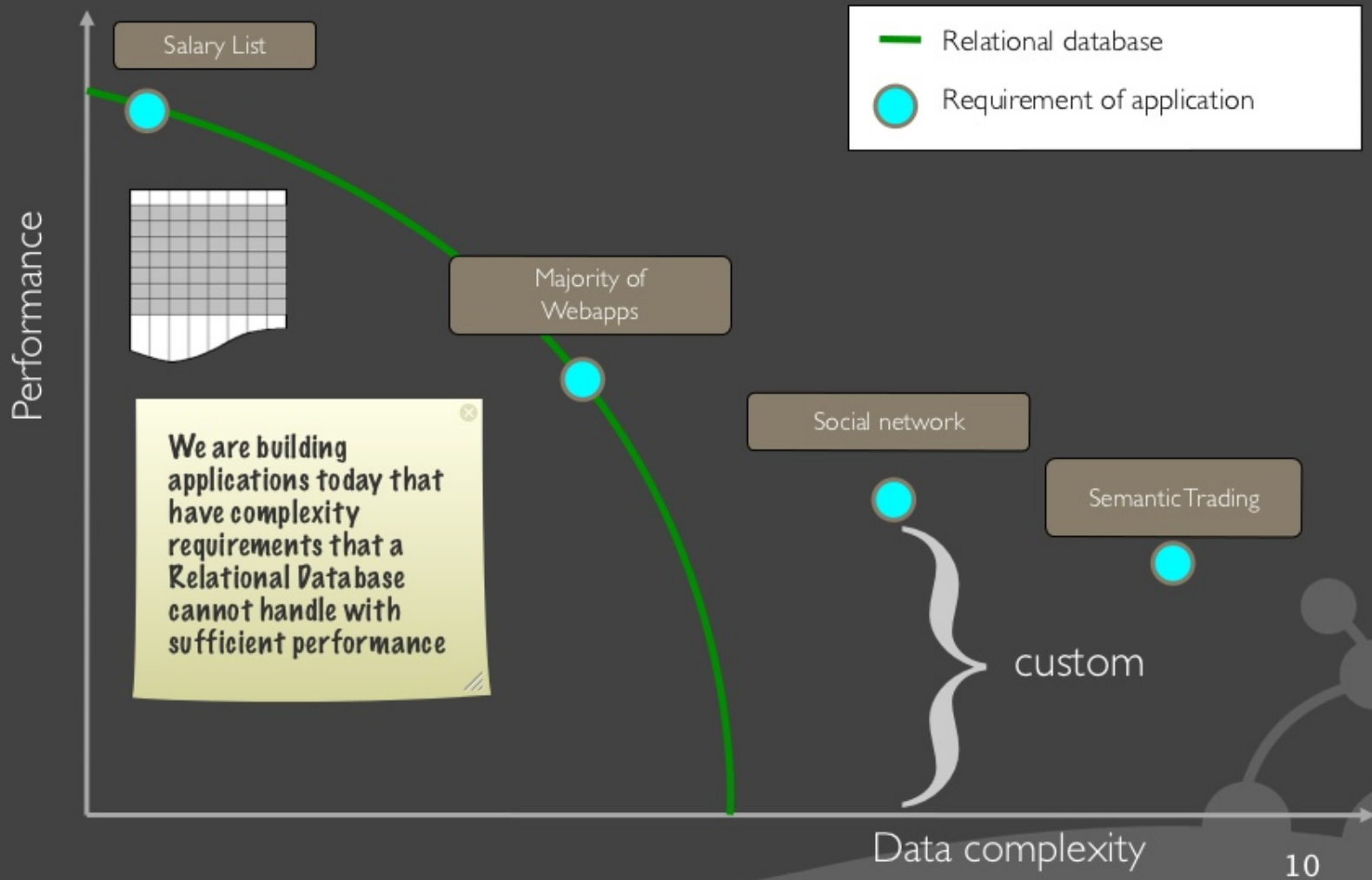
Set of Nodes
Set of Edges

Edges can be directed or undirected

Nodes can have a label and multiple attributes

Edges can have a label and multiple attributes

RDBMS performance



Neo4j

Introduction

- Intended not so much to store information about things
- But to tie them together and record their connections with each other
- Stores data as a graph
- Can store highly variable data
- Small enough to be embedded in any application
- But able to scale up easily

Neo4j

- GraphDB
- Non-relational (NoSQL)
- Schema free
- Open Source Software

Graphs are all around us

	A	B	C	D	...
1	17	3.14	3	=A1 * B1 / C1	
2	42	10.11	14	=A2 * B2 / C2	
3	316	6.66	1	=A3 * B3 / C3	
4	32	9.11	592	=A4 * B4 / C4	
5				=SUM(D2:D5)	
...					

With data dependencies the spread sheet turns out to be a graph.

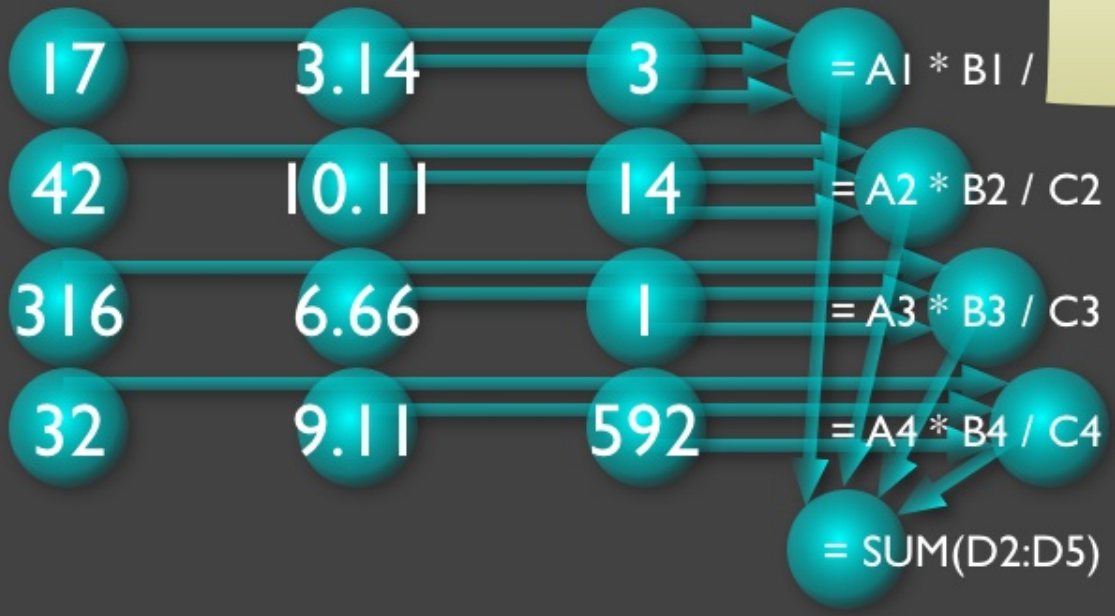
Graphs are all around us

	A	B	C	D	...
1	17	3.14	3	=A1 * B1 / C1	
2	42	10.11	14	=A2 * B2 / C2	
3	316	6.66	1	=A3 * B3 / C3	
4	32	9.11	592	=A4 * B4 / C4	
5				= SUM(D2:D5)	
...					

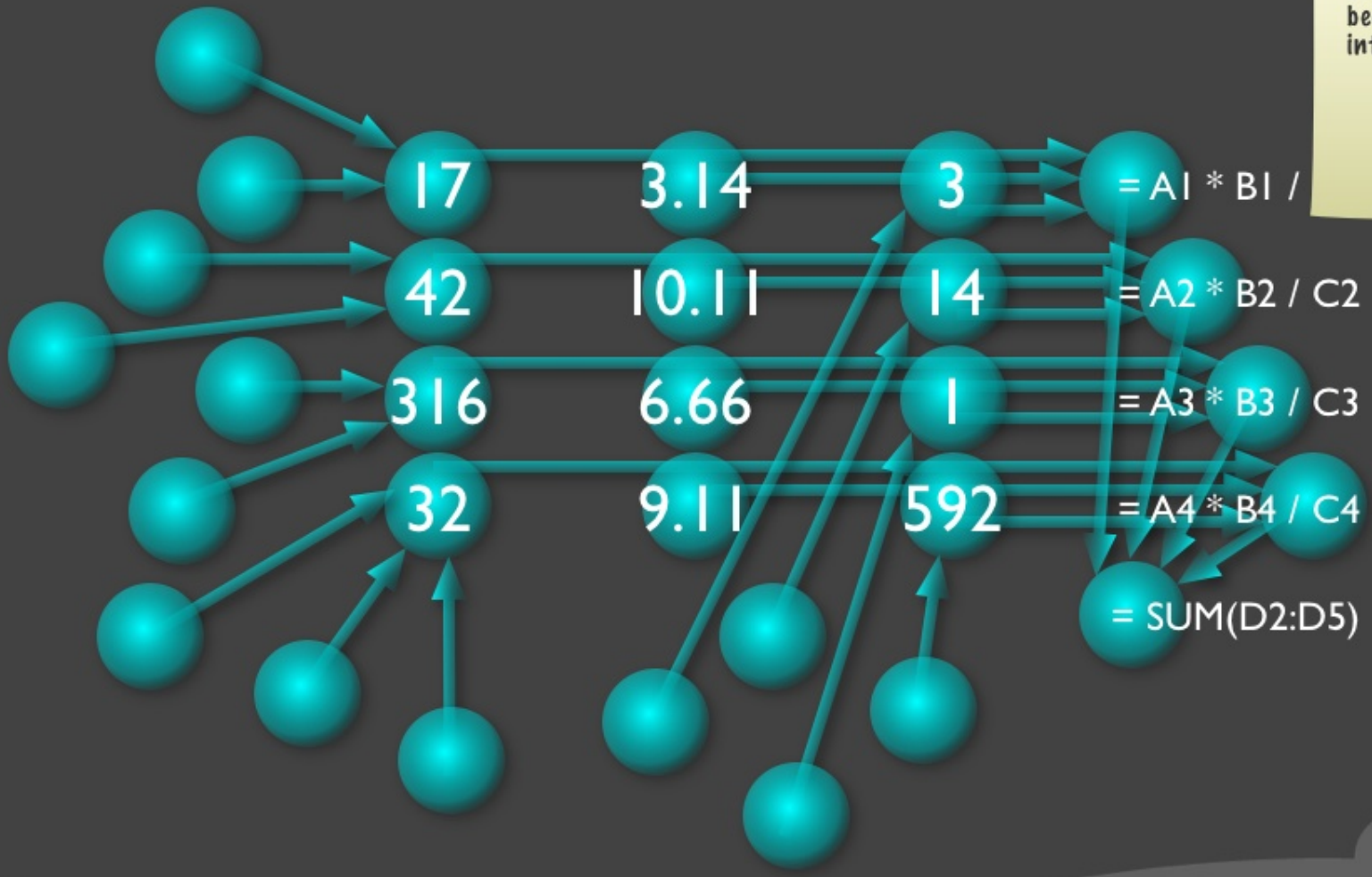
With data dependencies the spread sheet turns out to be a graph.

Graphs are all around us

If we add external data sources the problem becomes even more interesting...



Graphs are all around us



If we add external data sources the problem becomes even more interesting...

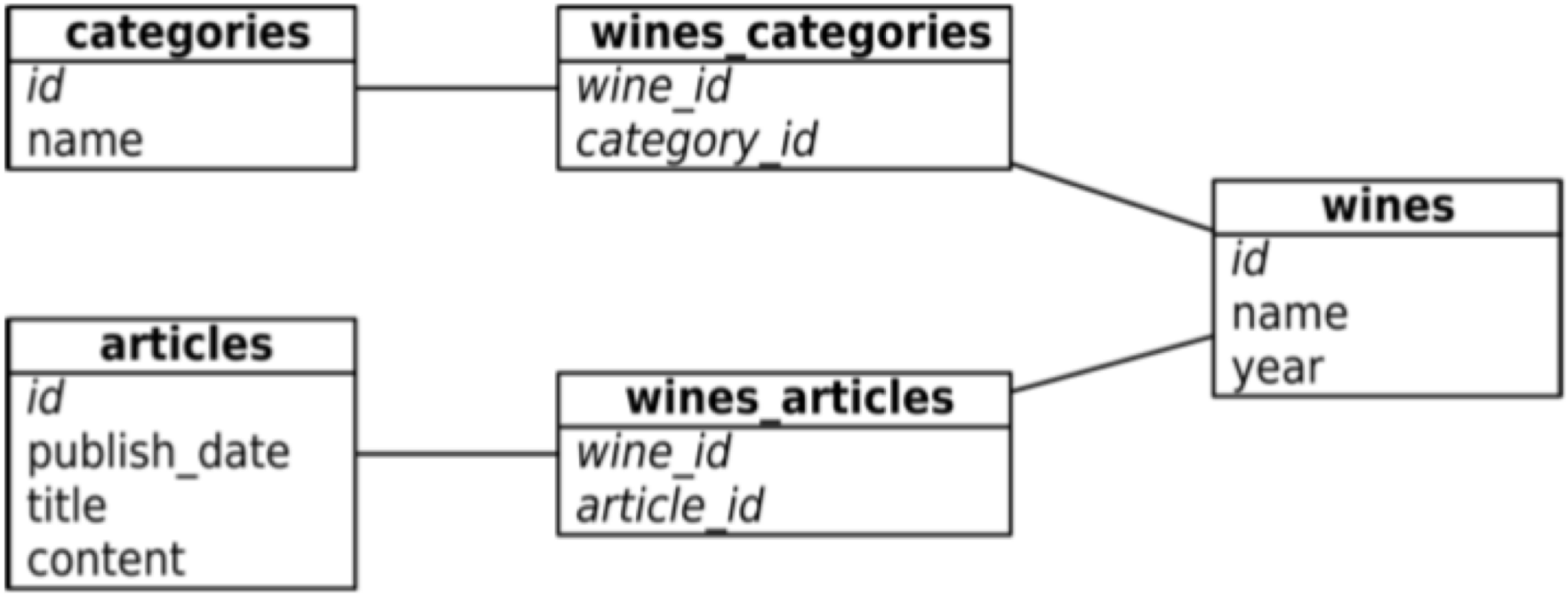
Wine suggestion engine

Wines are categorized by:

- Different varieties
- Regions
- Wineries
- Vintages
- Designations
- …

We need to keep track of other things, like articles describing those wines, and to enable users to track their favorite wines

Wine suggestion engine: Relational Model



Wine suggestion engine: Graph Model

Query Languages

- Traversal APIs
- SPARQL (“SQL for linked data”)
- Gremlin
- Cypher

Example

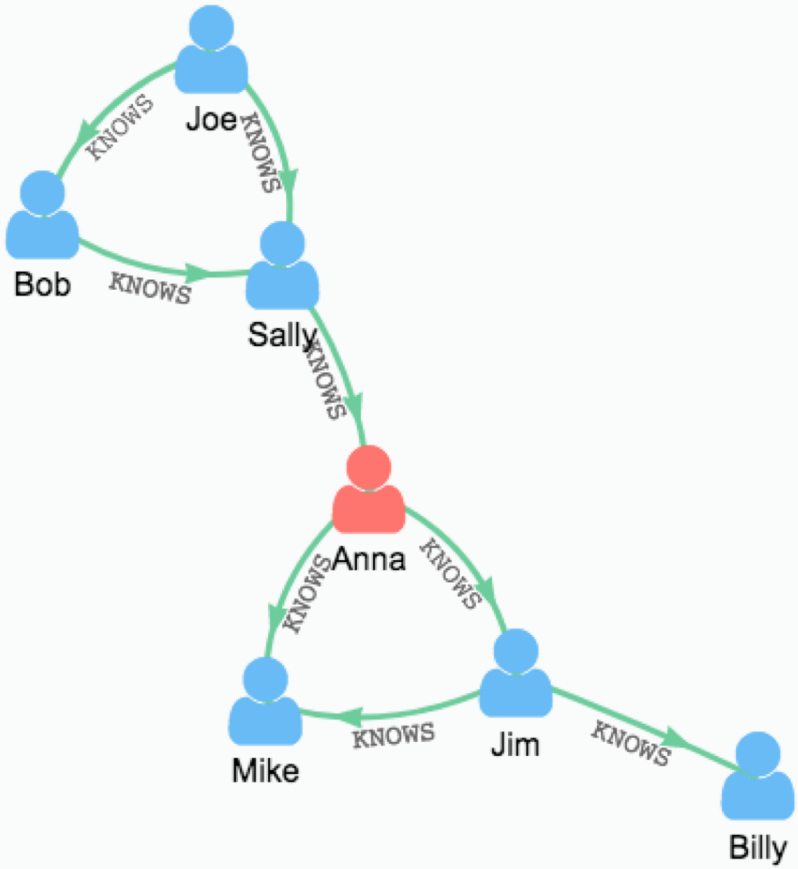
Friends of Friends | Common Friends | Connecting Paths

Friends of Friends

Find all of Joe's second-degree friends

```
MATCH
  (person:Person)-[:KNOWS]-(friend:Person)-[:KNOWS]-
  (foaf:Person)
WHERE
  person.name = "Joe"
  AND NOT (person)-[:KNOWS]-(foaf)
RETURN
  foaf
```

Joe knows Sally, and Sally knows Anna. Bob is excluded from the result because, in addition to being a 2nd-degree friend through Sally, he's also a first-degree friend.



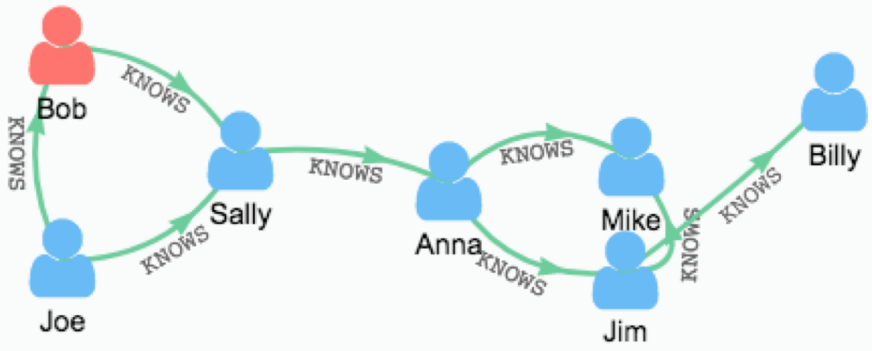
dbTrento

Common Friends

Find friends in common between Joe and Sally

```
MATCH
  (user:Person)-[:KNOWS]-(friend)-
    [:KNOWS]-(foaf:Person)
WHERE
  user.name = "Joe" AND foaf.name = "Sally"
RETURN
  friend.name AS friend
```

Joe and Sally both know Bob.

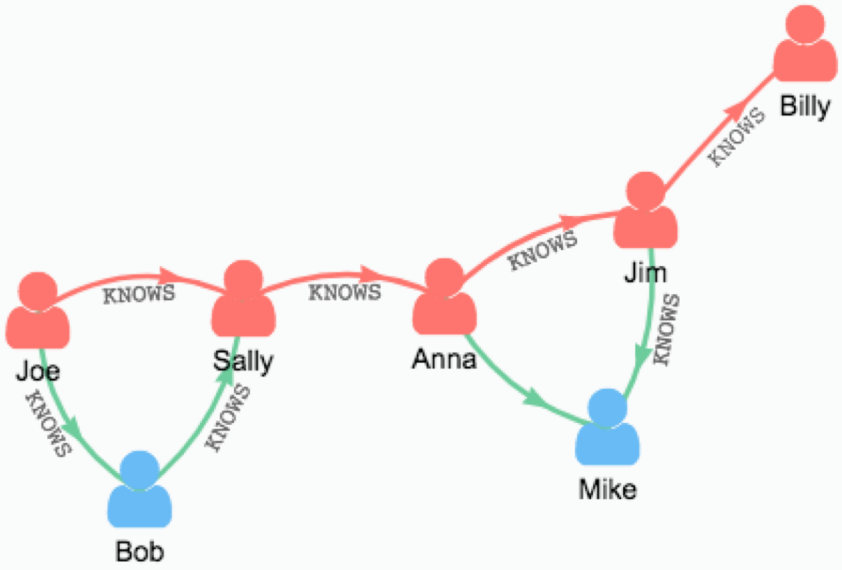


Connecting Paths

Find all paths, up to 6 degrees, between Joe and Billy

```
MATCH
  path = shortestPath(
    (p1:Person)-[:KNOWS*..6]-(p2:Person)
  )
WHERE
  p1.name = "Joe" AND p2.name = "Billy"
RETURN
  path
```

There's only one shortest path between Joe and Billy in this dataset, so that path is returned.



dbTrento

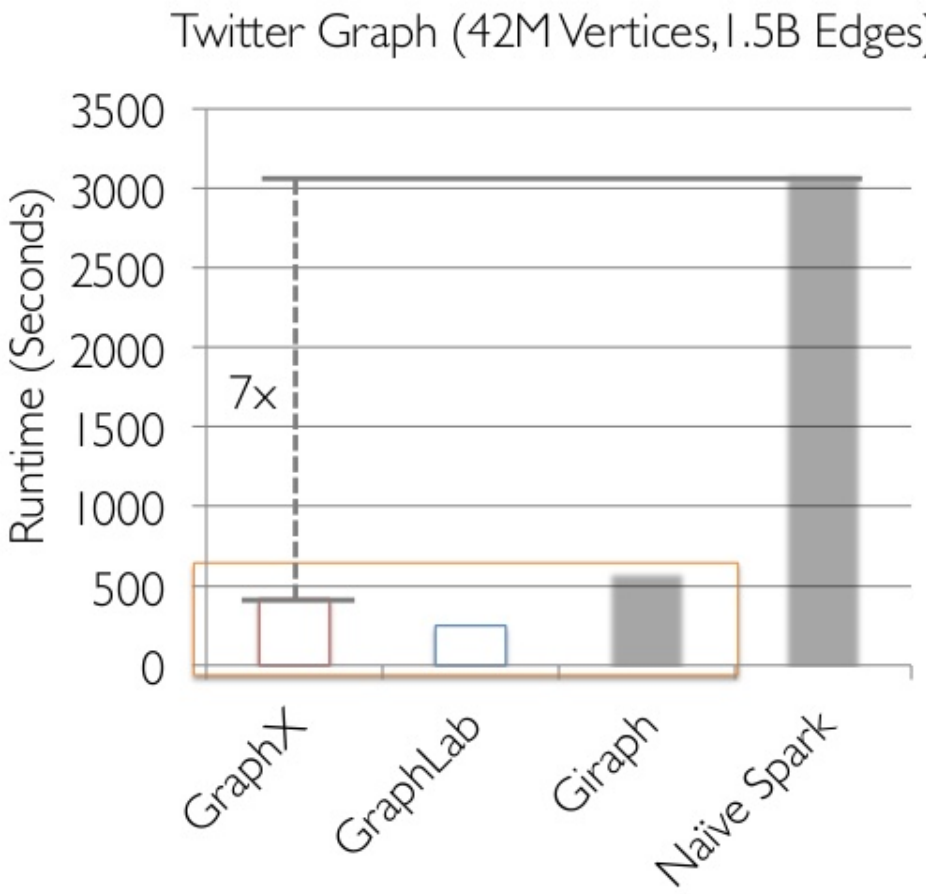
Spark GraphX

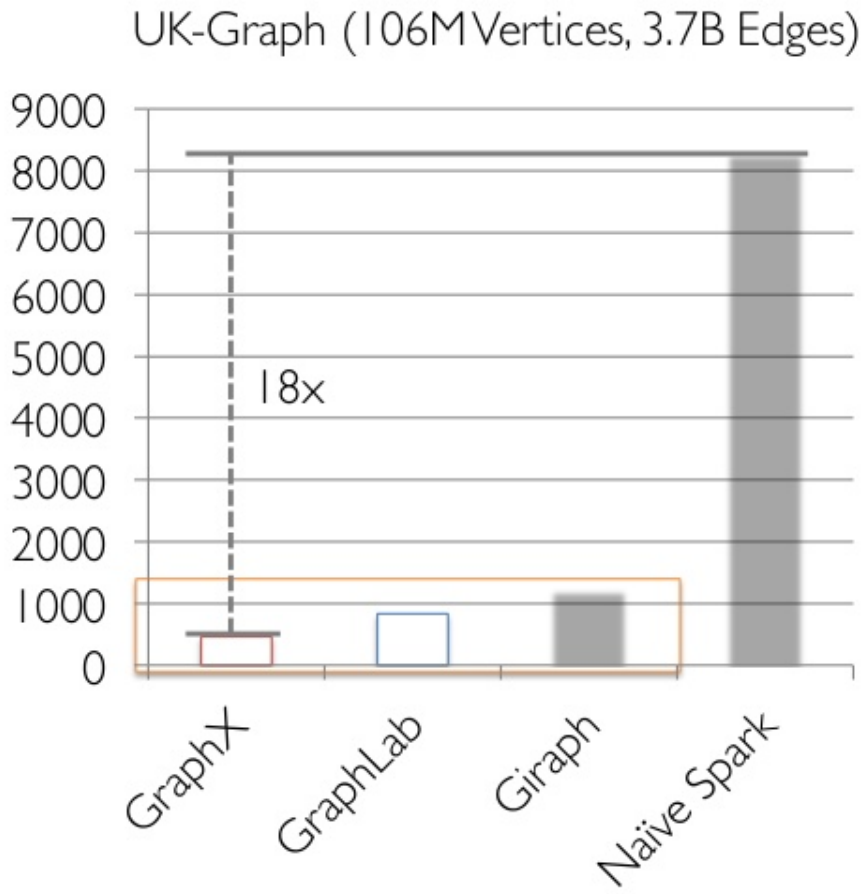
Spark GraphX

- Graph Processing System
- Graph composed by 2 RDDs, one for vertices (nodes), and one for relationships (edges)
- Graphs are immutable
- Perform a transformation to update it
- APIs available only in Scala

PageRank Benchmark

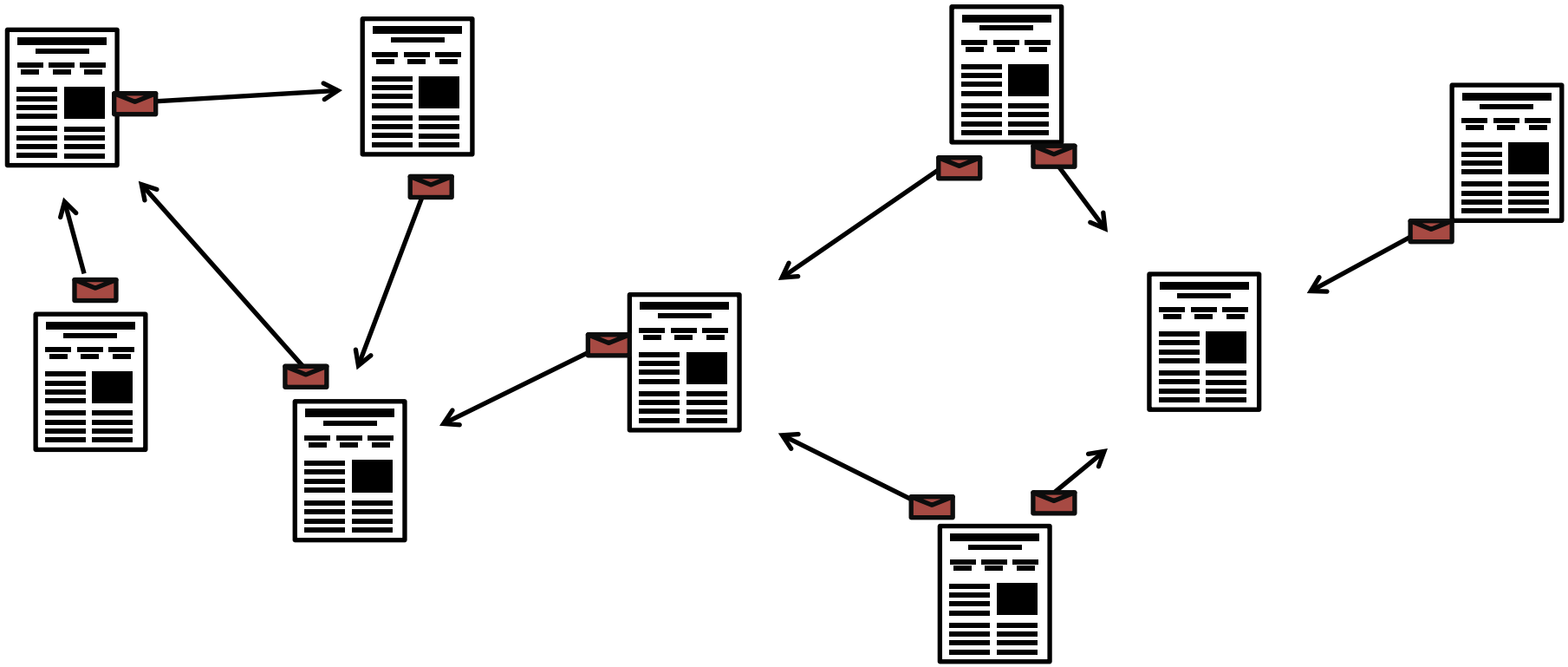
EC2 Cluster of 16 x m2.4xLarge (8 cores) + 1GigE



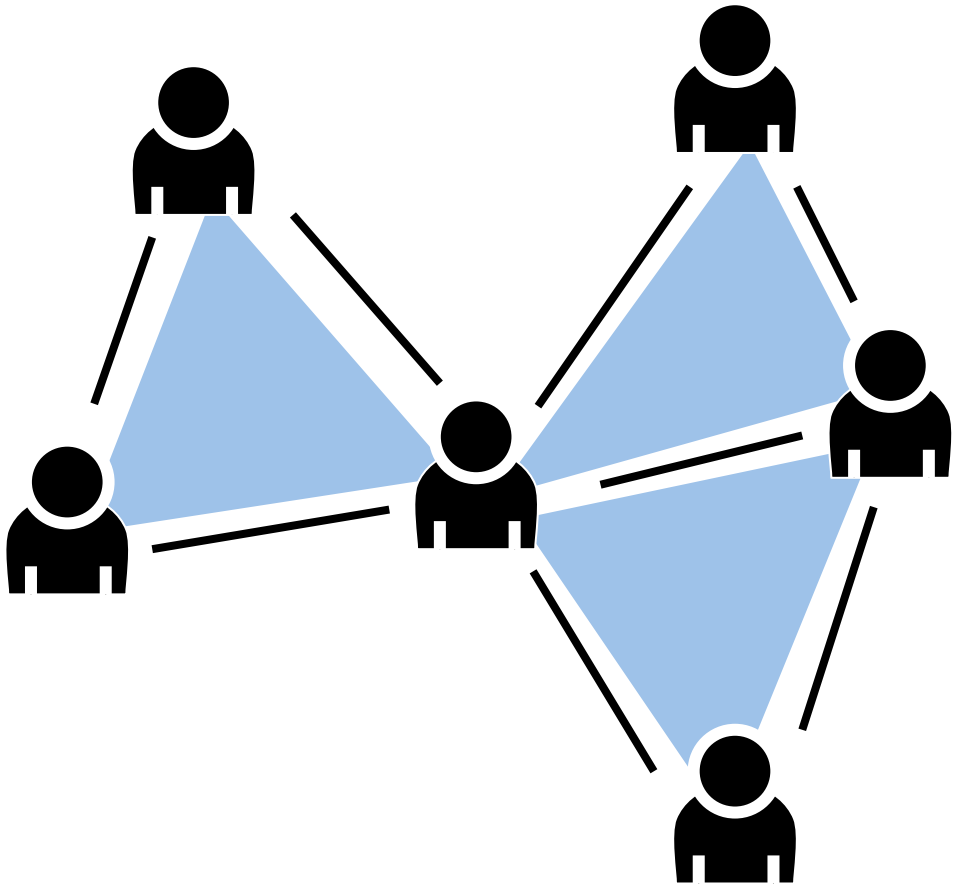


GraphX performs comparably to
state-of-the-art graph processing systems.

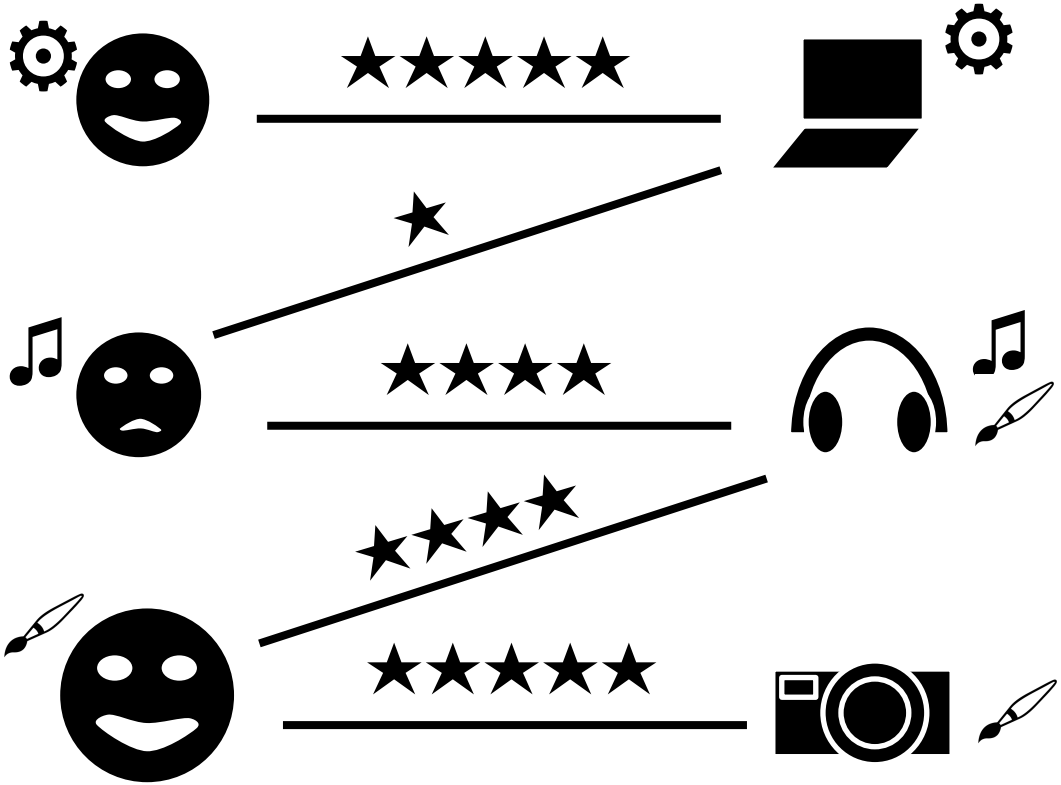
PageRank



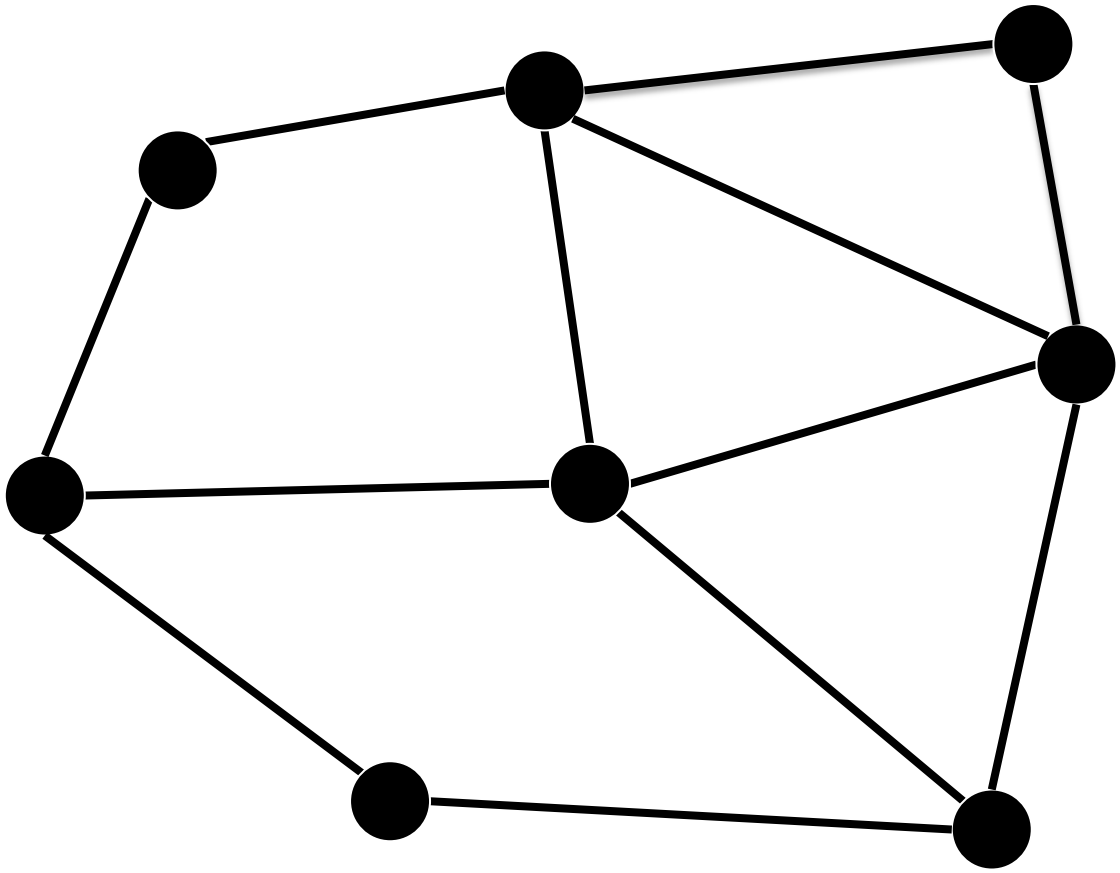
Triangle Counting



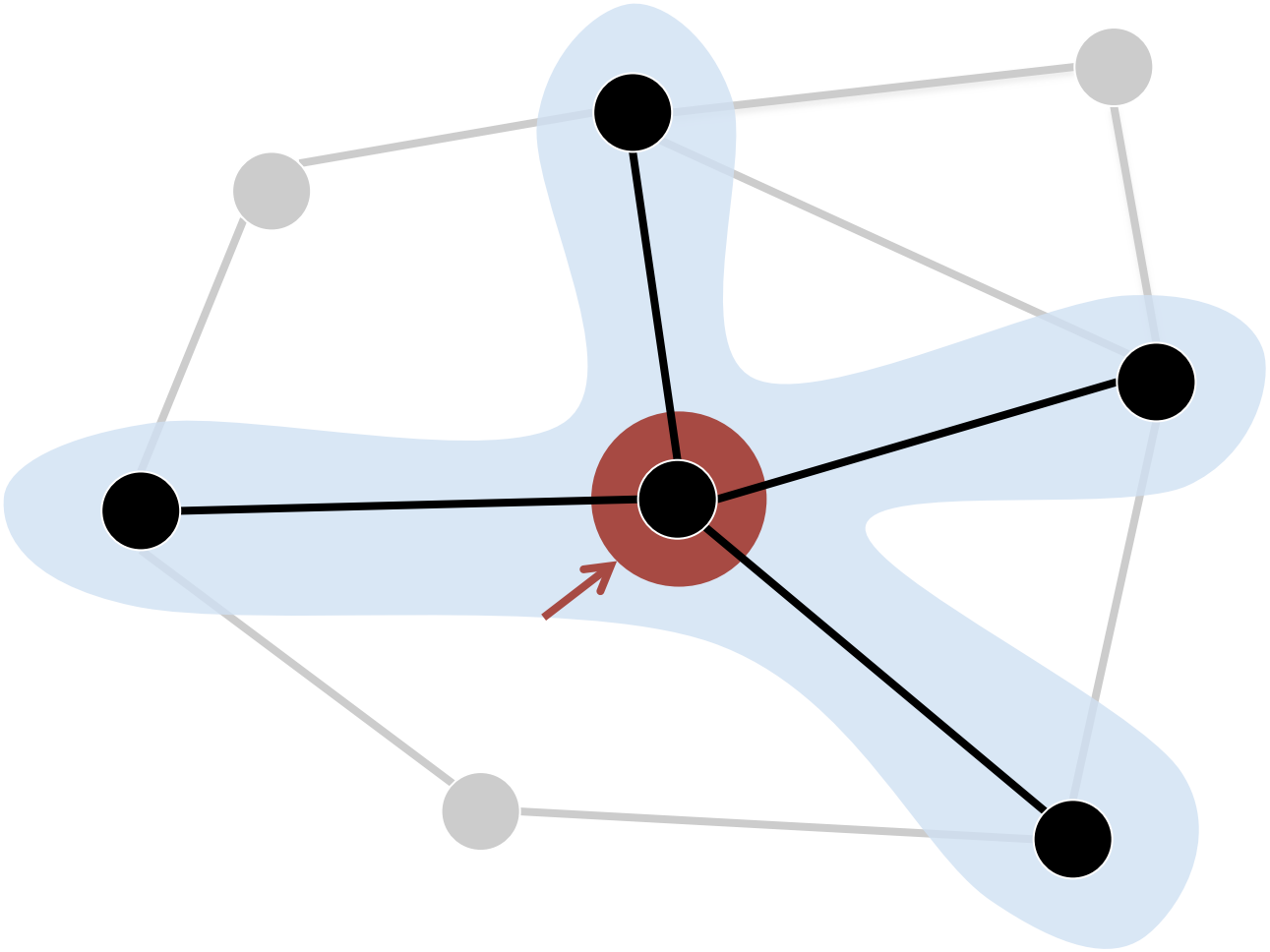
Collaborative Filtering



The Graph-Parallel Pattern



The Graph-Parallel Pattern



Many other algorithms

Collaborative Filtering
- » Alternating Least Squares
- » Stochastic Gradient Descent
- » Tensor Factorization

Structured Prediction
- » Loopy Belief Propagation
- » Max-Product Linear Programs
- » Gibbs Sampling

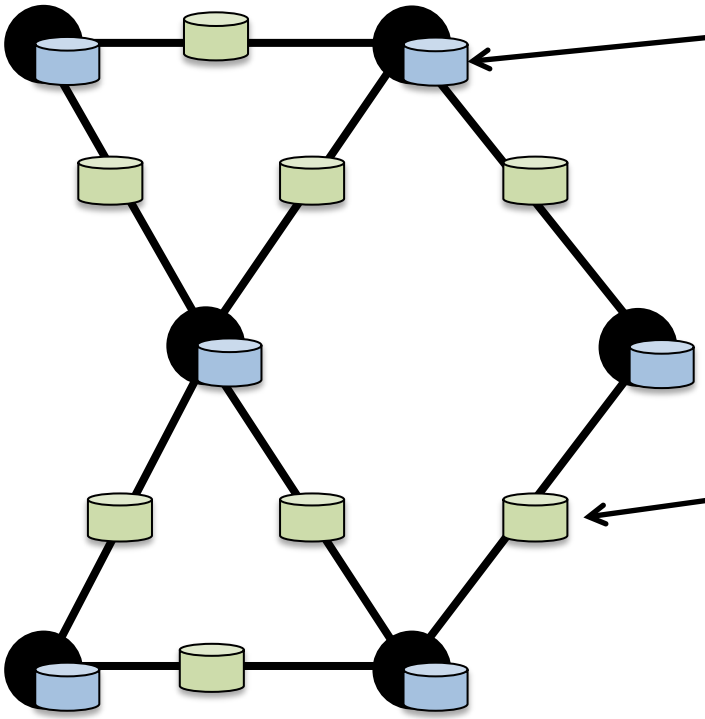
Semi-supervised ML
- » Graph SSL
- » CoEM

Community Detection
- » Triangle-Counting
- » K-core Decomposition
- » K-Truss

Graph Analytics
- » PageRank
- » Personalized PageRank
- » Shortest Path
- » Graph Coloring

Classification
- » Neural Networks

Property Graphs



Vertex Property:

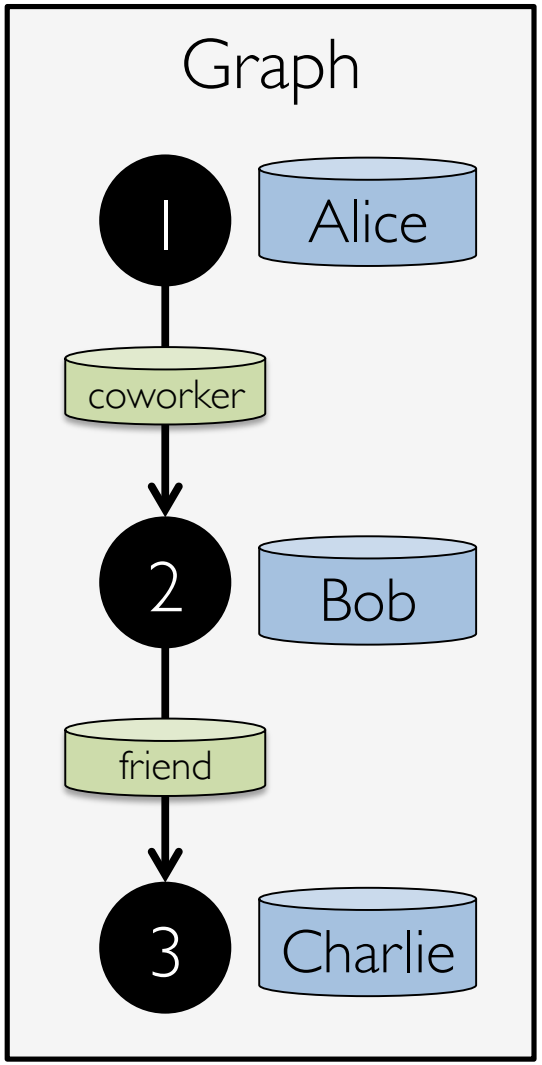
- User Profile
- Current PageRank Value

Edge Property:

- Weights
- Relationships
- Timestamps

Creating a Graph

```
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)
```



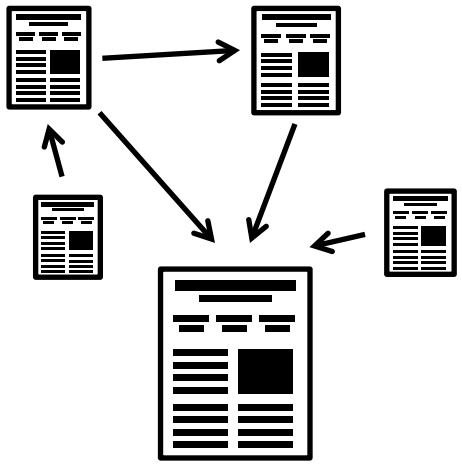
Graph Operations

```
class Graph[VD, ED] {
    // Table Views -----------------------
    def vertices: RDD[(VertexId, VD)]
    def edges: RDD[Edge[ED]]
    def triplets: RDD[EdgeTriplet[VD, ED]]
    // Transformations ------------------------------------
    def mapVertices[VD2](f: (VertexId, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](f: Edge[ED] => ED2): Graph[VD2, ED]
    def reverse: Graph[VD, ED]
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
    // Joins ----------------------------------------
    def outerJoinVertices[U, VD2]
        (tbl: RDD[(VertexId, U)])
        (f: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
    // Computation ----------------------------------
    def mapReduceTriplets[A](
        sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
        mergeMsg: (A, A) => A): RDD[(VertexId, A)]
```

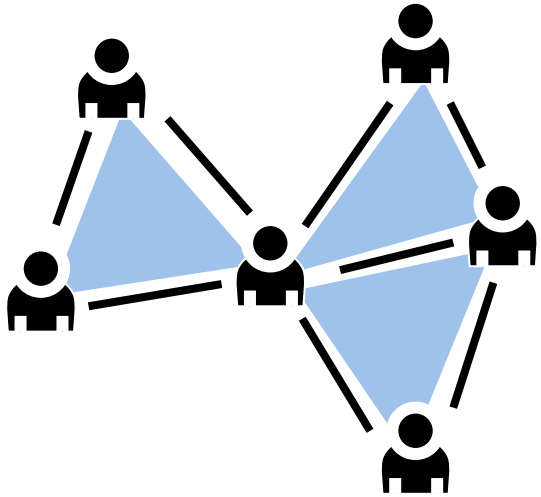
Built-in Algorithms

```
    // Continued from previous slide
    def pageRank(tol: Double): Graph[Double, Double]
    def triangleCount(): Graph[Int, ED]
    def connectedComponents(): Graph[VertexId, ED]
    // ...and more: org.apache.spark.graphx.lib
}
```

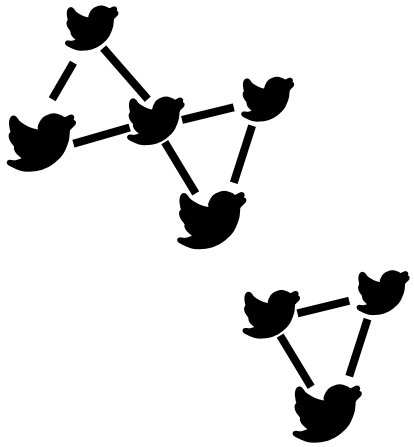
PageRank



Triangle Count

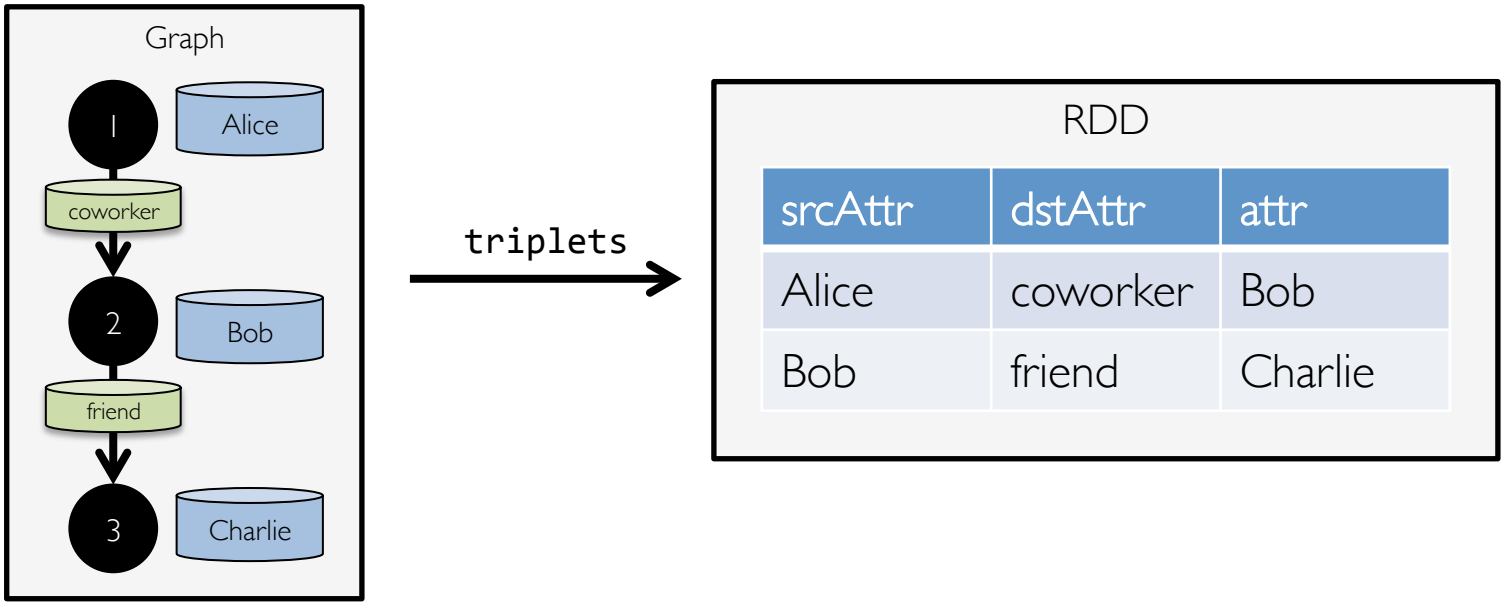


Connected Components



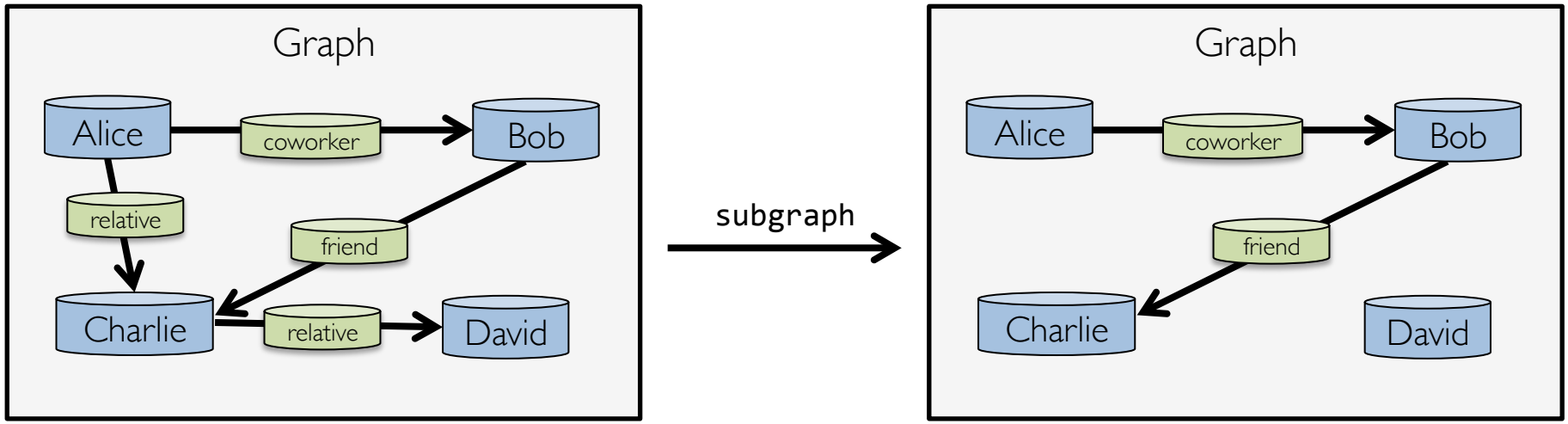
The triplets view

```
class Graph[VD, ED] {
    def triplets: RDD[EdgeTriplet[VD, ED]]
}

class EdgeTriplet[VD, ED](
  val srcId: VertexId, val dstId: VertexId, val attr: ED,
  val srcAttr: VD, val dstAttr: VD)
```



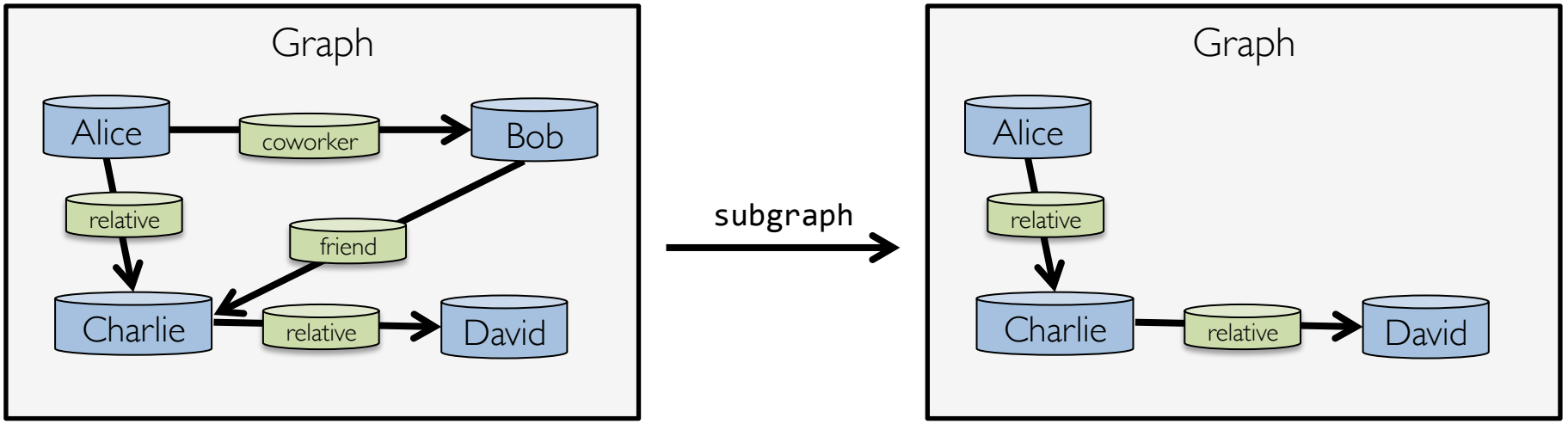
The subgraph transformation

```
class Graph[VD, ED] {
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(epred = (edge) => edge.attr != "relative")
```

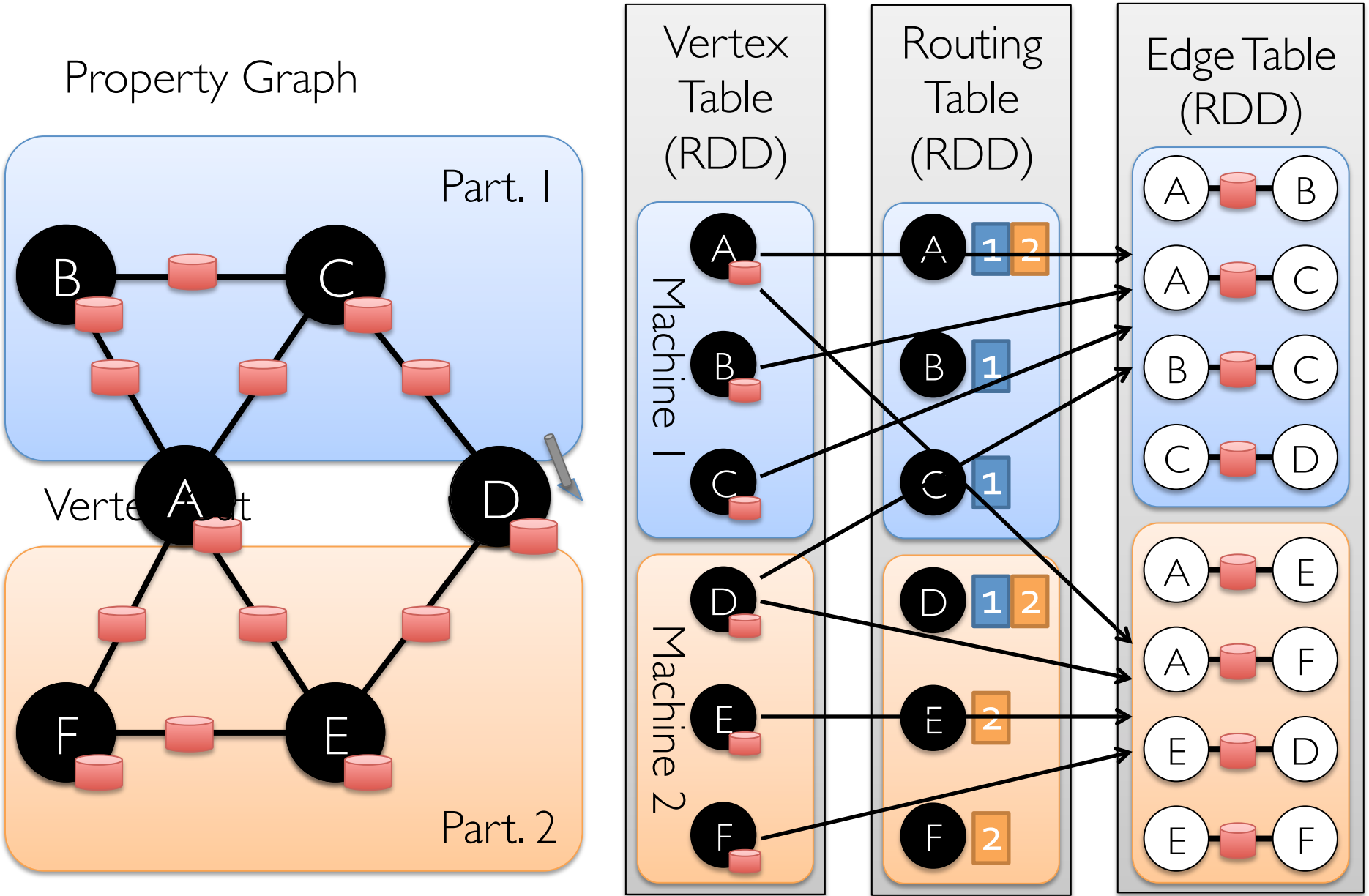


The subgraph transformation

```
class Graph[VD, ED] {
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(vpred = (id, name) => name != "Bob")
```

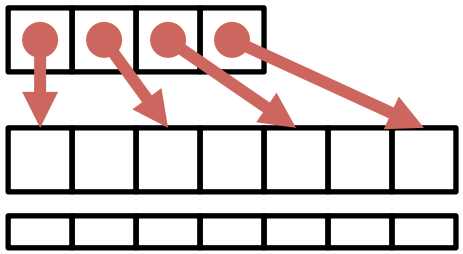


Encoding property Graphs as RDDs

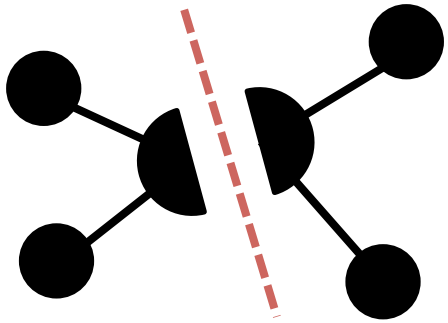


Graph System Optimizations

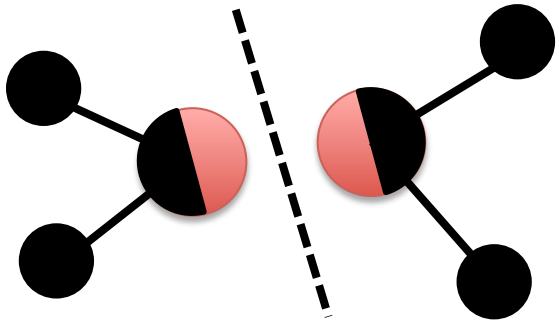
Specialized Data-Structures



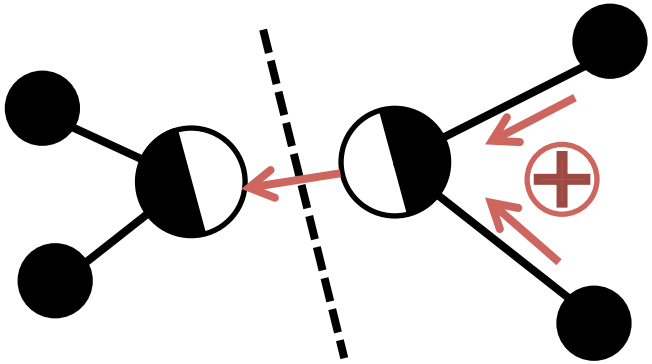
Vertex-Cuts Partitioning



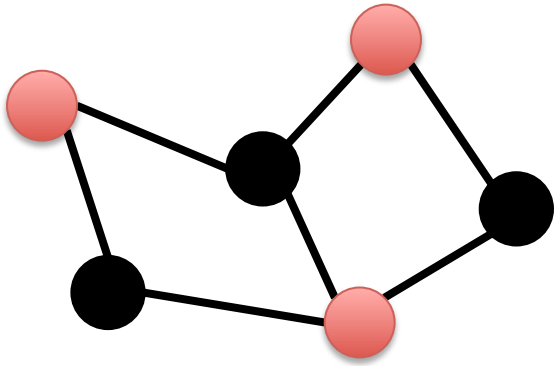
Remote Caching / Mirroring



Message Combiners

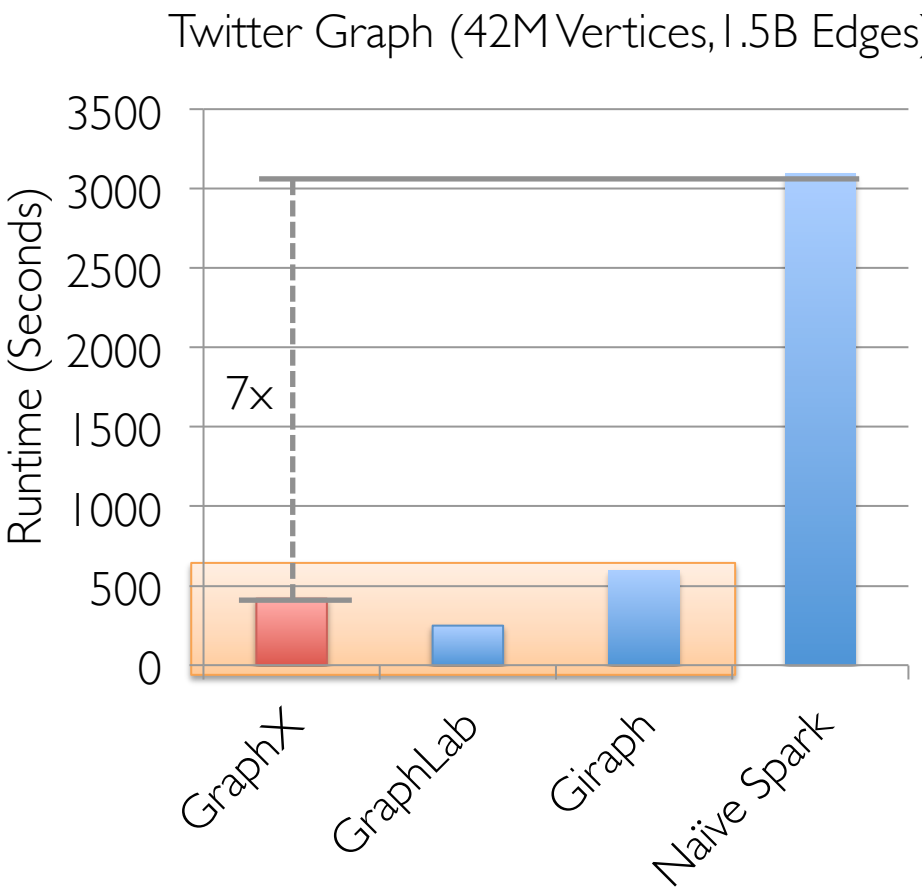


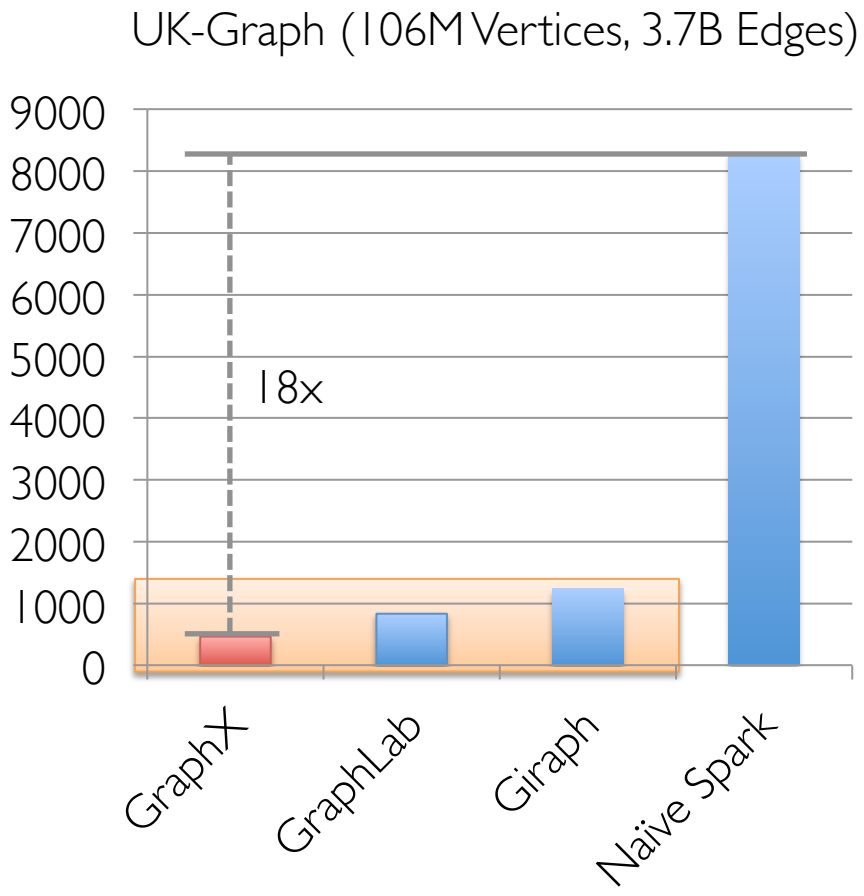
Active Set Tracking



PageRank Benchmark

EC2 Cluster of 16 x m2.4xLarge (8 cores) + 1GigE





dbTrento

Graph operators

```scala
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

```scala
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  // 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]]
  // Functions for caching graphs ==================================================================
  def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
  def cache(): Graph[VD, ED]
  def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
  // Change the partitioning heuristic  ============================================================
  def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
  // Transform vertex and edge attributes ==========================================================
  def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
  def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
  def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
  def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
    : Graph[VD, ED2]
  // Modify the graph structure ====================================================================
  def reverse: Graph[VD, ED]
  def subgraph(
      epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
      vpred: (VertexId, VD) => Boolean = ((v, d) => true))
    : Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
  // Join RDDs with the graph ======================================================================
  def joinVertices[U](table: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, U) => VD): Graph[VD, ED]
  def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])
      (mapFunc: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
  // Aggregate information about adjacent triplets =================================================
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId, VD)]]
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A]
  // Iterative graph-parallel computation ==========================================================
  def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
      vprog: (VertexId, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId,A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED]
  // Basic graph algorithms ========================================================================
  def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
  def connectedComponents(): Graph[VertexId, ED]
  def triangleCount(): Graph[Int, ED]
  def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]
}
```

Graph operators

```
// 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]]
// Functions for caching graphs =========================================================
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
def cache(): Graph[VD, ED]
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
// Change the partitioning heuristic  ===================================================
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
// Transform vertex and edge attributes ================================================
def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
  : Graph[VD, ED2]
```

Graph operators

```
// Modify the graph structure ================================================
def reverse: Graph[VD, ED]
def subgraph(
    epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
    vpred: (VertexId, VD) => Boolean = ((v, d) => true))
  : Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
// Join RDDs with the graph ==================================================
def joinVertices[U](table: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, U) => VD): Graph[VD, ED]
def outerJoinVertices[U, VD2](other: RDD[(VertexId, U)])
    (mapFunc: (VertexId, VD, Option[U]) => VD2)
  : Graph[VD2, ED]
// Aggregate information about adjacent triplets =============================
def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexId, VD)]]
def aggregateMessages[Msg: ClassTag](
    sendMsg: EdgeContext[VD, ED, Msg] => Unit,
    mergeMsg: (Msg, Msg) => Msg,
    tripletFields: TripletFields = TripletFields.All)
  : VertexRDD[A]
```

Graph operators

```
// Iterative graph-parallel computation ========================================================
def pregel[A](initialMsg: A, maxIterations: Int, activeDirection: EdgeDirection)(
    vprog: (VertexId, VD, A) => VD,
    sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId,A)],
    mergeMsg: (A, A) => A)
  : Graph[VD, ED]
// Basic graph algorithms ======================================================================
def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double, Double]
def connectedComponents(): Graph[VertexId, ED]
def triangleCount(): Graph[Int, ED]
def stronglyConnectedComponents(numIter: Int): Graph[VertexId, ED]
```

Property Operators

```
class Graph[VD, ED] {
  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 newVertices = graph.vertices.map { case (id, attr) => (id, mapUdf(id, attr)) }
val newGraph = Graph(newVertices, graph.edges)
```

Instead, use mapVertices to preserve the indices:

```
val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

```
// Given a graph where the vertex property is the out degree
val inputGraph: Graph[Int, String] =
  graph.outerJoinVertices(graph.outDegrees)((vid, _, degOpt) => degOpt.getOrElse(0))
// Construct a graph where each edge contains the weight
// and each vertex is the initial PageRank
val outputGraph: Graph[Double, Double] =
  inputGraph.mapTriplets(triplet => 1.0 / triplet.srcAttr).mapVertices((id, _) => 1.0)
```

Structural Operators

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

Structural Operations

```
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
                       (4L, ("peter", "student"))))
// Create an RDD for edges
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(4L, 0L, "student"),   Edge(5L, 0L, "colleague")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
// Notice that there is a user 0 (for which we have no information) connected to users
// 4 (peter) and 5 (franklin).
graph.triplets.map(
  triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1
).collect.foreach(println(_))
// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// The valid subgraph will disconnect users 4 and 5 by removing user 0
validGraph.vertices.collect.foreach(println(_))
validGraph.triplets.map(
  triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1
).collect.foreach(println(_))
```

Join Operators

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

```
val nonUniqueCosts: RDD[(VertexId, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
  (id, oldCost, extraCost) => oldCost + extraCost)
```

```
val outDegrees: VertexRDD[Int] = graph.outDegrees
val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
  outDegOpt match {
    case Some(outDeg) => outDeg
    case None => 0 // No outDegree means zero outDegree
  }
}
```

Aggregate Messages

```scala
class Graph[VD, ED] {
  def aggregateMessages[Msg: ClassTag](
      sendMsg: EdgeContext[VD, ED, Msg] => Unit,
      mergeMsg: (Msg, Msg) => Msg,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[Msg]
}
```

```scala
import org.apache.spark.graphx.{Graph, VertexRDD}
import org.apache.spark.graphx.util.GraphGenerators

// Create a graph with "age" as the vertex property.
// Here we use a random graph for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapVertices( (id, _) => id.toDouble )
// Compute the number of older followers and their total age
val olderFollowers: VertexRDD[(Int, Double)] = graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) =>
    value match { case (count, totalAge) => totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

dbTrento

Compute Degree information

```
// Define a reduce operation to compute the highest degree vertex
def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
  if (a._2 > b._2) a else b
}
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

```
class GraphOps[VD, ED] {
  def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexId]]
  def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[ Array[(VertexId, VD)] ]
}
```

PageRank

```scala
import org.apache.spark.graphx.GraphLoader

// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001).vertices
// Join the ranks with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// Print the result
println(ranksByUsername.collect().mkString("\n"))
```

Connected Components

```
import org.apache.spark.graphx.GraphLoader

// Load the graph as in the PageRank example
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt")
// Find the connected components
val cc = graph.connectedComponents().vertices
// Join the connected components with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ccByUsername = users.join(cc).map {
  case (id, (username, cc)) => (username, cc)
}
// Print the result
println(ccByUsername.collect().mkString("\n"))
```

Triangle Counting

```scala
import org.apache.spark.graphx.{GraphLoader, PartitionStrategy}

// Load the edges in canonical order and partition the graph for triangle count
val graph = GraphLoader.edgeListFile(sc, "data/graphx/followers.txt", true)
  .partitionBy(PartitionStrategy.RandomVertexCut)
// Find the triangle count for each vertex
val triCounts = graph.triangleCount().vertices
// Join the triangle counts with the usernames
val users = sc.textFile("data/graphx/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val triCountByUsername = users.join(triCounts).map { case (id, (username, tc)) =>
  (username, tc)
}
// Print the result
println(triCountByUsername.collect().mkString("\n"))
```

Contacts

For any problem, send a mail to

daniele.foroni@unitn.it