Programming Models and Tools for Distributed Graph Processing

Vasia Kalavri
kalavriv@inf.ethz.ch

31st British International Conference on Databases
10 July 2017, London, UK

ABOUT ME

- Postdoctoral Fellow at ETH Zürich
  - Systems Group: https://www.systems.ethz.ch/
- PMC member of Apache Flink
- Research interests
  - Large-scale graph processing
  - Streaming dataflow engines
- Current project:
  - Predictive datacenter analytics and management
  - Strymon: http://strymon.systems.ethz.ch/

TUTORIAL OUTLINE

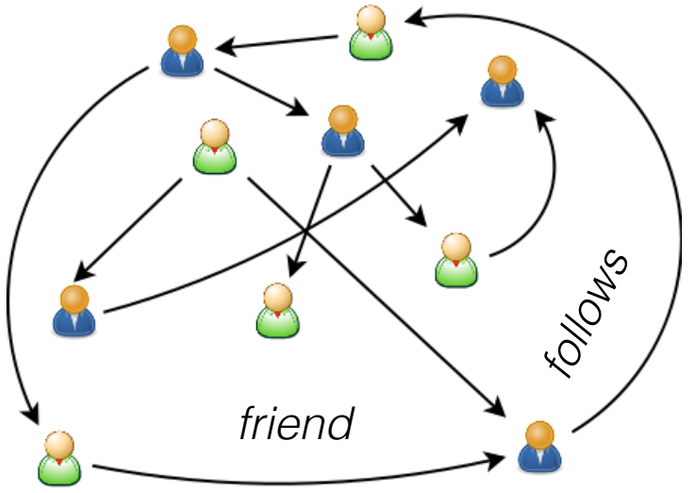
- Distributed Graph Processing (DGP)
  - when do we need distribution?
  - misconceptions and truths
- Specialized Models for DGP
  - execution semantics
  - user interfaces
  - performance issues
- General-Purpose Models for DGP
- Recap

THIS TUTORIAL IS NOT ABOUT

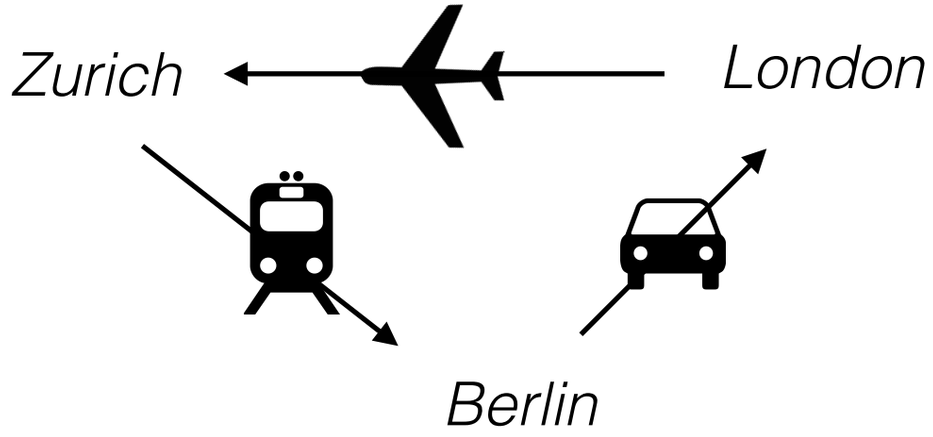
- Graph databases
- RDF stores
- Single-node systems
- Shared-memory systems
- Performance comparison of tools

Kalavri, Vasiliki, Vladimir Vlassov, and Seif Haridi.
"High-Level Programming Abstractions for Distributed Graph Processing."
arXiv preprint arXiv:1607.02646 (2016).

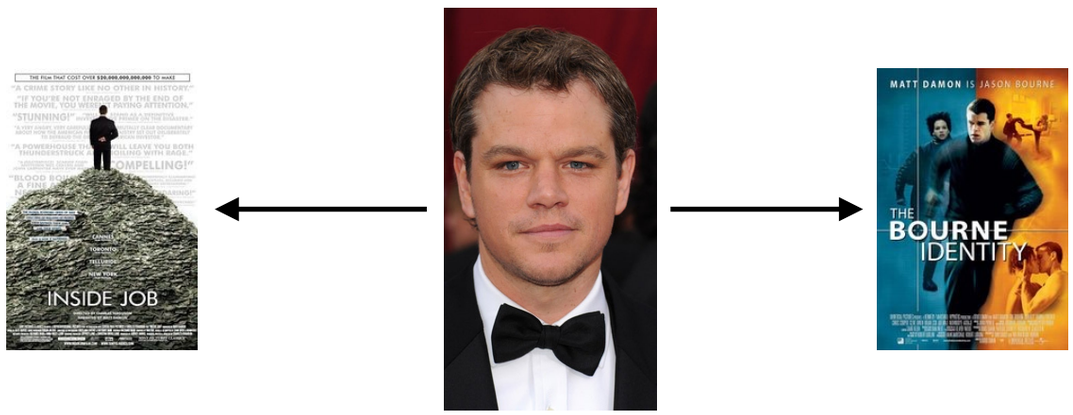
MODELING THE WORLD AS A GRAPH



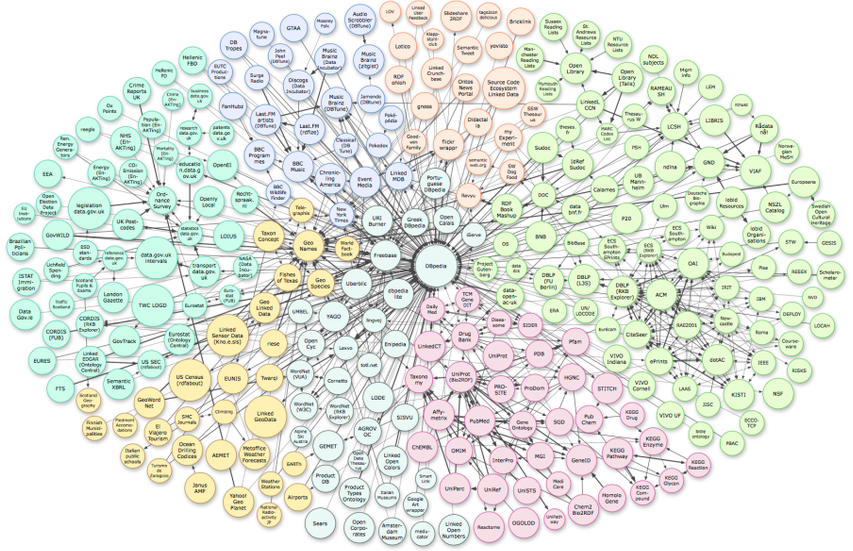
Social networks



Transportation networks

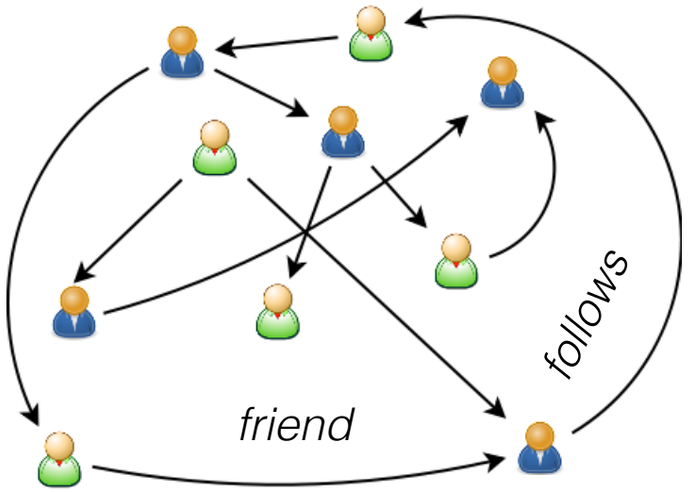


Actor-movie networks



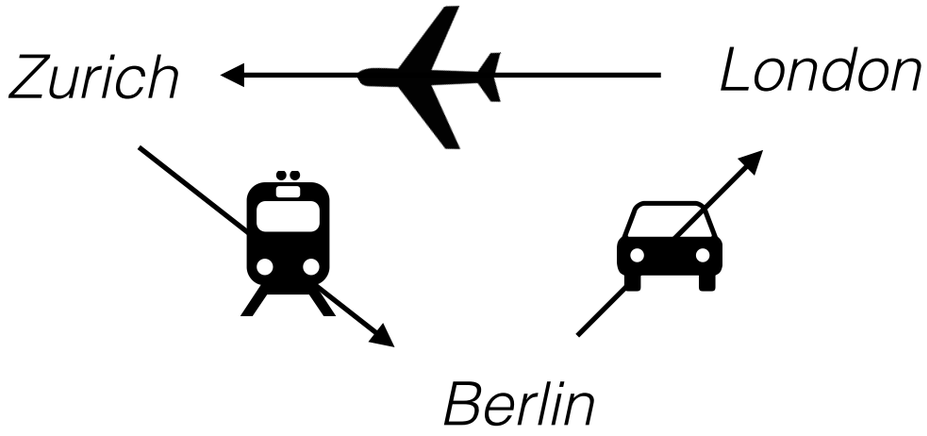
The web

ANALYZING GRAPHS

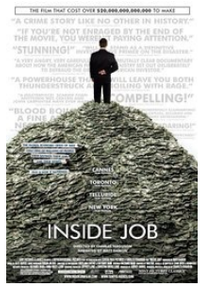


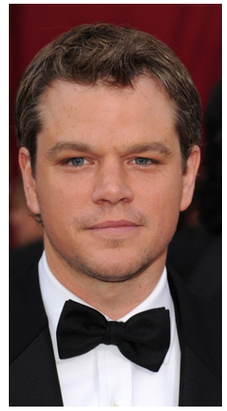
"conservative"

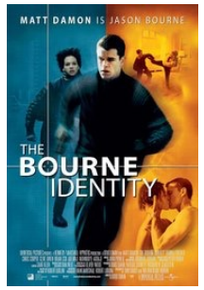
"liberal"



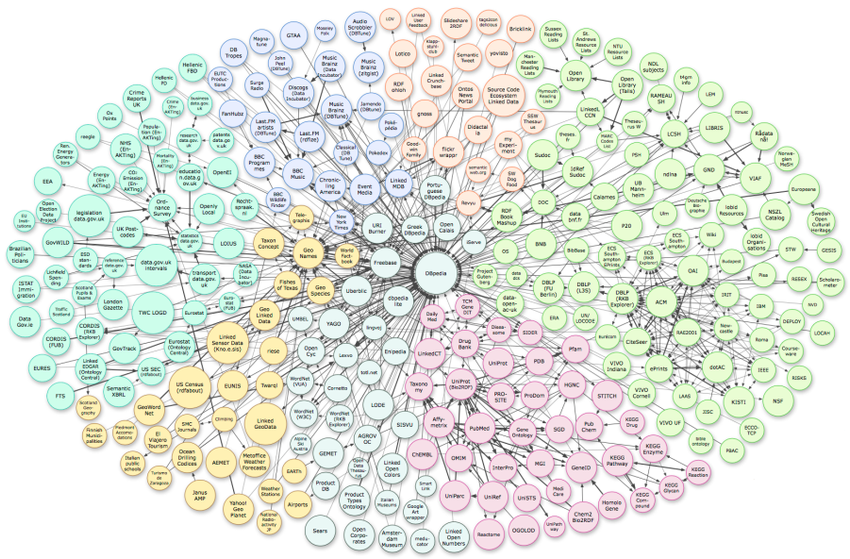
What's the cheapest way to reach Zurich from London through Berlin?





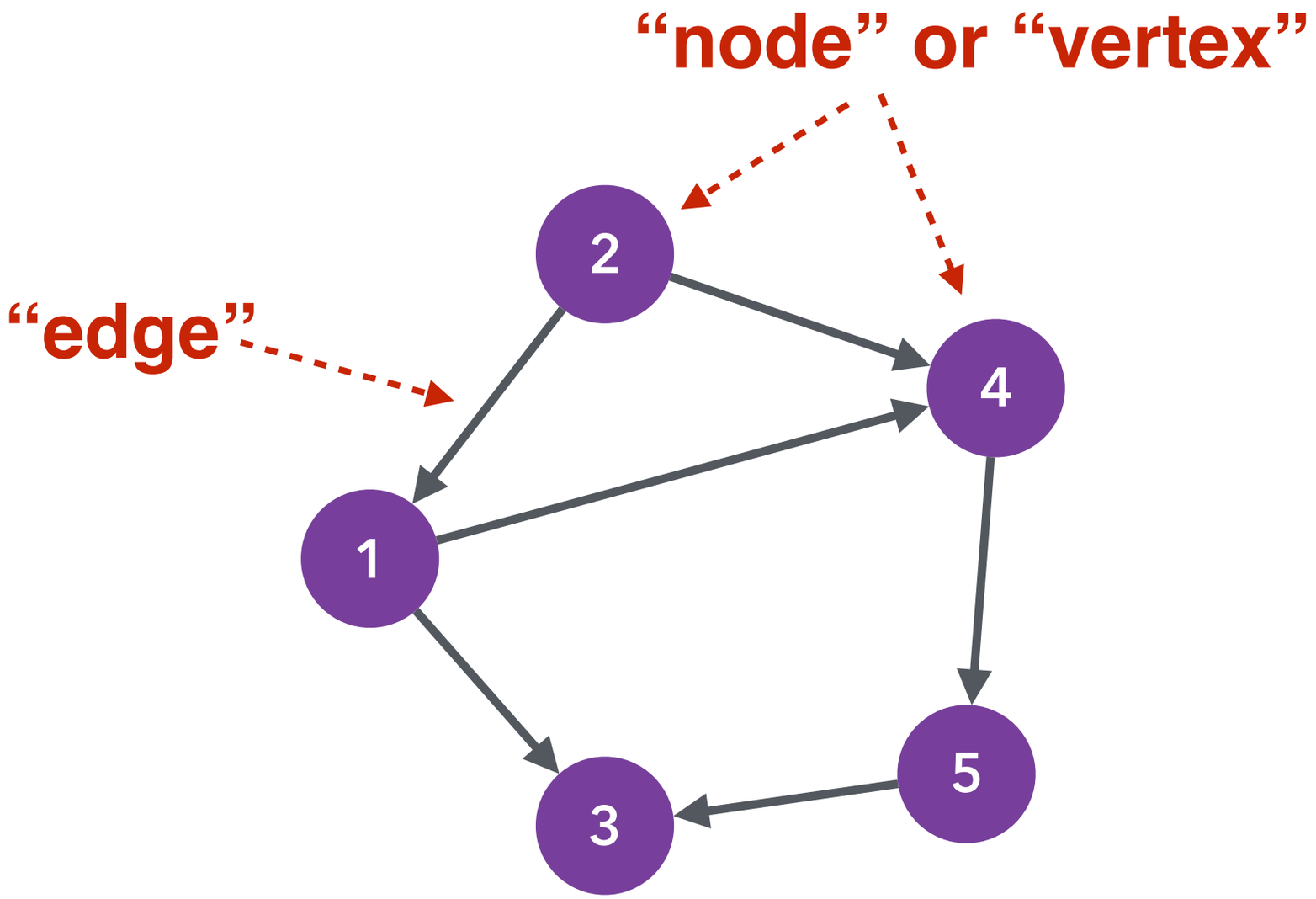


If you like "Inside job" you might also like "The Bourne Identity"

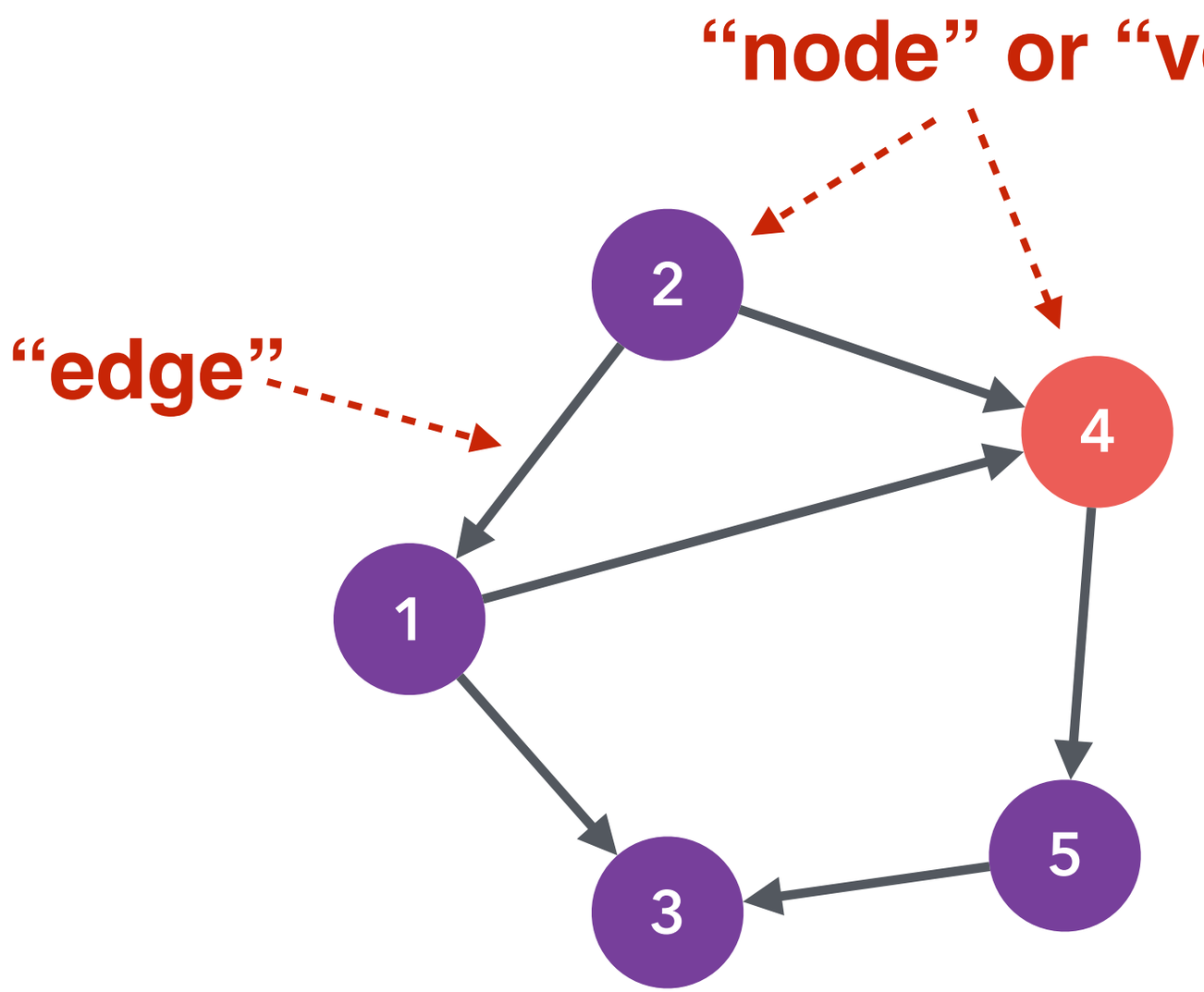


These are the top-10 relevant results for the search term "graph"

BASICS



BASICS



5 is *out-neighbor*

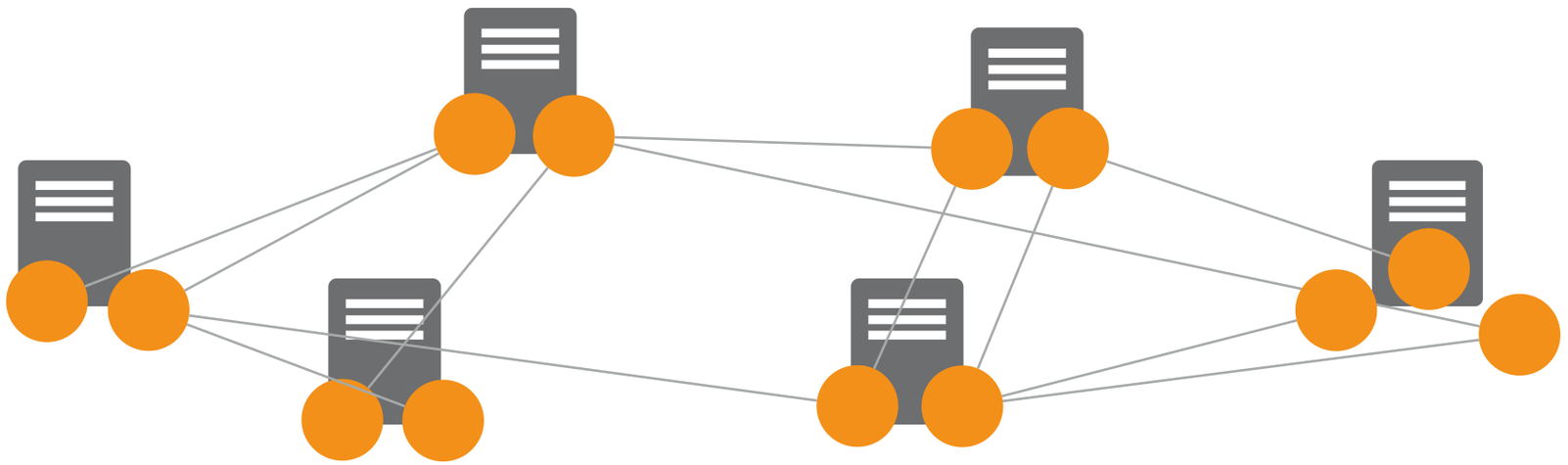
1, 2 are *in*-neighbors

3 is a *2-hop* neighbor

4 has *in-degree* = 2
and *out-degree* = 1

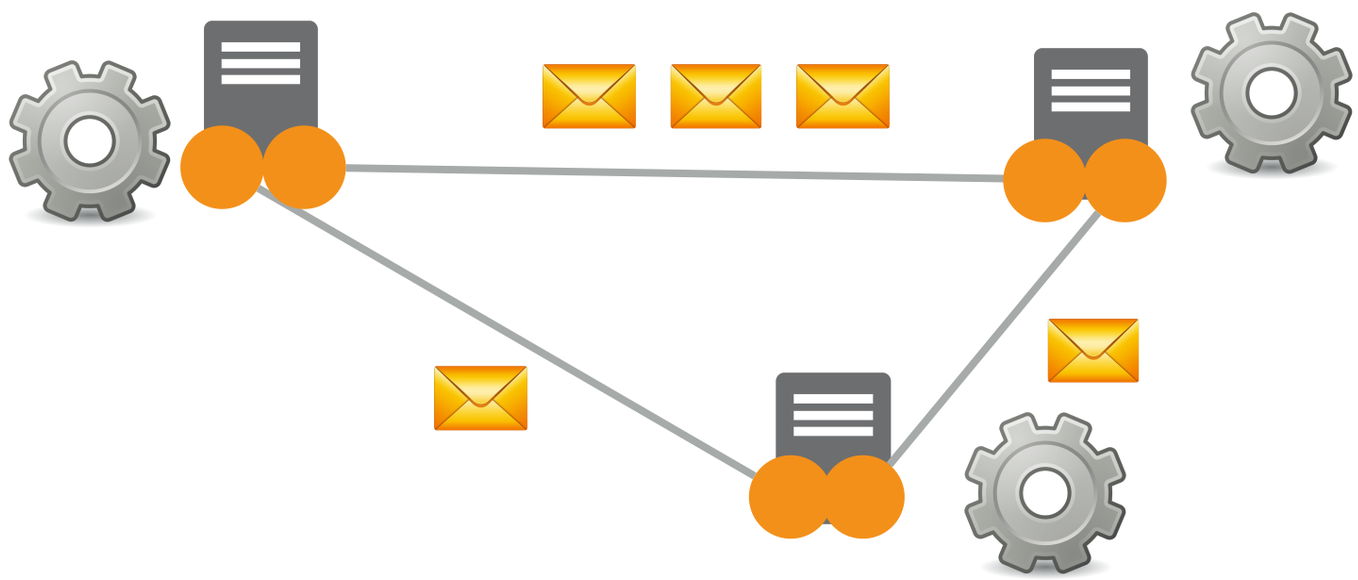
DISTRIBUTED GRAPH PROCESSING

- Shared-nothing memory model
- Distributed algorithms for analysis
- Graph partitioning



GRAPH PARTITIONING

- Communication usually “flows” along edges
- Minimize communication while balancing the computation load
- Many graphs have skewed degree distributions



WHEN DO YOU NEED DISTRIBUTED GRAPH PROCESSING?

MY GRAPH IS SO BIG, IT DOESN'T FIT IN A SINGLE MACHINE

Big Data Ninja

A SOCIAL NETWORK

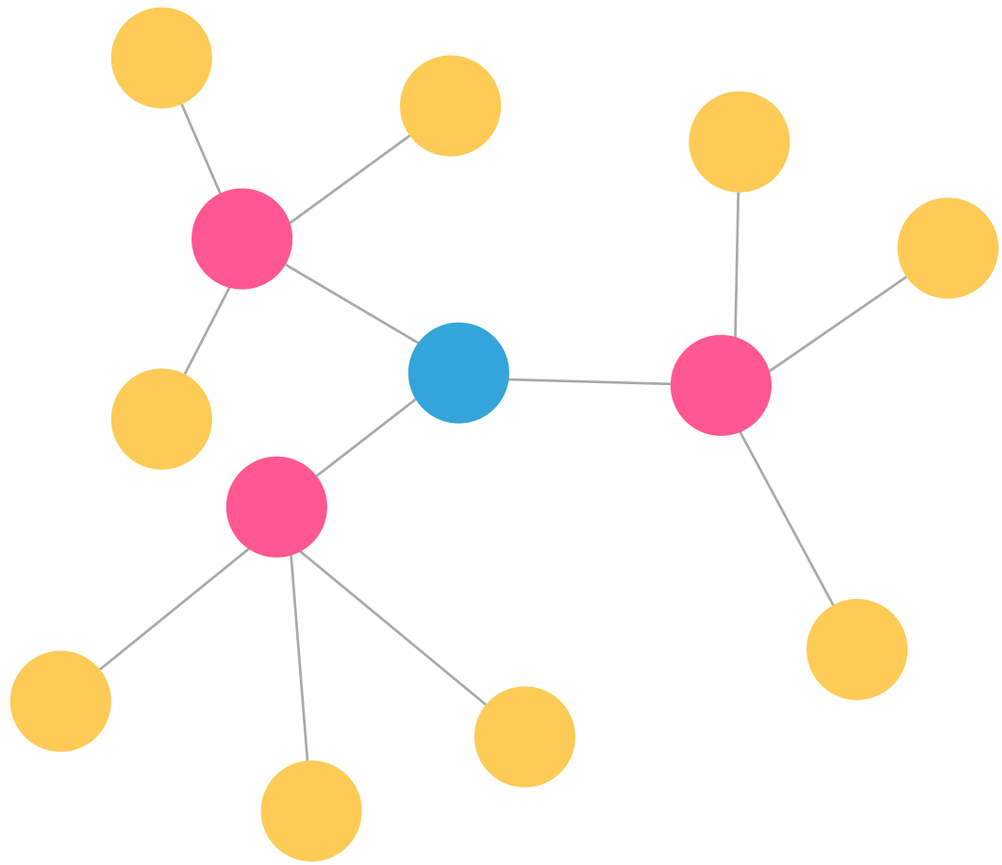
WTF: The Who to Follow Service at Twitter

Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, Reza Zadeh
Twitter, Inc.
@pankaj @ashishgoel @lintool @aneeshs @dongwang218 @reza_zadeh

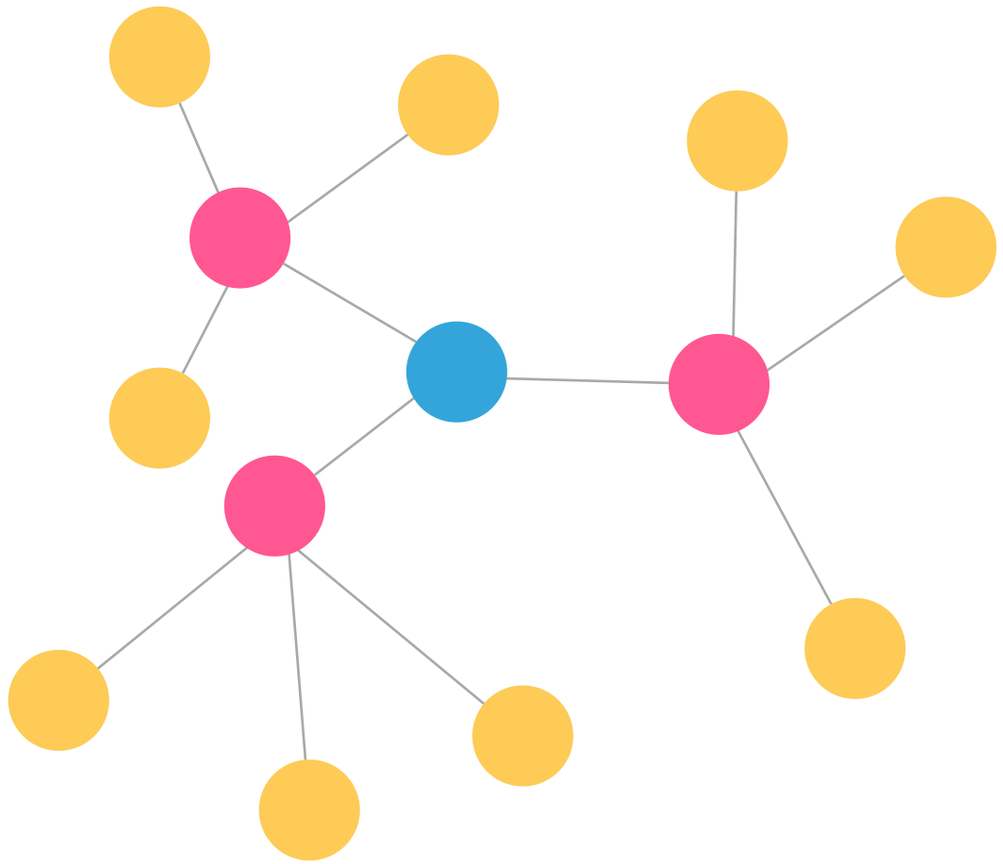
2 billion users, one could storing each vertex id as a 32-bit (signed) integer, in which case each edge would require eight bytes. On a machine with 72 GB memory, we could reasonably expect to handle graphs with approximately eight billion edges: 64 GB to hold the entire graph in memory, and 8 GB for the operating system, the graph processing engine, and memory for actually executing graph algorithms. Of course, storing the source and destination vertices of each edge in the manner described above is quite wasteful; the

NAIVE WHO(M)-TO-FOLLOW

- Naive Who(m) to Follow:
  - compute a friends-of-friends list per user
  - exclude existing friends
  - rank by common connections



DON'T JUST CONSIDER YOUR INPUT GRAPH SIZE. INTERMEDIATE DATA MATTERS TOO!



DISTRIBUTED PROCESSING IS ALWAYS FASTER THAN SINGLE-NODE

Data Science Rockstar

Scalability! But at what COST?

Frank McSherry
Unaffiliated

Michael Isard
Unaffiliated*

Derek G. Murray
Unaffiliated†

scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s

Scalability! But at what COST?

Frank McSherry
Unaffiliated

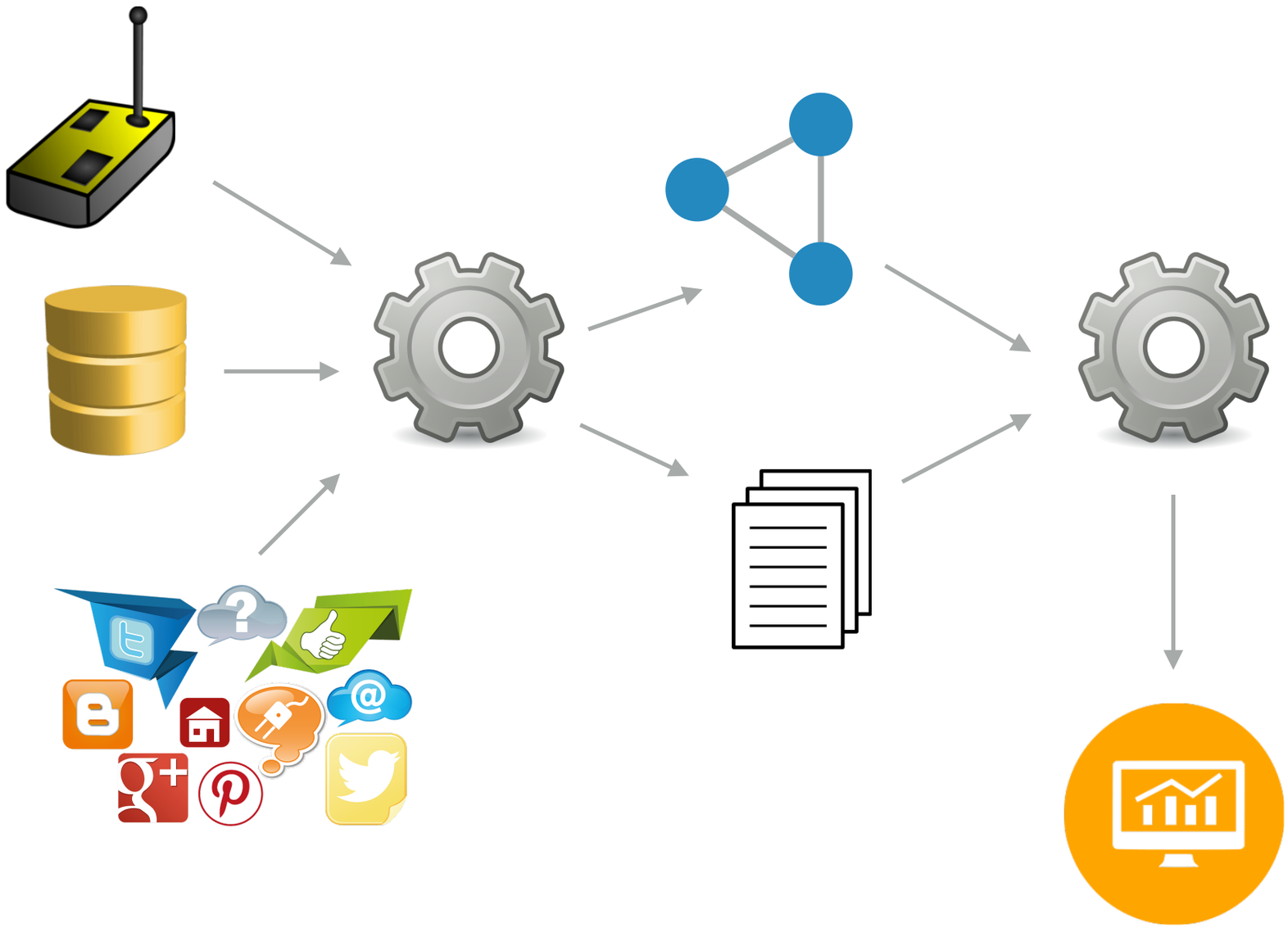
Michael Isard
Unaffiliated*

Derek G. Murray
Unaffiliated†

scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s
Single thread (SSD)	1	153s	417s

GRAPHS DON'T APPEAR OUT OF THIN AIR

GRAPHS DON'T APPEAR OUT OF THIN AIR



WHEN DO YOU NEED DISTRIBUTED GRAPH PROCESSING?

- When you do have **really big** graphs
- When the **intermediate data** of your computation can be very large
- When your data is **already distributed**
- When you have a **distributed data pipeline**

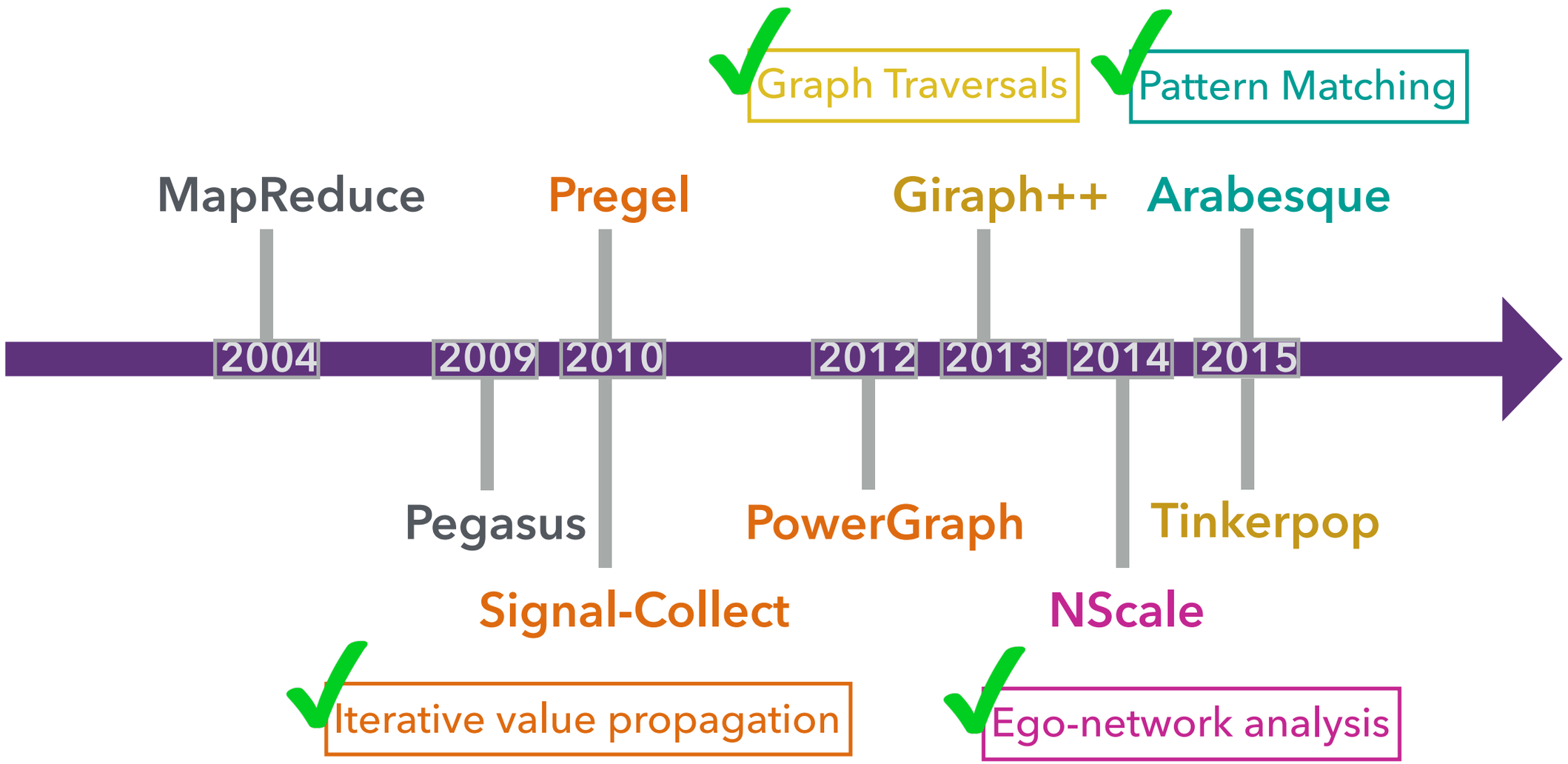
HOW DO WE EXPRESS A DISTRIBUTED GRAPH ANALYSIS TASK?

GRAPH APPLICATIONS ARE DIVERSE

- Iterative value propagation
  - PageRank, Connected Components, Label Propagation
- Traversals and path exploration
  - Shortest paths, centrality measures
- Ego-network analysis
  - Personalized recommendations
- Pattern mining
  - Finding frequent subgraphs

SPECIALIZED PROGRAMMING MODELS FOR DISTRIBUTED GRAPH PROCESSING

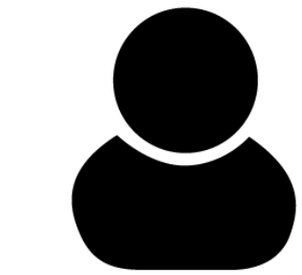
RECENT DISTRIBUTED GRAPH PROCESSING HISTORY



HIGH-LEVEL PROGRAMMING MODELS

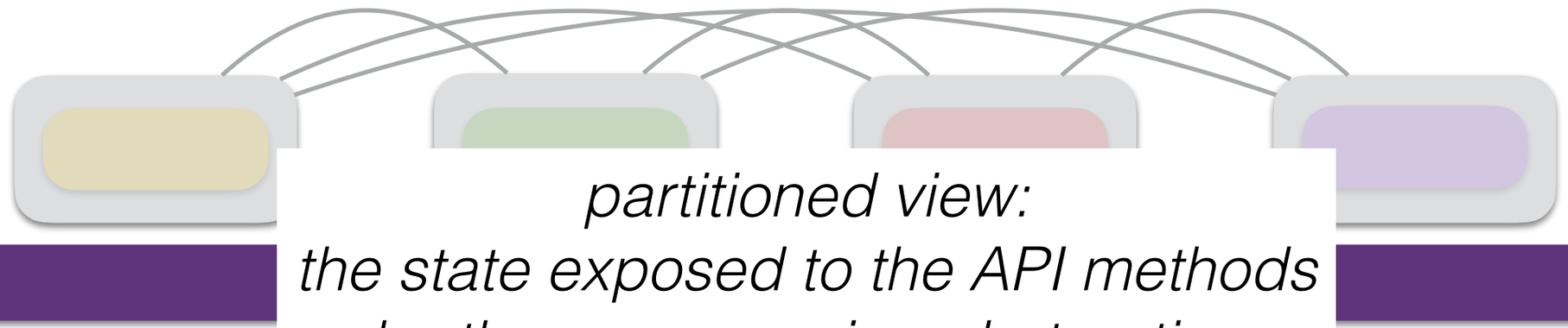
- Hide distribution complexity behind an *abstraction*
  - data partitioning
  - data representation
  - communication mechanisms
- Programmers can focus on the *logic* of their application
  - logical view of graph data
  - a set of methods to read, write, and communicate across views

programming view



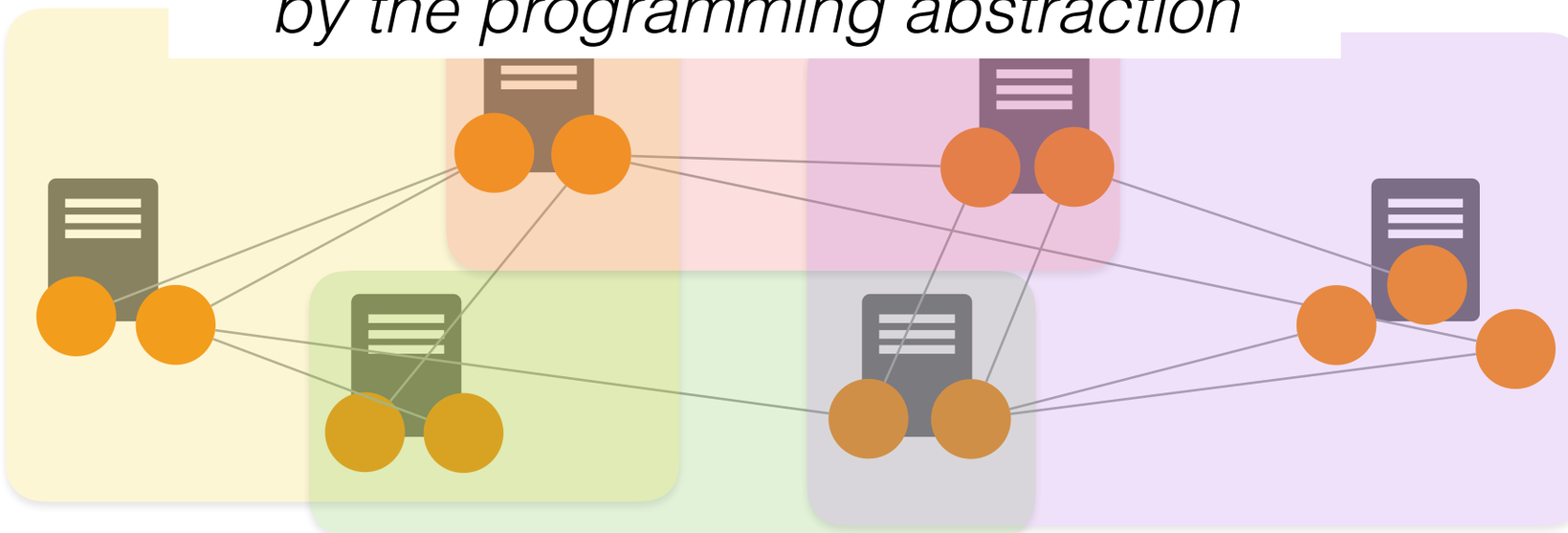
API

read
modify
communicate



partitioned view:
the state exposed to the API methods
by the programming abstraction

physical representation

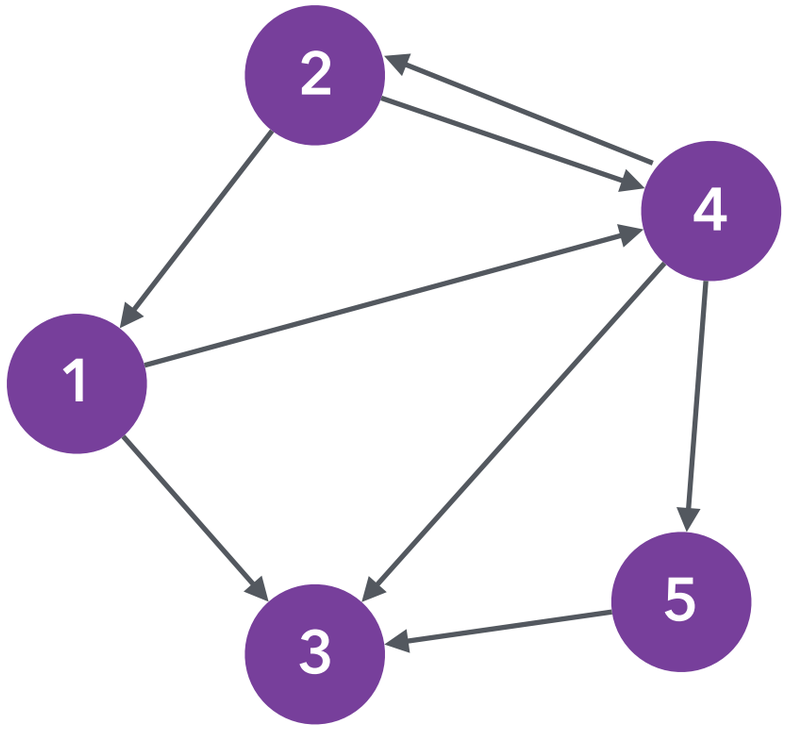


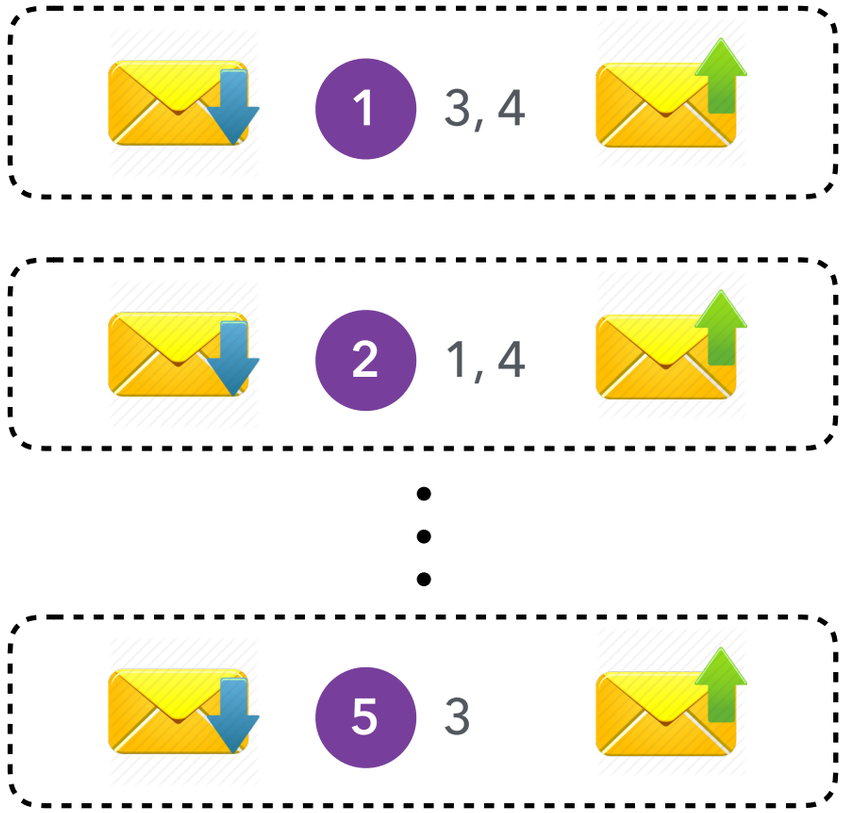
VERTEX-CENTRIC: THINK LIKE A VERTEX

- Express the computation from the view of a single vertex
- Vertices communicate through messages

Malewicz, Grzegorz, et al.
Pregel: a system for large-scale graph processing.
ACM SIGMOD, 2010.

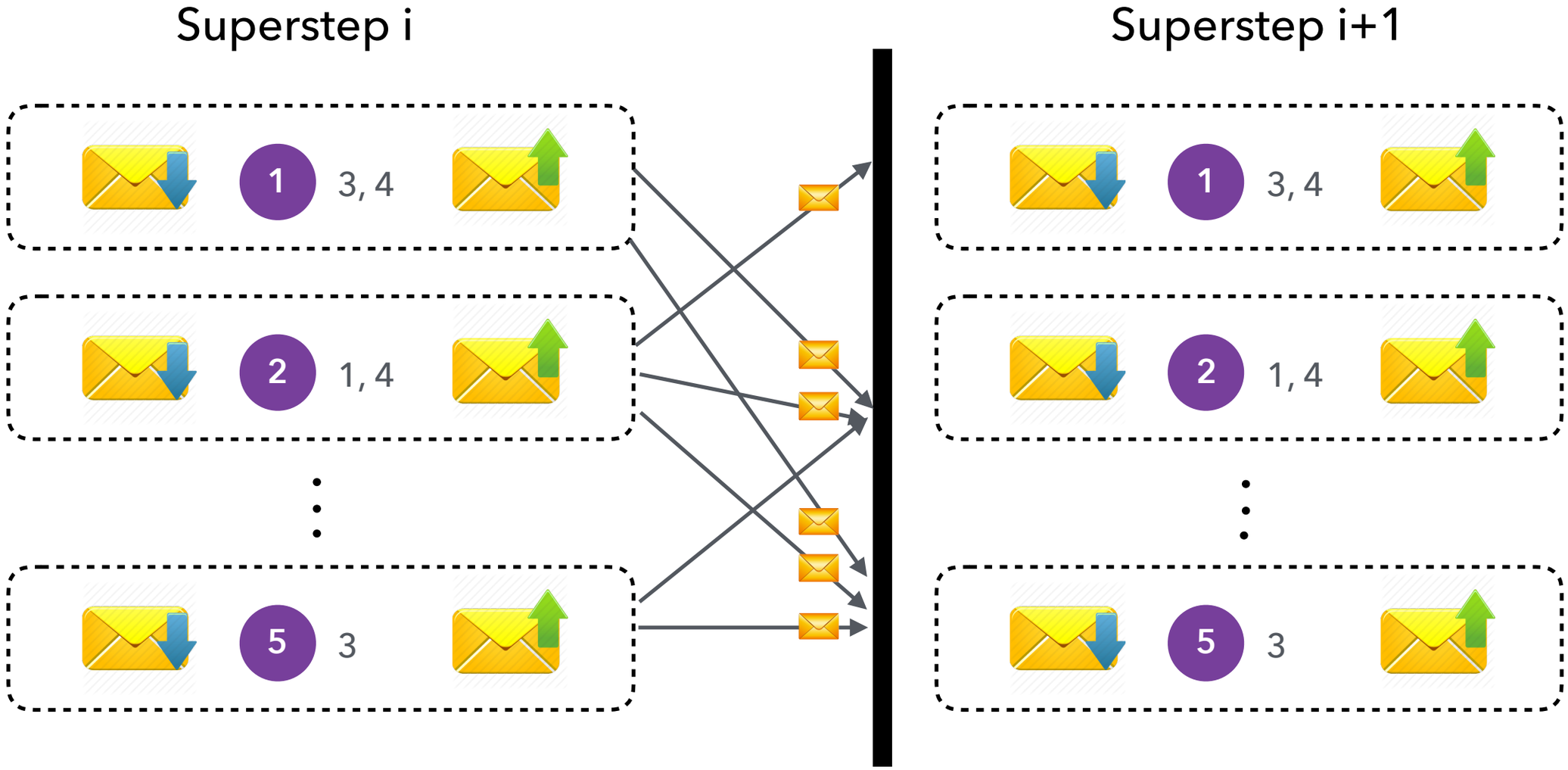
VERTEX-CENTRIC VIEW





The partitioned view consists of a vertex, its out-neighbors, an inbox, and an outbox

VC SUPERSTEPS



$(V_{i+1}, \text{outbox}) \leftarrow \text{compute}(V_i, \text{inbox})$

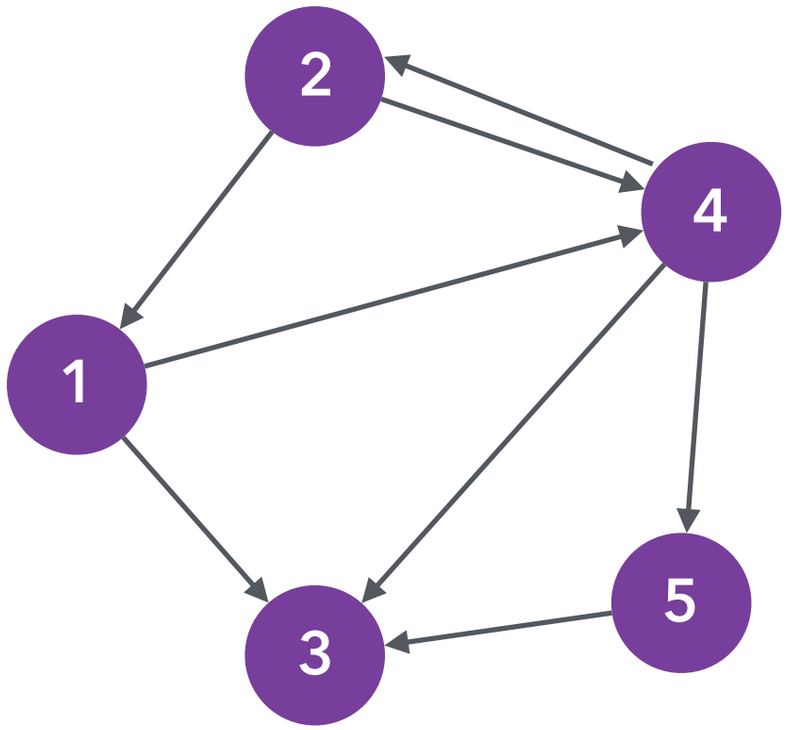
VC SEMANTICS

```
Input: directed graph G=(V,E)
activeVertices ← V
superstep ← 0

while activeVertices != ∅ do
   for v ∈ activeVertices do
      inbox_v ← receiveMessages(v)
      outbox_v = compute(inbox_v)
   end for
   superstep ← superstep + 1
end while
```

VC INTERFACE

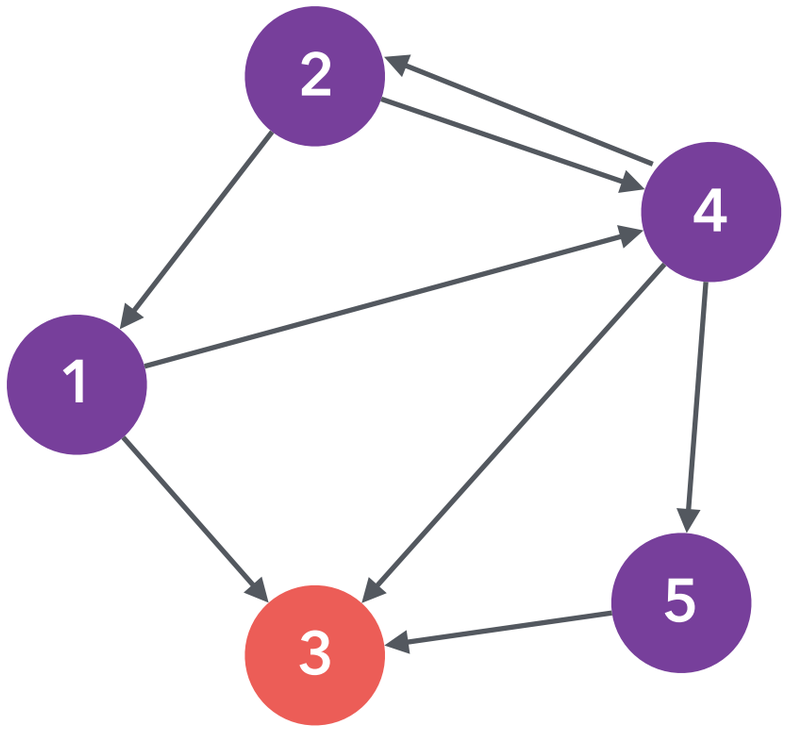
```
void compute(Iterator[M] messages);
VV getValue();
void setValue(VV newValue);
void sendMessageTo(I target, M message);
Iterator getOutEdges();
int superstep();
void voteToHalt();
```

PAGERANK: THE WORD COUNT OF GRAPH PROCESSING



VertexID	Out-degree	Transition Probability
1	2	1/2
2	2	1/2
3	0	-
4	3	1/3
5	1	1

PAGERANK: THE WORD COUNT OF GRAPH PROCESSING



VertexID	Out-degree	Transition Probability
1	2	1/2
2	2	1/2
3	0	-
4	3	1/3
5	1	1

$$PR(3) = 0.5*PR(1) + 0.33*PR(4) + PR(5)$$

VERTEX-CENTRIC PAGERANK

```
void compute(messages):
   sum = 0.0

   for (m <- messages) do
      sum = sum + m
   end for

   setValue(0.15/numVertices + 0.85*sum)

   for (edge <- getOutEdges()) do
      sendMessageTo(
          edge.target(), getValue()/numEdges)
   end for
```

sum up received messages

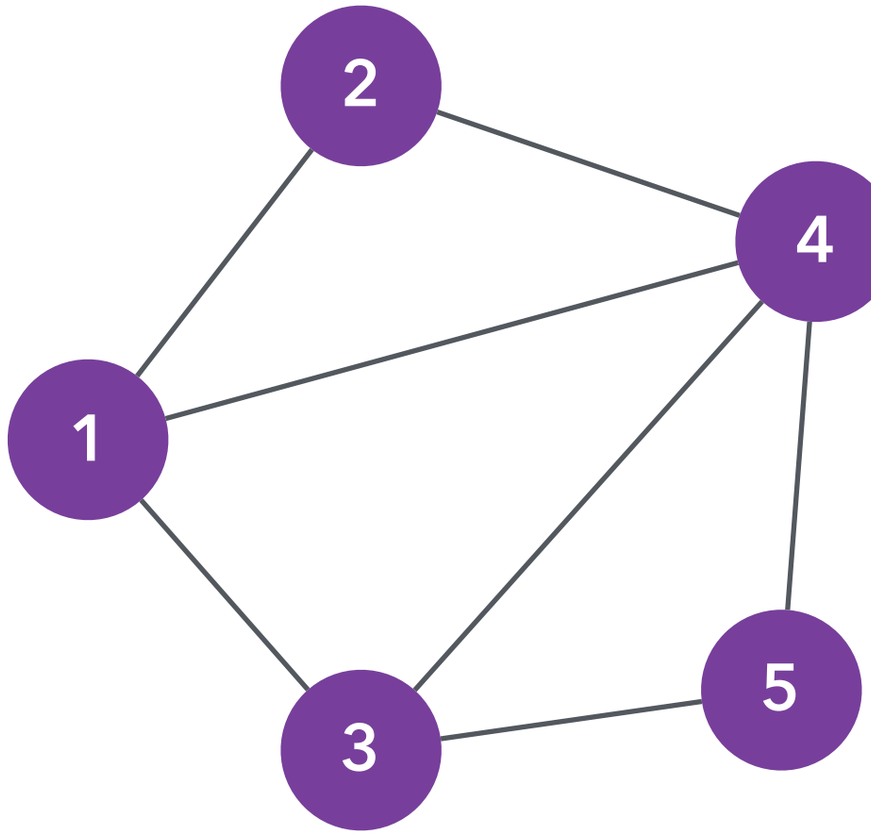
update vertex rank

distribute rank to neighbors

VC ANTI-PATTERNS

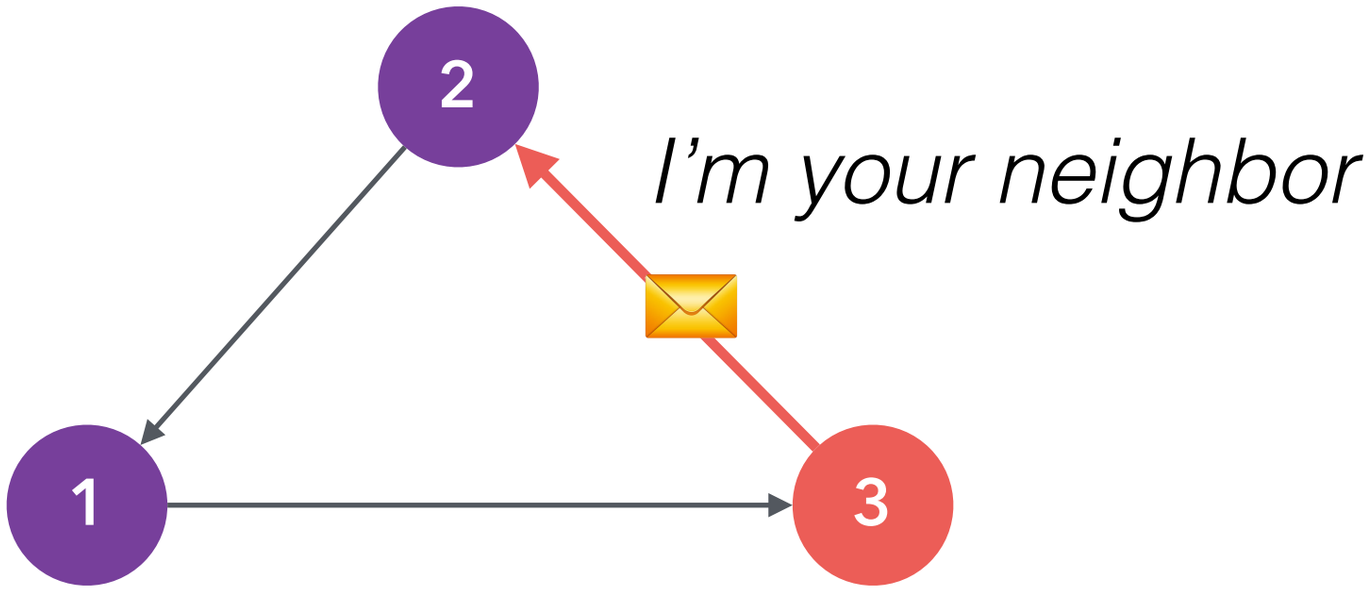
- Non-iterative algorithms
  - Superstep execution
- Non-local state access
  - Propagate a message in 2 supersteps to access 2-hop neighborhood
- Communication with in-neighbors
  - Insert opposite-direction edges to regard in-neighbors as out-neighbors

TRIANGLE COUNTING



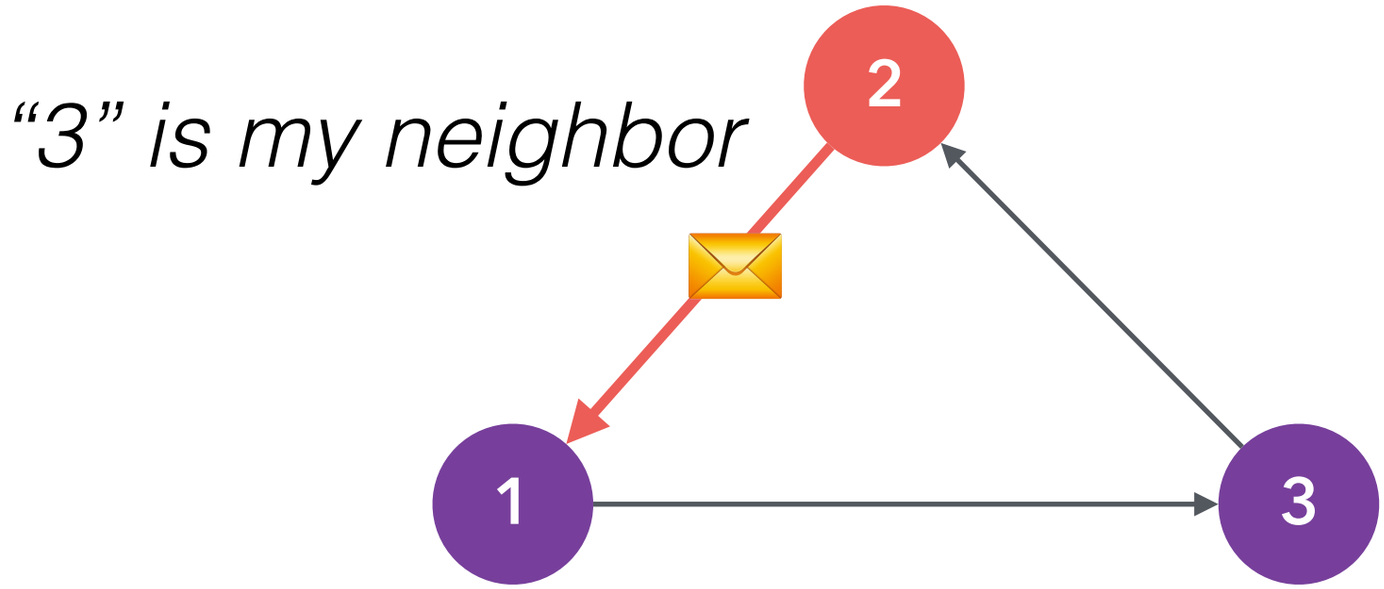
- A vertex needs to know whether there is an edge between its neighbors
- It has to detect this through messages
- It takes 3 supersteps to propagate a message along the triangle's edges

TRIANGLE COUNTING



superstep #1

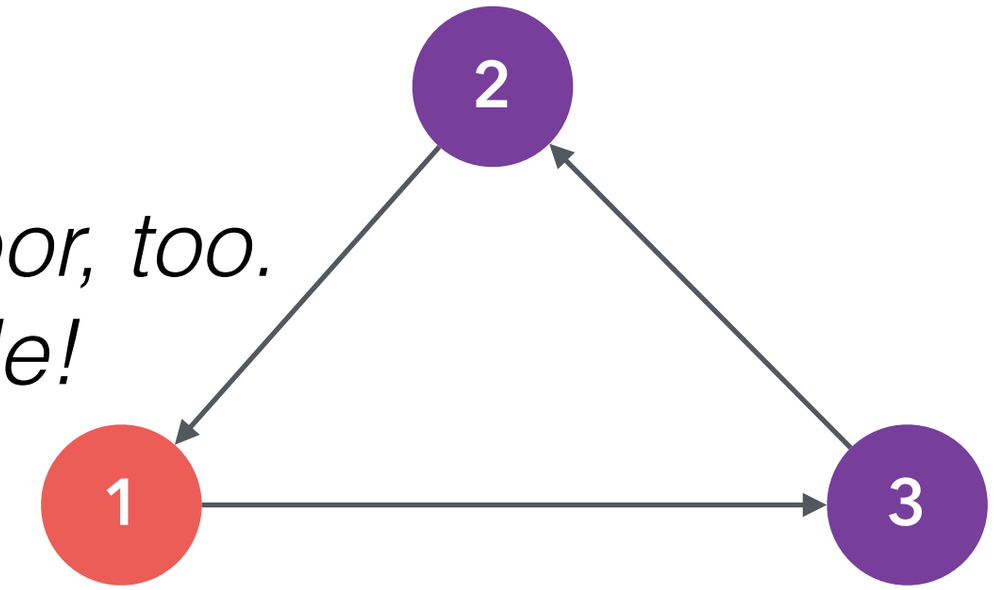
TRIANGLE COUNTING



superstep #2

TRIANGLE COUNTING

“3” is my neighbor, too.
It’s a triangle!



superstep #3

PERFORMANCE ISSUES

- Skewed degree distribution
  - high communication load
  - high memory requirements

smart partitioning
copy high-degree vertices
split supersteps into several sub-supersteps

- Synchronization

suport asynchronous and semi-synchronous execution

- Asymmetrical convergence

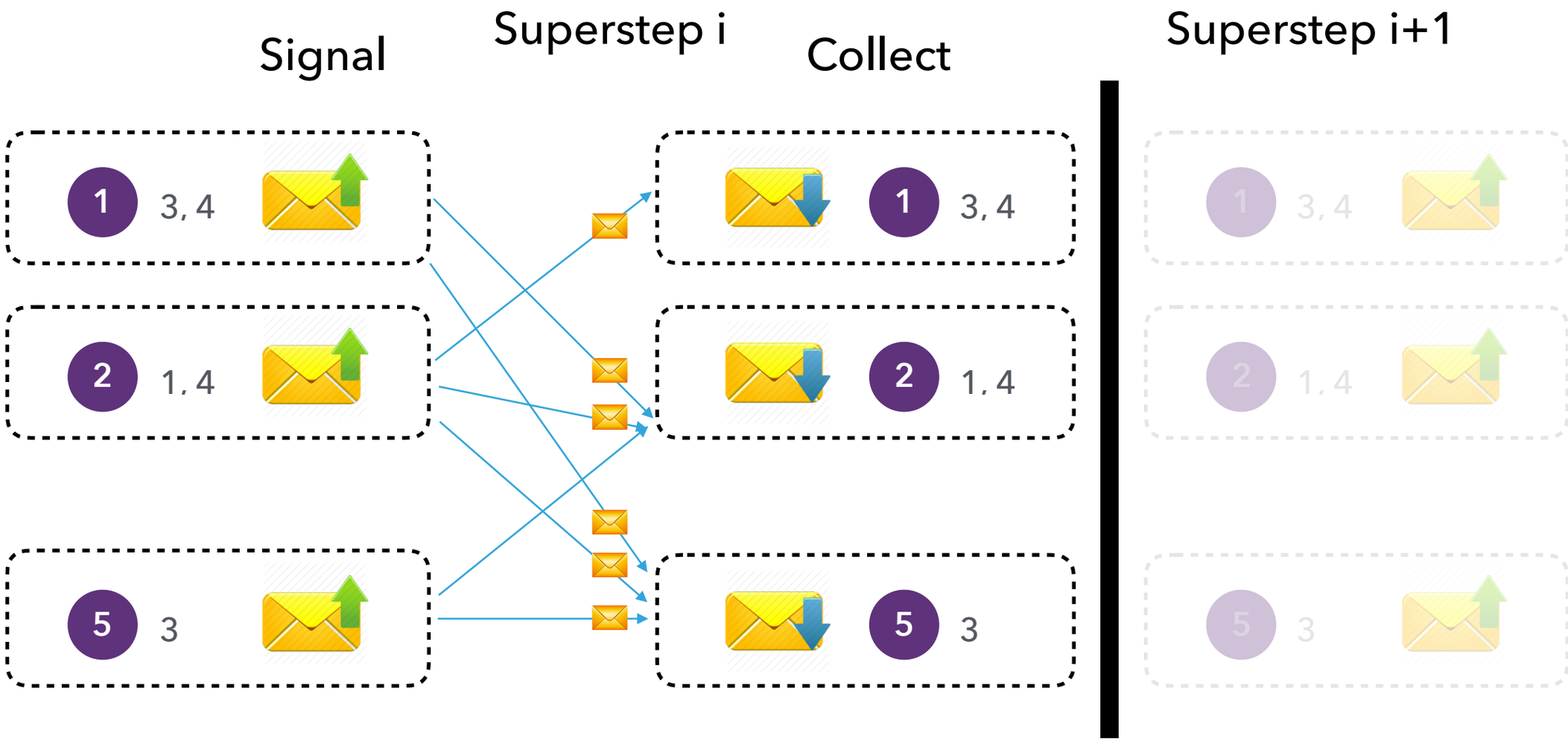
monitor the “active” portion of the graph

SIGNAL-COLLECT (SCATTER-GATHER)

- Express the computation from the view of a single vertex
- Vertices send their values as *signals* to their in-neighbors and *collect signals* from their out-neighbors to compute their new values

Stutz, Philip, Abraham Bernstein, and William Cohen.
"Signal/collect: graph algorithms for the (semantic) web."
The Semantic Web–ISWC 2010 (2010): 764-780.

SIGNAL-COLLECT (SCATTER-GATHER)



outbox <– signal(V_i)

V_{i+1} <– collect(inbox)

No concurrent access to inbox and outbox

SIGNAL-COLLECT SEMANTICS

```
Input: directed graph G=(V,E)
activeVertices ← V
superstep ← 0

while activeVertices != ∅ do
  for v ∈ activeVertices do
    outbox_v ← signal(v)
    newState ← collect(inbox_v, v.state)
    if newState != v.state do
      v.state = newState
      activeVertices
  end for
  superstep ← superstep + 1
end while
```

SIGNAL-COLLECT INTERFACE

```
void signal();
VV getValue();
void sendMessageTo(I target, M message);
Iterator getOutEdges();
int superstep();

void collect(Iterator[M] messages);
void setValue(VV newValue);
VV getValue();
int superstep();
```

SIGNAL-COLLECT PAGERANK

```
void signal():
  for (edge <- getOutEdges()) do
    sendMessageTo(
        edge.target(), getValue()/numEdges)
  end for
```

distribute rank to neighbors

```
void collect(messages):
  sum = 0.0
  for (m <- messages) do
    sum = sum + m
  end for
```

sum up received messages

```
  setValue(0.15/numVertices + 0.85*sum)
```

update vertex rank

SIGNAL-COLLECT ANTI-PATTERNS

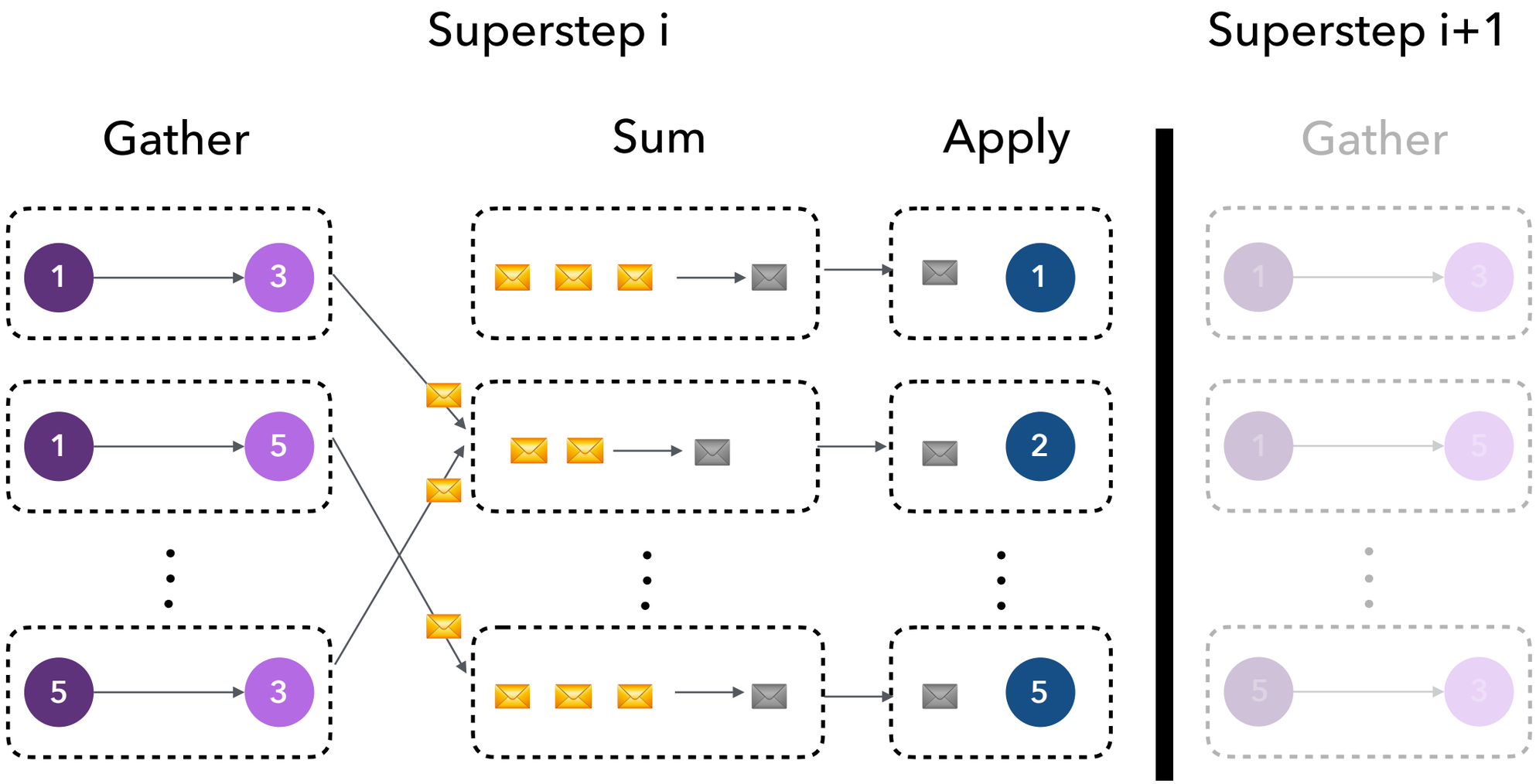
- Algorithms that require concurrent access to the inbox and outbox
  - signal has **read-access** to the vertex state and **write-access** to the outbox
  - collect has **read-access** to the inbox and **write-access** to the state
- Vertices cannot generate messages and update their states in the same phase
  - e.g. decide whether to propagate a message based on its content
  - **workaround:** store the message in the vertex-value

GATHER-SUM-APPLY-SCATTER (GSA)

- Express the computation from the view of a single vertex
- Vertices produce a message per edge, gather and aggregate partial results, update their state with the final aggregate

Gonzalez, Joseph E., et al.
"PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs."
OSDI. Vol. 12. No. 1. 2012.

GATHER-SUM-APPLY SUPERSTEPS



Message generation is parallelized over the edges!

GSA SEMANTICS

```
Input: directed graph G=(V,E)
a_v ← empty

for v∈V do
   for n∈v.inNeighbors do
      a_v ← sum(a_v, gather(S_v, S_(v,n), S_n))
   end for
   S_v ← apply(S_v, a_v)
   S(v,n) ← scatter(S_v, S_(v,n), S_n)
end for
```

GSA INTERFACE

```
T gather(VV sourceV, EV edgeV, VV targetV);

T sum(T left, T right);

VV apply(VV value, T sum);

EV scatter(VV newV, EV edgeV, VV oldV);
```

GSA PAGERANK

```
double gather(source, edge, target):
  return target.value() / target.numEdges()
```

compute partial rank

```
double sum(rank1, rank2):
  return rank1 + rank2
```

combine partial ranks

```
double apply(sum, currentRank):
  return 0.15 + 0.85*sum
```

update rank

VC VS. SIGNAL-COLLECT VS. GSA

	Update Function Properties	Update Function Logic	Communication Scope	Communication Logic
Vertex-Centric	arbitrary	arbitrary	any vertex	arbitrary
Signal-Collect	arbitrary	based on received messaged	any vertex	based on vertex state
GSA	associative & commutative	based on neighbors' values	neighborhood	based on vertex state

PROBLEMS WITH VERTEX-PARALLEL MODELS

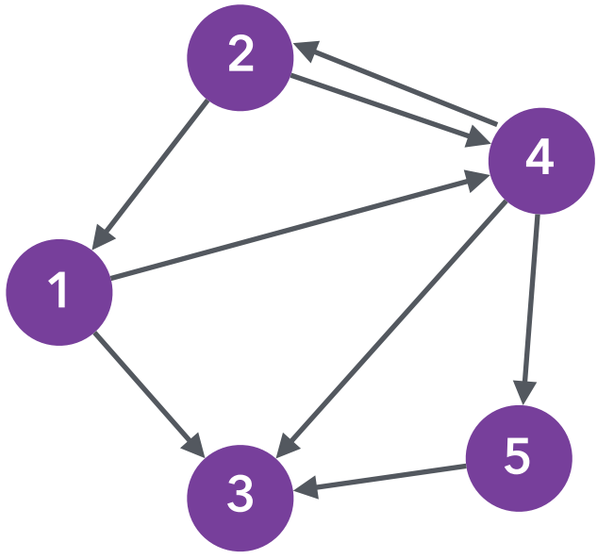
- Excessive communication
- Worker load imbalance
- Global Synchronization
- High memory requirements
  - inbox /outbox can grow too large
  - overhead for low-degree vertices in GSA

PARTITION-CENTRIC

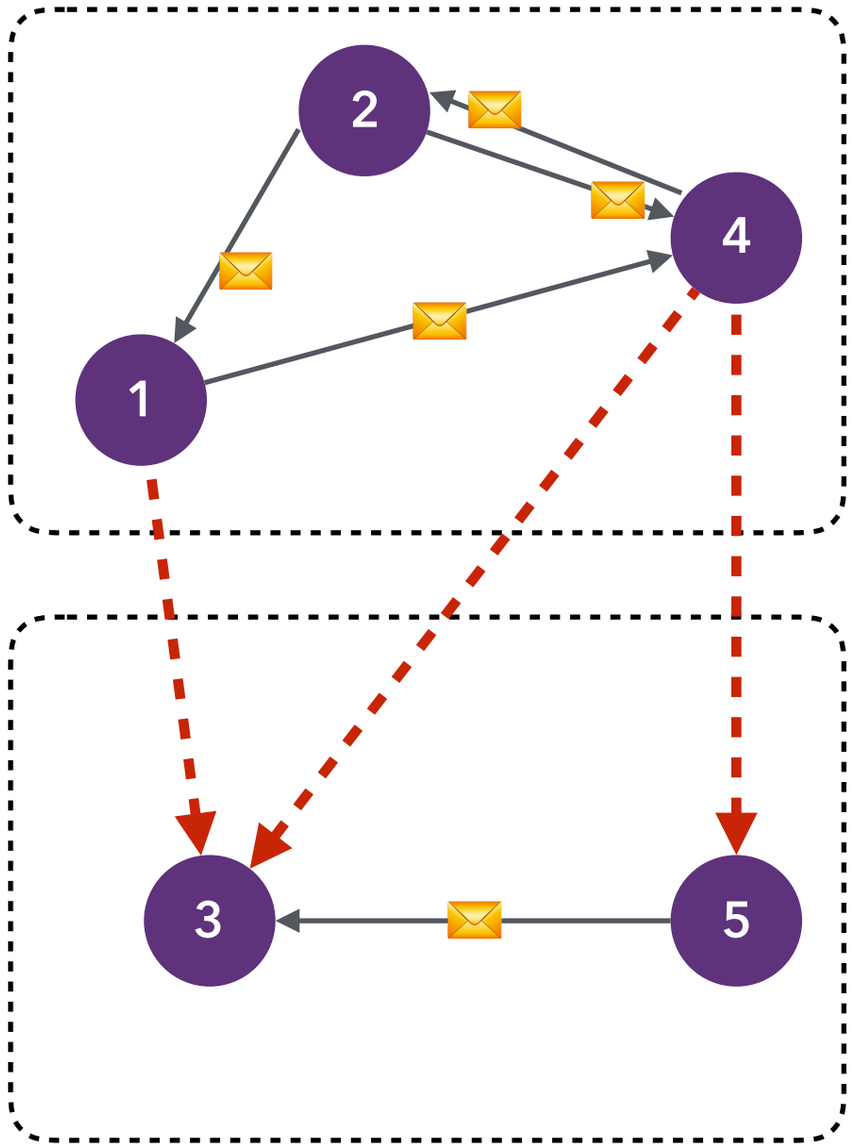
- Express the computation from the view of a partition
- Differentiate between *internal* and *boundary* vertices

Tian, Yuanyuan, et al.
"From think like a vertex to think like a graph."
Proceedings of the VLDB Endowment 7.3 (2013): 193-204.

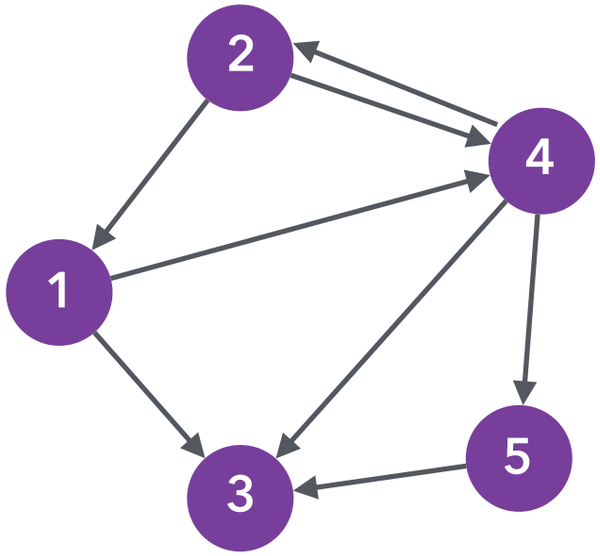
THINK LIKE A (SUB)GRAPH



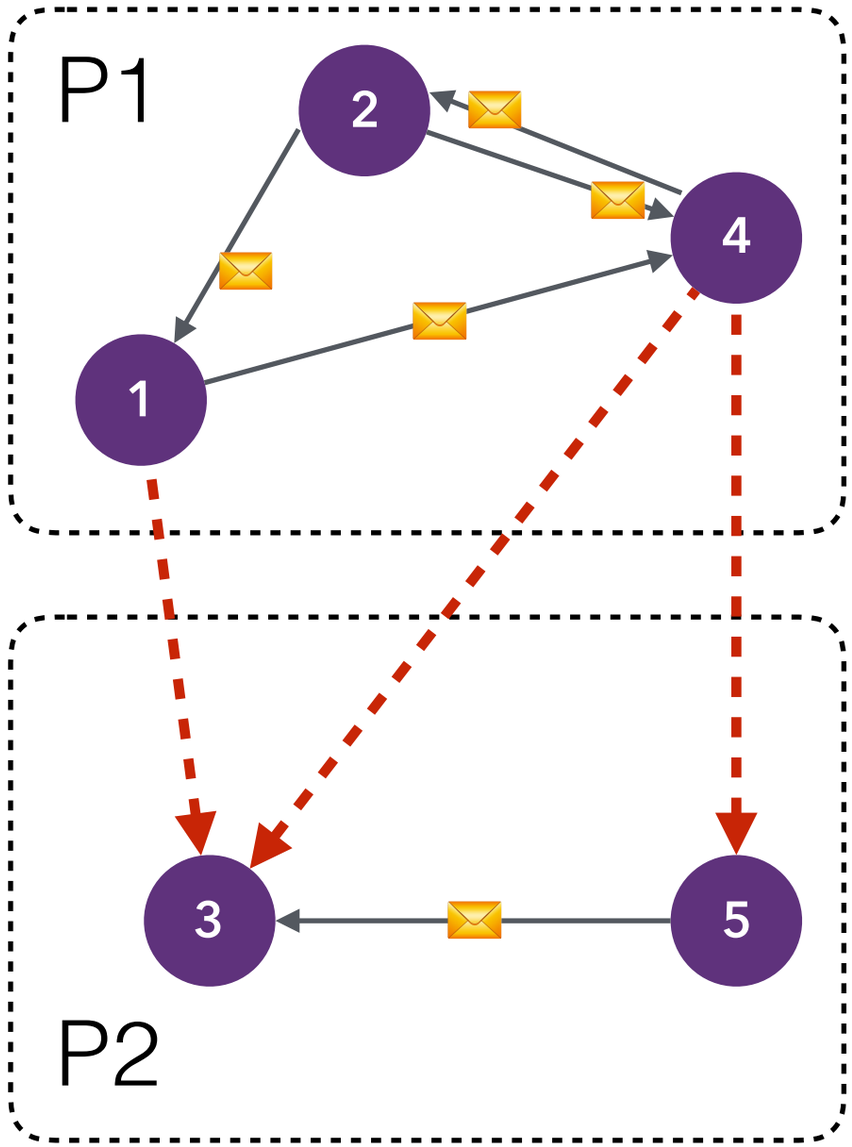
- compute() on the entire partition
- Information flows freely inside each partition
- Network communication between partitions, not vertices



THINK LIKE A (SUB)GRAPH

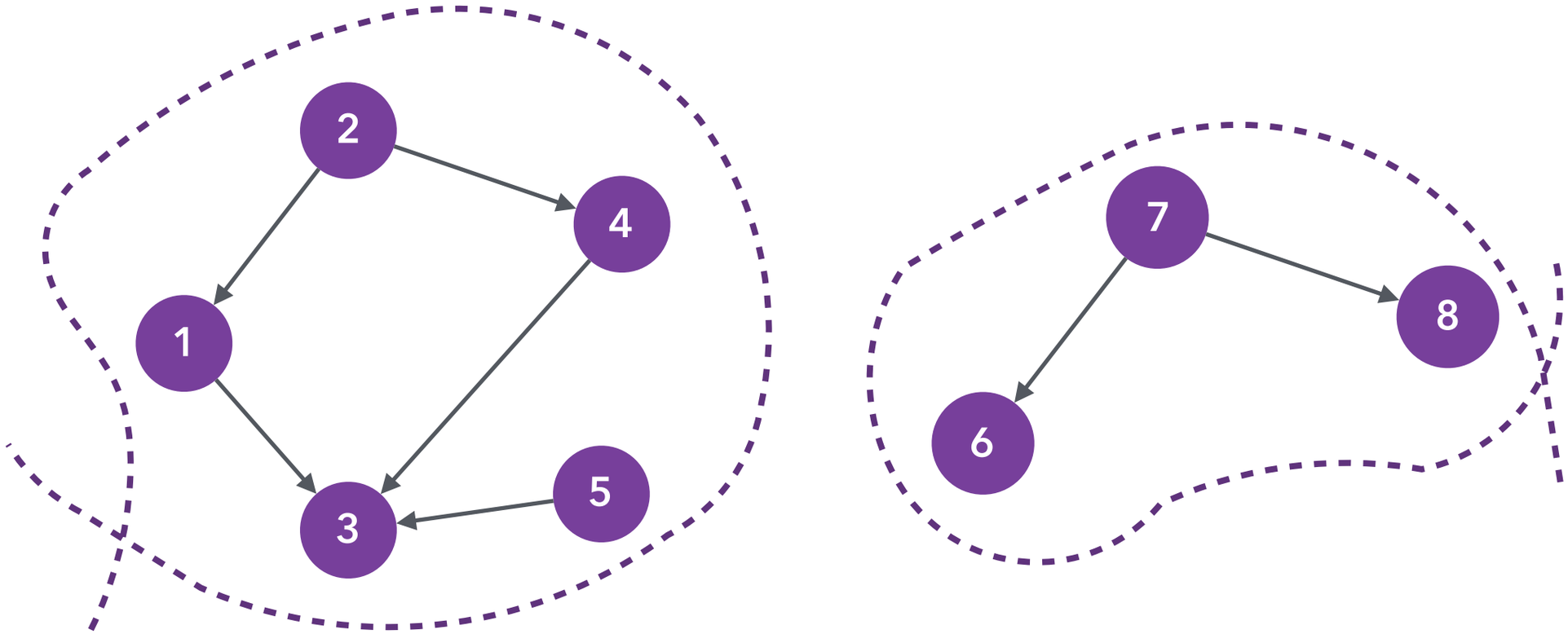


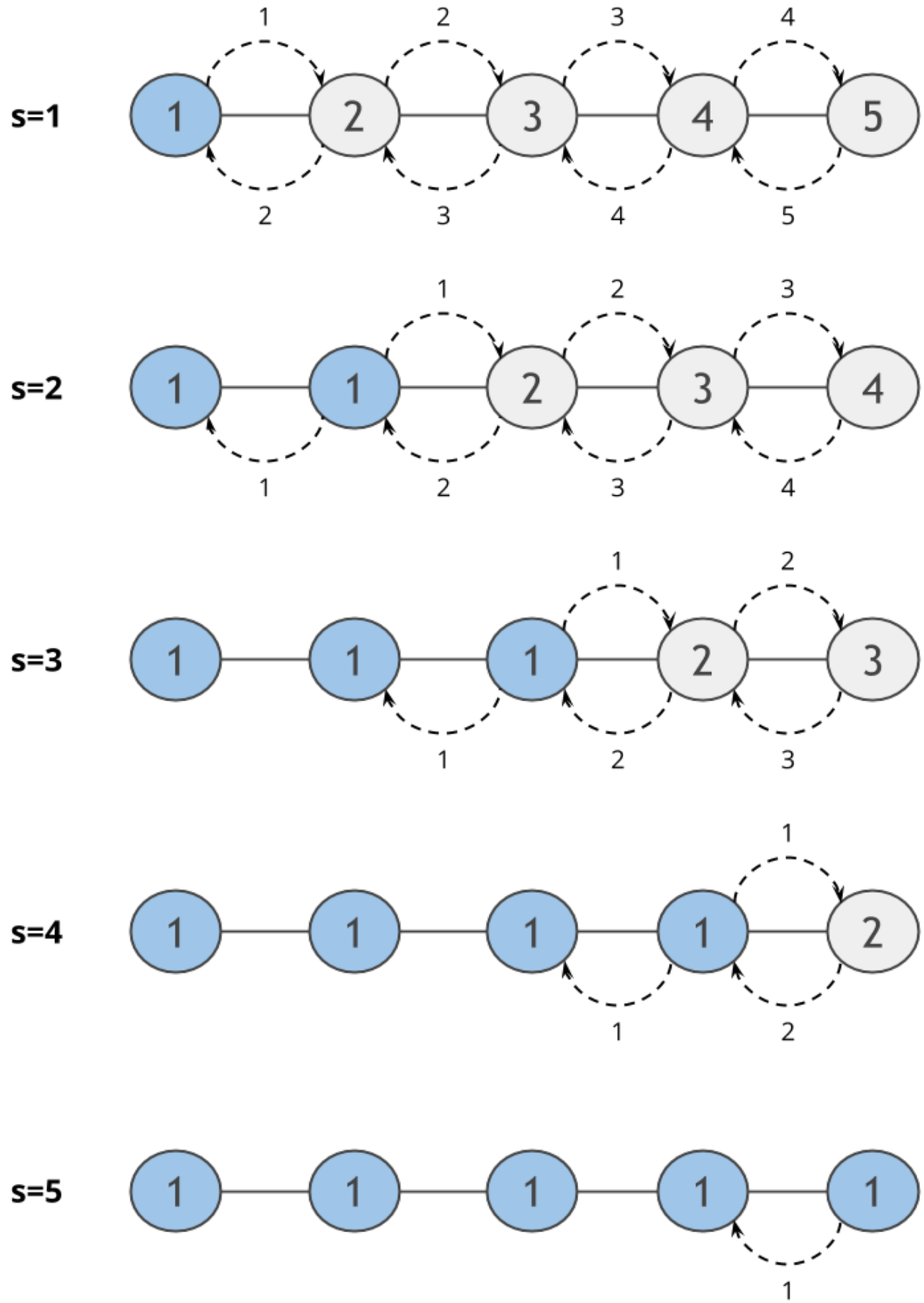
- 2 in an **internal** vertex in P1
- 1, 4 are **boundary** vertices



VERTEX-CENTRIC CONNECTED COMPONENTS

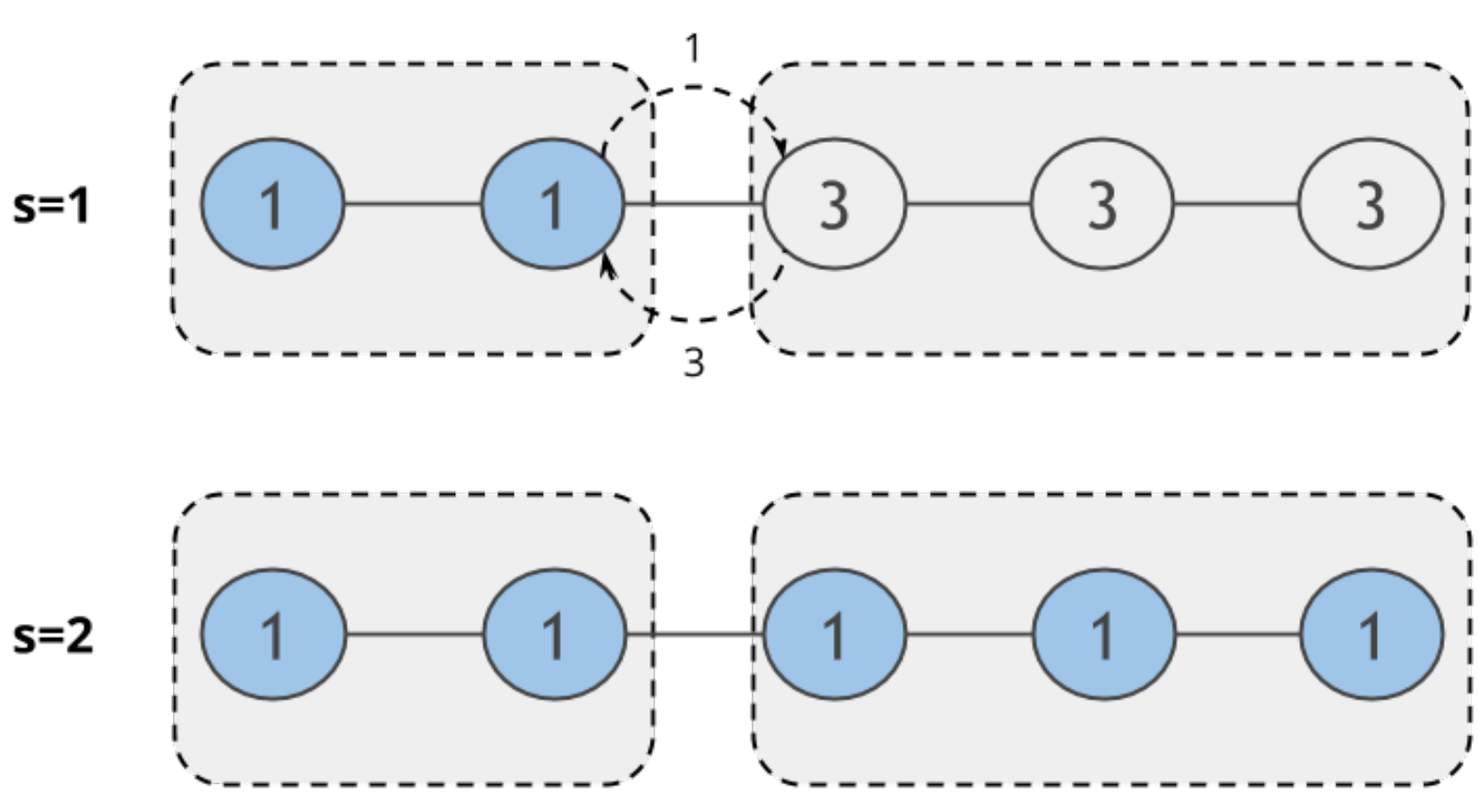
- Propagate the minimum value through the graph
- In each superstep, the value propagates one hop
- Requires diameter + 1 supersets to converge




s=1
1
2
3
4
5
1
2
3
4
s=2
1
1
2
3
4
1
2
3
4
s=3
1
1
1
2
3
1
2
1
2
3
s=4
1
1
1
1
2
1
2
1
s=5
1
1
1
1
1
1

PARTITION-CENTRIC CONNECTED COMPONENTS

- In each superstep, the value propagates throughout each subgraph
- Communication between partitions only
- Fewer supersteps until convergence



PARTITION-CENTRIC INTERFACE

```
void compute();

void sendMessageTo(I target, M message);

int superstep();

void voteToHalt();

boolean containsVertex(I id);

boolean isInternalVertex(I id);

boolean isBoundaryVertex(I id);

Collection getInternalVertices();

Collection getBoundaryVertices();

Collection getAllVertices();
```

PARTITION-CENTRIC PAGERANK

```
void compute():
  if superstep() == 0 then
    for v ∈ getAllVertices() do
      v.getValue().pr = 0
      v.getValue().delta = 0
    end for
end if
for iv ∈ internalVertices() do
  outEdges = iv.getNumOutEdges()
  if superstep() == 0 then
    iv.getValue().delta+ = 0.15
  end if
  iv.getValue().delta+ = iv.getMessages()
  if iv.getValue().delta > 0 then
    iv.getValue().pr+ = iv.getValue().delta
    u = 0.85 * iv.getValue().delta/outEdges
    while iv.iterator.hasNext() do
      neighbor = getVertex(iv.iterator().next())
      neighbor.getValue().delta+ = u
    end while
  end if
  iv.getValue().delta = 0
end for
for bv ∈ boundaryVertices() do
  bvID = bv.getVertexId()
  if bv.getValue().delta > 0 then
    sendMessageTo(bvID, bv.getValue().delta)
    bv.getValue().delta = 0
  end if
end for
```

PARTITION-CENTRIC PAGERANK

```
void compute():
  if superstep() == 0 then
    for v ∈ getAllVertices() do
      v.getValue().pr = 0
      v.getValue().delta = 0
    end for
  end if
  for iv ∈ internalVertices() do
    outEdges = iv.getNumOutEdges()
    if superstep() == 0 then
      iv.getValue().delta+ = 0.15
    end if
    iv.getValue().delta+ = iv.getMessages()
    if iv.getValue().delta > 0 then
      iv.getValue().pr+ = iv.getValue().delta
      u = 0.85 * iv.getValue().delta/outEdges
      while iv.iterator.hasNext() do
        neighbor = getVertex(iv.iterator().next())
        neighbor.getValue().delta+ = u
      end while
    end if
    iv.getValue().delta = 0
  end for
  for bv ∈ boundaryVertices() do
    bvID = bv.getVertexId()
    if bv.getValue().delta > 0 then
      sendMessageTo(bvID, bv.getValue().delta)
      bv.getValue().delta = 0
    end if
  end for
```

Initialization

```
if superstep() == 0 then
  for v ∈ getAllVertices() do
    v.getValue().pr = 0
    v.getValue().delta = 0
  end for
end if
```

PARTITION-CENTRIC PAGERANK

```
void compute():
  if superstep() == 0 then
    for v ∈ getAllVertices() do
      v.getValue().pr = 0
      v.getValue().delta = 0
    end for
  end if
  for iv ∈ internalVertices() do
    outEdges = iv.getNumOutEdges()
    if superstep() == 0 then
      iv.getValue().delta+ = 0.15
    end if
    iv.getValue().delta+ = iv.getMessages()
    if iv.getValue().delta > 0 then
      iv.getValue().pr+ = iv.getValue().delta
      u = 0.85 * iv.getValue().delta/outEdges
      while iv.iterator.hasNext() do
        neighbor = getVertex(iv.iterator().next())
        neighbor.getValue().delta+ = u
      end while
    end if
    iv.getValue().delta = 0
  end for
  for bv ∈ boundaryVertices() do
    bvID = bv.getVertexId()
    if bv.getValue().delta > 0 then
      sendMessageTo(bvID, bv.getValue().delta)
      bv.getValue().delta = 0
    end if
  end for
```

Internal Vertices

```
for iv ∈ internalVertices() do
  outEdges = iv.getNumOutEdges()
  if superstep() == 0 then
    iv.getValue().delta+ = 0.15
  end if
  iv.getValue().delta+ = iv.getMessages()
  if iv.getValue().delta > 0 then
    iv.getValue().pr+ = iv.getValue().delta
    u = 0.85 * iv.getValue().delta/outEdges
    while iv.iterator.hasNext() do
      neighbor = getVertex(iv.iterator().next())
      neighbor.getValue().delta+ = u
    end while
  end if
  iv.getValue().delta = 0
end for
```

PARTITION-CENTRIC PAGERANK

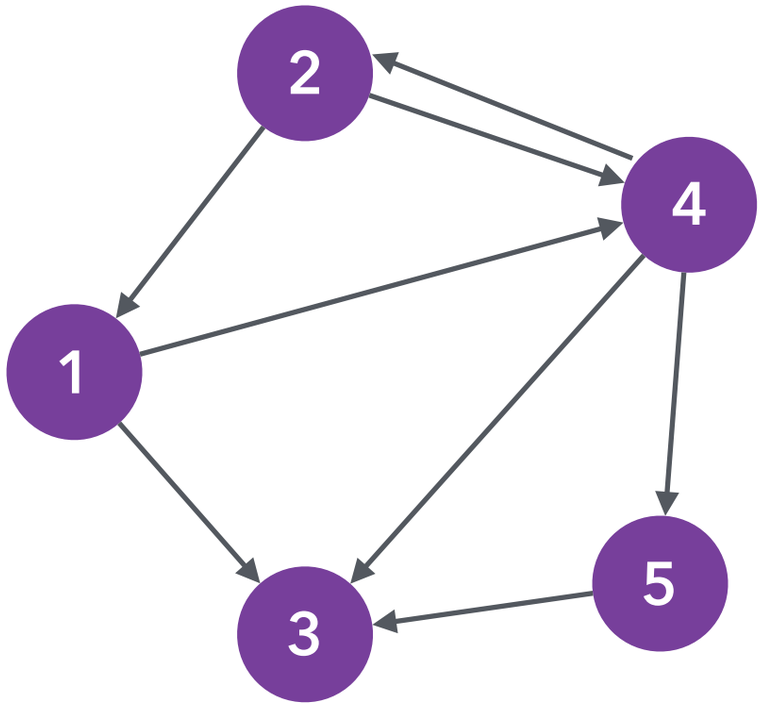
```
void compute():
  if superstep() == 0 then
    for v ∈ getAllVertices() do
      v.getValue().pr = 0
      v.getValue().delta = 0
    end for
  end if
  for iv ∈ internalVertices() do
    outEdges = iv.getNumOutEdges()
    if superstep() == 0 then
      iv.getValue().delta+ = 0.15
    end if
    iv.getValue().delta+ = iv.getMessages()
    if iv.getValue().delta > 0 then
      iv.getValue().pr+ = iv.getValue().delta
      u = 0.85 * iv.getValue().delta/outEdges
      while iv.iterator.hasNext() do
        neighbor = getVertex(iv.iterator().next())
        neighbor.getValue().delta+ = u
      end while
    end if
    iv.getValue().delta = 0
  end for
  for bv ∈ boundaryVertices() do
    bvID = bv.getVertexId()
    if bv.getValue().delta > 0 then
      sendMessageTo(bvID, bv.getValue().delta)
      bv.getValue().delta = 0
    end if
  end for
```

Boundary Vertices

```
for bv ∈ boundaryVertices() do
  bvID = bv.getVertexId()
  if bv.getValue().delta > 0 then
    sendMessageTo(bvID, bv.getValue().delta)
    bv.getValue().delta = 0
  end if
end for
```

GENERAL-PURPOSE PROGRAMMING MODELS FOR DISTRIBUTED GRAPH PROCESSING

LINEAR ALGEBRA

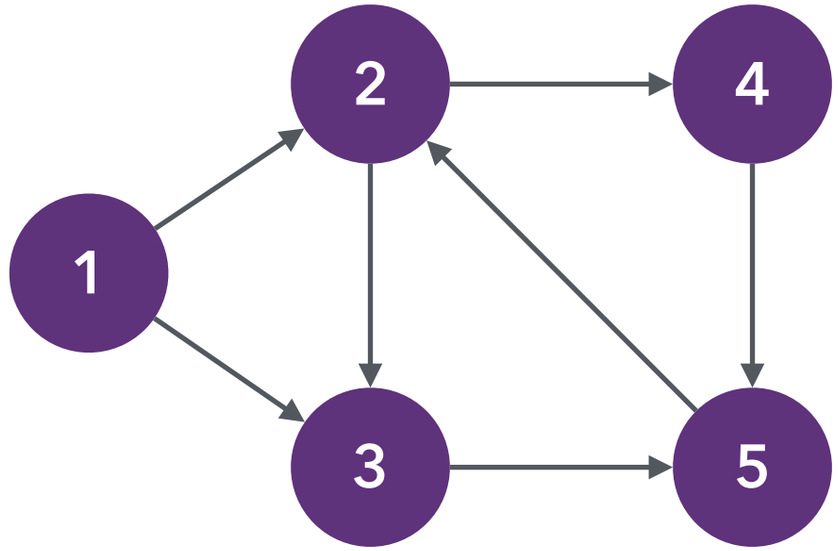


Adjacency Matrix

	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	1	1	0	1
5	0	0	1	0	0

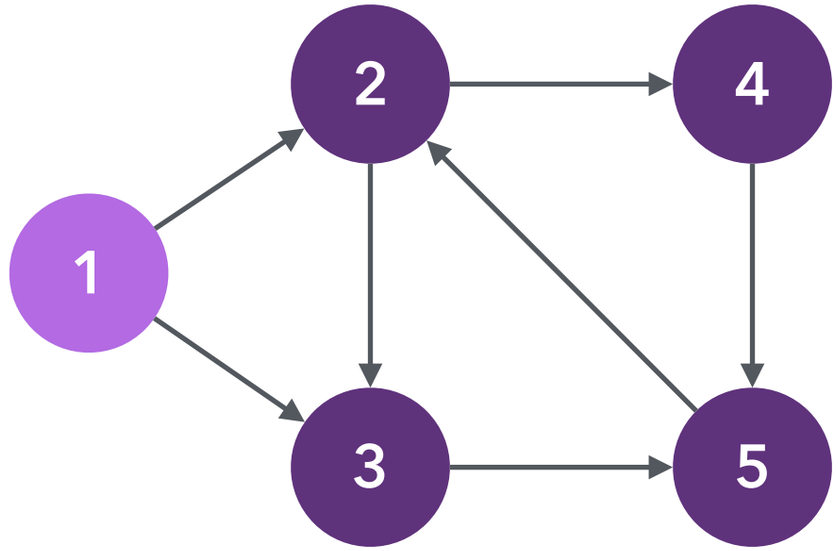
- Partition by rows, columns, blocks
- Efficient compressed-row/column representations
- Algorithms expressed as **vector-matrix** multiplications

BREADTH-FIRST SEARCH



	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	1	1	0	1
5	0	0	1	0	0

BREADTH-FIRST SEARCH



1	0	0	0	0

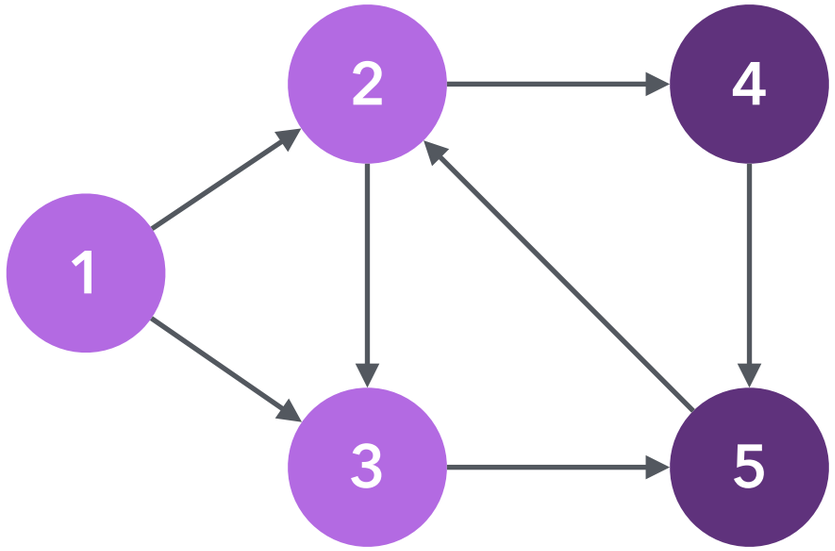
X

	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	1	1	0	1
5	0	0	1	0	0

=

0	1	1	0	0

BREADTH-FIRST SEARCH



1	0	0	0	0

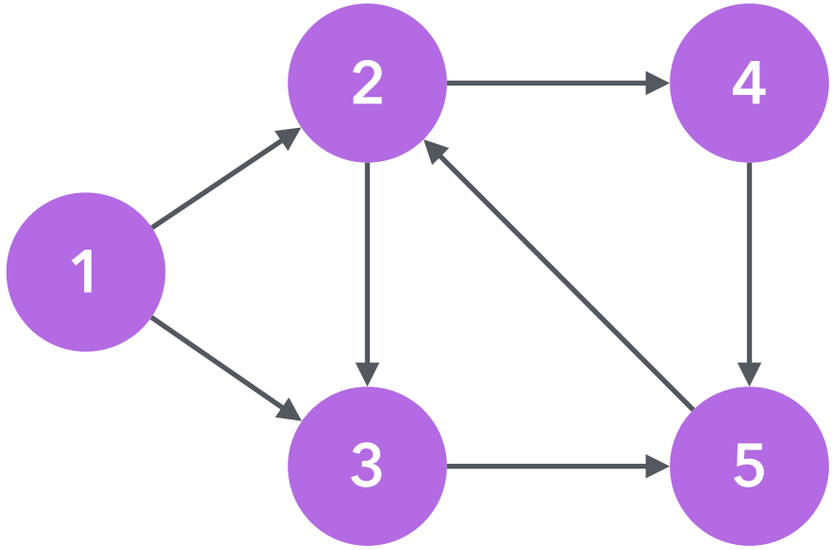
X

	1	2	3	4	5
1	0	0	**1**	**1**	0
2	**1**	0	0	**1**	0
3	0	0	0	0	0
4	0	**1**	**1**	0	**1**
5	0	0	**1**	0	0

=

0	1	1	0	0

BREADTH-FIRST SEARCH



0	1	1	0	0

X

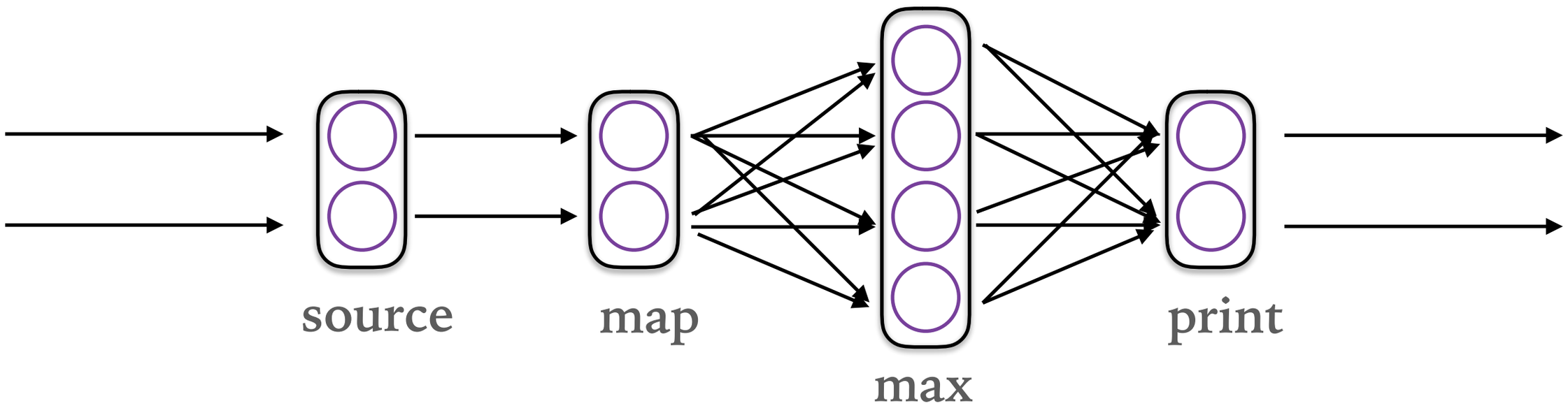
	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	1	1	0	1
5	0	0	1	0	0

=

0	0	1	1	1

DISTRIBUTED DATAFLOWS

- Dataflow programs are directed graphs, where nodes are data-parallel operators (computations) and edges represent data dependencies
- e.g.: Apache Spark, Apache Flink, Naiad
- Graphs are represented with 2 datasets: vertices and edges



PAGERANK IN APACHE SPARK

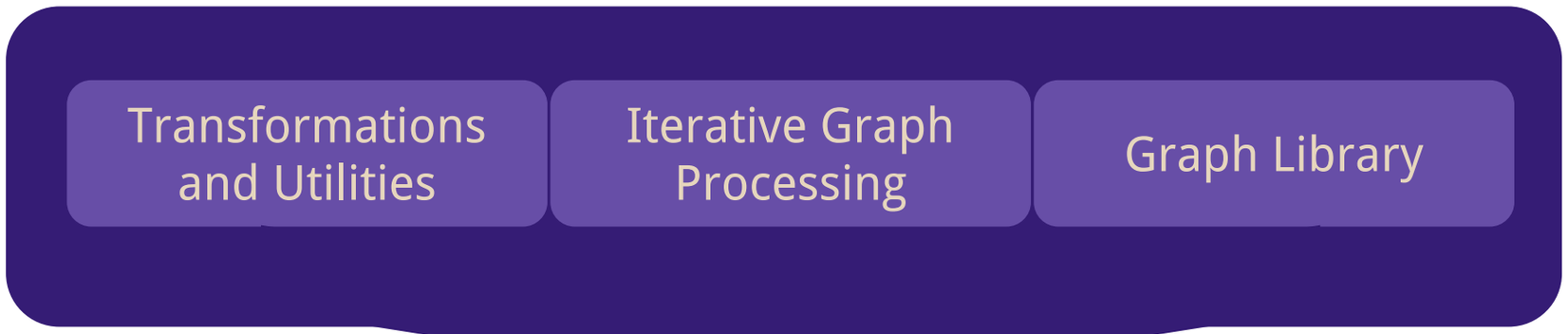
```
for (i <- 1 to iters) {
  val contribs =
    links.join(ranks).values
         .flatMap {
           case (urls, rank) =>
             val size = urls.size
             urls.map(url => (url, rank / size))
         }
  ranks = contribs.reduceByKey(_+_)
                  .mapValues(0.15 + 0.85 *_)
}
val output = ranks.collect()
```

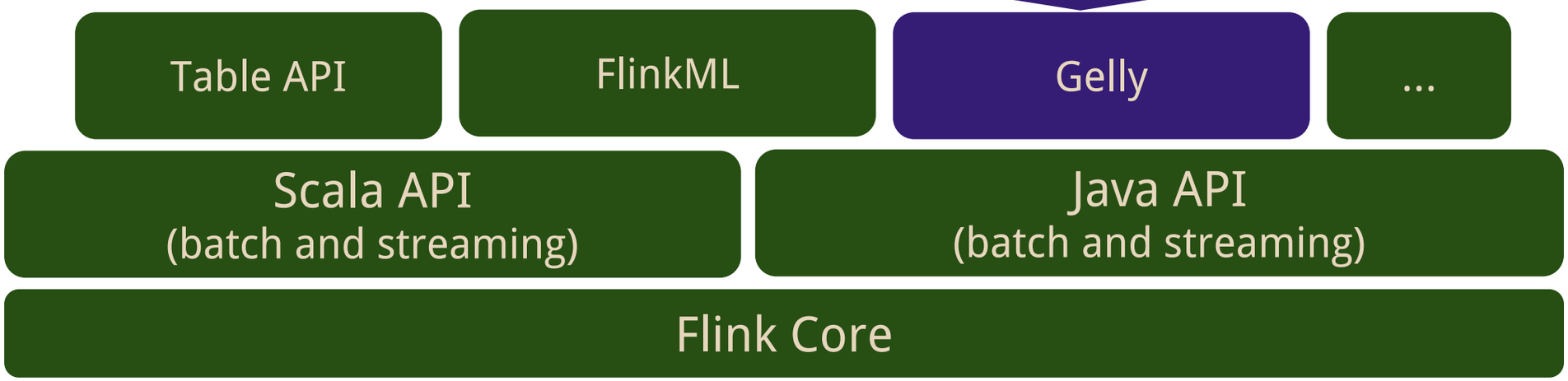
HIGH-LEVEL GRAPH APIS ON DATA FLOWS

- Gonzalez, Joseph E., et al. "**GraphX: Graph Processing in a Distributed Dataflow Framework.**" OSDI. Vol. 14. 2014.
- Bu, Yingyi, et al. "**Pregelix: Big (ger) graph analytics on a dataflow engine.**" Proceedings of the VLDB Endowment 8.2 (2014): 161-172.
- Murray, Derek G., et al. "**Naiad: a timely dataflow system.**" OSDI, ACM, 2013.

GELLY: THE APACHE FLINK GRAPH API

- Java & Scala Graph APIs on top of Flink's DataSet API





WHY GRAPH PROCESSING WITH APACHE FLINK?

- Native Iteration Operators
- DataSet Optimizations
- Ecosystem Integration
- Memory Management and Custom Serialization

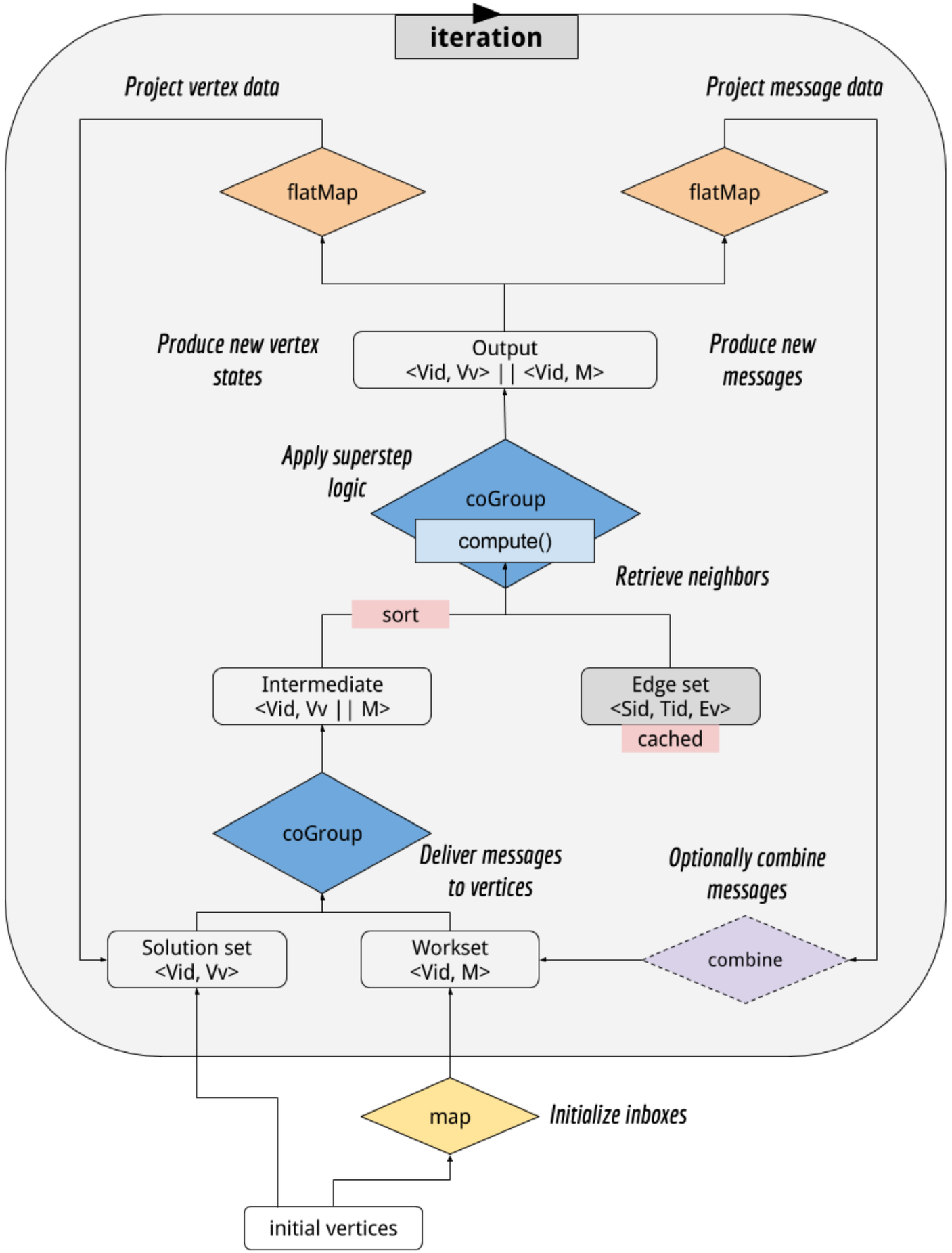
FAMILIAR ABSTRACTIONS IN GELLY

- Gelly maps high-level abstractions to dataflows
  - vertex-centric
  - scatter-gather
  - gather-sum-apply
  - partition-centric

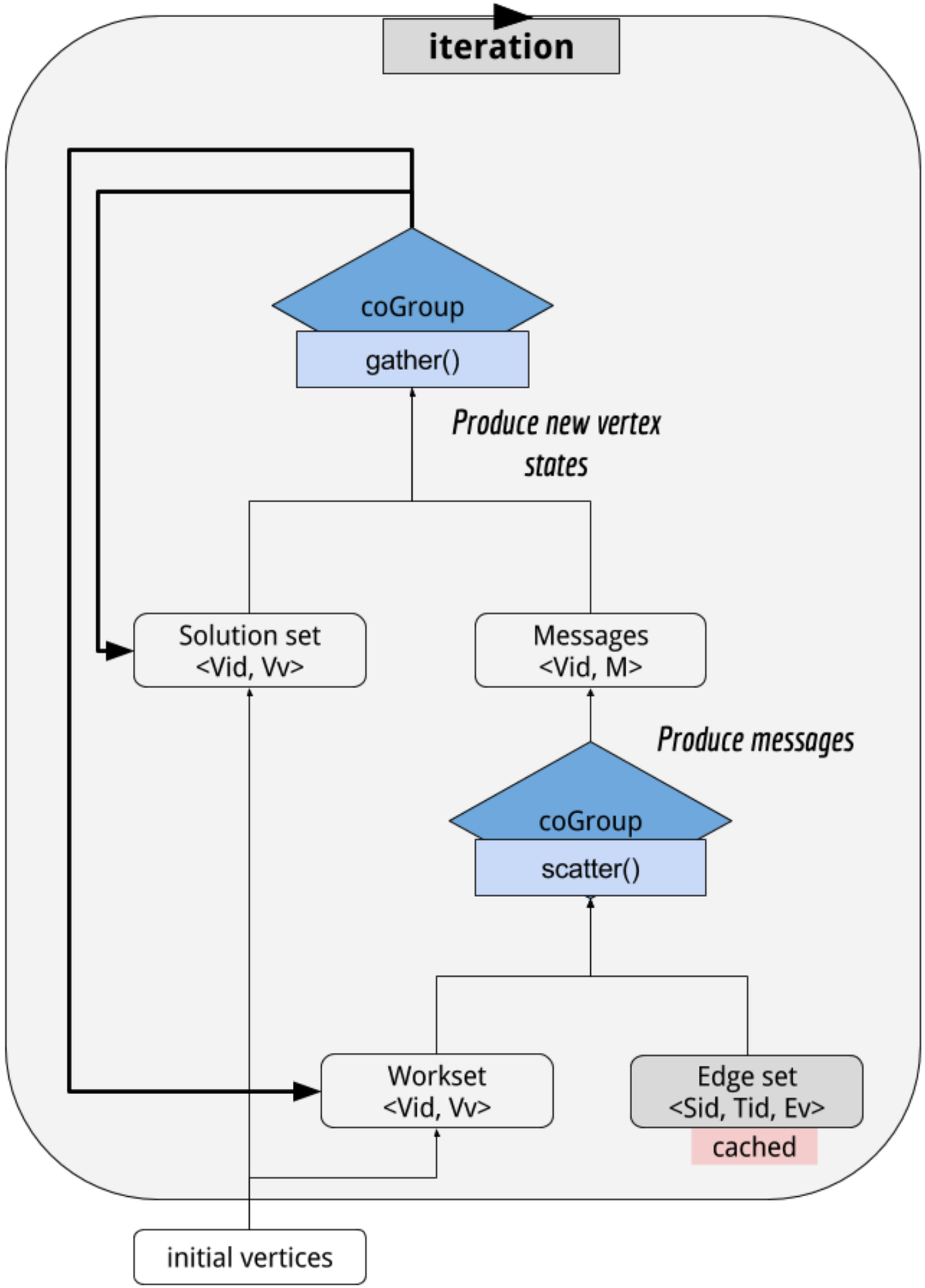
GELLY VERTEX-CENTRIC SHORTEST PATHS

```
final class SSSPComputeFunction extends ComputeFunction {

  override def compute(vertex: Vertex, messages: MessageIterator) = {

  var minDistance = if (vertex.getId == srcId) 0 else Double.MaxValue

    while (messages.hasNext) {
        val msg = messages.next
        if (msg < minDistance)
           minDistance = msg
    }

    if (vertex.getValue > minDistance) {
      setNewVertexValue(minDistance)
      for (edge: Edge <- getEdges)
        sendMessageTo(edge.getTarget, vertex.getValue + edge.getValue)
}
```

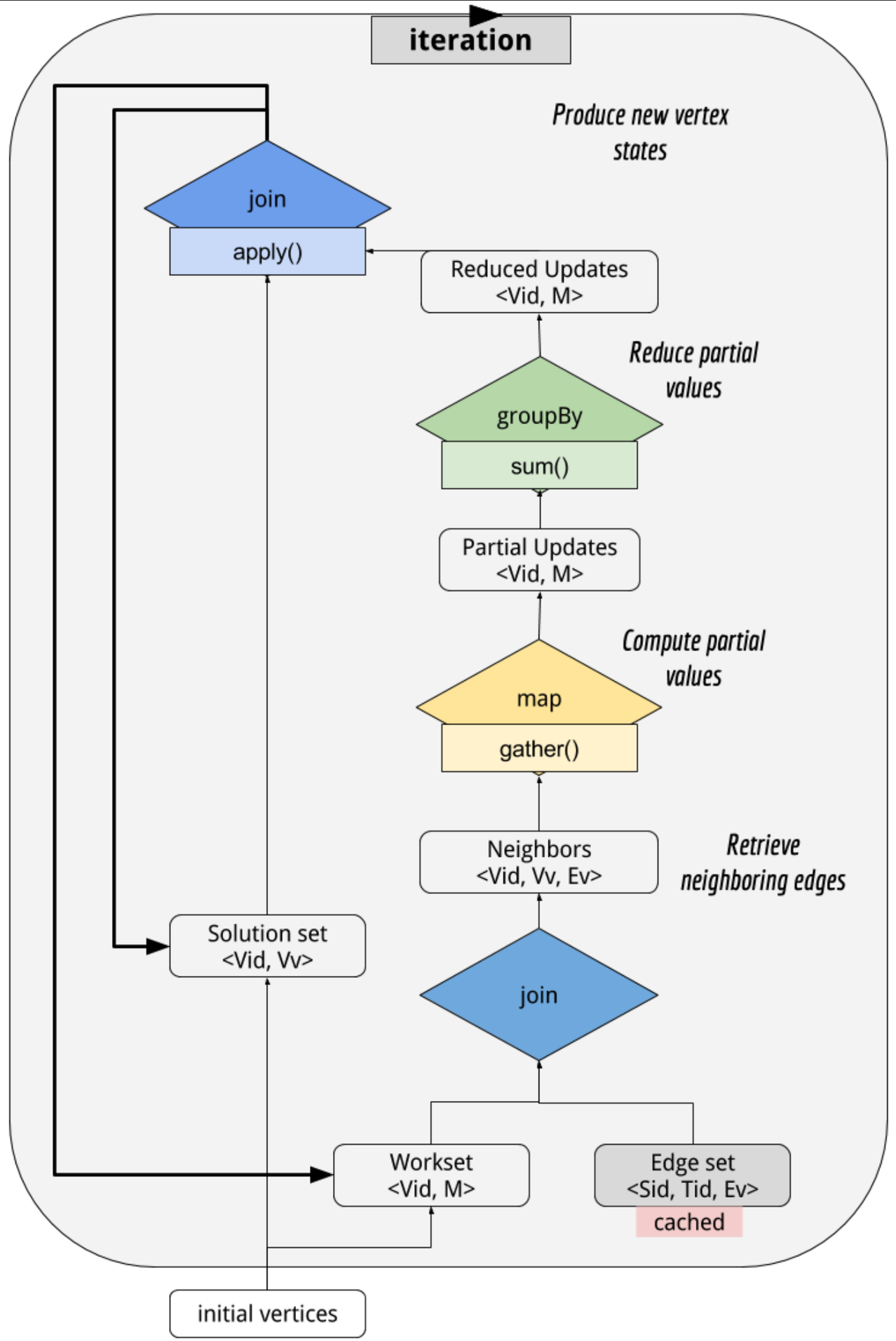
Vertex-Centric Dataflow



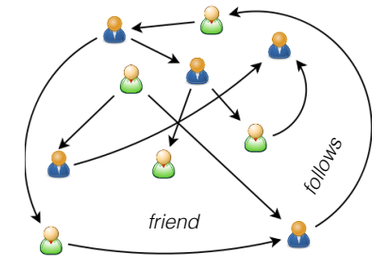
Scatter-Gather Dataflow



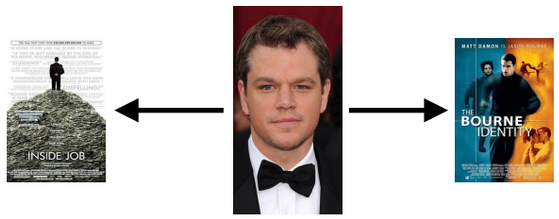
Gather-Sum-Apply Dataflow



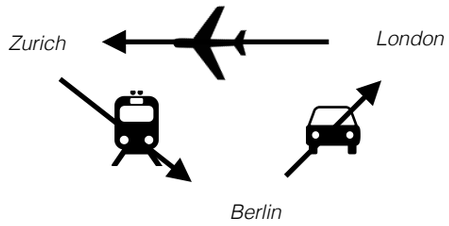
RECAP



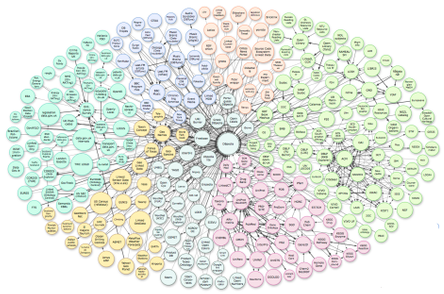
"conservative" *"liberal"*



If you like "Inside job"
you might also like "The Bourne Identity"



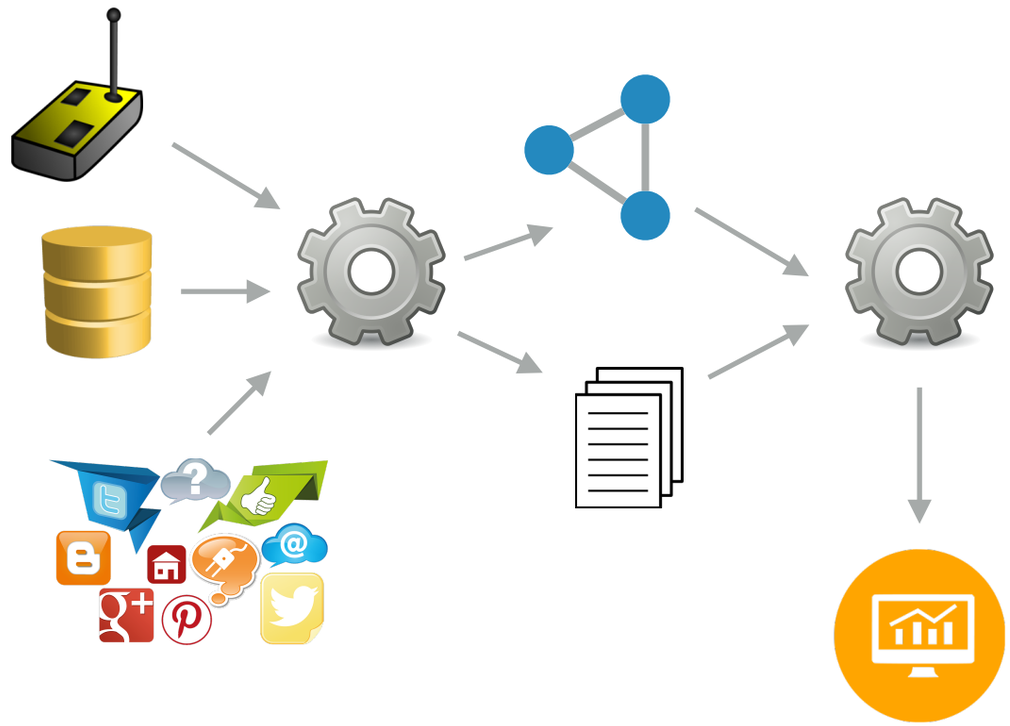
What's the cheapest way
to reach Zurich from London through Berlin?



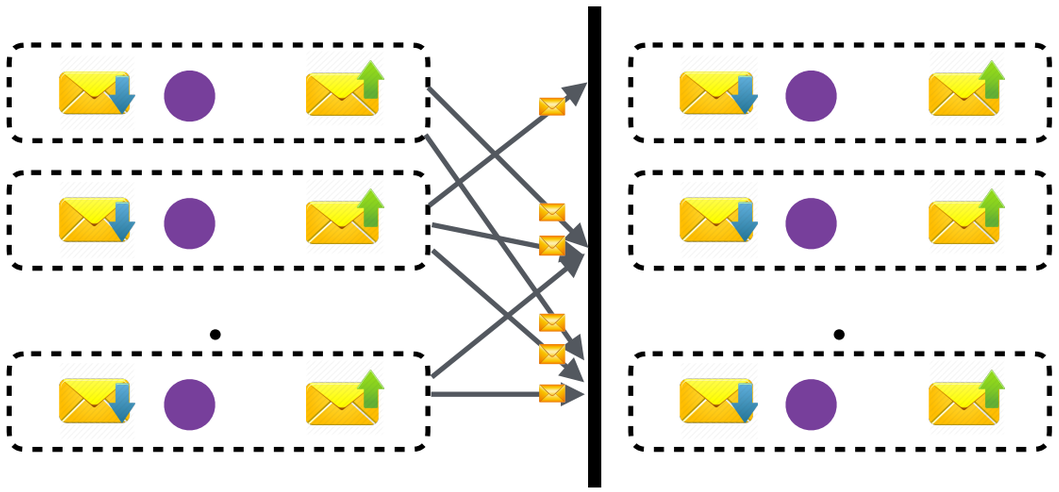
These are the top-10 relevant results
for the search term "graph"

Diverse graph models and applications

Do you need distributed graph processing?



Specialized graph processing abstractions

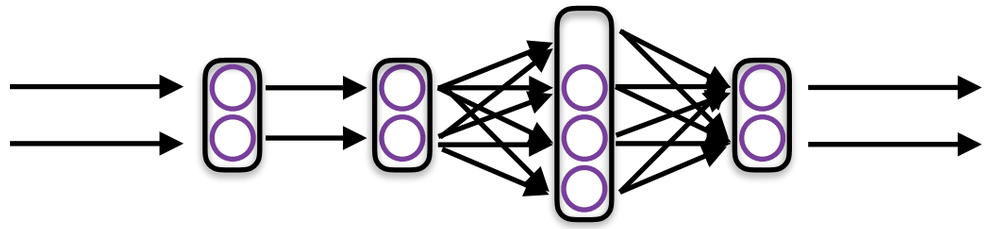


	Update Function Properties	Update Function Logic	Communication Scope	Communication Logic
Vertex-Centric	arbitrary	arbitrary	any vertex	arbitrary
Signal-Collect	arbitrary	based on received messaged	any vertex	based on vertex state
GSA	associative & commutative	based on neighbors' values	neighborhood	based on vertex state

Vertex-parallel models are very widespread
Beware of performance issues and anti-patterns!

General-purpose models for graph processing

	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	0
4	0	1	1	0	1
5	0	0	1	0	0



Linear algebra primitives
Distributed dataflows

Kalavri, Vasiliki, Vladimir Vlassov, and Seif Haridi.
"High-Level Programming Abstractions for Distributed Graph Processing."
arXiv preprint arXiv:1607.02646 (2016).

Vasia Kalavri
kalavriv@inf.ethz.ch

Programming Models and Tools for Distributed Graph Processing

Vasia Kalavri
kalavriv@inf.ethz.ch

31st British International Conference on Databases
10 July 2017, London, UK