

# Introduction to Graph Analytics

EMSE 6992 Introduction to Data Science

Benjamin S. Harvey

# Outline

1. Graph structured data
2. Common properties of graph data
3. Graph algorithms
4. Systems for large-scale graph computation
5. GraphX: Graph Computation in Spark
6. Summary of other graph frameworks

# Graph structured data is everywhere ...

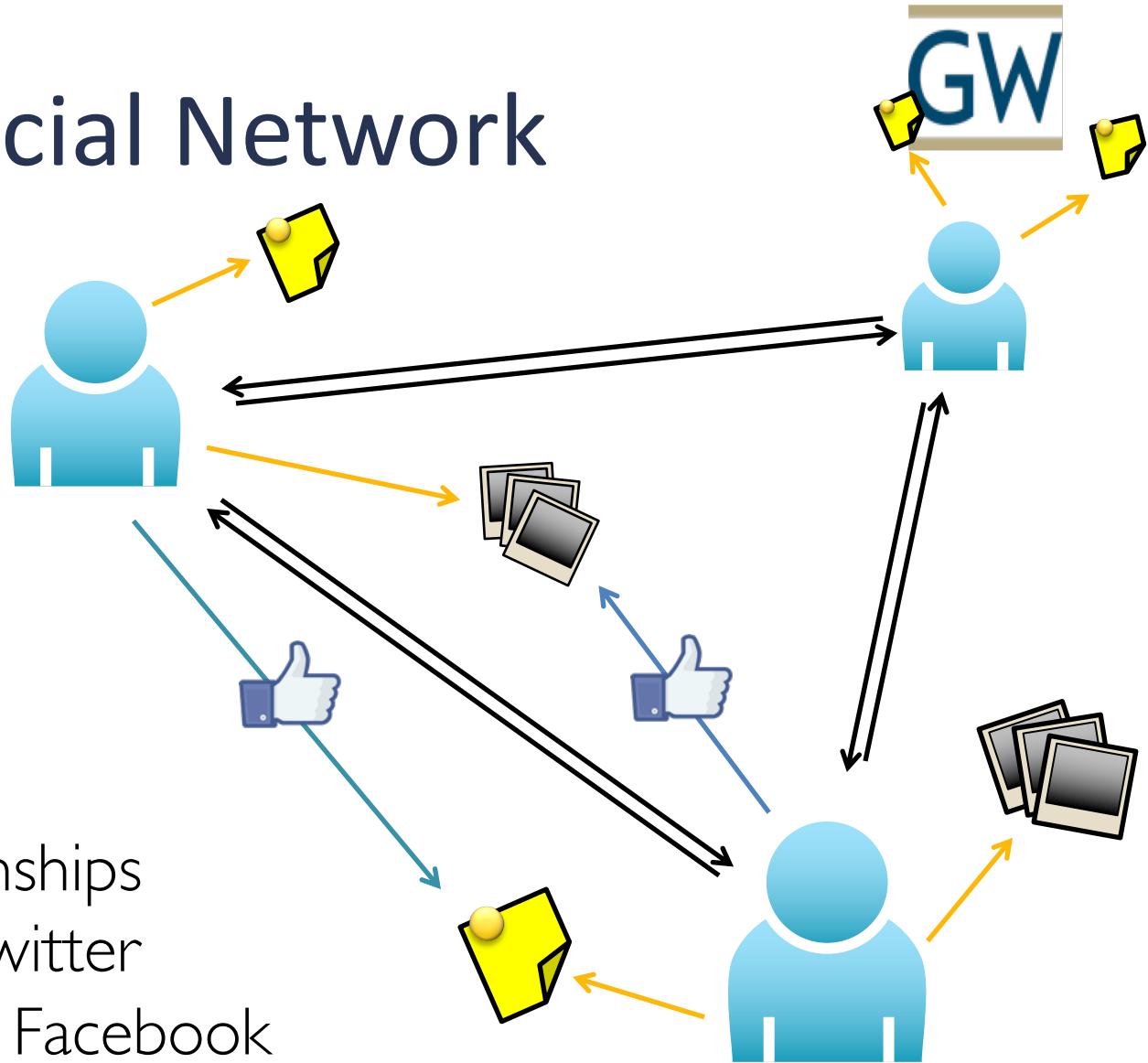
# Social Network

## Vertices

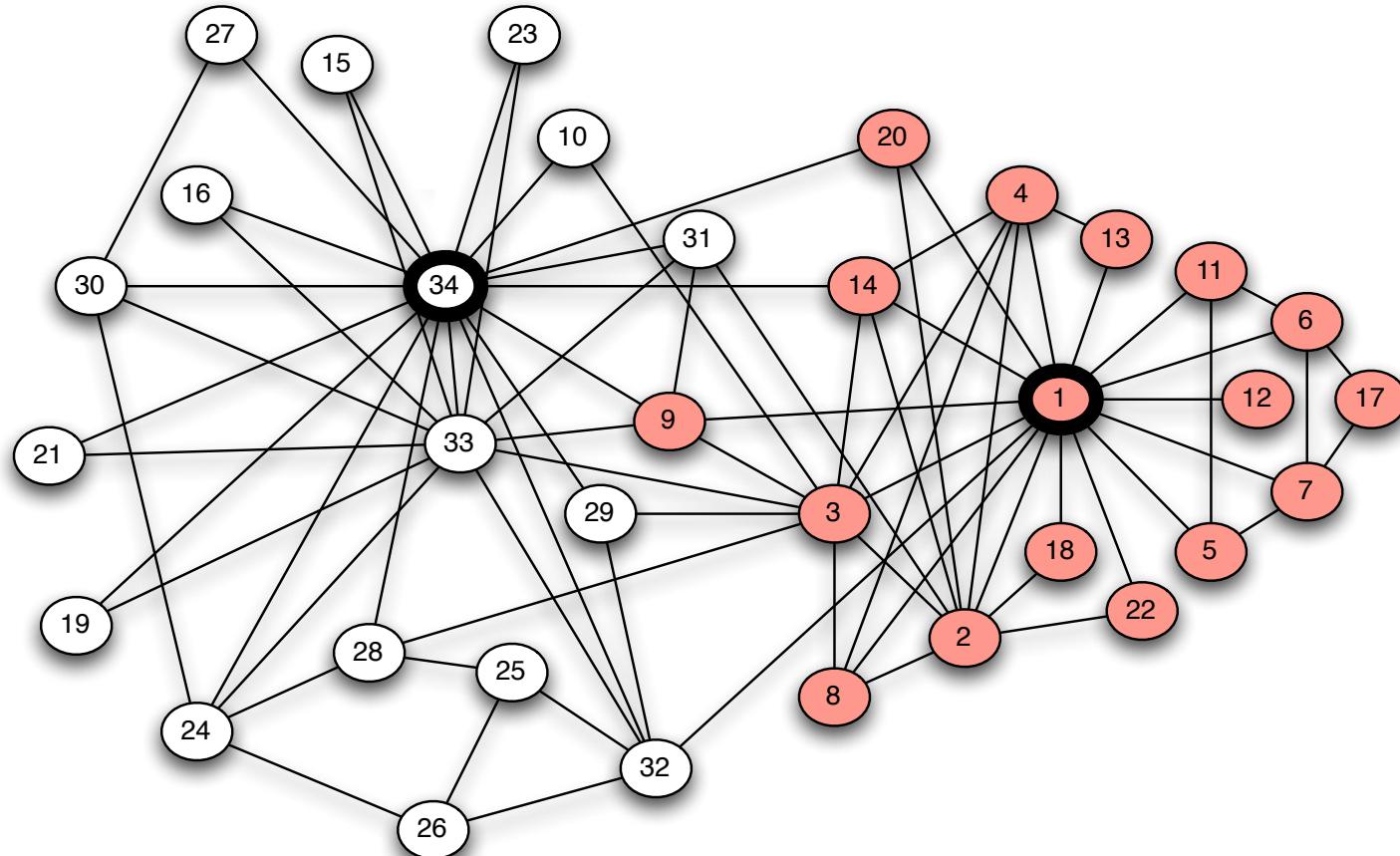
- Users
- Posts / Images

## Edges

- Social Relationships
  - *Directed*: Twitter
  - *Undirected*: Facebook
- Likes



# Actual Social Graph

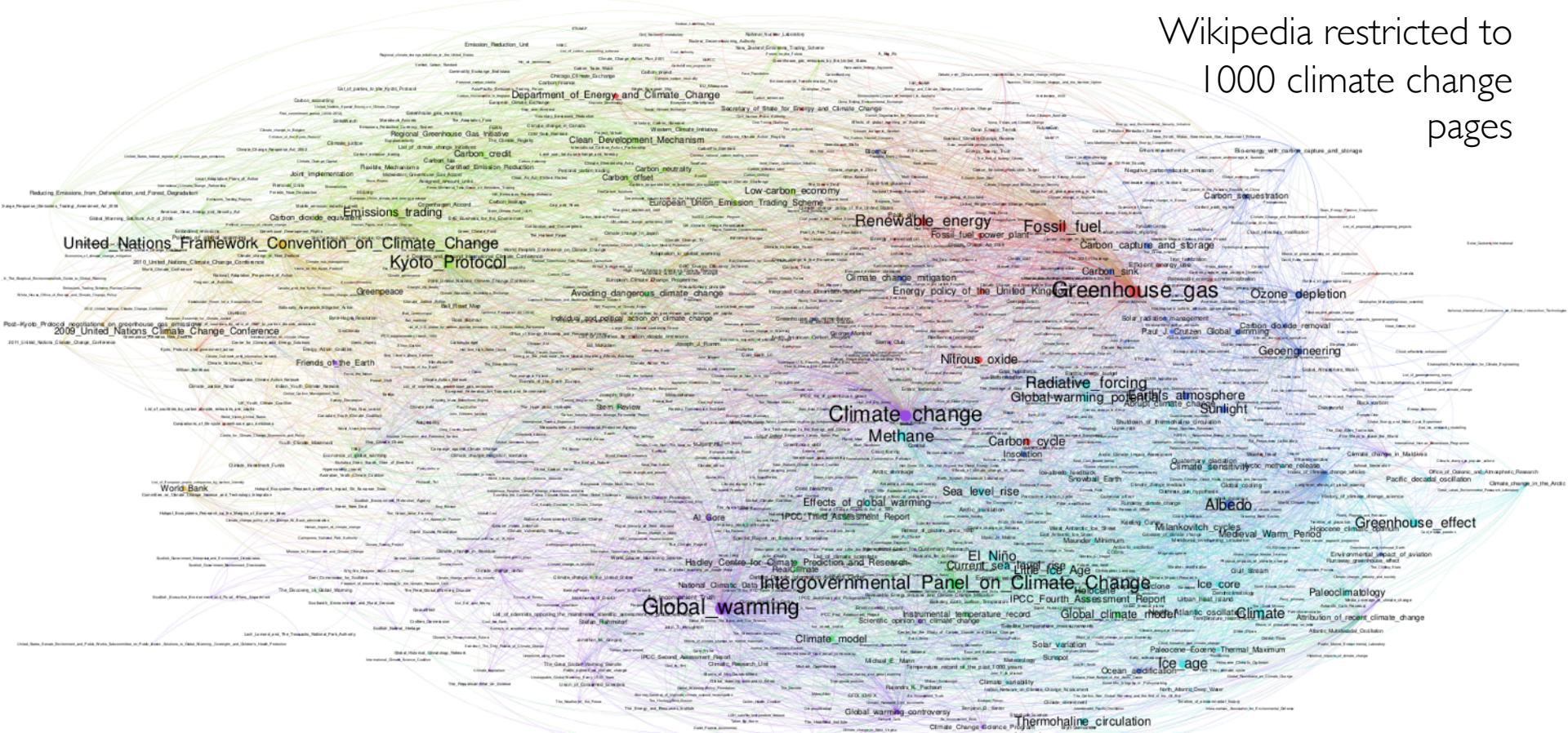


Karate Club Network

# Web Graphs



# Wikipedia restricted to 1000 climate change pages



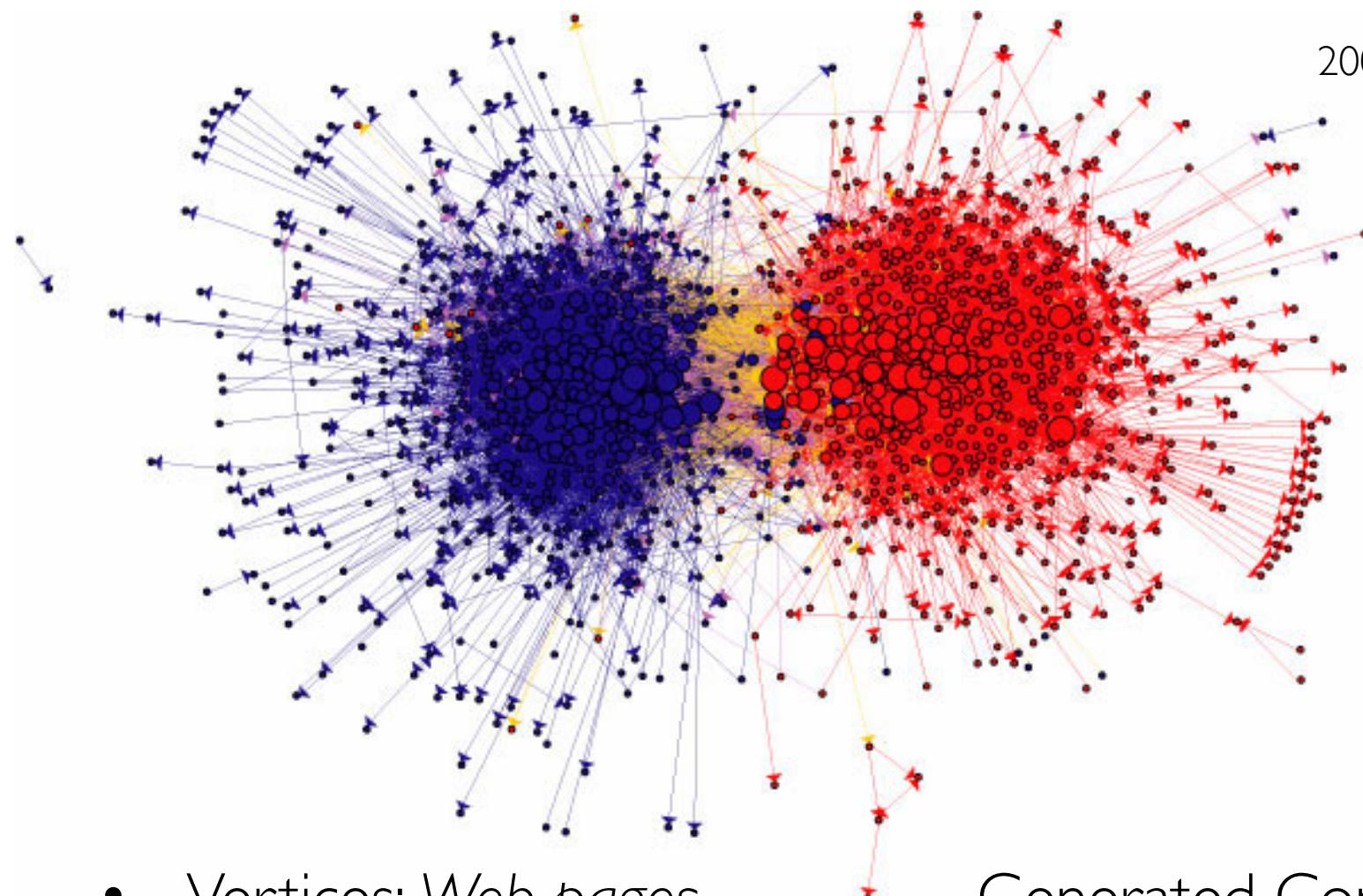
- Vertices: Web-pages
  - Edges: Links (Directed)

## Generated Content:

- Click-streams

# Web Graphs

2004 Political Blogs

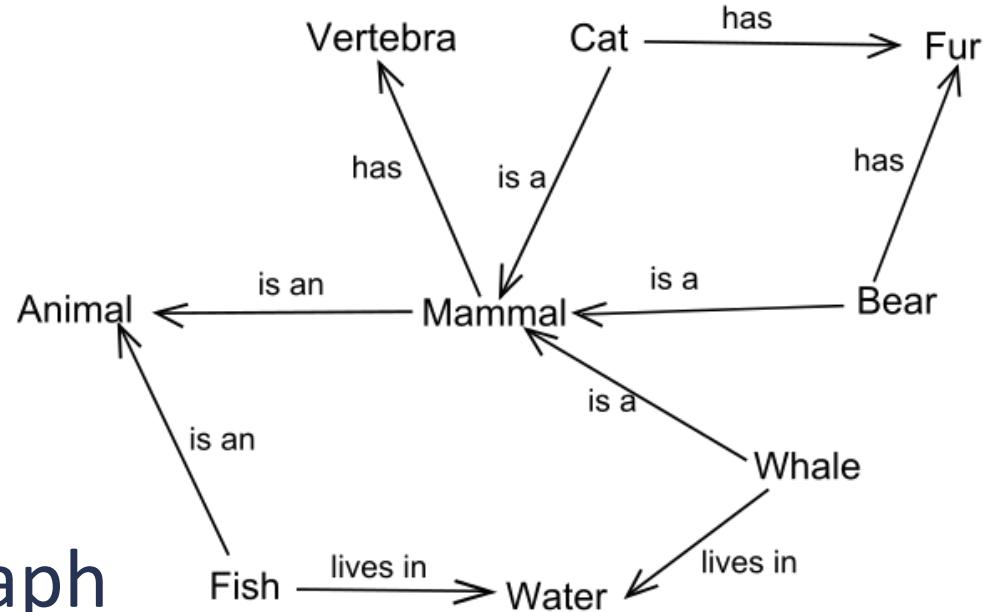


- Vertices: Web-pages
- Edges: Links (*Directed*)

- Generated Content:
- Click-streams

# Semantic Networks

- Organize Knowledge
- Vertices: Subject, Object
- Edges: Predicates
- Example:  
Google Knowledge Graph
  - 570M Vertices
  - 18B Edges



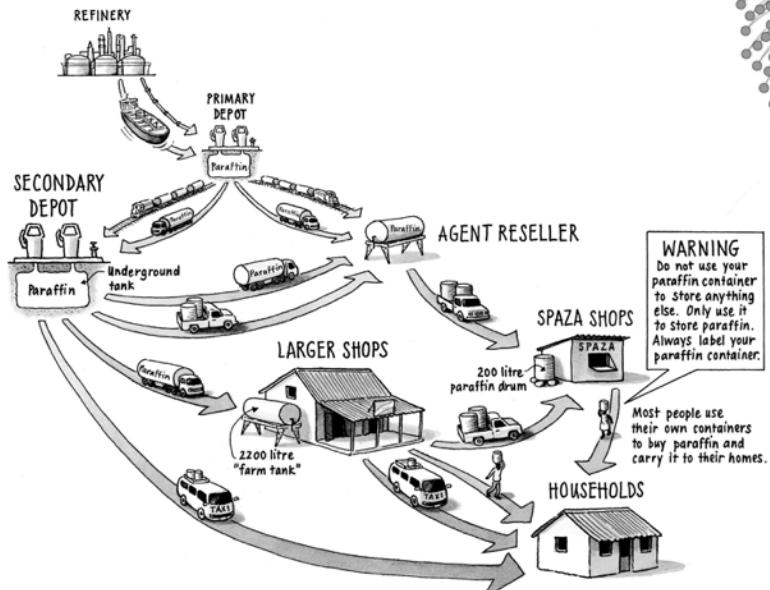
<http://wiki.dbpedia.org>

# Transaction Networks

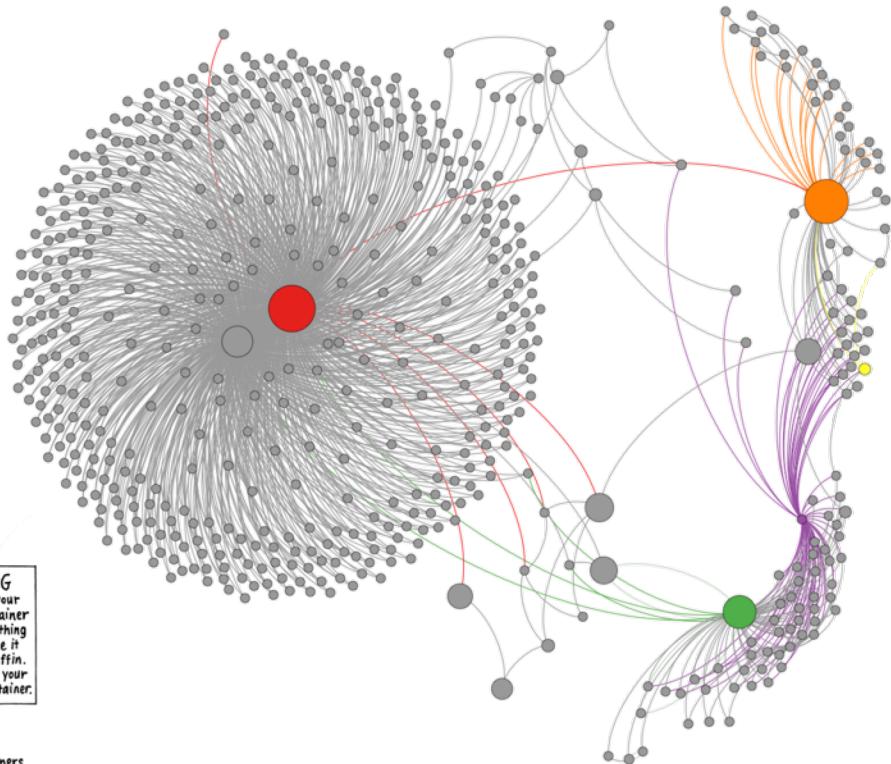
Supply Chain:

Vertices: Suppliers/Consumers

Edges: Exchange of Goods



<http://anonymity-in-bitcoin.blogspot.com/2011/07/bitcoin-is-not-anonymous.html>



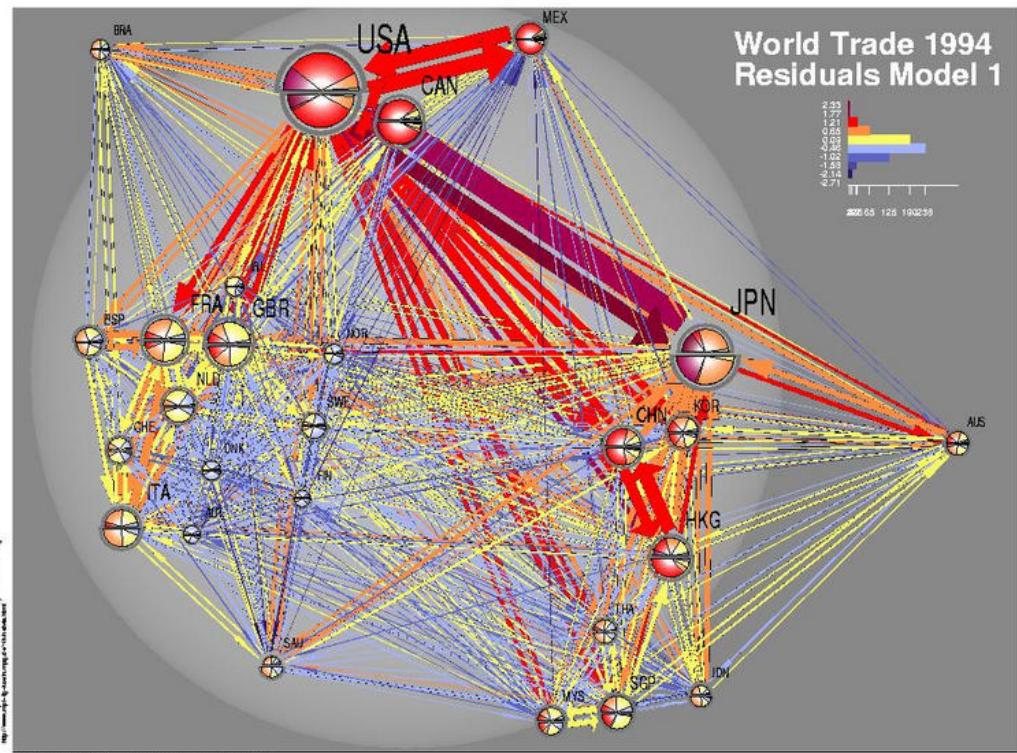
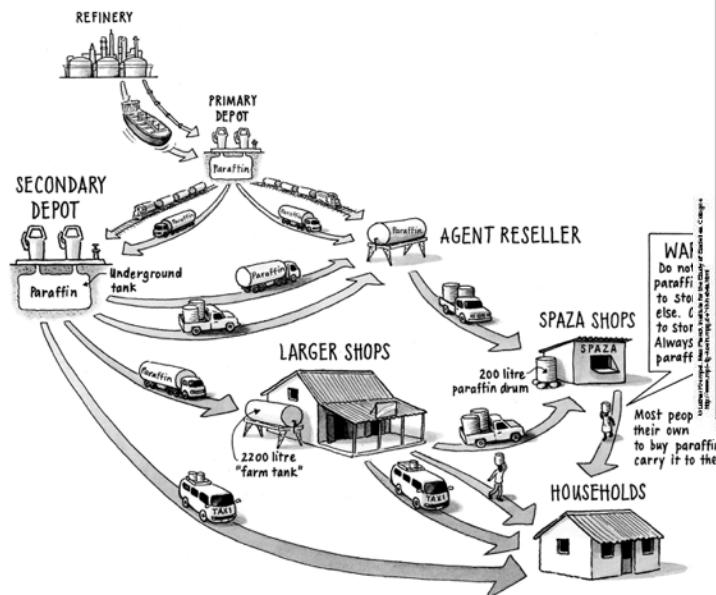
Transaction Networks (e.g., Bitcoin):  
Vertices: Users  
Edges: Exchange of Currency

# Transaction Networks

Supply Chain:

Vertices: Suppliers/Consumers

Edges: Exchange of Goods



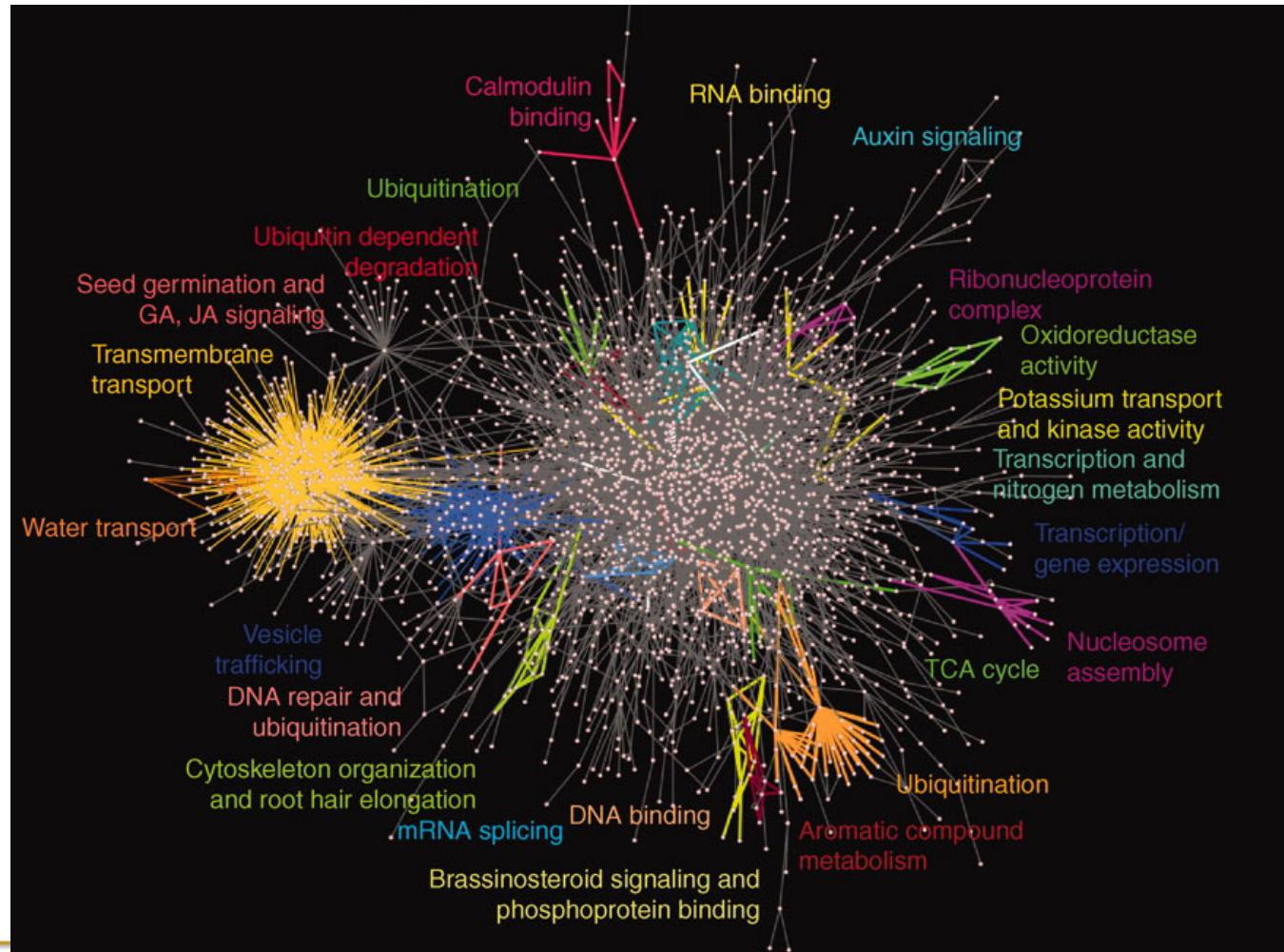
Transaction Networks (e.g., Bitcoin):  
 Vertices: Users  
 Edges: Exchange of Currency

# Biological Networks

Protein-Protein Interaction Networks (Interactomes)

Vertices: Proteins

Edges: Interactions

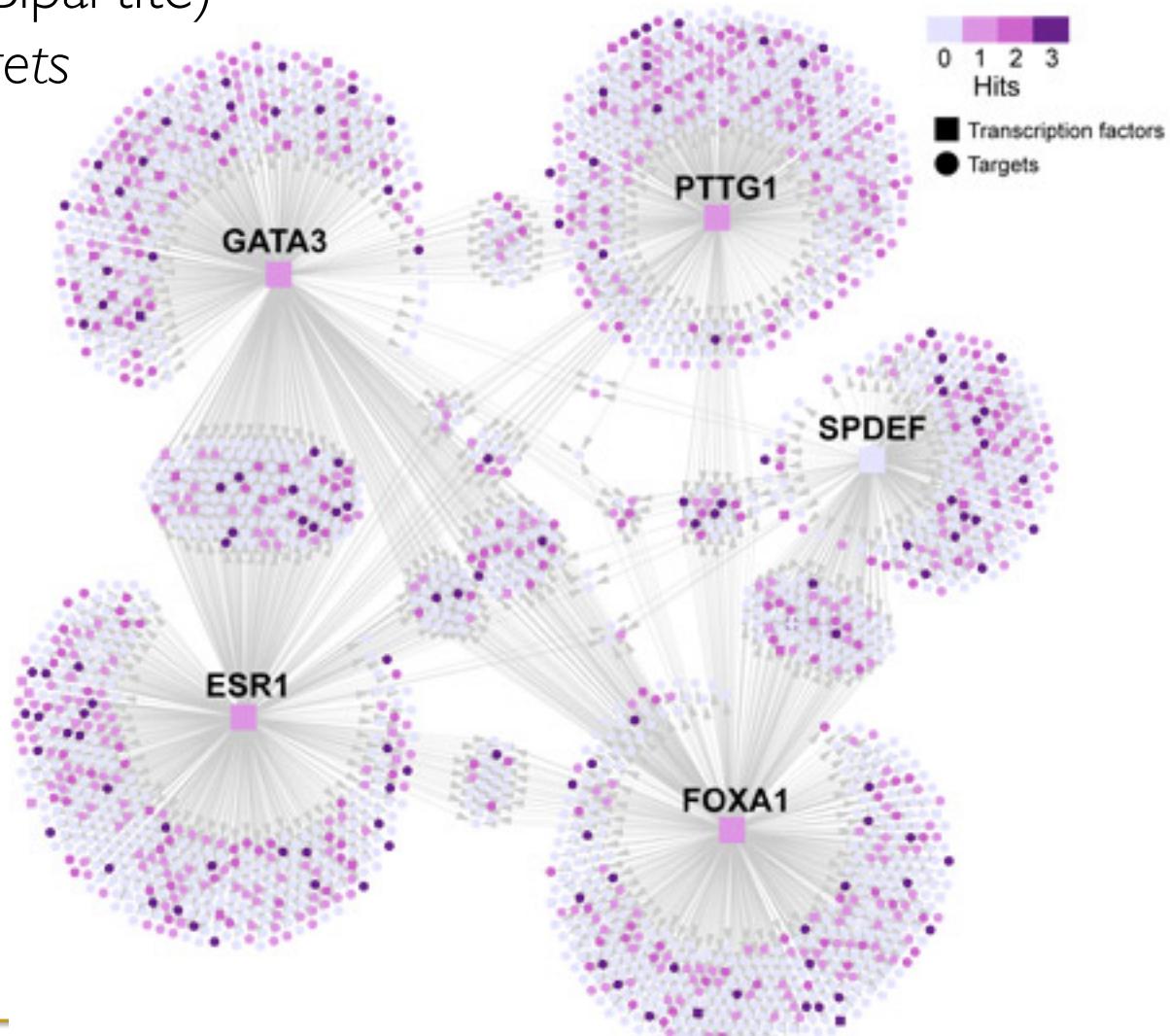


# Biological Networks

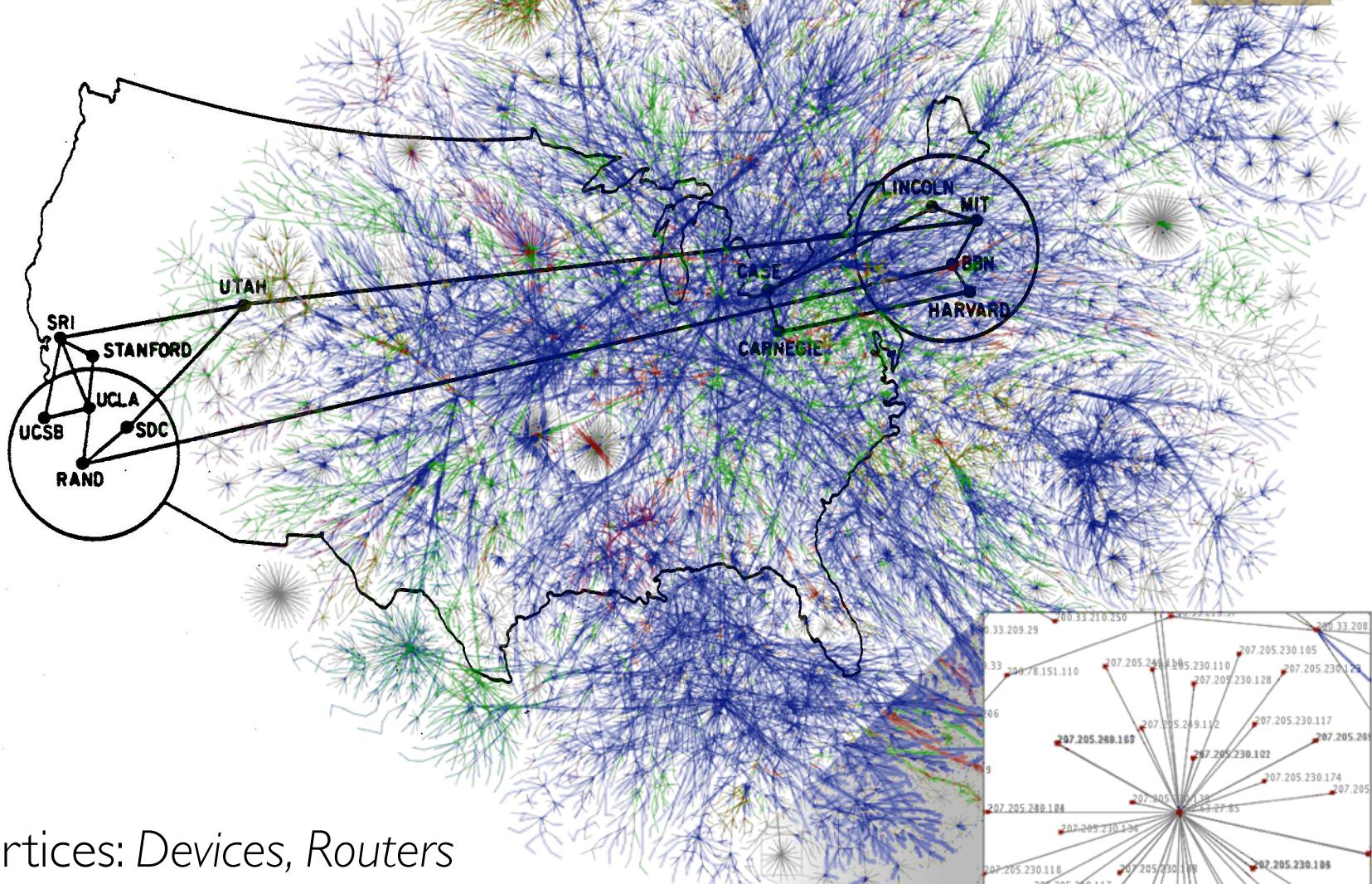
Regulatory Networks (Bipartite)

Vertices: Regulators, targets

Edges: Regulates target



# Communication Networks

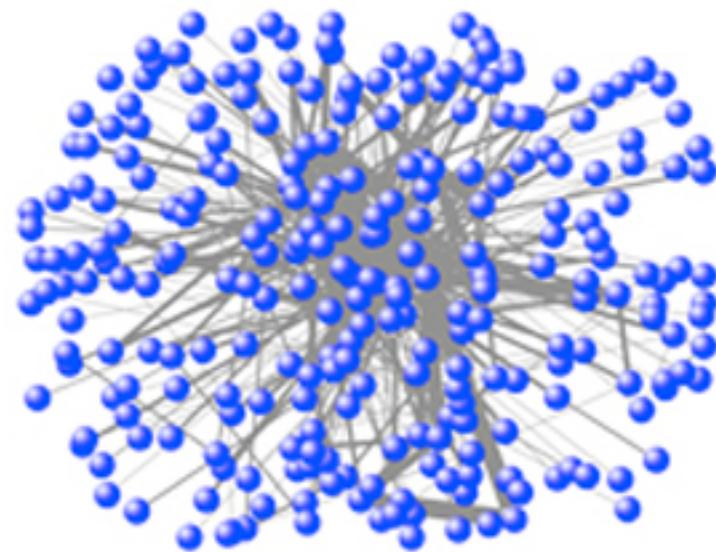
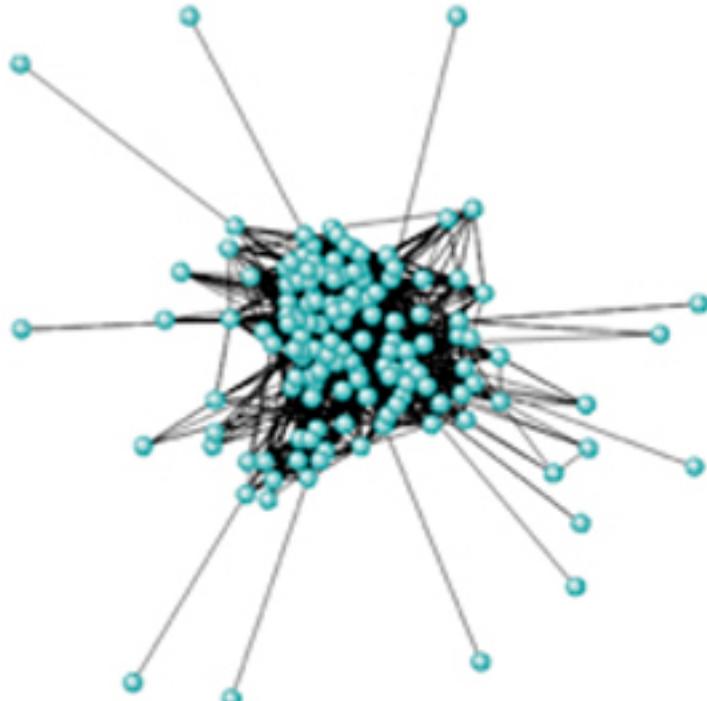


Vertices: Devices, Routers

Directed Edges: Network Flows

# Who Talks to Whom Graph

Enron Email Graphs



Vertices: Users

Directed Edges: *Email From → To*

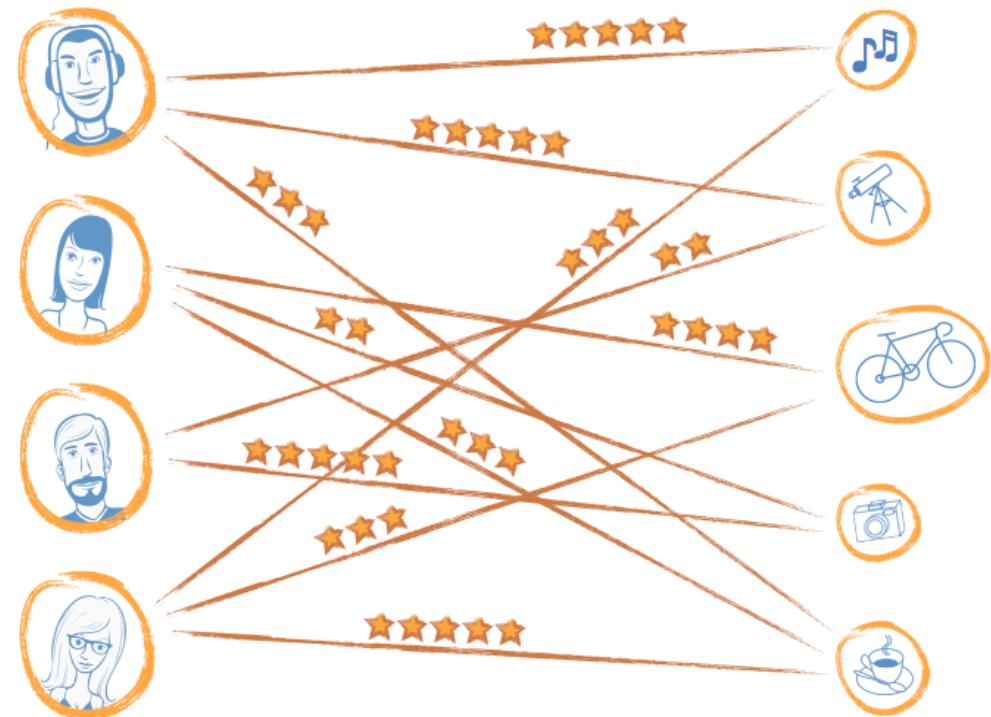
# User - Item Graphs (Recommender Systems)



Bipartite Graphs

Vertices: Users and Items

Edges: Ratings

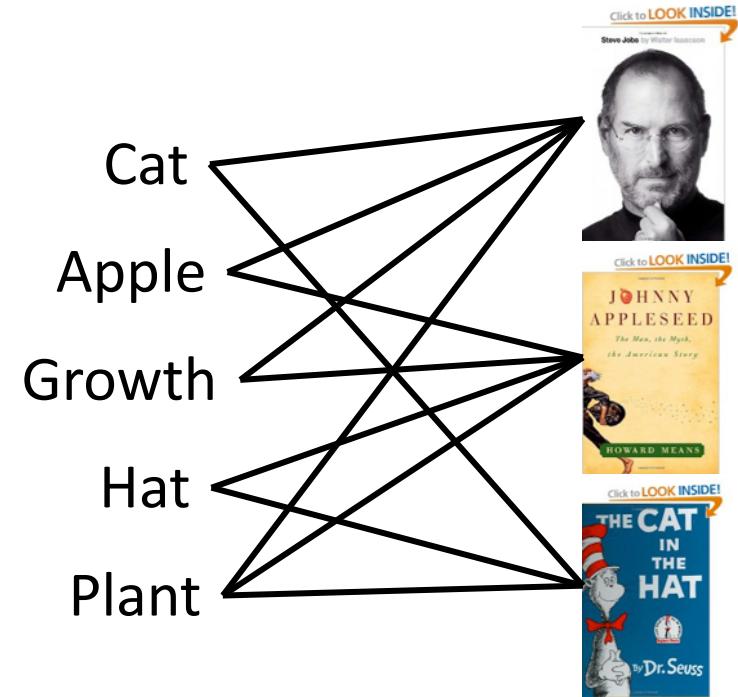
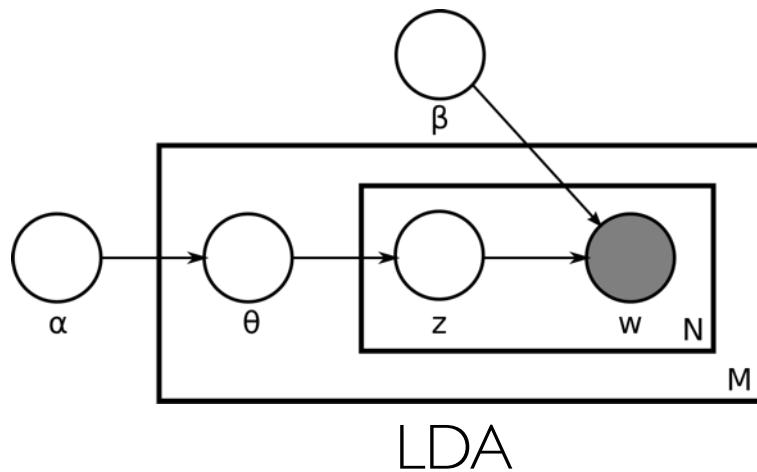


# Graphical Models



Vertices: Random Variables, Factors

Edges: Statistical Dependencies



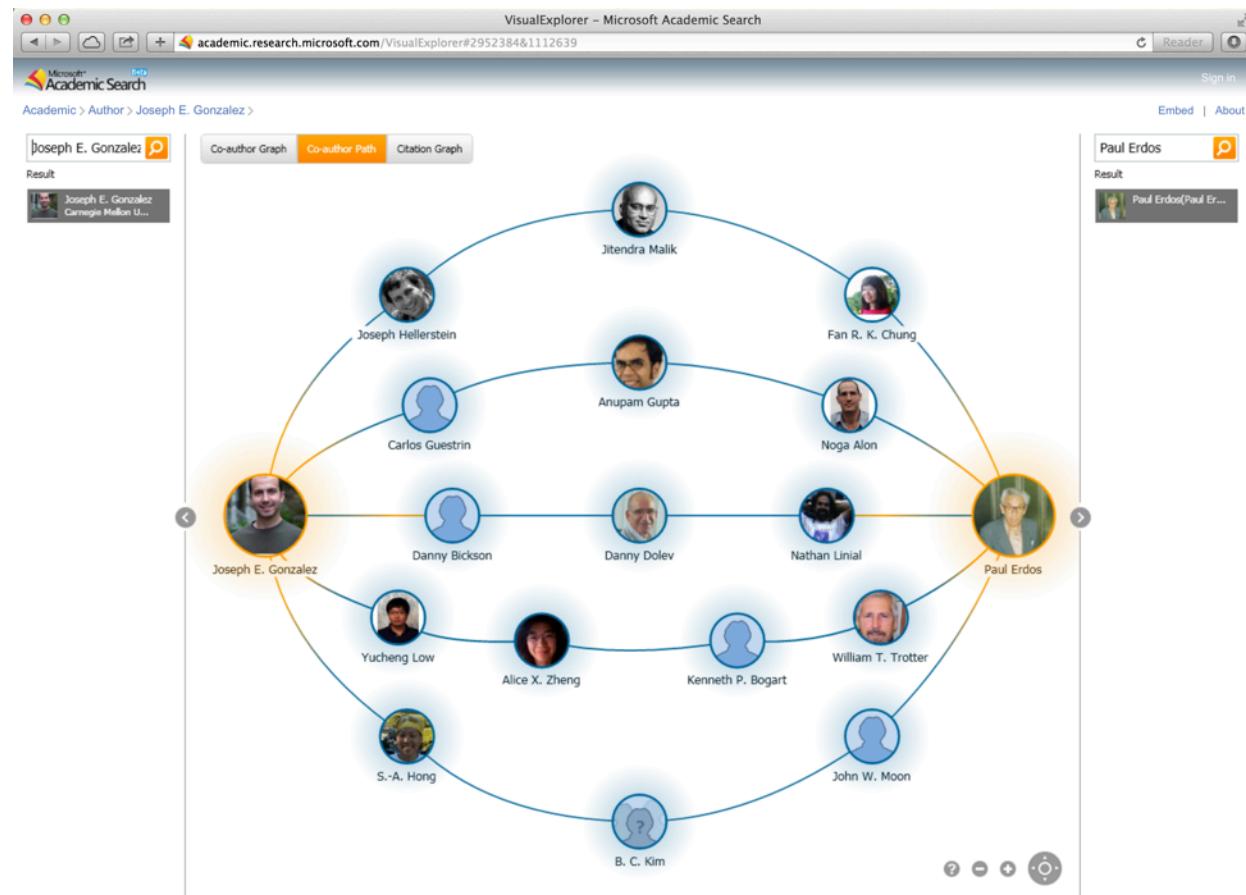
# Co-Authorship Network



Vertices: Authors  
Edges: Co-authorship

<http://academic.research.microsoft.com/VisualExplorer#2952384&1112639>

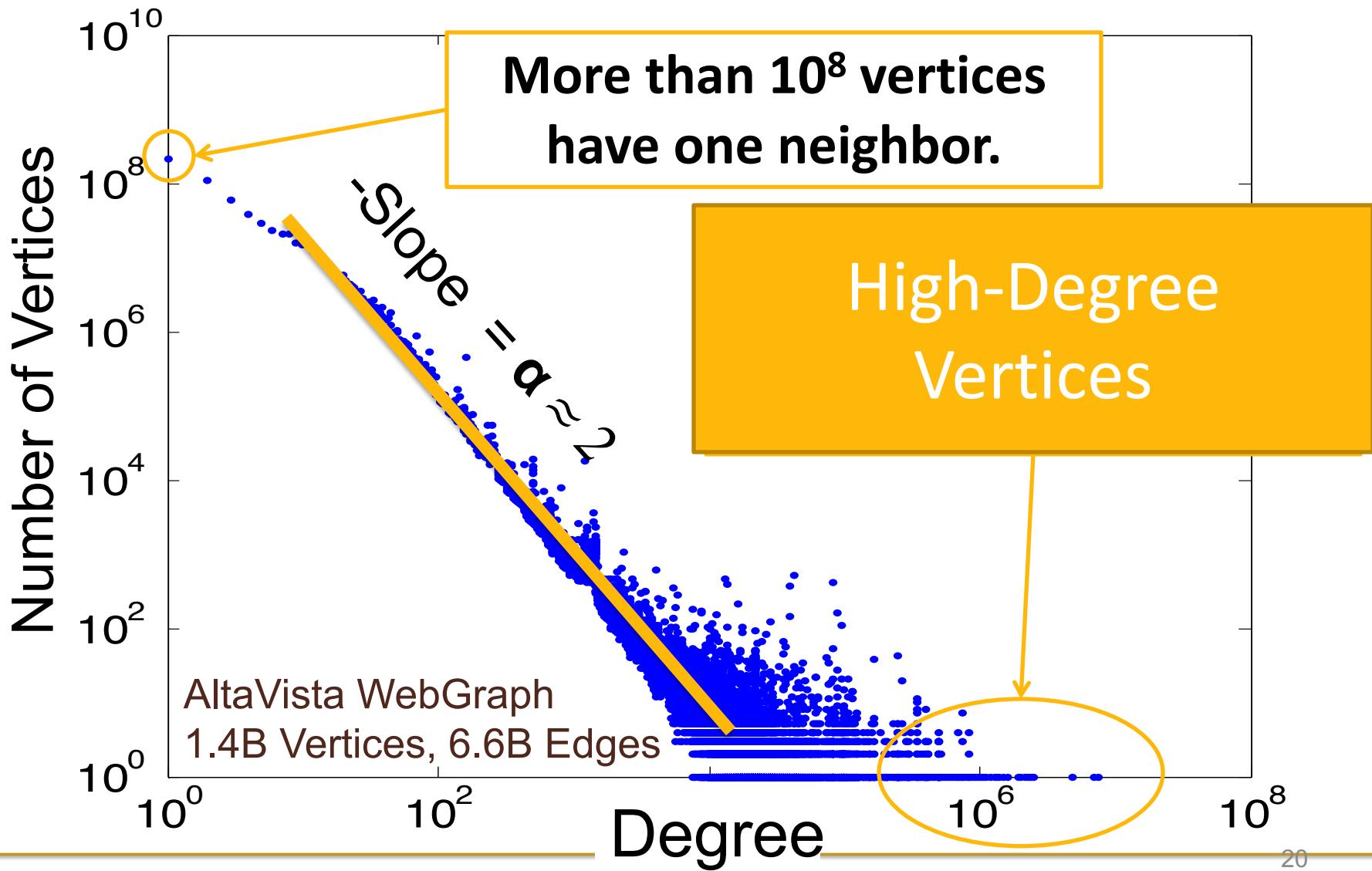
Example: Erdos Number



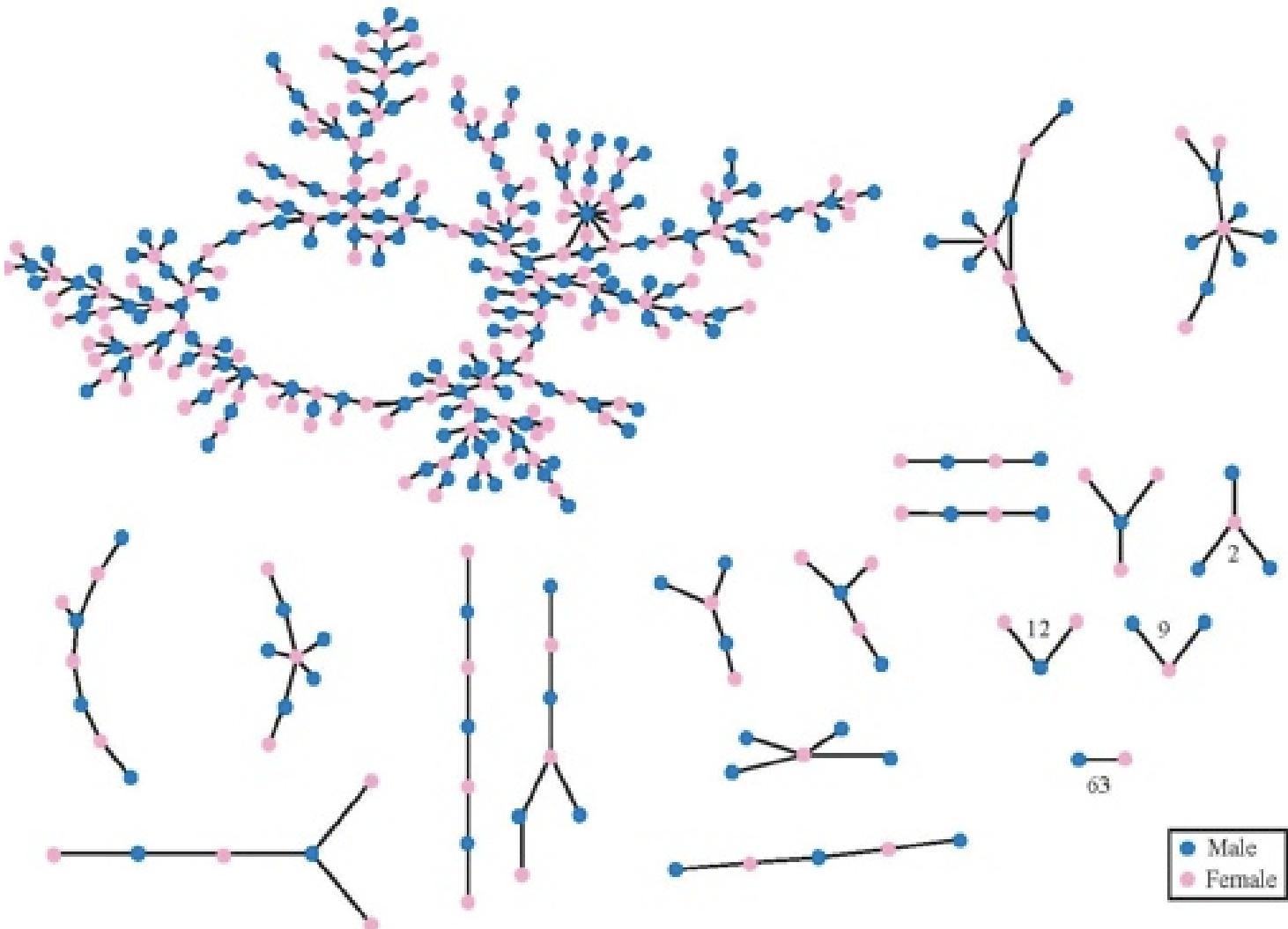
# Others?

# Common properties of graphs derived from natural phenomena

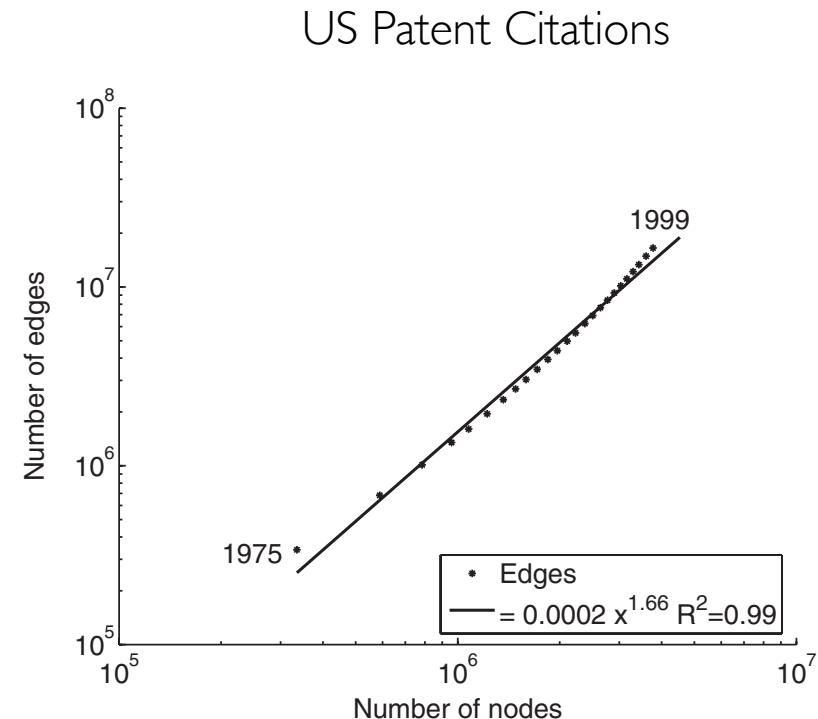
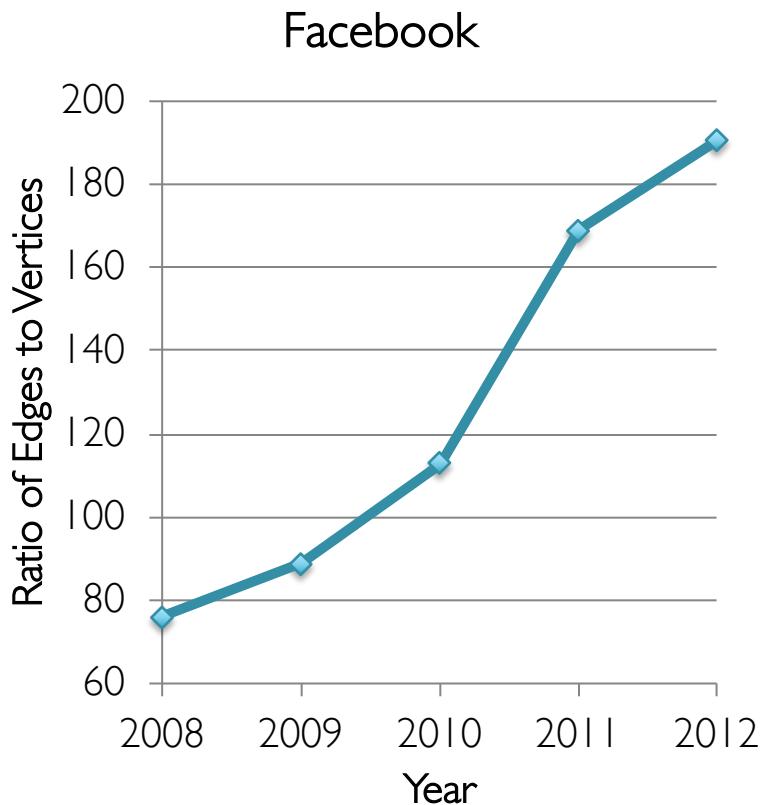
# Power-Law Degree Distribution



# Giant Connected Component

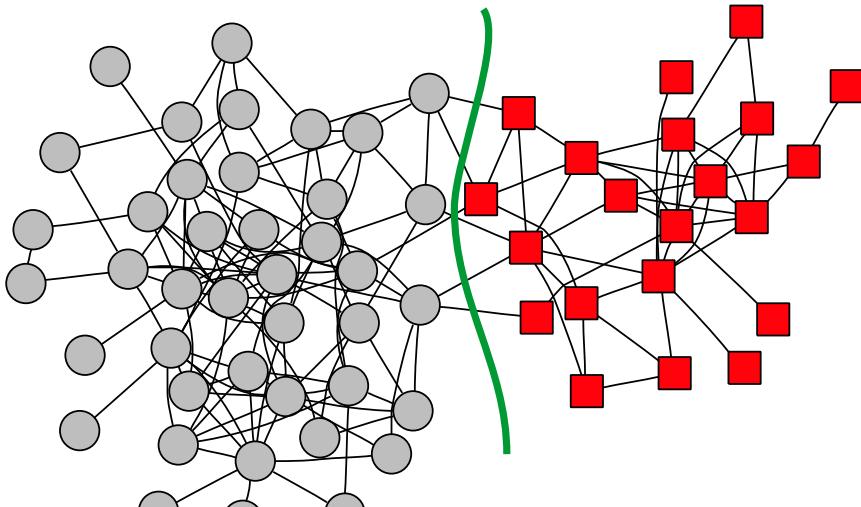


# Densification

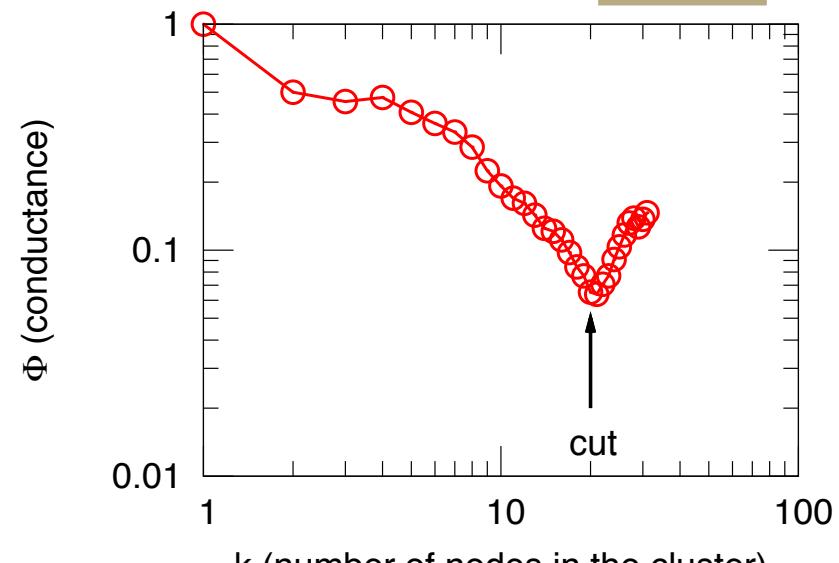


Average distance between nodes reduces over time.

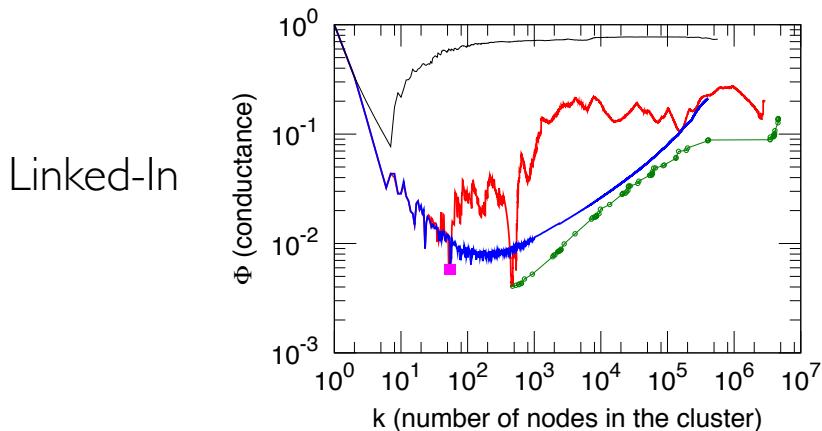
# Community Structure



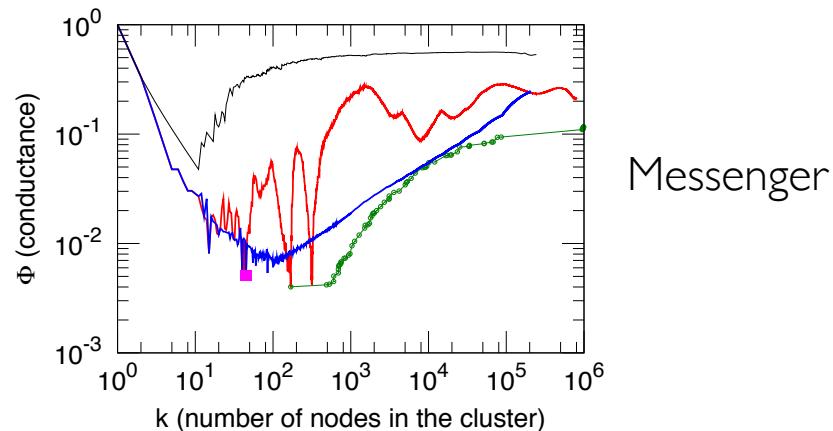
(c) Dolphins social network ...



(d) ... and it's community profile plot



LinkedIn



Messenger



# Graph Algorithms

*“Think Globally, Act Locally”*

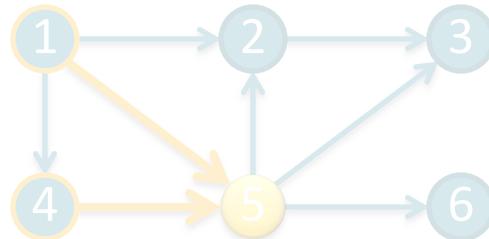
# PageRank (Centrality Measures)

- Recursive Relationship:

$$R[i] = \alpha + (1 - \alpha) \sum_{(j,i) \in E} \frac{1}{L[j]} R[j]$$

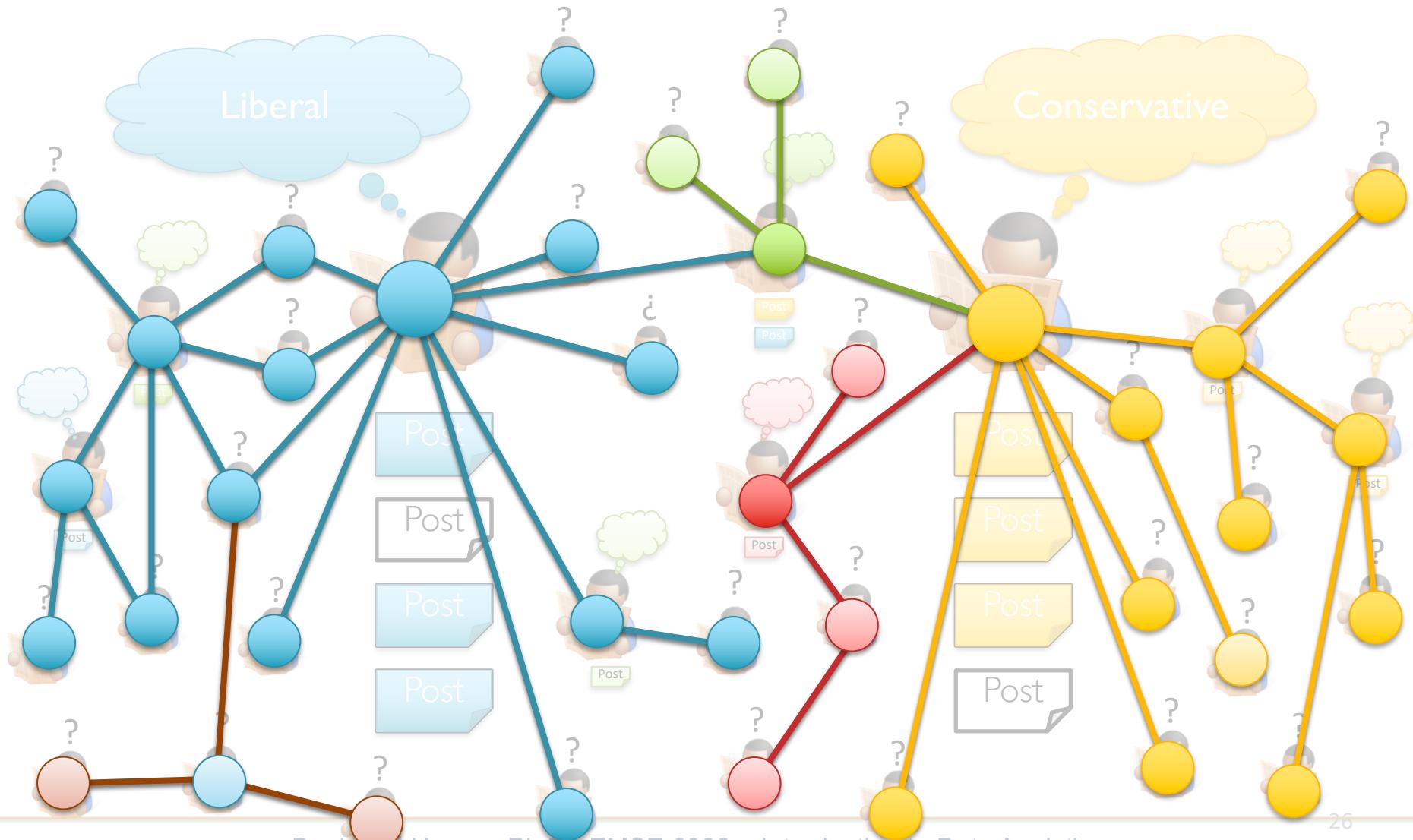
- Where:

- $\alpha$  is the random reset probability (typically 0.15)
- $L[j]$  is the number of links on page  $j$



$$R[5] = \alpha + (1 - \alpha) \left( \frac{1}{3} R[1] + \frac{1}{1} R[4] \right)$$

# Predicting Behavior



# Label Propagation (Structured Prediction)

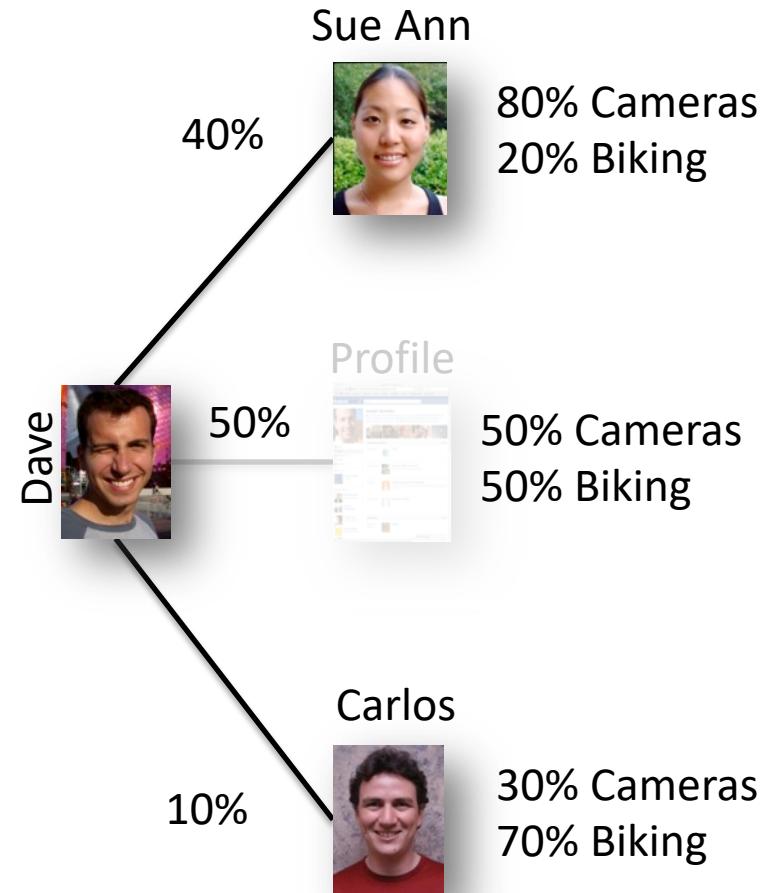
- Social Arithmetic:
  - 50% What I list on my profile
  - 40% Sue Ann Likes
  - + 10% Carlos Like

---

I Like: 60% Cameras, 40% Biking
- Recurrence Algorithm:

$$Likes[i] = \sum_{j \in Friends[i]} W_{ij} \times Likes[j]$$

– iterate until convergence



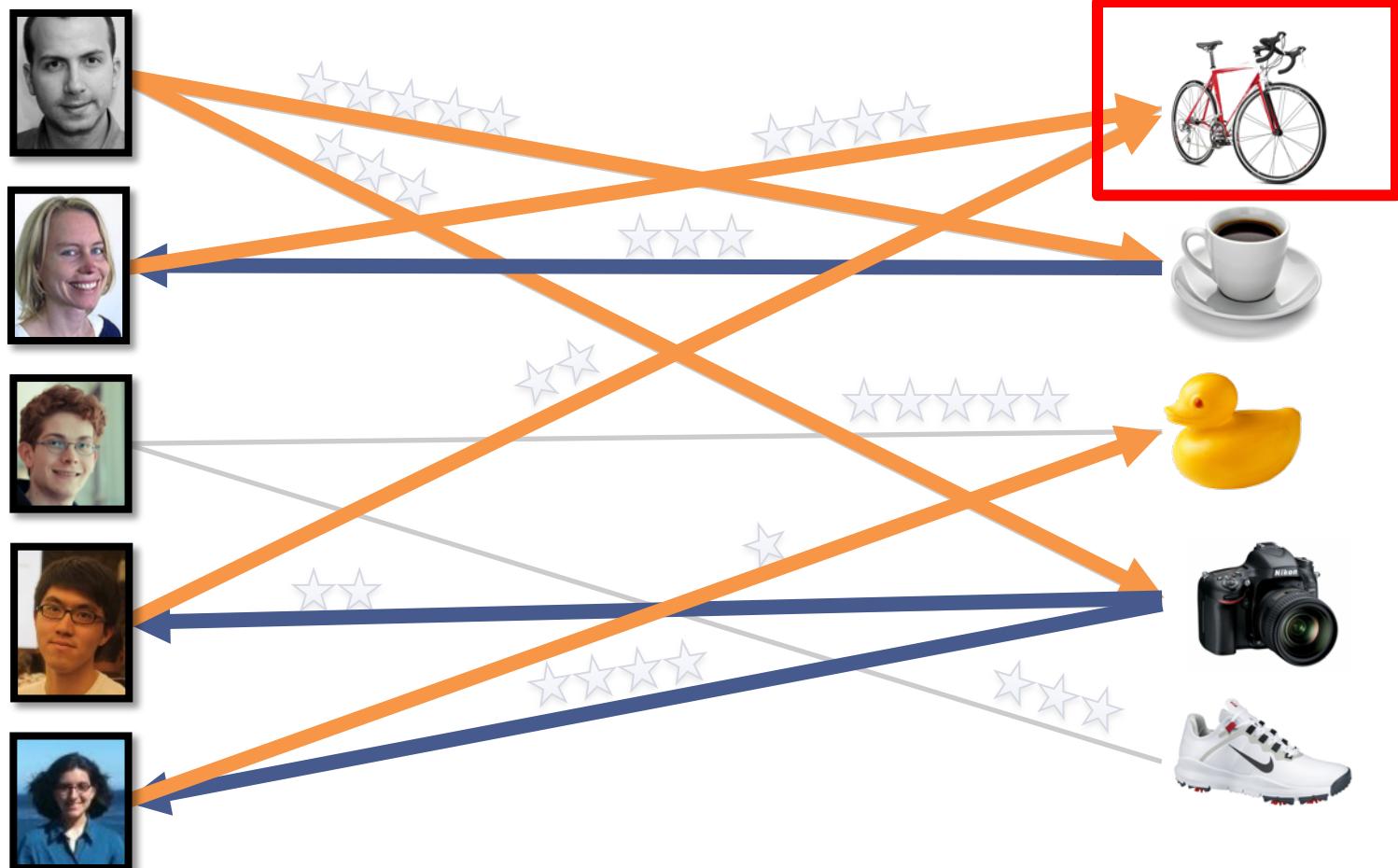
# Recommending Products



Users

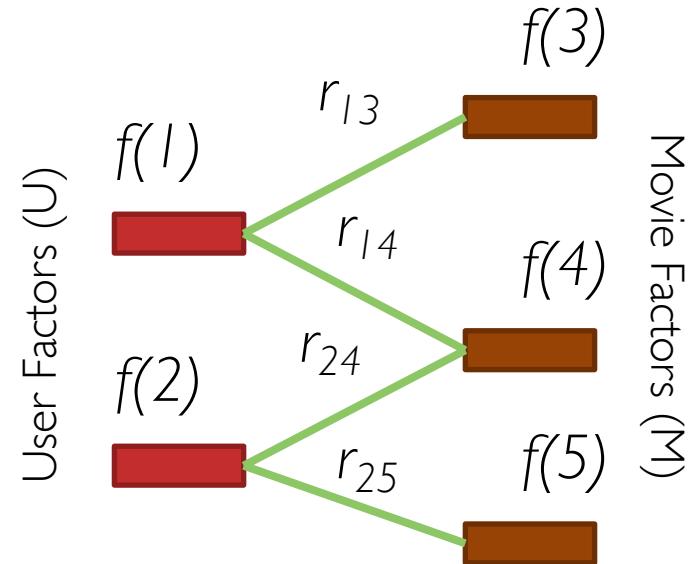
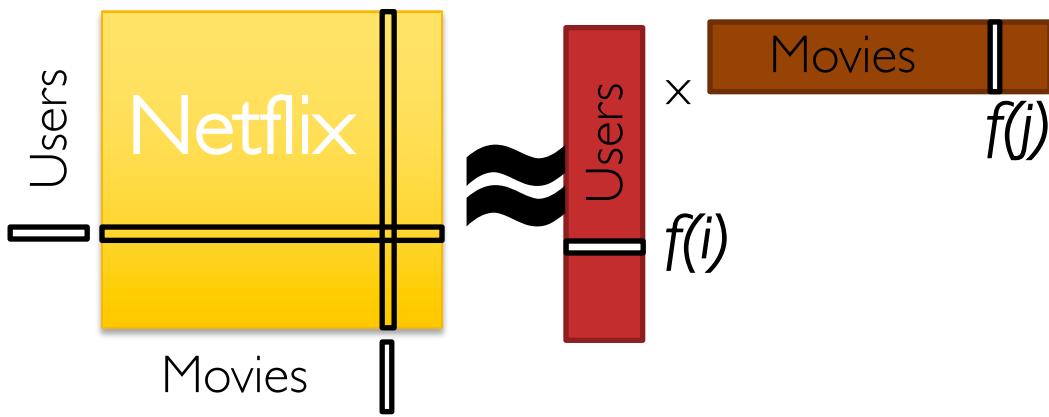
Ratings

Items



# Recommending Products

## Low-Rank Matrix Factorization:



Iterate:

$$f[i] = \arg \min_{w \in \mathbb{R}^d} \sum_{j \in \text{Nbrs}(i)} (r_{ij} - w^T f[j])^2 + \lambda \|w\|_2^2$$

Example of matrix factorization

$$A = \begin{bmatrix} A_1 & A_3 \\ \begin{matrix} 1 & 2 & 3 & 5 \\ 2 & 4 & 8 & 12 \\ 3 & 6 & 7 & 13 \end{matrix} \end{bmatrix} \quad X = \begin{bmatrix} 1 & 3 \\ 2 & 8 \\ 3 & 7 \end{bmatrix}, \quad Y = \begin{bmatrix} 1 & 2 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$A_1 = 1 \cdot A_1 + 0 \cdot A_3$$

$$A_2 = 2 \cdot A_1 + 0 \cdot A_3$$

$$A_3 = 0 \cdot A_1 + 1 \cdot A_3$$

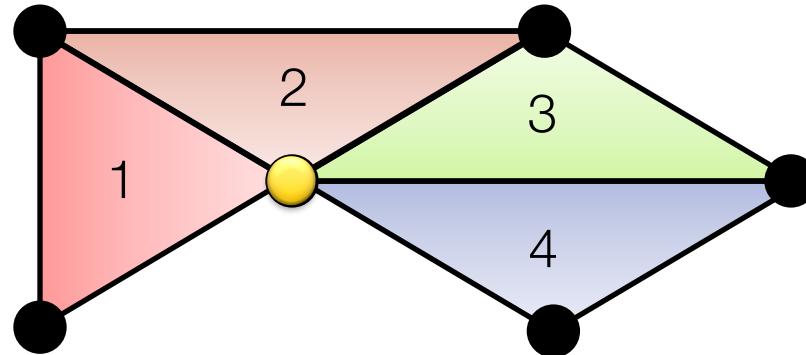
$$A_4 = 2 \cdot A_1 + 1 \cdot A_3$$

$$XY = \begin{bmatrix} 1 & 2 & 3 & 5 \\ 2 & 4 & 8 & 12 \\ 3 & 6 & 7 & 13 \end{bmatrix}$$



# Finding Communities

- Count triangles passing through each vertex:

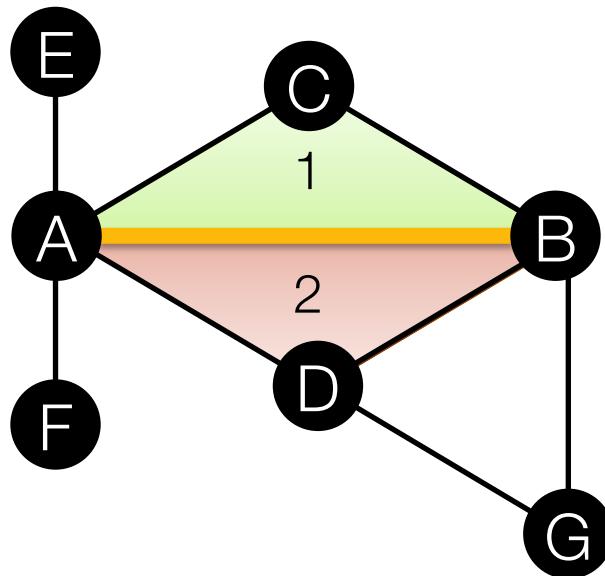


- Measure “cohesiveness” of local community

$$\text{ClusterCoeff}[i] = \frac{2 * \# \text{Triangles}[i]}{\text{Deg}[i] * (\text{Deg}[i] - 1)}$$

# Counting Triangles

- Count triangles passing through each vertex by counting triangles on each edge:

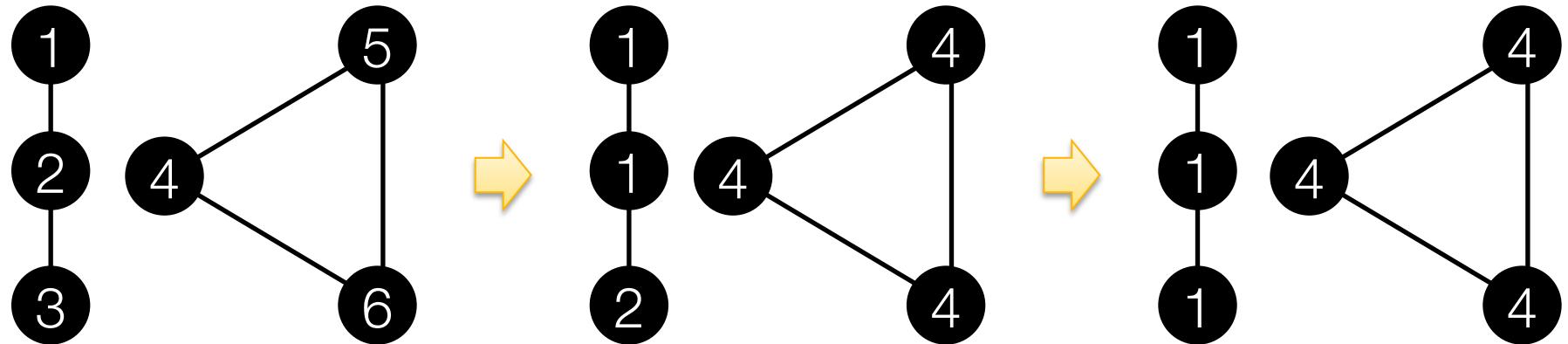


$$\left\{ \begin{array}{c} B \\ C \\ D \\ E \\ F \end{array} \right\} \cap \left\{ \begin{array}{c} A \\ C \\ D \\ G \end{array} \right\} = \left\{ \begin{array}{c} C \\ D \end{array} \right\}$$

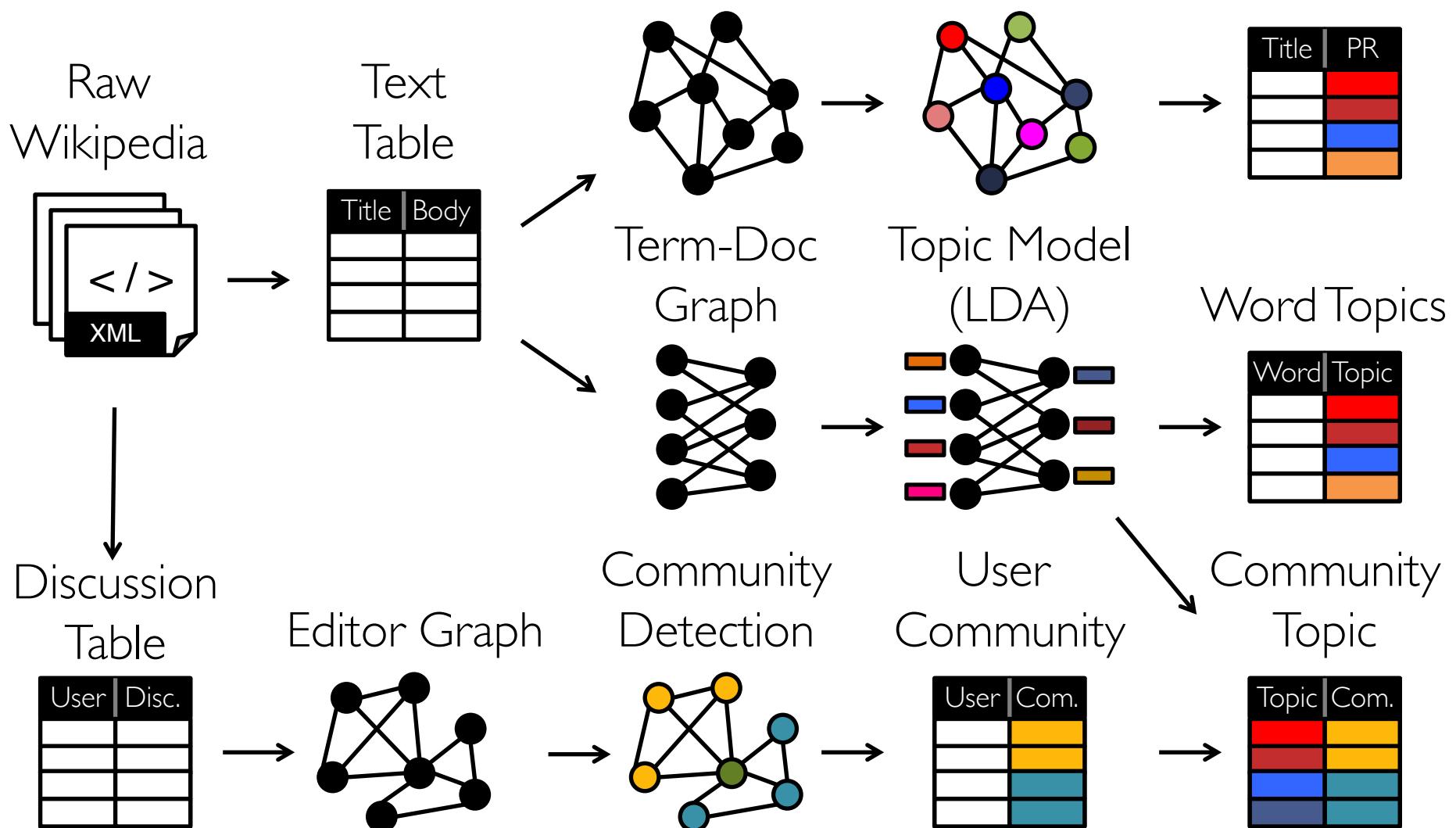
# Connected Components

- Every vertex starts out with a unique component id (typically it's vertex id):

$$CC[i] = \arg \min_{\{i,j\} \in E} CC[j]$$



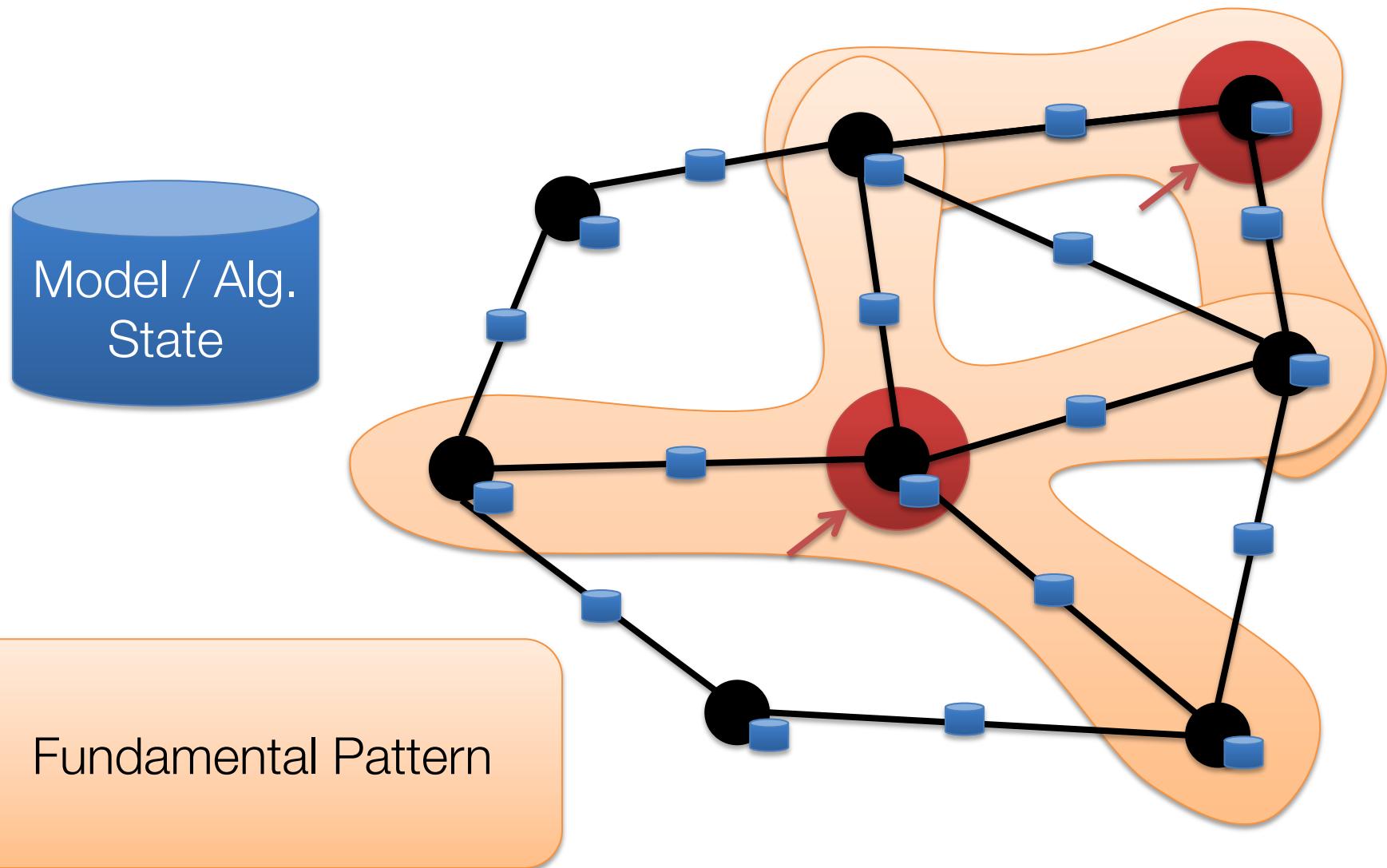
# Putting it All Together



# Many Other Graph 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

# The Graph-Parallel Pattern



# Graph-Parallel Systems

Pregel

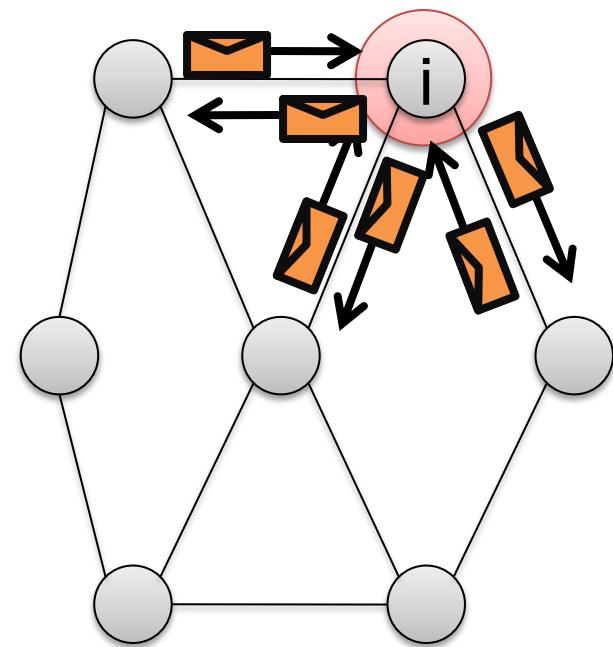


*Expose specialized APIs to simplify  
graph programming.*

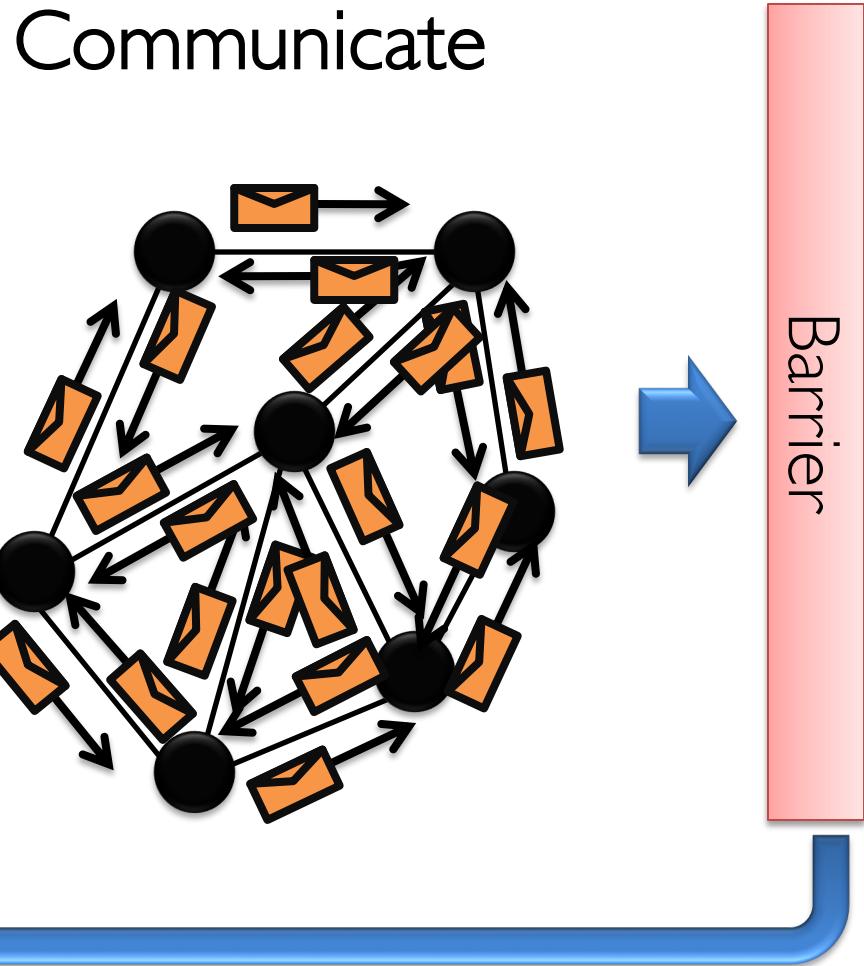
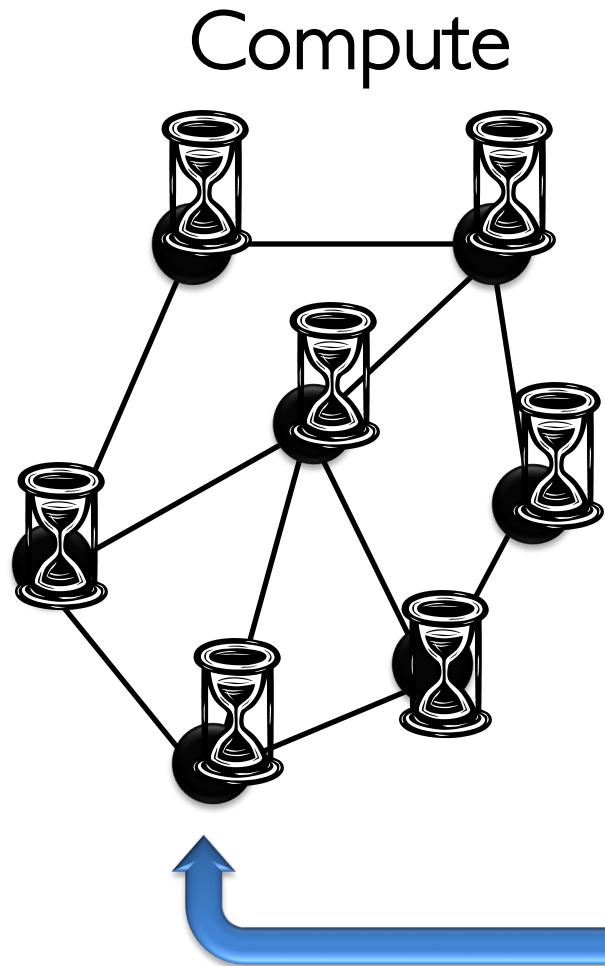
# The Vertex Program Abstraction

Vertex-Programs interact by sending messages.

```
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] = 0.15 + total  
  
    // Send new messages to neighbors  
    foreach(j in out_neighbors[i]) :  
        Send msg(R[i]) to vertex j
```



# Iterative Bulk Synchronous Execution



# Graph-Parallel Systems

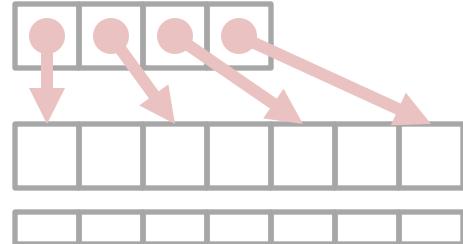
Pregel



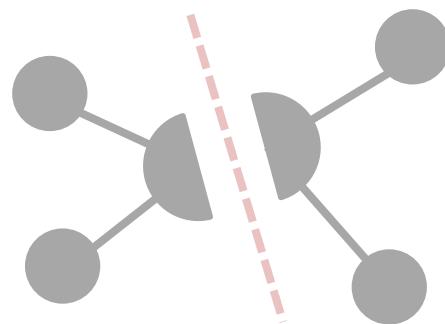
*Exploit graph structure to achieve  
orders-of-magnitude performance gains  
over more general data-parallel systems.*

# Graph System Optimizations

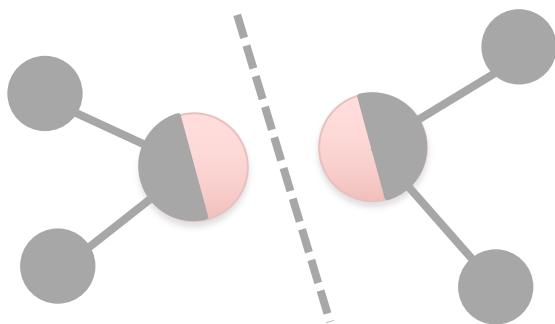
Specialized  
Data-Structures



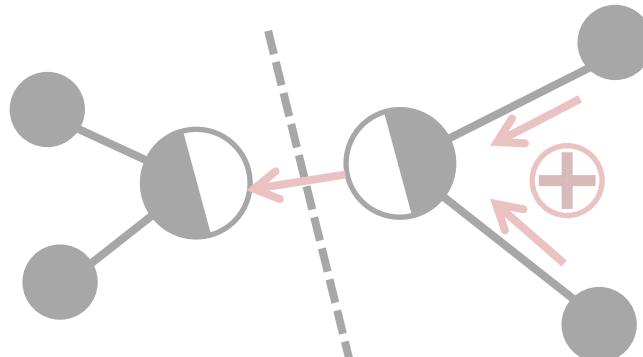
Vertex-Cuts  
Partitioning



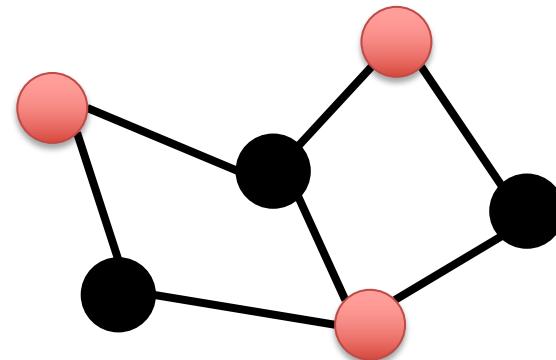
Remote  
Caching / Mirroring



Message Combiners



Active Set Tracking

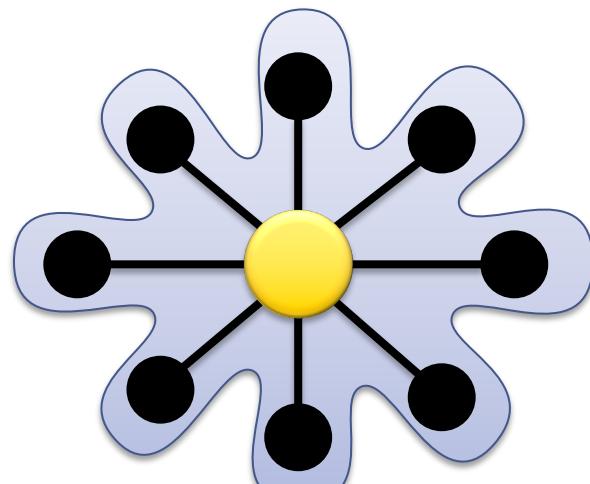


# GraphLab

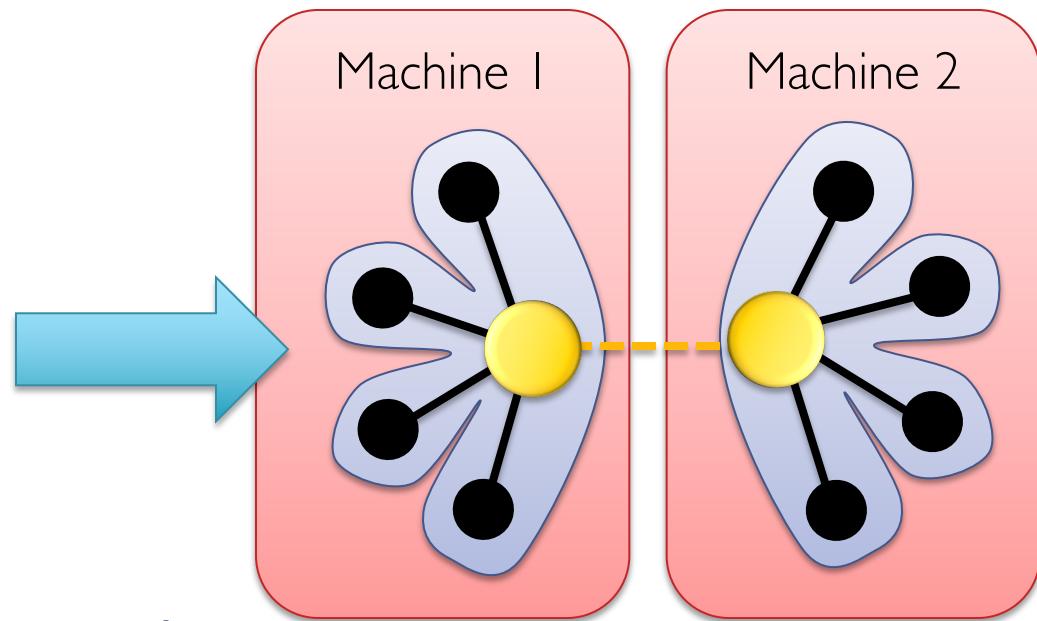


(PowerGraph, OSDI'12)

Program This



Run on This

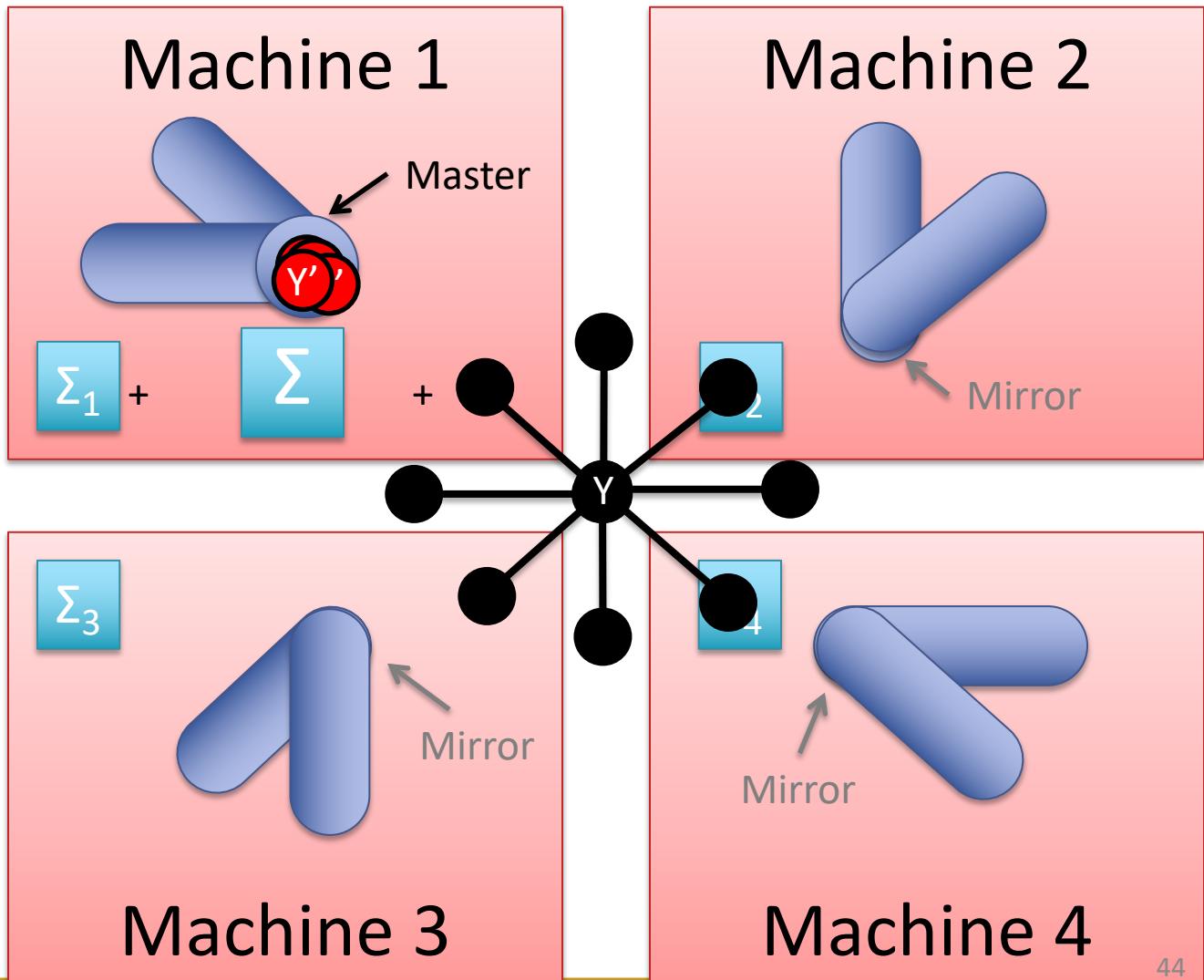


- Split High-Degree vertices
- New Abstraction → Equivalence on Split Vertices

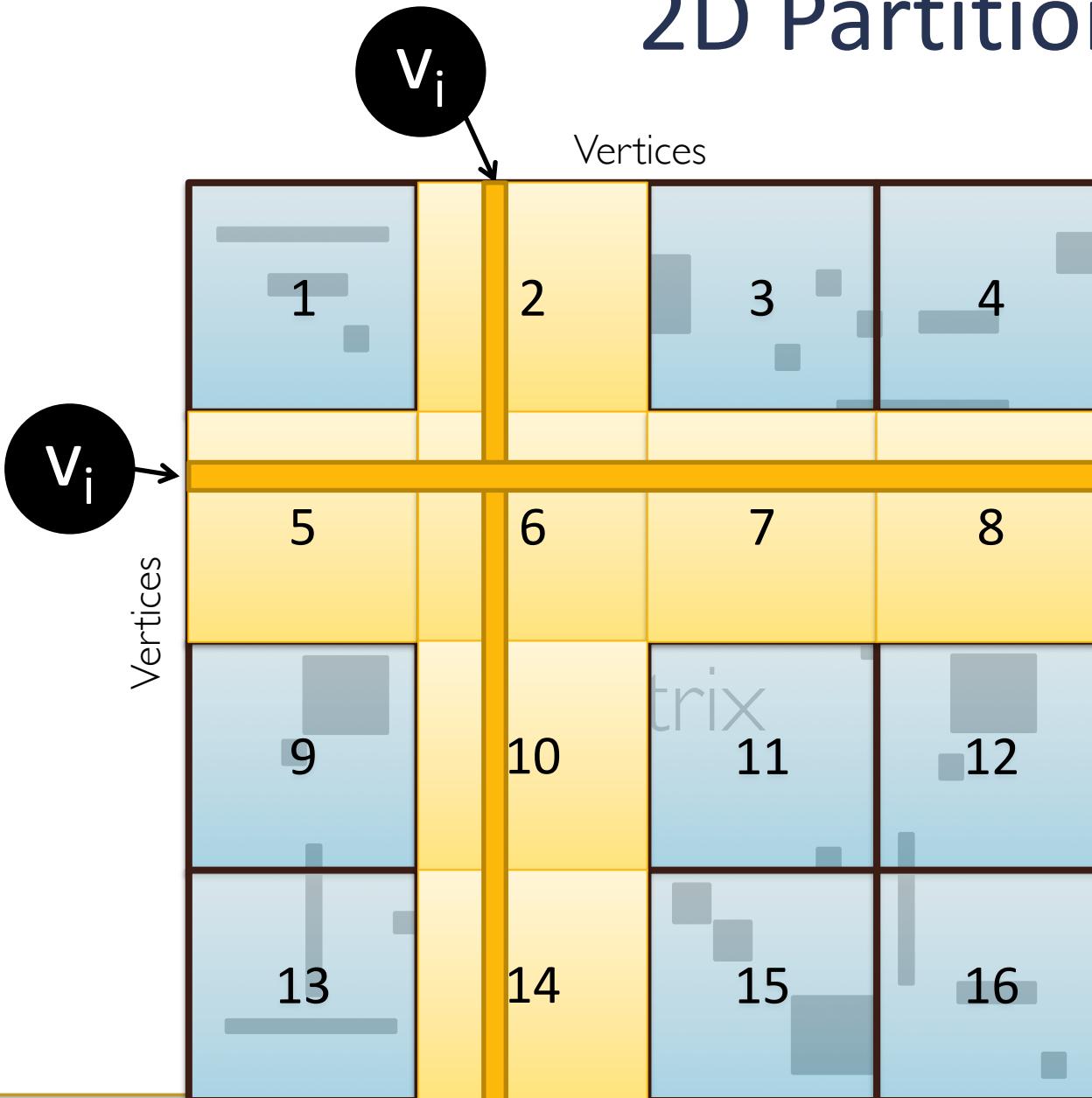
# GAS Decomposition



Gather  
Apply  
Scatter



# 2D Partitioning



16 Machines

$v_i$  only has  
neighbors on  
7 machines

# Triangle Counting on Twitter

40M Users, 1.4 Billion Links

Counted: 34.8 Billion Triangles

Hadoop

[WWW'11]

1536 Machines

423 Minutes

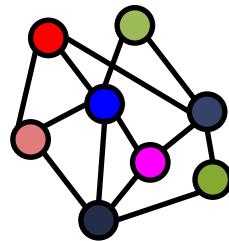
GraphLab

64 Machines

15 Seconds

1000 × Faster

# PageRank



# Tables

Raw  
Wikipedia



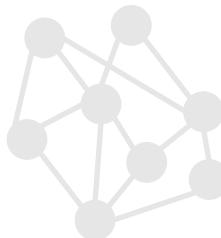
Text  
Table

Title	Body

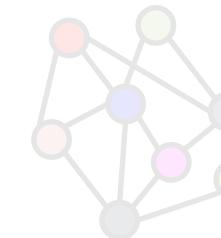
Discussion  
Table

User	Disc.

Hyperlinks



PageRank



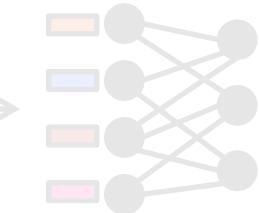
Top 20 Pages

Title	PR

Term-Doc  
Graph



Topic Model  
(LDA)



Word Topics

Word	Topic

Community  
Detection



User  
Community

User	Com.

Community  
Topic

Topic	Com.

Editor Graph

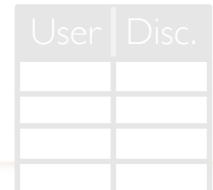


# Graphs

Raw  
Wikipedia



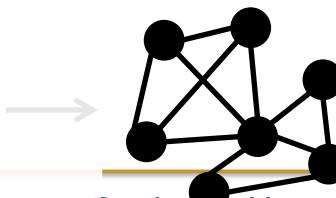
Discussion  
Table



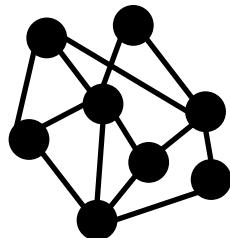
Text  
Table



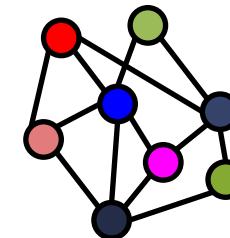
Editor Graph



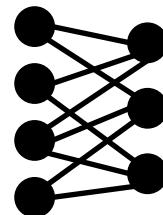
Hyperlinks



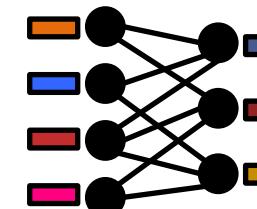
PageRank



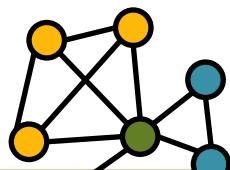
Term-Doc  
Graph



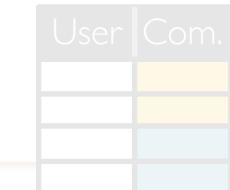
Topic Model  
(LDA)



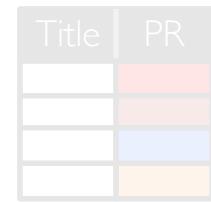
Community  
Detection



User  
Community



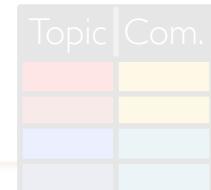
Top 20 Pages



Word Topics



Community  
Topic

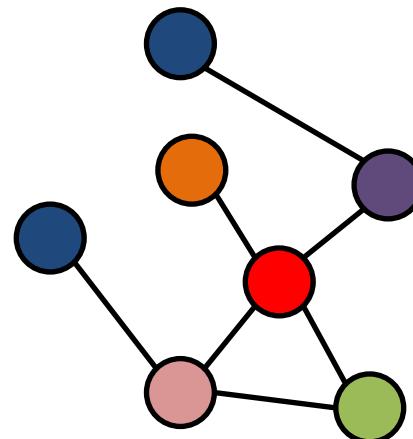


# Separate Systems

Tables

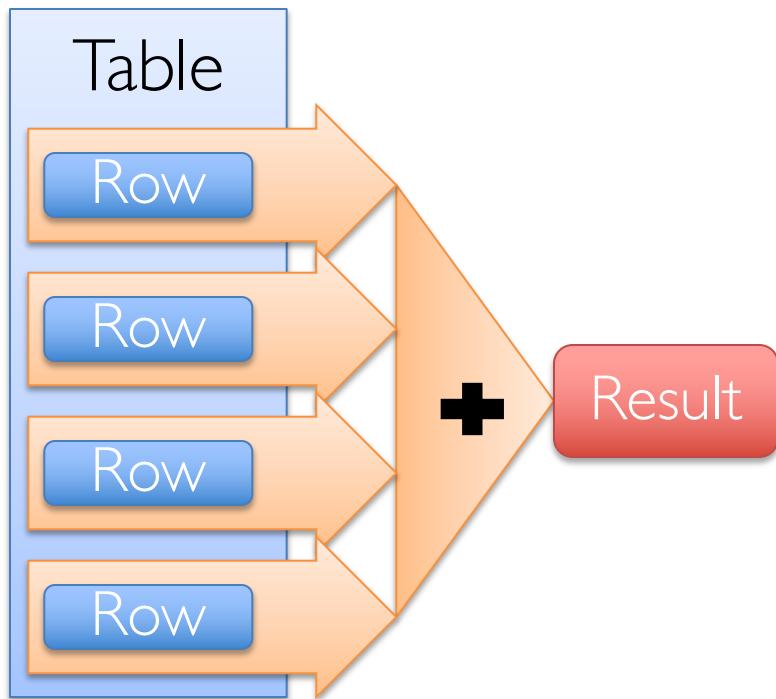


Graphs

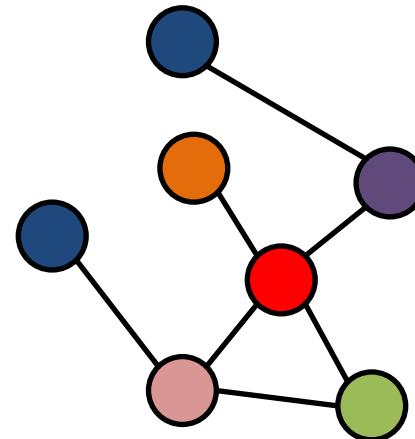


# Separate Systems

## Dataflow Systems

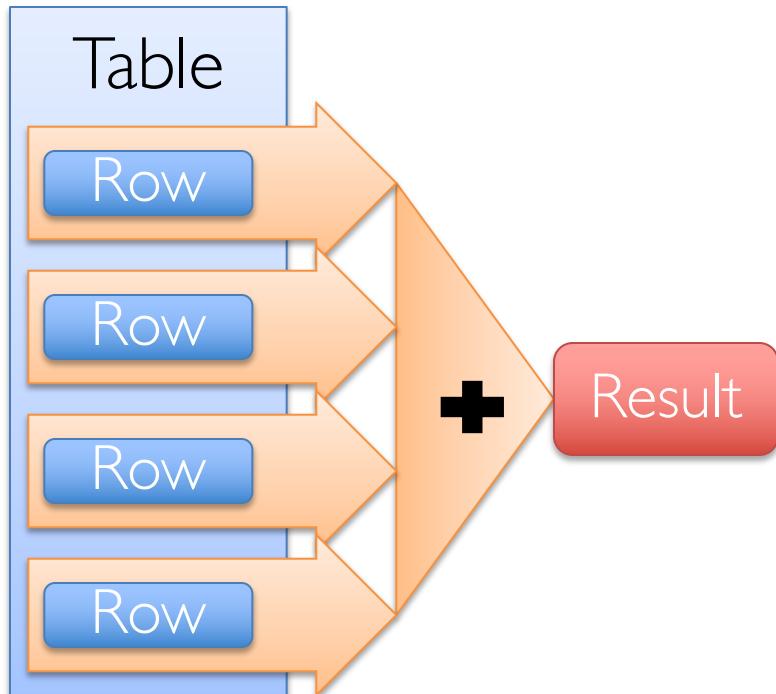


## Graphs



# Separate Systems

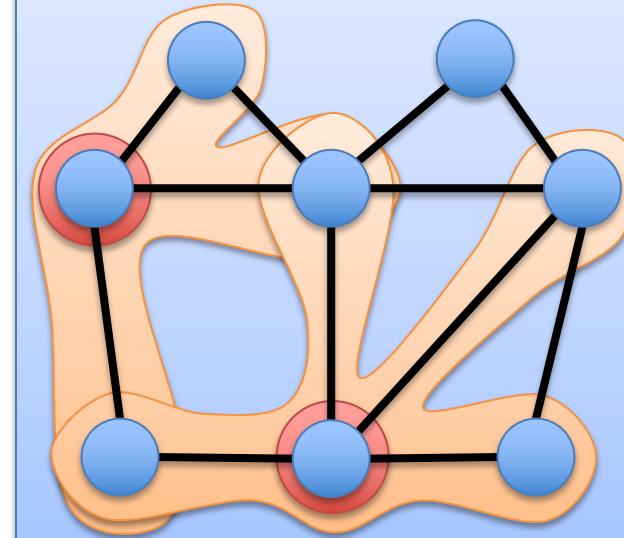
## Dataflow Systems



## Graph Systems



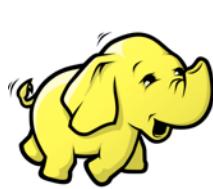
## Dependency Graph



Separate systems  
for each view can be  
*difficult to use and inefficient*

# Difficult to Program and Use

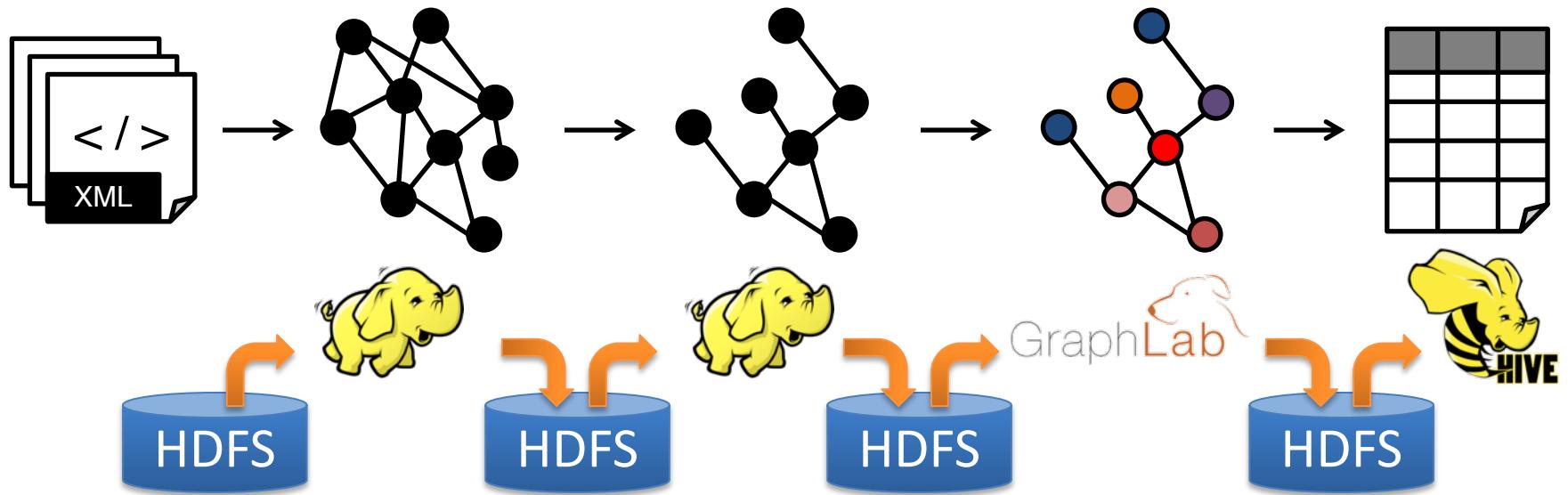
Users must *Learn*, *Deploy*, and *Manage* multiple systems



Leads to brittle and often complex interfaces

# Inefficient

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

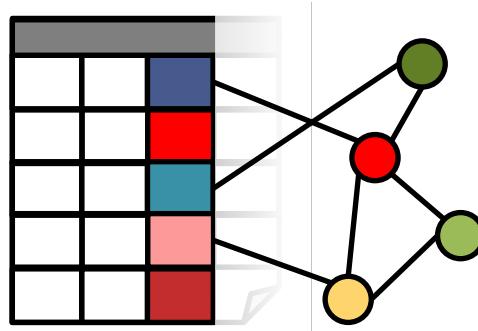


Limited reuse internal data-structures  
across stages

# The GraphX Unified Approach

## New API

*Blurs the distinction between  
Tables and Graphs*



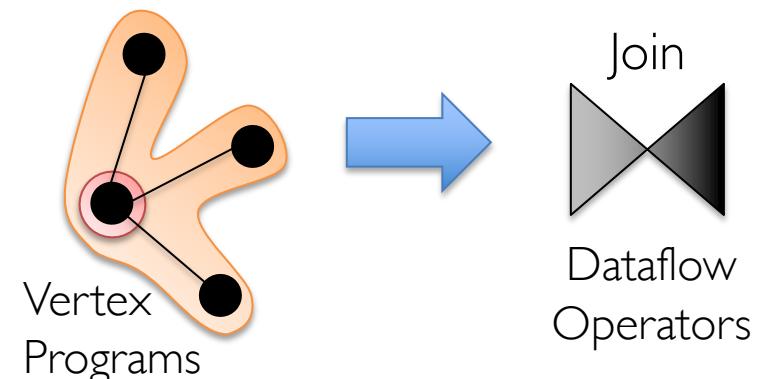
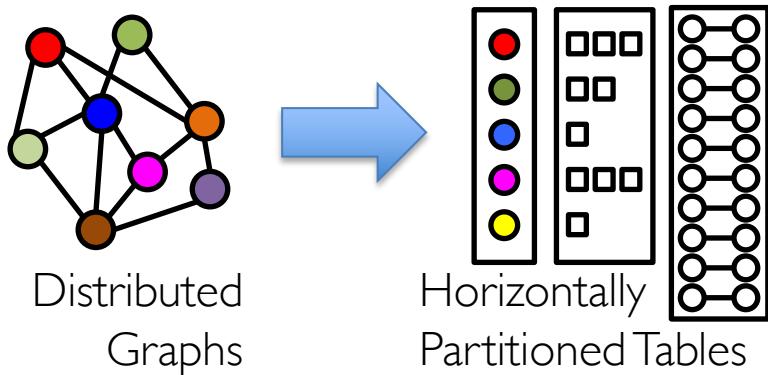
## New System

*Combines Data-Parallel  
Graph-Parallel Systems*



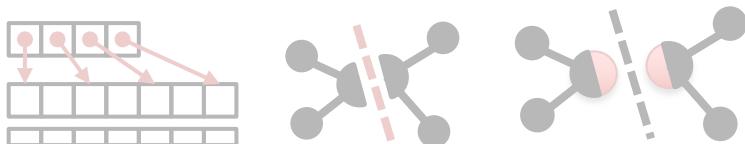
Enabling users to **easily** and **efficiently**  
express the entire graph analytics pipeline

# Representation

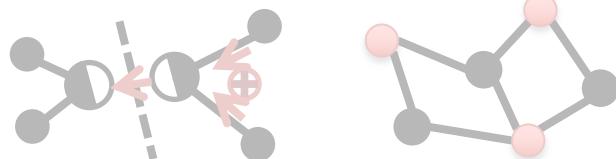


# Optimizations

Advances in Graph Processing Systems



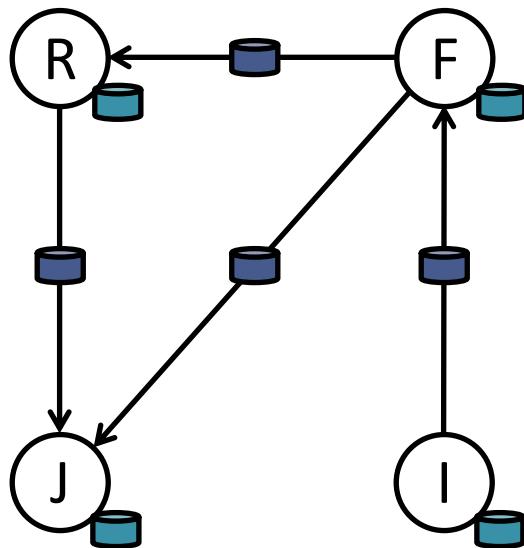
Distributed Join  
Optimization



Materialized View  
Maintenance

# View a Graph as a Table

## Property Graph



Vertex Property Table

<b>Id</b>	<b>Property (V)</b>
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Edge Property Table

<b>SrcId</b>	<b>DstId</b>	<b>Property (E)</b>
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

# Spark 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 (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) => 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]  
}
```

# Graph Operators (Scala)

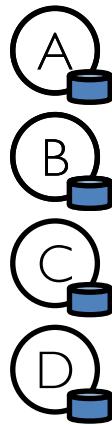
```
class Graph [ V, E ] {  
    def Graph(vertices: Table[ (Id, V) ],  
              edges: Table[ (Id, Id, E) ])  
    // Table views -----  
    def vertices: Table[ (Id, V) ]  
    def edges: Table[ (Id, Id, E) ]  
    def triplets: Table [ ((Id, V), (Id, V), E) ]  
    // Transformations -----  
    def reverse: Graph[V, E]  
    def subgraph(pV: (Id, V) => Boolean,  
                pE: Edge[V, E] => Boolean): Graph[V, E]  
    def mapV(m: (Id, V) => T): Graph[T, E]  
    def mapE(m: (Id, Id, E) => T): Graph[V, T]  
    // Joins -----  
    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]  
}
```

capture the *Gather-Scatter* pattern from specialized graph-processing systems

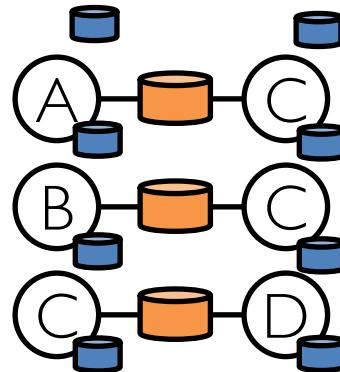
# Triplets Join Vertices and Edges

The *triplets* operator joins vertices and edges:

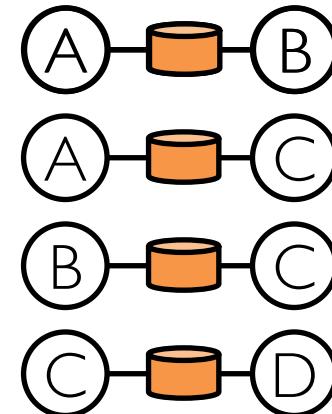
Vertices



Triplets

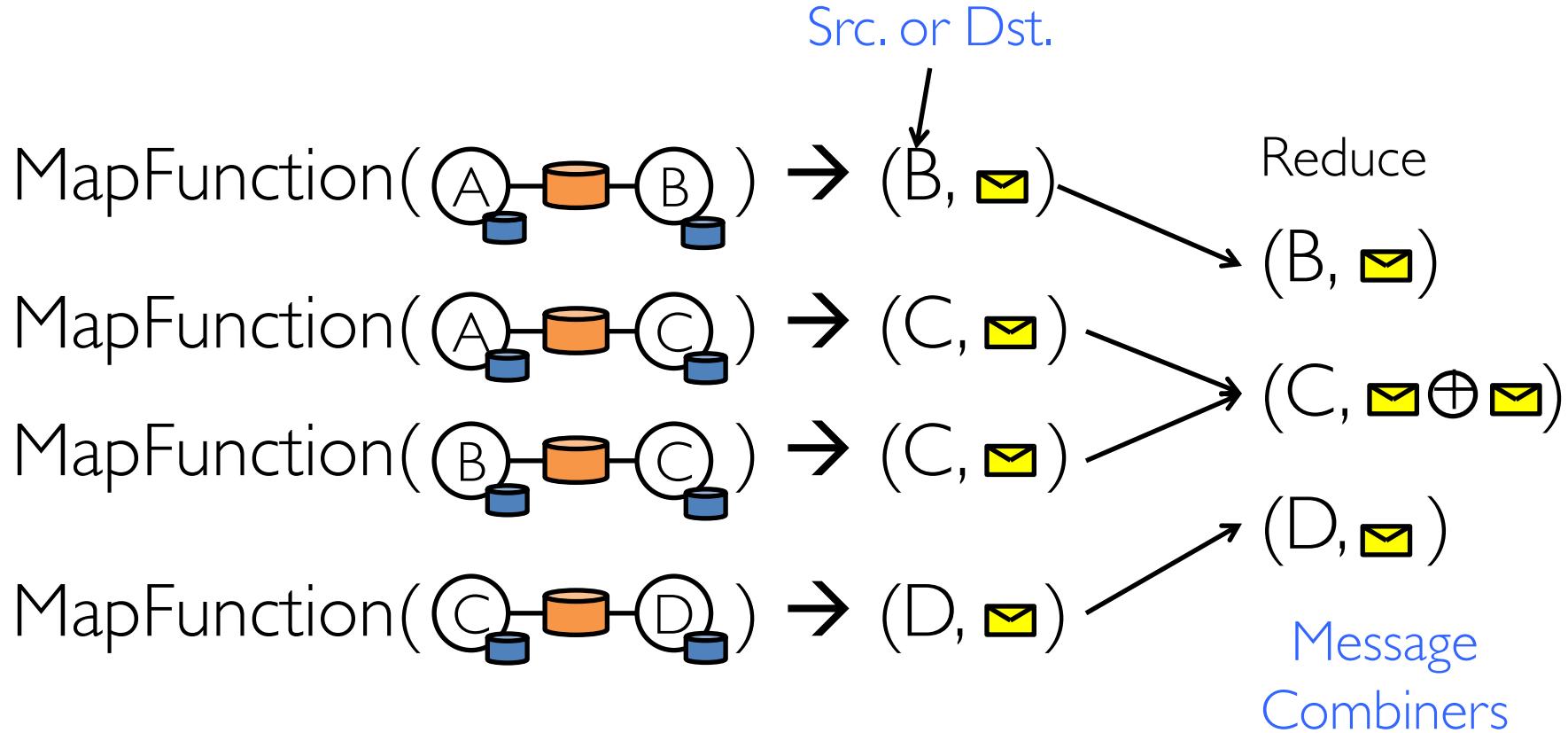


Edges



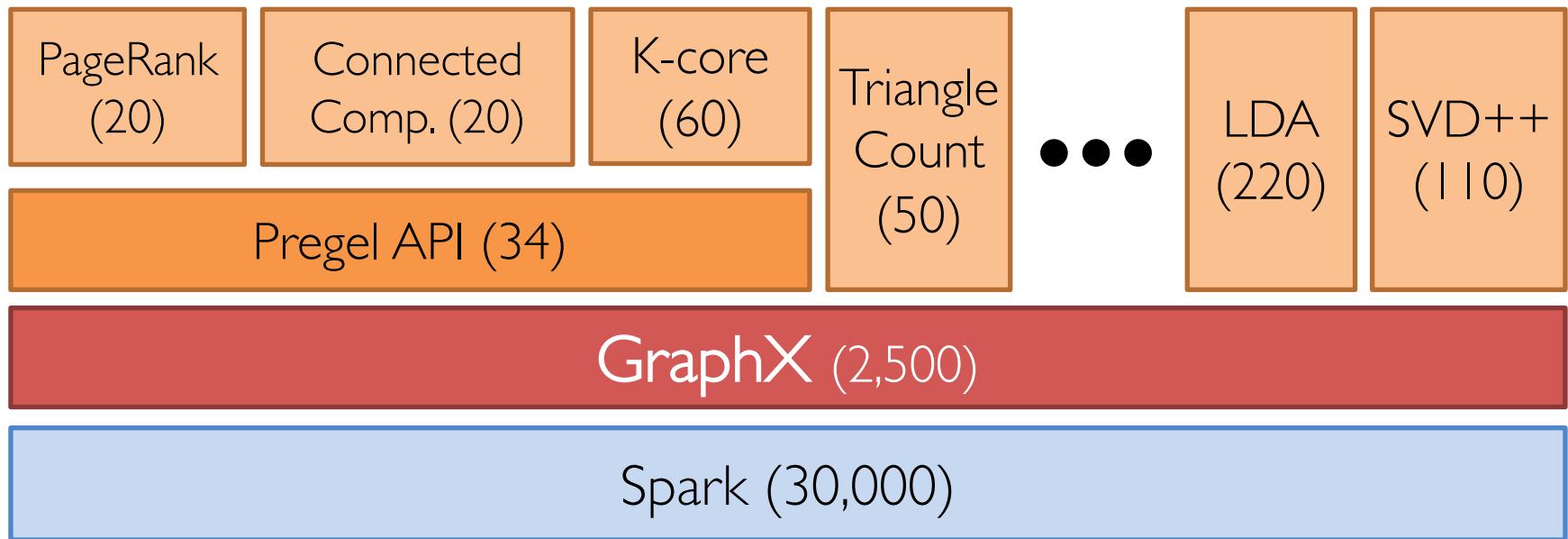
# Map-Reduce Triplets

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



Using these basic GraphX operators  
we implemented Pregel and GraphLab  
in under 50 lines of code!

# The GraphX Stack (Lines of Code)



Some algorithms are more naturally expressed using the GraphX primitive operators

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

# Enhanced Pregel in GraphX



```
pregelPR(i, me messageSum
```

Require Message  
Combiners

```
// Receive all the messages  
total = 0 messageSum  
foreach( msg in messageList ) :  
    total = total + msg
```

```
// Update the rank of this vertex  
R[i] = 0.15 + total  
combineMsg(a, b):  
// Compute sum of two messages  
sendMsg(j, newMessages, toNeighbors)  
foreach( a in b.outNeighbors[i] ) :  
    // Compute single message  
    Send msg(R[i]/E[i,j]) to vertex  
return msg(R[i]/E[i,j])
```

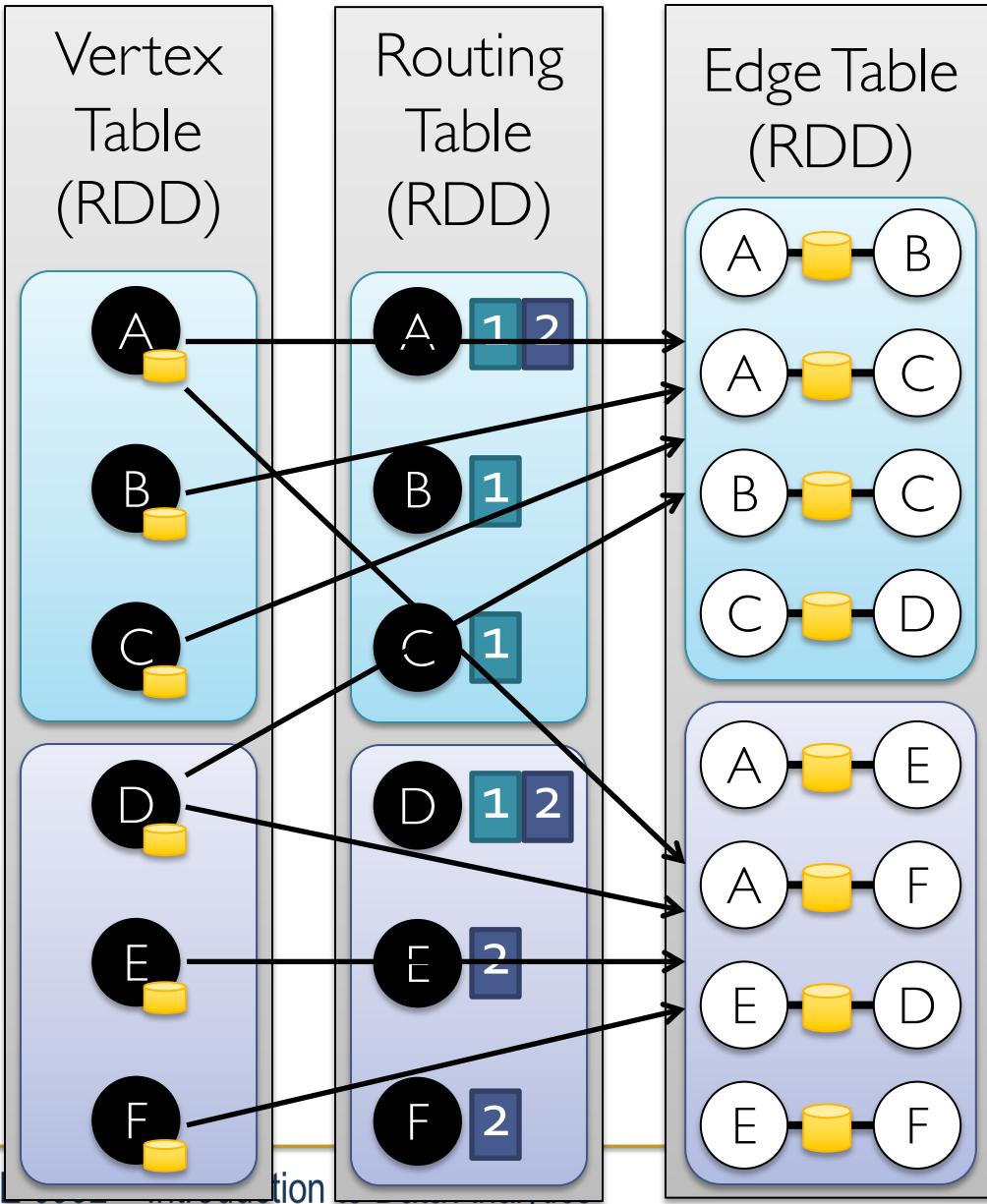
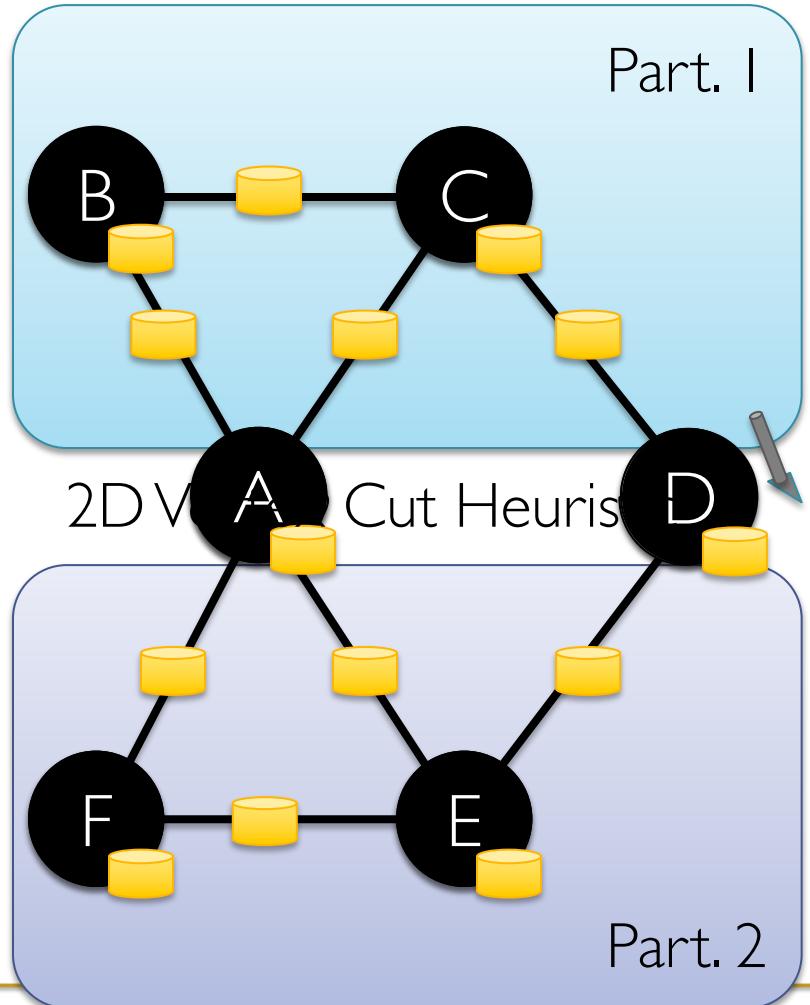
Remove Message  
Computation  
from the  
Vertex Program

# GraphX System Design

# Distributed Graphs as Tables (RDDs)



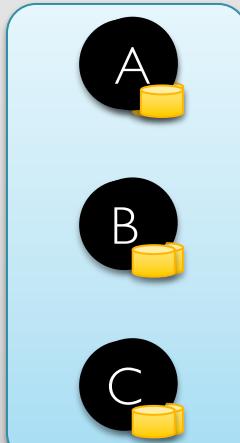
Property Graph



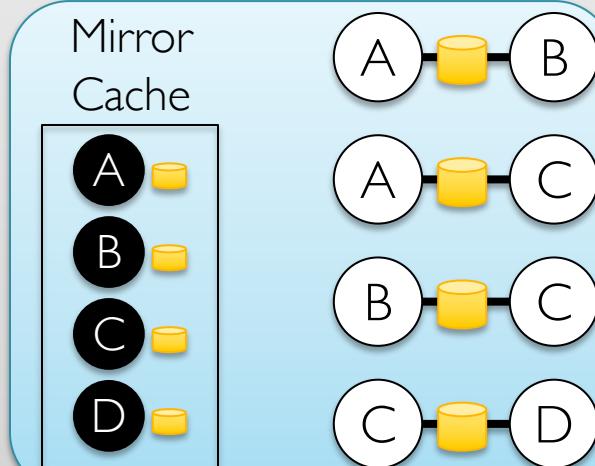
# Caching for Iterative mrTriplets

GW

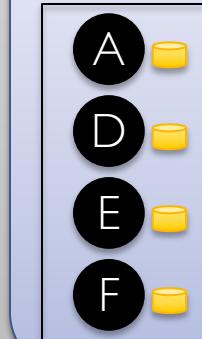
Vertex  
Table  
(RDD)



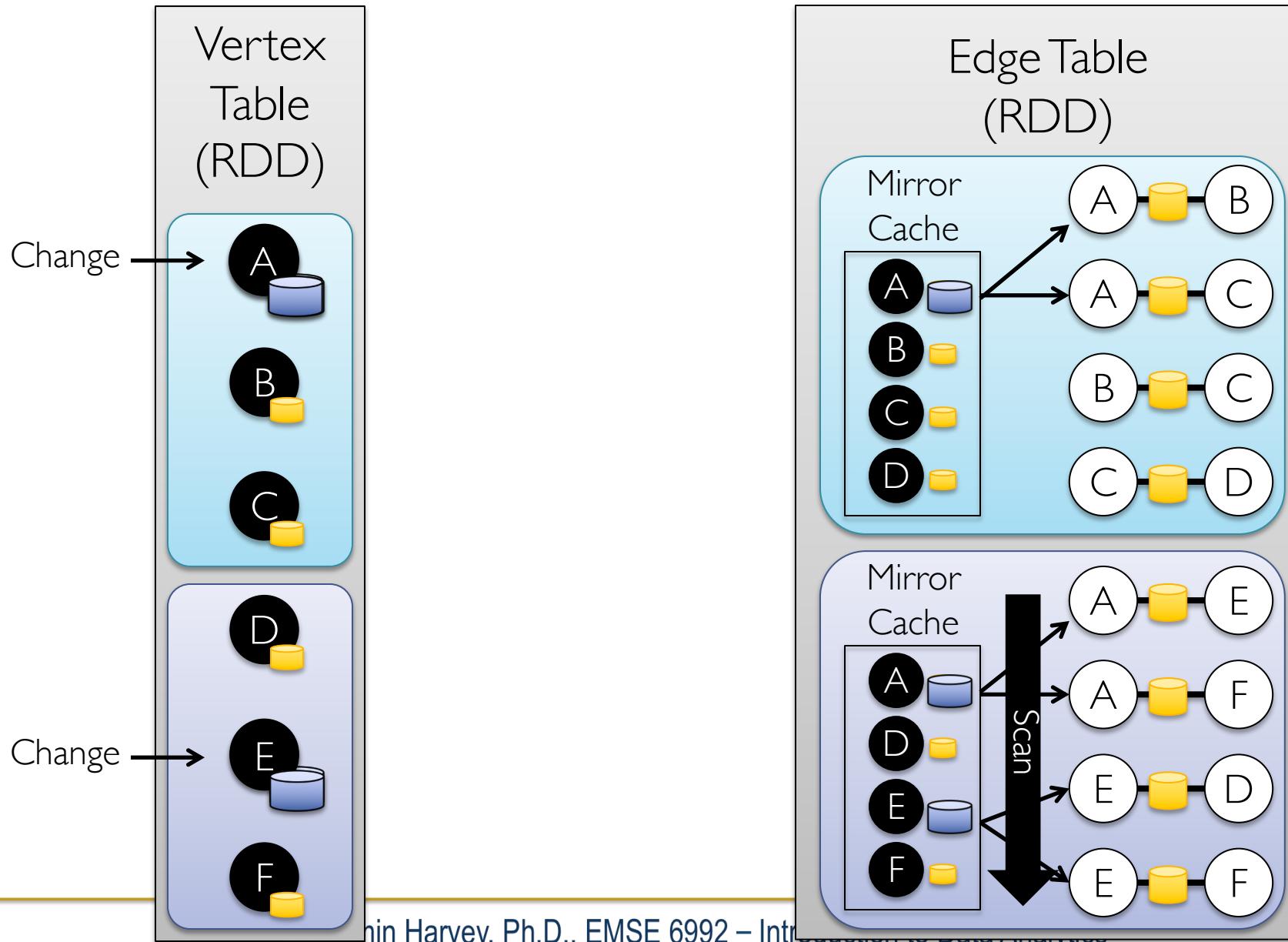
Edge Table  
(RDD)



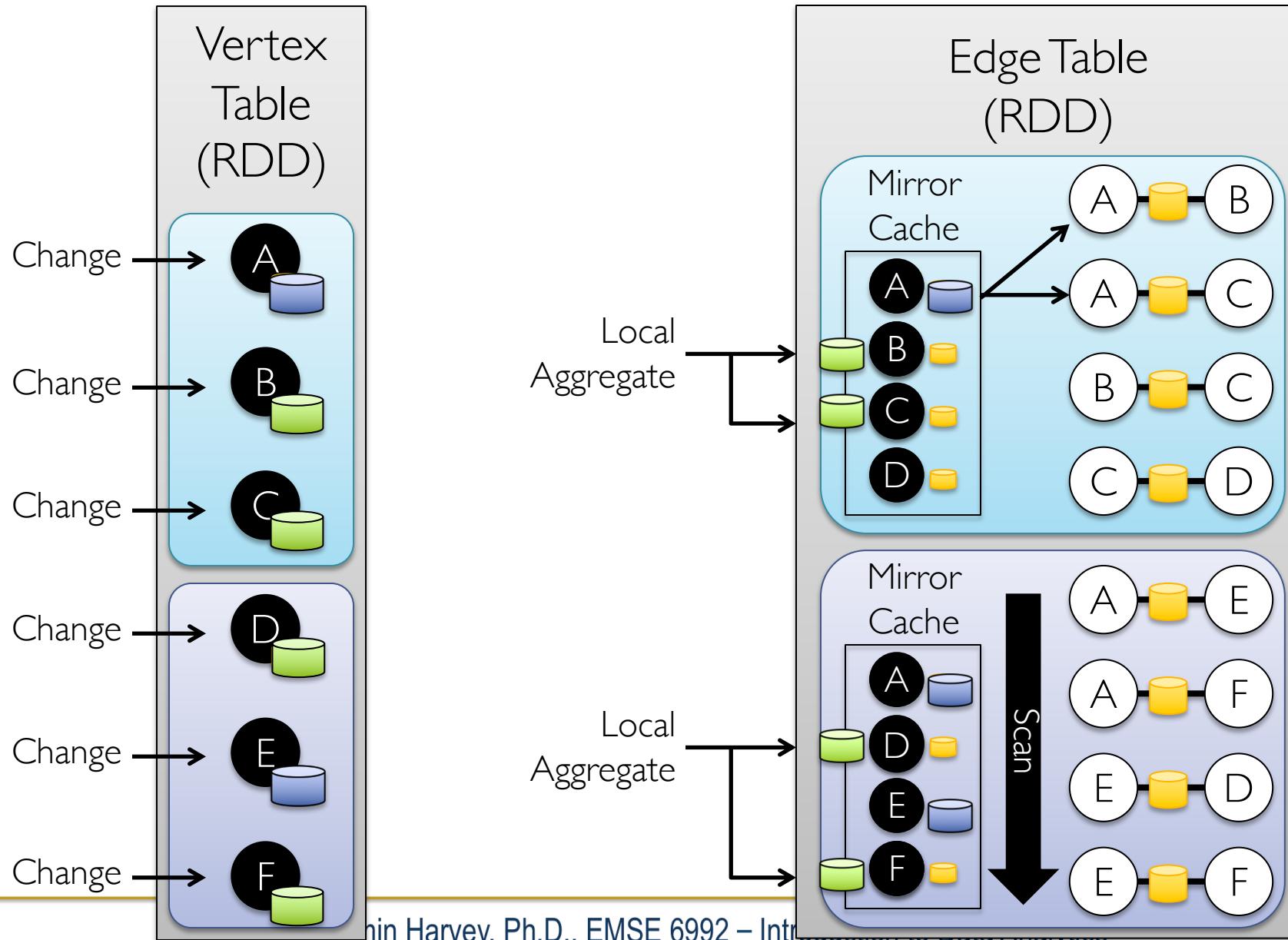
Mirror  
Cache



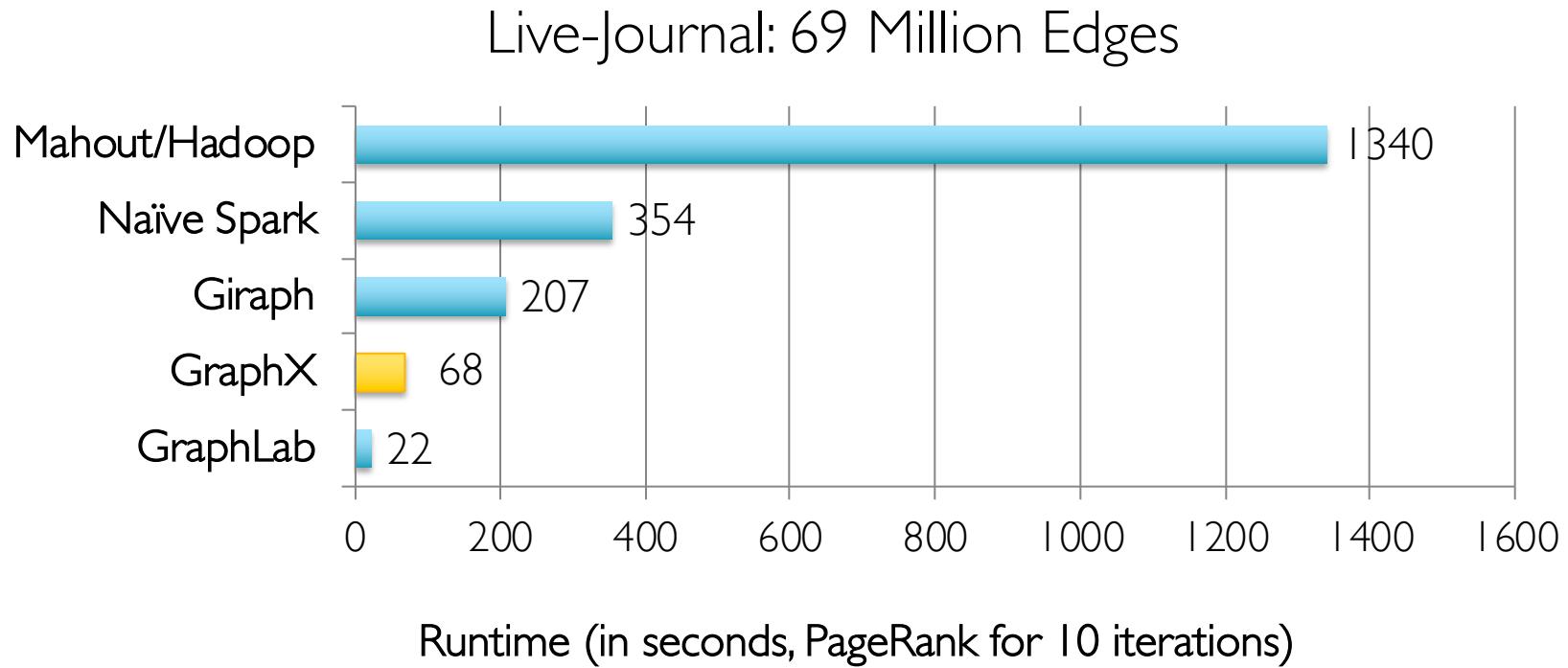
# Incremental Updates for Iterative mrTriplets



# Aggregation for Iterative mrTriplets

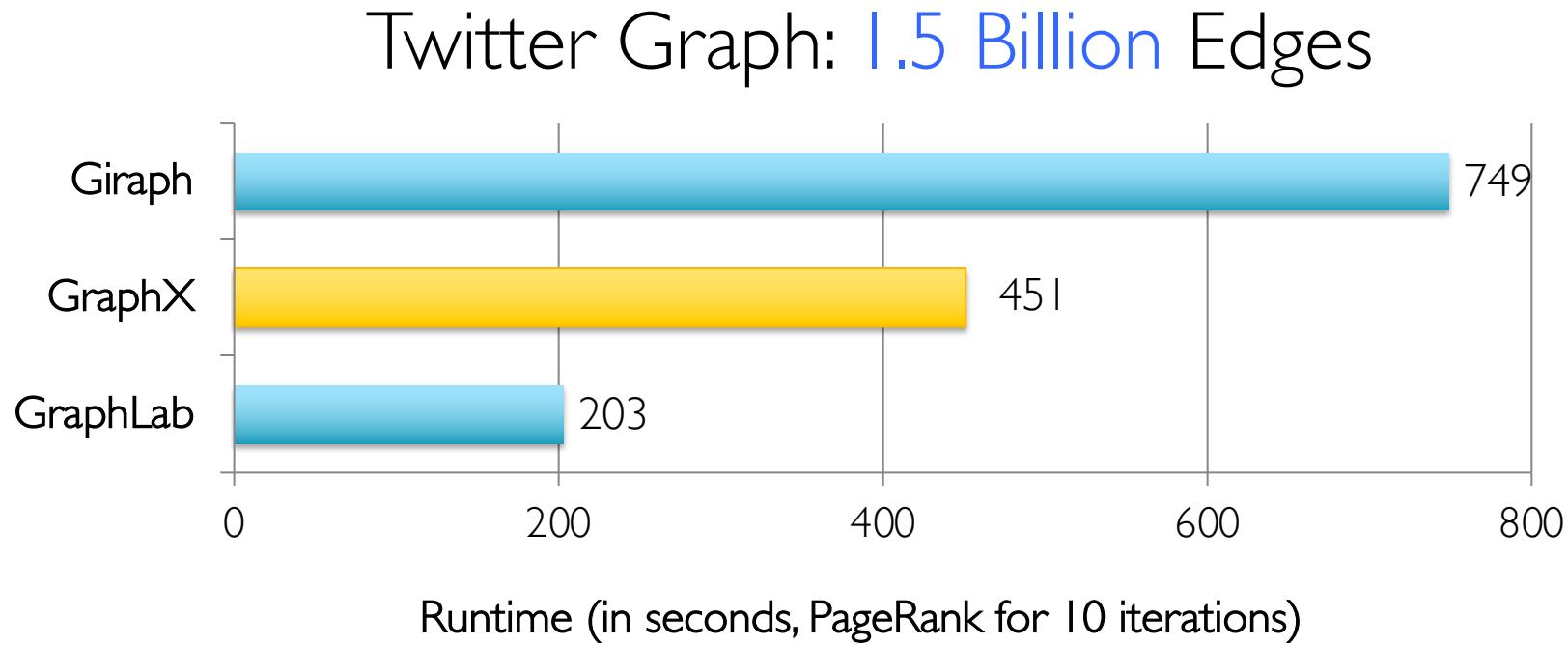


# Performance Comparisons



GraphX is roughly *3x slower* than GraphLab

# GraphX scales to larger graphs



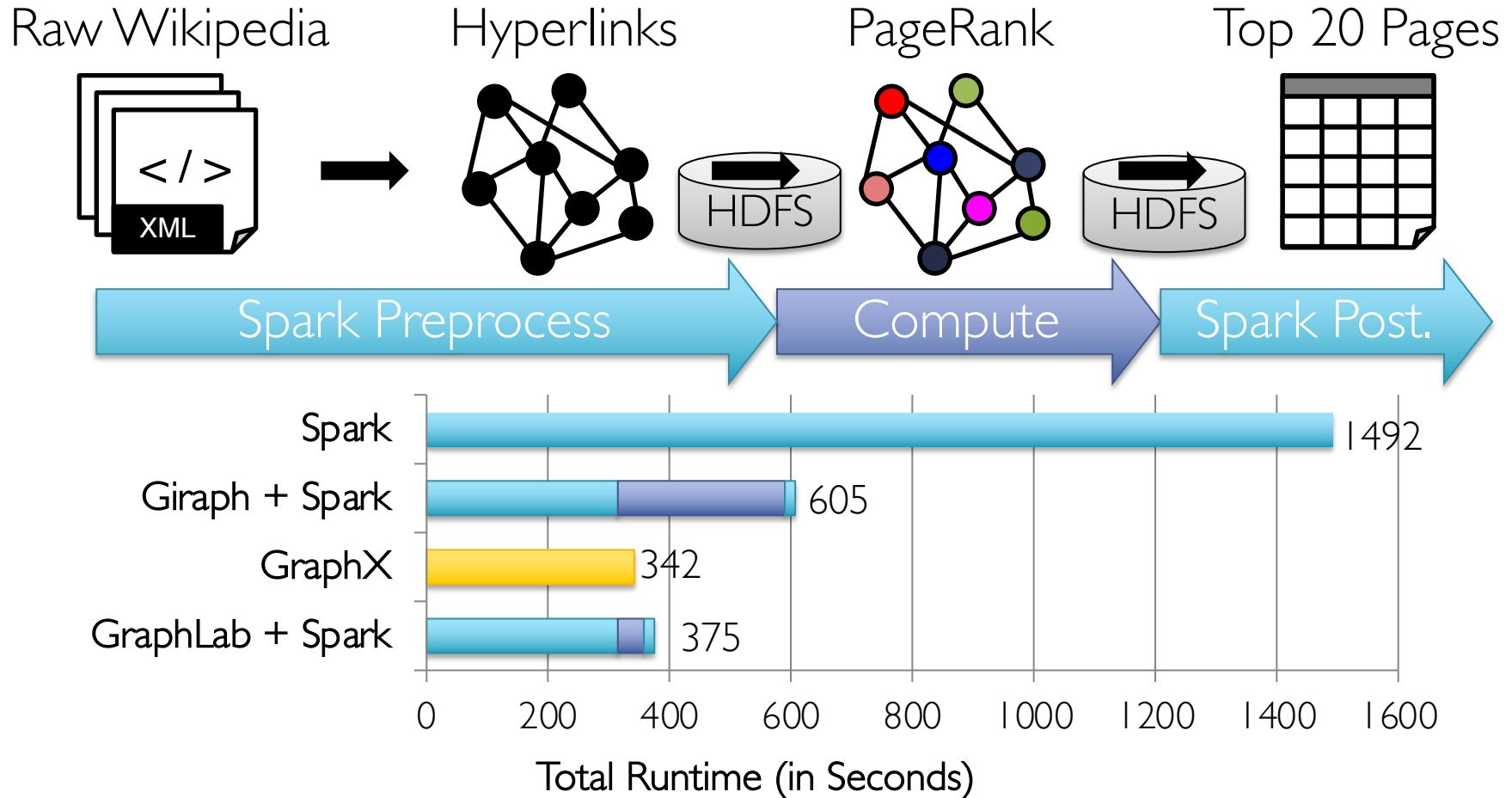
GraphX is roughly **2x slower** than GraphLab

- » Scala + Java overhead: Lambdas, GC time, ...
- » No shared memory parallelism: **2x increase** in comm.

PageRank is just one stage....

What about a pipeline?

# A Small Pipeline in GraphX



Timed end-to-end GraphX is *faster* than GraphLab

# Open Source Project



- Alpha release since Spark 0.9

The screenshot shows a web browser displaying the "GraphX Programming Guide – Spark 0.9.0 Documentation" at [spark.incubator.apache.org/docs/latest/graphx-programming-guide.html](http://spark.incubator.apache.org/docs/latest/graphx-programming-guide.html). The page features the Spark 0.9.0 logo and navigation links for Overview, Programming Guides, API Docs, Deploying, and More. A large graphic on the left illustrates a graph with colored nodes (purple, red, green, pink) connected by edges, adjacent to a grid of colored squares (orange, red, blue, green). The word "GraphX" is written in a large, stylized, italicized font next to the graphic. Below the graphic, the "Overview" section is visible, providing a brief introduction to GraphX and its Resilient Distributed Property Graph API. The "Background on Graph-Parallel Computation" section follows, comparing Data-Parallel systems like Hadoop and Spark with Graph-Parallel systems like Pregel, GraphLab, and Giraph.

**Overview**

GraphX is the new (alpha) Spark API for graphs and graph-parallel computation. At a high-level, GraphX extends the Spark [RDD](#) by introducing the [Resilient Distributed Property Graph](#): a directed multigraph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., `subgraph`, `joinVertices`, and `mapReduceTriplets`) as well as an optimized variant of the [Pregel](#) API. In addition, GraphX includes a growing collection of graph [algorithms](#) and [builders](#) to simplify graph analytics tasks.

**Background on Graph-Parallel Computation**

From social networks to language modeling, the growing scale and importance of graph data has driven the development of numerous new *graph-parallel* systems (e.g., [Giraph](#) and [GraphLab](#)). By restricting the types of computation that can be expressed and introducing new techniques to partition and distribute graphs, these systems can efficiently execute sophisticated graph algorithms orders of magnitude faster than more general *data-parallel* systems.

**Data-Parallel**

**Graph-Parallel**

- Contrik

# Graph Processing Systems



- Apache Giraph: java Pregel implementation
- GraphLab.org: C++ GraphLab implementation
- NetworkX: python API for small graphs
- GraphLab Create: commercial GraphLab python framework for large graphs and ML
- Gephi: graph visualization framework

# Graph Database Technologies

- Property graph data-model for storing and retrieving graph structured data.
- [Neo4j](#): popular commercial graph database
- [Titan](#): open-source distributed graph database

# About Scala

- High-level language for the Java VM
  - Object-oriented + functional programming
- Statically typed
  - Comparable in speed to Java
  - But often no need to write types due to type inference
- Interoperates with Java
  - Can use any Java class, inherit from it, etc; can also call Scala code from Java

# Quick Tour

- Declaring variables:
- `var x: Int = 7`  
`var x = 7 // type inferred`
- `val y = "hi" // read-only`

## Functions:

```
def square(x: Int): Int = x*x  
  
def min(a:Int, b:Int): Int = {  
    if (a < b) a else b  
}  
  
def announce(text: String) {  
    println(text)  
}
```

- Java equivalent:
- `int x = 7;`
- `final String y = "hi";`

## Java equivalent:

```
int square(int x) {  
    return x*x;  
}
```

```
void announce(String text) {  
    System.out.println(text);  
}
```

# Quick Tour

- Generic types:
- `var arr = new Array[Int](8)`

```
var lst = List(1, 2, 3)
// type of lst is
List[Int]
```

- Java equivalent:
- `int[] arr = new int[8];`
- `List<Integer> lst = new
ArrayList<Integer>();
lst.add(...)`

## Indexing:

```
arr(5) = 7
println(lst(5))
```

## Java equivalent:

```
arr[5] = 7;
System.out.println(lst.get(5));
```

# Quick Tour

- Processing collections with functional programming:

Function expression (closure)

- `val list = List(1, 2, 3)`
- `list.foreach(x => println(x)) // prints 1, 2, 3`  
`list.foreach(println) // same`
- `list.map(x => x + 2) // => List(3, 4, 5)`  
`list.map(_ + 2) // same, with placeholder notation`
- `list.filter(x => x % 2 == 1) // => List(1, 3)`  
`list.filter(_ % 2 == 1) // => List(1, 3)`
- `list.reduce((x, y) => x + y) // => 6`  
`list.reduce(_ + _) // => 6`

All of these leave the list unchanged (List is immutable)

# Other Collection Methods



- Scala collections provide many other functional methods; for example, Google for “Scala Seq”

Method on Seq[T]	Explanation
<code>map(f: T =&gt; U): Seq[U]</code>	Pass each element through f
<code>flatMap(f: T =&gt; Seq[U]): Seq[U]</code>	One-to-many map
<code>filter(f: T =&gt; Boolean): Seq[T]</code>	Keep elements passing f
<code>exists(f: T =&gt; Boolean): Boolean</code>	True if one element passes
<code>forall(f: T =&gt; Boolean): Boolean</code>	True if all elements pass
<code>reduce(f: (T, T) =&gt; T): T</code>	Merge elements using f
<code>groupBy(f: T =&gt; K): Map[K, List[T]]</code>	Group elements by f(element)
<code>sortBy(f: T =&gt; K): Seq[T]</code>	Sort elements by f(element)
...	