Introduction to Parallel & Distributed Computing

Graph Processing Frameworks

Lecture 17, Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

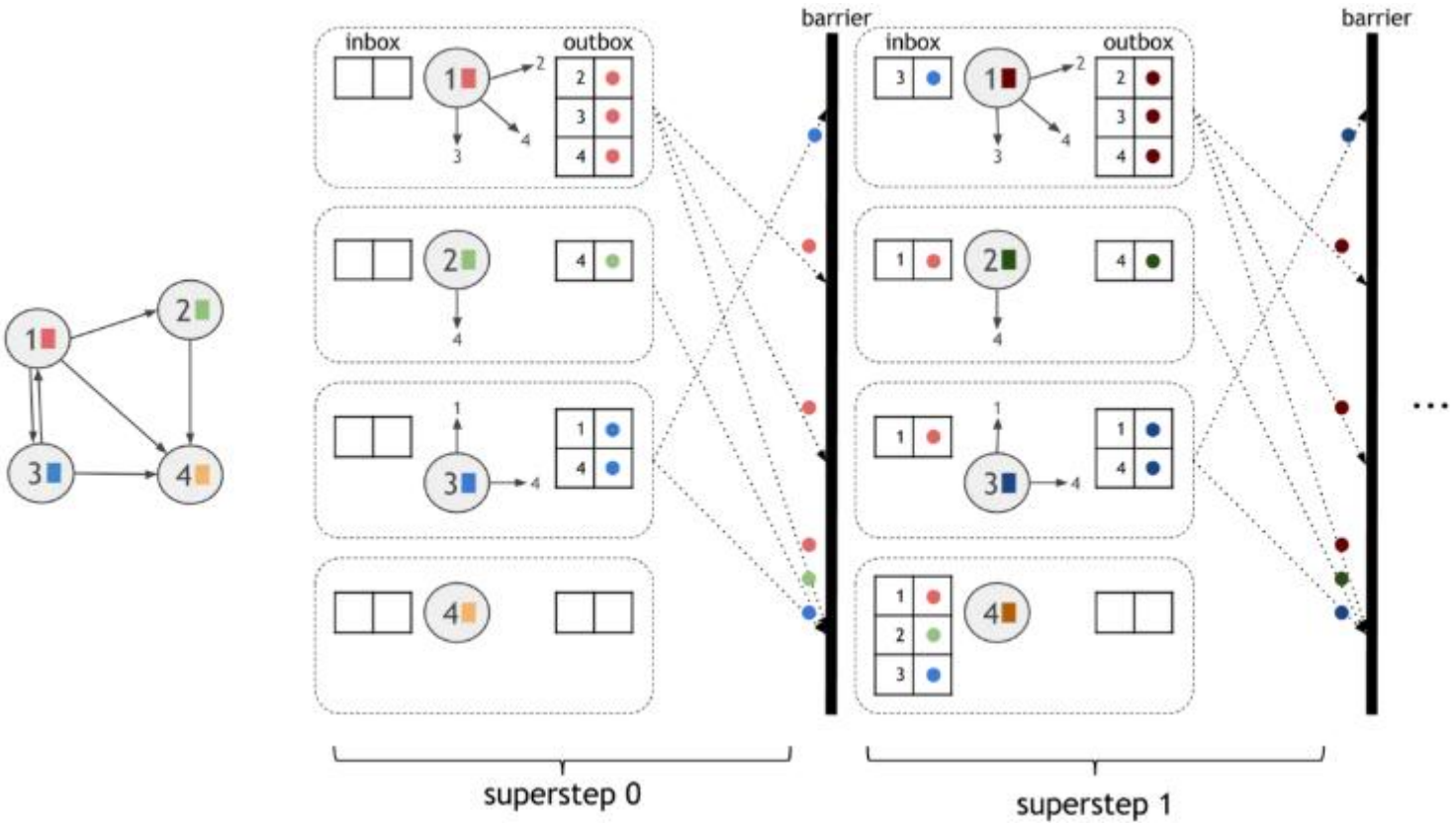
Issues in Privious Parallel Frameworks

- **High-level (e.g., MapReduce)**
  - Not expressive enough
- **Low-level (e.g., MPI)**
  - Difficult to program
- **New abstractions needed**
  - For graph processing, machine learning, etc.

Graph Processing Frameworks

- **Distributed machines**
  - Pregel, GraphLab, PowerGraph, GraphX, etc.
- **Single machine**
  - Ligra, GraphChi, X-Stream, Galois, etc.
- **GPU oriented**
  - Single machine: Garaph, Tigr, CuSha, MapGraph, etc.
  - Distributed: Lux, etc.

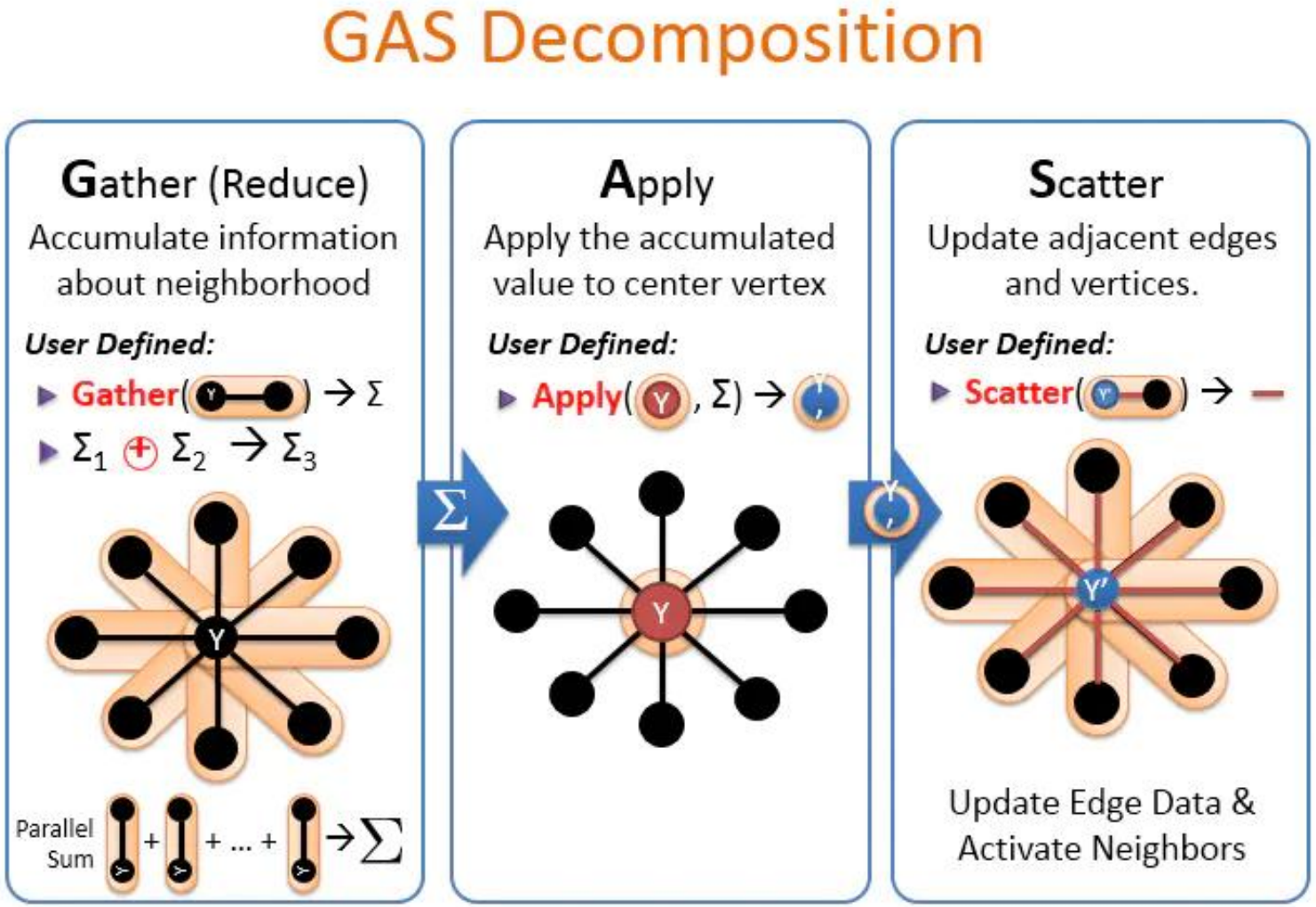
Pregel's Bulk Synchronization



Short Introductions

- **Pregel [SIGMOD'10]**
  - Think like a vertex
  - Supersteps
  - Message passing
- **GraphLab [VLDB'12]**
  - Vertex-centric
  - Asynchronous
    - No barrier sync
  - Shared memory abstraction
    - No message passing
    - Fold/reduce instead

PowerGraph's Gather-Apply-Scatter



Short Introductions

- **PowerGraph [OSDI'12]**
  - For power law graphs
  - Shared-memory GAS system

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
   return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
   rnew = 0.15 + 0.85 * acc
   D_u.delta = (rnew - D_u.rank)/
               #outNbrs(u)
   D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u, D_(u,v), D_v):
   if(|D_u.delta|>ε) Activate(v)
   return delta
```

```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
     SendMsg(nbr, vertex.val/num_out_nbrs);
```

```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
     accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

Short Introductions

- **Ligra [PPoPP'13]**
  - Takes advantage of "frontier-based" nature of many algorithms
  - (active set is dynamic and often small)
- **Garaph [ATC'17]**
  - Adopt the GAS interface
  - Replication-based gather
    - to relieve write contention & work imbalance

Short Introductions

- **Lux [VLDB'17]**
  - Programming interface: init, compute, update
    - Similar to GAS interface
  - Support pull execution instead of push execution
    - Pull: process all vertices and edges at each iteration
    - Push: maintains frontier of vertices
- **Tigr [ASPLOS'18]**
  - Basic idea: split a high degree node into multiple nodes
  - Uniform-degree tree transformation (UDT)

Selected Examples

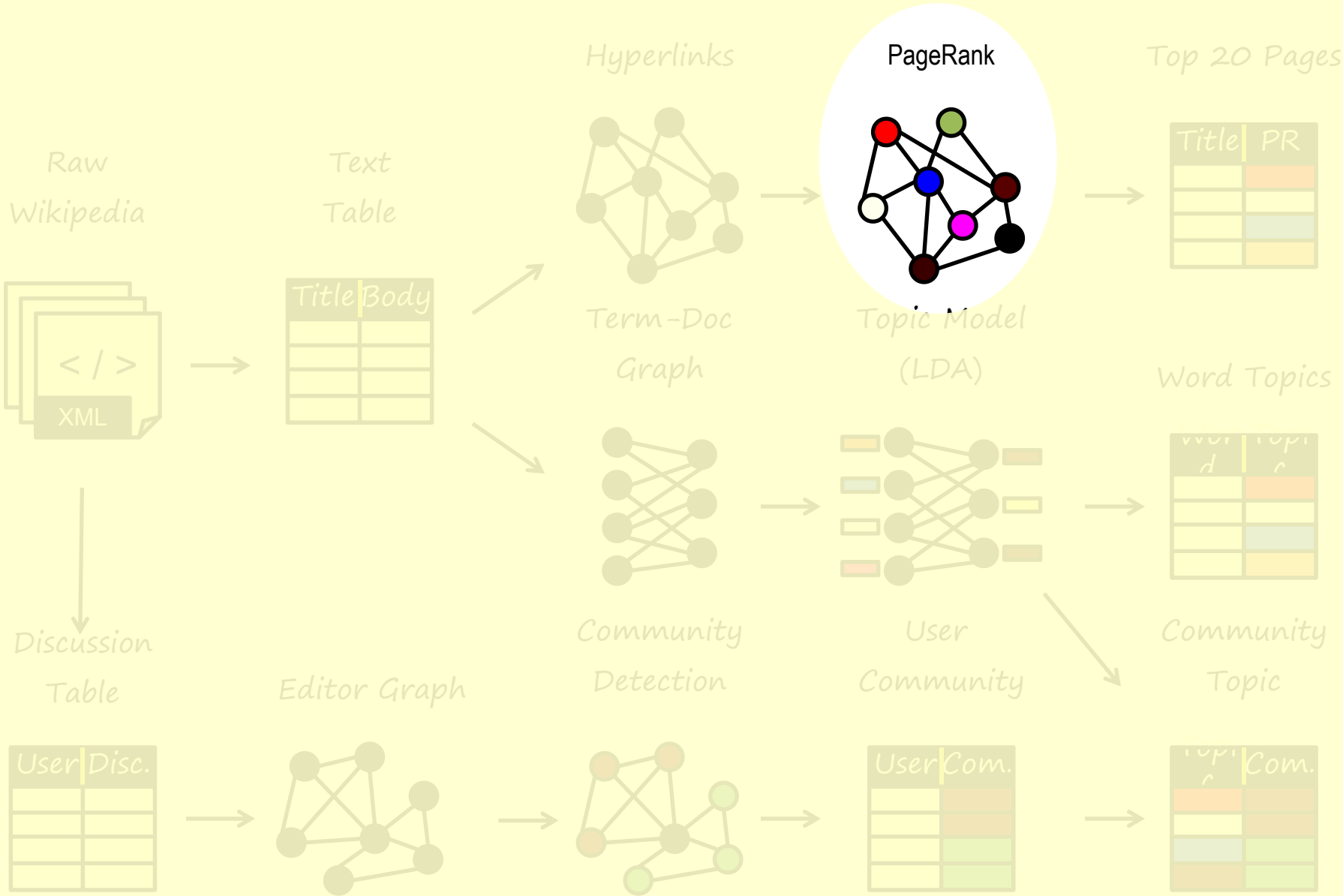
- GraphX
- Galois

GraphX: Outline

- **Motivation**
  - Unifying Data-Parallel and Graph-Parallel Analytics
- **Abstraction**
- **System design**

J. Gonzalez et al., "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics" Strata 2014

Graphs are Central to Analytics



PageRank: Identifying Leaders

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

Rank of user *i*

Weighted sum of neighbors' ranks

Update ranks in parallel

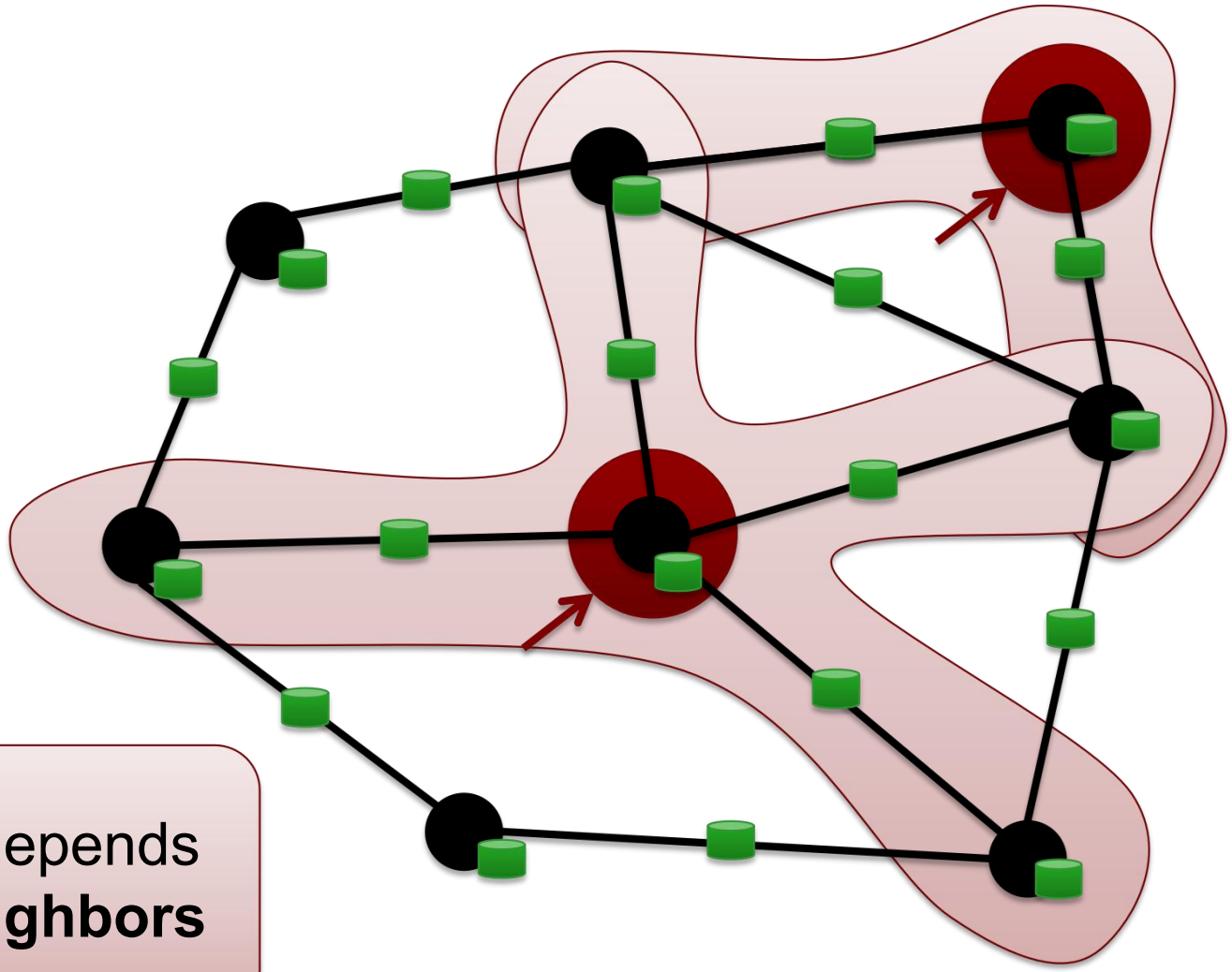
Iterate until convergence

J. Gonzalez et al., "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics" Strata 2014

13

The Graph-Parallel Pattern

Model / Alg. State



Computation depends only on the **neighbors**

14

Many Graph-Parallel Algorithms

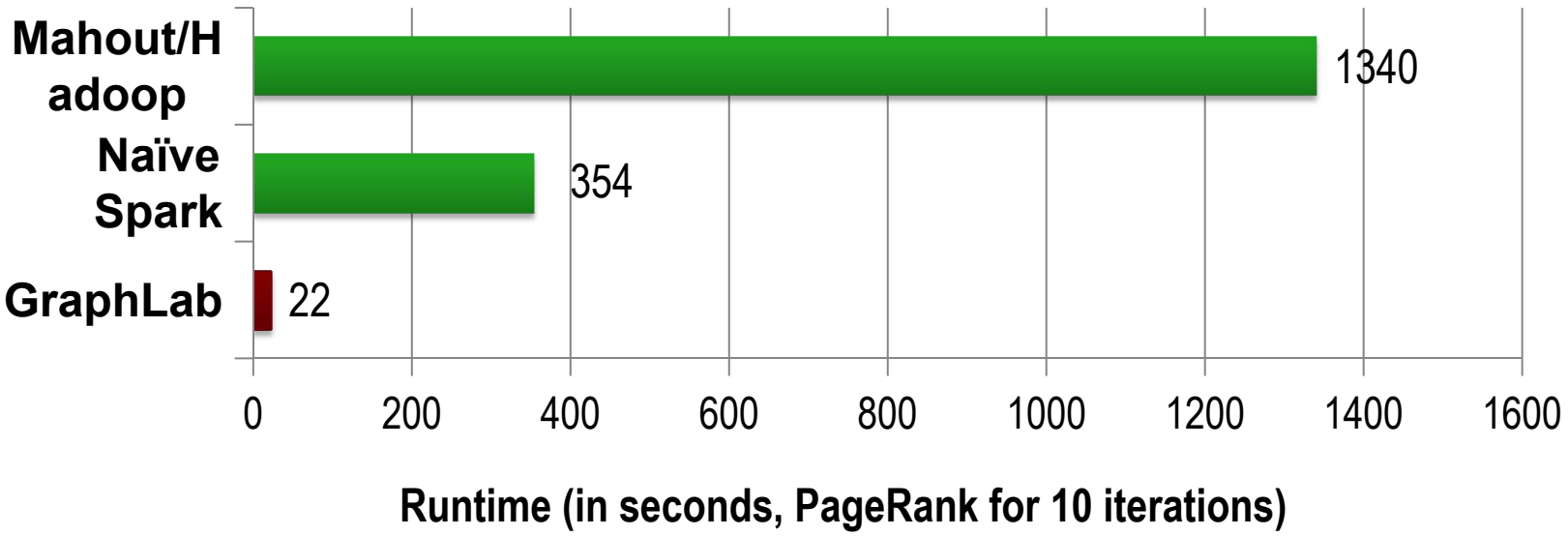
- **Collaborative Filtering**
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - Tensor Factorization
- **Structured Prediction**
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- **Semi-supervised ML**
  - Graph SSL
  - CoEM
- **Community Detection**
 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

Graph-Parallel Systems

Expose specialized APIs to simplify graph programming.

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

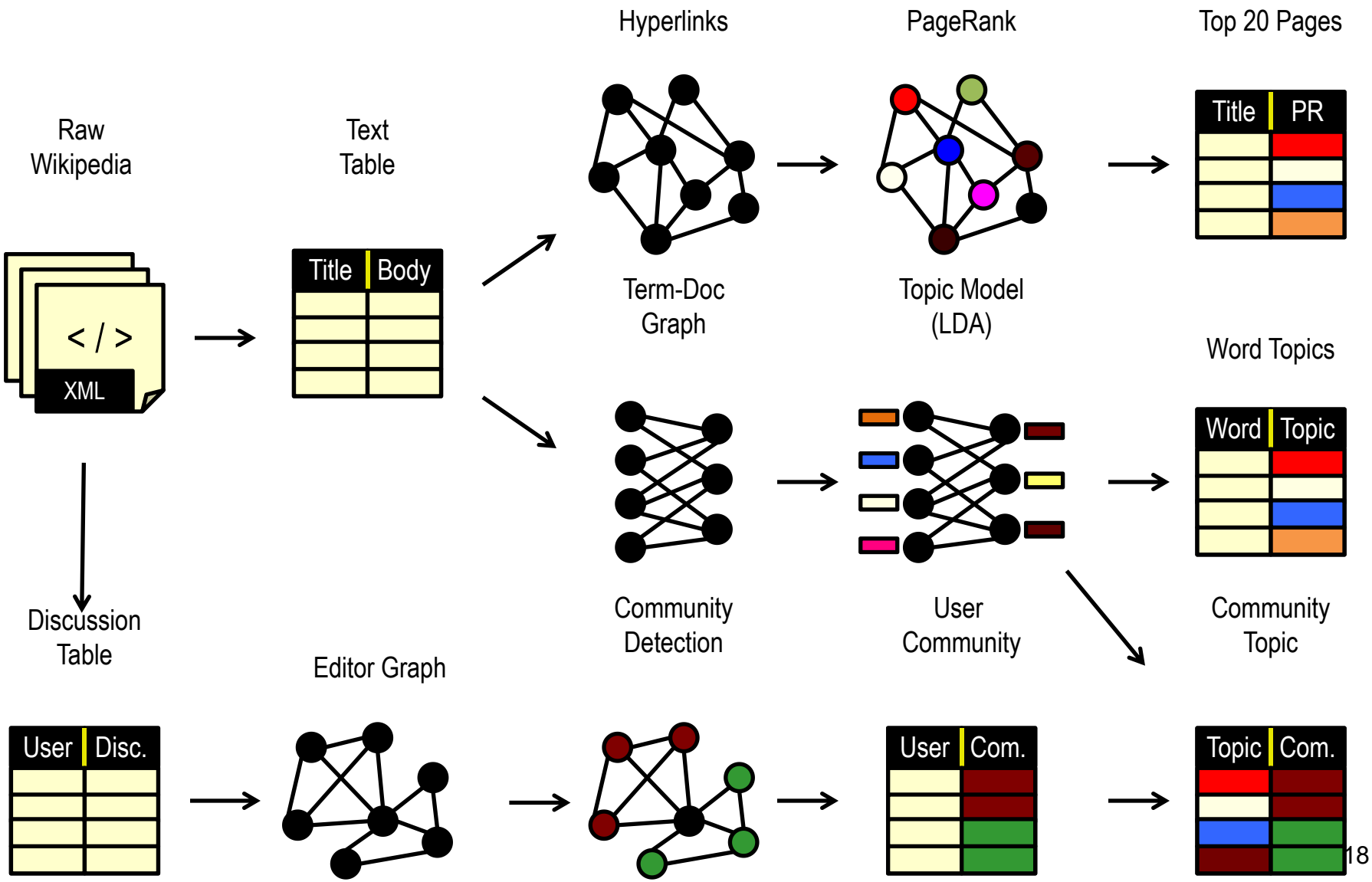
PageRank on the Live-Journal Graph



GraphLab is *60x faster* than Hadoop

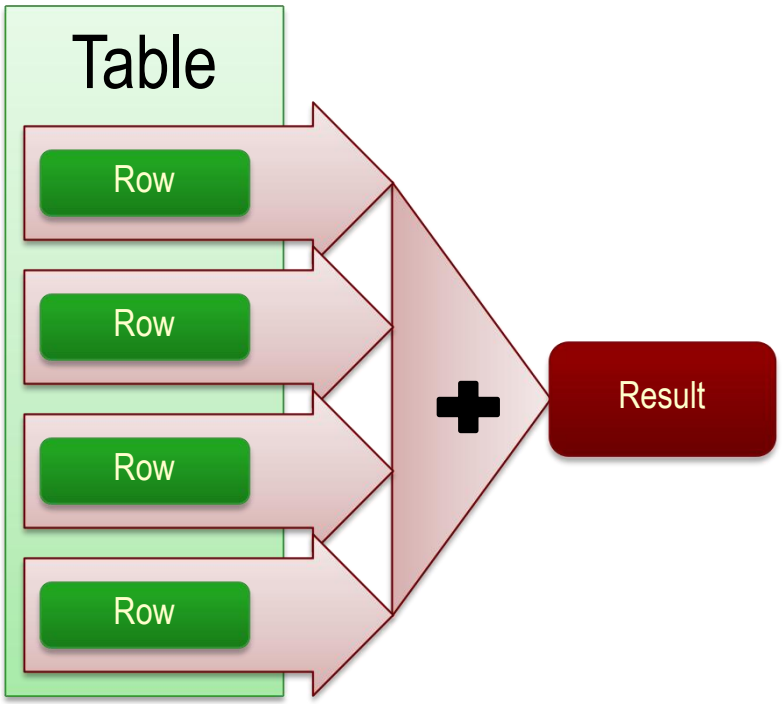
GraphLab is *16x faster* than Spark

Graphs are Central to Analytics

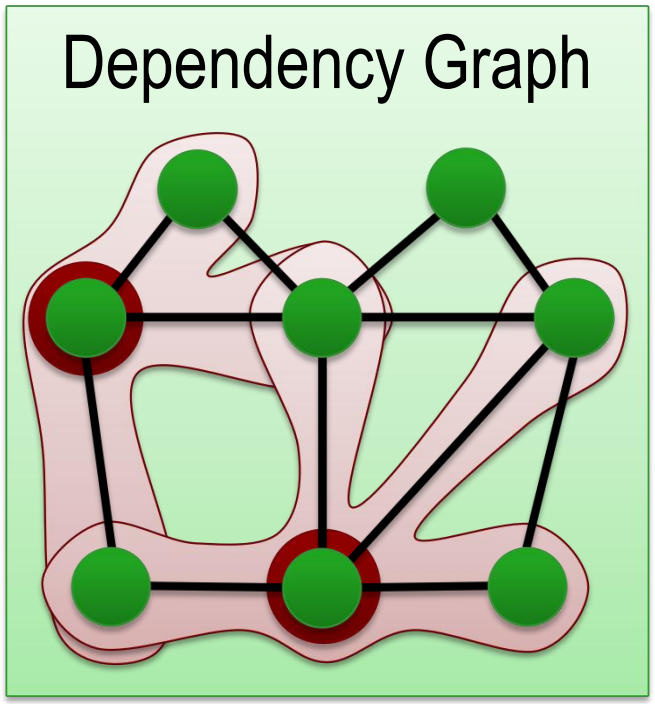


Separate Systems to Support Each View

Table View



Graph View



Difficult to Program and Use

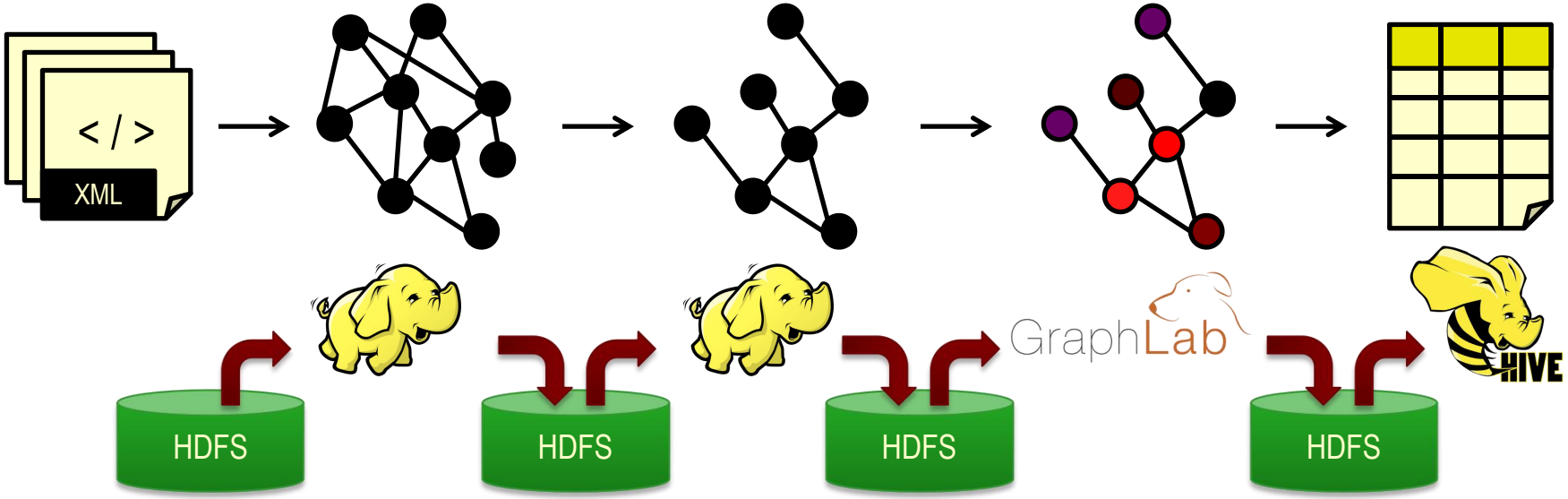
- **Users must Learn, Deploy, and Manage multiple systems**

- **Leads to brittle and often complex interfaces**

20

Inefficient

Extensive data movement and duplication across the network and file system

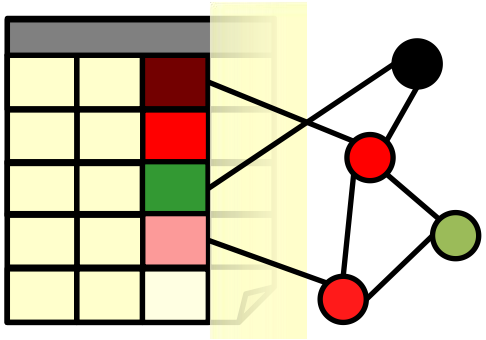


Limited reuse internal data-structures across stages

Solution: The GraphX Unified Approach

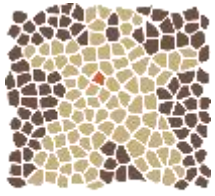
New API

Blurs the distinction between Tables and Graphs



New System

Combines Data-Parallel Graph-Parallel Systems



APACHE GIRAPH

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

Tables and Graphs are composable views of the same physical data

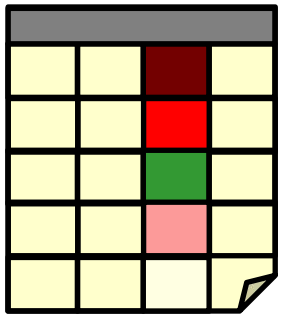
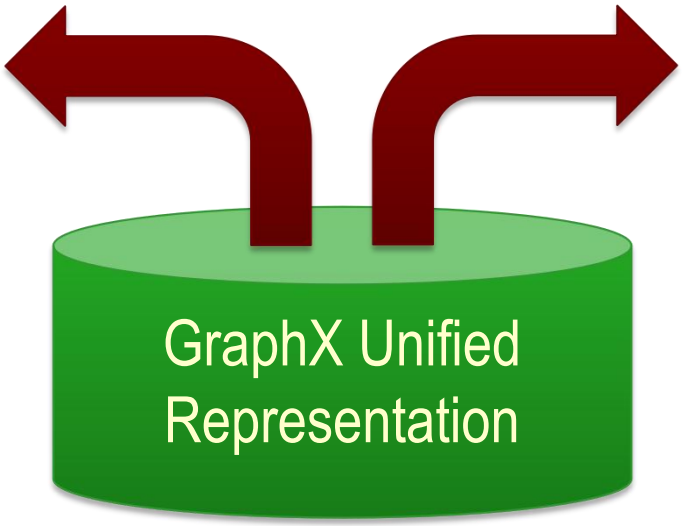
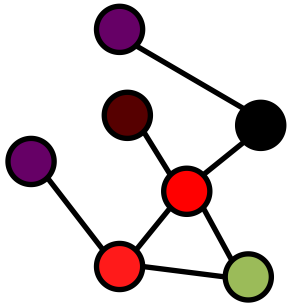


Table View



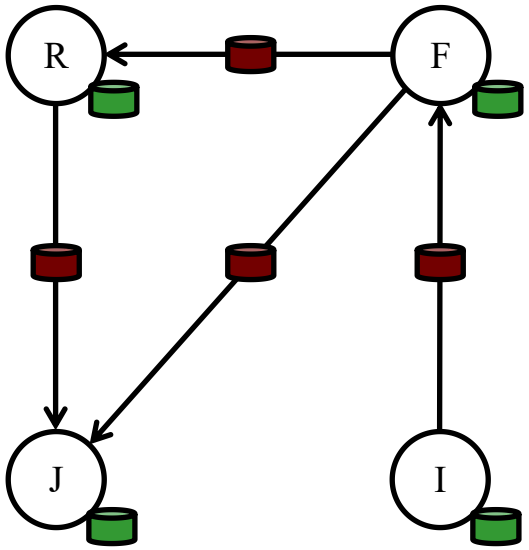


Graph View

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

View a Graph as a Table

Property Graph



Vertex Property Table

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

Edge Property Table

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

Table Operators

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

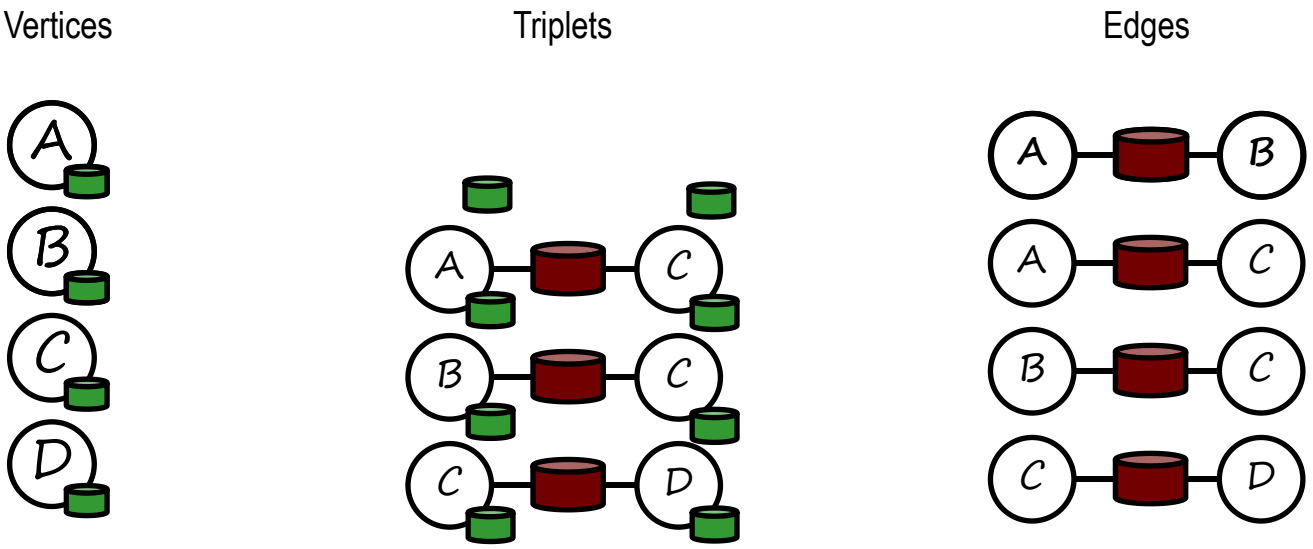
map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
leftOuterJoin	cross	save
rightOuterJoin	zip	...

Graph Operators

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

Triplets Join Vertices and Edges

- **The *triplets* operator joins vertices and edges:**

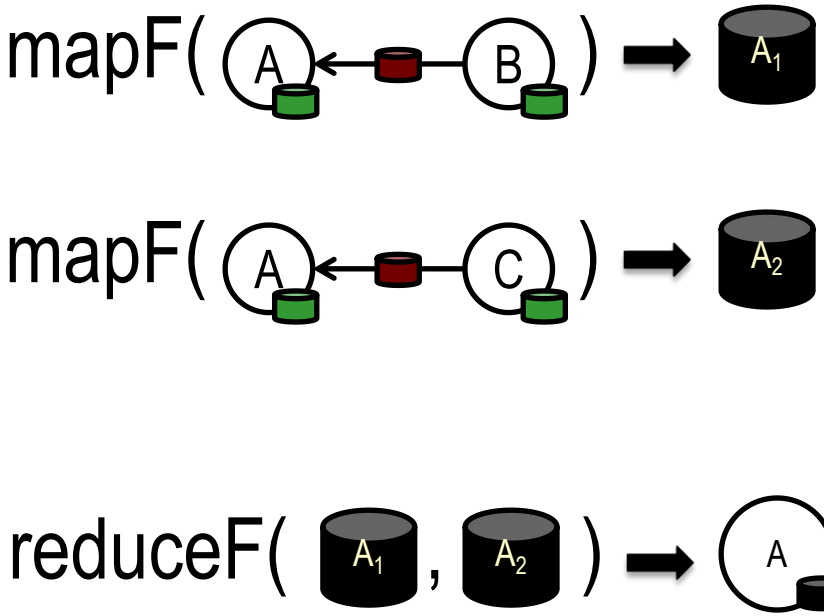


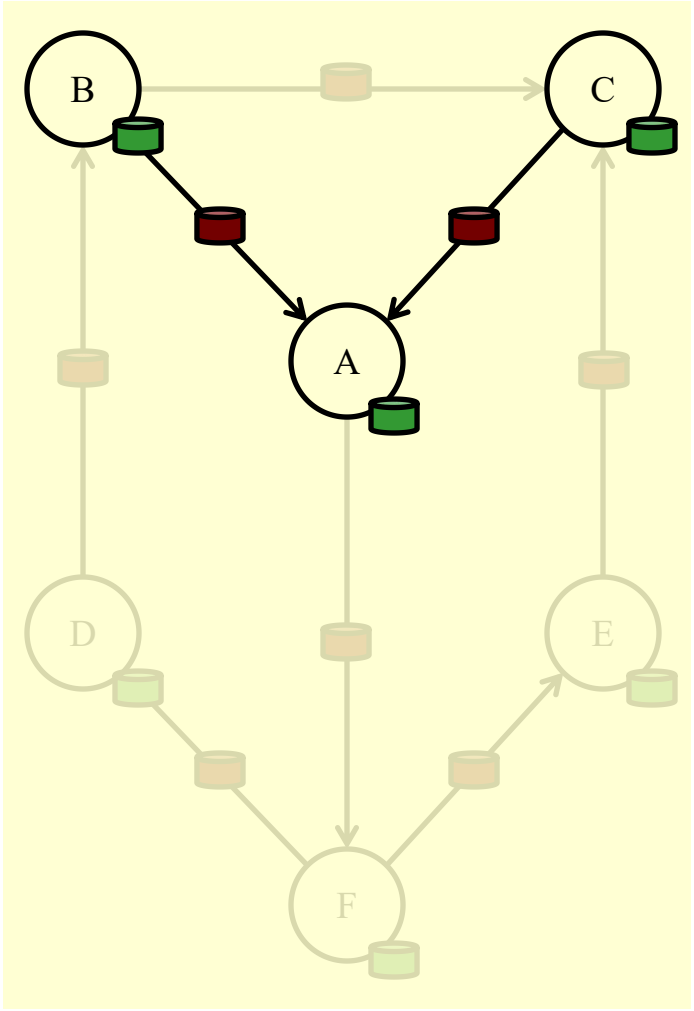
The ***mrTriplets*** operator sums adjacent triplets.

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

Map Reduce Triplets

- ***Map-Reduce* for each vertex**

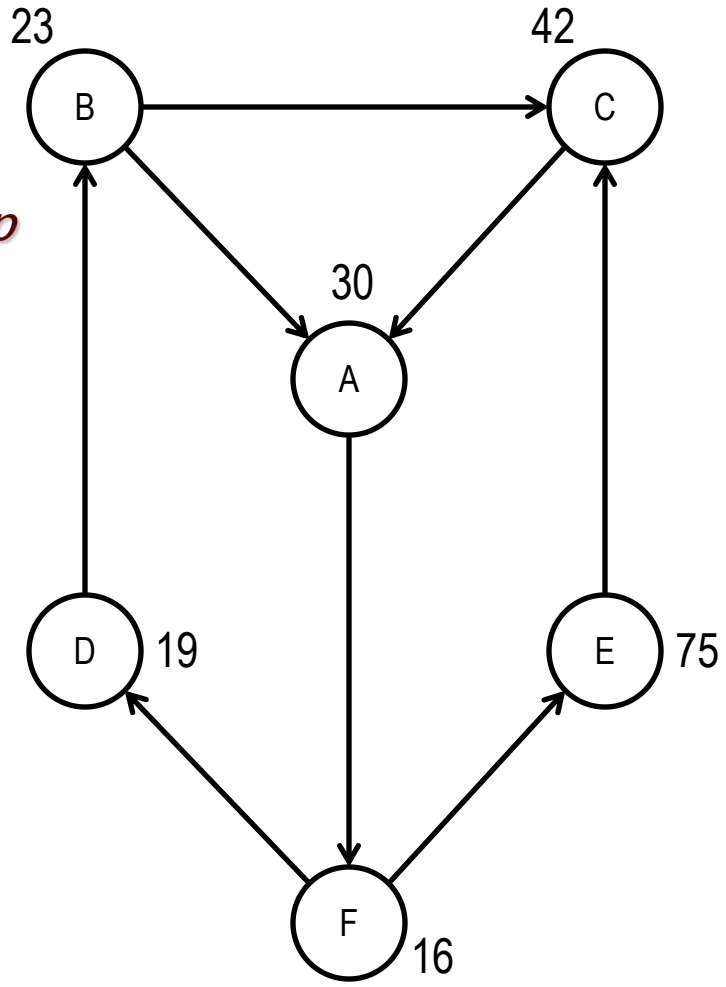




Example: Oldest Follower

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

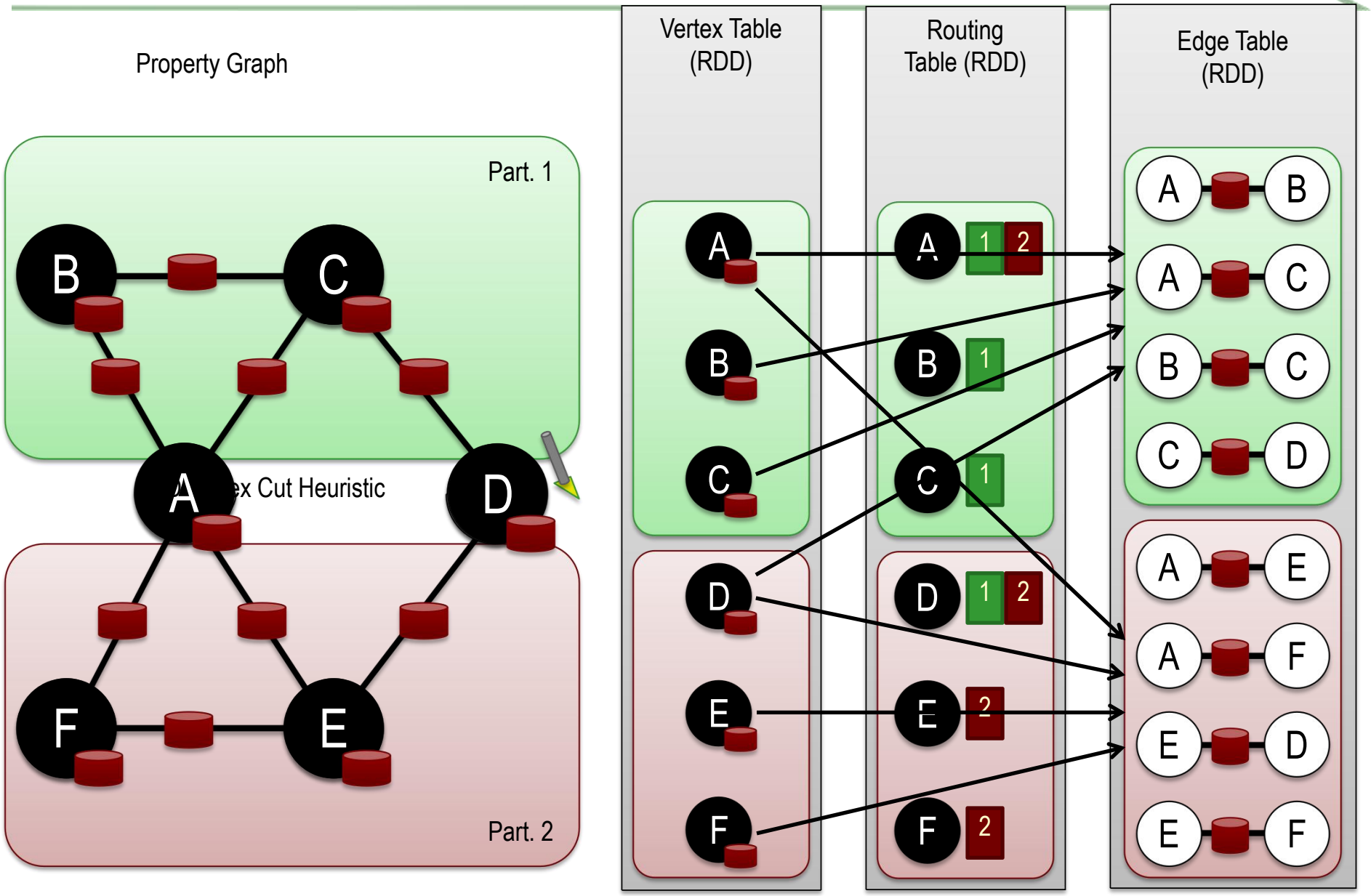
```
val oldestFollowerAge = graph
  .mrTriplets(
    e=> (e.dst.id, e.src.age),//Map
    (a,b)=> max(a, b) //Reduce
  )
  .vertices
```



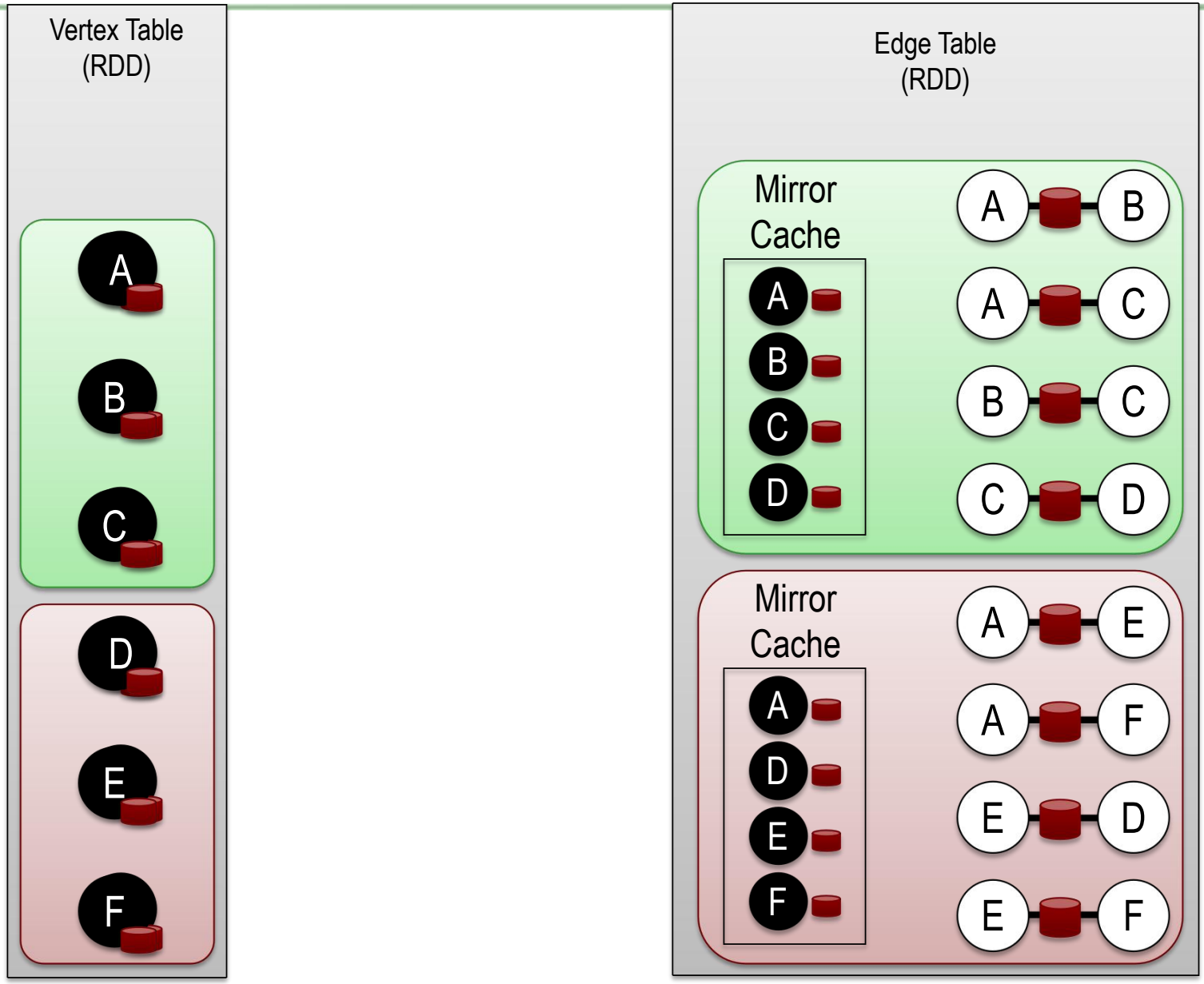
GraphX: Outline

- **Motivation**
  - Unifying Data-Parallel and Graph-Parallel Analytics
- **Abstraction**
- **System design**

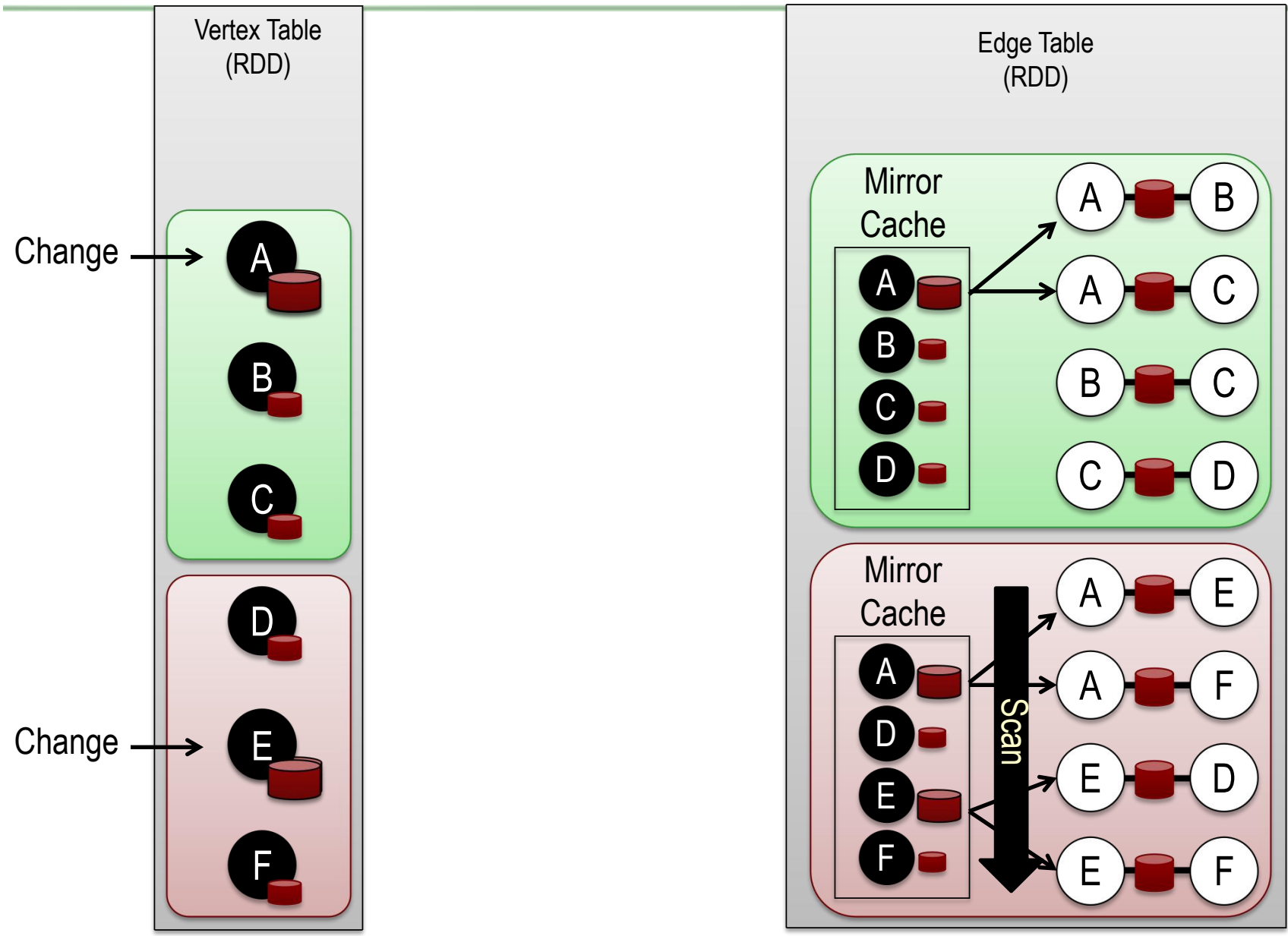
Distributed Graphs as Tables (RDDs)



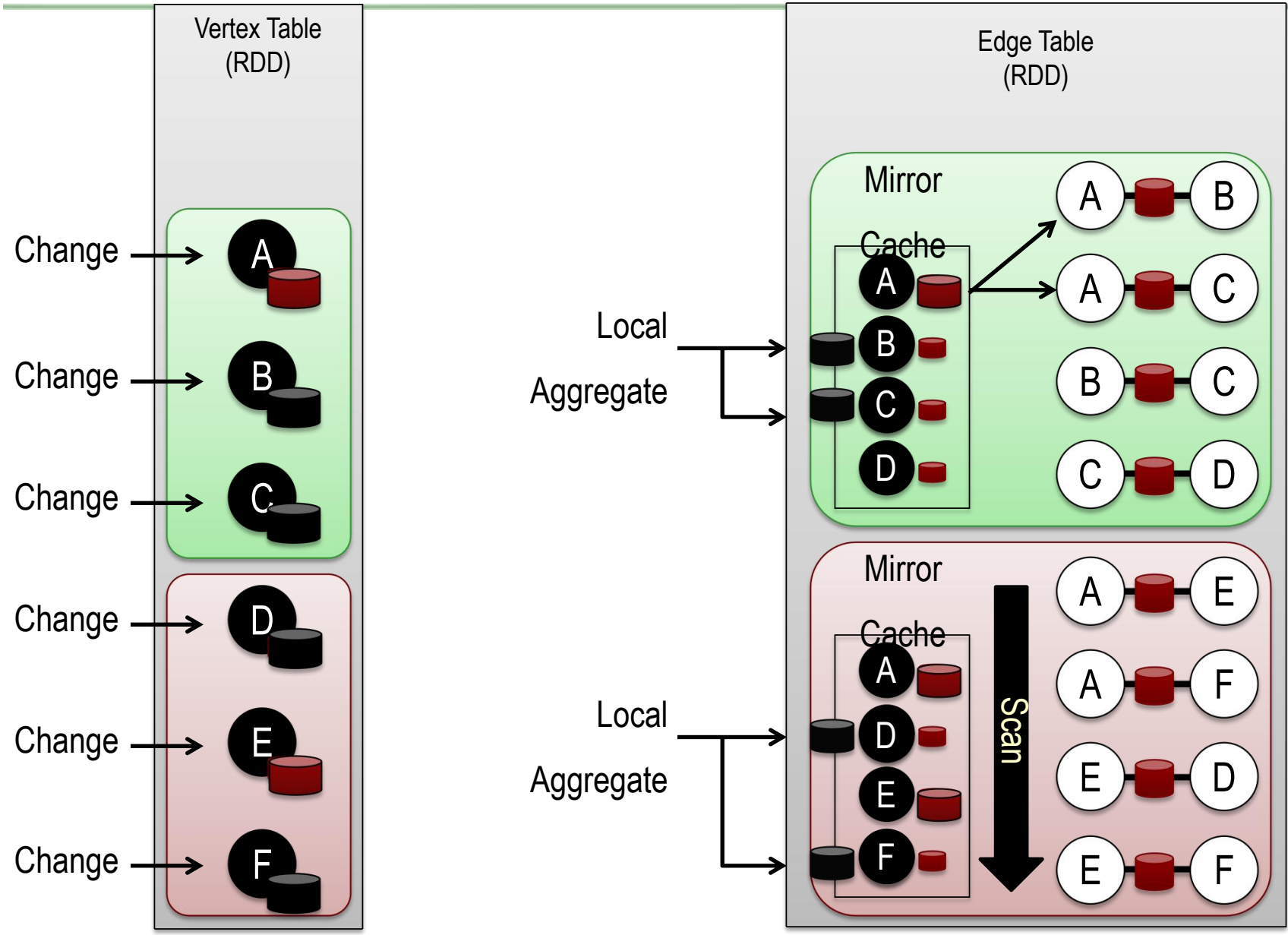
Caching for Iterative mrTriplets



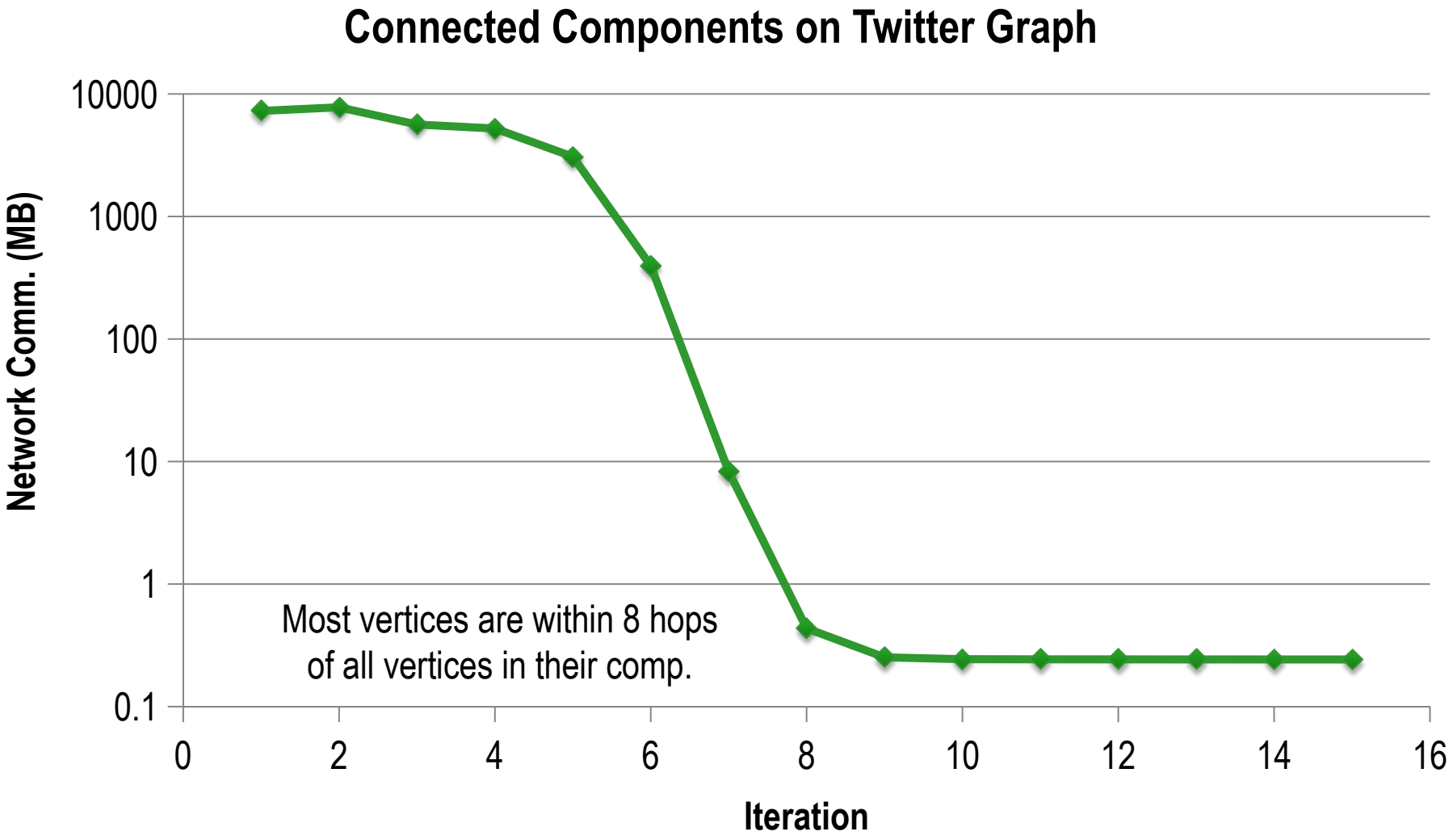
Incremental Updates for Iterative mrTriplets



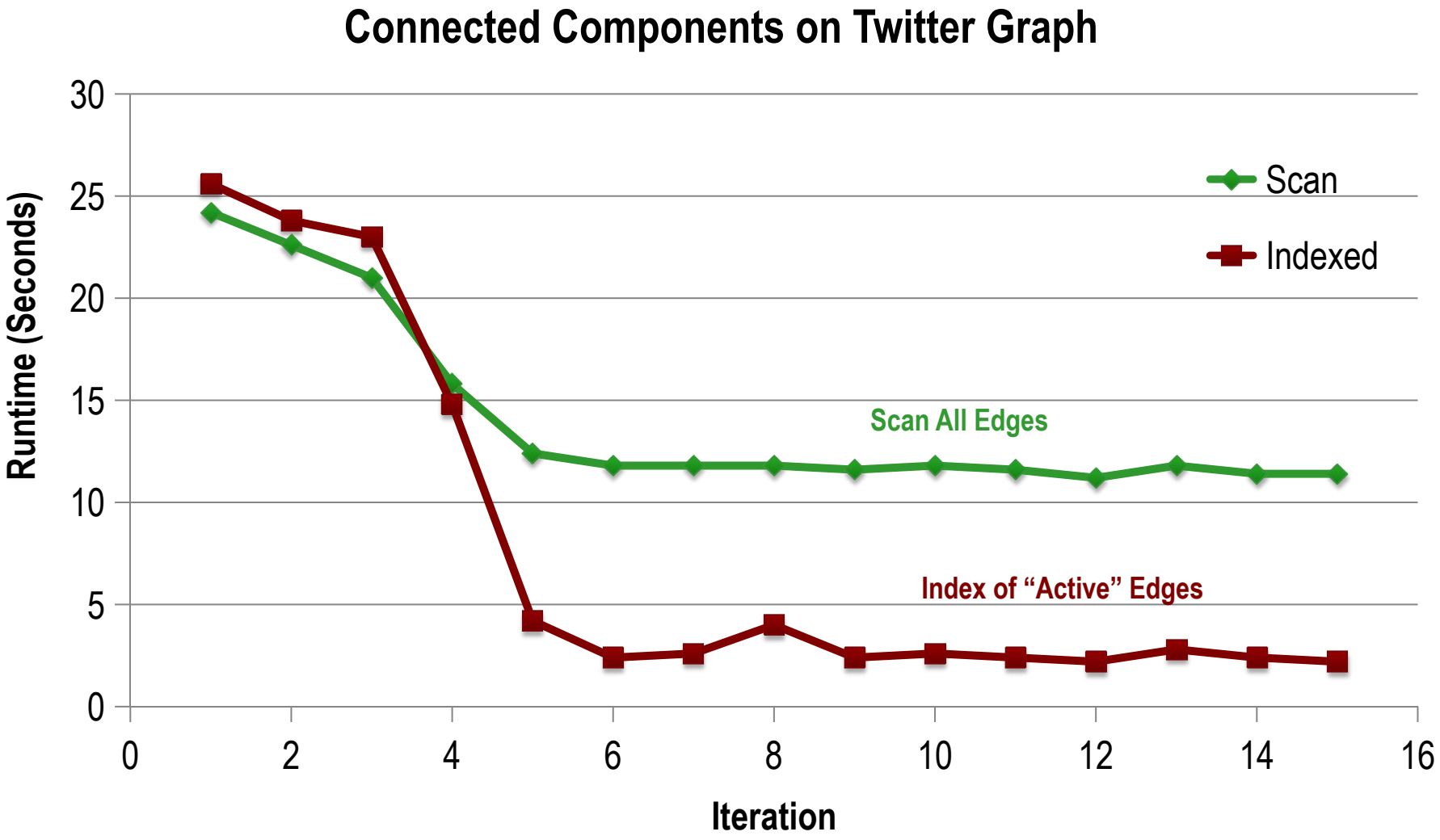
Aggregation for Iterative mrTriplets



Reduction in Communication Due to Cached Updates



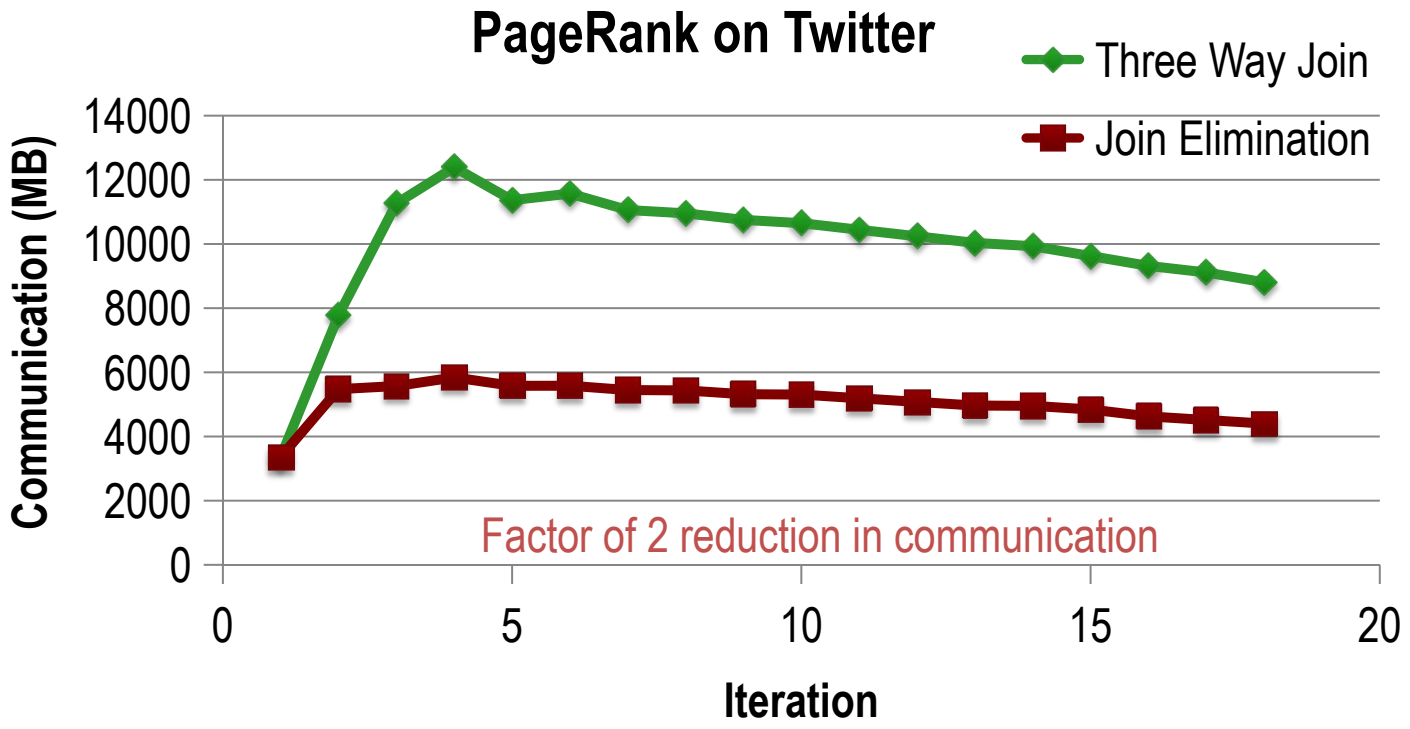
Benefit of Indexing Active Edges



Join Elimination

Identify and bypass joins for unused triplets fields

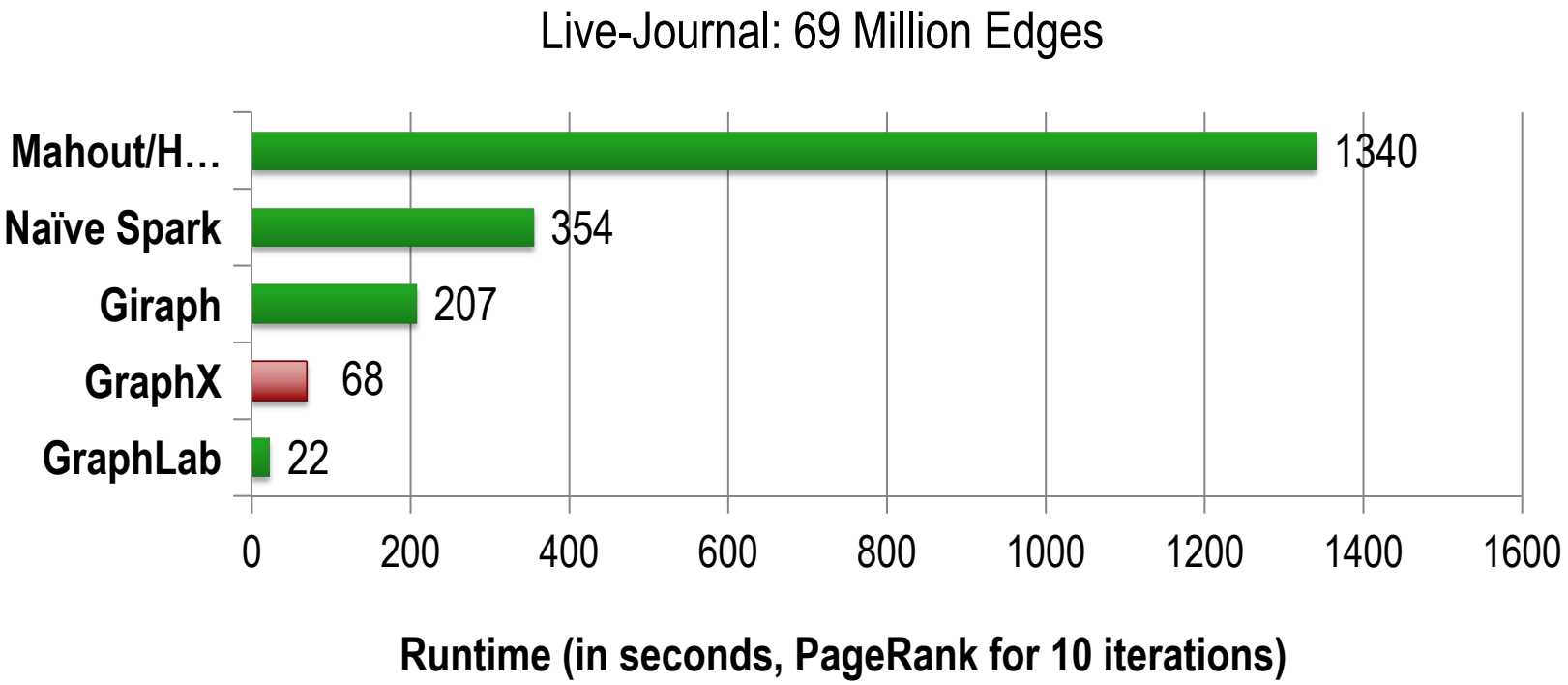
- *Example:* PageRank only accesses source attribute



Additional Query Optimizations

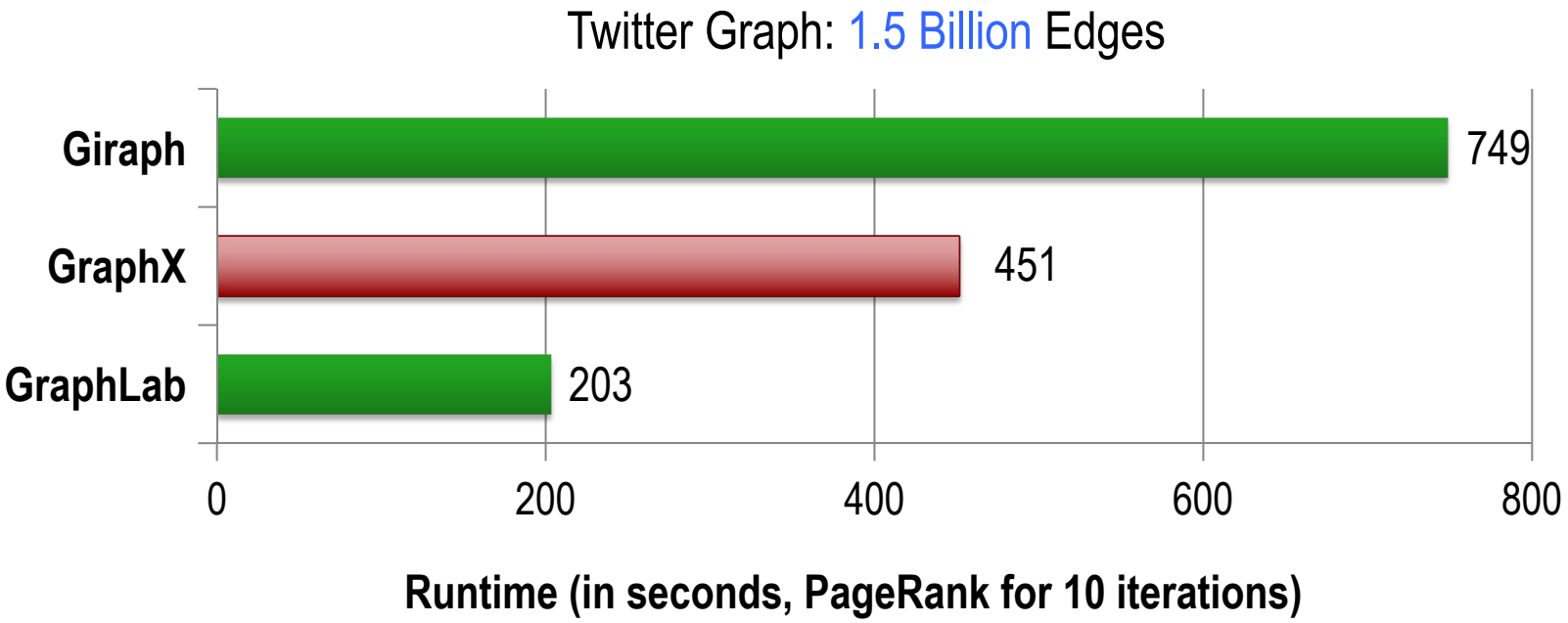
- **Indexing and Bitmaps:**
  - To accelerate joins across graphs
  - To efficiently construct sub-graphs
- **Substantial Index and Data Reuse:**
  - Reuse routing tables across graphs and sub-graphs
  - Reuse edge adjacency information and indices

Performance Comparisons



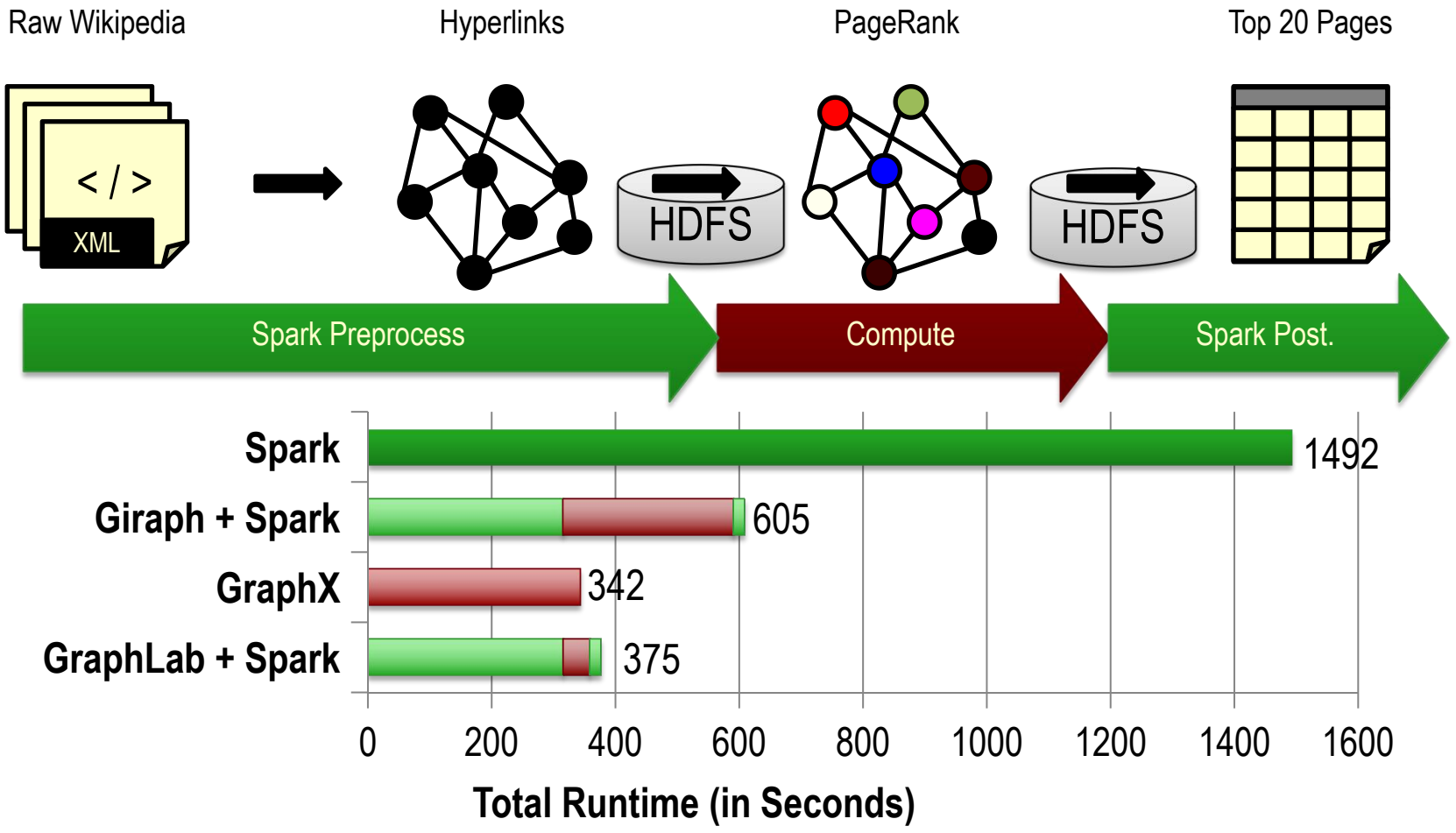
GraphX is roughly 3x slower than GraphLab

GraphX scales to larger graphs



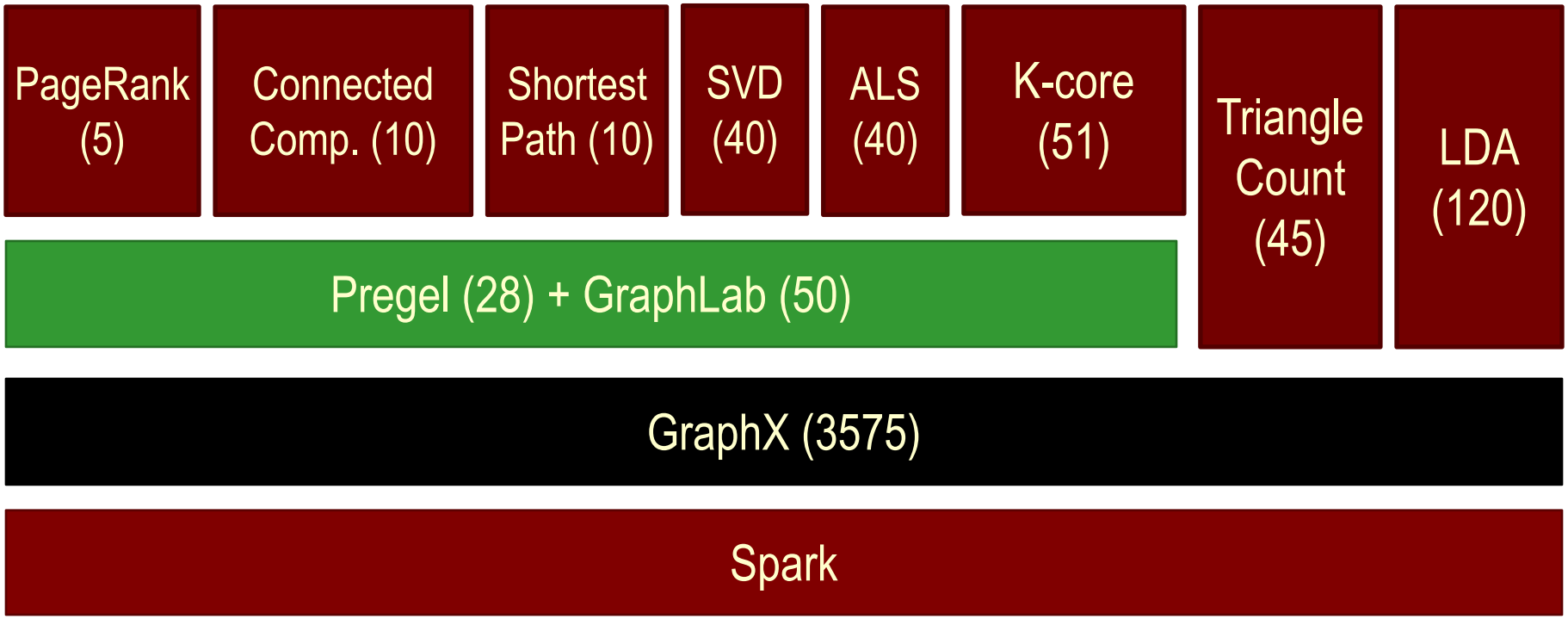
- **GraphX is roughly 2x slower than GraphLab**
  - Scala + Java overhead: Lambdas, GC time, …
  - No shared memory parallelism: 2x increase in comm.

A Small Pipeline in GraphX



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

The GraphX Stack (Lines of Code)



Conclusion and Observations

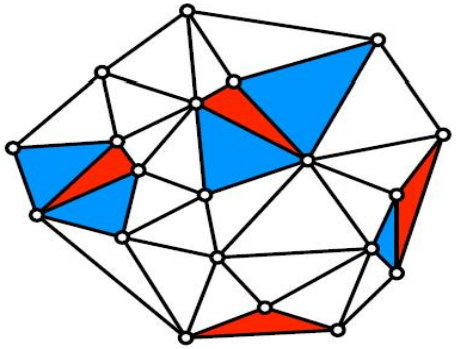
- **Domain specific views: *Tables* and *Graphs***
  - tables and graphs are first-class composable objects
  - specialized operators which exploit view semantics
- **Single system that efficiently spans the pipeline**
  - minimize data movement and duplication
  - eliminates need to learn and manage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

J. Gonzalez et al., "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics" Strata 2014

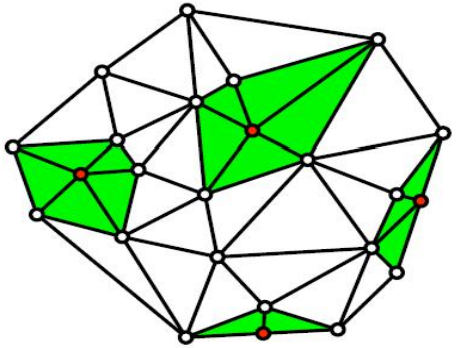
Selected Examples

- GraphX
- Galois

Compute-centric View of Algorithm



Before



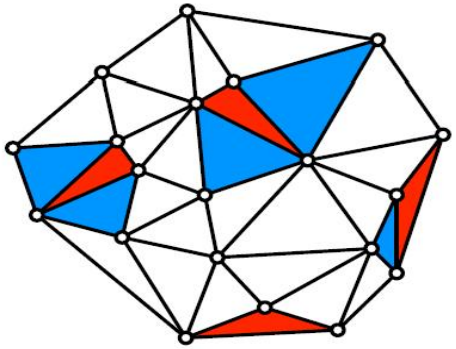
After

Delaunay mesh refinement (DMR)
Red Triangle: badly shaped triangle
Blue triangles: cavity of bad triangle

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(m.badTriangles());
while (true) {
   if (wl.empty()) break;
   Element e = wl.get();
   if (e no longer in mesh)
      continue;
   Cavity c = new
      Cavity();
   c.expand();
   c.retriangulate();
   m.update(c);//update mesh
   wl.add(c.badTriangles());
}
```

Computation-centric

Data-centric View of Algorithm

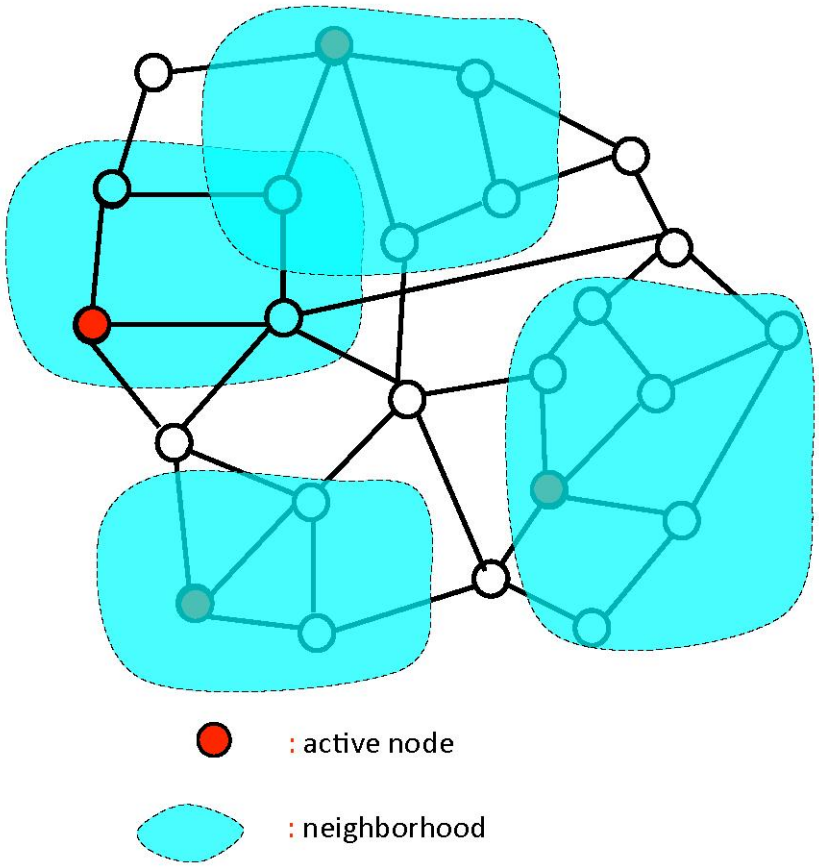


Delaunay mesh refinement (DMR)
Red Triangle: badly shaped triangle
Blue triangles: cavity of bad triangle

- Algorithm
  - composition of unitary actions on data structures
- Actions: operator
  - DMR: {find cavity, retriangulate, update mesh}
- Composition of actions:
  - specified by a schedule
- Parallelism
  - disjoint actions can be performed in parallel
- Parallel data structures
  - graph
  - worklist of bad triangles

Operator formulation of algorithms

- Active element
  - Site where computation is needed
- Operator
  - Computation at active element
  - Activity: application of operator to active element
- Neighborhood
  - Set of nodes/edges read/written by activity
  - Distinct usually from neighbors in graph
- Ordering : scheduling constraints on execution order of activities
  - Unordered algorithms: no semantic constraints but performance may depend on schedule
  - Ordered algorithms: problem-dependent order
- Amorphous data-parallelism
  - Multiple active nodes can be processed in parallel subject to neighborhood and ordering constraints



Parallel program = Operator + Schedule + Parallel data structure

Parallelization strategies: Binding Time

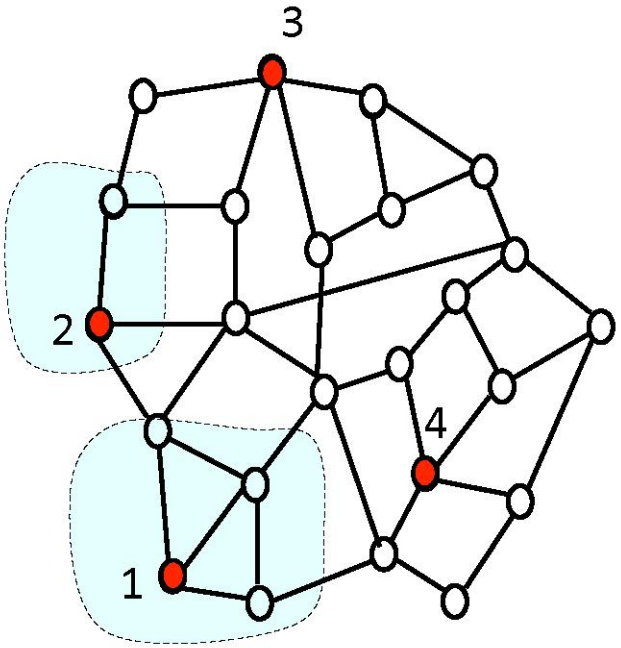
When do you know the active nodes and neighborhoods?

Compile-time — **Static parallelization (stencil codes, FFT, dense linear algebra)**

After input is given — **Inspector-executor (Bellman-Ford)**

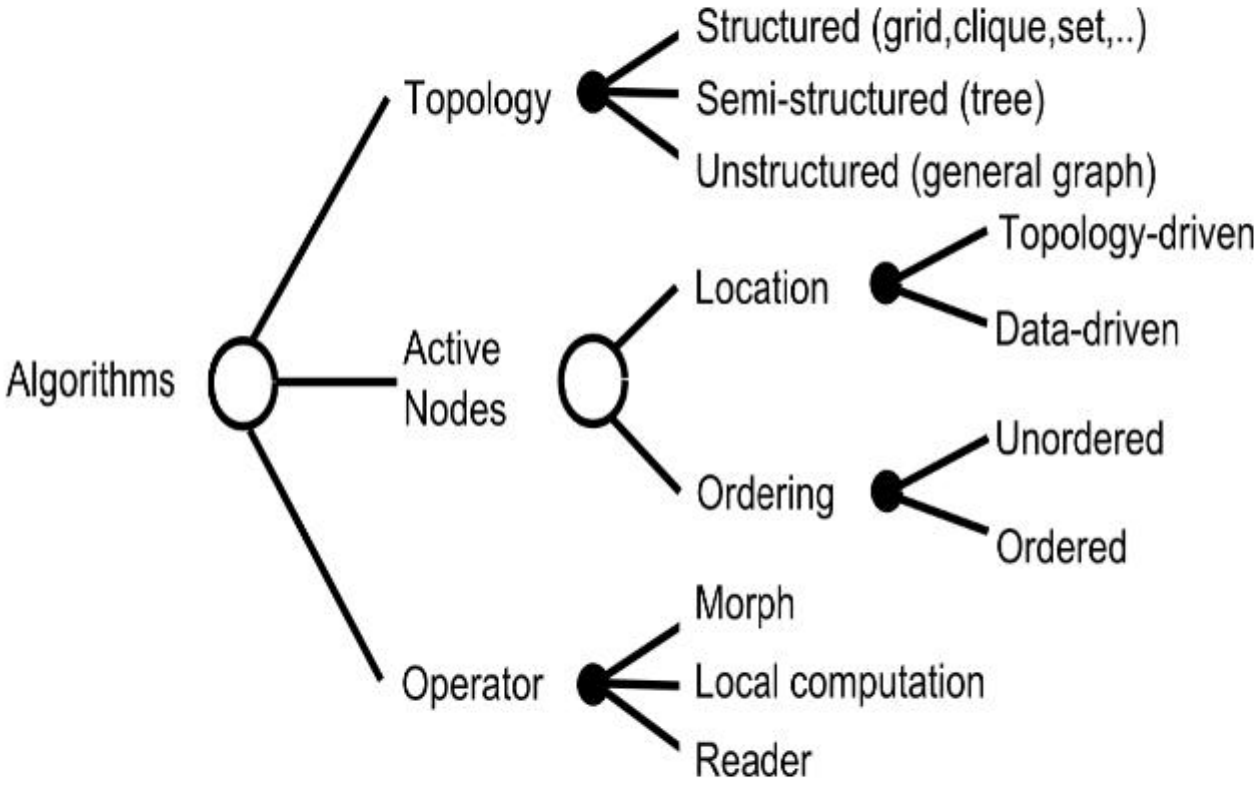
During program execution — **Interference graph (DMR, chaotic SSSP)**

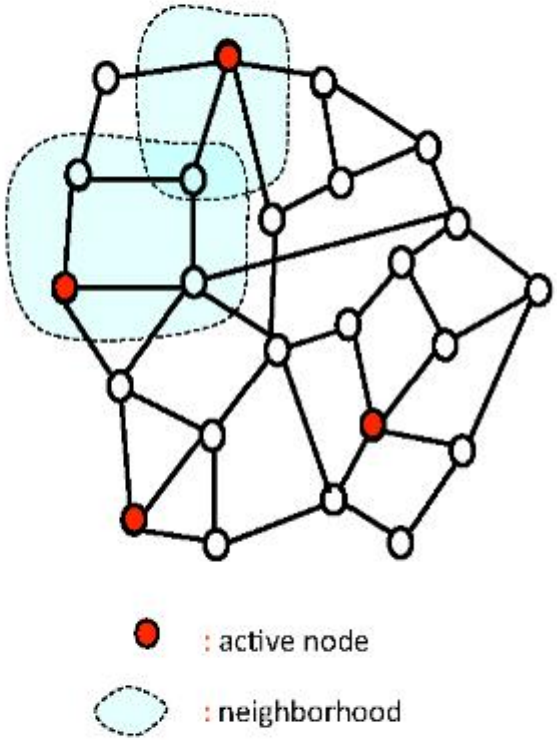
After program is finished — **Optimistic Parallelization (Time-warp)**



"The TAO of parallelism in algorithms" Pingali et al, PLDI 2011

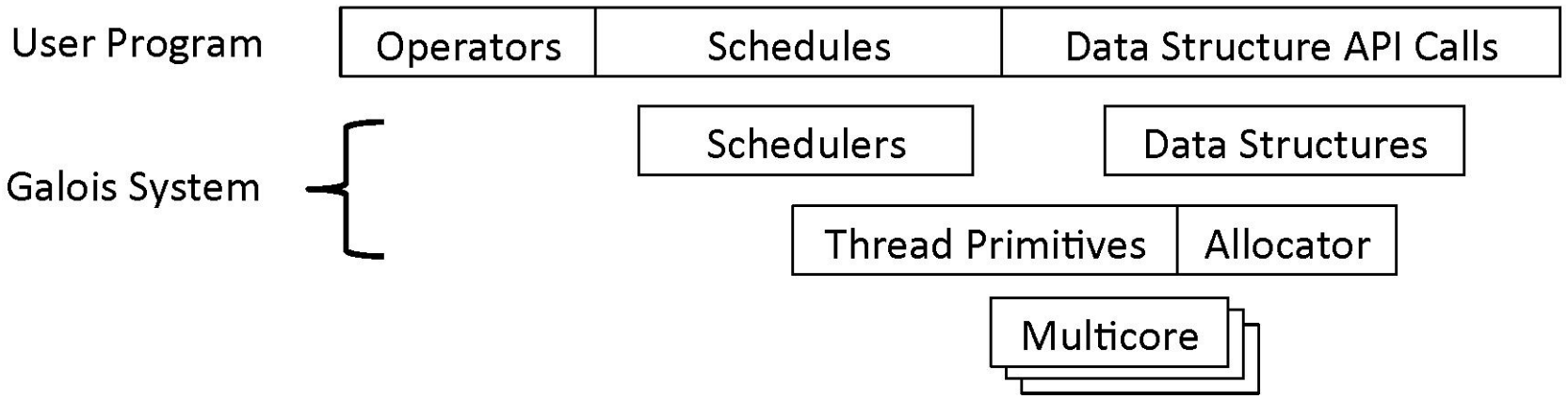
TAO analysis: Structure in algorithms (PLDI 2011)





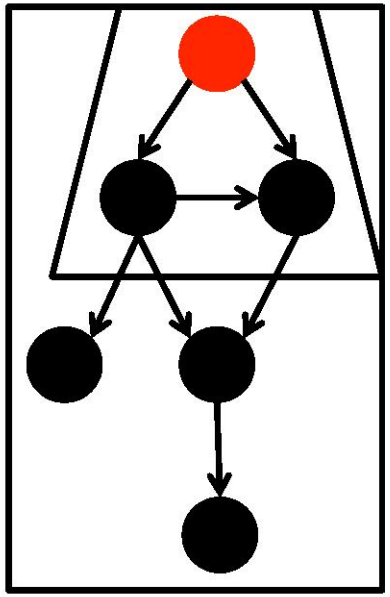
Galois System

Parallel Program = Operator + Schedule + Parallel Data Structure

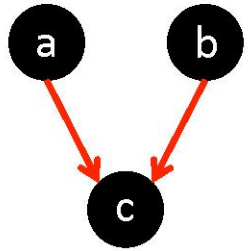


Parallel Program = Operator + Schedule + Parallel Data Structure

Algorithm



- What is the operator?
  - Other graph analytics frameworks: only vertex programs
  - Galois: Unrestricted, may even morph graph by adding/removing nodes and edges
- Where/When does it execute?
  - Autonomous scheduling: activities execute asynchronously and transactionally
  - Coordinated scheduling: activities execute in rounds
    - Read values refer to previous rounds
    - Multiple updates to the same location are resolved with reduction, etc.



“Hello graph” Galois Program

```
#include “Galois/Galois.h”
#include “Galois/Graphs/LCGraph.h”

struct Data { int value; float f; };

typedef Galois::Graph::LC_CSR_Graph<Data,void> Graph;
typedef Galois::Graph::GraphNode Node;

Graph graph;

struct P {
  void operator()(Node n, Galois::UserContext<Node>& ctx) {
    graph.getData(n).value += 1;
  }
};

int main(int argc, char** argv) {
  graph.structureFromGraph(argv[1]);
  Galois::for_each(graph.begin(), graph.end(), P());
  return 0;
}
```

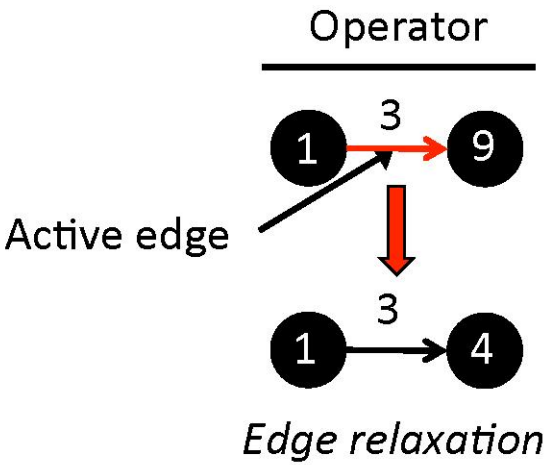
Data structure Declarations

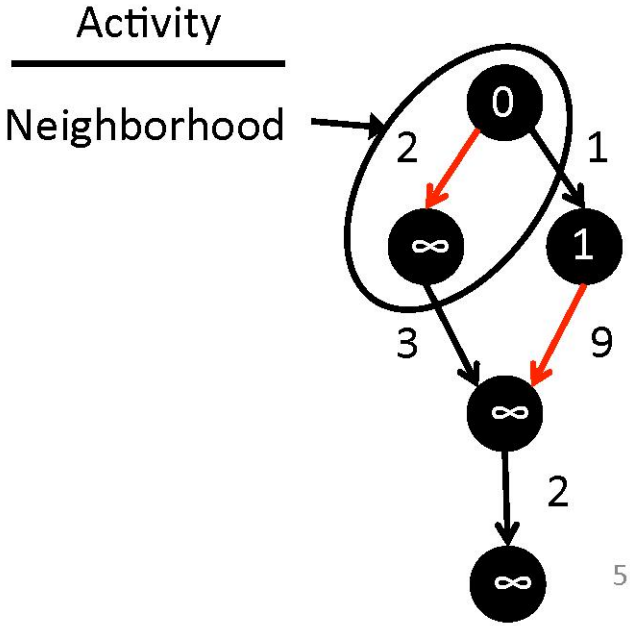
Operator

Galois Iterator

Example: SSSP

- Find the shortest distance from source node to all other nodes in a graph
  - Label nodes with tentative distance
  - Assume non-negative edge weights
- Algorithms
  - Chaotic relaxation $O(2^V)$
  - Bellman-Ford O(VE)
  - Dijkstra's algorithm O(E log V)
    - Uses priority queue
  - Δ-stepping
    - Uses sequence of bags to prioritize work
    - Δ=1, O(E log V)
    - Δ=∞, O(VE)
- Different algorithms are different schedules for applying relaxations
  - SSSP needs priority scheduling for work efficiency





Algorithmic Variants == Scheduling

- Chaotic Relaxation:
  - Specify a non-priority scheduler
    - E.g. dChunkedFIFO
- Dijkstra:
  - Use Ordered Executor
- Delta-Stepping Like:
  - Specify OBIM priority scheduler
- Bellman-Ford
  - Push every edge in non-priority scheduler
  - Execute
  - Repeat #nodes times

Simple (PUSH) SSSP in Galois

```
struct SSSP {
  void operator()(UpdateRequest& req,
          Galois::UserContext<UpdateRequest>& ctx) const {
    unsigned& data = graph.getData(req.second);
    if (req.first > data) return;

    for (Graph::edge_iterator ii
=graph.edge_begin(req.second),
          ee = graph.edge_end(req.second); ii != ee; ++ii)
      relax_edge(data, ii, ctx);
  }
};
```

Relax Edge (PUSH)

```
void relax_edge(unsigned src_data, Graph::edge_iterator ii,
                Galois::UserContext<UpdateRequest>& ctx) {
  GNode dst = graph.getEdgeDst(ii);
  unsigned int edge_data = graph.getEdgeData(ii);
  unsigned& dst_data = graph.getData(dst);
  unsigned int newDist = dst_data + edge_data;
  if (newDist < dst_data) {
    dst_data = newDist;
    ctx.push(std::make_pair(newDist, dst));
  }
}
```

Specifying Schedule and Running

Load

```
Galois::Graph::readGraph(graph, filename);
Galois::for_each(graph.begin(), graph.end(), Init());
```

WorkList

```
using namespace Galois::WorkList;
typedef dChunkedLIFO<16> dChunk;
typedef OrderedByIntegerMetric<UpdateRequestIndexer,dChunk>
OBIM;
```

SSSP

```
graph.getData(*graph.begin()) = 0;
Galois::for_each(std::make_pair(0U, *graph.begin()), SSSP(),
                        Galois::wl<OBIM>());
```

Implementation Variants: Push V.S. Pull

- Simple optimization to control concurrency costs, locks, etc.
- Push: Look at node and update neighbors
- Pull: Look at neighbors and update self
- Pull seems “obviously” better, but in practice it depends on algorithm, scheduling, and data

Pull SSSP

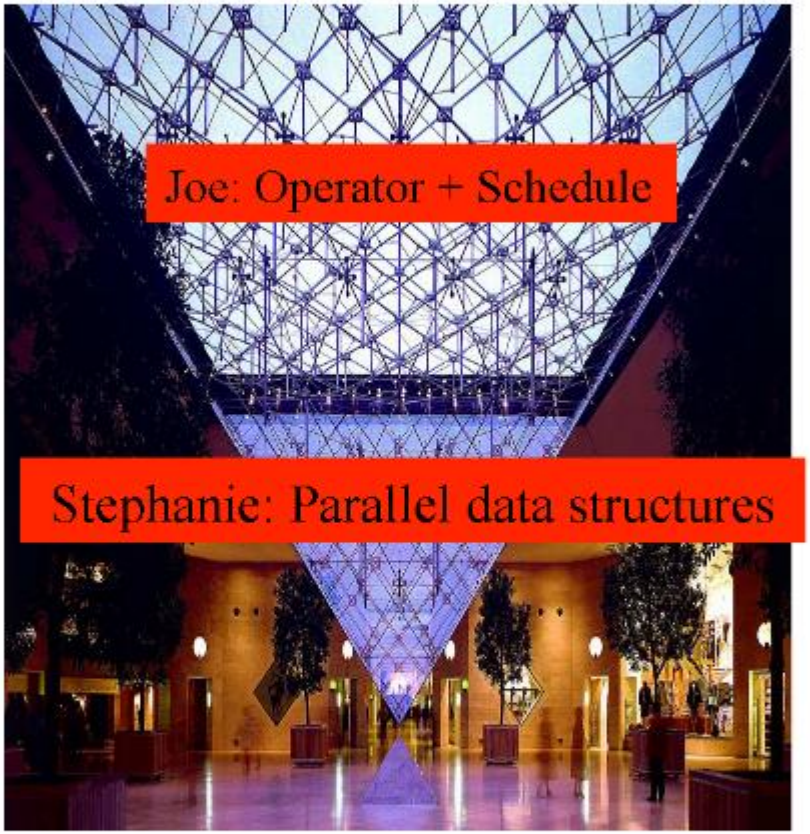
```
struct SSSP {
  void operator()(GNode req, Galois::UserContext<UpdateRequest>& ctx) {
//update self
   for (auto ii = graph.edge_begin(req),  ee = graph.edge_end(req); ii != ee; ++ii) {
     auto edist = graph.getEdgeData(ii), ndist = graph.getData(graph.getEdgeDst(ii));
     if (edist + ndist < data)
       data  = edist + ndist;
   }
//push higher neighbors
 for (auto ii = graph.edge_begin(req),  ee = graph.edge_end(req); ii != ee; ++ii) {
     auto edist = graph.getEdgeData(ii), ndist = graph.getData(graph.getEdgeDst(ii));
     if (ndist  > data + edist)
        ctx.push(graph.getEdgeDst(ii));
   }
};
```

SSSP Demo

- Start with chaotic algorithm and vary scheduling policy
  - Different policies give different amounts of work and scalability but all policies produce correct executions
- Policies
  - FIFO
  - ChunkedFIFO
    - FIFO of fixed size chunks of items
  - dChunkedFIFO
    - A ChunkedFIFO per package with stealing between ChunkedFIFOs
  - OBIM
    - Generalization of sequence of bags when sequence is sparse

Conclusions

- Yesterday:
  - Computation-centric view of parallelism
- Today:
  - Data-centric view of parallelism
  - Operator formulation of algorithms
  - Permits a unified view of parallelism and locality in algorithms
  - Joe/Stephanie programming model
  - Galois system is an implementation
- Tomorrow:
  - DSLs for different applications
  - Layer on top of Galois



Parallel program = Operator + Schedule + Parallel data structure

Overall Summary

- **Distributed machines**
  - Pregel, GraphLab, PowerGraph, GraphX, etc.
- **Single machine**
  - Ligra, GraphChi, X-Stream, Galois, etc.
- **GPU oriented**
  - Single machine: Garaph, Tigr, CuSha, MapGraph, etc.
  - Distributed: Lux, etc.