IEMS 5730 and IERG4330

Analyzing Massive Graphs and Graph-based Big Learning Platforms

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - "Data-Intensive Information Processing Applications," by Jimmy Lin, University of Maryland.



This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States. See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details

  - CS246 Mining Massive Data-sets, by Jure Leskovec, Stanford University.
  - Introduction to Advanced Computing Platform for Data Analysis, by Ruoming Jin, Kent University.
  - G. Malewicz et al, "Pregel: A System for Large-Scale Graph Processing," ACM SIGMOD 2010, http://www.slideshare.net/shatteredNirvana/pregel-a-system-for-largescale-graph-processing
  - Carlos Guestrin et al, "GraphLab 2: Parallel Machine Learning for Large-Scale Natural Graphs," NIPS Big Learning Workshop 2011, http://www.select.cs.cmu.edu/code/graphlab/presentations/nips-biglearn-2011.pptx
  - Yucheng Low, Joseph Gonzalez et al, "GraphLab: A New Framework for Parallel Machine Learning," http://select.cs.cmu.edu/code/graphlab/uai2010_graphlab.pptx
  - Joseph Gonzalez et al, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," talk for OSDI 2012

- All copyrights belong to the original authors of the material.

Roadmap

- Graph problems and representations
- Emerging Parallel Processing Platforms for Graph-based Big Learning
  - Problems of MapReduce for Graph-based Processing/ MLDM
  - Pregel
  - GraphLab

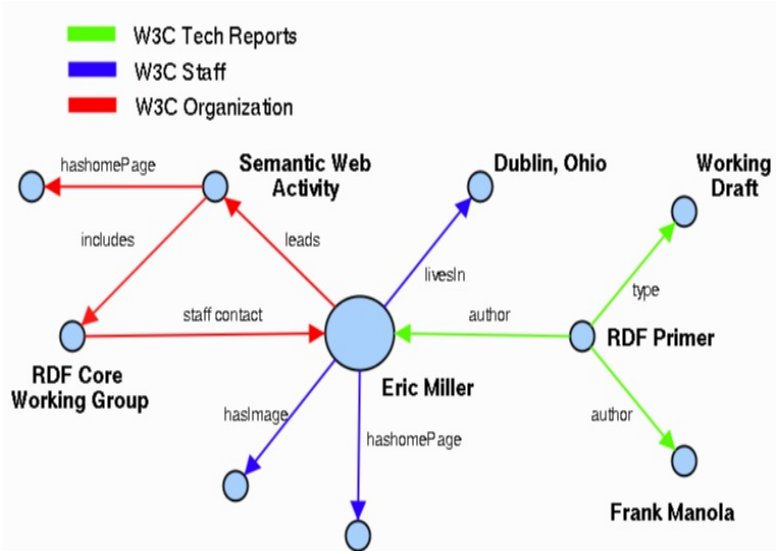
What's a graph?

- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles
- Graphs are everywhere:
  - Hyperlink structure of the Web
  - Physical structure of computers on the Internet
  - Interstate highway system
  - Social networks

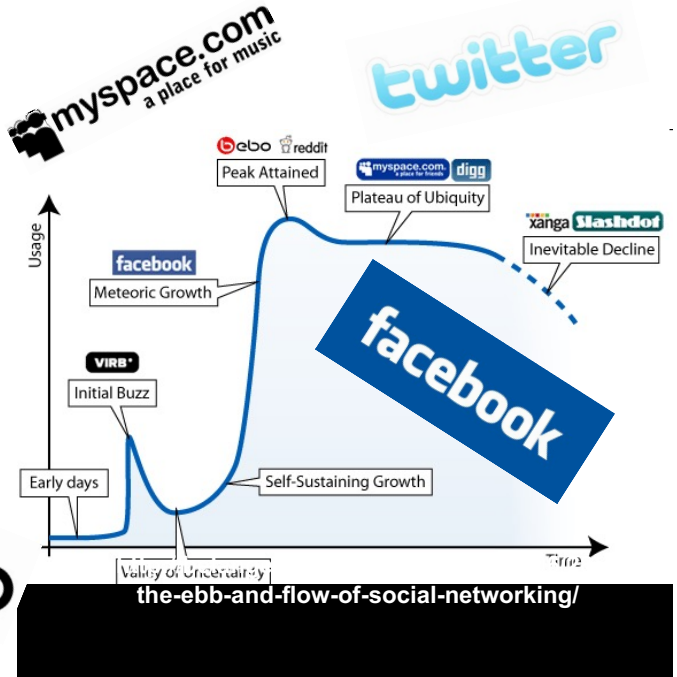
Some Graph Problems

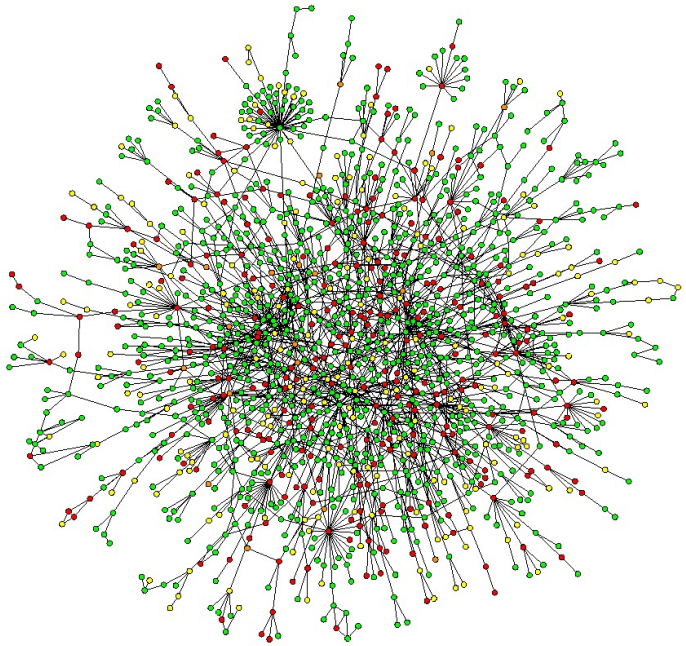
- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank

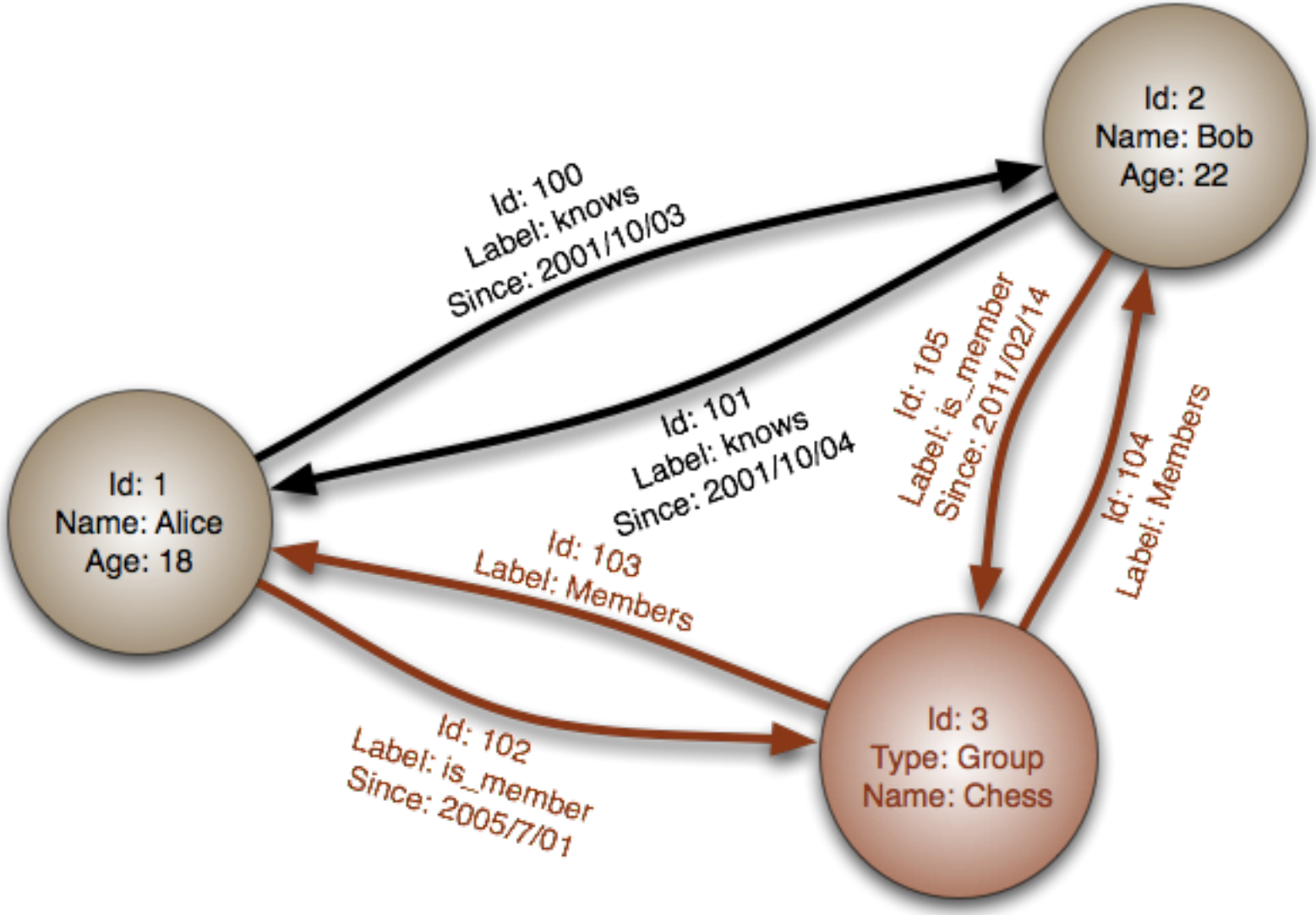
Ubiquitous Network (Graph) Data



Semantic Search, Guha et. al., WWW' 03






Id: 2
Name: Bob
Age: 22
Id: 100
Label: knows
Since: 2001/10/03
Id: 101
Label: knows
Since: 2001/10/04
Id: 105
Label: is_member
Since: 2011/02/14
Id: 104
Label: Members
Id: 1
Name: Alice
Age: 18
Id: 103
Label: Members
Id: 102
Label: is_member
Since: 2005/7/01
Id: 3
Type: Group
Name: Chess

Graph (and Relational) Analytics

- General Graph
  - Count the number of nodes whose degree is equal to 5
  - Find the diameter of the graphs
- Web Graph
  - Rank each webpage in the webgraph or each user in the twitter graph using PageRank, or other centrality measure
- Transportation Network
  - Return the shortest or cheapest flight/road from one city to another
- Social Network
  - Determine whether there is a path less than 4 steps which connects two users in a social network
- Financial Network
  - Find the path connecting two suspicious transactions;
- Temporal Network
  - Compute the number of computers who were affected by a particular computer virus in three days, thirty days since its discovery

Graph analytics industry in practice

- Graph data in many industries
- Graph analytics are powerful and can bring great business values/insights
- Graph analytics not utilized enough in small and medium sized enterprises due to lack of available platforms/tools (except leading tech companies which have high caliber in house engineering teams and resources)
- Luckily, this is changing fast with the new Graph-based Parallel/Big Learning Platforms like GraphLab

Challenge in Dealing with Graph Data

- Flat Files
  - No Query Support
- RDBMS
  - Can Store the Graph
  - Limited Support for Graph Query
    - Connect-By (Oracle)
    - Common Table Expressions (CTEs) (Microsoft)
    - Temporal Table

Native Graph Databases

- Emerging Field - http://en.wikipedia.org/wiki/Graph_database
- Storage and Basic Operators
  - Neo4j (an open source graph database)
  - InfiniteGraph
  - VertexDB

Representing Graphs

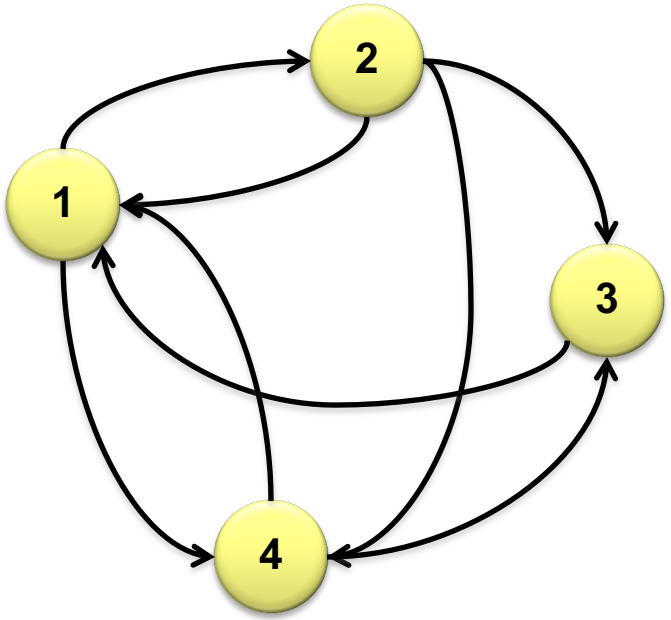
- G = (V, E)
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an *n* x *n* square matrix *M*

- *n* = |V|
- M_{ij} = 1 means a link from node *i* to *j*

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



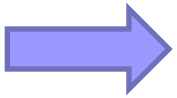
Adjacency Matrices: Critique

- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

An Example of Big Graph Processing Application

Label Propagation in Online Social Networks (Graphs)

Label Propagation Algorithm

- Social Arithmetic:

  50% What Joe lists on his profile
  40% Sue Ann Likes
  \+ 10% Carlos Like

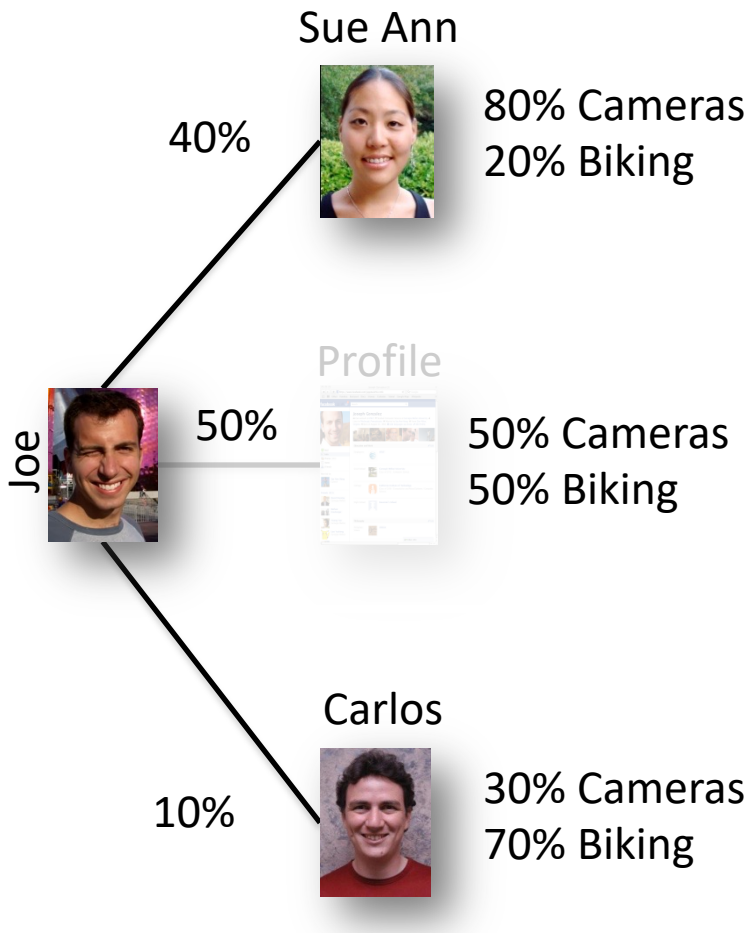
  Joe Likes: 60% Cameras, 40% Biking

- Recurrence Algorithm:

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

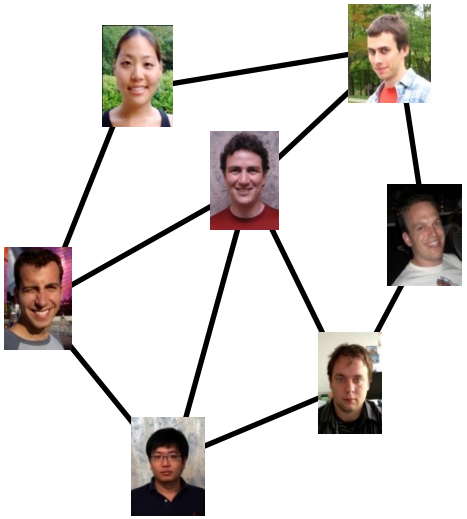
  - iterate until convergence

- Parallelism:
  - Compute all *Likes[i]* in parallel

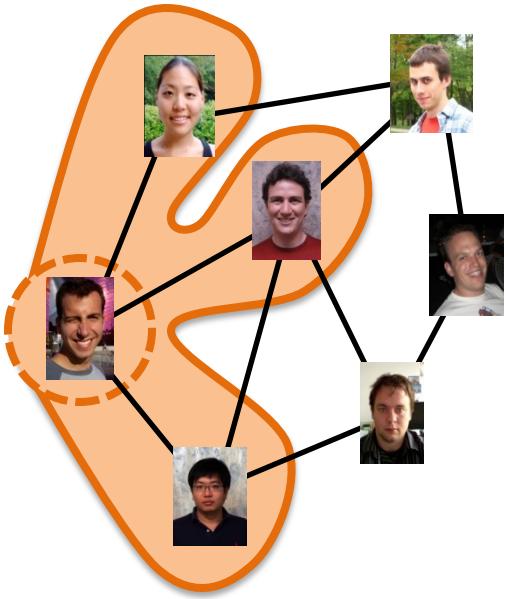


Properties of Graph Parallel Algorithms

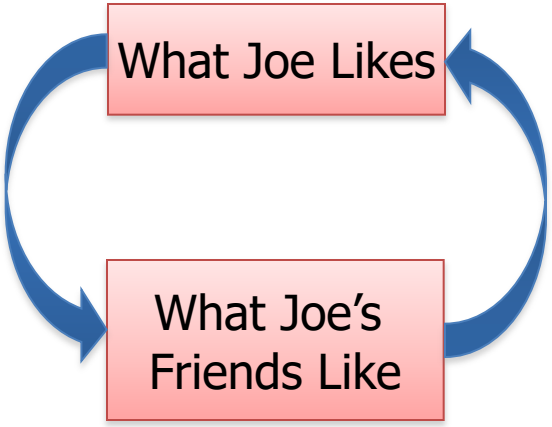
Dependency Graph



Factored Computation



Iterative Computation



Graphs Algorithms and Graph-based Parallel Processing

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Design Challenges
  - Very little computation work required per vertex.
  - Changing degree of parallelism over the course of execution.
- Generic recipe:
  - Represent graphs in some form of data structure, e.g. adjacency lists
  - Perform local computations in each vertex (node)
  - Pass along partial results via outlinks to destination vertices
  - Perform aggregation in each destination vertex (node) after receiving information from inlinks of a node
  - Iterate until convergence

Efficient Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies

Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!



Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

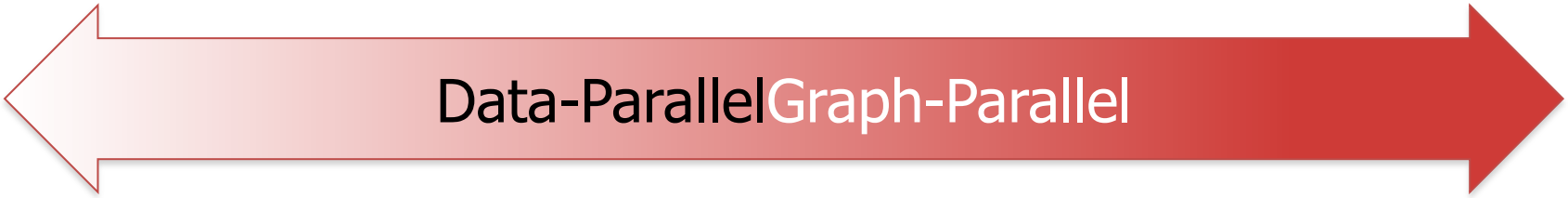
Embarrassingly Parallel Tasks

Is there more to Machine Learning

?

Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!



Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Embarrassingly Parallel Tasks

Map Reduce?

Lasso

Label Propagation

Kernel Methods

Belief Propagation

Tensor Factorization

PageRank

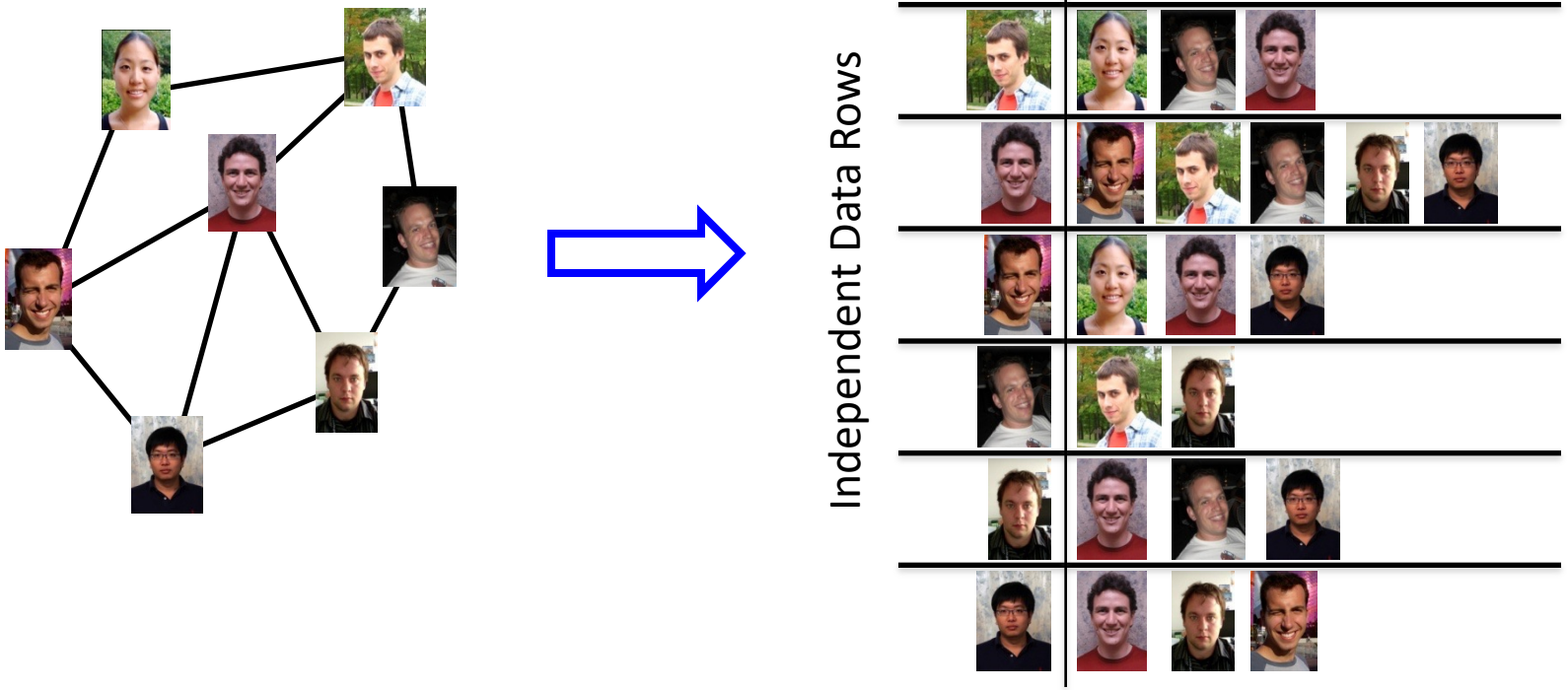
Deep Belief Networks

Neural Networks

Why not use Map-Reduce for Graph Parallel Algorithms?

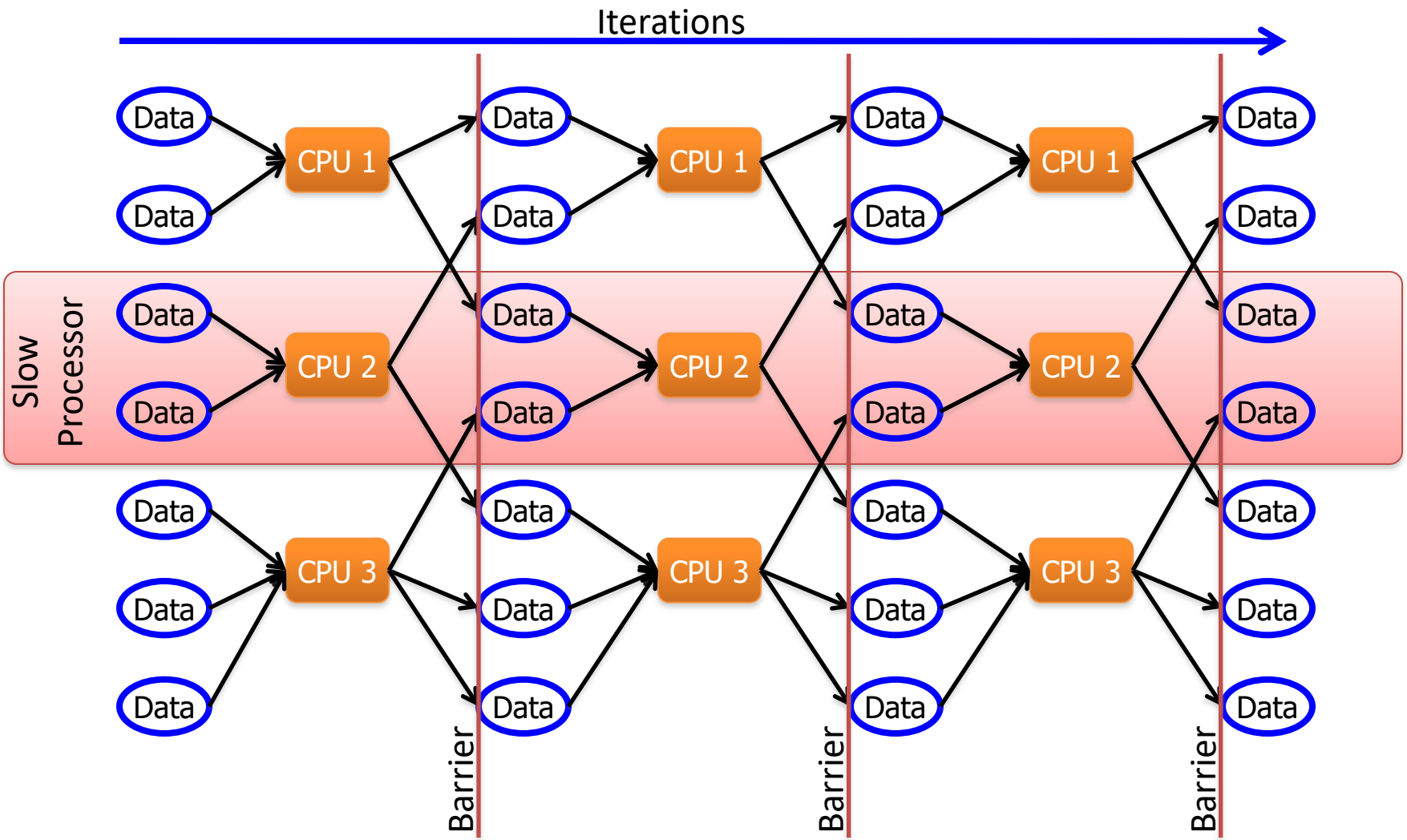
Data Dependencies

- Map-Reduce does not efficiently express dependent data
  - User must code substantial data transformations
  - Costly data replication



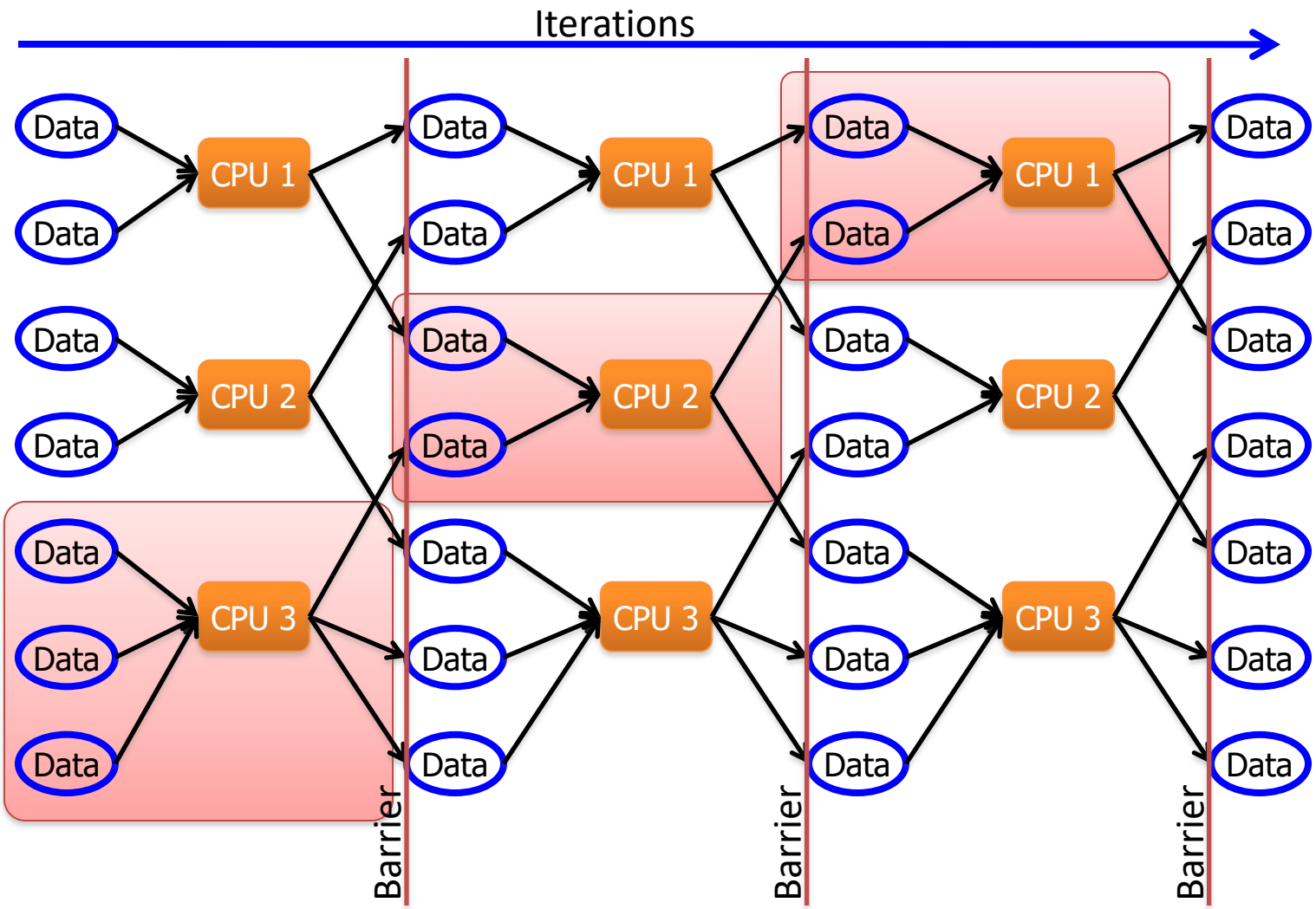
Iterative Algorithms

- Map-Reduce not efficiently express iterative algorithms:



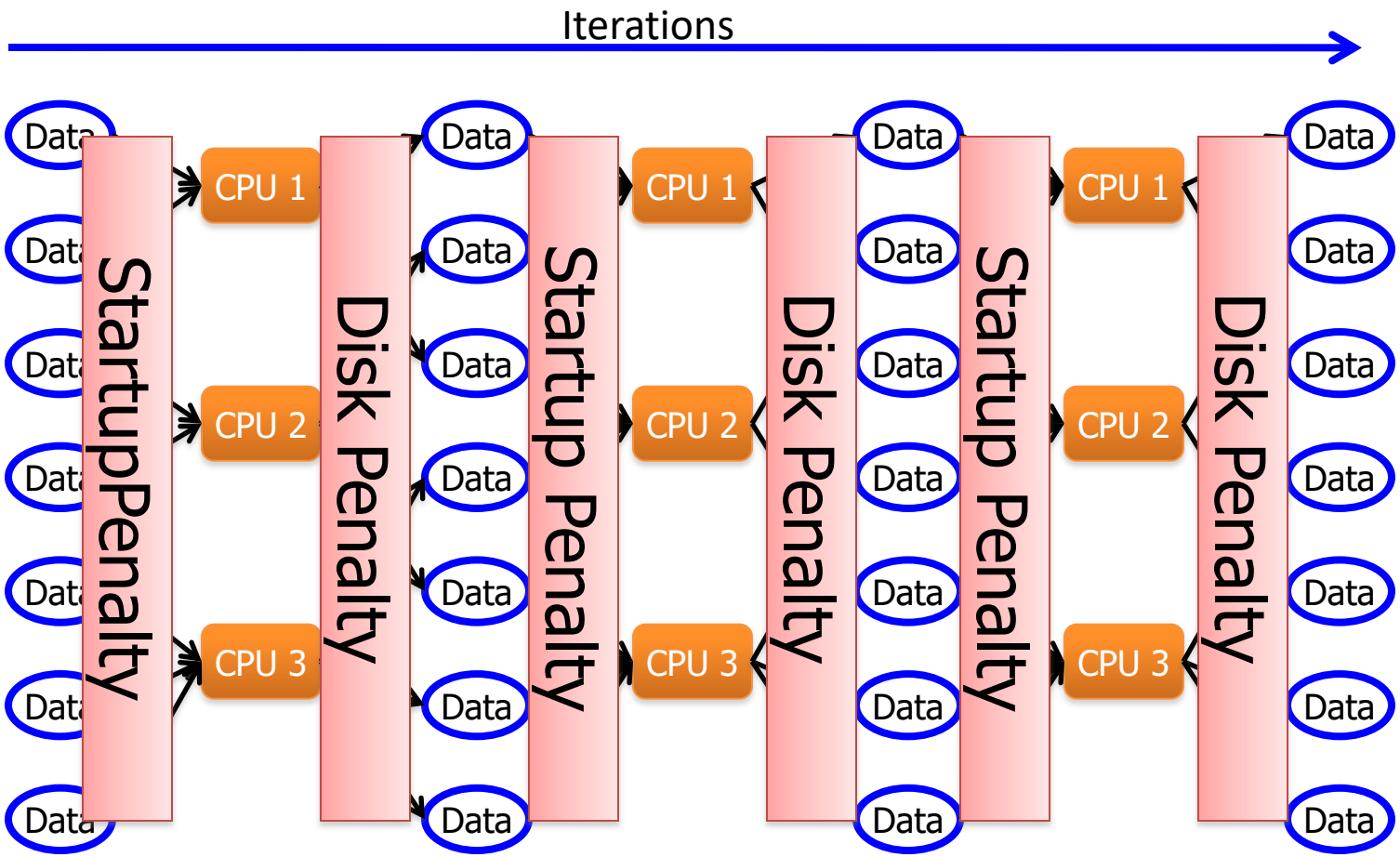
MapAbuse: Iterative MapReduce

- Only a subset of data needs computation:



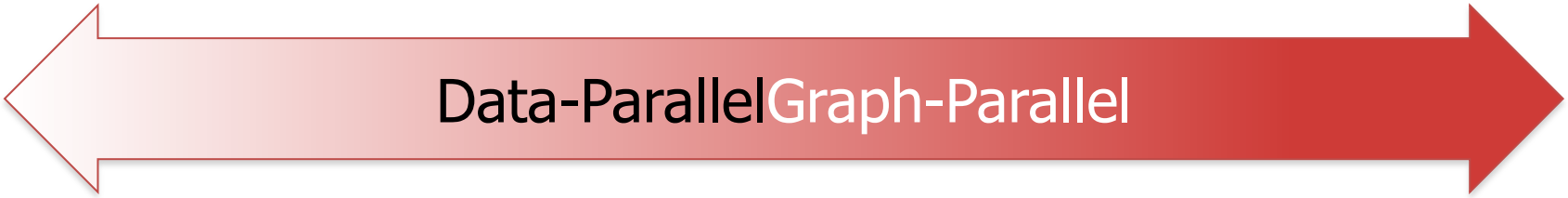
MapAbuse: Iterative MapReduce

- System is not optimized for iteration:



Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!



Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Pregel (Giraph)?

Lasso

SVM

Kernel Methods

Belief Propagation

Tensor Factorization

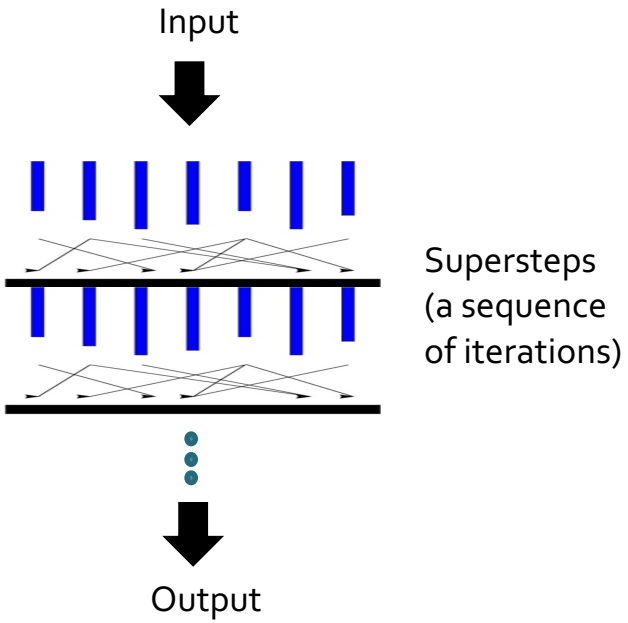
PageRank

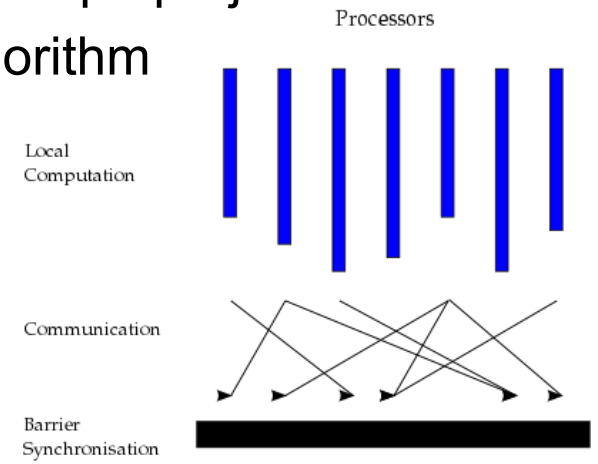
Deep Belief Networks

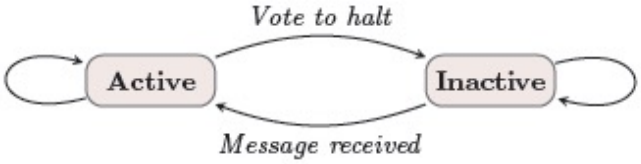
Neural Networks

Pregel (Giraph)

- Google's Pregel for Distributed Graph Processing (mostly in-memory-only)
  - Vertex-centric computation with barrier between successive iterations (aka Super-steps)
  - Inspired by Valiant's Bulk Synchronous Parallel model[4]
  - Open-source version under the Apache Giraph project
  - API with flexibility to express arbitrary algorithm
  - Scalable and Fault-tolerant platform

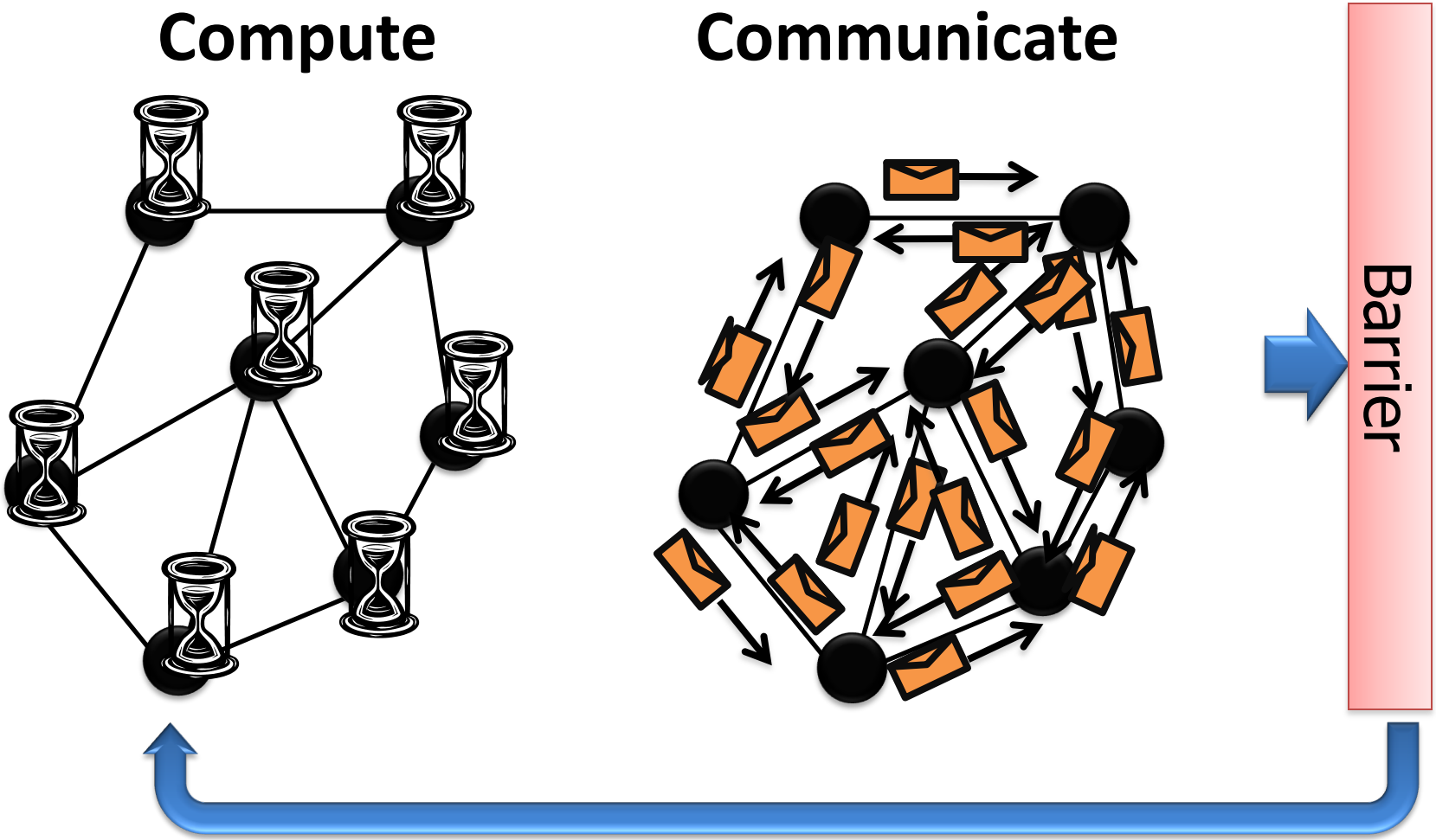






Pregel (Giraph)

- Bulk Synchronous Parallel Model:



PageRank in Giraph (Pregel)

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

```
bsp_page_rank() {
  sum = 0
  forall (message in in_messages())
     sum = sum + message
  rank = ALPHA + (1-ALPHA) * sum;
  set_vertex_value(rank);

  if (current_super_step() < MAX_STEPS) {
     nedges = num_out_edges()
     forall (neighbors in out_neighbors())
        send_message(rank / nedges);
  } else vote_to_halt();
}
```

Sum PageRank over incoming messages

Send new messages to neighbors or terminate

Computation Model for Pregel

- Within each Super-Step, concurrent computation and communication need not be ordered in time
- Communication through message passing
- Each vertex
  - Receives messages sent in the previous Super-step
  - Executes the same user-defined function
  - Modifies its value or that of its outgoing edges
  - Sends messages to other vertices (to be received in the next superstep)
  - Mutates the topology of the graph
  - Votes to halt if it has no further work to do

Problem

Bulk synchronous computation can be highly inefficient.

Example:

Loopy Belief Propagation

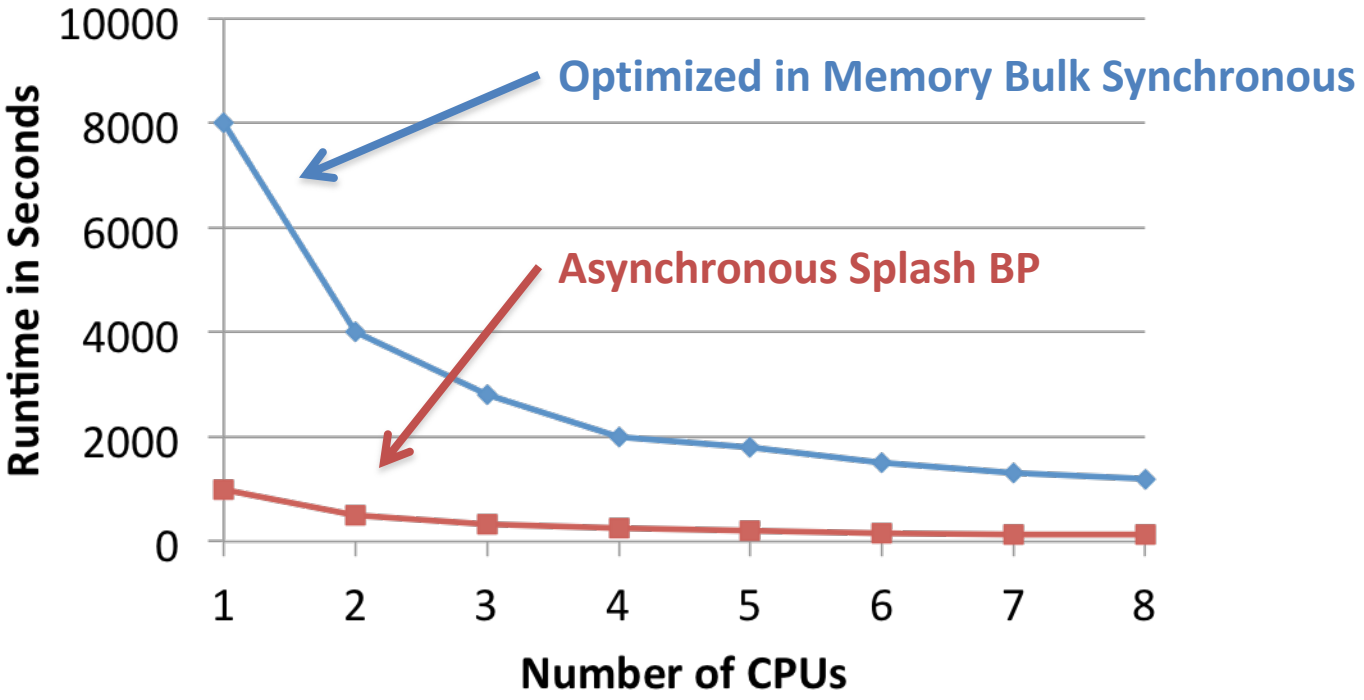
Data-Parallel Algorithms can be **Inefficient**

Residual Splash for Optimally Parallelizing Belief Propagation

Joseph E. Gonzalez
Carnegie Mellon University

Yucheng Low
Carnegie Mellon University

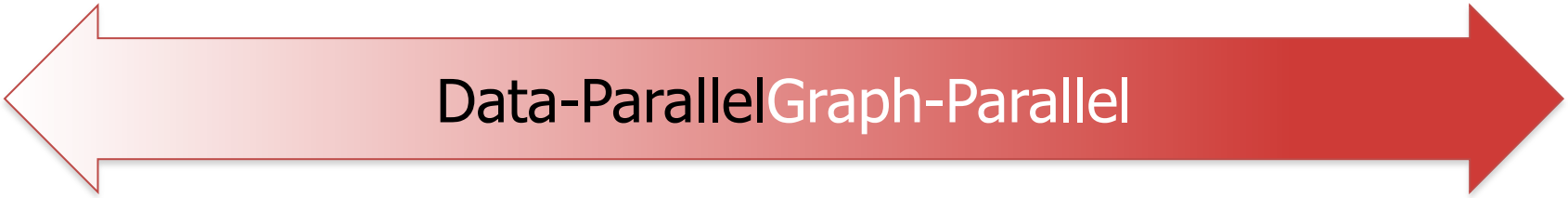
Carlos Guestrin
Carnegie Mellon University



The limitations of the Map-Reduce abstraction can lead to inefficient parallel algorithms.

The Need for a New Abstraction

- Map-Reduce is not well suited for Graph-Parallelism



Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Pregel (Giraph)

Carnegie Mellon

SVM

Kernel Methods

Belief Propagation

Tensor Factorization

PageRank

Deep Belief Networks

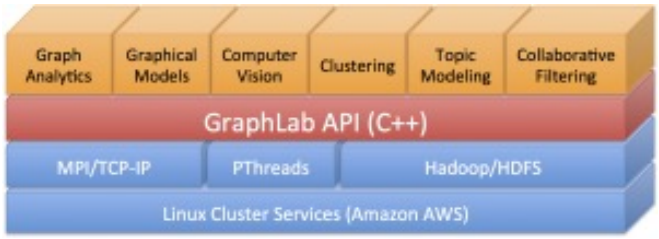
Neural Networks

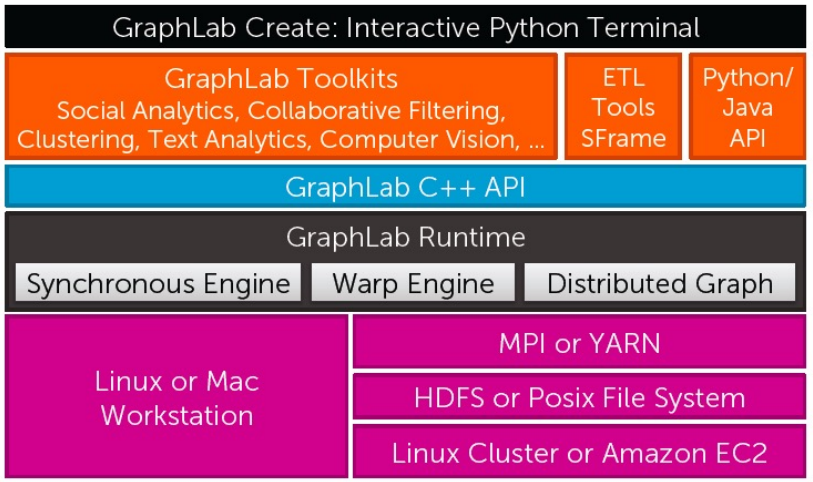
Lasso

What is GraphLab?

Graph-based Big Learning/ Parallel Processing Platforms (cont'd)

- GraphLab – another vertex-centric model (http://GraphLab.org/projects, http://GraphLab.com) ; Company renamed to Dato, and then to Turi, which was acquired by Apple in Aug. 2016.
  - Originated from CMU and now by UWashington@Seattle ;
  - Different versions supporting wide-range of platforms:
    - GraphLab 1.0 was designed to run on closely-coupled, shared-memory multicore machine.
    - GraphChi enables a Single PC to process graphs with billions of edges
    - GraphLab (Ver2.x) or so-called the PowerGraph model targets for seriously-imbalanced node degrees found in practical (Natural) graphs and support parallel processing on Share-Nothing Cluster architecture
      - Taking the split-vertex instead split-edge approach
    - GraphCreate (Beta) allows you to code in your PC using Python but deploy to run over Cloud-based shared-nothing clusters.





The GraphLab Framework

Graph Based

Data Representation

Update Functions

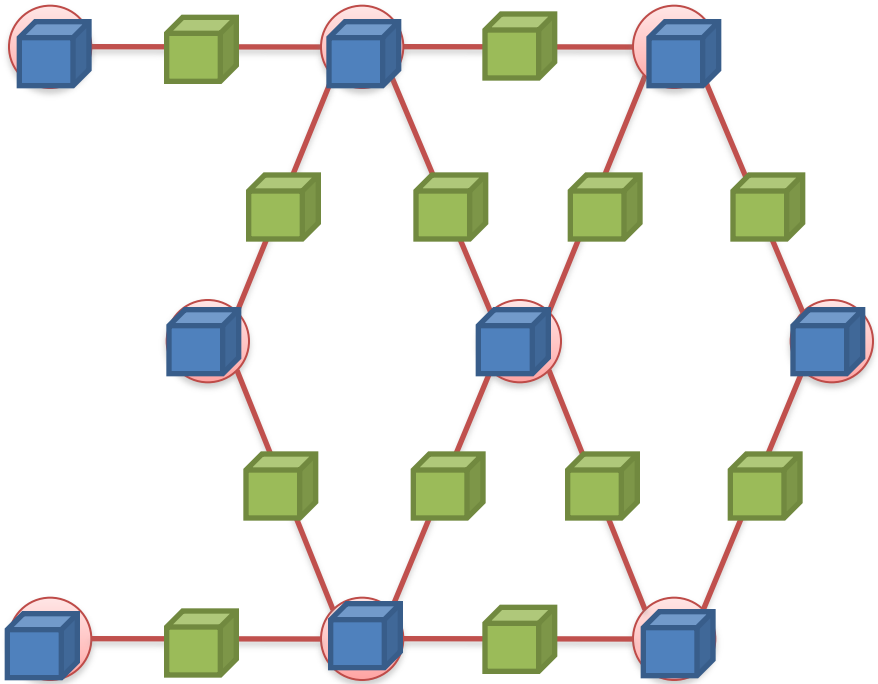
User Computation

Scheduler

Consistency Model

Data Graph

A **graph** with arbitrary data (C++ Objects) associated with each vertex and edge.



Graph:



• Social Network

Vertex Data:



•User profile text

• Current interests estimates

Edge Data:



• Similarity weights

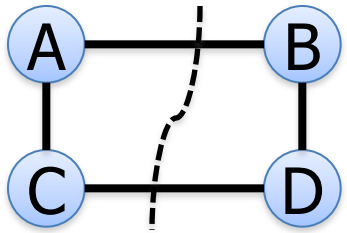
Implementing the Data Graph

Multicore Setting

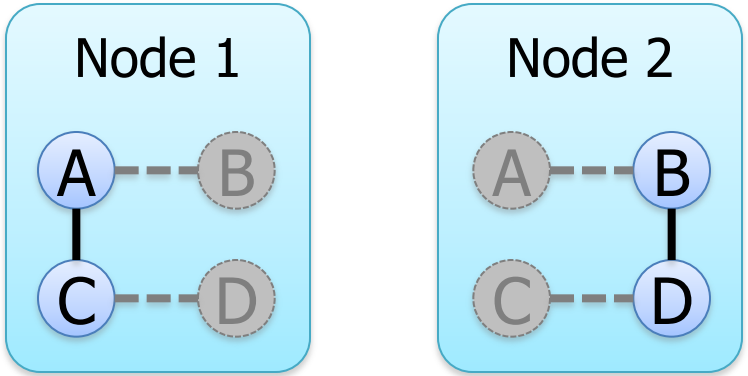
- **In Memory**
- Relatively Straight Forward
  - vertex_data(vid) → data
  - edge_data(vid,vid) → data
  - neighbors(vid) → vid_list
- Challenge:
  - Fast lookup, low overhead
- Solution:
  - Dense data-structures
  - Fixed Vdata&Edata types
  - Immutable graph structure

Cluster Setting

- **In Memory**
- Partition Graph:
  - ParMETIS or Random Cuts



- Cached Ghosting



The GraphLab Framework

Graph Based

Data Representation

Update Functions

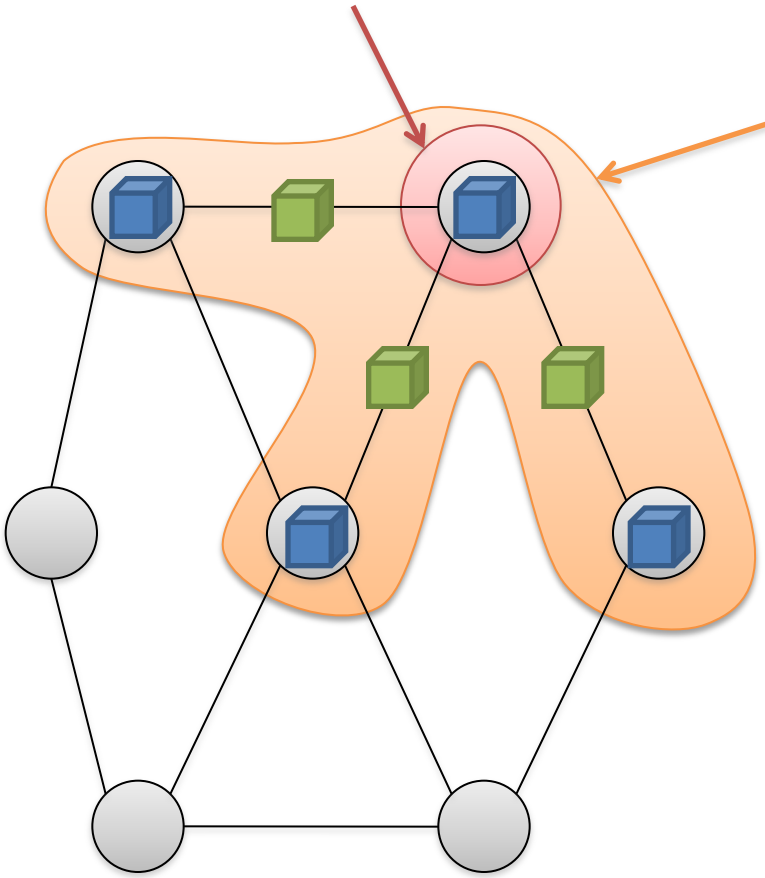
User Computation

Scheduler

Consistency Model

Update Functions

An **update function** is a user defined program which when applied to a **vertex** transforms the data in the **scope**of the vertex



```
label_prop(i, scope){
// Get Neighborhood data
  (Likes[i], W_ij, Likes[j]) ←scope;

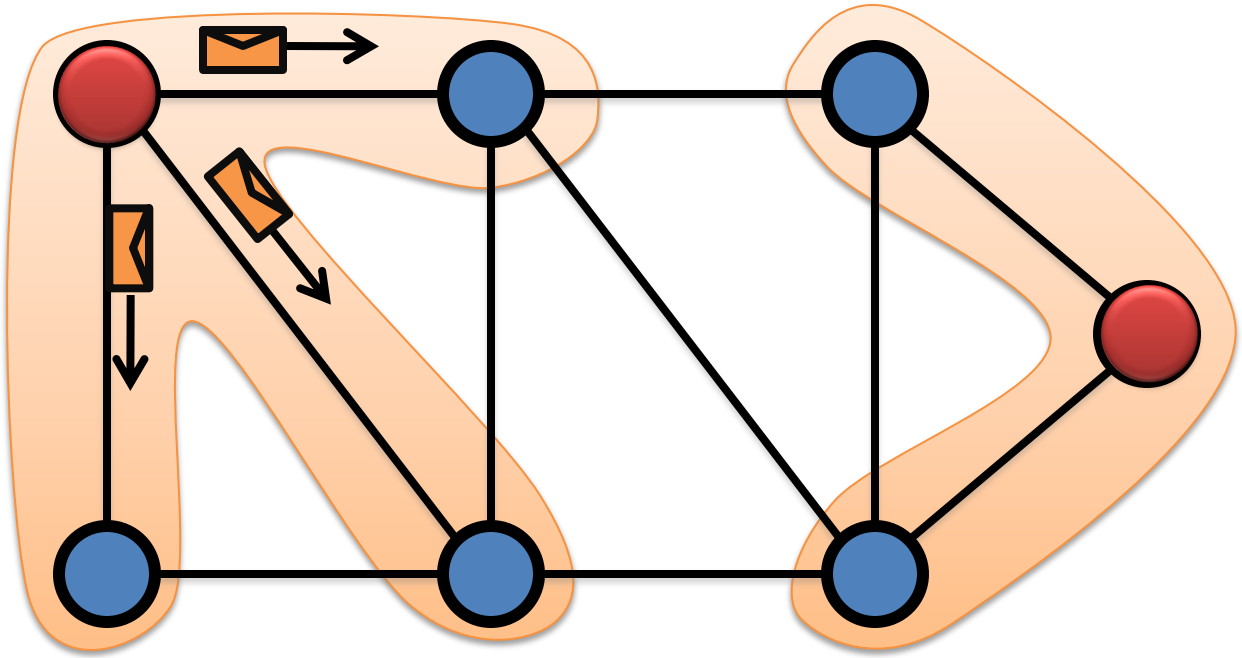
// Update the vertex data

  Likes[i] ← Σ_{j∈Friends[i]} W_ij × Likes[j];

// Reschedule Neighbors if needed
  if Likes[i] changes then
reschedule_neighbors_of(i);
}
```

The **Graph-Parallel** Abstraction

- A user-defined **Vertex-Program** runs on each vertex
- **Graph** constrains **interaction** along edges
  - Using **messages** (e.g. **Pregel** [PODC'09, SIGMOD'10])
  - Through **shared state** (e.g., **GraphLab** [UAI'10, VLDB'12])
- **Parallelism**: run multiple vertex programs simultaneously



PageRank Algorithm

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

Rank of user *i*

Weighted sum of neighbors' ranks

- Update ranks in parallel
- Iterate until convergence

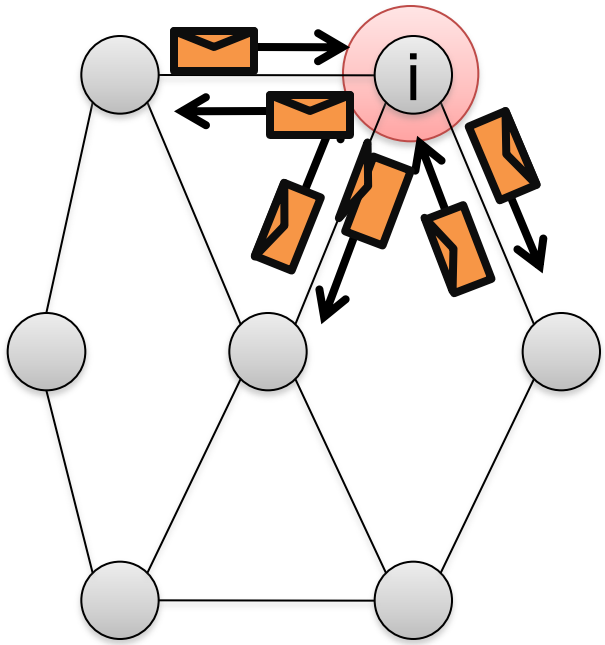
The Pregel Abstraction

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] * w_ij) to vertex j
```



Malewicz et al. [PODC'09, SIGMOD'10]

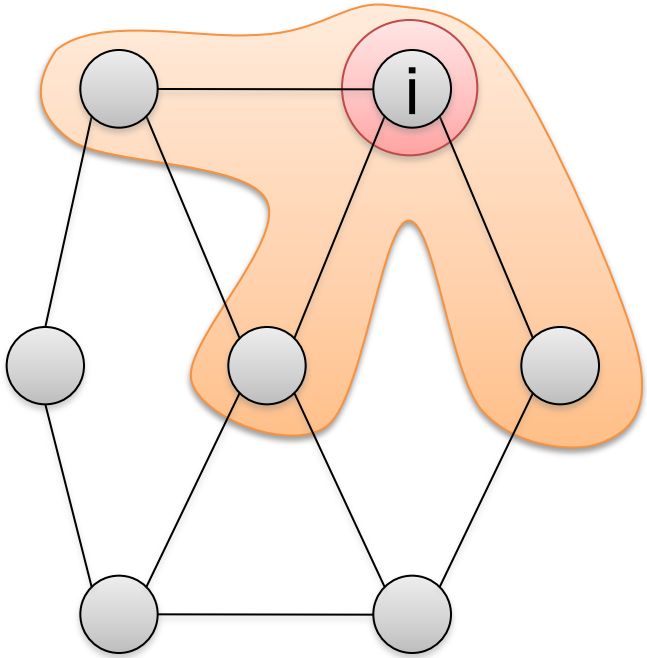
The GraphLab Abstraction

Vertex-Programs directly **read** the neighbors state

```
GraphLab_PageRank(i)
  // Compute sum over neighbors
  total = 0
  foreach( j in in_neighbors(i)):
    total = total + R[j] * w_ji

  // Update the PageRank
  R[i] = 0.15 + total

  // Trigger neighbors to run again
  if R[i] not converged then
    foreach( j in out_neighbors(i)):
      signal vertex-program on j
```



Low et al. [UAI'10, VLDB'12]

The GraphLab Framework

Graph Based

Data Representation

Update Functions

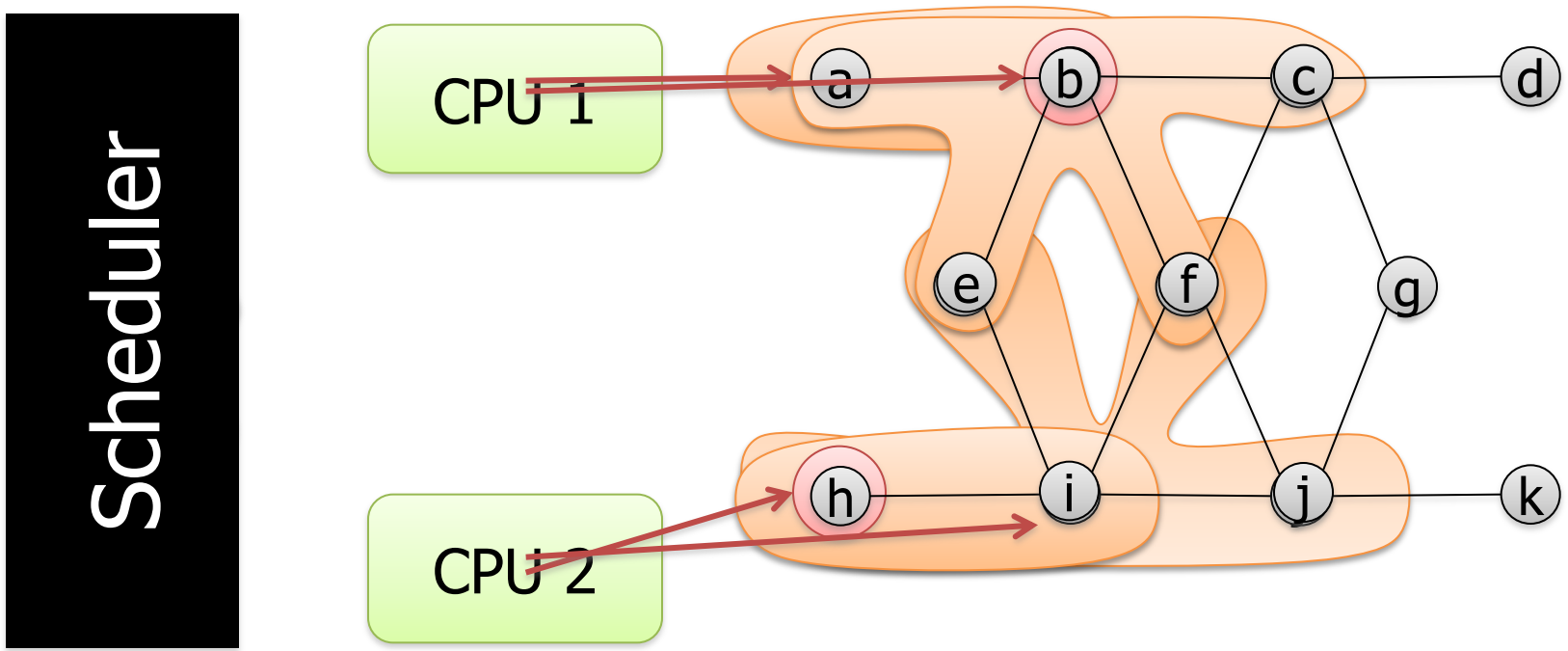
User Computation

Scheduler

Consistency Model

The Scheduler

The **scheduler** determines the order that vertices are updated.



The process repeats until the scheduler is empty.

Choosing a Schedule

The choice of schedule affects the correctness and parallel performance of the algorithm

- GraphLab provides several different schedulers
  - Round Robin: vertices are updated in a fixed order
  - FIFO: Vertices are updated in the order they are added
  - Priority: Vertices are updated in priority order

Obtain different algorithms by simply changing a flag!

```
--scheduler=roundrobin
        --scheduler=fifo
                --scheduler=priority
```

The GraphLab Framework

Graph Based

Data Representation

Update Functions

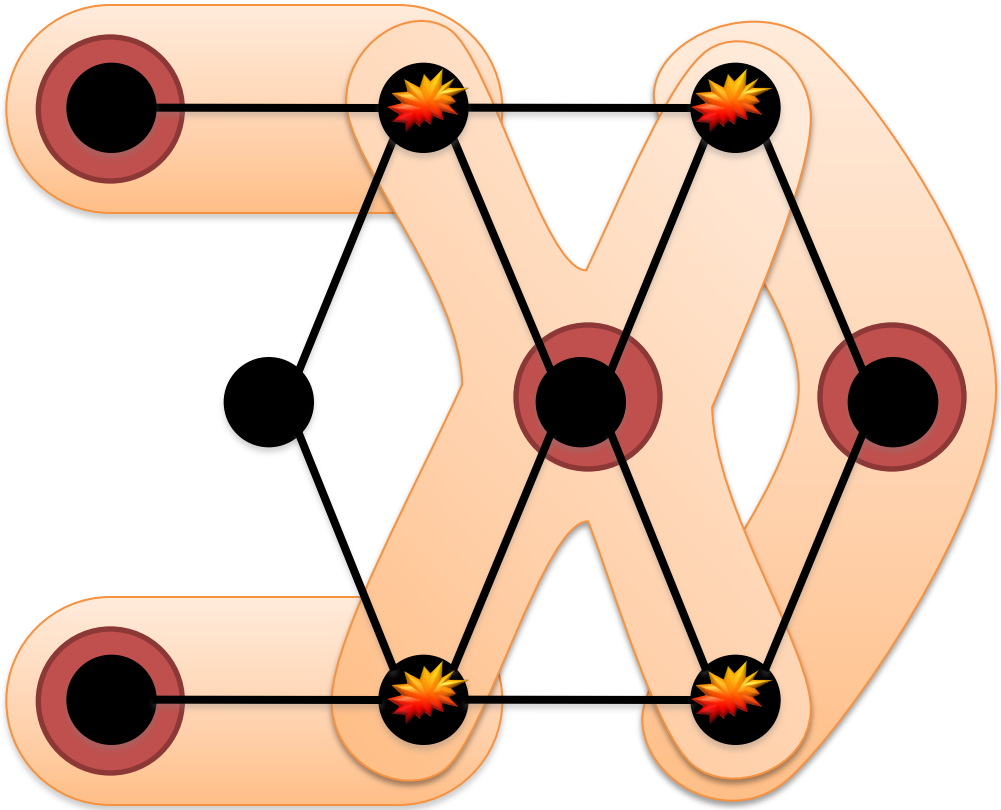
User Computation

Scheduler

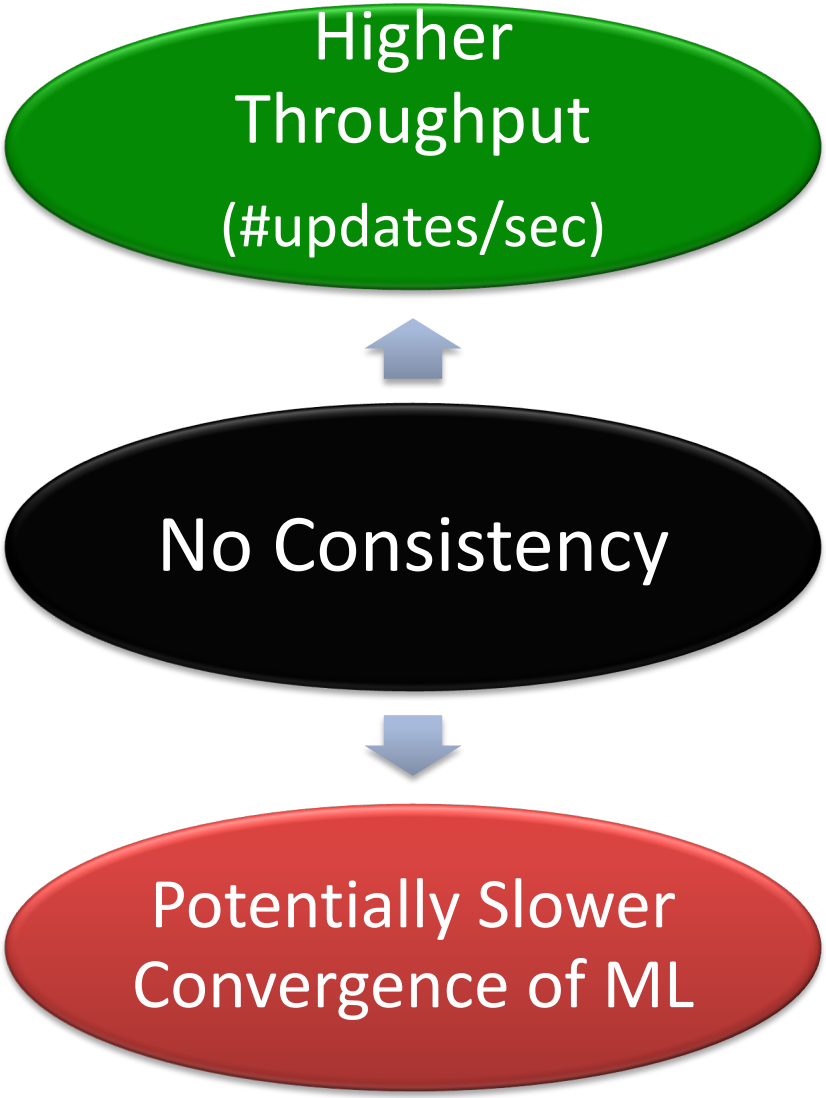
Consistency Model

Ensuring Race-Free Code

- How much can computation **overlap**?

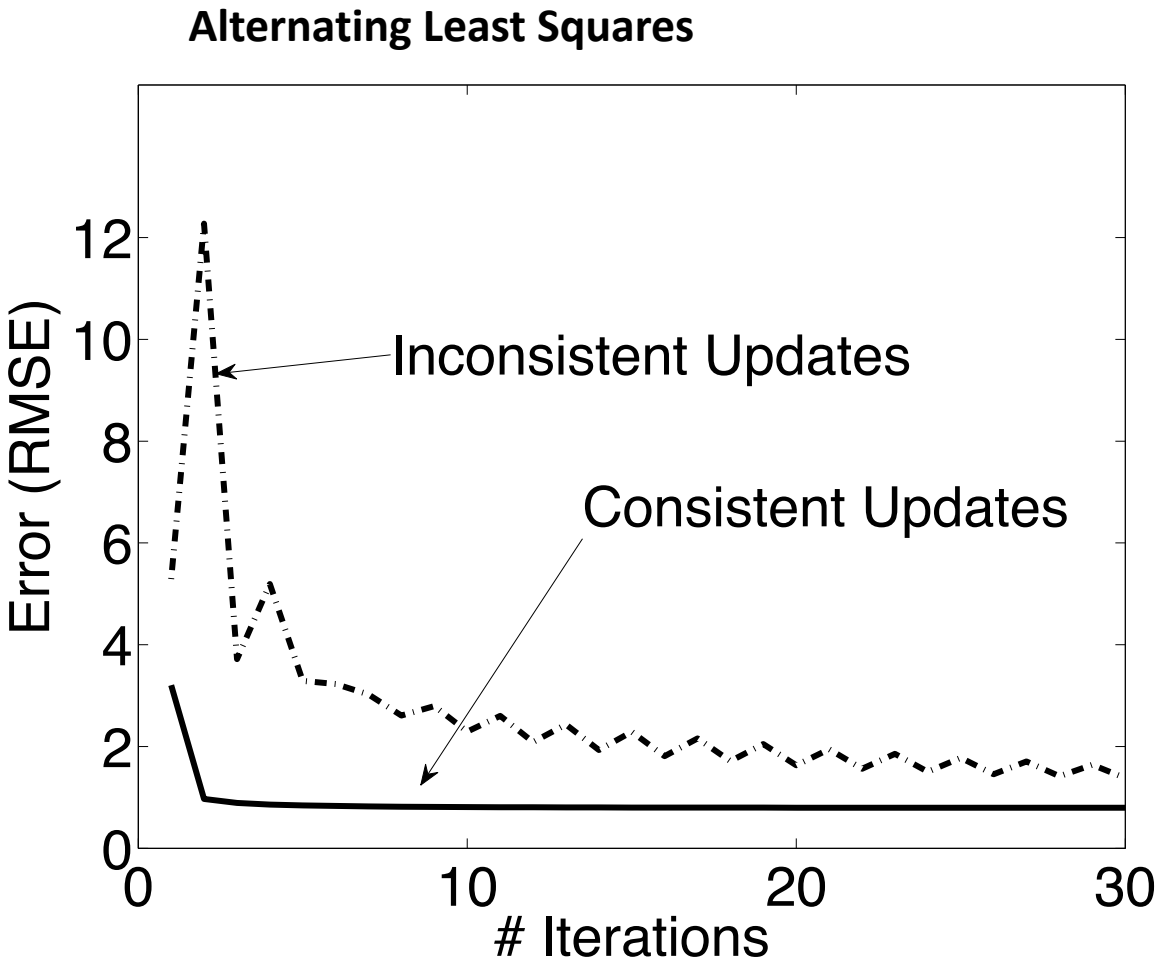


Need for Consistency?



Importance of Consistency

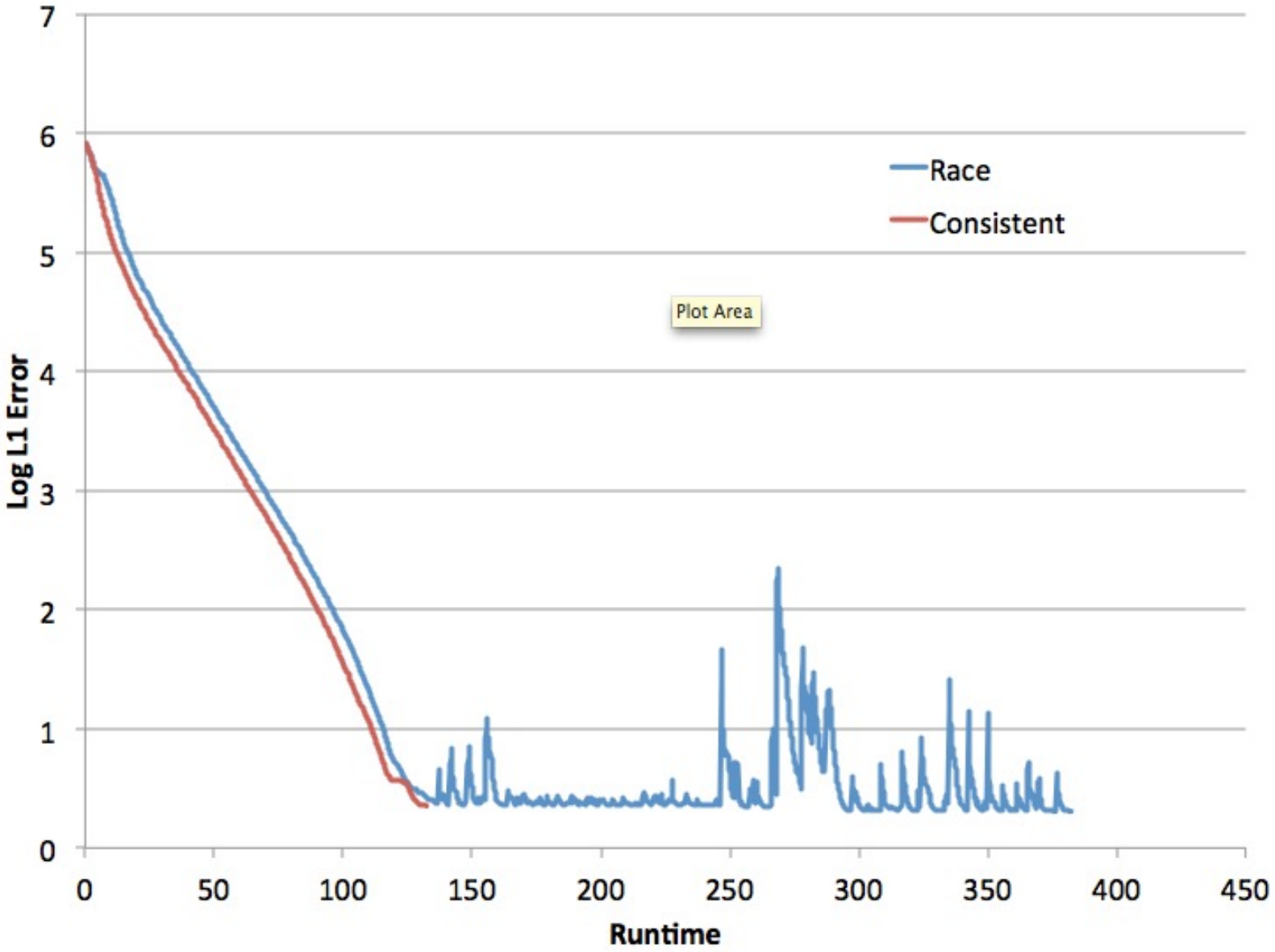
Many algorithms require strict consistency, or performs significantly better under strict consistency.



Even Simple PageRank can be Dangerous

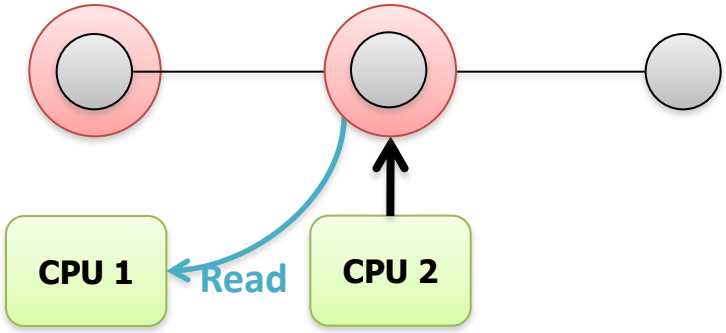
```
GraphLab_pagerank(scope) {
      ref sum = scope.center_value
      sum = 0
      forall (neighbor in scope.in_neighbors )
            sum = sum + neighbor.value / nbr.num_out_edges
      sum = ALPHA + (1-ALPHA) * sum
      …
```

Inconsistent PageRank



Even Simple PageRank can be Dangerous

```
GraphLab_pagerank(scope) {
        ref sum = scope.center_value
        sum = 0
        forall (neighbor in scope.in_neighbors)
                sum = sum + neighbor.value / nbr.num_out_edges
        sum = ALPHA + (1-ALPHA) * sum
        …
```



Read-write race ➔
CPU 1 reads bad PageRank estimate,
as CPU 2 computes value

Race Condition Can Be Very Subtle

Unstable

```
GraphLab_pagerank(scope) {
        ref sum = scope.center_value
        sum = 0
        forall (neighbor in scope.in_neighbors)
                sum = sum + neighbor.value /
neighbor.num_out_edges
        sum = ALPHA + (1-ALPHA) * sum
        …
```

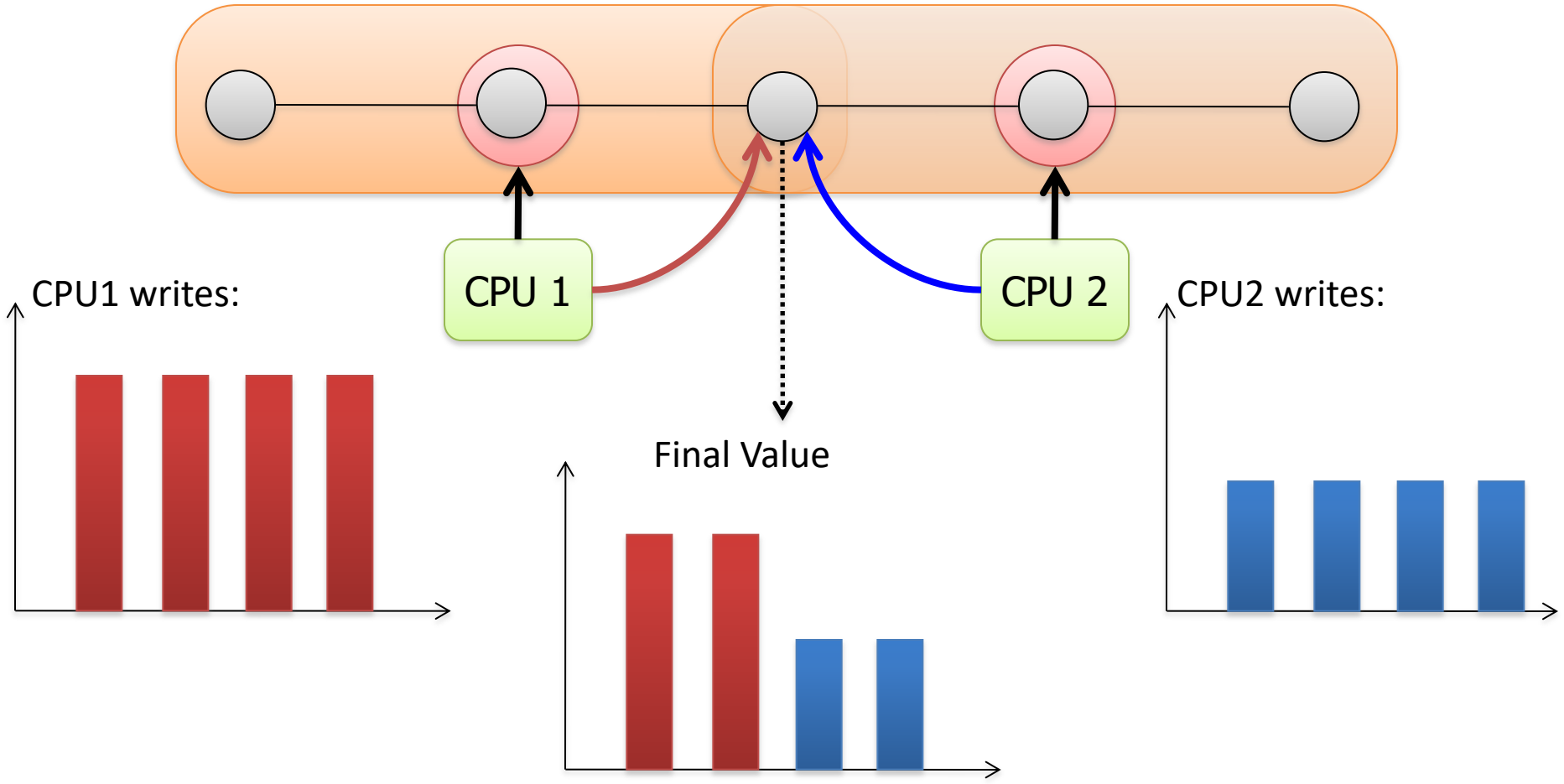
Stable

```
GraphLab_pagerank(scope) {
        sum = 0
        forall (neighbor in scope.in_neighbors)
                sum = sum + neighbor.value /
nbr.num_out_edges
        sum = ALPHA + (1-ALPHA) * sum
        scope.center_value = sum
        …
```

This was actually encountered in user code.

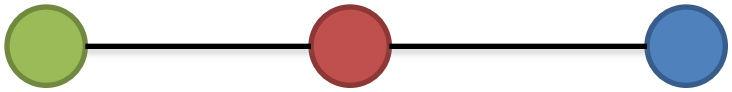
Common Problem: Write-Write Race

Processors running **adjacent update functions** simultaneously modify shared data:

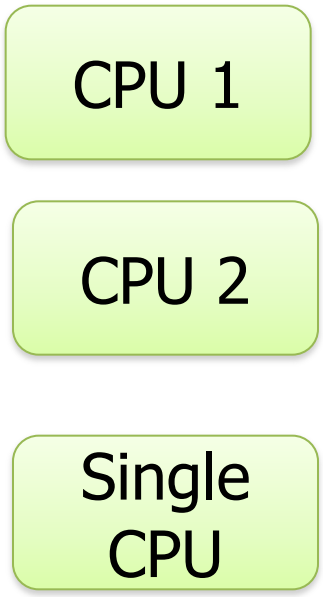


GraphLab Supports Serializability

Serializability: For a group of **concurrent (parallel) transactions,** e.g. executing the update functions for different vertices, the results produced by these concurrent transactions are the same as if each transaction has taken place one after another (without interleaving) in some sequential order.



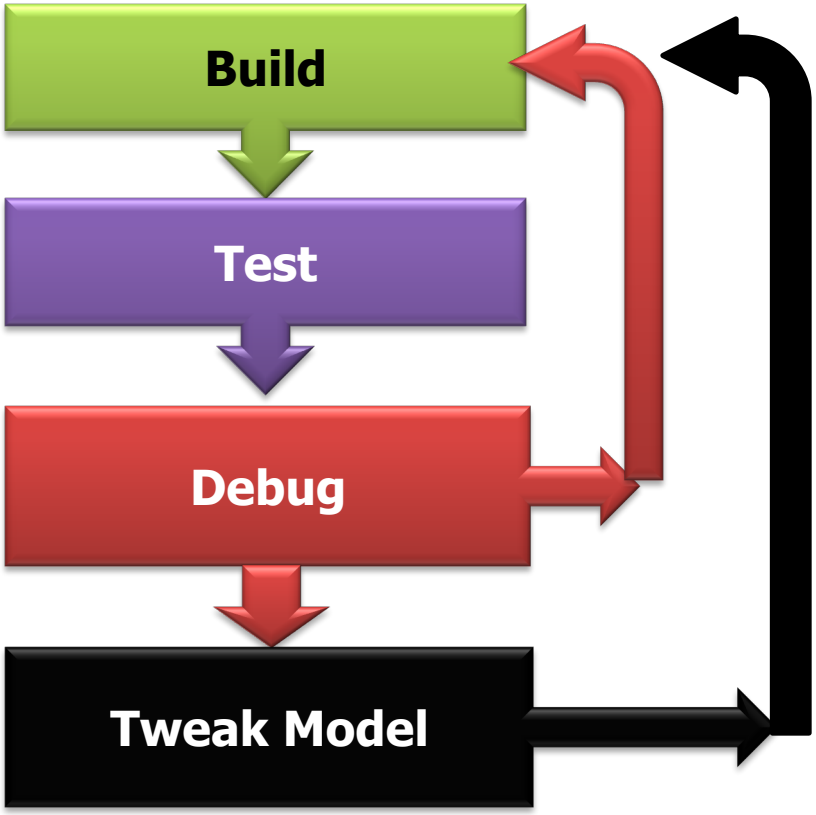
Parallel



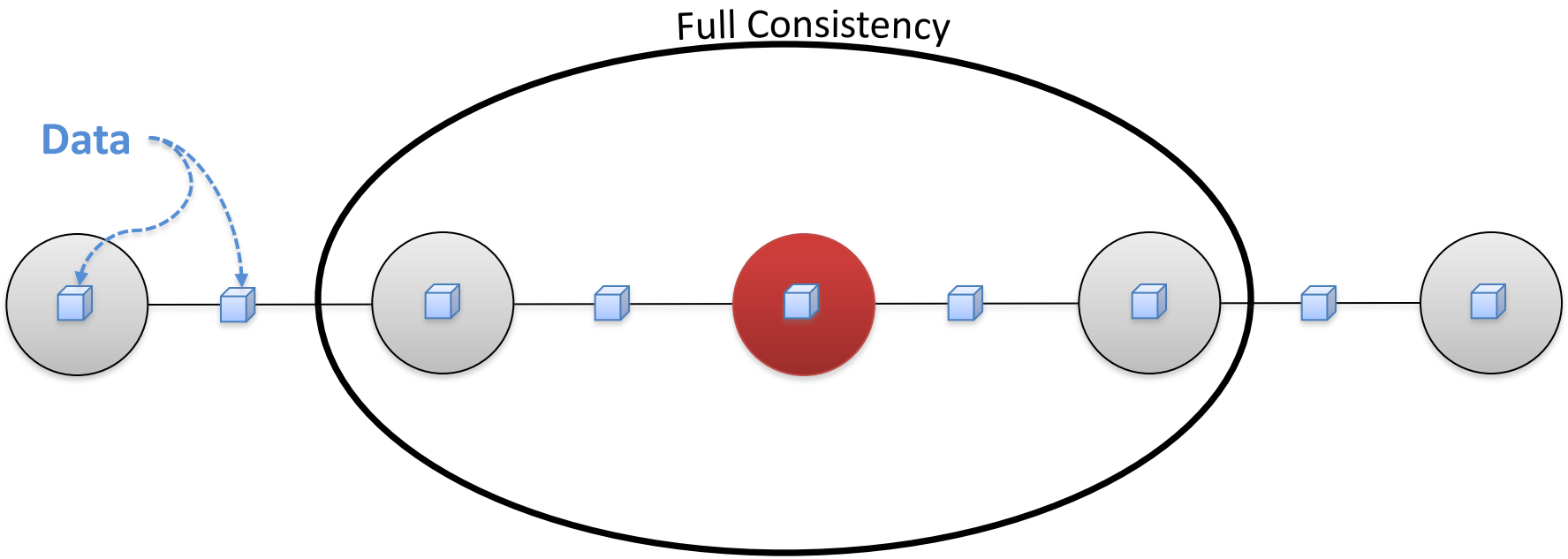
Sequential

Importance of Consistency

Machine learning algorithms require “model debugging”

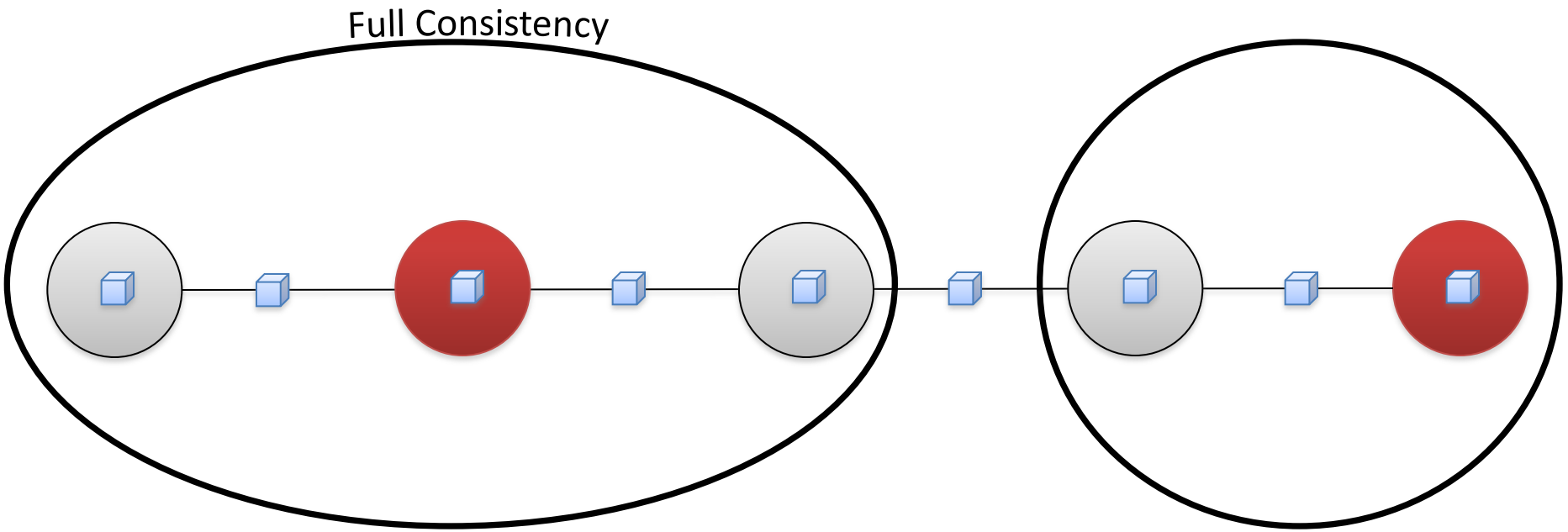


Consistency Rules

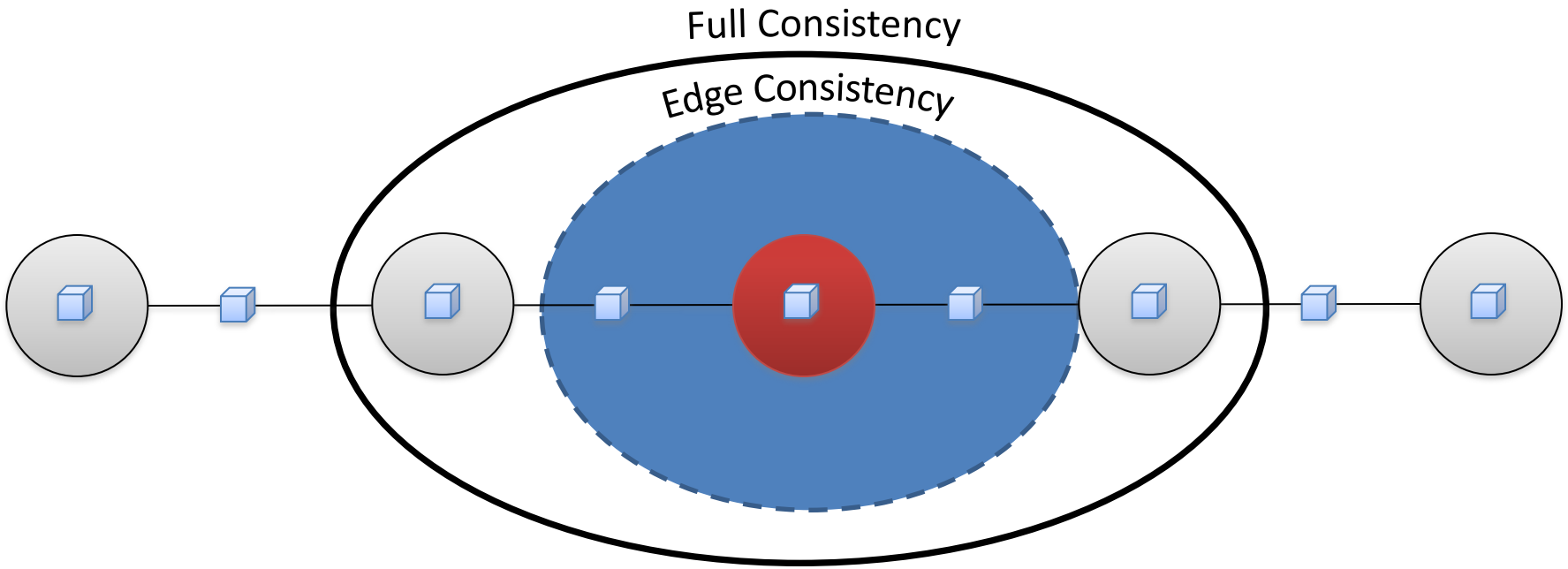


Guarantee serializability for all update functions

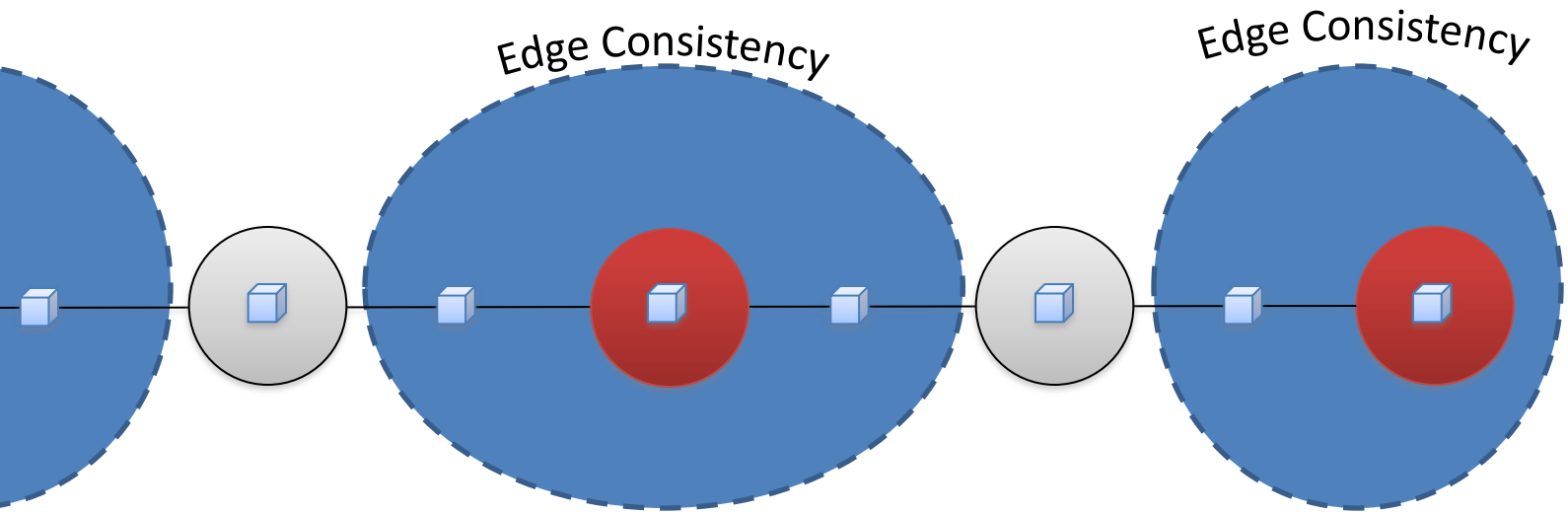
Full Consistency



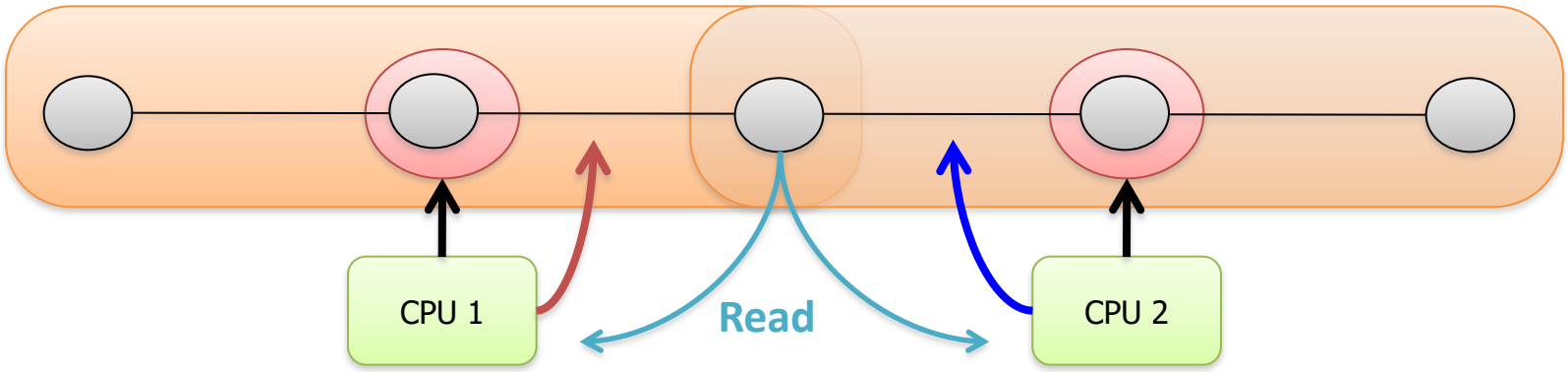
Obtaining More Parallelism



Edge Consistency

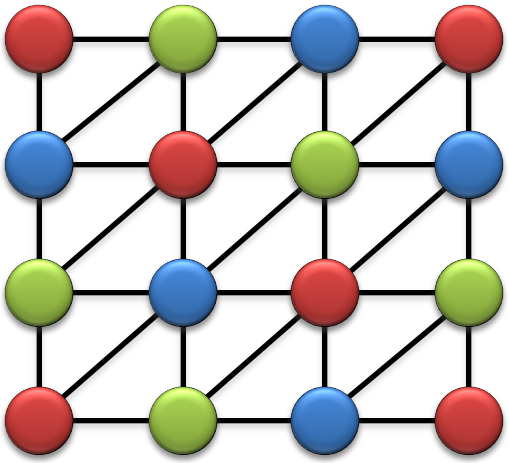


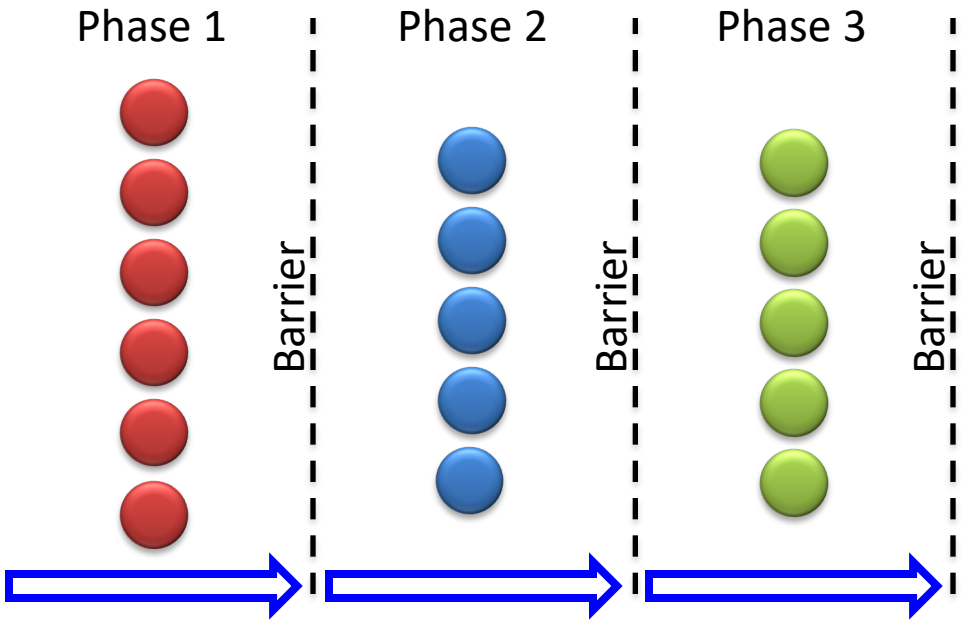
Safe



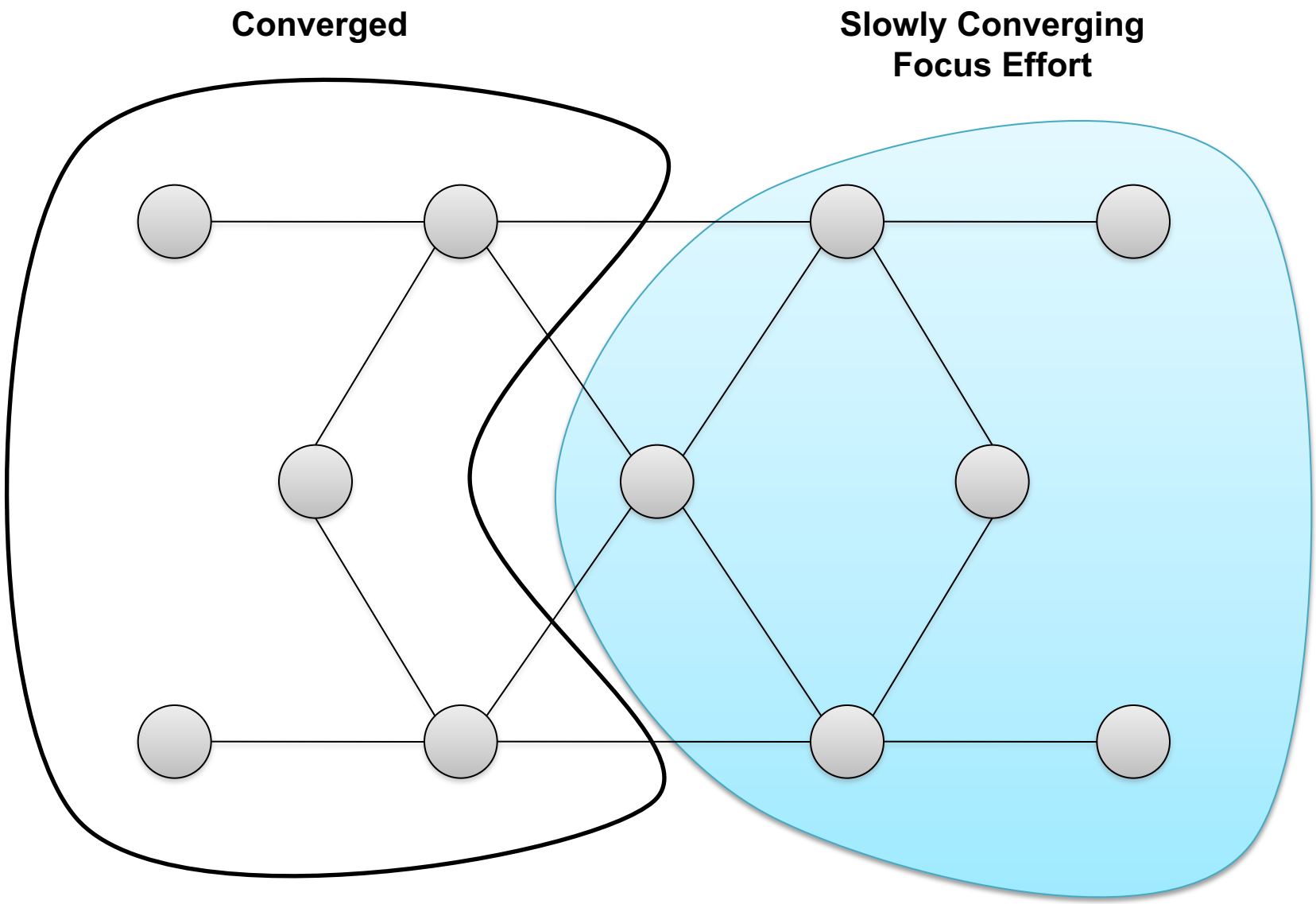
Consistency Through Scheduling

- Edge Consistency Model:
  - Two vertices can be **Updated** *simultaneously* if they do not share an edge.
- Graph Coloring:
  - Two vertices can be assigned the same color if they do not share an edge.





Dynamic Computation



PageRank Update Function

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

```
GraphLab_pagerank(scope) {
      double sum = 0;
      forall ( nbr in scope.in_neighbors() )
            sum = sum + neighbor.value() /
nbr.num_out_edges();

      double old_rank = scope.ve
      scope.center_value() = ALP

      double residual = abs(scope.center_value() -
old_rank);
      if (residual > EPSILON)
            reschedule_out_neighbors();
}
```

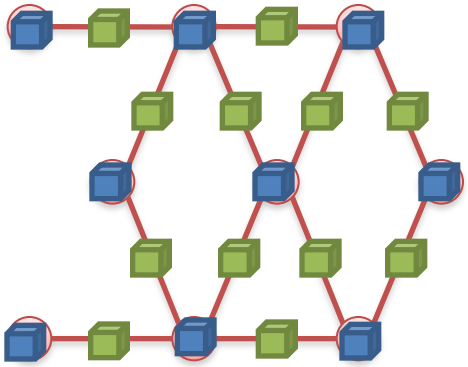
Directly Read Neighbor Values

Dynamically Schedule **Computation**

The GraphLab Framework

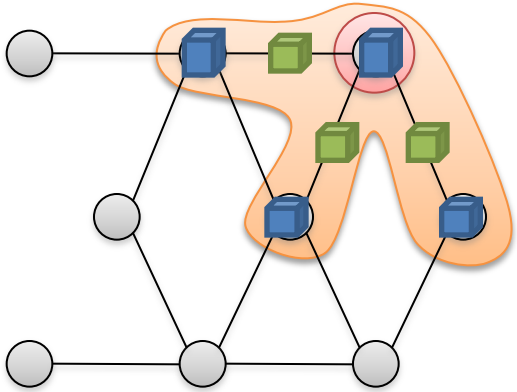
Graph Based

Data Representation

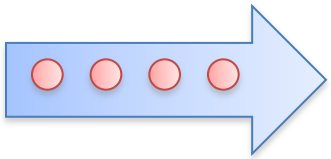


Update Functions

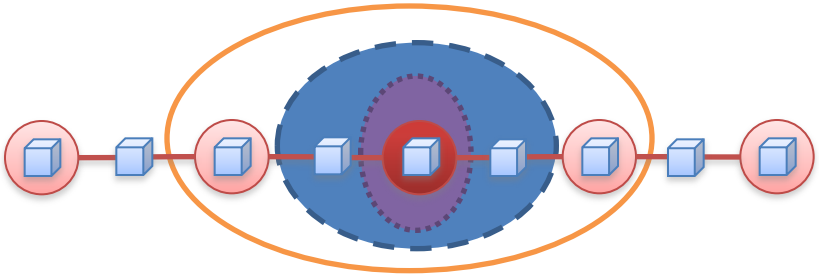
User Computation



Scheduler



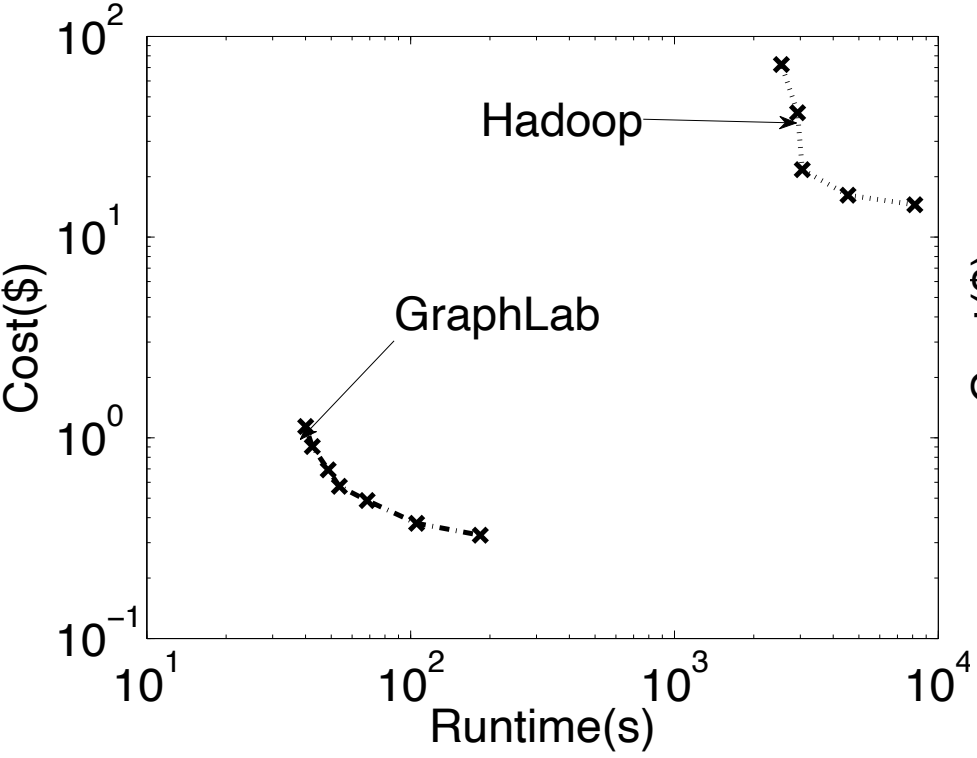
Consistency Model

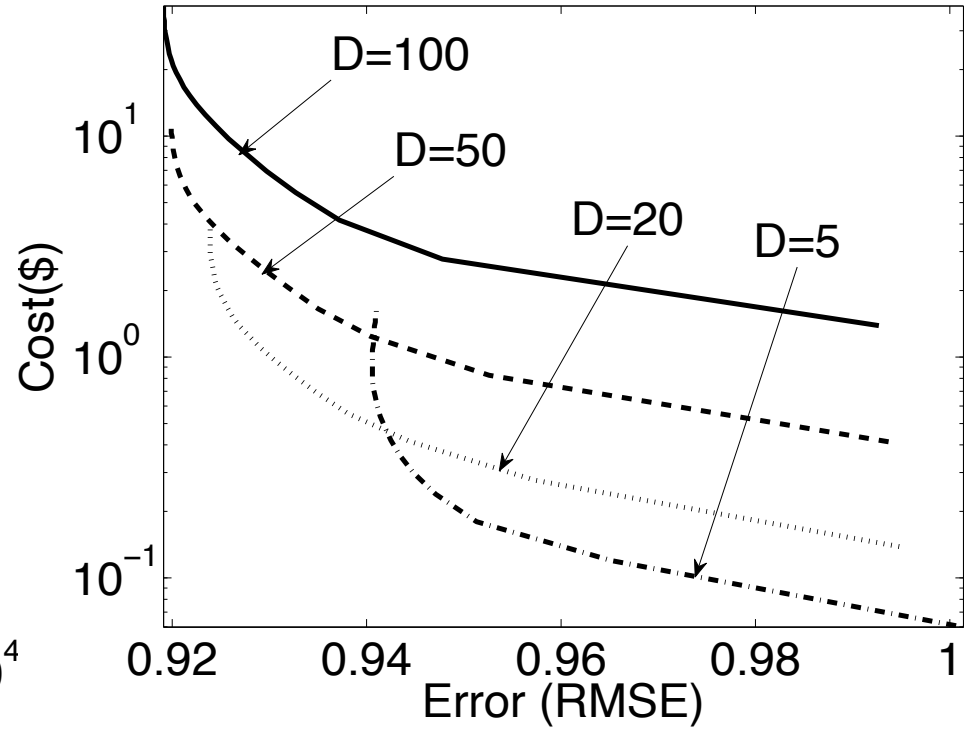


Algorithms Implemented in GraphLab (1.x)

- PageRank
- K-Means++
- Matrix Factorization
- 5-line codes for a real Recommendation Systems
- Label-Propagation
- Loopy Belief Propagation
- Gibbs Sampling
- CoEM
- Graphical Model Parameter Learning
- Probabilistic Matrix/Tensor Factorization
- Alternating Least Squares
- Lasso with Sparse Features
- Support Vector Machines with Sparse Features
- …

The Cost of Hadoop





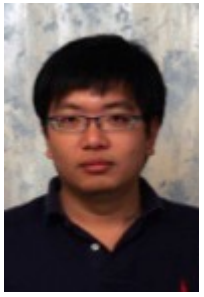
PowerGraph (GraphLab Ver.2)

Distributed Graph-Parallel Computation on Natural Graphs

Joseph Gonzalez



Joint work with:



Yucheng Low



Haijie Gu



Danny Bickson



Carlos Guestrin

Carnegie Mellon University

Problem:

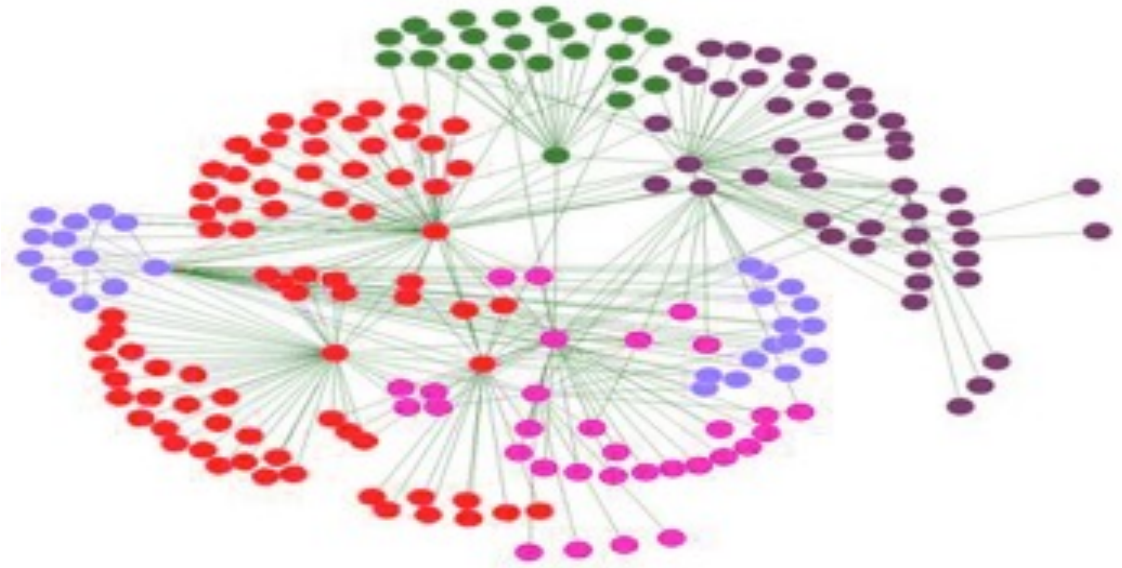
Existing ***distributed*** graph computation systems, including GraphLab v1.x, perform poorly on **Natural Graphs**.



Natural Graphs

Graphs derived from natural phenomena

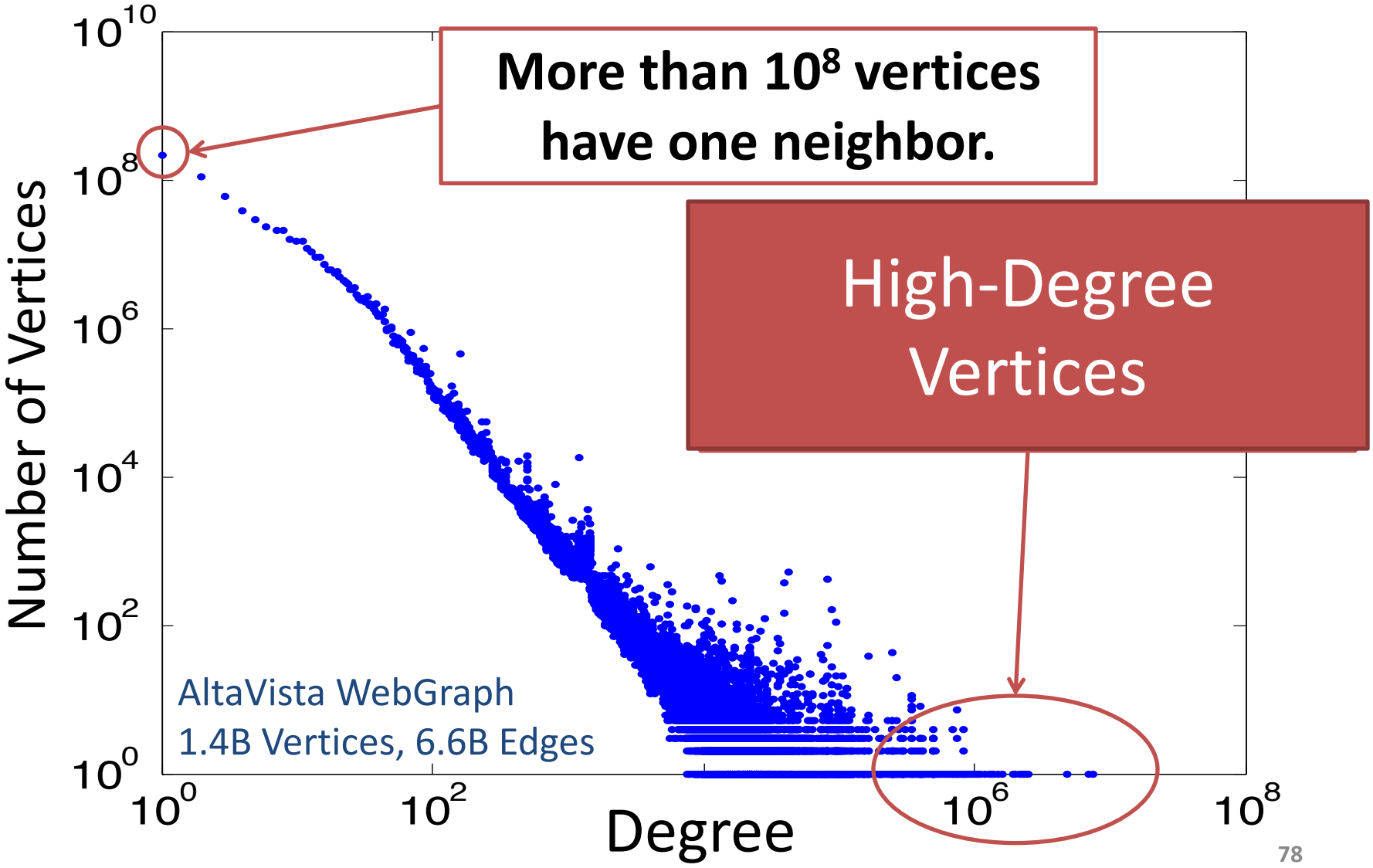
Properties of Natural Graphs



Power-Law Degree Distribution

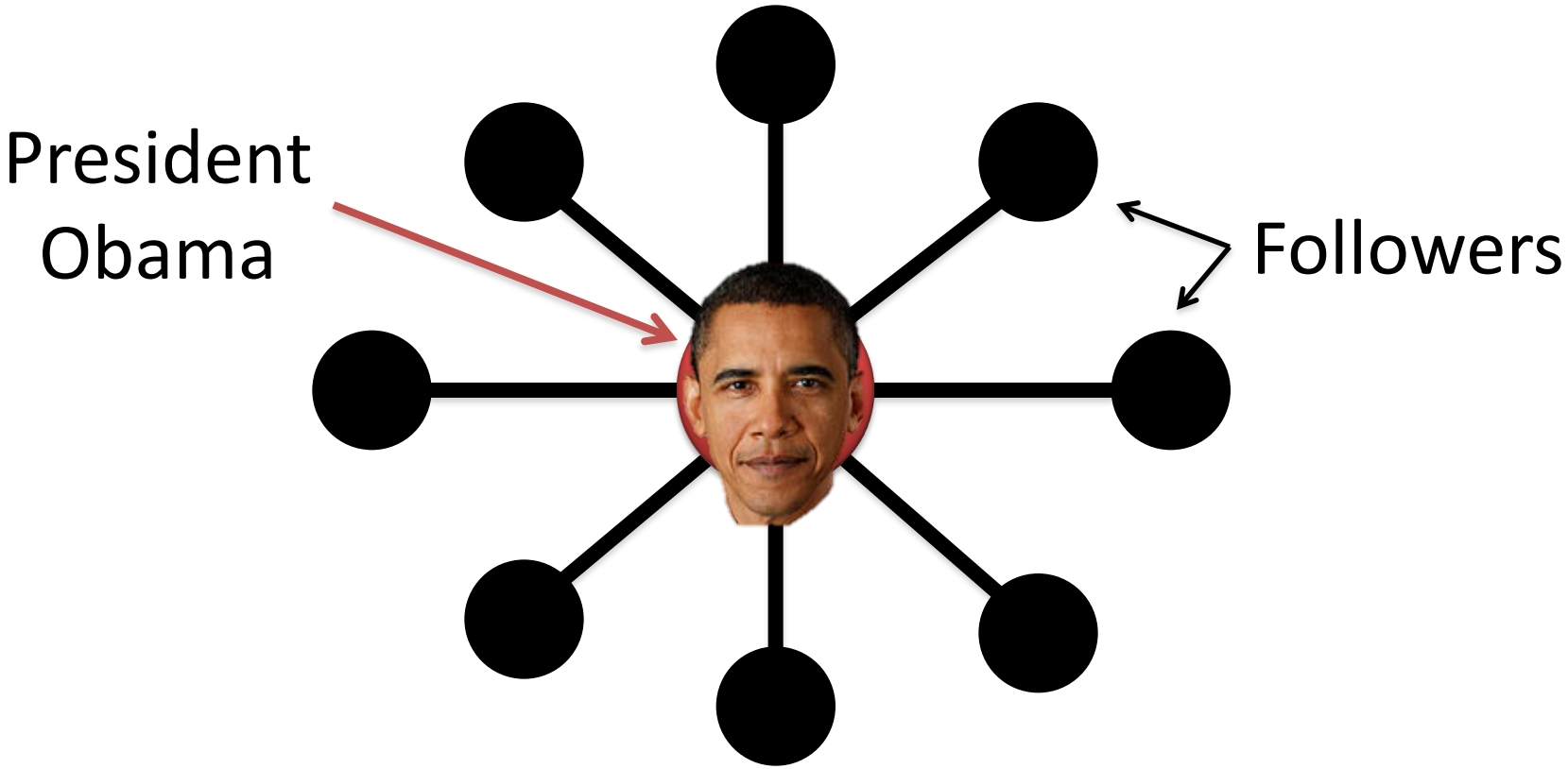
Reference: Zipf, Power-Laws and Pareto: A Ranking Tutorial, by L. Adamic, http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html

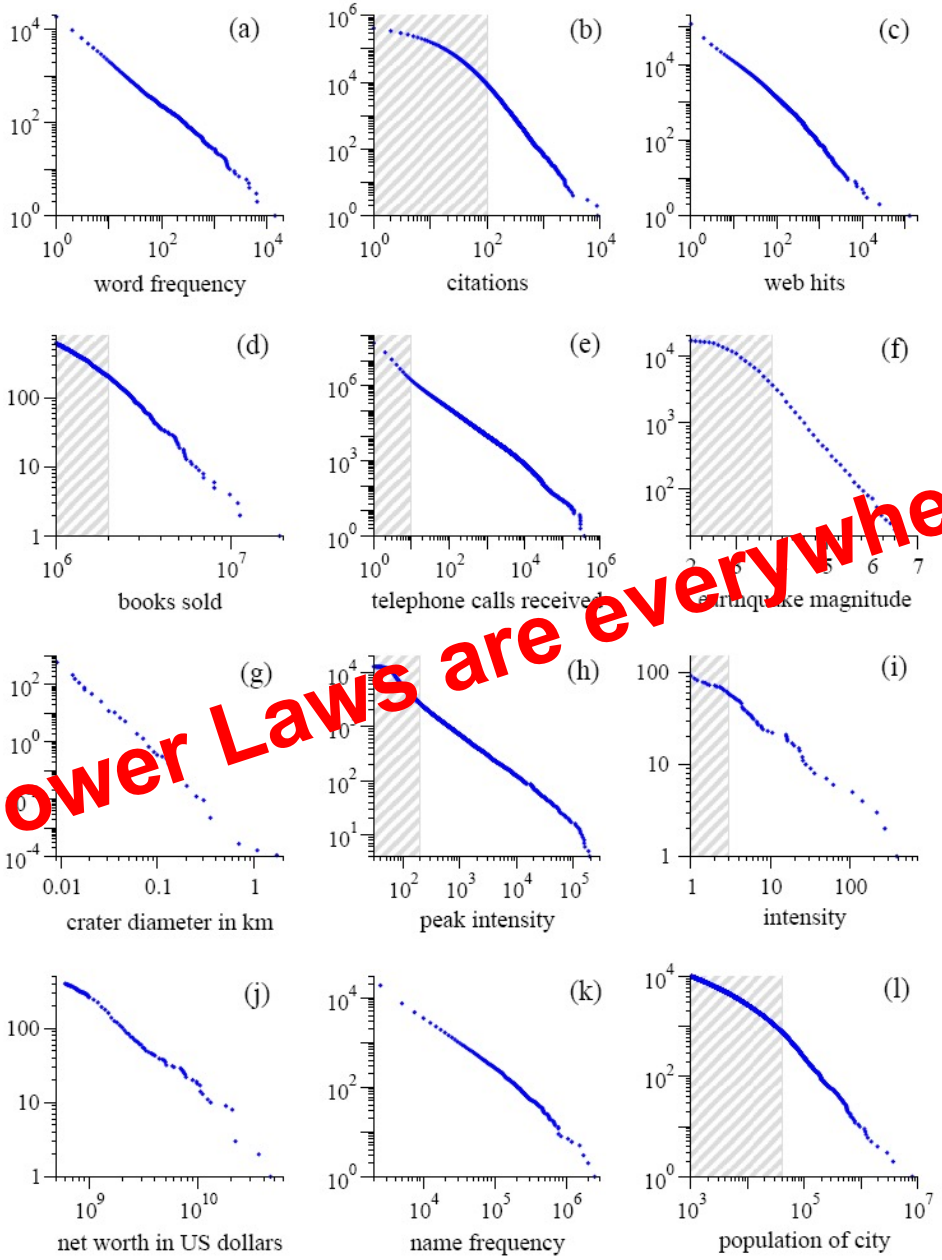
Power-Law Degree Distribution



Power-Law Degree Distribution

"Star Like" Motif





Power Laws are everywhere!

Figure from: Newman, M. E. J. (2005) "Power laws, Pareto distributions and Zipf's law." Contemporary Physics 46:323–351.

Properties of Natural Graphs

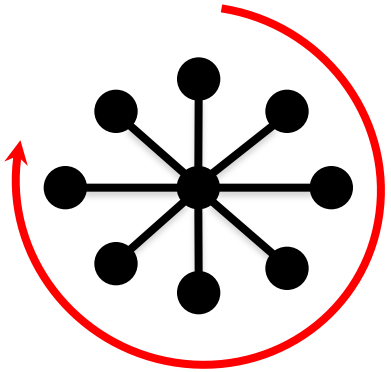


High-degree Vertices

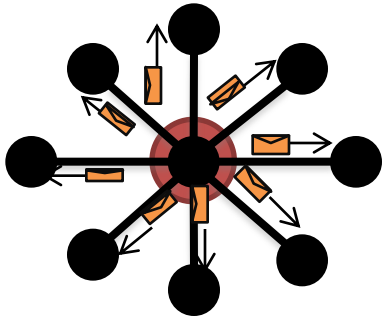
Power-Law Degree Distribution

Low Quality Partition

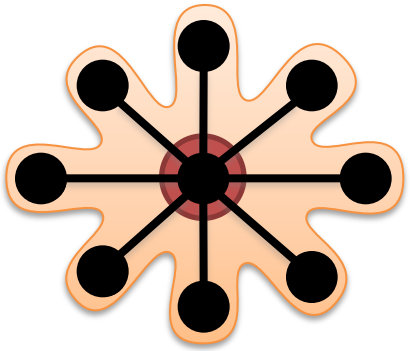
Challenges of **High-Degree** Vertices



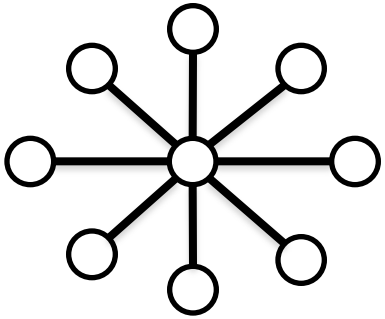
Sequentially process edges



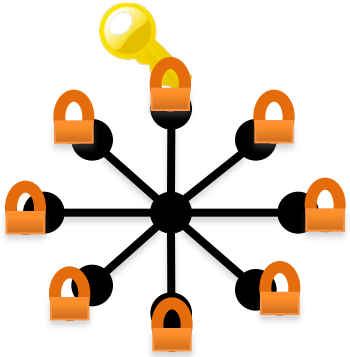
Sends many messages (Pregel)



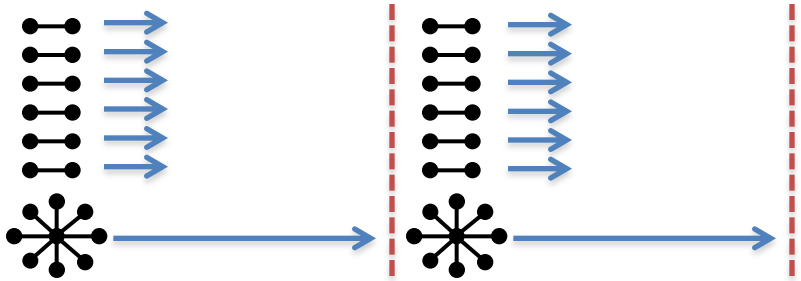
Touches a large fraction of graph (GraphLab)



Edge meta-data too large for single machine



Asynchronous Execution requires heavy locking (GraphLab)

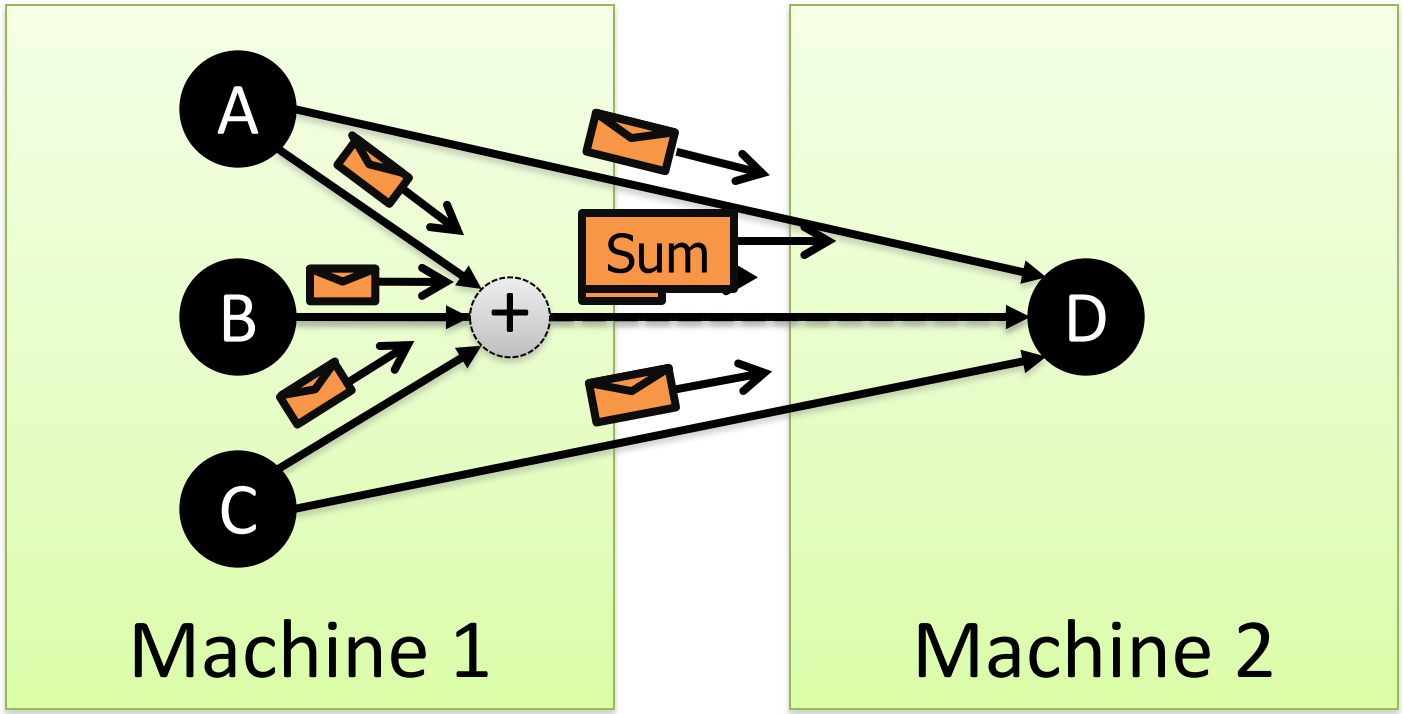


Synchronous Execution prone to stragglers (Pregel)

Communication Overhead for High-Degree Vertices

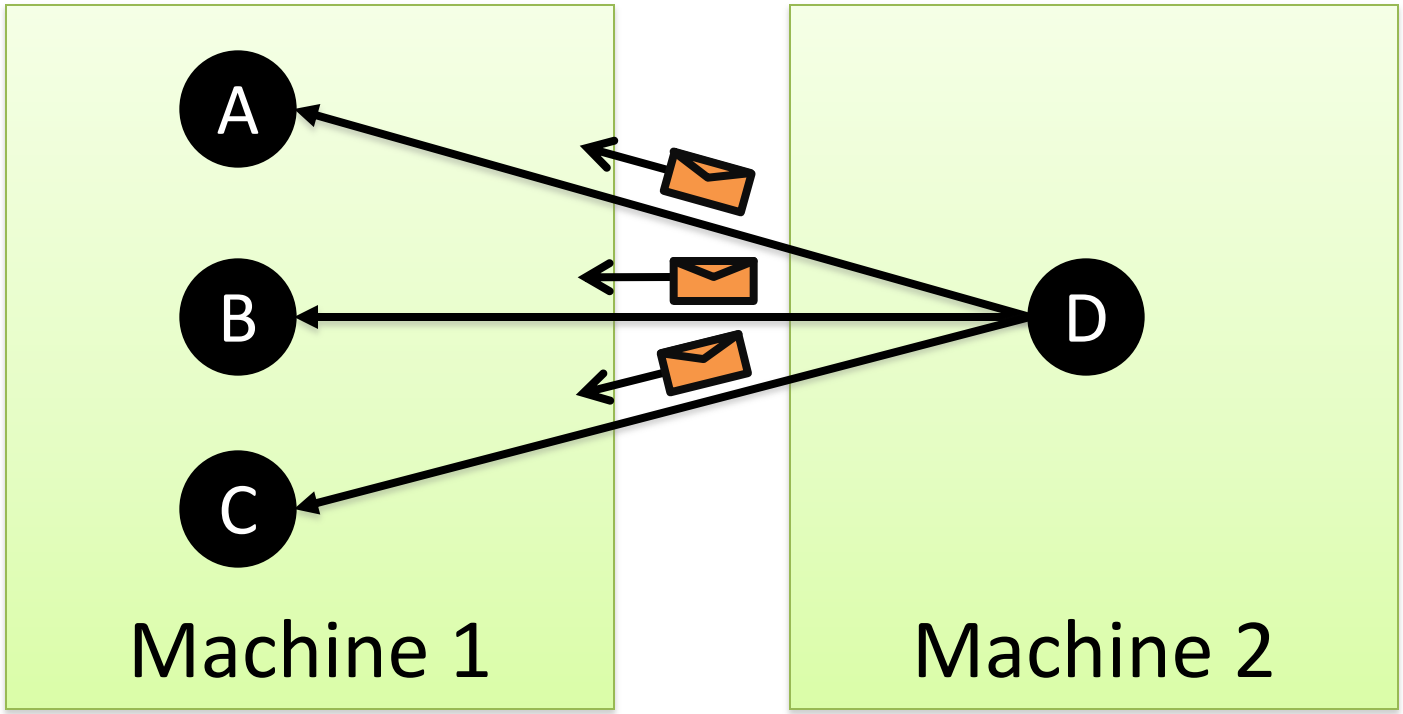
Fan-In vs. Fan-Out

Pregel **Message Combiners** on **Fan-In**



- User defined **commutative associative** (+) message operation:

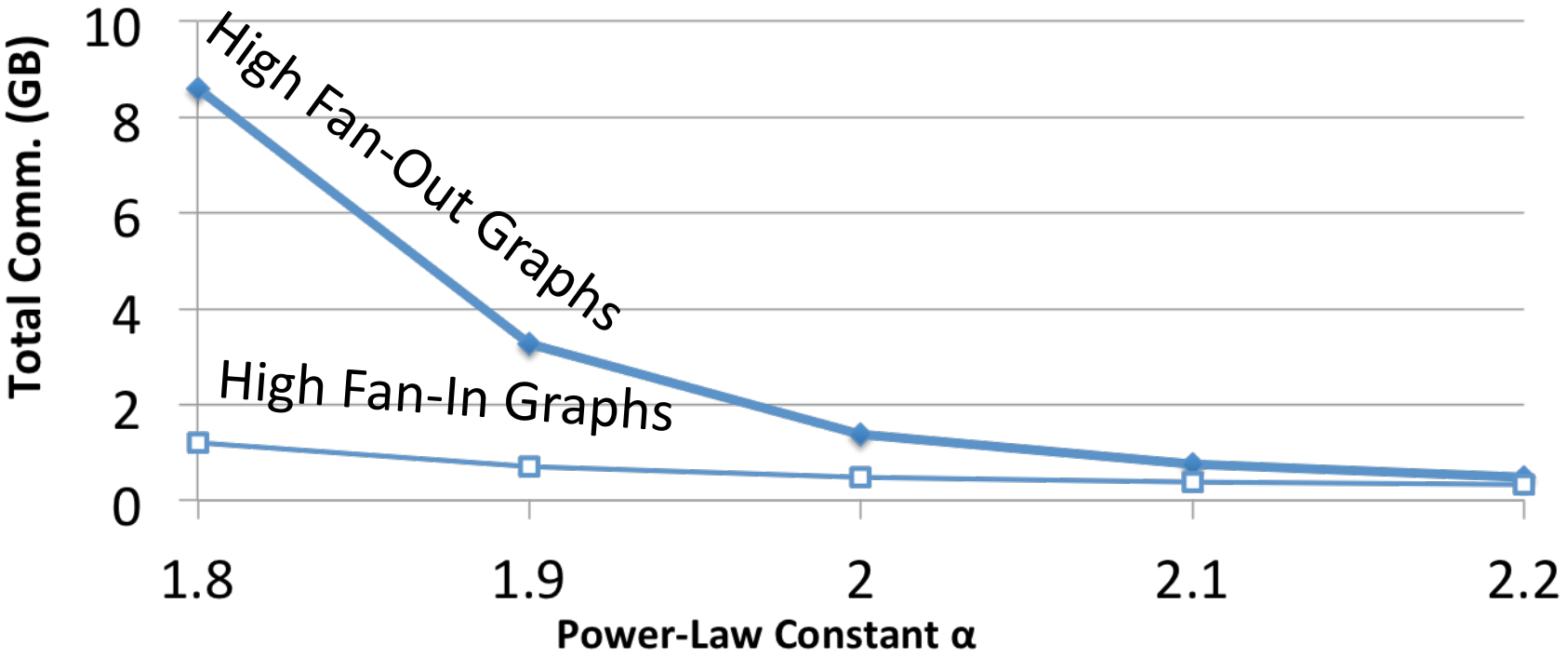
Pregel Struggles with **Fan-Out**



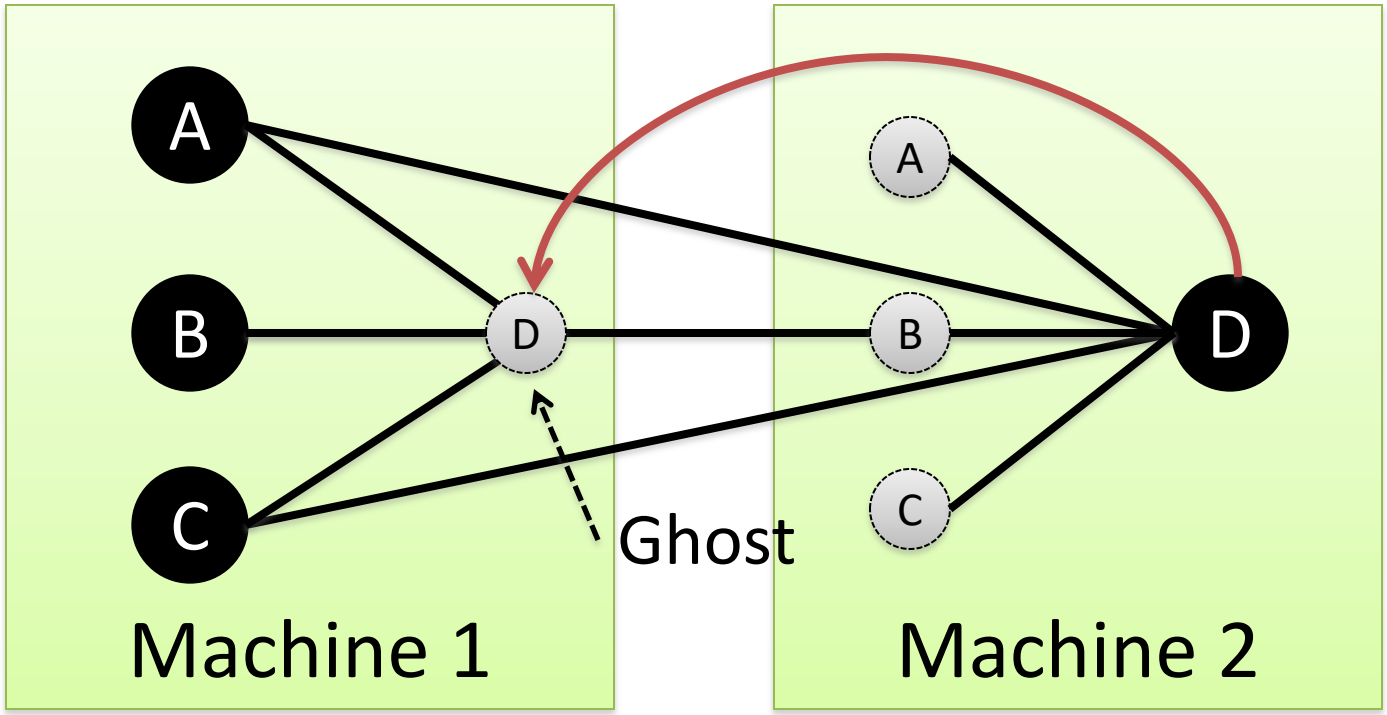
- **Broadcast** sends many copies of the same message to the same machine!

Fan-In and Fan-Out Performance

- PageRank on synthetic Power-Law Graphs
  - Piccolo was used to simulate Pregel with combiners

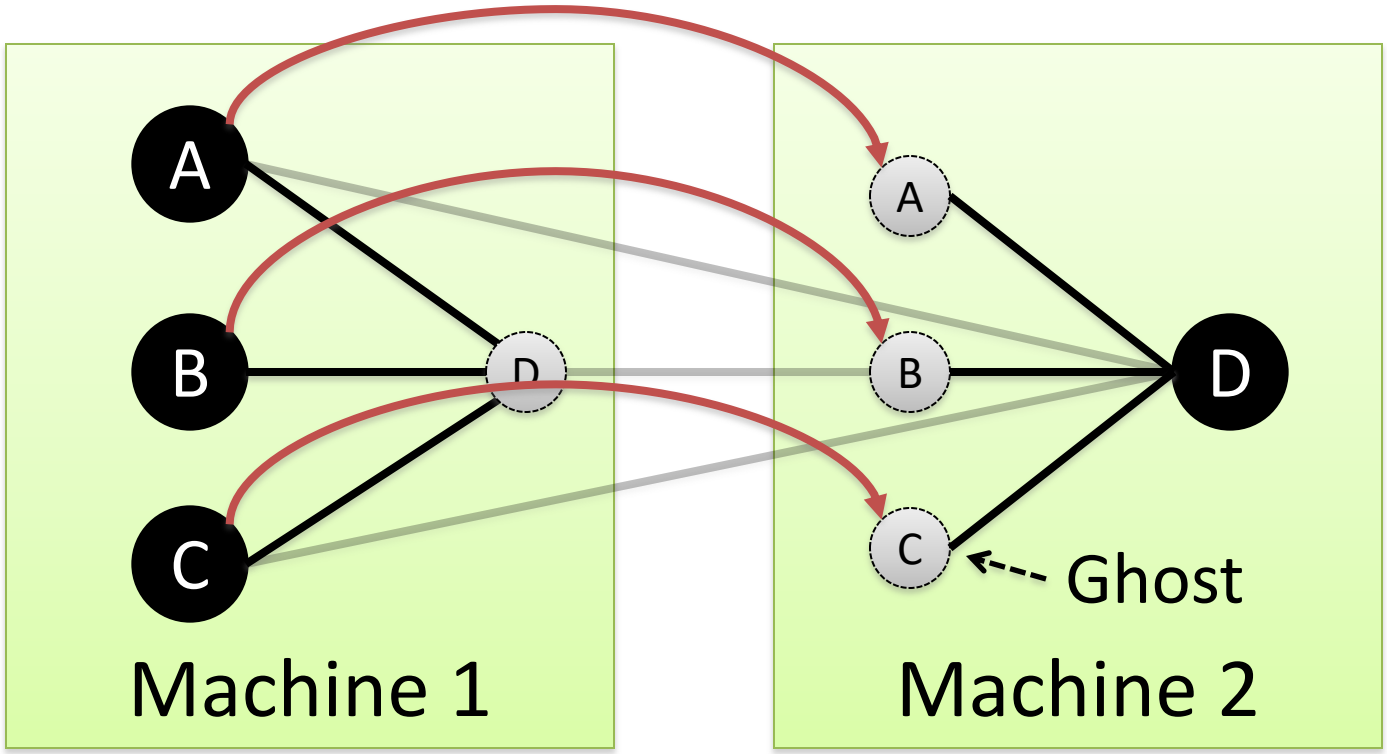


GraphLab Ghosting



- Changes to master are synced to ghosts

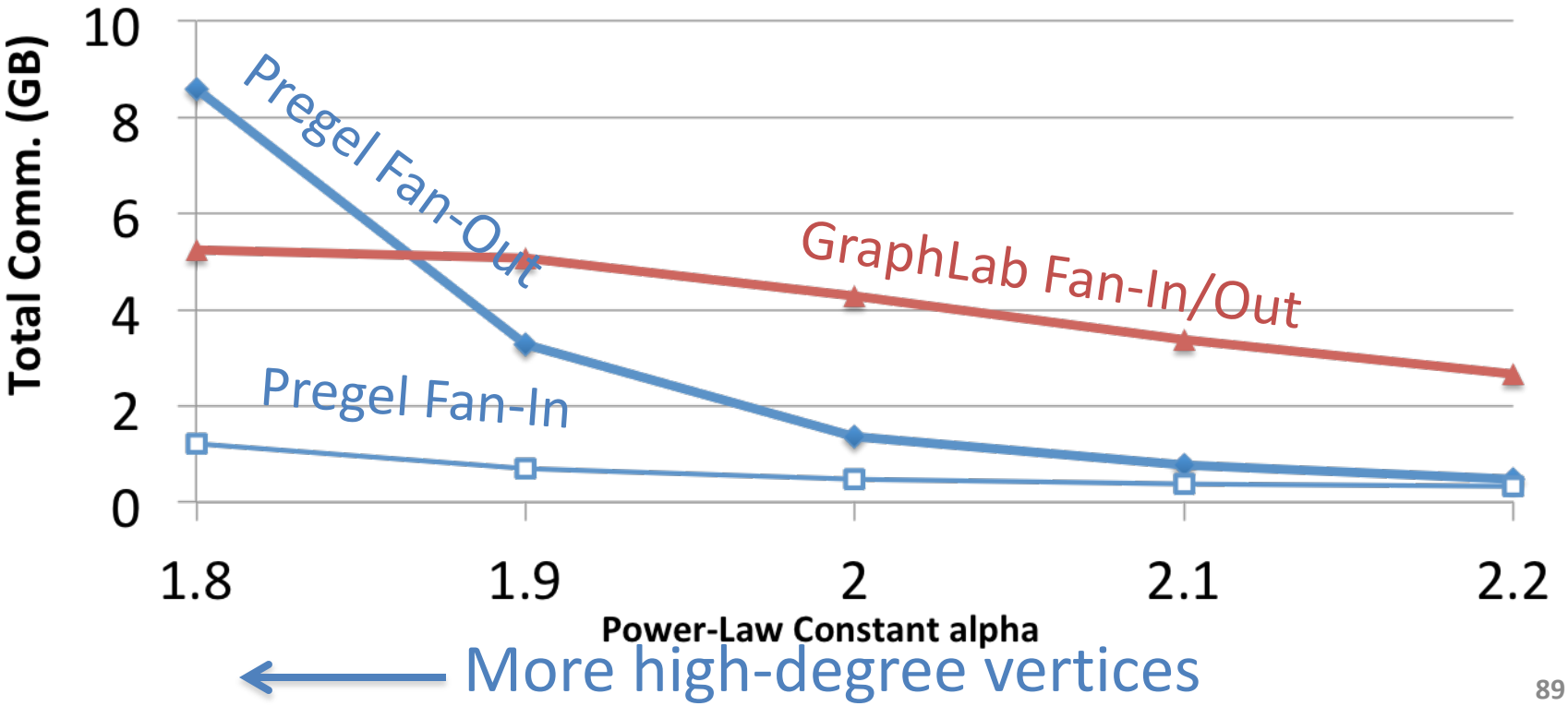
GraphLab Ghosting



- Changes to **neighbors** of **high degree vertices** creates substantial network traffic

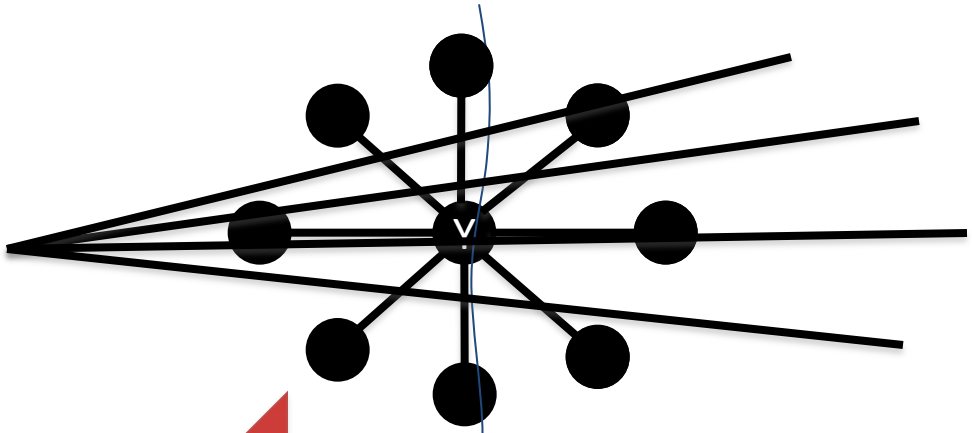
Fan-In and Fan-Out Performance

- PageRank on synthetic Power-Law Graphs
- GraphLab is **undirected**

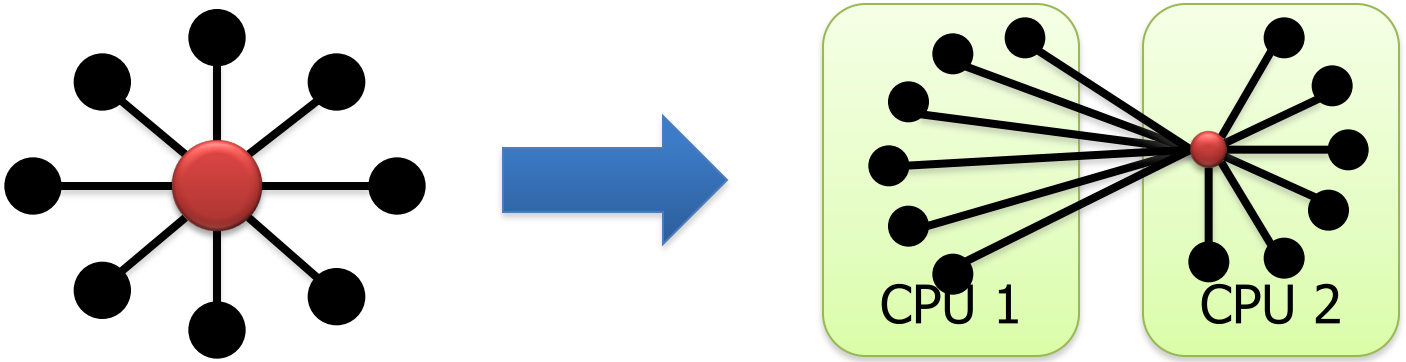


Graph Partitioning

- Graph parallel abstractions rely on partitioning:
  - Minimize communication
  - Balance computation and storage



Power-Law Graphs are **Difficult to Partition**

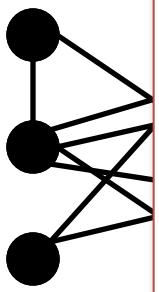


- Power-Law graphs do not have **low-cost** balanced cuts *[Leskovec et al. 08, Lang 04]*
- Traditional graph-partitioning algorithms perform poorly on Power-Law Graphs. *[Abou-Rjeili et al. 06]*

Random Partitioning

- Both GraphLab and Pregel resort to **random** (hashed) partitioning on **natural graphs**

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|E|}\right] = 1 - \frac{1}{p}$$



10 Machines → 90% of edges cut

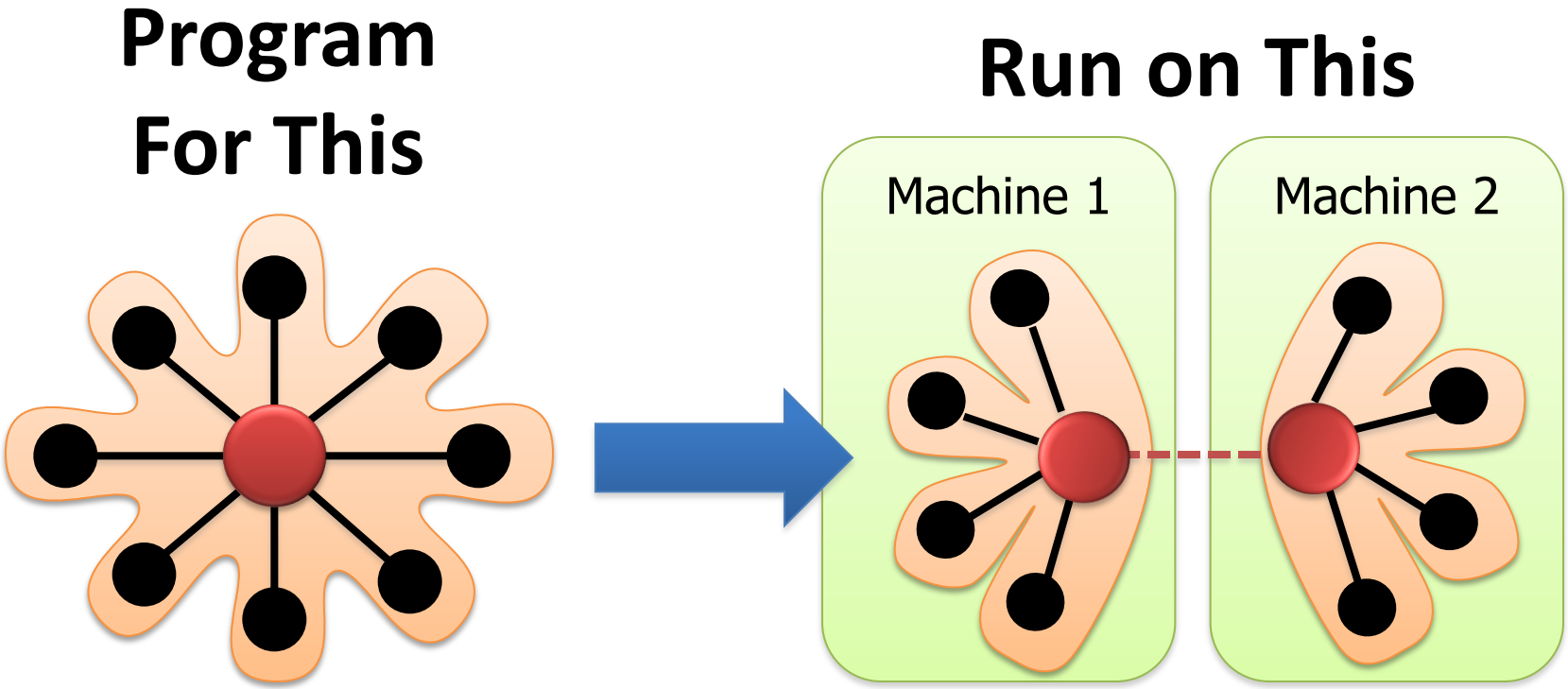
100 Machines → 99% of edges cut!

PowerGraph

- **GAS Decomposition**: distribute vertex-programs
  - Move computation to data
  - Parallelize **high-degree** vertices

- **Vertex Partitioning:**
  - Effectively distribute large power-law graphs

PowerGraph



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

Minimizing Communication in PowerGraph



Communication is linear in the number of machines each vertex spans

A **vertex-cut** minimizes machines each vertex spans

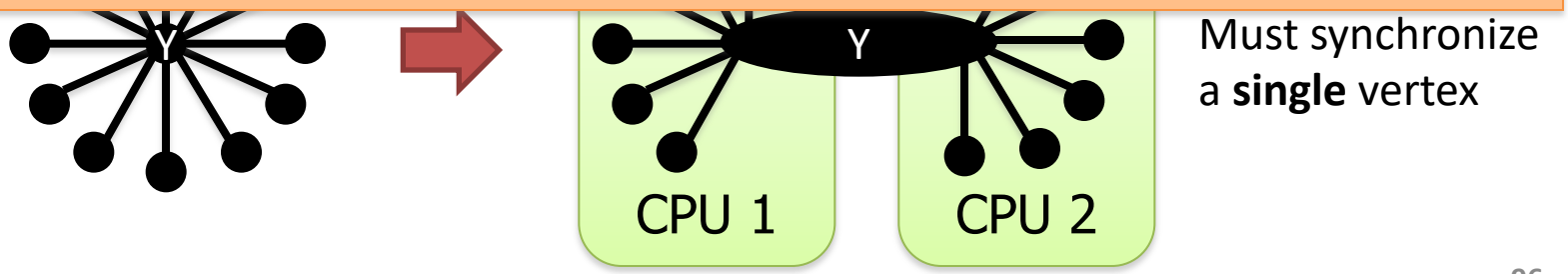
*Percolation theory suggests that power law graphs have **good vertex cuts**. [Albert et al. 2000]*

New Approach to Partitioning

- Rather than cut edges:
-

New Theorem:

*For **any edge-cut** we can directly construct a vertex-cut which requires **strictly less** communication and storage.*



A **Common Pattern** for Vertex-Programs

GraphLab_PageRank(i)

```
// Compute sum over neighbors
total = 0
foreach( j in in_neighbors(i)):
  total = total + R[j] * w_ji
```

Gather Information About Neighborhood

```
// Update the PageRank
R[i] = 0.1 + total
```

Update Vertex

```
// Trigger neighbors to run again
if R[i] not converged then
  foreach( j in out_neighbors(i))
    signal vertex-program on j
```

Signal Neighbors & Modify Edge Data

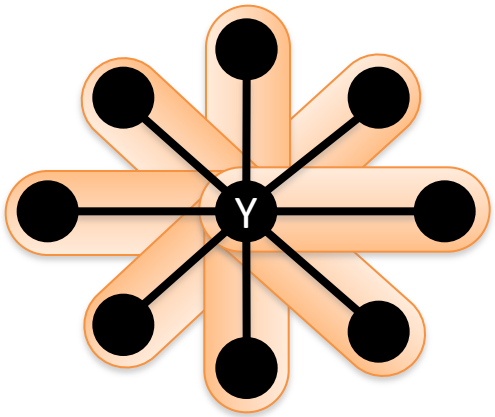
GAS Decomposition

Gather (Reduce)

Accumulate information about neighborhood

User Defined:

- **Gather**(Y) → Σ
- $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$



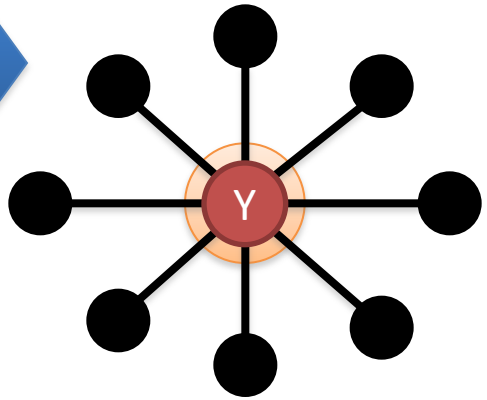
Parallel Sum  → Σ

Apply

Apply the accumulated value to center vertex

User Defined:

- **Apply**(Y , Σ) → Y'

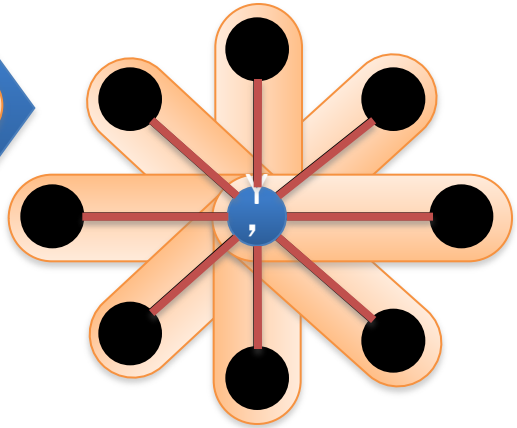


Scatter

Update adjacent edges and vertices.

User Defined:

- **Scatter**(Y') → _



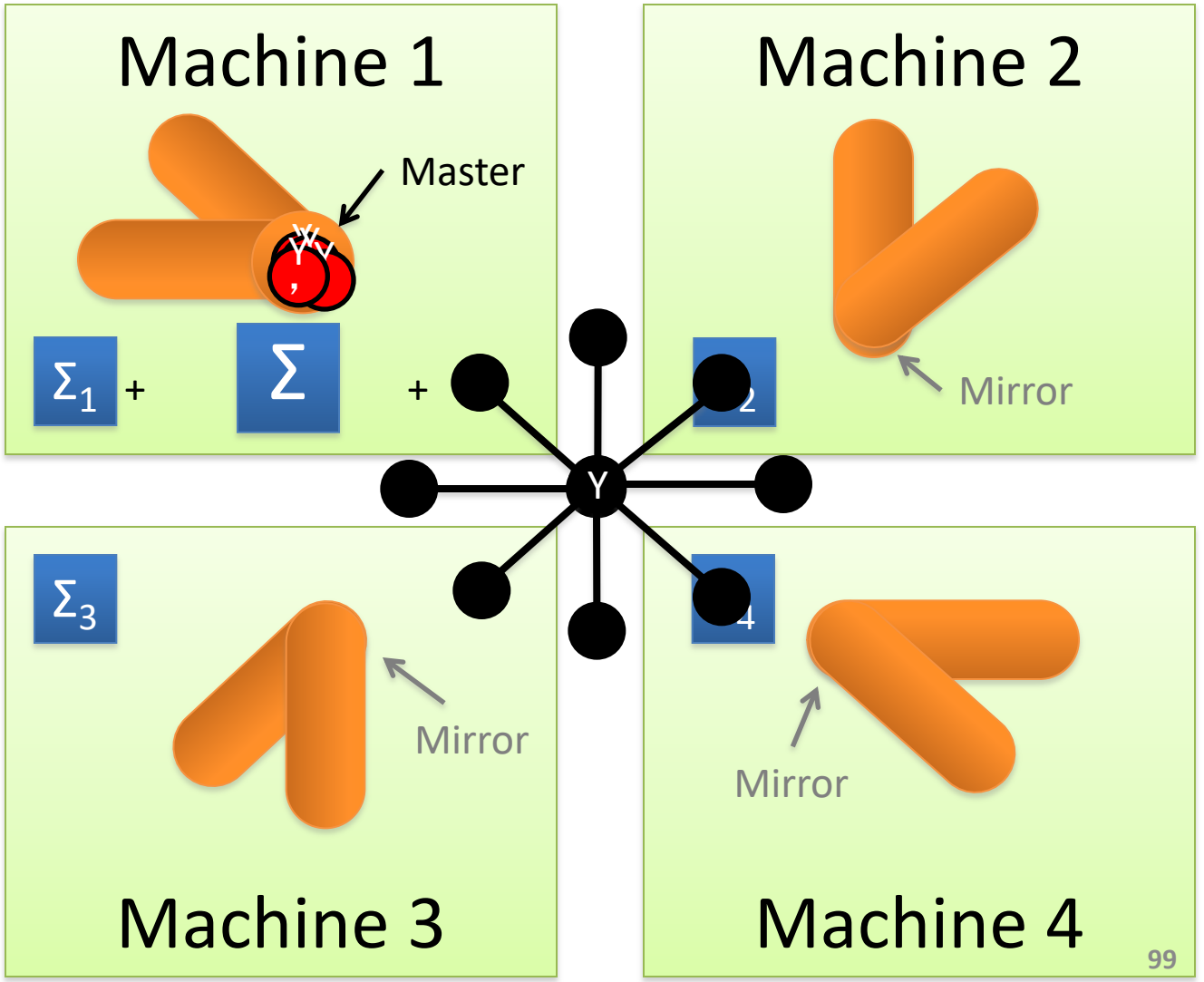
Update Edge Data & Activate Neighbors

Distributed Execution of a PowerGraph Vertex-Program

Gather

Apply

Scatter



PageRank in PowerGraph

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

PowerGraph_PageRank(i)

Gather(j → i) : return $w_{ji} * R[j]$

sum(a, b) : return a + b;

Apply(i, Σ) : $R[i] = 0.15 + \Sigma$

Scatter(i → j) :

if *R[i]* changed then trigger *j* to be **recomputed**

Constructing Vertex-Cuts

- **Evenly** assign **edges** to machines
  - Minimize machines spanned by each vertex
- Assign each edge **as it is loaded**
  - Touch each edge only once
- Propose three **distributed** approaches:
  - ***Random*** *Edge Placement*
  - ***Coordinated Greedy*** *Edge Placement*
  - ***Oblivious Greedy*** *Edge Placement*

Random Edge-Placement

- Randomly assign edges to machines

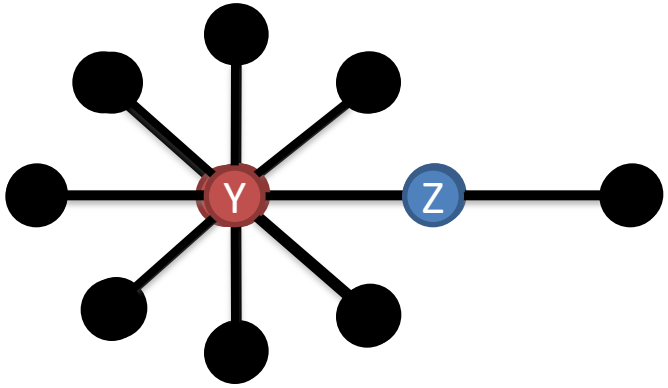
Machine 1	Machine 2	Machine 3

Balanced Vertex-Cut

Y Spans 3 Machines

Z Spans 2 Machines

Not cut!

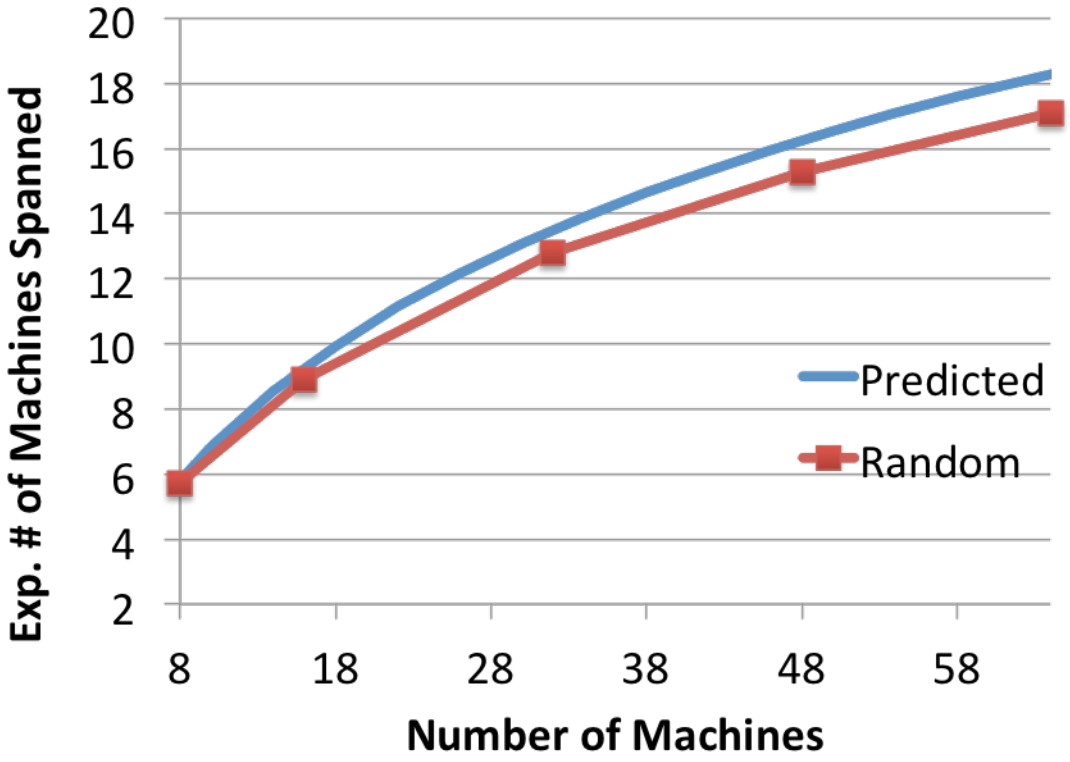


Analysis Random Edge-Placement

- Expected number of machines spanned by a vertex:

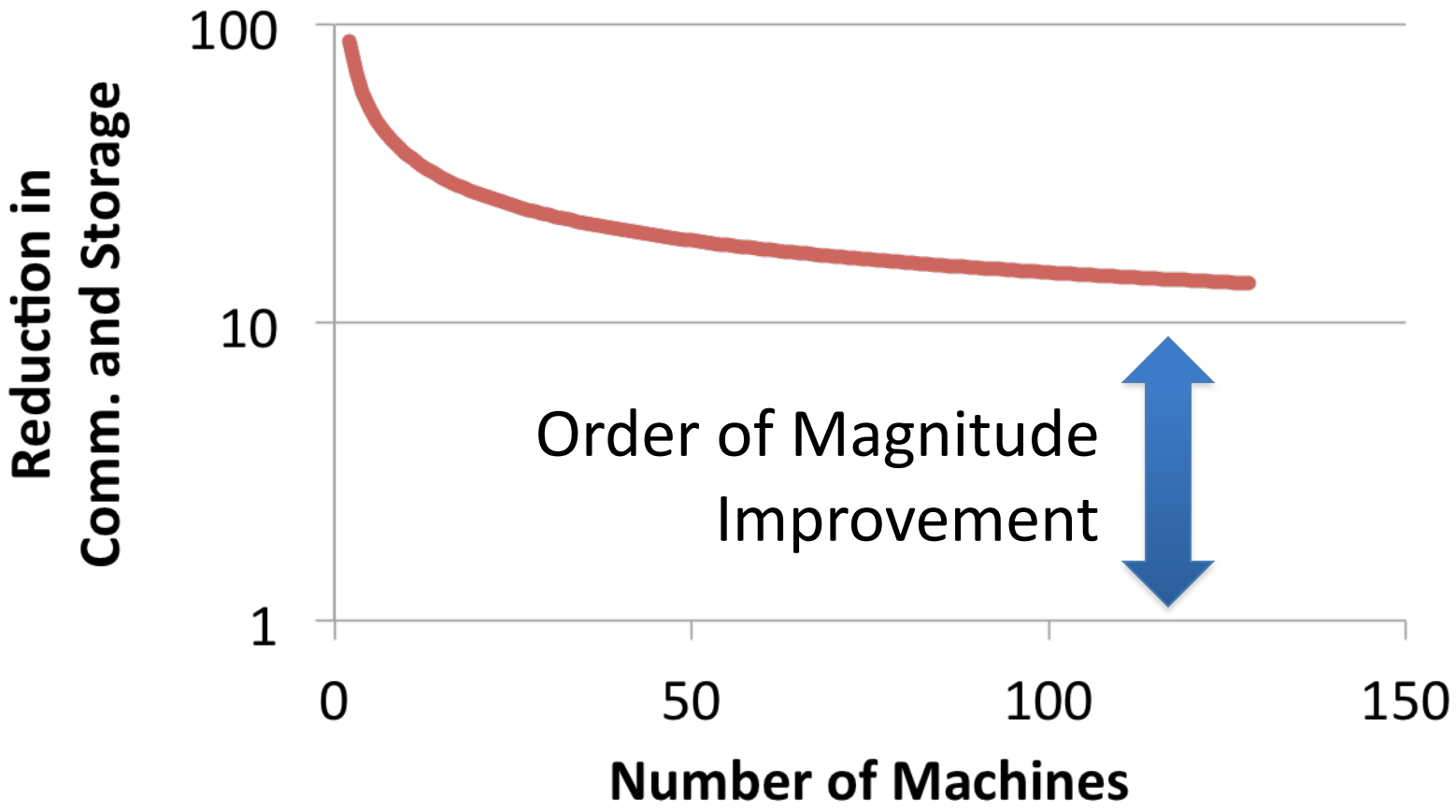
Twitter Follower Graph
41 Million Vertices
1.4 Billion Edges

Accurately Estimate Memory and Comm. Overhead



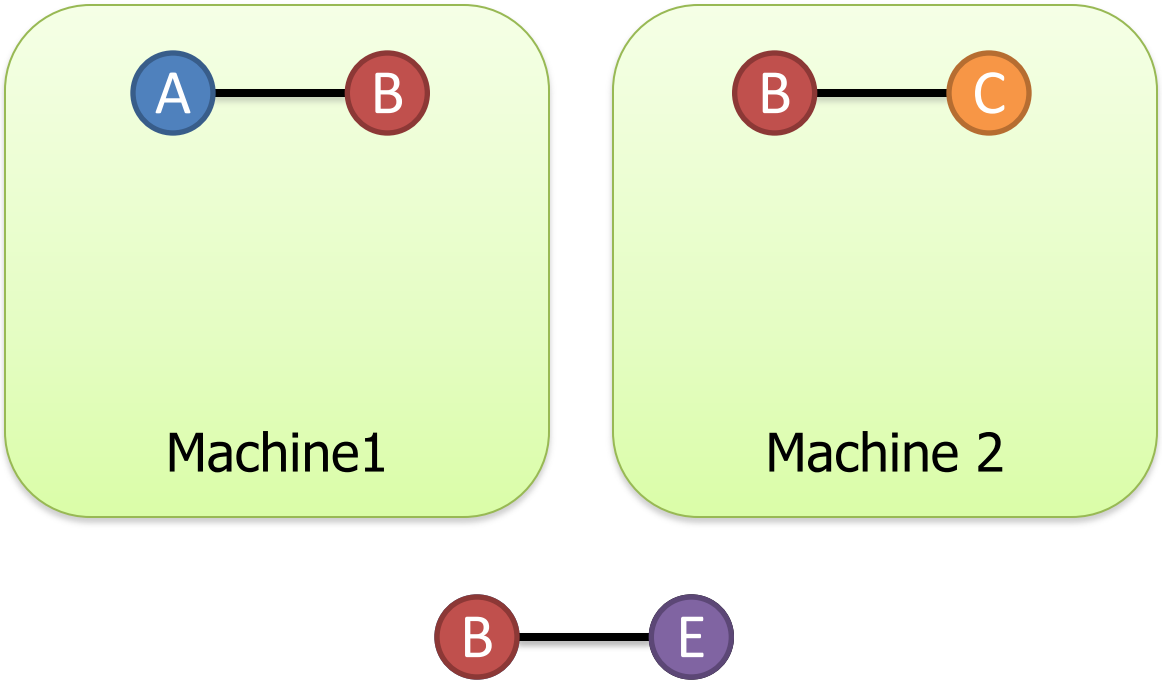
Random Vertex-Cuts vs. Edge-Cuts

- Expected improvement from vertex-cuts:



Greedy Vertex-Cuts

- Place edges on machines which already have the vertices in that edge.



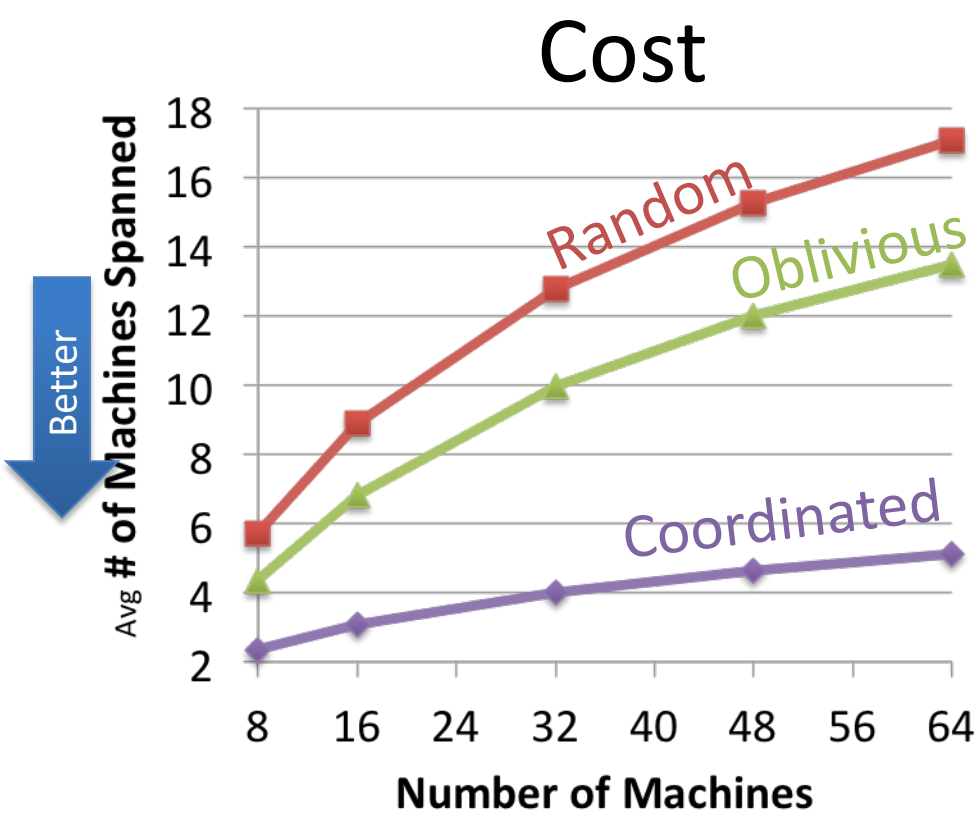
Greedy Vertex-Cuts

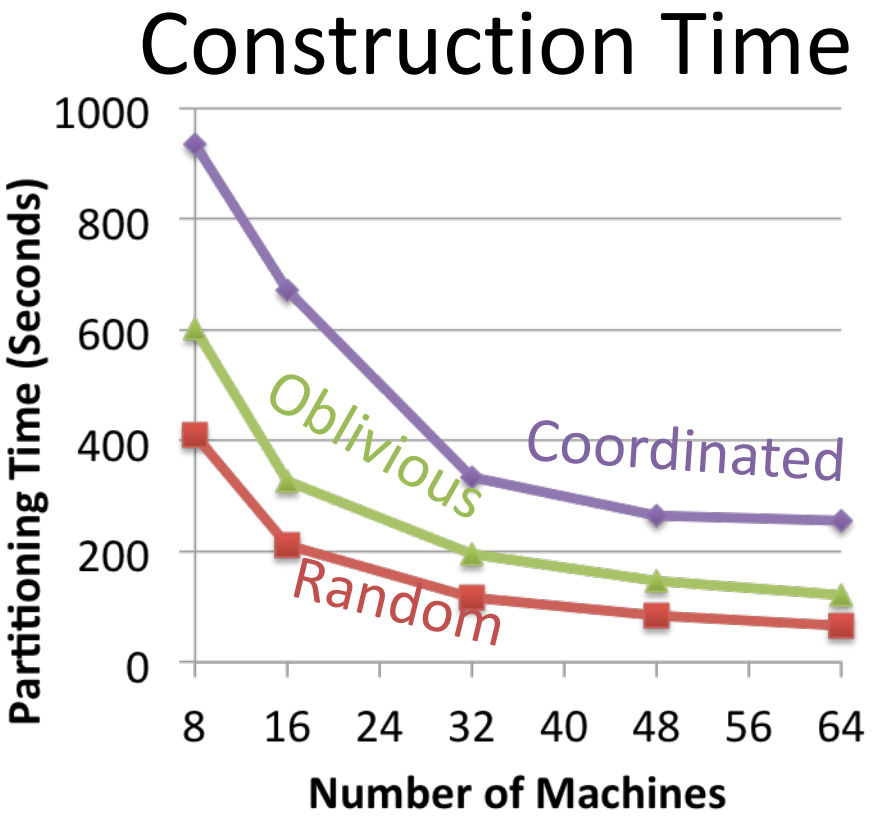
- **De-randomization** → greedily minimizes the expected number of machines spanned

- **Coordinated** Edge Placement
  - Requires coordination to place each edge
  - Slower: higher quality cuts
- **Oblivious** Edge Placement
  - Approx. greedy objective without coordination
  - Faster: lower quality cuts

Partitioning Performance

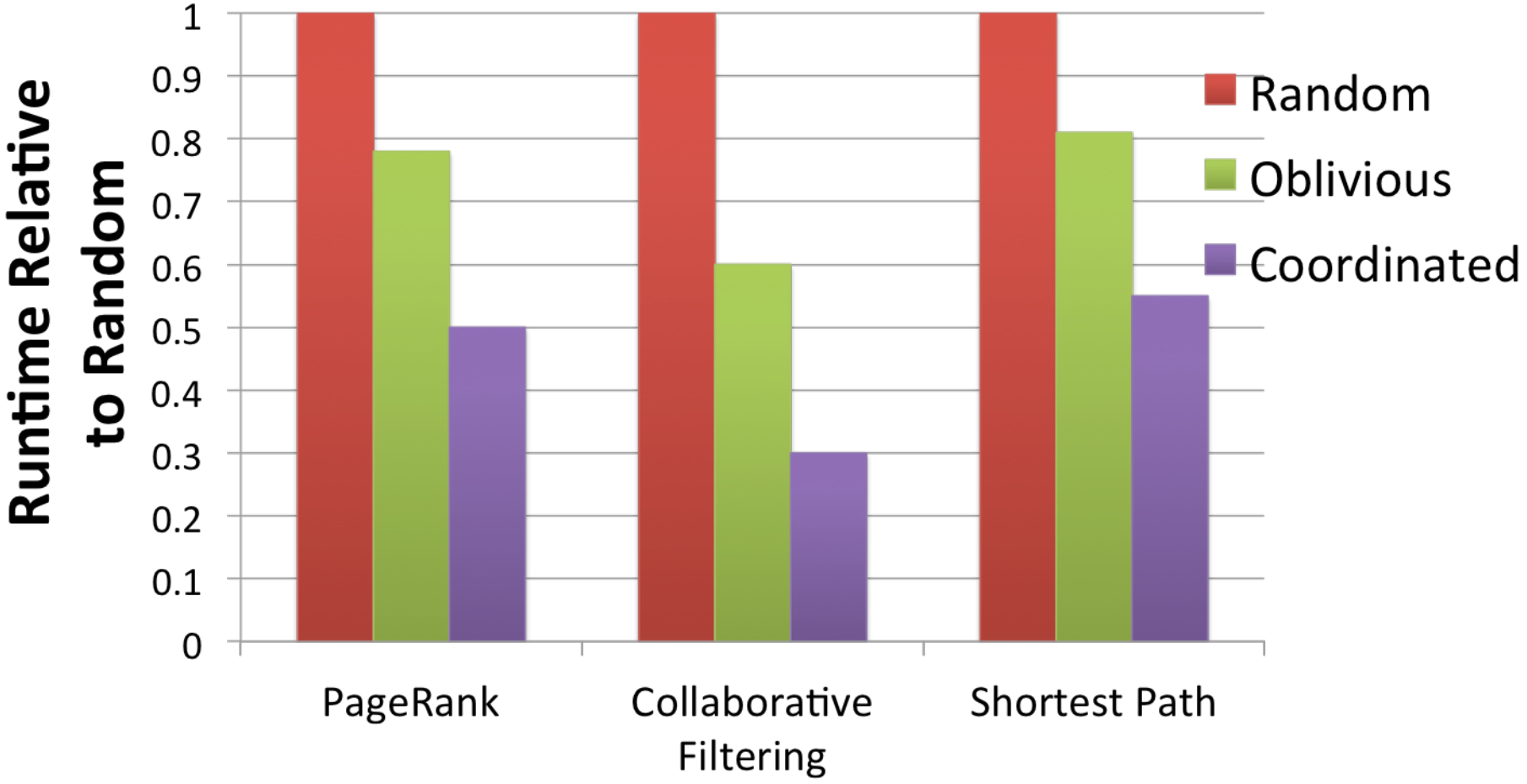
Twitter Graph: 41M vertices, 1.4B edges





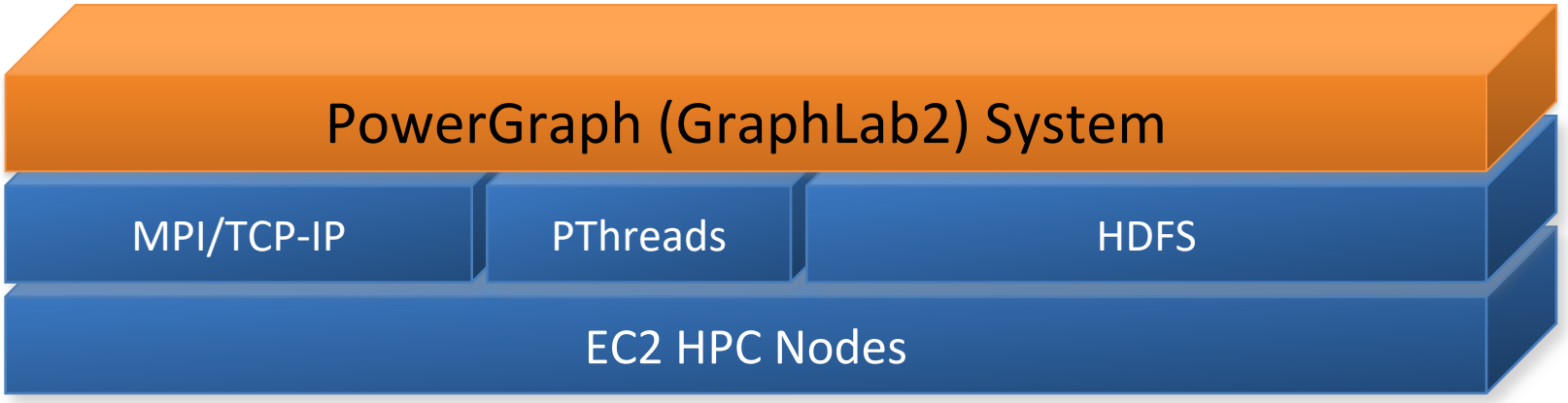
Oblivious balances cost and partitioning time.

Greedy Vertex-Cuts Improve Performance



Greedy partitioning improves computation performance.

PowerGraph System Design



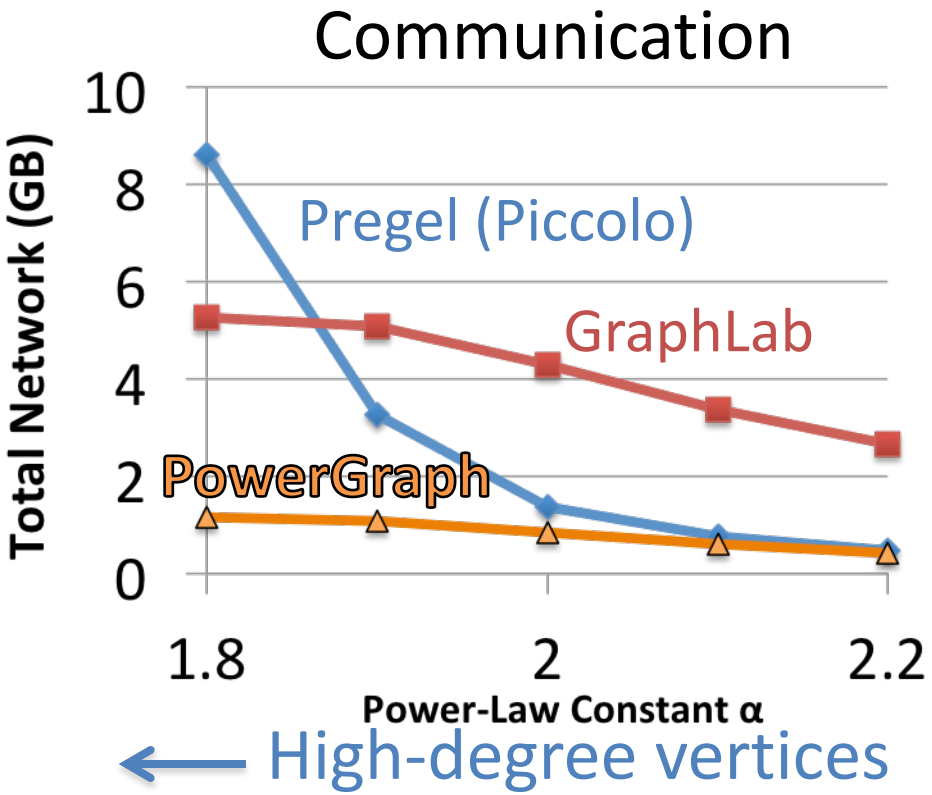
- Implemented as C++ API
- Uses HDFS for Graph Input and Output
- Fault-tolerance is achieved by check-pointing
  - Snapshot time < 5 seconds for twitter network

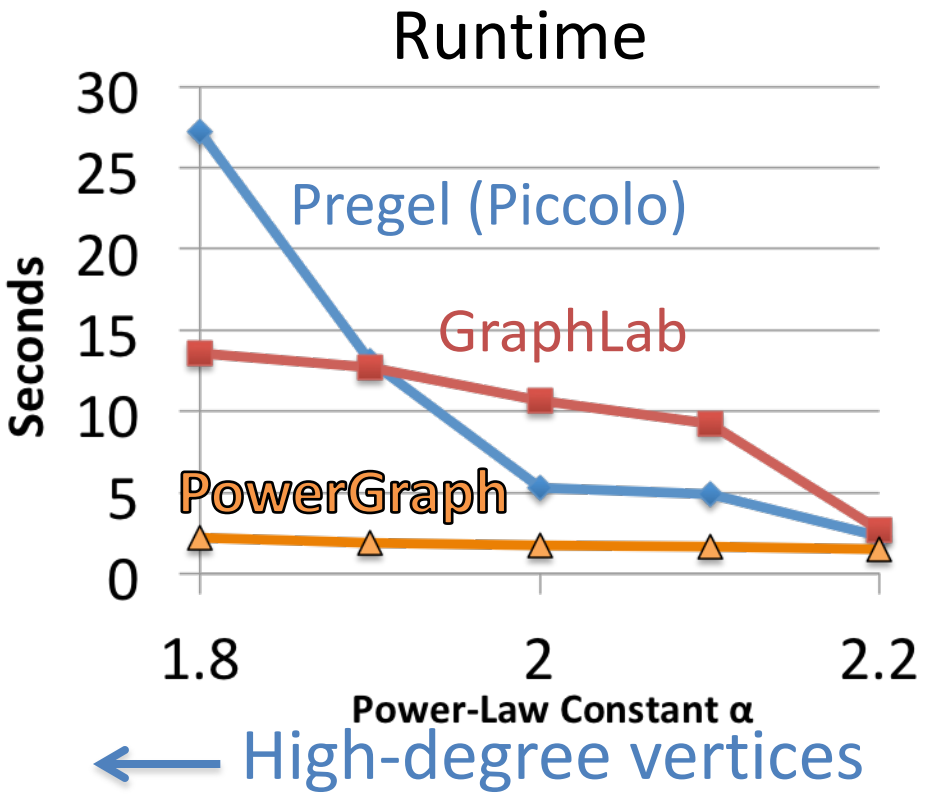
Implemented Many Algorithms

- **Collaborative Filtering**
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - SVD
  - Non-negative MF
- **Statistical Inference**
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- **Graph Analytics**
  - PageRank
  - Triangle Counting
  - Shortest Path
  - Graph Coloring
  - K-core Decomposition
- **Computer Vision**
  - Image stitching
- **Language Modeling**
  - LDA

Comparison with GraphLab & Pregel

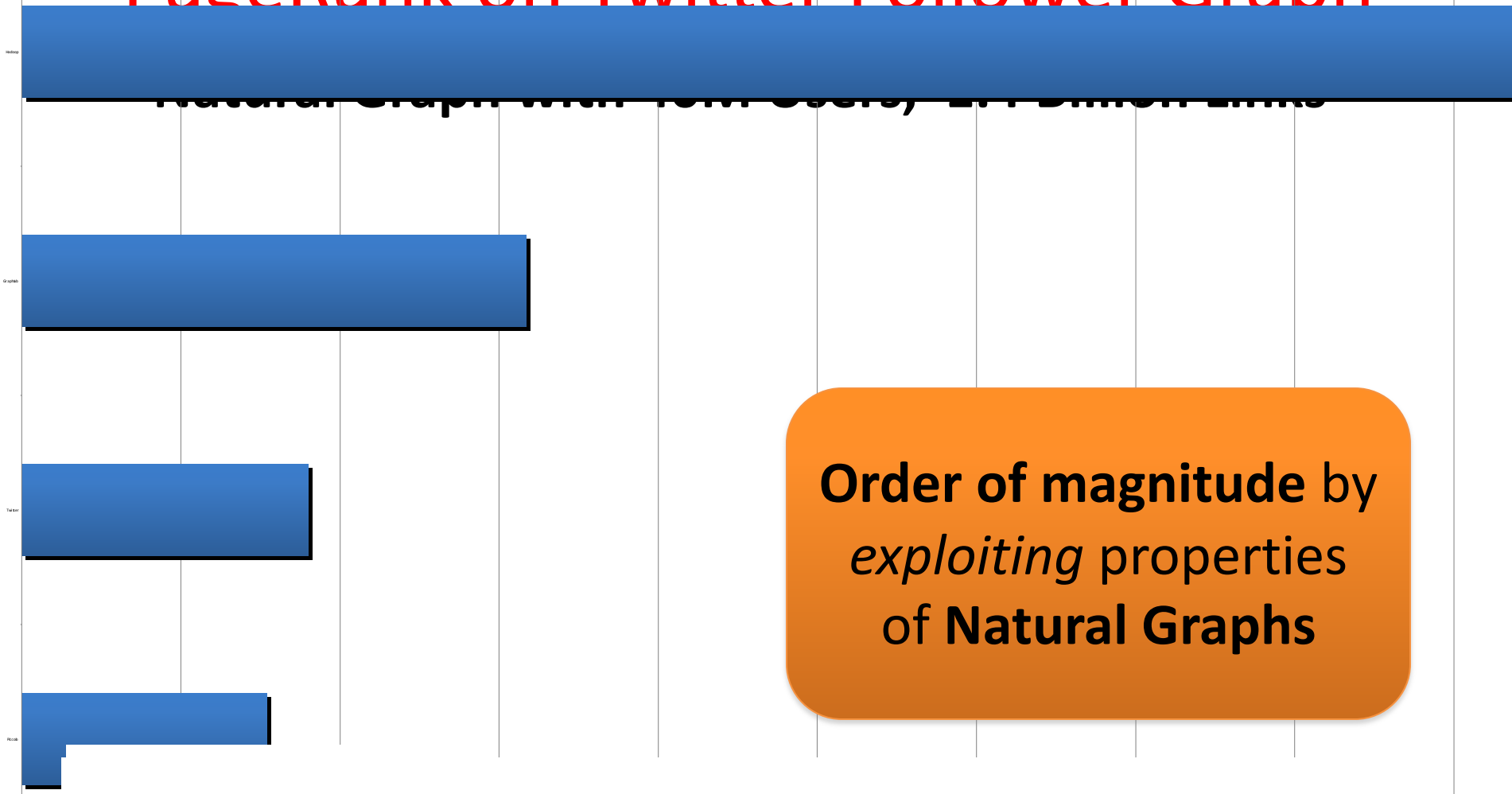
- PageRank on Synthetic Power-Law Graphs:





PowerGraph is robust to **high-degree** vertices.

PageRank on Twitter Follower Graph



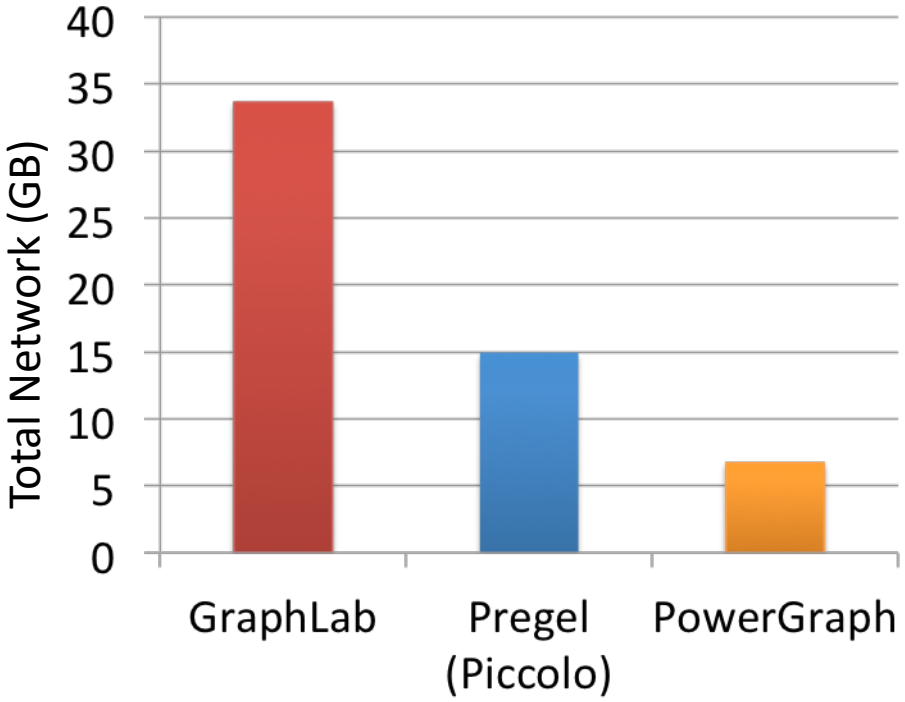
Order of magnitude by *exploiting* properties of **Natural Graphs**

Hadoop results from [Kang et al. '11]
Twister (in-memory MapReduce) [Ekanayake et al. '10]

PageRank on the Twitter Follower Graph

Natural Graph with 40M Users, 1.4 Billion Links

Communication



Runtime

Reduces Communication

Runs Faster

32 Nodes x 8 Cores (EC2 HPC cc1.4x)

PowerGraph is Scalable

Yahoo Altavista Web Graph (2002):

One of the largest publicly available web graphs

1.4 Billion Webpages, 6.6 Billion Links

7 Seconds per Iter.

1B links processed per second

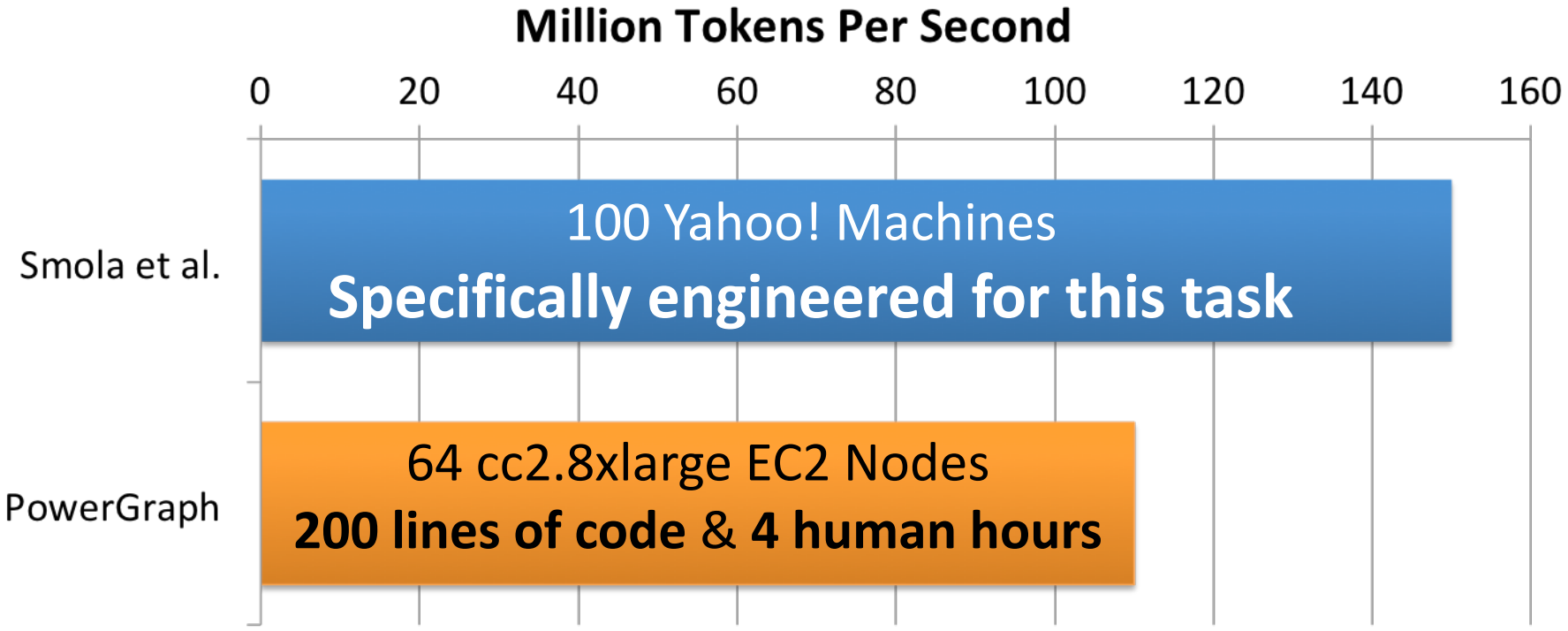
30 lines of user code

amazon web services™
64 HPC Nodes
intel
(2048 HT)

Topic Modeling



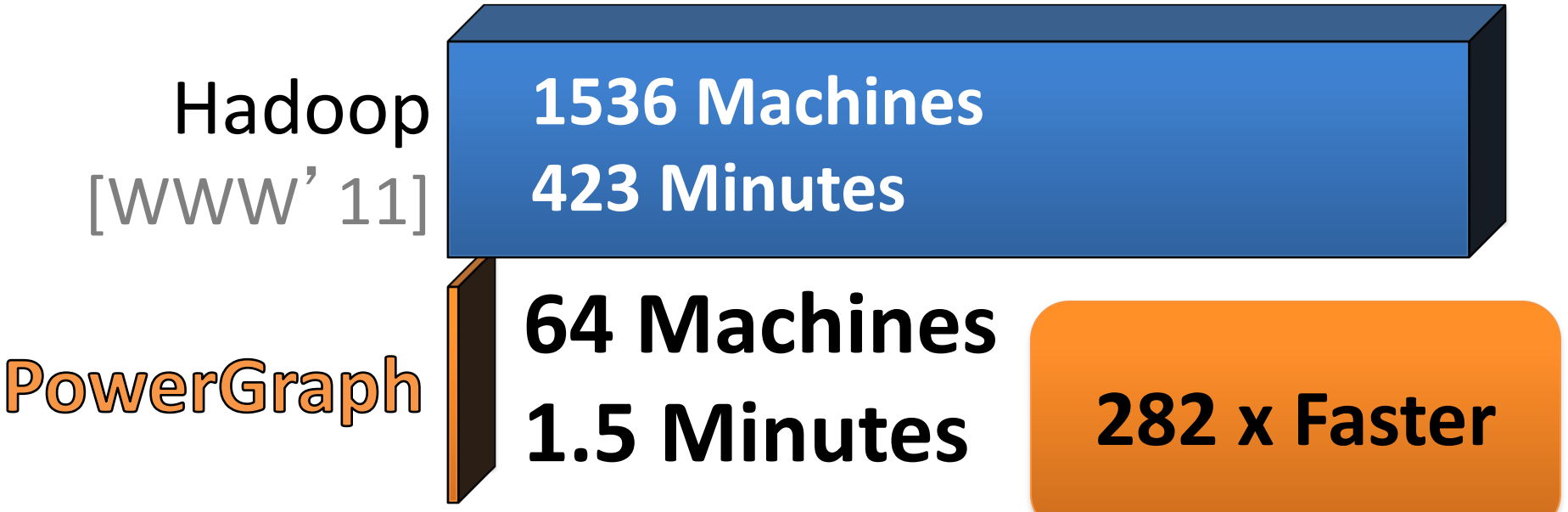
- English language Wikipedia
  - 2.6M Documents, 8.3M Words, 500M Tokens
  - Computationally intensive algorithm



Triangle Counting on The Twitter Graph

Identify individuals with **strong communities.**

Counted: 34.8 Billion Triangles



Why? **Wrong Abstraction** →

Broadcast O(degree^2) messages per Vertex

S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," WWW' 11

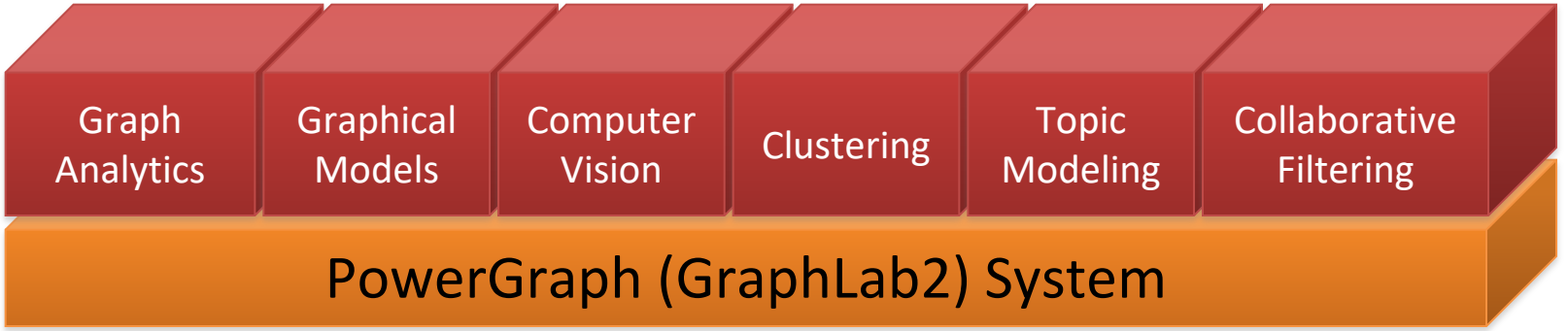
Summary

- *Problem*: Computation on **Natural Graphs** is challenging
  - High-degree vertices
  - Low-quality edge-cuts

- *Solution*: **PowerGraph System**
  - **GAS Decomposition**: split vertex programs
  - **Vertex-partitioning**: distribute natural graphs

- PowerGraph **theoretically** and **experimentally** outperforms existing graph-parallel systems.

Machine Learning and **Data-Mining** Toolkits

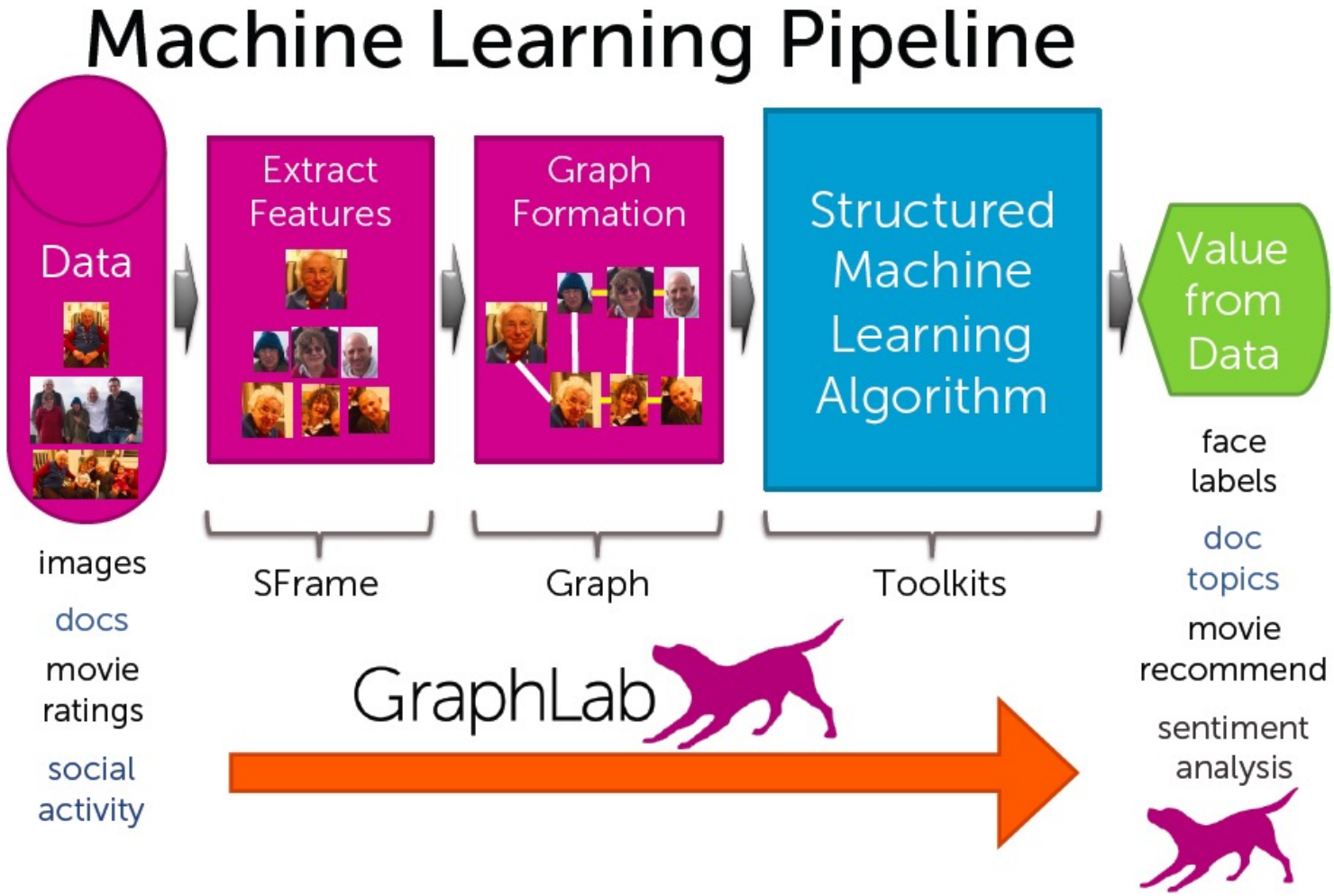


PowerGraph

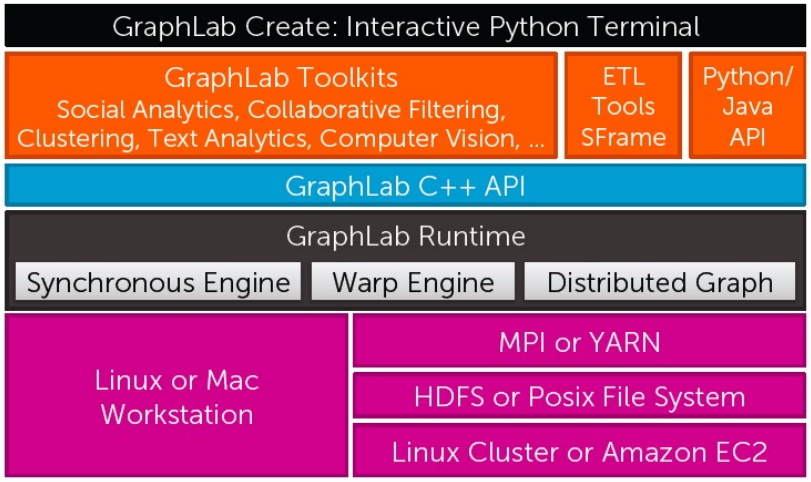
is GraphLab Version 2.1

Apache 2 License

GraphLab for Big Learning (MLDM) Applications



Summary: Different Versions of GraphLab



- GraphLab 1.0 (phased out):
  - Designed to run on closely-coupled, shared-memory multicore machine, performed poorly with PowerLaw Graphs.
- GraphChi: Doing BigData with Small Machine:
  - enables a Single PC to process graphs with billions of edges
- GraphLab (Ver2.x) or so-called the PowerGraph
  - Model targets for seriously-imbalanced node degrees found in practical (Natural) graphs and support parallel processing on Share-Nothing Cluster architecture
  - Taking the split-vertex instead split-edge approach
- GraphCreate (Product of a Startup, Turi.com, founded by GraphLab team)
  - allows you to code in your PC using Python but deploy to run over Cloud-based shared-nothing clusters ; Turi was acquired by Apple in 2016.

IEMS5730 & IERG4330



Big Graph Analytics

Prof. Wing C. Lau

Department of Information Engineering

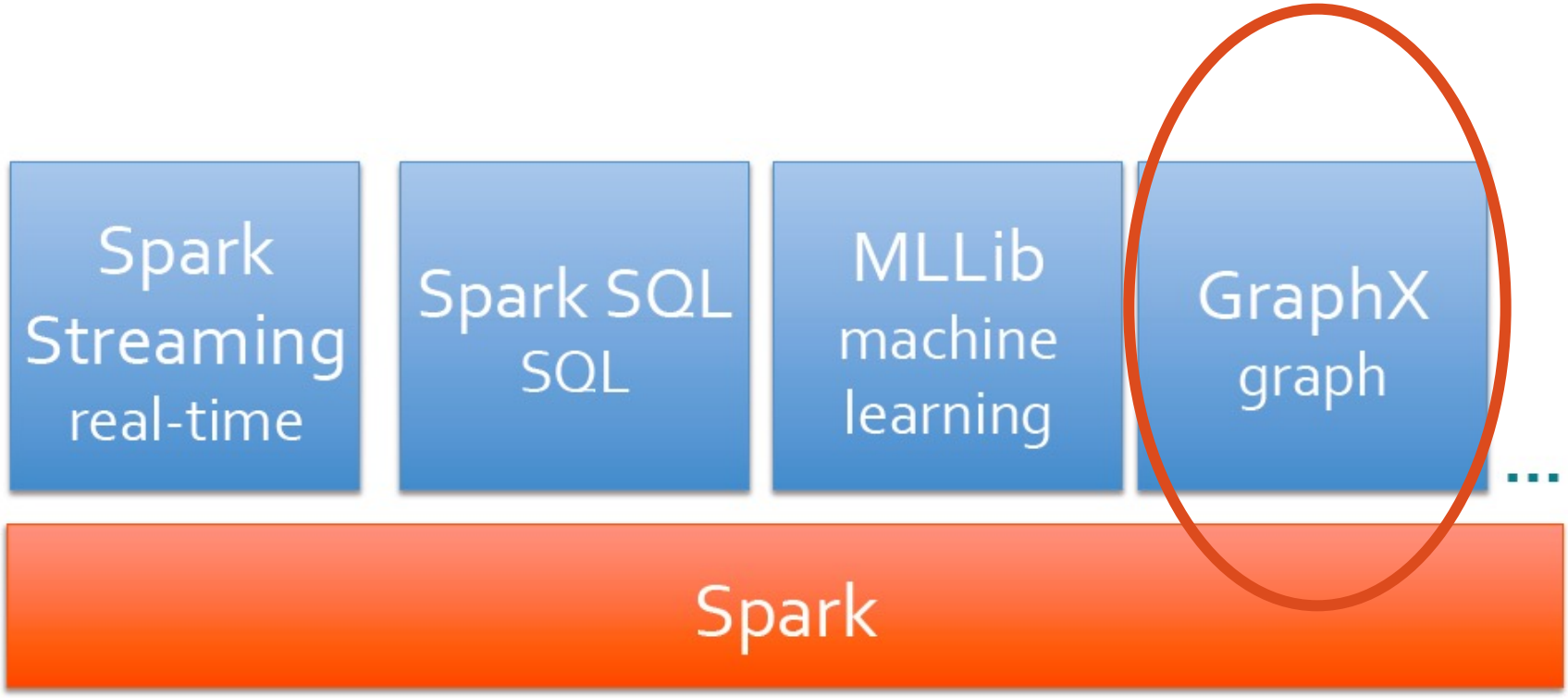
wclau@ie.cuhk.edu.hk

Acknowledgements

- These slides are adapted from the following sources:
  - Matei Zaharia, “Spark 2.0,” Spark Summit East Keynote, Feb 2016.
  - Reynold Xin, “The Future of Real-Time in Spark,” Spark Summit East Keynote, Feb 2016.
  - Michael Armburst, “Structuring Spark: SQL, DataFrames, DataSets, and Streaming,” Spark Summit East Keynote, Feb 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Spark SQL,” Spark Summit East, Feb 2016.
  - Michael Armburst, “Spark DataFrames: Simple and Fast Analytics on Structured Data,” Spark Summit Amsterdam, Oct 2015.
  - Michael Armburst et al, “Spark SQL: Relational Data Processing in Spark,” SIGMOD 2015.
  - Michael Armburst, “Spark SQL Deep Dive,” Melbourne Spark Meetup, June 2015.
  - Reynold Xin, “Spark,” Stanford CS347 Guest Lecture, May 2015.
  - Joseph K. Bradley, “Apache Spark MLlib’s past trajectory and new directions,” Spark Summit Jun 2017.
  - Joseph K. Bradley, “Distributed ML in Apache Spark,” NYC Spark MeetUp, June 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Apache Spark SQL,” Spark Summit, June 2016.
  - Joseph K. Bradley, “GraphFrames: DataFrame-based graphs for Apache Spark,” NYC Spark MeetUp, April 2016.
  - Joseph K. Bradley, “Practical Machine Learning Pipelines with MLlib,” Spark Summit East, March 2015.
  - Joseph K. Bradley, “Spark DataFrames and ML Pipelines,” MLconf Seattle, May 2015.
  - Ameet Talwalkar, “MLlib: Spark’s Machine Learning Library,” AMPCamps 5, Nov. 2014.
  - Shivaram Venkataraman, Zongheng Yang, “SparkR: Enabling Interactive Data Science at Scale,” AMPCamps 5, Nov. 2014.
  - Tathagata Das, “Spark Streaming: Large-scale near-real-time stream processing,” O’Reilly Strata Conference, 2013.
  - Joseph Gonzalez et al, “GraphX: Graph Analytics on Spark,” AMPCAMP 3, 2013.
  - Jules Damji, “Jumpstart on Apache Spark 2.X with Databricks,” Spark Sat. Meetup Workshop, Jul 2017.
  - Sameer Agarwal, “What’s new in Apache Spark 2.3,” Spark+AI Summit, June 2018.
  - Reynold Xin, Spark+AI Summit Europe, 2018.
  - Hyukjin Kwon of Hortonworks, “What’s New in Spark 2.3 and Spark 2.4,” Oct 2018.
  - Matel Zaharia, “MLflow: Accelerating the End-to-End ML Lifecycle,” Nov. 2018.
  - Jules Damji, “MLflow: Platform for Complete Machine Learning Lifecycle,” PyData, Jan 2019.

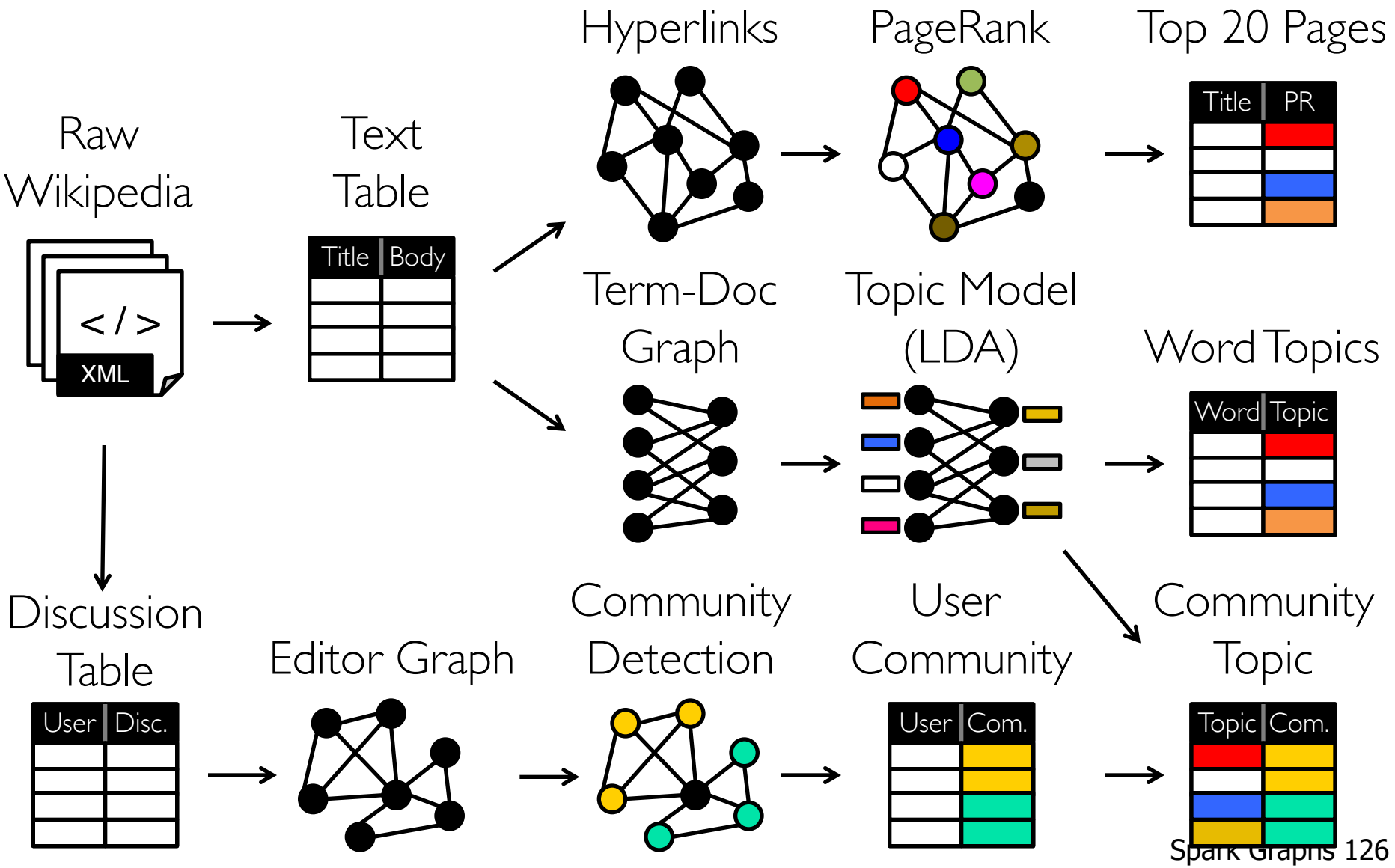
- All copyrights belong to the original authors of the materials.

Major Modules in Spark



GraphX:
Unifying Data-Parallel and Graph-Parallel Analytics

Graphs are Central to Analytics

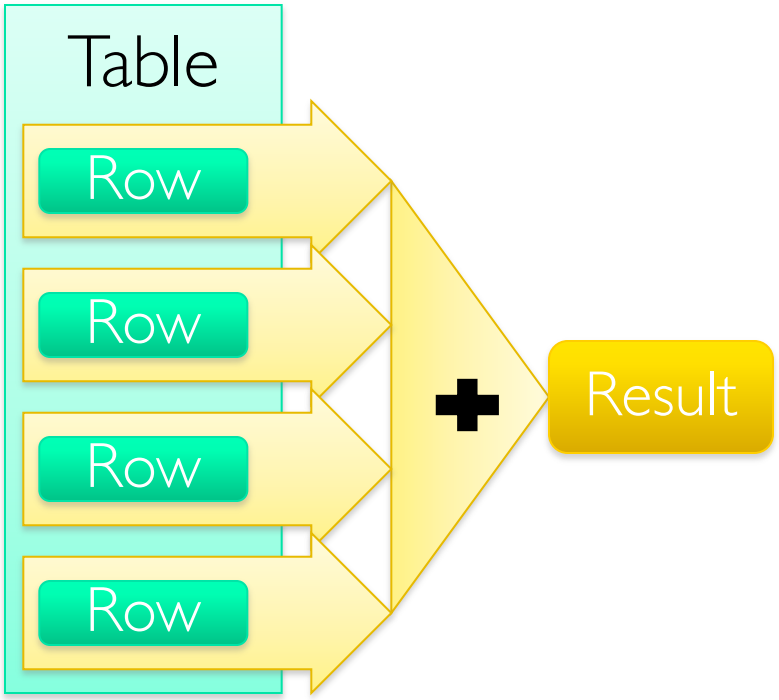


Separate Systems to Support Each View

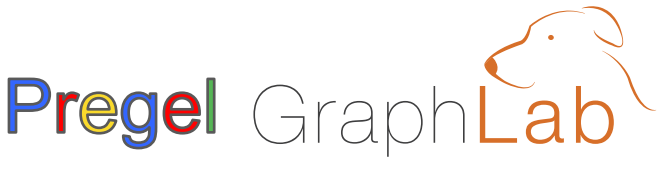
Table View

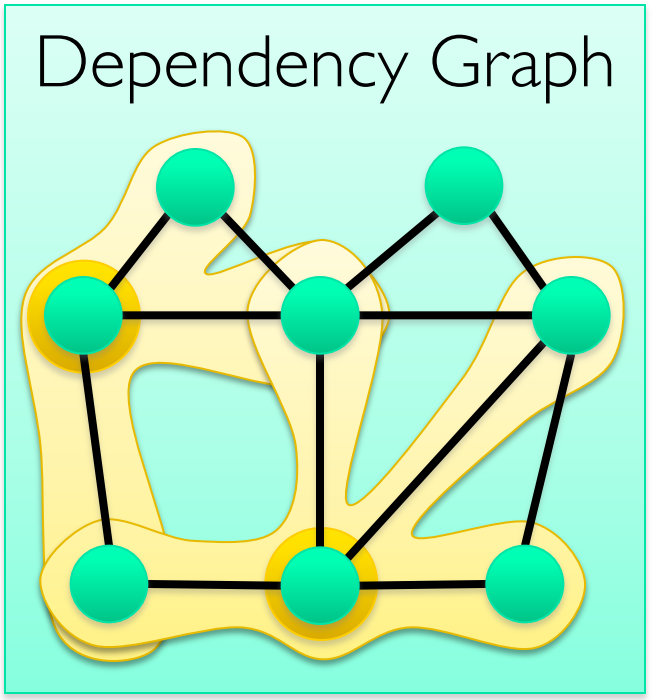






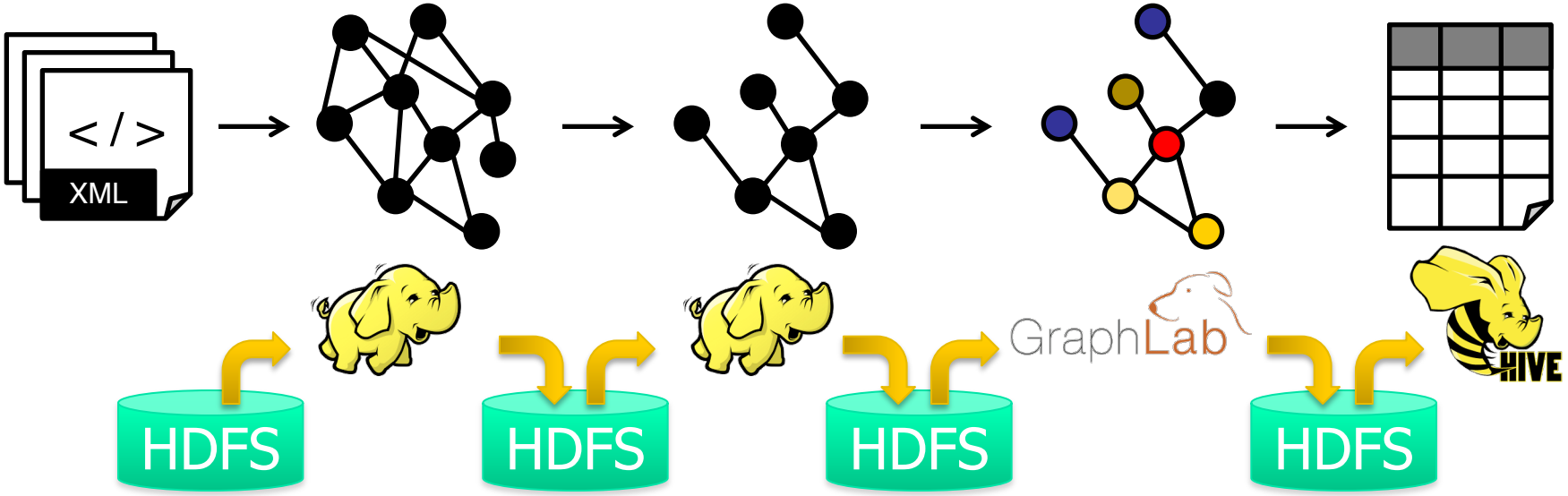
Graph View





Inefficient

Expensive data movement and duplication across the network and file system



Limited reuse internal data-structures across stages

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

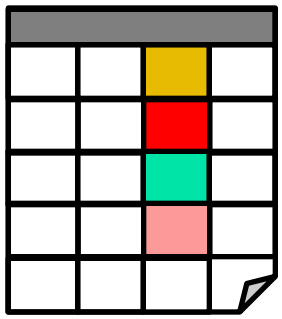
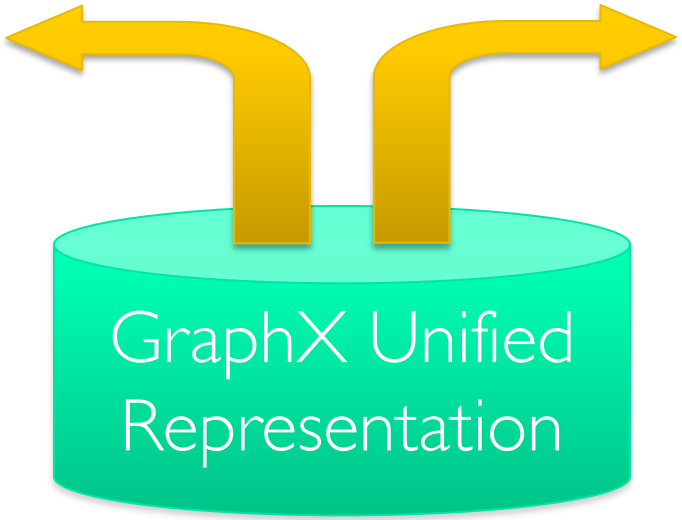
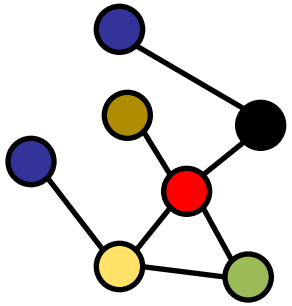


Table View

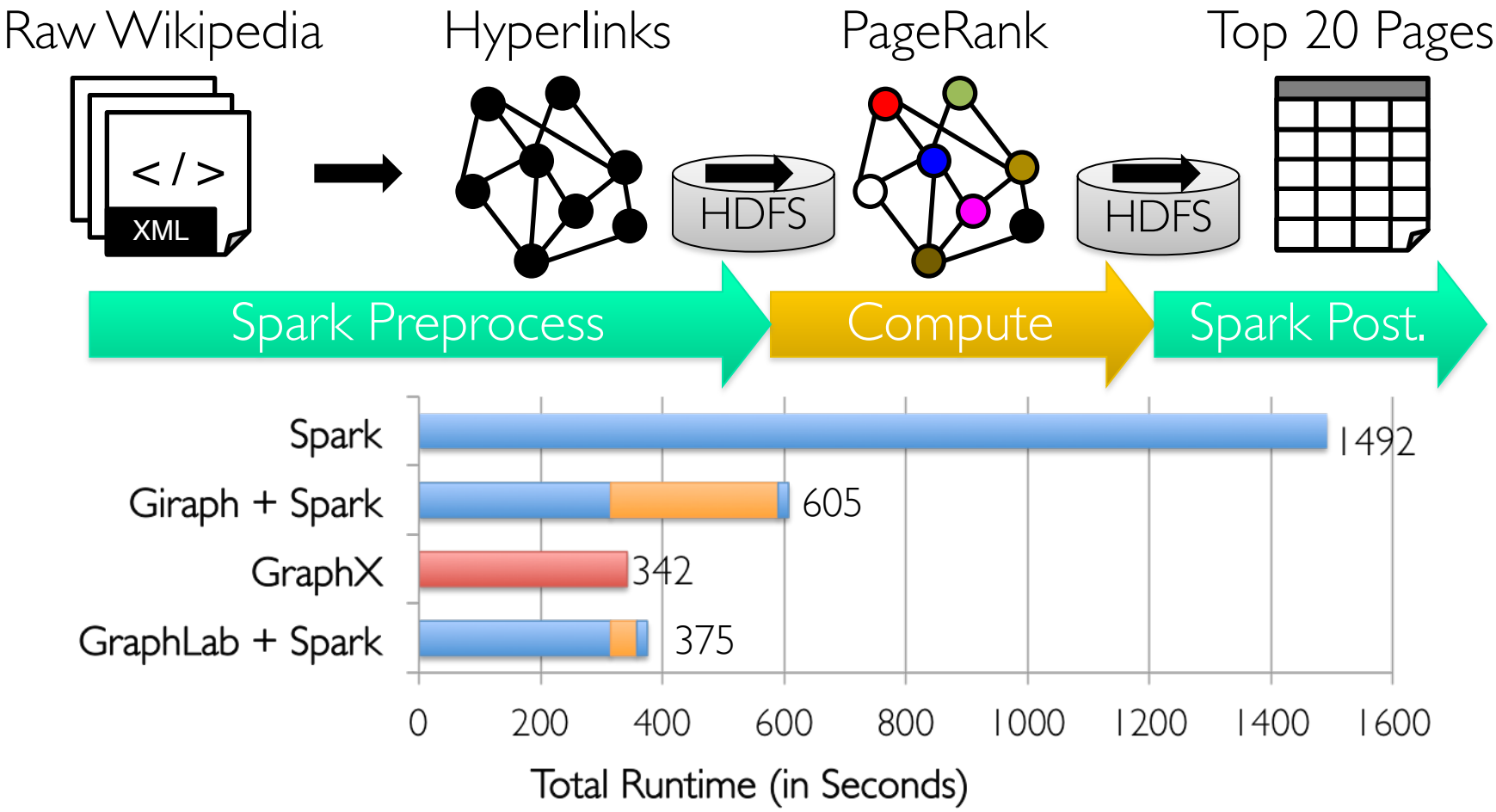




Graph View

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

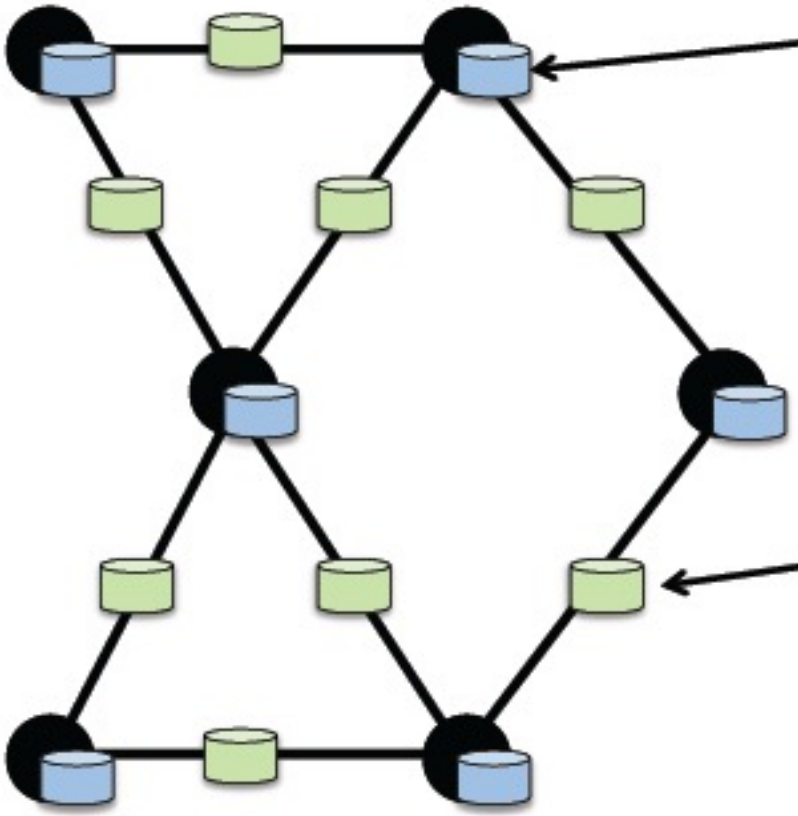
A Small Pipeline in GraphX



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

The GraphX API

Property Graphs



Vertex Property:

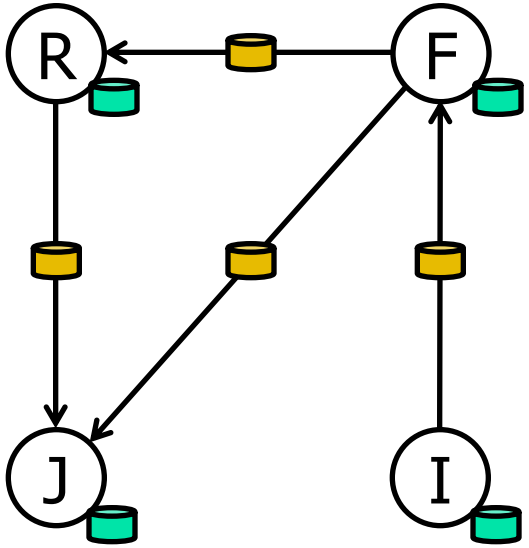
- User Profile
- Current PageRank Value

Edge Property:

- Weights
- Relationships
- Timestamps

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

Distributed Graphs as Tables (RDDs)

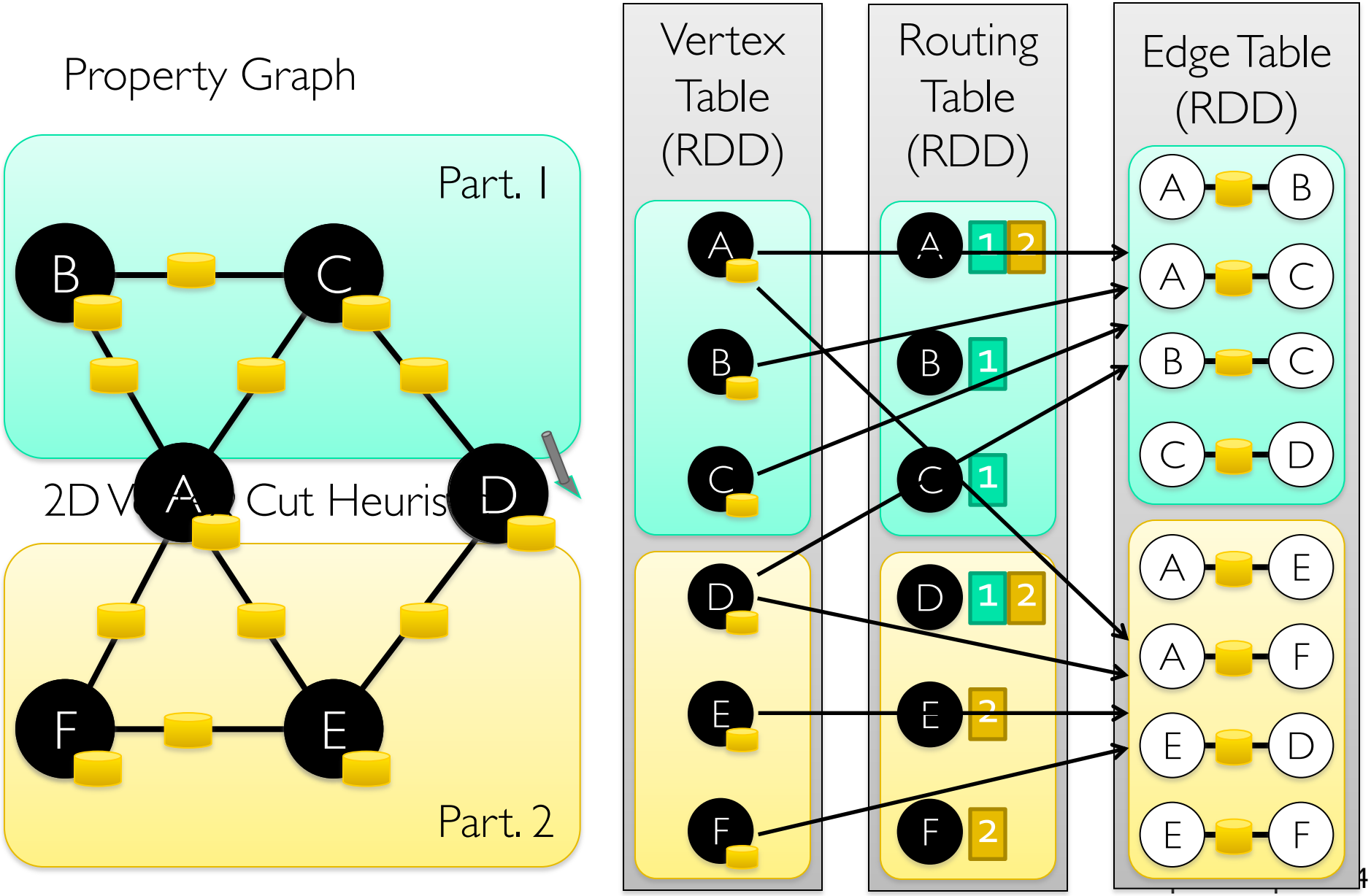


Table Operators

- 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                 ...
```

Creating a Graph (Scala)

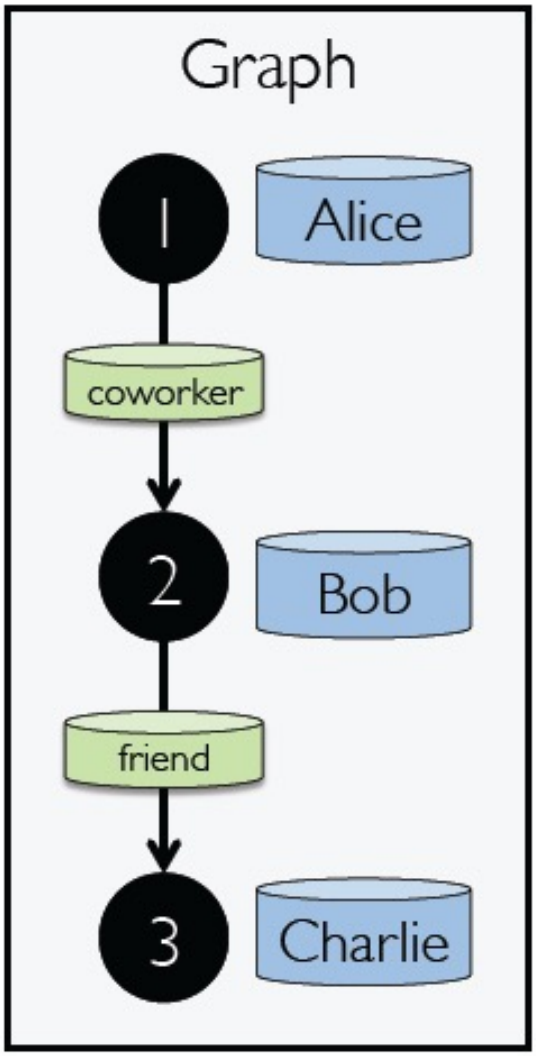
```
type VertexId = Long

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)
```



Graph Operations (Scala)

```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  ============================================
```

Graph Operations (Scala)

```
  // 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[E
D2])
    : 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]
```

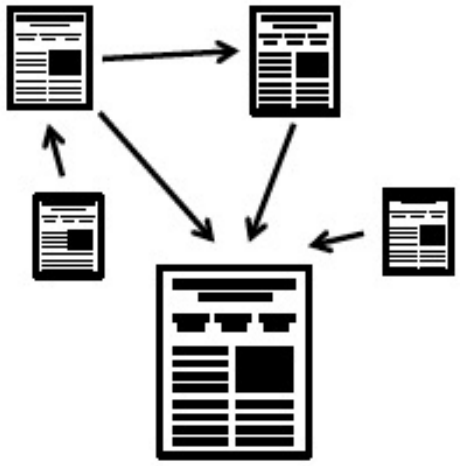
```
  // Join RDDs with the graph ================================================================
  def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph[VD, ED]

  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]
}
```

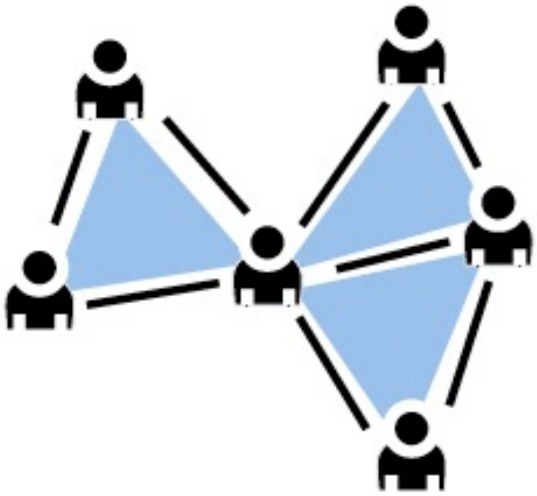
Built-in Algorithms (Scala)

```
    // 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
}
```

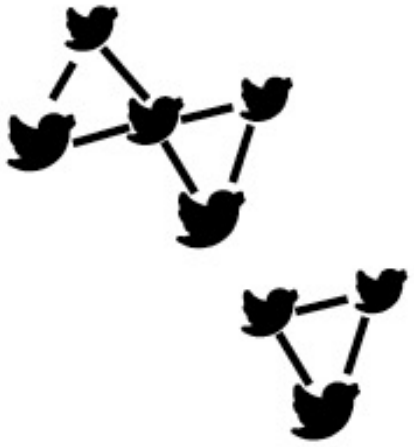
PageRank



Triangle Count



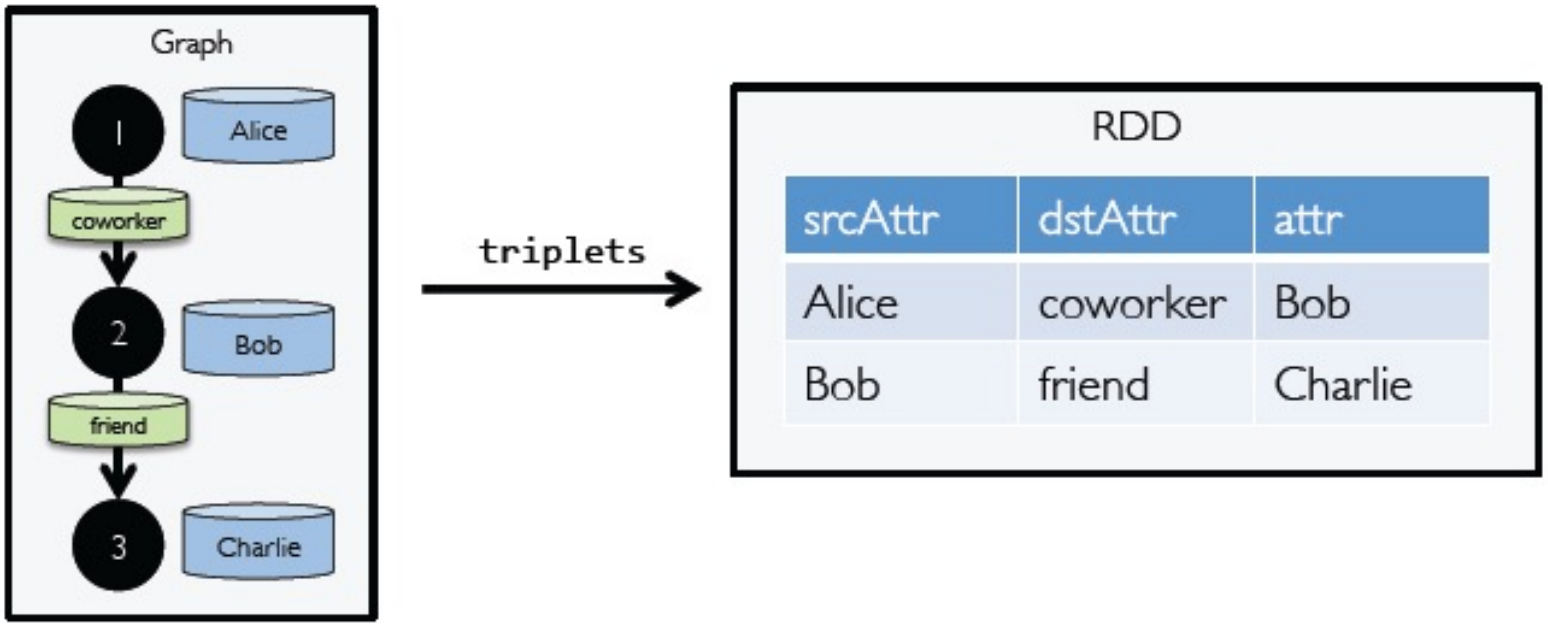
Connected Components



The "triplets" view

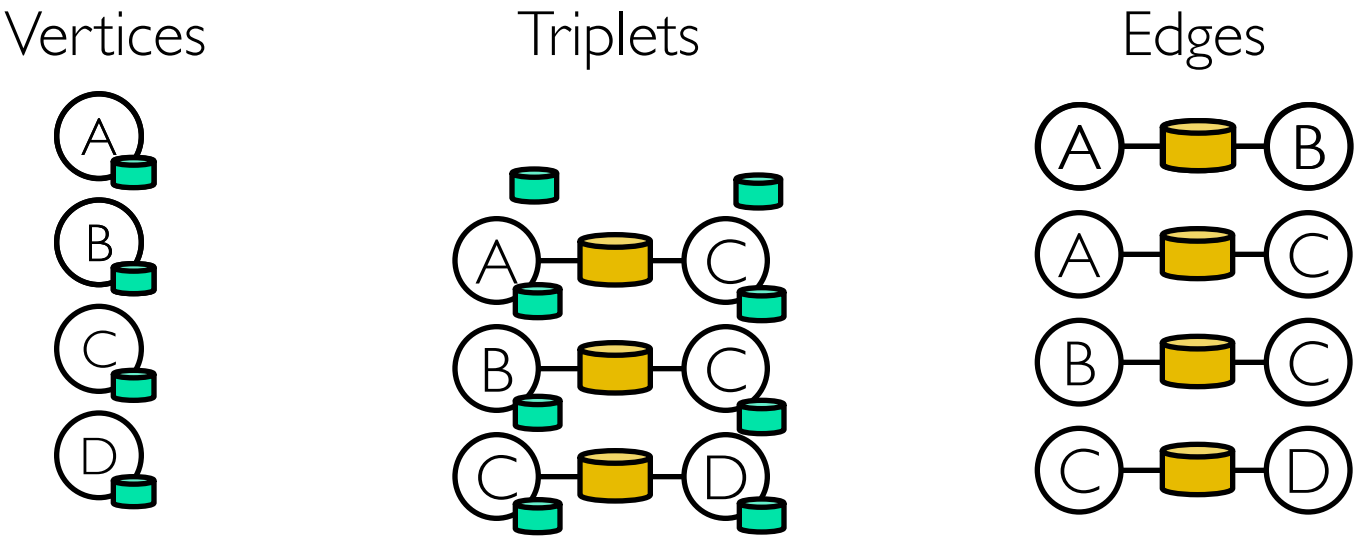
```
class Graph[VD, ED] {
    def triplets: RDD[EdgeTriplet[VD, ED]]
}

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



Triplets Join Vertices and Edges

- The *triplets* operator joins vertices and edges:

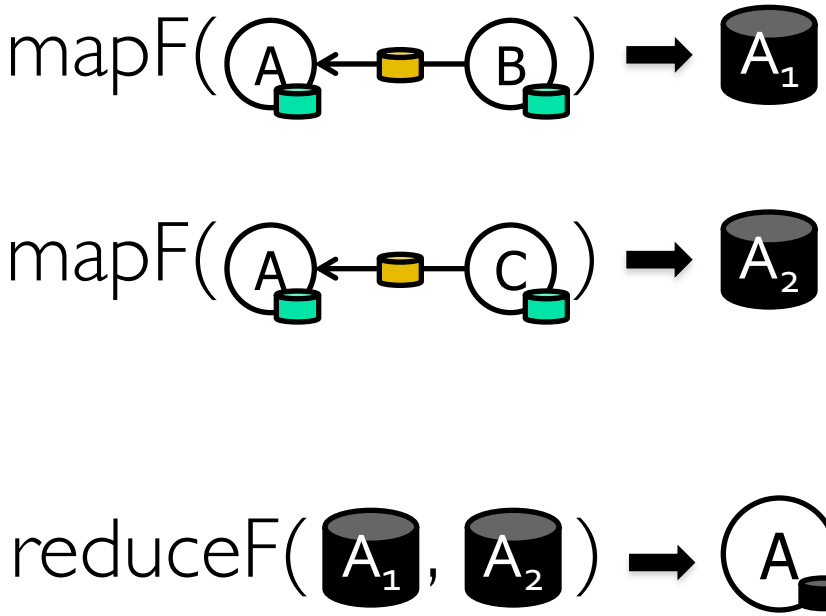


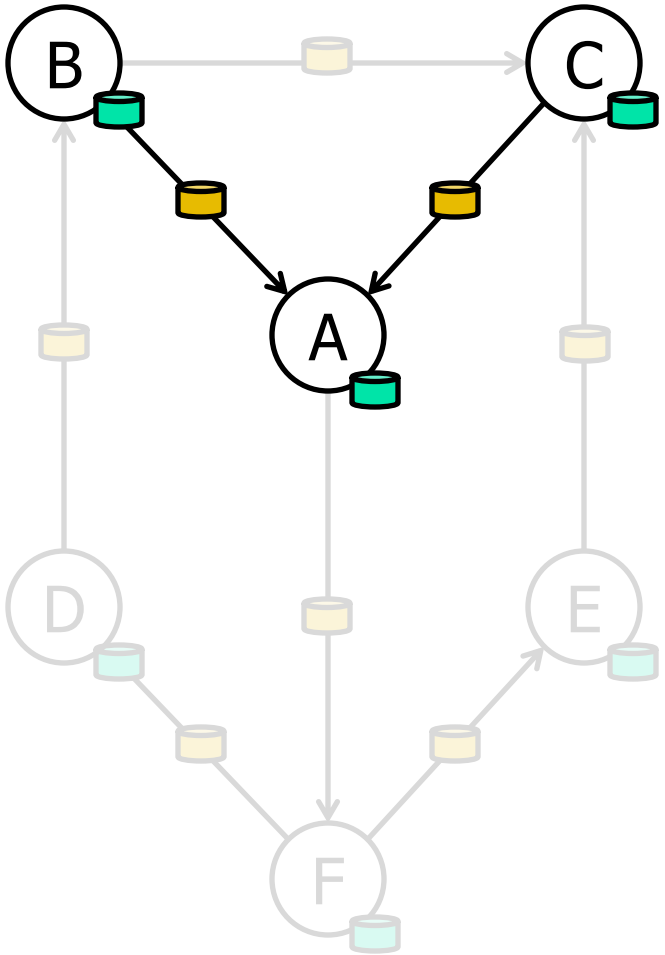
The *mapreduceTriplets* operator sums adjacent triplets.

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

Map Reduce Triplets

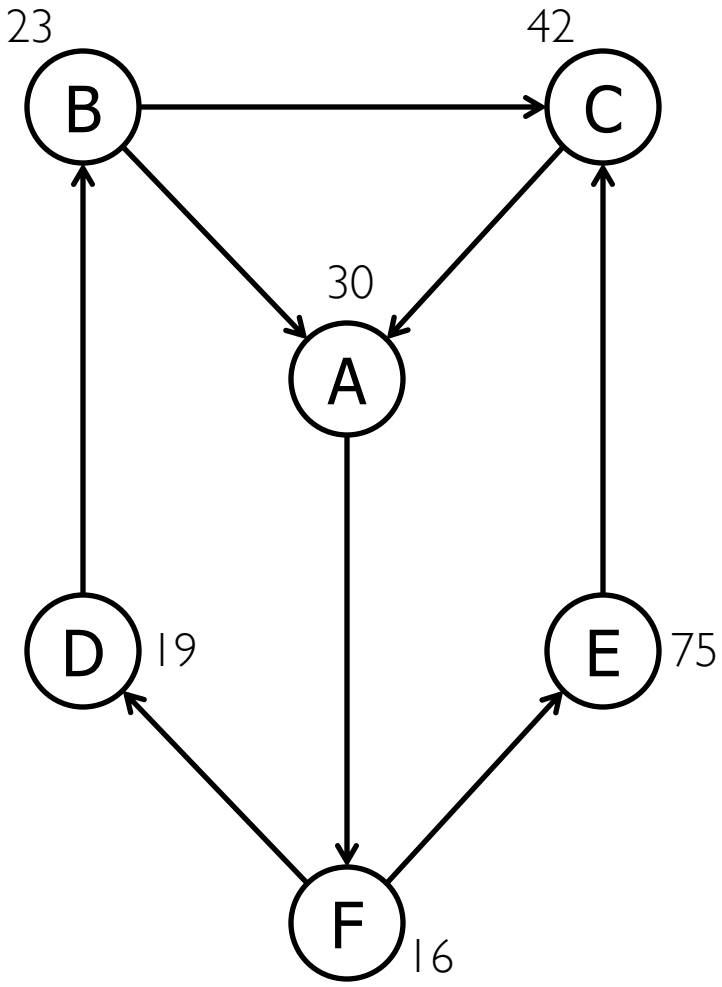
- *Map-Reduce* for each vertex





Example: Oldest Follower

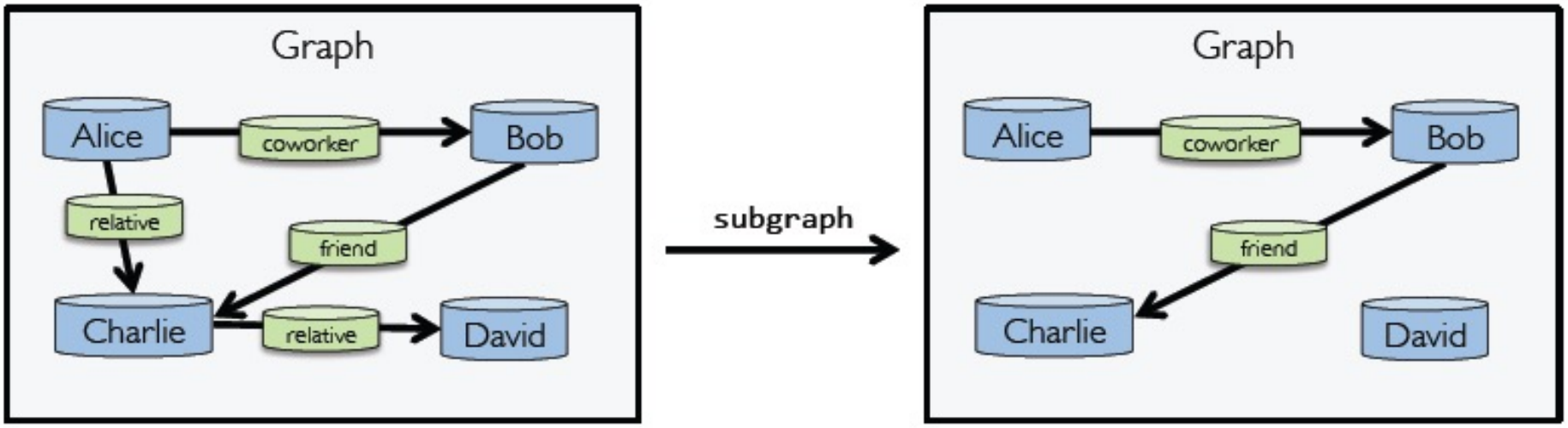
- *What is the age of the oldest follower for each user?*
-
```
val oldestFollowerAge = graph
  .mapreduceTriplets(
    e=> (e.dst.id,
e.src.age),//Map
    (a,b)=> max(a, b) //Reduce
  )
  .vertices
```



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")
```



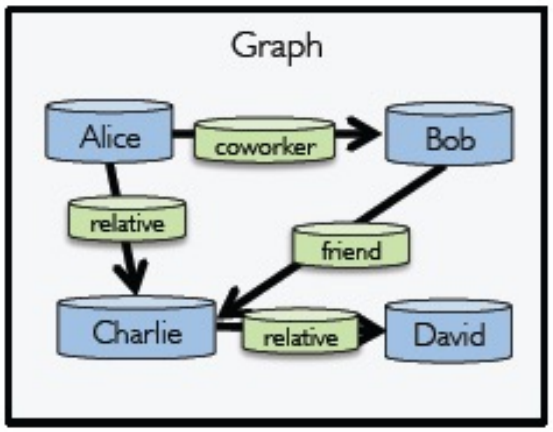
Computation w/ `mapReduceTriplets`

```
class Graph[VD, ED] {
  def mapReduceTriplets[A](
    sendMsg: (EdgeTriplet[VD, ED]) => Iterator[(VertexId, A)],
    mergeMsg: (A, A) => A): RDD[(VertexId, A)]
}
```



upgrade to **aggregateMessages** in Spark 1.2.0

```
graph.mapReduceTriplets(
  edge => Iterator(
    (edge.srcId, 1),
    (edge.dstId, 1)),
  _ + _)
```



`mapReduceTriplets`

RDD

vertex id	degree
Alice	2
Bob	2
Charlie	3
David	1

Computation w/ aggregateMessages

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

The "aggregateMessages" operator:

(1) Apply a user-defined sendMsg function to each *edge triplet* in the graph and then
(2) Use the another user-defined mergeMsg function to aggregate those messages at their destination vertex.

Example: Compute Average Age of Older Followers of each node using `aggregateMessages`

```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(_))
```

Refer to "examples/src/main/scala/org/apache/spark/examples/graphx/AggregateMessagesExample.scala" in Spark repo for the full source code of this example

Have Expressed 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.

Re-implementation of the Pregel abstraction using the GraphX API

```
def Pregel(g: Graph[V, E],
      vprog: (Id, V, M) => V,
      sendMsg: (Triplet) => M,
      gather: (M, M) => M): Collection[V] = {
 // Set all vertices as active
 g = g.mapV((id, v) => (v, halt=false))
 // Loop until convergence
 while (g.vertices.exists(v => !v.halt)) {
   // Compute the messages
   val msgs: Collection[(Id, M)] =
     // Restrict to edges with active source
     g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
     // Compute messages
      .mrTriplets(sendMsg, gather)
   // Receive messages and run vertex program
   g = g.leftJoinV(msgs).mapV(vprog)
 }
 return g.vertices
}
```

Finding Connected Components using the GraphX variant of Pregel

```
def ConnectedComp(g: Graph[V, E]) = {
  g = g.mapV(v => v.id) // Initialize vertices
  def vProg(v: Id, m: Id): Id = {
    if (v == m) voteToHalt(v)
    return min(v, m)
  }
  def sendMsg(t: Triplet): Id =
    if (t.src.cc < t.dst.cc) t.src.cc
    else None // No message required
  def gatherMsg(a: Id, b: Id): Id = min(a, b)
  return Pregel(g, vProg, sendMsg, gatherMsg)
}
```

Listing 6: **Connected Components:** For each vertex we compute the lowest reachable vertex id using Pregel.

GraphX System Design

Graph Partitioning Strategies

Edge Cut in GraphLab 1.0 vs. Vertex Cut in GraphLab 2.0 in PowerGraph and GraphX

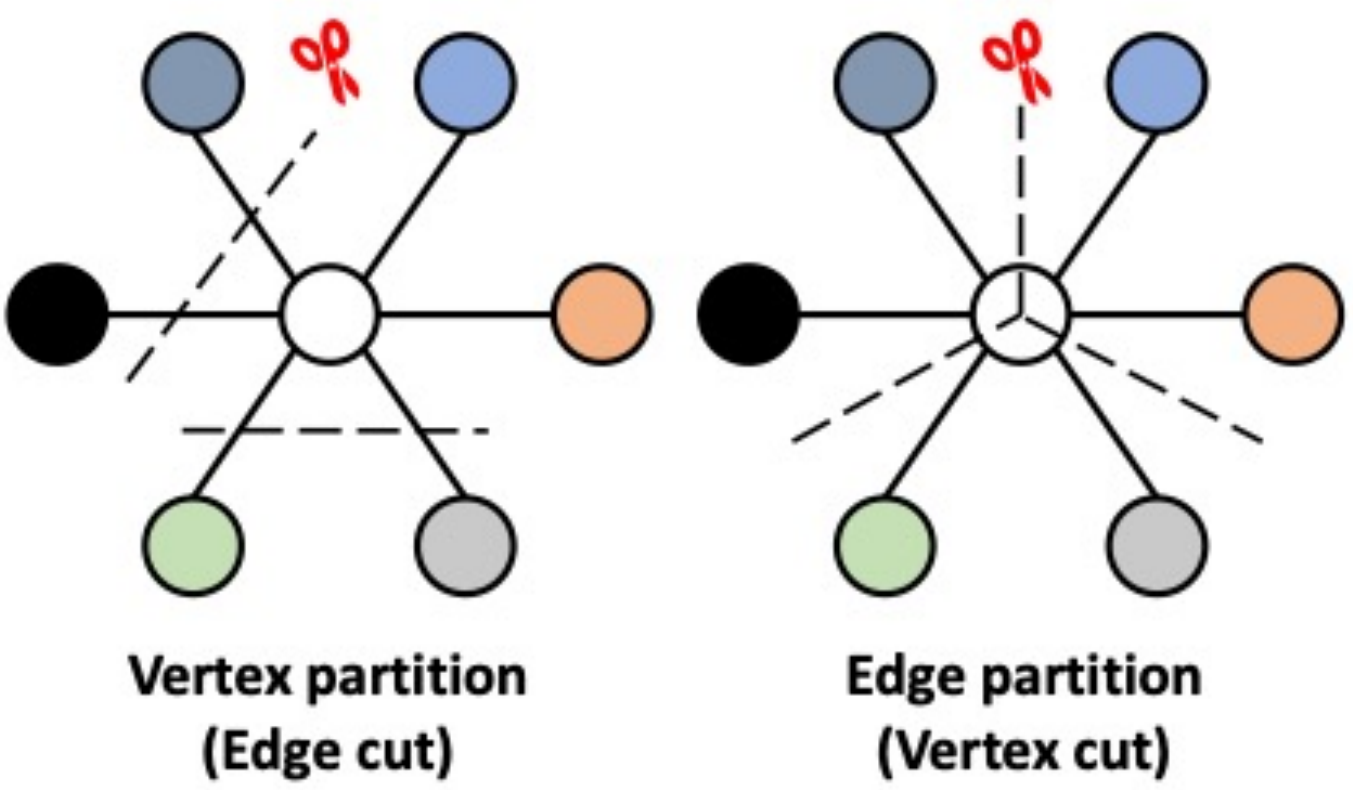
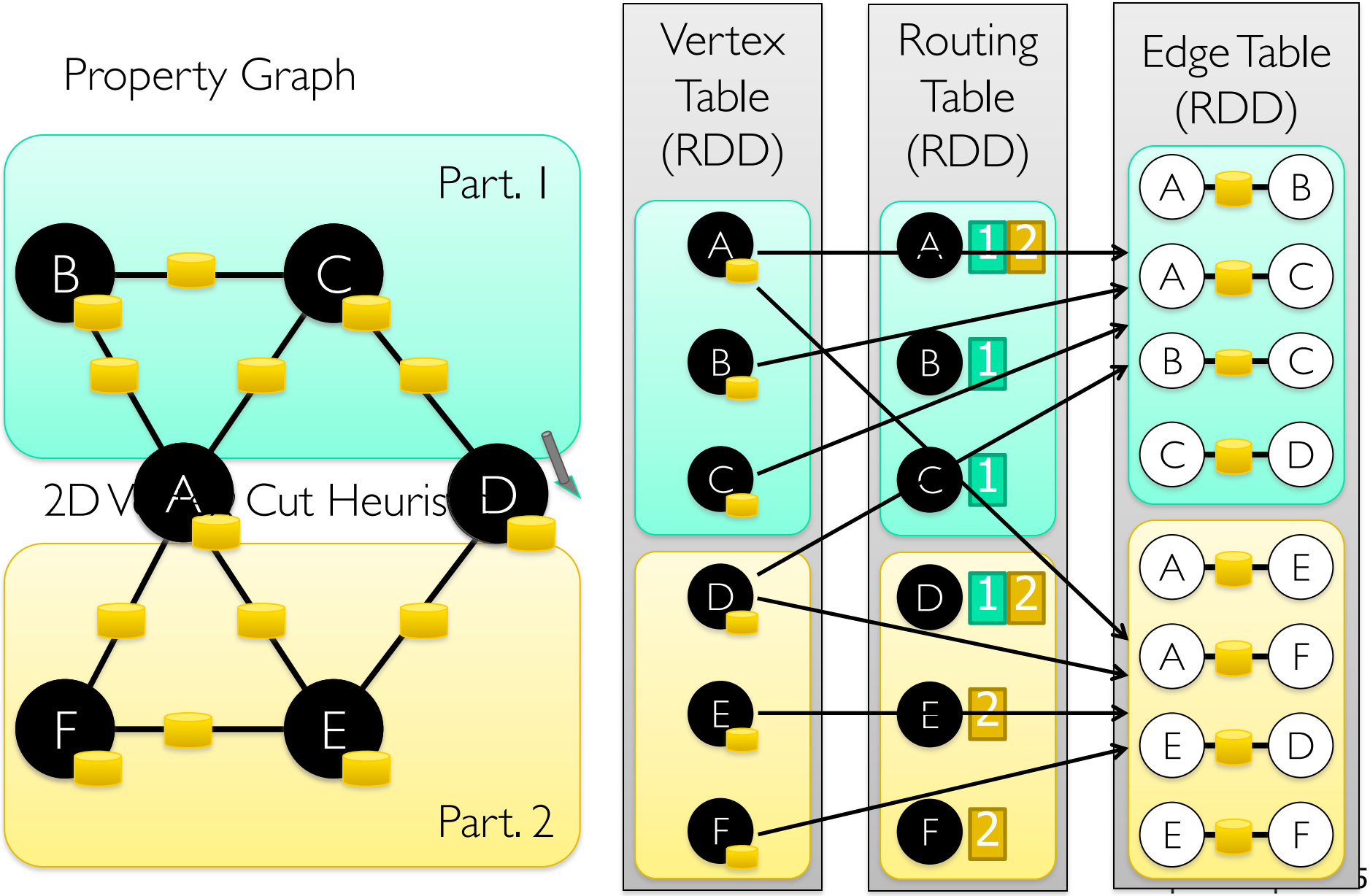


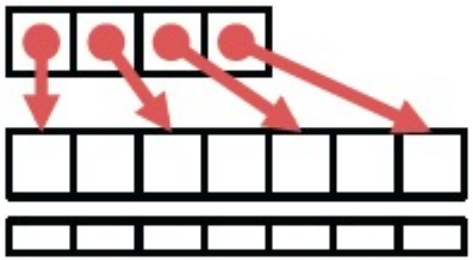
Fig. 2. Partitioning strategies.

Distributed Graphs as Tables (RDDs)

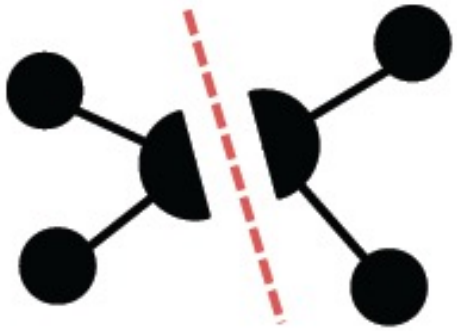


Graph System Optimizations

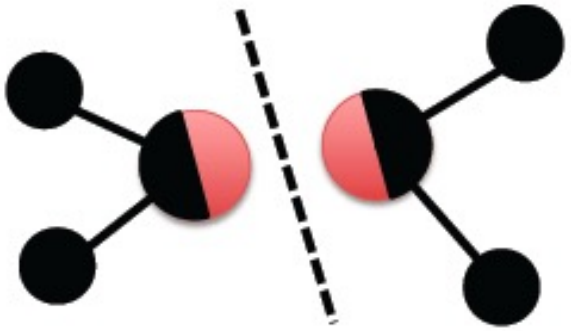
Specialized Data-Structures



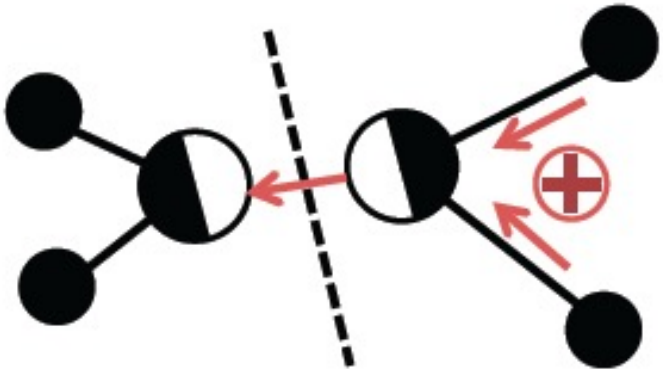
Vertex-Cuts Partitioning



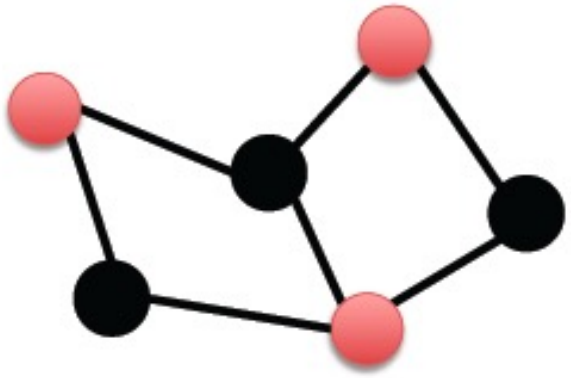
Remote Caching / Mirroring



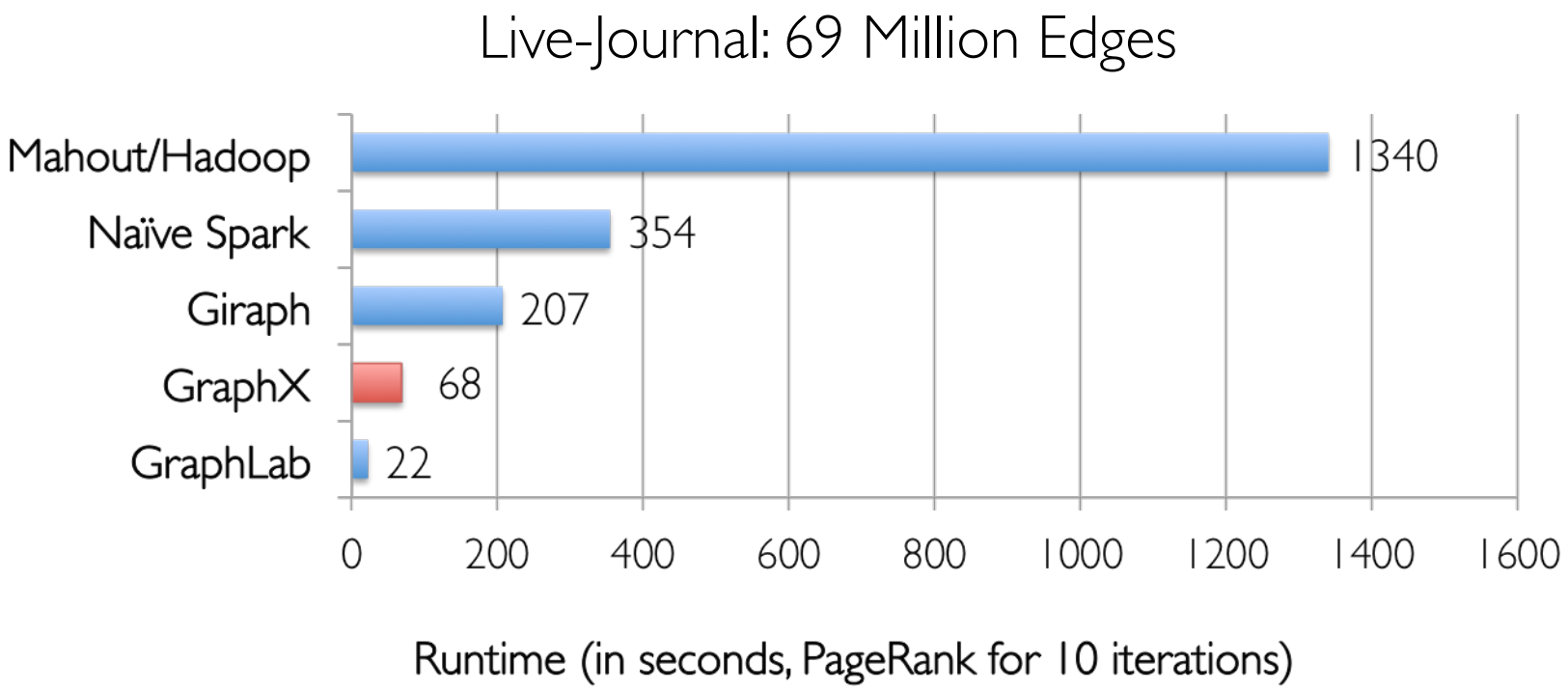
Message Combiners



Active Set Tracking

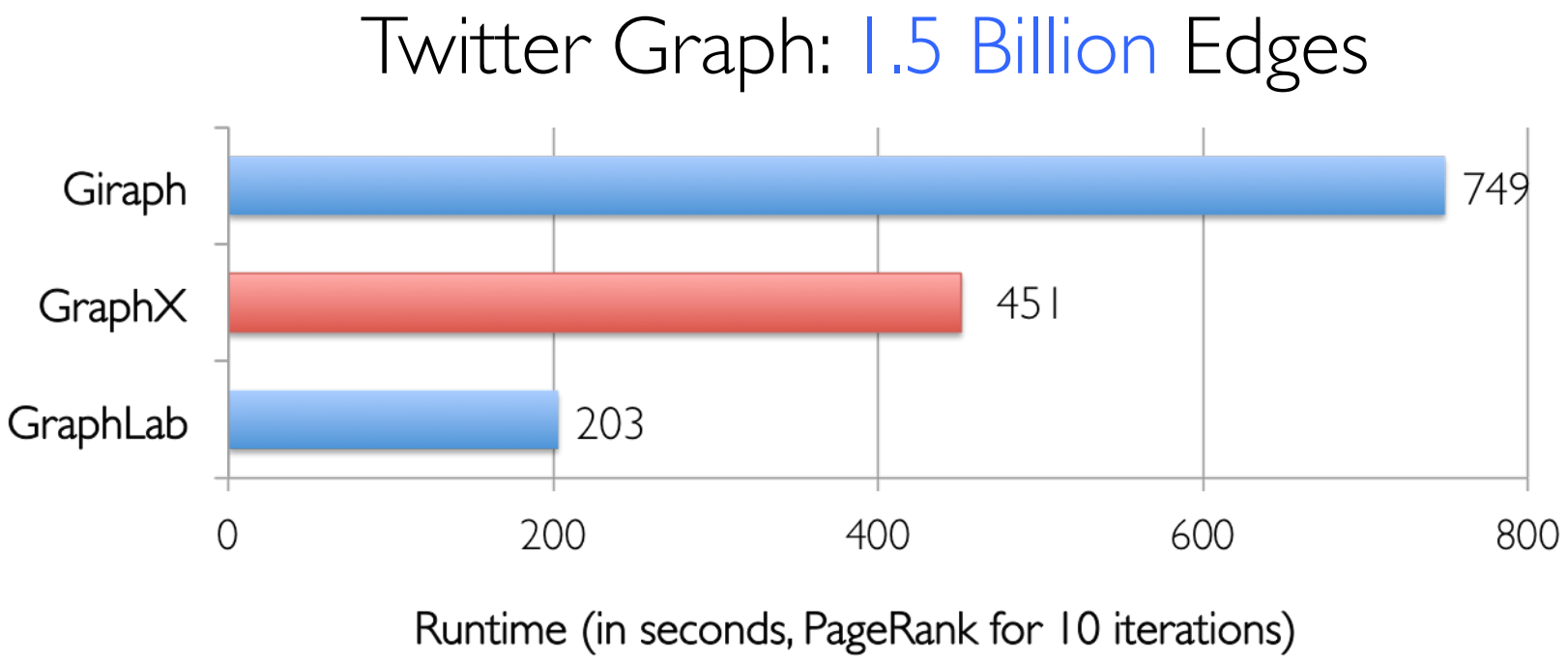


Performance Comparisons



GraphX is roughly 3x slower than GraphLab

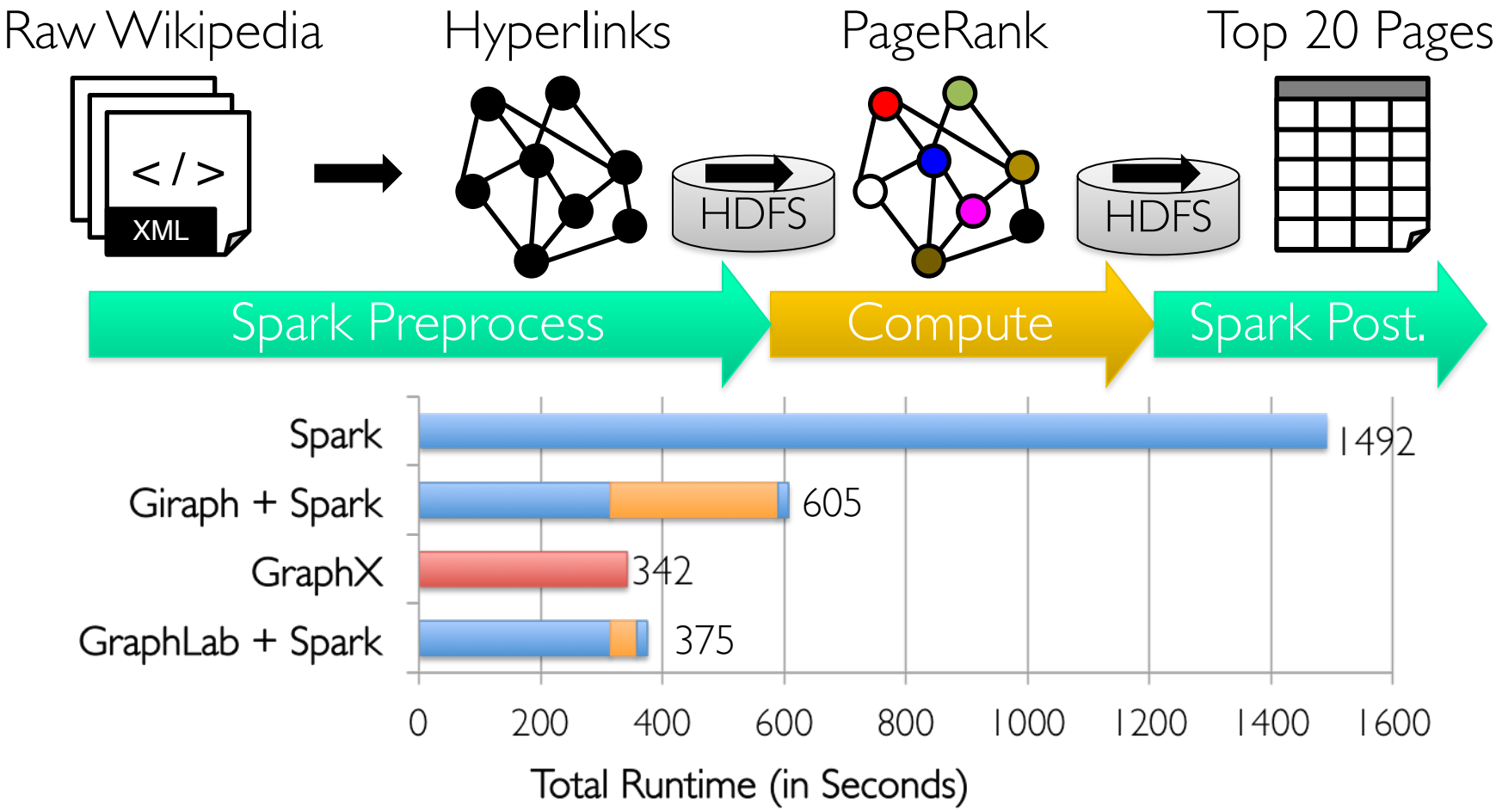
GraphX scales to larger graphs



GraphX is roughly 2x slower than GraphLab

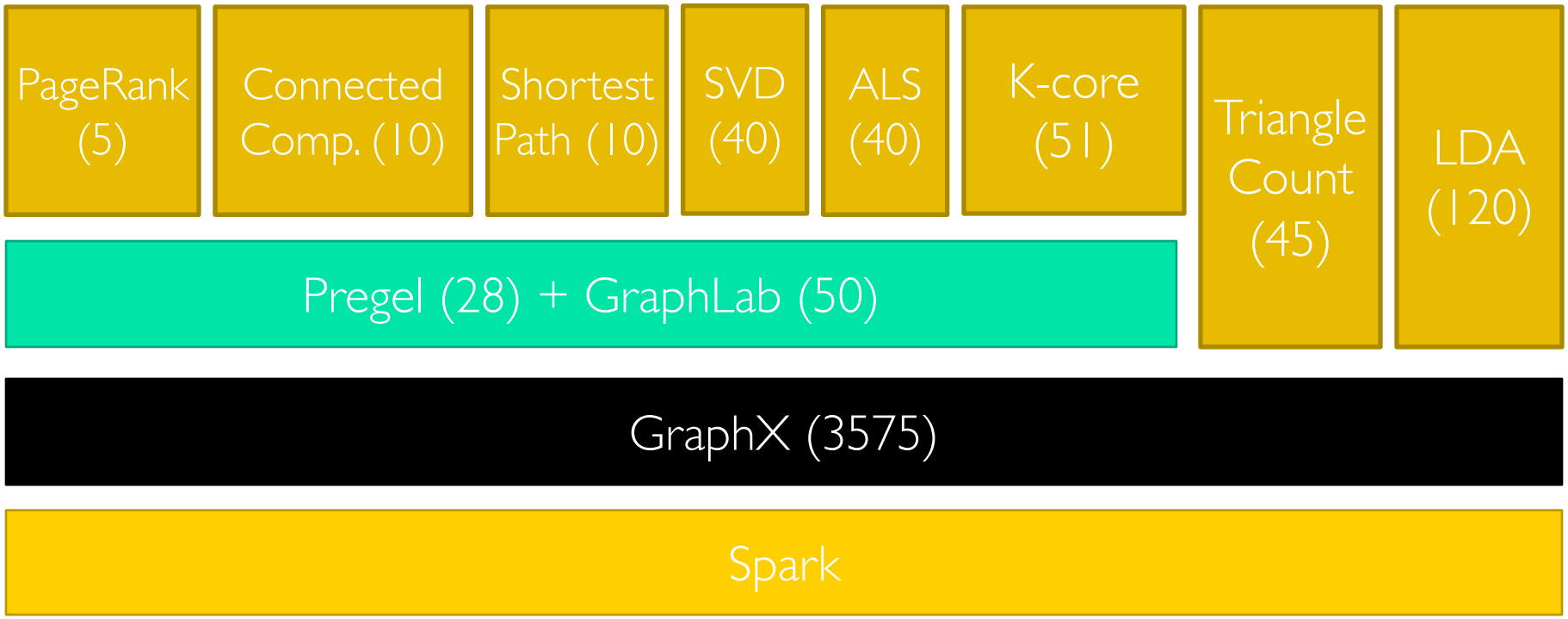
- Scala + Java overhead: Lambdas, GC time, …
- No shared memory parallelism: 2x increase in comm.

A Small Pipeline in GraphX



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

The GraphX Stack (Lines of Code)



GraphX: Summary and Observations

- Domain specific views: *Tables* and *Graphs*
  - tables and graphs are first-class composable objects
  - specialized operators which exploit view semantics
- Single system that efficiently spans the pipeline
  - minimize data movement and duplication
  - eliminates need to learn and manage multiple systems
- Graphs through the lens of database systems
  - Graph-Parallel Pattern → Triplet joins in relational alg.
  - Graph Systems → Distributed join optimizations

Directions for Further Development of GraphX

- Static Data → Dynamic Data, Time-Evolving Big Graphs
  - Apply GraphX unified approach to time evolving data
  - Model and analyze relationships over time

  => e.g. See the GraphTau paper in GRADES 2016.

- Serving Graph Structured Data
  - Allow external systems to interact with GraphX
  - Unify distributed graph databases with relational database technology

  => Refer to the next topic: Graphframes

Summary of Apache Spark's GraphX library

Strength

- General-purpose graph processing library
- Optimized for fast distributed computing
- A rich library of algorithms: PageRank, Connected Components, etc

Limitations

- No Java, Python APIs
- Lower-level RDD-based API (vs. DataFrames)
- Cannot use recent Spark (SQL) optimizations: Catalyst query optimizer, Tungsten memory management.

See http://amplab.github.io/graphx/ for more details.

Enter GraphFrames

https://github.com/graphframes.graphframe

Motivations for GraphFrames

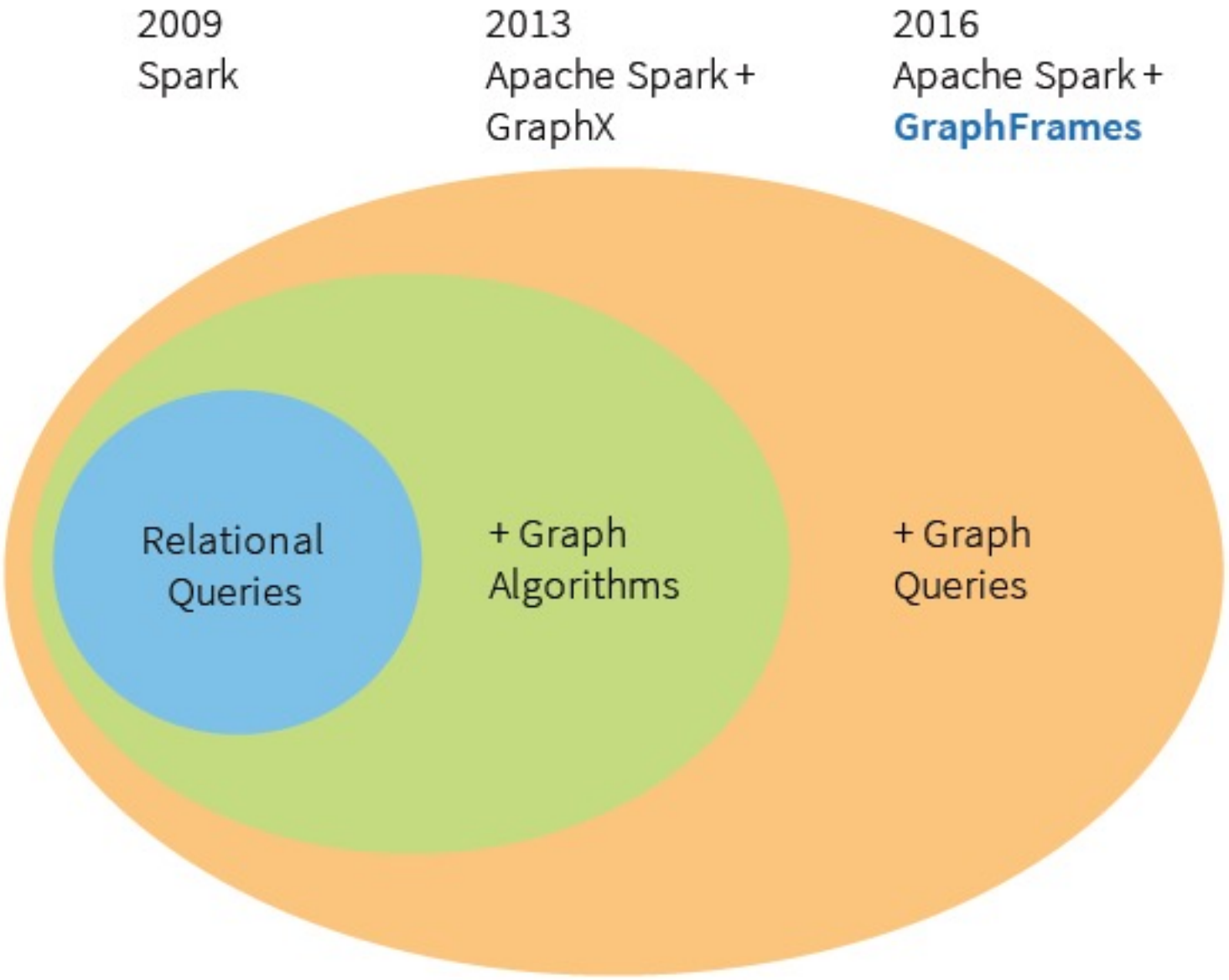
Goal: Support DataFrame-based Graph processing on Spark

- Simplify Interactive Queries
- Support Motif-finding for Structural Pattern Search
- Benefit from DataFrame Optimization

Collaborations between Databricks, UC Berkeley & MIT

- Now with open-source community contributors

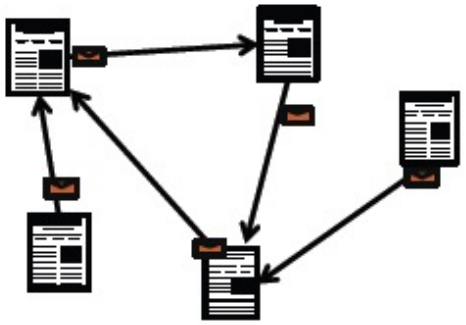
Evolution of Graph Processing Support in Spark



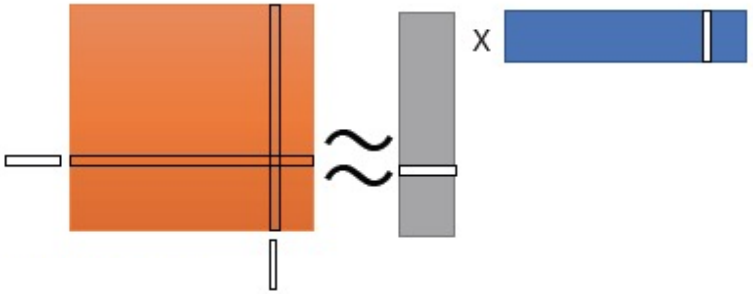
Graph Algorithms vs. Graph Queries

Graph Algorithms

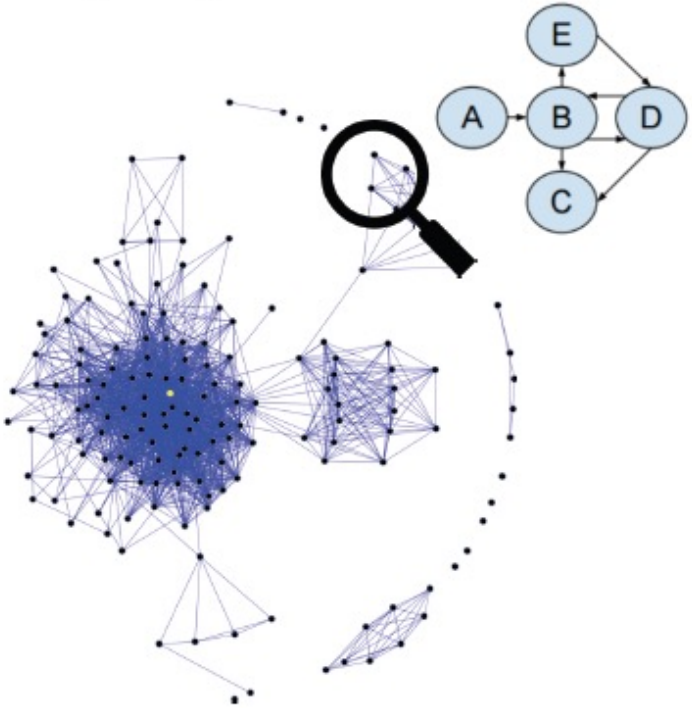
PageRank



Alternating Least Squares

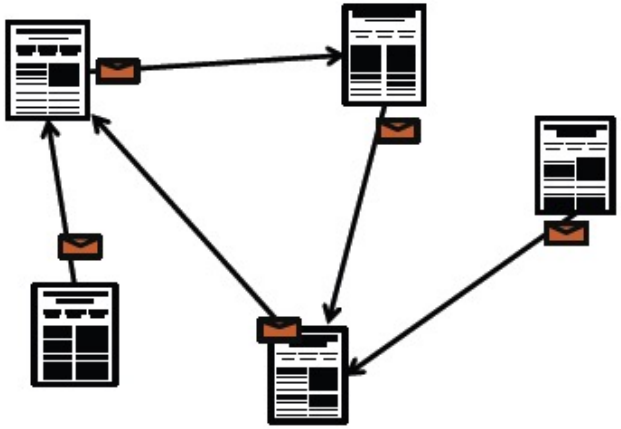


Graph Queries

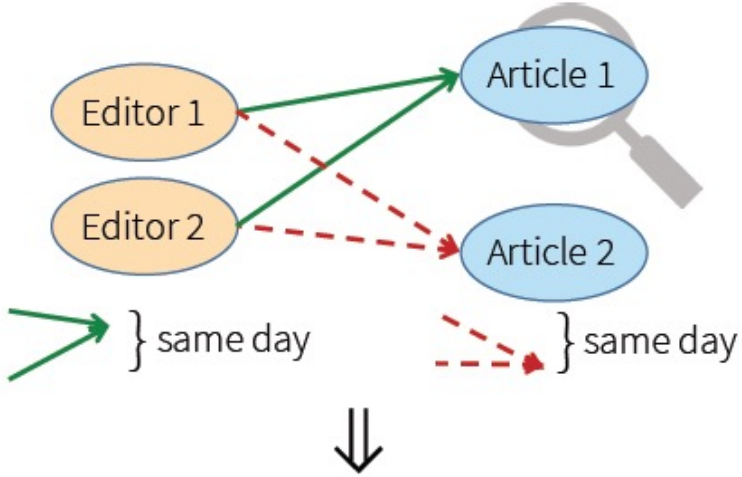


Graph Algorithms vs. Graph Queries, an Example

Graph Algorithm: PageRank



Graph Query: Wikipedia Collaborators



Editor 1	Editor 2	Article 1	Article 2

Graph Algorithms vs. Graph Queries, an Example

Graph Algorithm: PageRank

```
// Iterate until convergence
wikipedia.pregel(
  sendMsg = { e =>
    e.sendToDst(e.srcRank * e.weight)
  },
  mergeMsg = _ + _,
  vprog = { (id, oldRank, msgSum) =>
    0.15 + 0.85 * msgSum
  })
```

Graph Query: Wikipedia Collaborators

```
wikipedia.find(
  "(u1)-[e11]->(article1);
   (u2)-[e21]->(article1);
   (u1)-[e12]->(article2);
   (u2)-[e22]->(article2)")
 .select(
   "*",
   "e11.date - e21.date".as("d1"),
   "e12.date - e22.date".as("d2"))
 .sort("d1 + d2".desc).take(10)
```

Before GraphFrames:

Separate Graph Database/ Frameworks to support Graph Algorithms and Graph Queries

Graph algorithms







Standard & custom algorithms
Optimized for batch processing

Graph queries



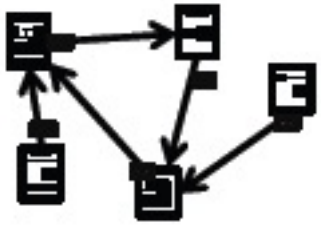




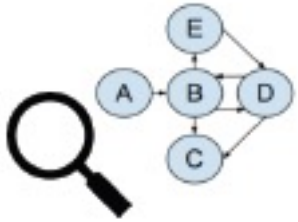
Motif finding
Point queries & updates

GraphFrames: Both algorithms & queries (but not point updates)

System Architecture of GraphFrames



Graph Algorithms



Graph Queries





GraphFrames API

Pattern Query Optimizer

Spark SQL

GraphFrames vs. GraphX

	GraphFrames	GraphX
Built on	DataFrames	RDDs
Languages	Scala, Java, Python	Scala
Use cases	Queries & algorithms	Algorithms
Vertex IDs	Any type (in Catalyst)	Long
Vertex/edge attributes	Any number of DataFrame columns	Any type (VD, ED)
Return types	GraphFrame or DataFrame	Graph[VD, ED], or RDD[Long, VD]

GraphFrames API

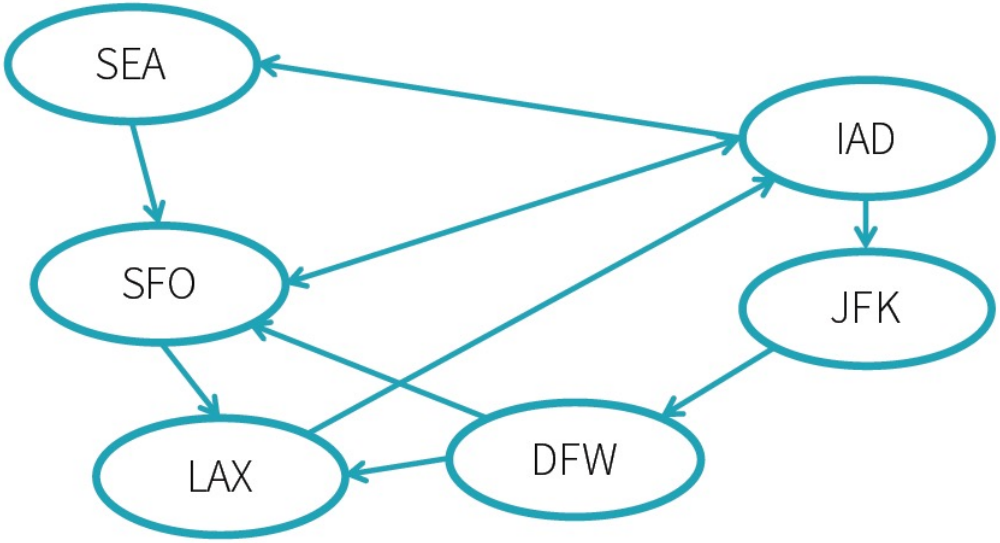
- Unifies graph algorithms, graph queries, and DataFrames
- Available in Scala, Java, and Python

```
class GraphFrame {
  def vertices: DataFrame
  def edges: DataFrame

  def find(pattern: String): DataFrame
  def registerView(pattern: String, df: DataFrame): Unit

  def degrees(): DataFrame
  def pageRank(): GraphFrame
  def connectedComponents(): GraphFrame
  ...
}
```

Representing a Graph in GraphFrames - an Example



"vertices" DataFrame

- 1 vertex per Row
- id: column with unique ID

id	City	State
"JFK"	"New York"	NY
"SEA"	"Seattle"	WA

Extra columns store vertex or edge data (a.k.a. attributes or properties).

"edges" DataFrame

- 1 edge per Row
- src, dst: columns using IDs from vertices.id

src	dst	delay	tripID
"JFK"	"SEA"	45	1058923
"DFW"	"SFO"	-7	4100224

Saving & Loading Graphs

Save & load the DataFrames.

```
vertices = sqlContext.read.parquet(...)
edges = sqlContext.read.parquet(...)
g = GraphFrame(vertices, edges)
g.vertices.write.parquet(...)
g.edges.write.parquet(...)
```

In the future...

- SQL data sources for graph formats

Build and show a Graph in GraphFrames - an Example

```
> # Load & prepare vertices DataFrame  # Set File Paths tripdelaysFilePath = "/dat ...
> # Prepare edges DataFrame  # Build `departureDelays_geo` DataFrame #  Obtain key ...
```

```
> display(airports)
```

id	City	State	Country
LRD	Laredo	TX	USA
INL	International Falls	MN	USA
SAF	Santa Fe	NM	USA
MSO	Missoula	MT	USA
GRR	Grand Rapids	MI	USA

```
> from graphframes import *

  # Note that we already cached our Vertices and Edges
  tripGraph = GraphFrame(airports, departureDelays_geo)
> tripGraph.vertices
```

```
Out[6]: DataFrame[id: string, City: string, State: string, Country: string]
```

```
> # Airport and trip counts
  print "Airports: %d" % tripGraph.vertices.count()
  print "Trips: %d" %  tripGraph.edges.count()
```

```
Airports: 279
Trips: 1361141
```

Example of Simple Queries with GraphFrames

What trips are most likely to have significant delays?

```
> display(tripGraph.edges
          .groupBy("src", "dst")
          .avg("delay")
          .sort(desc("avg(delay)")))
```

src	dst	avg(delay)
JFK	JAC	322
JAC	JFK	307
SYR	BTV	257
CRW	DTW	131

```
> # States with the longest cumulative delays (with individual delays > 100 minutes) (origin: Seattle)
  srcSeattleByState = tripGraph.edges.filter("src = 'SEA' and delay > 100")
  display(srcSeattleByState)
```

```
> display(tripGraph.degrees
          .sort(desc("degree"))
          .limit(10))
```

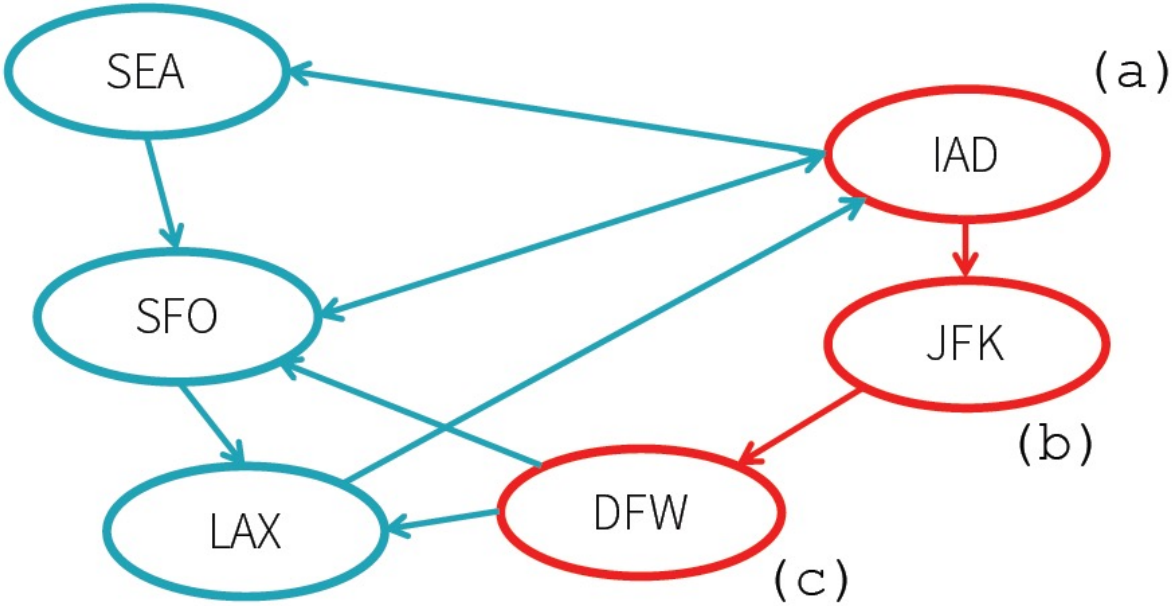
Motif Finding with GraphFrames

Search for structural patterns within a graph.

```
val paths: DataFrame =
  g.find("(a)-[e1]->(b);
          (b)-[e2]->(c);
         !(c)-[]->(a)")
```

Then filter using vertex & edge data.

```
paths.filter("e1.delay > 20")
```



E.G.: Study City/Flight Relationships via Motif-Finding

What delays might we blame on SFO?

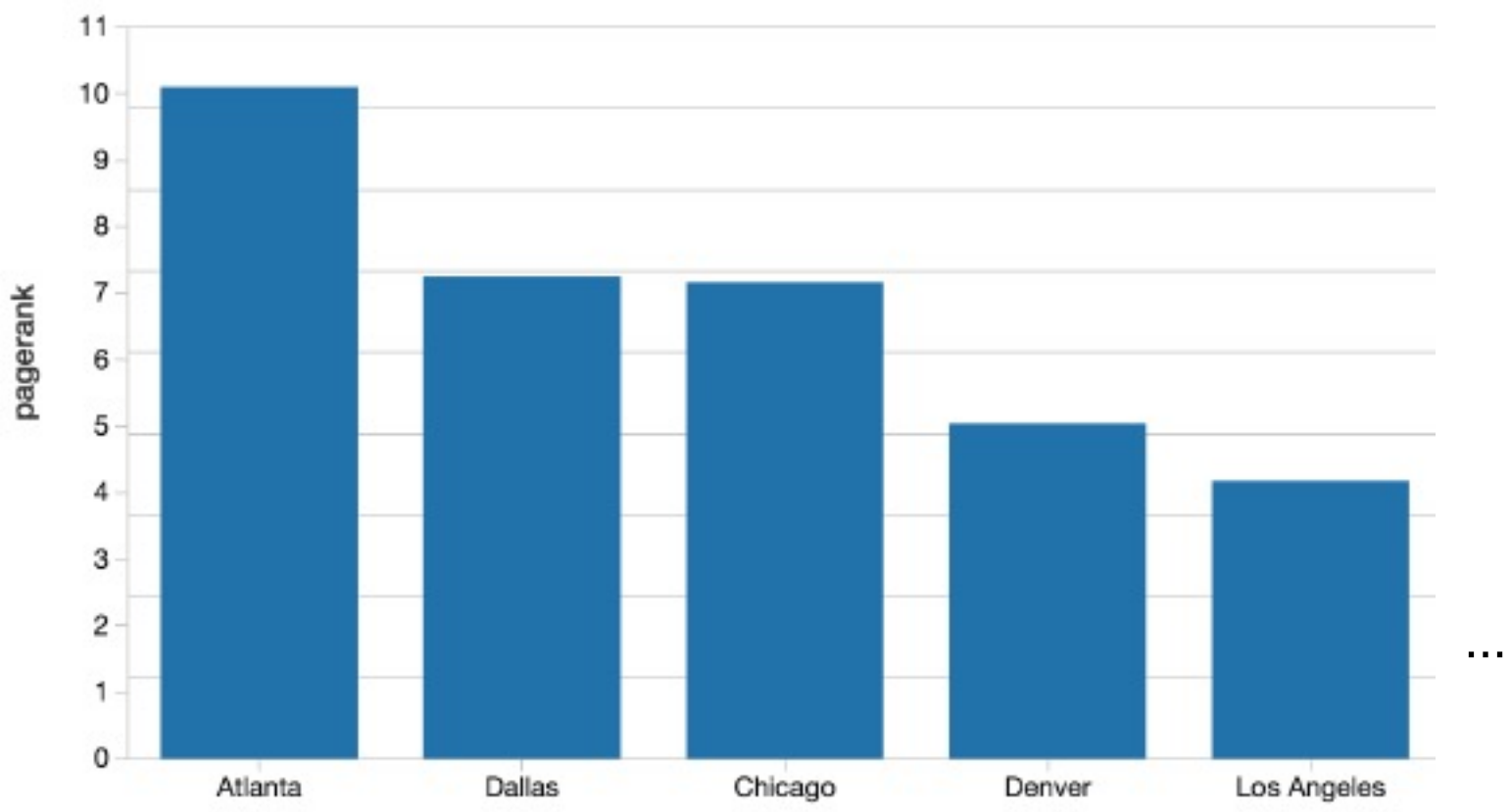
```
> motifs = tripGraph.find("(a)-[e1]->(b); (b)-[e2]->(c)")\
    .filter("(b.id = 'SFO') and (e1.delay > 500 or e2.delay > 500) and e1.tripid < e2.tripid")
  display(motifs)
```

e1	a
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City
▸{"tripid":1011126,"localdate":"2014-01-01T11:26:00.000+0000","delay":-4,"distance":1421,"src":"IAH","dst":"SFO","city_src":"Houston","city_dst":"San Francisco","state_src":"TX","state_dst":"CA"}	▸{"id":"IAH","City

Showing the first 1000 rows.

E.G.: Determine Airport Importance via PageRank

```
> ranks = tripGraph.pageRank(maxIter=5)
  display(ranks.vertices
          .sort(ranks.vertices.pagerank.desc())
          .limit(10))
```



...

Built-in Graph Algorithms for GraphFrames

Find important vertices
- PageRank

Find paths between sets of vertices
- Breadth-first search (BFS)
- Shortest paths

Find groups of vertices (components, communities)
- Connected components
- Strongly connected components
- Label Propagation Algorithm (LPA)

Other
- Triangle counting
- SVDPlusPlus

Built-in Algorithm Implementation for GraphFrames

Mostly wrappers for GraphX

- PageRank
- Shortest paths
- Connected components
- Strongly connected components
- Label Propagation Algorithm (LPA)
- SVDPlusPlus

Some algorithms implemented using DataFrames

- Breadth-first search
- Triangle counting

GraphX compatibility for GraphFrames

Simple conversions between GraphFrames & GraphX.

```
val g: GraphFrame = ...

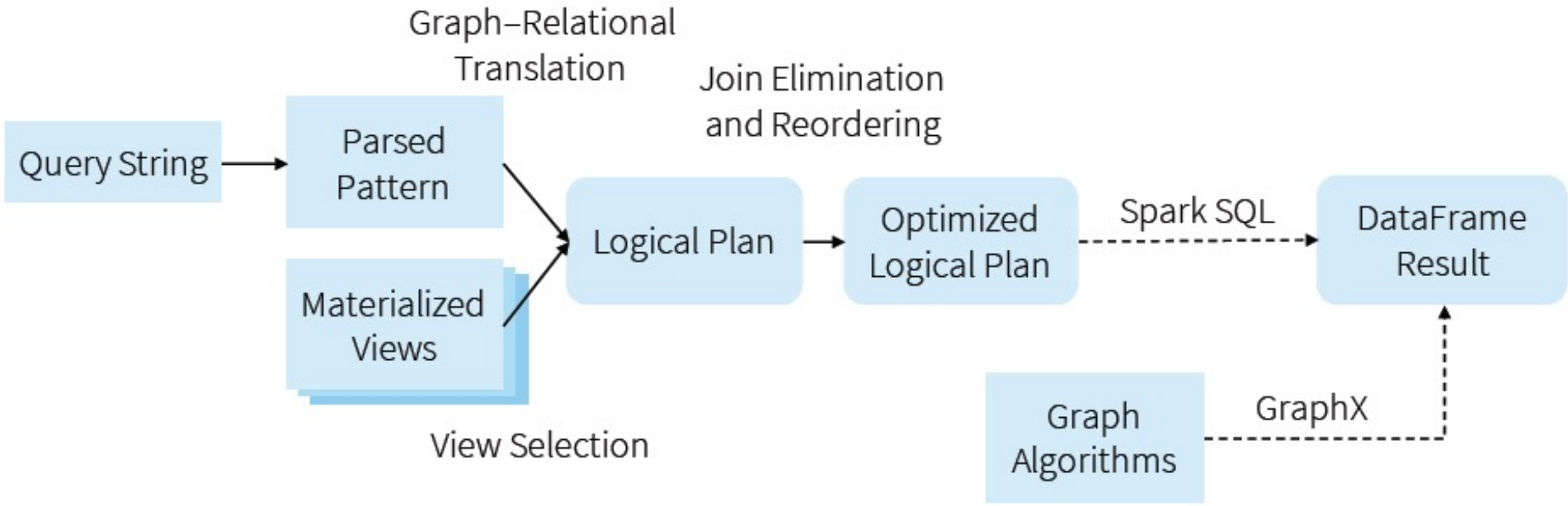
// Convert GraphFrame → GraphX
val gx: Graph[Row, Row] = g.toGraphX

// Convert GraphX → GraphFrame
val g2: GraphFrame = GraphFrame.fromGraphX(gx)
```


Vertex & edge attributes are Rows in order to handle non-Long IDs

Wrapping existing GraphX code: See Belief Propagation example:
https://github.com/graphframes/graphframes/blob/master/src/main/scala/org/graphframes/examples/BeliefPropagation.scala

GraphFrames System Implementation



Resources for Learning more about GraphFrames

User guide + API docs http://graphframes.github.io/

- Quick-start
- Overview & examples for all algorithms
- Also available as executable notebooks:
  - Scala: http://go.databricks.com/hubfs/notebooks/3-GraphFrames-User-Guide-scala.html
  - Python: http://go.databricks.com/hubfs/notebooks/3-GraphFrames-User-Guide-python.html

Blog posts

- Intro: https://databricks.com/blog/2016/03/03/introducing-graphframes.html
- Flight delay analysis: https://databricks.com/blog/2016/03/16/on-time-flight-performance-with-spark-graphframes.html

https://www.datascience.com/blog/graph-computations-apache-spark

Another Graph Processing Framework for Spark:

Jiawei Jiang et al, "PSGraph: How Tencent trains extremely large graphs with Spark ?", IEEE ICDE 2020