TDDD43 Advanced Data Models and Databases

Graph Data Systems

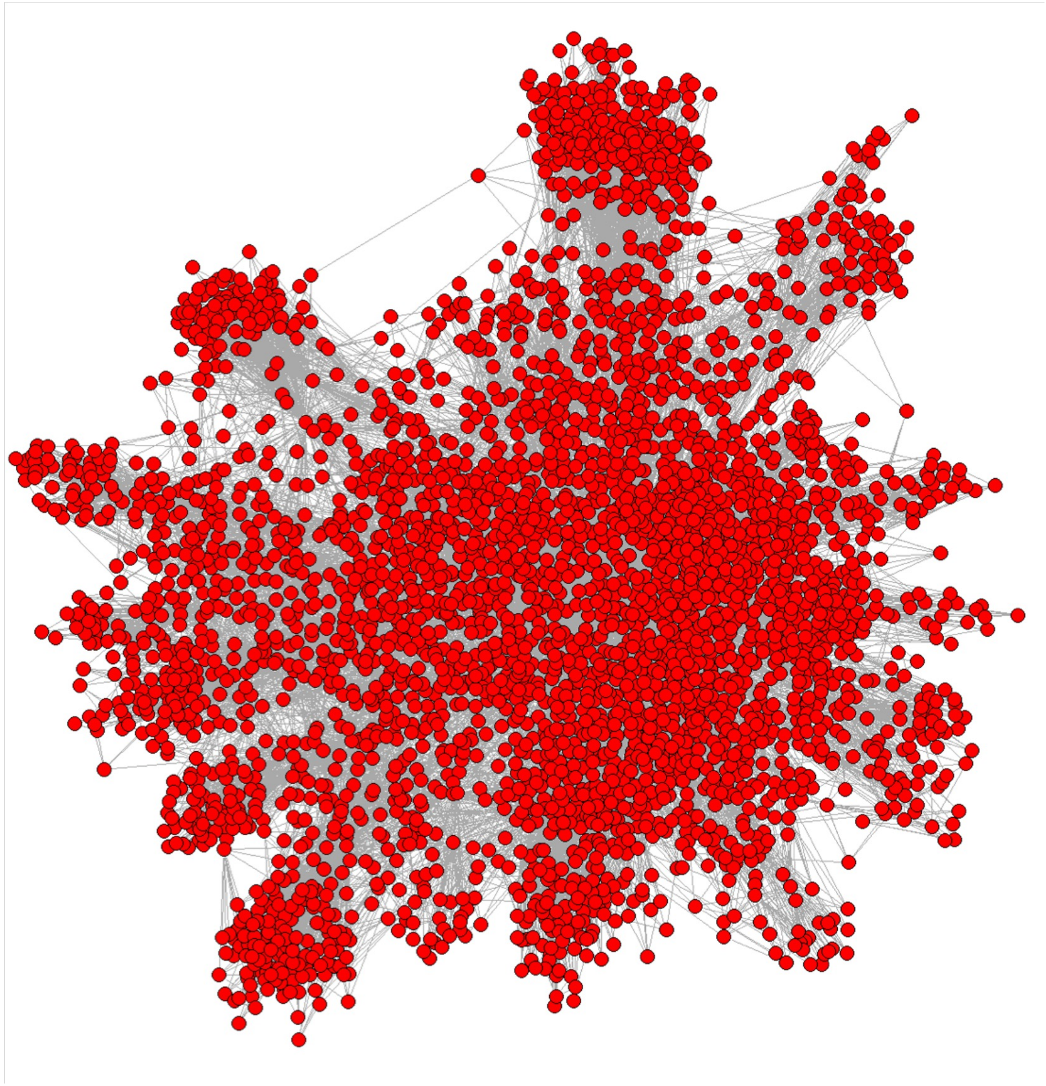
Huanyu Li
huanyu.li@liu.se

Based on slides by Olaf Hartig

LINKÖPING UNIVERSITY

Graphs are Everywhere

- Transportation networks
- Bibliographic networks
- Computer networks
- Social networks
- Topic maps
- Knowledge bases
- Protein interactions
- Biological food chains
- etc.



Different Graph Data Systems

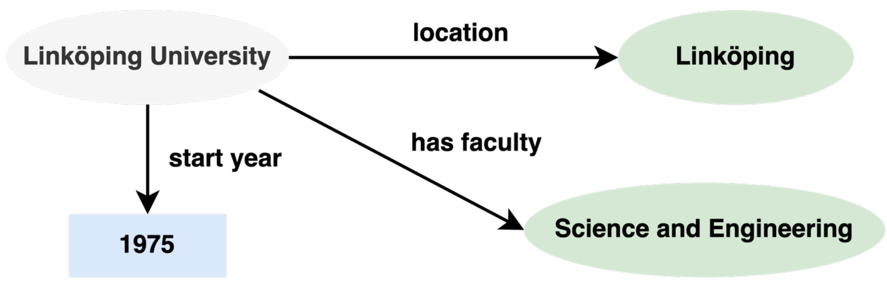
- Triple stores
  - Data model: RDF
  - Typically, pattern matching queries
- Graph databases
  - Prevalent data model: property graphs
  - Typically, navigational queries
- Graph processing systems
  - Prevalent data model: generic graphs
  - Typically, complex graph analysis tasks

Graph Data Models

LINKÖPING
UNIVERSITY

Recap of RDF Data Model

- Data is represented as a set of triples
  - A triple: (subject, predicate, object)
- Subject: resources
- Predicate: properties
- Object: literals or resources



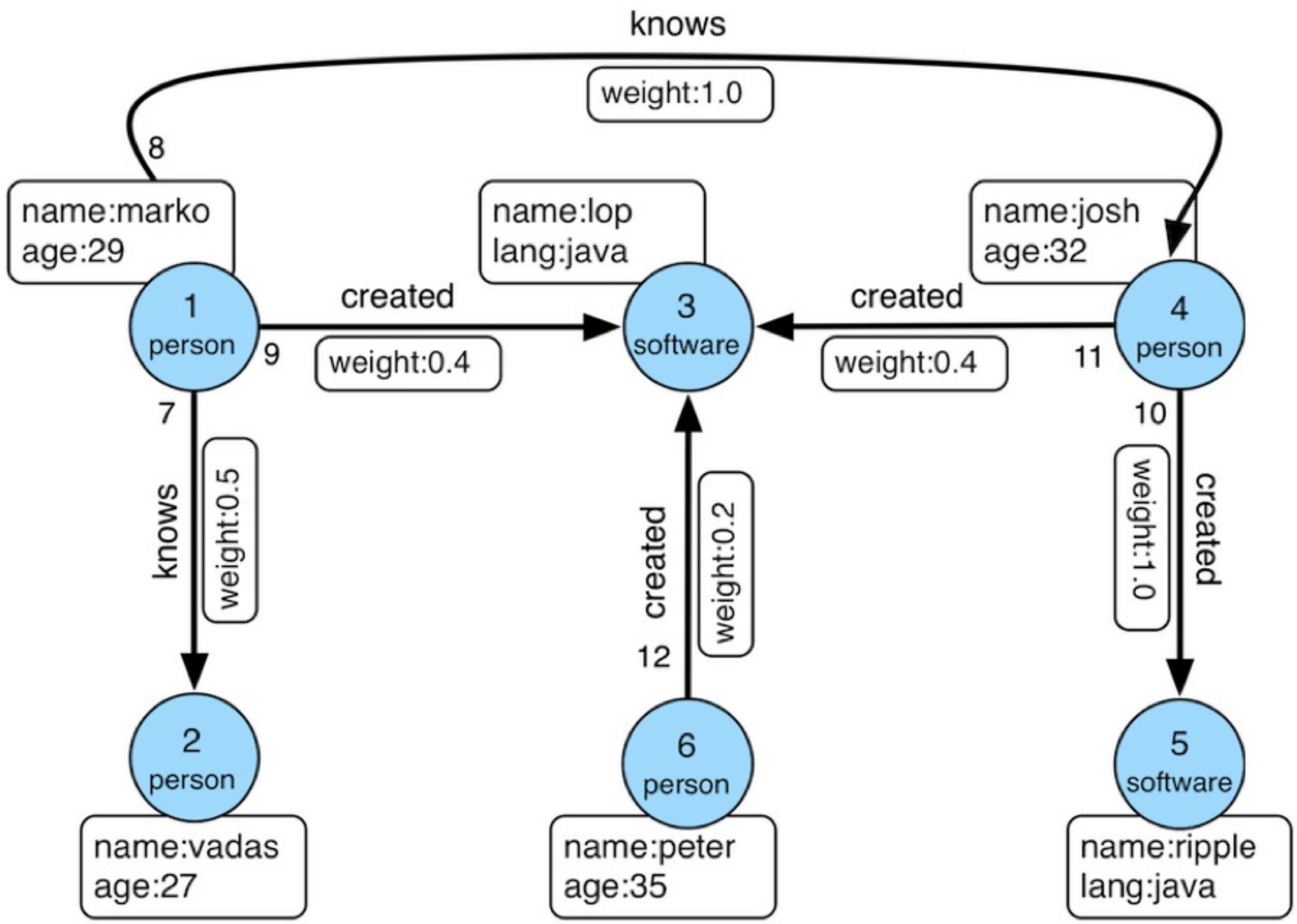
- Such a set of triples may be understood as a graph
  - Triples as directed edges
  - Subjects and objects as vertexes
  - Edges labeled by predicate
- W3C recommendation and standardization

Property Graph

- *“A property graph is made up of nodes, relationships, and properties.*
- *Nodes contain properties […] in the form of arbitrary key-value pairs. The keys are strings and the values are arbitrary data types.*
- *A relationship always has a direction, a label, and a start node and an end node.*
- *Like nodes, relationships can also have properties.” [1]*

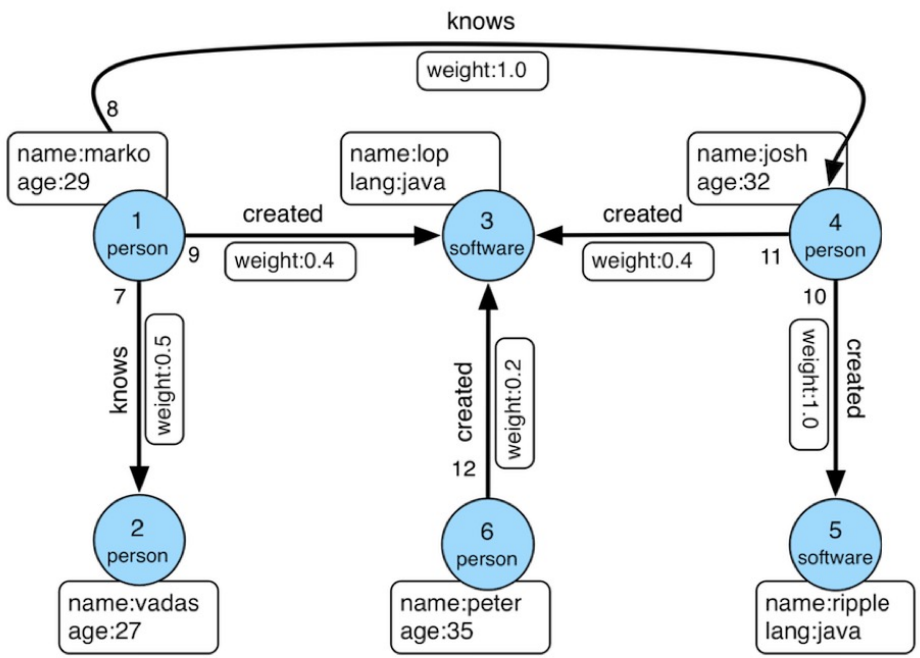
[1] Ian Robinson, Jim Webber, and Emil Eifŕem. Graph Databases. O’Reilly Media, 2013.

(Labeled) Property Graph



(Labeled) Property Graph

- Directed multigraph
  - Multiple edges between the same pair of nodes
- Any node and any edge may have a label
- Any node and any edge may have an arbitrary set of key-value pairs (“properties”)



Property Graphs versus RDF Graphs

- Similarities
  - Directed multigraphs
  - Labels on edges and on nodes
  - Attributes with values on nodes

- Differences
  - No edge properties in RDF graphs
  - Edge labels cannot appear as nodes in a PG (in RDF, we may have <s1, p1, o1> and <p1, p2, o2>)
  - No multivalued (node) properties in a PG
  - Node and edge identifiers in a PG are local to the PG, while URIs in RDF graphs are globally unique identifiers

Exercise: converting RDF to Property Graph

- Given a set of RDF triples

ex:restaurant_A	rdf:type	ex:Restaurant
ex:restaurant_A	ex:hasWebsite	"http://resaurtantA.org"
ex:restaurant_A	ex:hasSite	ex:Linköping
ex:restaurant_A	ex:startDate	"2012-02-01"
ex:restaurant_B	rdf:type	ex:Restaurant
ex:restaurant_B	ex:hasWebsite	"http://resaurtantB.org"
ex:restaurant_B	ex:hasSite	ex:Linköping
ex:restaurant_B	ex:startDate	"2013-02-01"
ex:Linköping	rdf:type	ex:City

Generic Graphs

- Data model
  - Directed multigraphs
  - Arbitrary user-defined data structure can be used as value of a vertex (node) or an edge (e.g., a Java object)
- Example (Flink Gelly API)

```
// create new vertexes with a Long ID and a String value
Vertex<Long, String> v1 = new Vertex<Long, String>(1L, "foo");
Vertex<Long, String> v2 = new Vertex<Long, String>(2L, "bar");

Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);
```

- Pros: give users maximum flexibility
- Cons: systems cannot provide built-in operators related to vertex data or edge data

Examples of Graph DB Systems

- Systems that focus on graph databases
  - Neo4j
  - Sparksee
  - Titan
  - Infinite Graph
- Multi-model NoSQL databases with support for graphs
  - OrientDB
  - ArangoDB
- Triple stores with Apache TinkerPop support
  - Stardog

Apache TinkerPop

- Graph computing framework
  - Vendor-agnostic
- For graph databases (a graph structure API)
  - Formerly known as Blueprints API
  - Creating and modifying property graphs
  - Example:

```
Graph graph = …
Vertex marko = graph.addVertex(T.label, "person", T.id, 1, "name", "marko", "age", 29);
Vertex vadas = graph.addVertex(T.label, "person", T.id, 2, "name", "vadas", "age", 27);
marko.addEdge("knows", vadas, T.id, 7, "weight", 0.5f);
```

- For graph analytic systems (a process API)
  - Graph-parallel engine
  - Graph traversal/query, based on Gremlin language

Gremlin Graph Traversal (Query) Language

- Part of the Apache TinkerPop framework
- Powerful domain-specific language (DSL) with embeddings in different programming languages
- Expressions specify a concatenation of traversal steps
  - A chain of operations/functions that are evaluated from left to right

Gremlin Examples

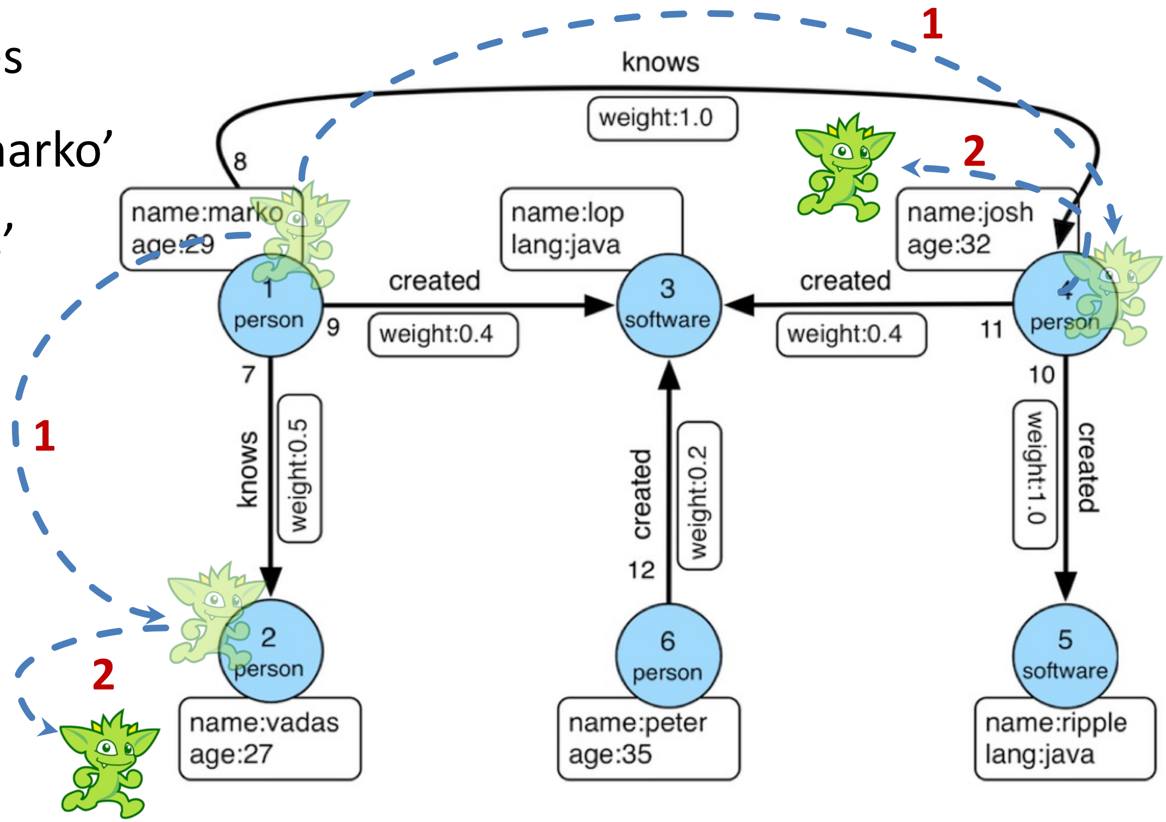
g.V().has('name', 'marko').out('knows').values('name')

Result:

==> vadas

==> josh

- ***g***: for the current graph traversal
- ***V()***: for all vertices in the graph
- ***has('name', 'marko')***: filters the vertices down to those with 'name' property 'marko'
- ***out('knows')***: traverse outgoing 'knows' edges
- ***values('name')***: extracts the values of 'name' property



Gremlin Examples

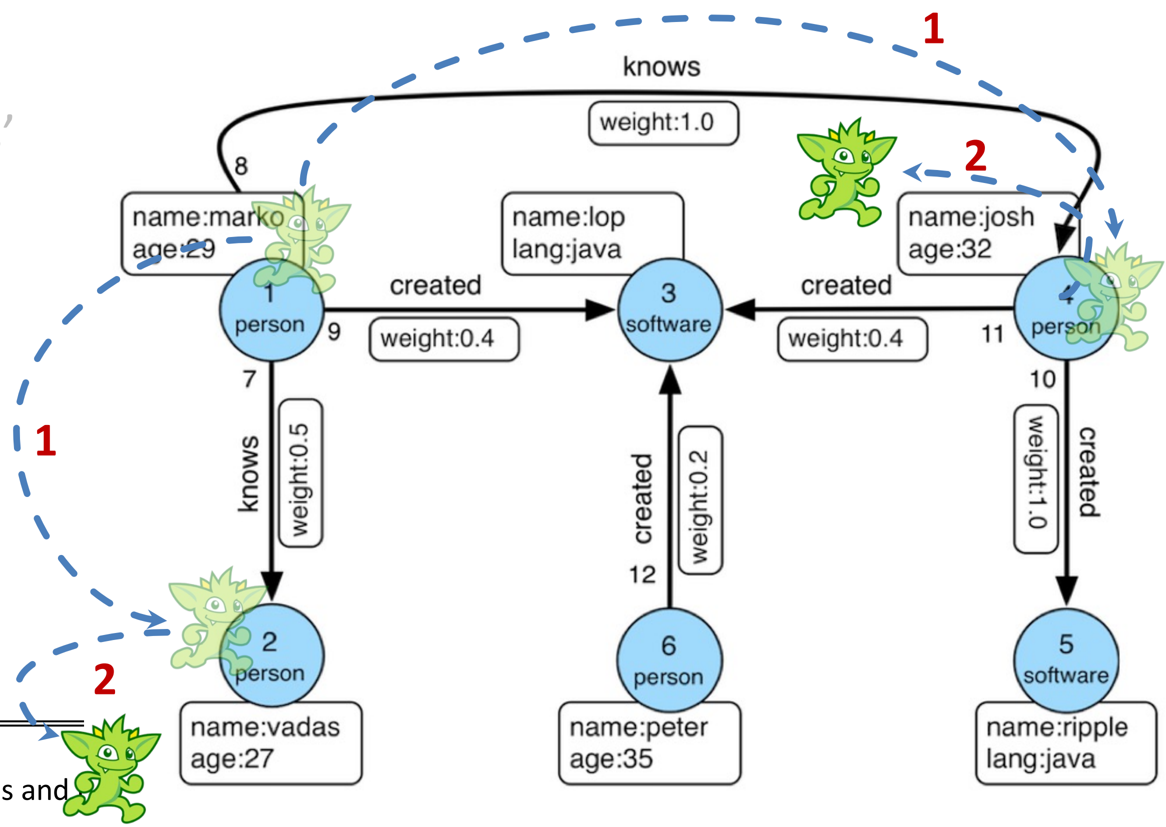
g.V().has(‘name’, ‘marko’).out(‘knows’).values(‘name’).path()

Result:

==> [v[1],v[2],vadas]

==> [v[1],v[4],josh]

- *g*: for the current graph traversal
- *V()*: for all vertices in the graph
- ***has(‘name’, ‘marko’)***: filters the vertices down to those with ‘name’ property ‘marko’
- ***out(‘knows’)***: traverse outgoing ‘knows’ edges
- ***values(‘name’)***: extracts the values of ‘name’ property
- ***path()***: returns the history of the traverser



Gremlin Examples

g.V().has('name', 'marko').repeat(out()).times(2).path().by('name')

Result: ==> [marko, josh, ripple]

==> [marko, josh, lop]

or

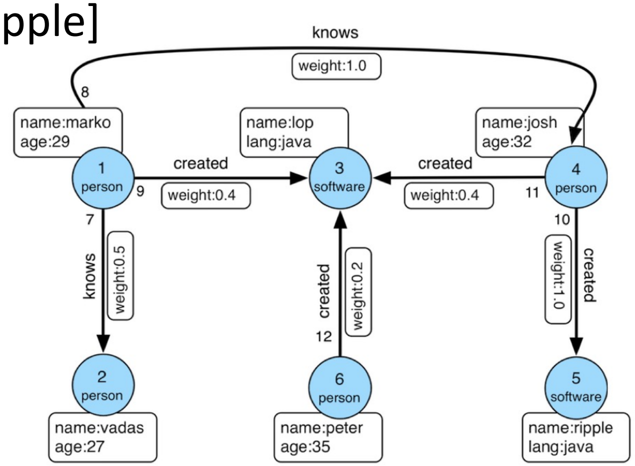
g.V().until('name', 'ripple').repeat(out()).path().by('name')

Result: ==> [marko, josh, ripple]

==> [josh, ripple]

==> [ripple]

- ***times(N)***: the number of traverses (N)
- ***by('name')***: element property projection
- ***repeat()***: loops over a traversal given some break predicate



Cypher

- Declarative graph database query language
- Proprietary (used by Neo4j)
- The OpenCypher project aims to deliver an open specification
- Example
  - Recall our initial Gremlin example

    g.V().has('name', 'marko').out('knows').values('name')
  - In Cypher, we could express this query as follows:

    MATCH({name: 'marko'})-[:knows]->(x)

    RETURN x.name

Possible Clauses in Cypher Queries

- CREATE - creates nodes and edges
- DELETE - removes nodes, edges, properties
- SET - sets values of properties
- MATCH - specifies a *pattern* to match in the data graph
- WHERE - filters pattern matching results
- RETURN - which nodes / edges / properties in the matched data should be returned
- UNION - merges results from two or more queries
- WITH - chains subsequent query parts (like piping in Unix commands)
  - manipulate the output before it is passed on to the following query parts

Node Patterns in Cypher

- Node patterns may have different forms
  ( ) – matches any node
  (:person)-> – matches nodes whose label is person
  ({name: 'marko'}) – matches nodes having a property name='marko'
  (:person {name: 'marko'}) – matches nodes having both the label person and a property name='marko'

- Every node pattern can be assigned a variable
  - Can be used to refer to the matching node in another query clause or to express joins
  - For instance, (x), (x:person)

Relationship Patterns in Cypher

- Relationship pattern must be placed between two node patterns and it may have different forms
  - `-->` or `<--` – matches any edge (with the given direction)
  - `-[:knows]->` – matches edges whose label is knows
  - `-[ {weight:0.5} ]->` – matches edges having a property weight=0.5
  - `-[:knows {weight:0.5} ]->` – matches edges having both the label knows and a property weight=0.5
  - `-[:knows*..4]->` – matches paths of knows edges of up to length 4

- Every relationship pattern can be assigned a variable
  - For instance, `-[x:knows]->`

More complex Cypher Patterns

- Node patterns and relationship patterns are just basic building blocks that can be combined into more complex patterns
  - *MATCH*: searches for an existing node, relationship, label, property, or pattern in the database (like SELECT in SQL).
  - *RETURN*: specifies what values or results you might want to return from a Cypher query.
- Examples:

```
MATCH (a)-[:knows]->()-[:knows]->(a)
RETURN a

MATCH p=shortestPath(
  (:person {name: 'marko'})-[*]->(:person {name:'josh'})
)
RETURN p
```

Filtering in Cypher

- Pattern matching results can be filtered out by using *WHERE* clause
- Examples:
  - MATCH (a:person)-[x:knows]->(b:person)
    WHERE x.weight >0.5 AND x.weight<0.9
    RETURN a, b
  - MATCH ()-[x:knows]->()
    WHERE exists(x.weight)
    RETURN x
  - MATCH (a)-[:knows]->(b)-[x:knows]->(c)
    WHERE NOT (a)-[:knows]->(c)
    RETURN a, b, c

Updating in Cypher

- CREATE, SET, DELETE, REMOVE
- Examples:
  - CREATE (friend:Person {name: 'Mark’})
    RETURN friend
  - MATCH (a:person)-[x:knows]->(b:person)
    SET x.weight = 0.5
    RETURN x
  - MATCH ()-[x:knows]->()
    WHERE NOT exists(x.weight)
    DELETE x
  - MATCH (a:person)-[:knows]->(b)-[x:knows]->(c)
    REMOVE a.organization

Different Graph Data Systems

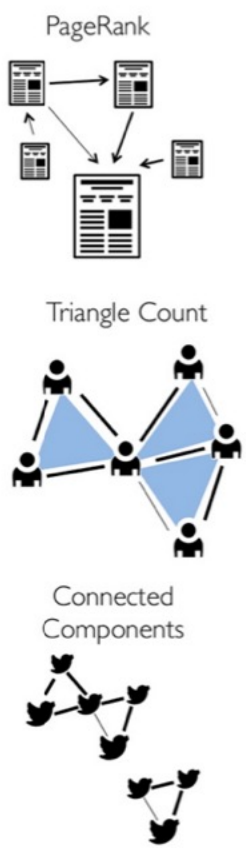
✓ Triple stores
  - Typically, pattern matching queries
  - Data model: RDF

✓ Graph databases
  - Typically, navigational queries
  - Prevalent data model: property graphs

- Graph processing systems
  - Typically, complex graph analysis tasks
  - Prevalent data model: generic graphs

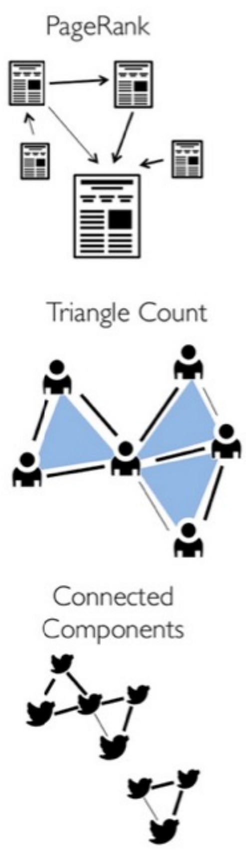
Complex Graph Analysis Tasks?

- Tasks that require an iterative processing of the entire graph or large portions
- Examples
  - Centrality analysis (e.g., PageRank)
  - Clustering, connected components
  - Graph coloring
  - All-pairs shortest path
  - Graph pattern mining (e.g., frequent sub-graphs, community detection)
  - Machine learning



Properties of Computation on Graphs

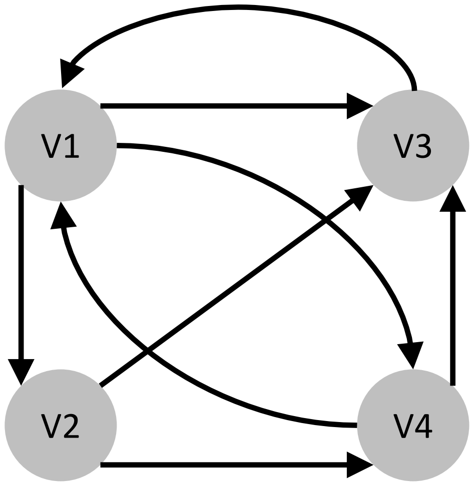
- Dependency graph
  - Dependencies among vertexes
- Local updates
  - The value of a vertex is only influenced by its neighbours
- Iterative Computation
  - E.g., PageRank



PageRank

- Google Search
- A link analysis algorithm
- An algorithm to rank web pages in results from search engine
  - Counting number and quality of links to a page for determining how important a website is

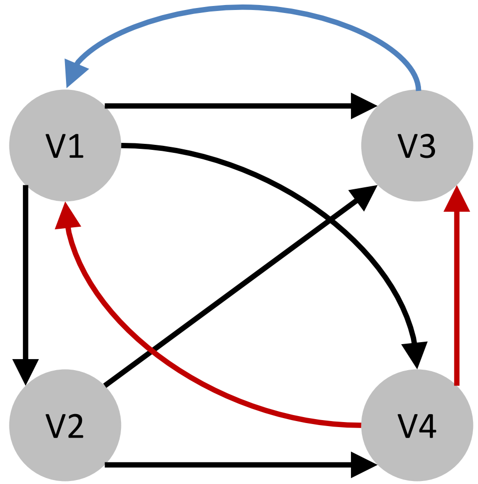
Example: PageRank



$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25						
$PR_k(V_2)$	0.25						
$PR_k(V_3)$	0.25						
$PR_k(V_4)$	0.25						

Example: PageRank

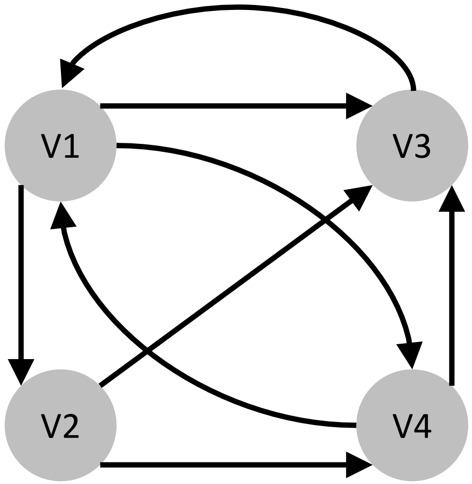


$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN})/|Out(V_{IN})|$$

$$\begin{aligned} PR_2(V1) &= PR_1(V3)/|Out(V3)| + PR_1(V4)/|Out(V4)| \\ &= PR_1(V3)/1 + PR_1(V4)/2 \\ &= 0.25/1 + 0.25/2 \\ &= 0.375 \end{aligned}$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25	0.37					
$PR_k(V_2)$	0.25						
$PR_k(V_3)$	0.25						
$PR_k(V_4)$	0.25						

Example: PageRank



$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

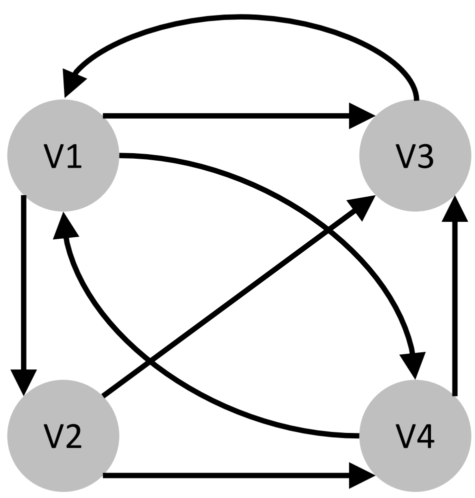
$$\begin{aligned} PR_2(V1) &= PR_1(V3)/|Out(V3)| + PR_1(V4)/|Out(V4)| \\ &= PR_1(V3)/1 + PR_1(V4)/2 \\ &= 0.25/1 + 0.25/2 \\ &= 0.375 \end{aligned}$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25	0.37	0.43	0.45	0.39	0.38	0.38
$PR_k(V_2)$	0.25	0.08	0.12	0.14	0.11	0.13	0.13
$PR_k(V_3)$	0.25	0.33	0.27	0.29	0.29	0.28	0.28
$PR_k(V_4)$	0.25	0.20	0.16	0.20	0.19	0.19	0.19

Convergence

Observation

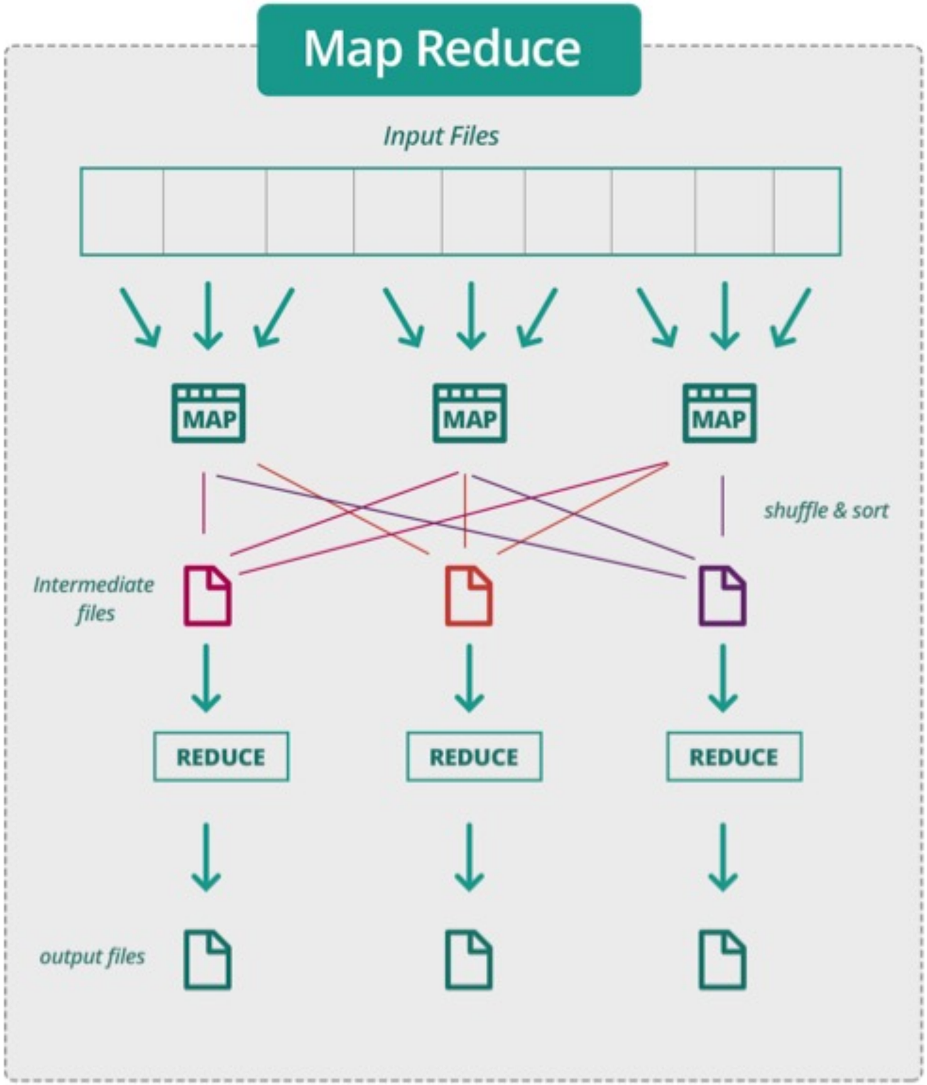
- Many such algorithms iteratively propagate data along the graph structure by transforming intermediate vertex and edge values



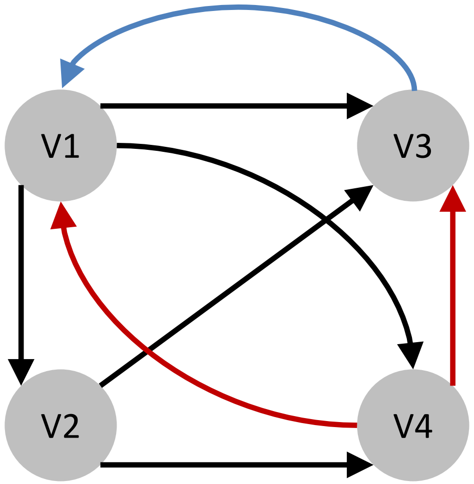
$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

$$\begin{aligned} PR_2(V1) &= PR_1(V3)/|Out(V3)| + PR_1(V4)/|Out(V4)| \\ &= PR_1(V3)/1 + PR_1(V4)/2 \\ &= 0.25/1 + 0.25/2 \\ &= 0.375 \end{aligned}$$

Can we use MapReduce?



Can we use MapReduce?



$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

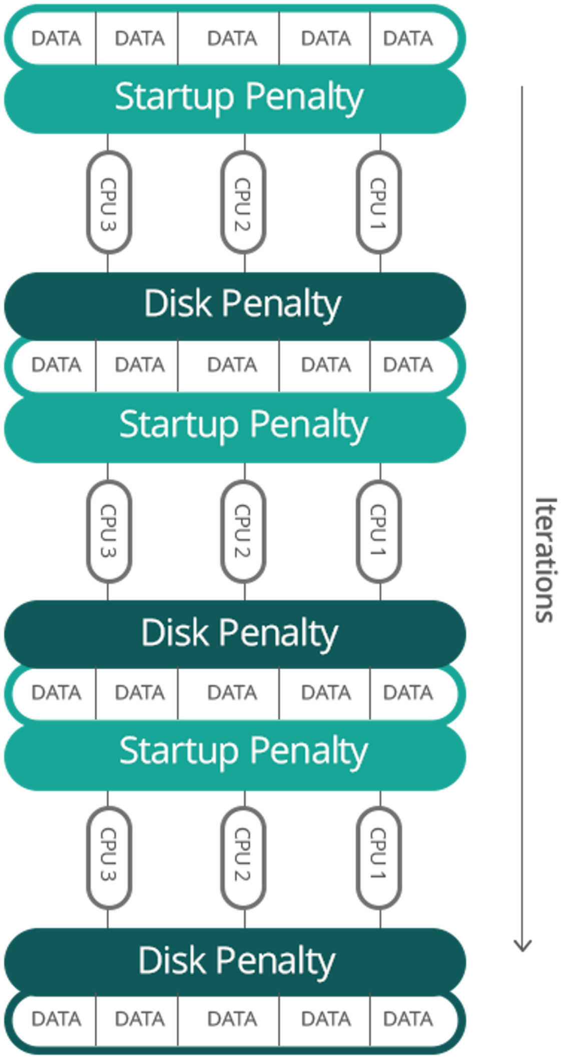
$$\begin{aligned} PR_2(V1) &= PR_1(V3)/|Out(V3)| + PR_1(V4)/|Out(V4)| \\ &= PR_1(V3)/1 + PR_1(V4)/2 \\ &= 0.25/1 + 0.25/2 \\ &= 0.375 \end{aligned}$$

- Map:
  - produces weights of a vertex that assigns to other vertexes e.g., (v3, (0.25,[V1])), (v4, (0.125, [v1,v3]))
  - For iterations, keeps topology information, e.g., (v3, [v1]), (v4, [v1,v3])
  - For checking convergence, keeps vertexes' values, e.g., (v3, 0.25), (v4, 0.25)
- Reduce
  - Handle all the above (3 kinds) information, computes new values and compares with values from last iteration

Can we use MapReduce?

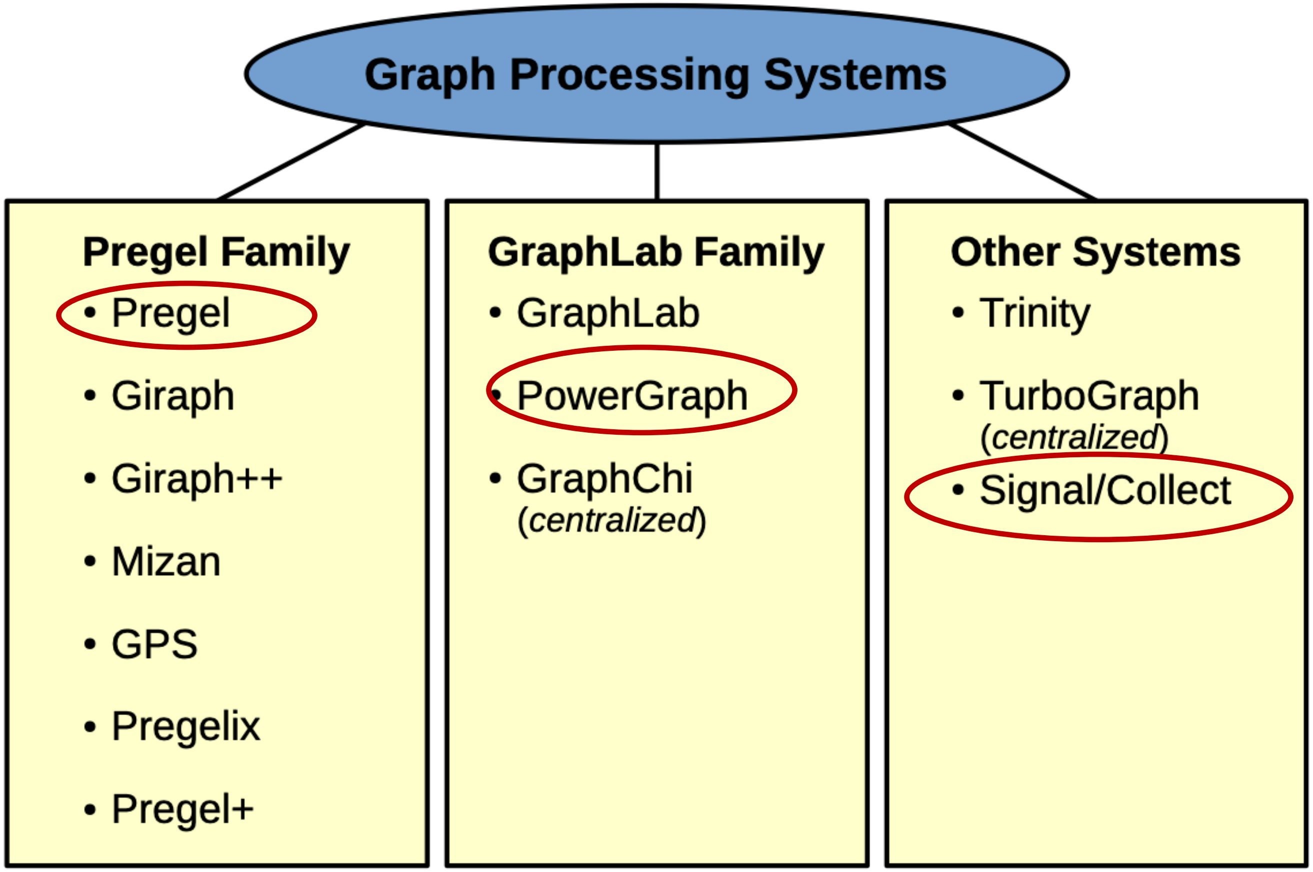
- MapReduce does not directly support iterative algorithms
- Materializing intermediate results at each M/R iteration harms performance
- Extra M/R job on each iteration for checking whether a fixed point has been reached

- Additional issue for graph algorithms
  - Invariant graph-topology data reloaded and reprocessed at each iteration
  - Wastes I/O, CPU, and network bandwidth



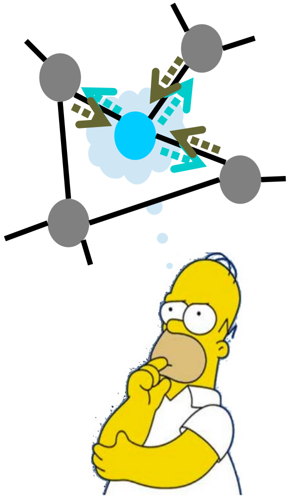
Graph Processing Systems

- Pregel Family
- GraphLab Family
- Other Systems



Vertex-centric Abstraction

- Many such algorithms iteratively propagate data along the graph structure by transforming intermediate vertex and edge values
  - These transformations are defined in terms of functions on the values of adjacent vertexes and edges
  - Hence, such algorithms can be expressed by specifying a function that can be applied to any vertex separately
- "Think like a vertex"



Vertex-centric Abstraction

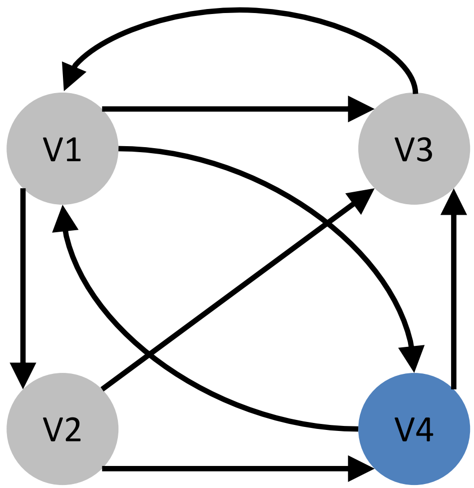
- Vertex compute function consists of three steps:
  1. Read all incoming messages from neighbors
  2. Update the value of the vertex
  3. Send messages to neighbors
- Additionally, the function may “vote to halt” if a local convergence criterion is met

- Overall execution can be parallelized! Terminates when all vertexes have halted and no messages in transit

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may "vote to halt" if a local convergence criterion is met



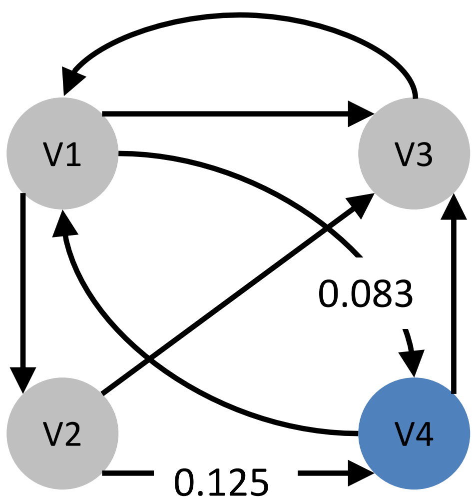
$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25						
$PR_k(V_2)$	0.25						
$PR_k(V_3)$	0.25						
$PR_k(V_4)$	0.25						

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may "vote to halt" if a local convergence criterion is met



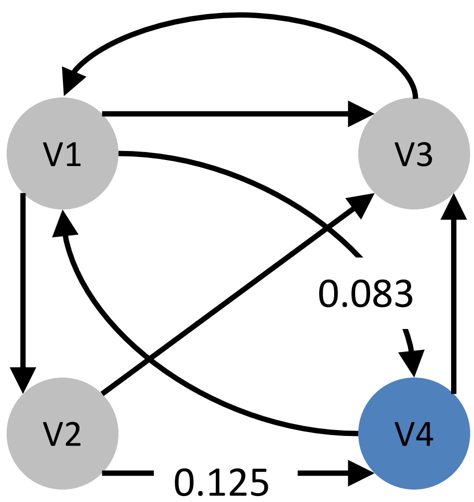
$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25						
$PR_k(V_2)$	0.25						
$PR_k(V_3)$	0.25						
$PR_k(V_4)$	0.25						

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may "vote to halt" if a local convergence criterion is met



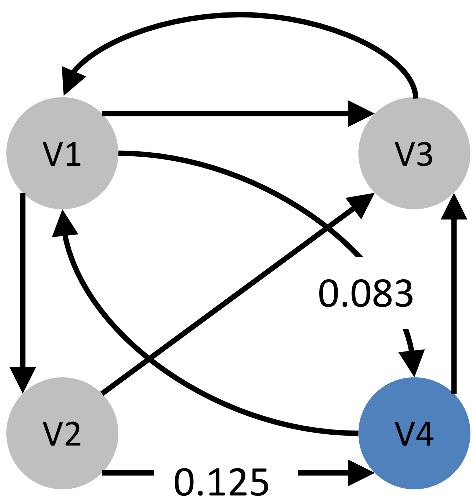
$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25						
$PR_k(V_2)$	0.25						
$PR_k(V_3)$	0.25						
$PR_k(V_4)$	0.25	0.20					

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may “vote to halt” if a local convergence criterion is met



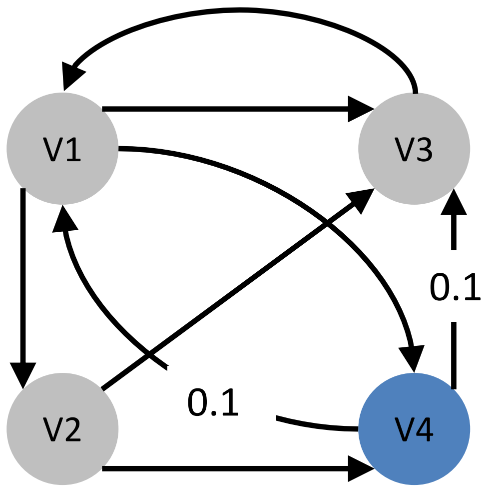
$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN})/|Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25	0.37					
$PR_k(V_2)$	0.25	0.08					
$PR_k(V_3)$	0.25	0.33					
$PR_k(V_4)$	0.25	0.20					

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may "vote to halt" if a local convergence criterion is met



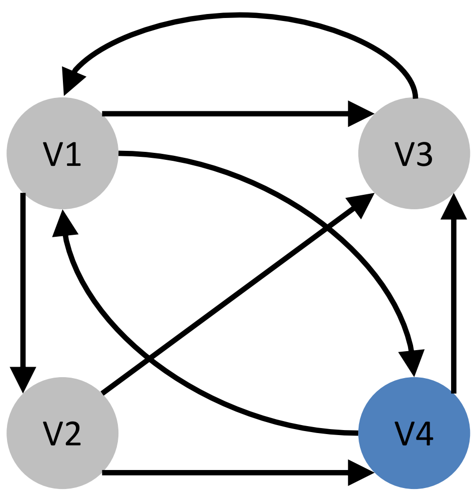
$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25	0.37					
$PR_k(V_2)$	0.25	0.08					
$PR_k(V_3)$	0.25	0.33					
$PR_k(V_4)$	0.25	0.20					

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may "vote to halt" if a local convergence criterion is met



$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

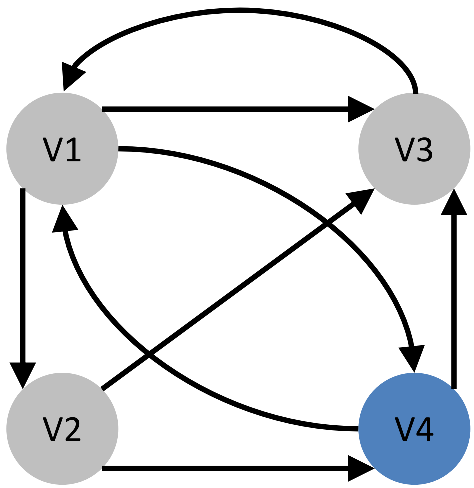
	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25	0.37	0.43	0.35	0.39	0.38	
$PR_k(V_2)$	0.25	0.08	0.12	0.14	0.11	0.13	
$PR_k(V_3)$	0.25	0.33	0.27	0.29	0.29	0.28	
$PR_k(V_4)$	0.25	0.20	0.16	0.20	0.19	0.19	

Local Convergence

Vertex-centric PageRank

1. Read all incoming messages from neighbors
2. Update the value of the vertex
3. Send messages to neighbors

Additionally, the function may "vote to halt" if a local convergence criterion is met

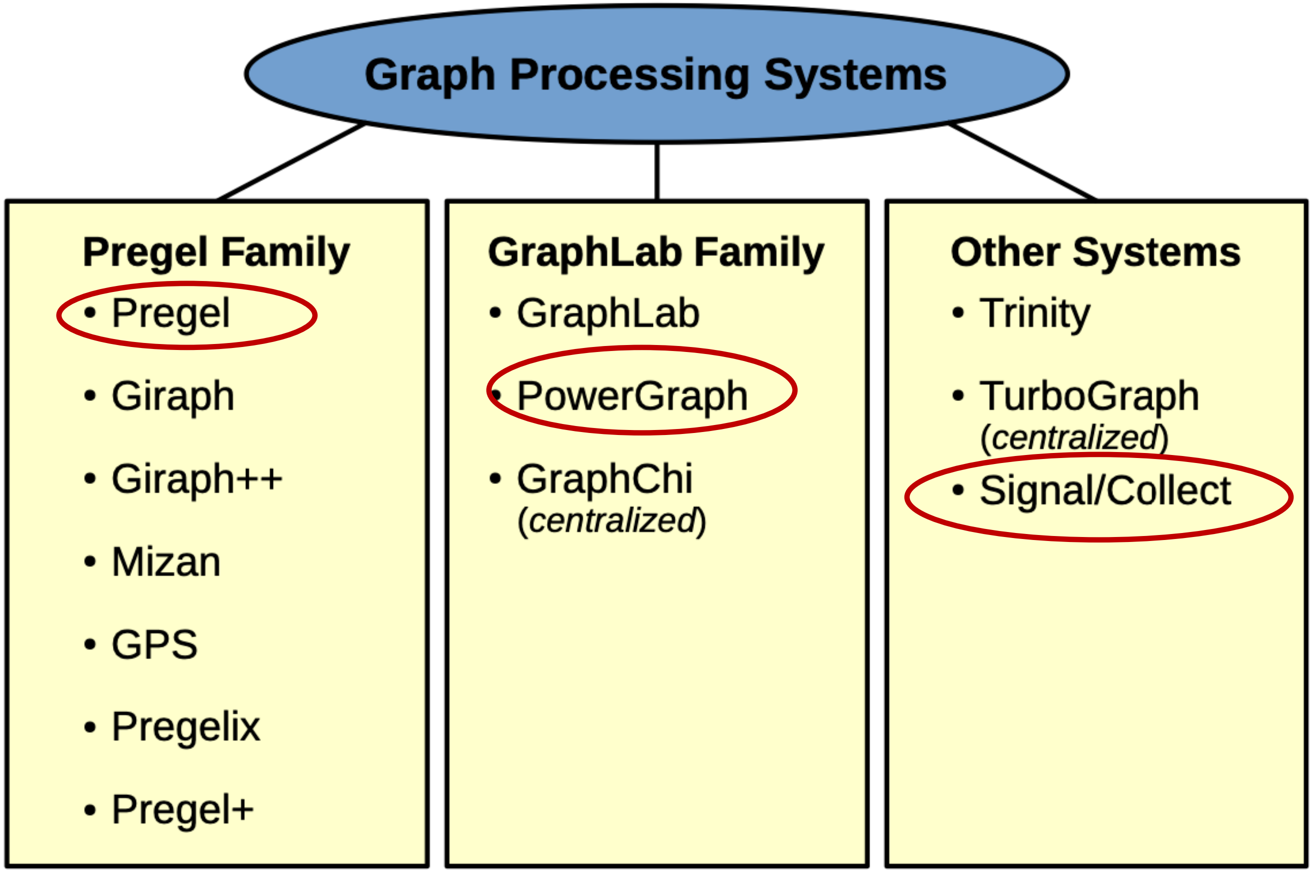


$$PR_{k+1}(V) = \Sigma_{V_{IN}} PR_k(V_{IN}) / |Out(V_{IN})|$$

	k=0	k=1	k=2	k=3	k=4	k=5	k=6
$PR_k(V_1)$	0.25	0.37	0.43	0.35	0.39	0.38	0.38
$PR_k(V_2)$	0.25	0.08	0.12	0.14	0.11	0.13	0.13
$PR_k(V_3)$	0.25	0.33	0.27	0.29	0.29	0.28	0.28
$PR_k(V_4)$	0.25	0.20	0.16	0.20	0.19	0.19	0.19

Graph Processing Systems

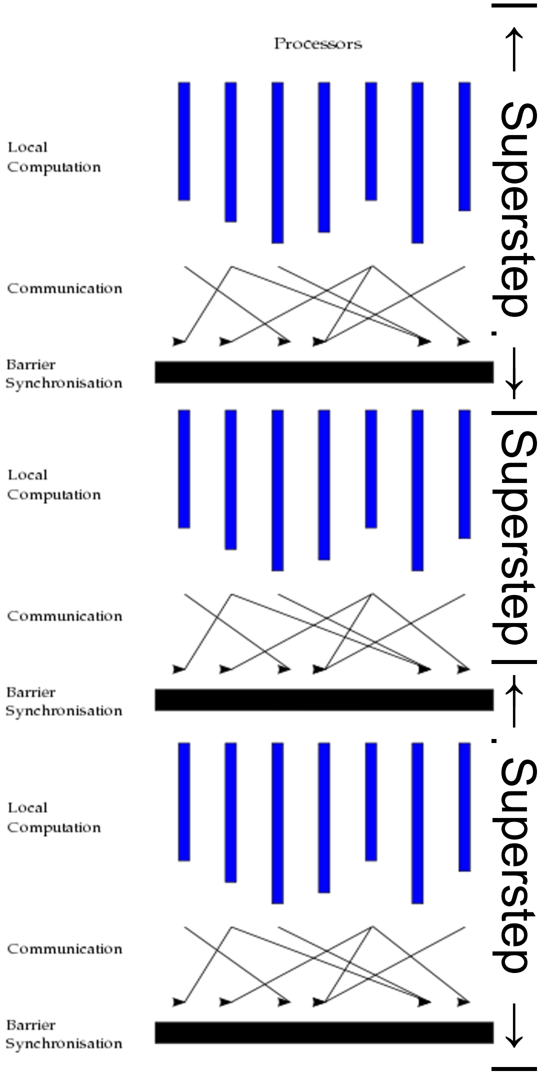
- Pregel Family
- GraphLab Family
- Other Systems



Apache Flink: iterative Graph Processing: https://nightlies.apache.org/flink/flink-docs-release-1.7/dev/libs/gelly/iterative_graph_processing.html

Bulk Synchronous Parallel (BSP)

- Bulk Synchronous Parallel (BSP) programming model
  - A sequence of iterations (each called a superstep)
  - Supersteps with synchronization barriers
  - During a superstep, a user-defined function is invoked for each vertex
- BSP algorithms features
  - Concurrent computation: every participating processor may perform local computations
  - Communication: The processes exchange data to facilitate remote data storage
  - Barrier synchronization: When a process reaches this point (the barrier), it waits until all other processes have reached the same barrier
- Application
  - Google Pregel
  - BSP on top of Hadoop (open project)

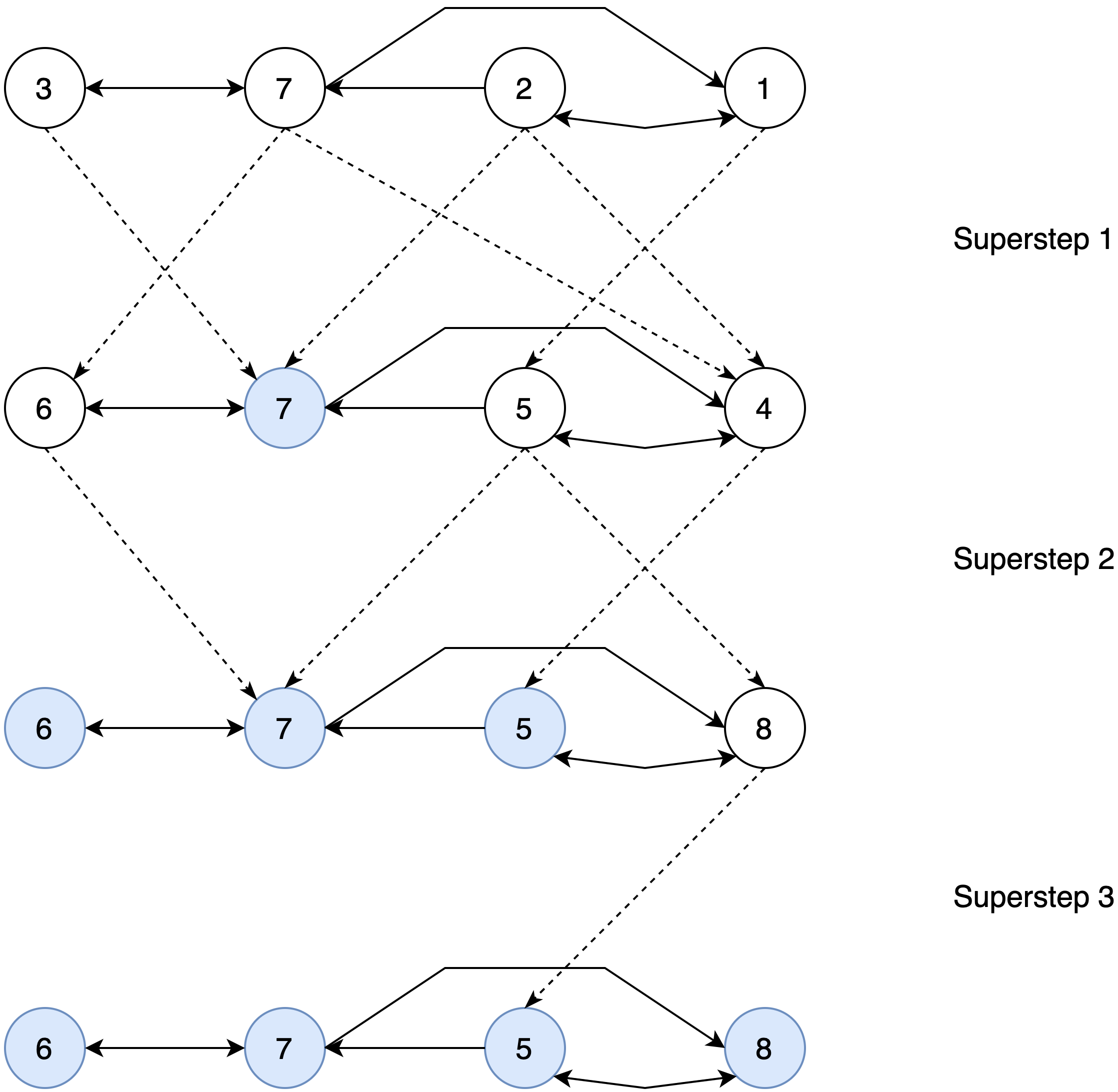


Google Pregel (vertex-centric)

- To solve problems which are difficult to solve using MapReduce

- Each vertex has two statuses:
    - Active and inactive (halt)
- Initially, every vertex is active
- Each vertex sends messages to neighbors
- Within a superstep: after a vertex receives a message, based on its function and criterion, it may need to compute a new value (active) or not (inactive)
- Start next superstep, the computation ends until all vertex are inactive (no need to compute)

Google Pregel



- In each superstep, each vertex executes one user-defined function
- Vertices communicate with other vertices through messages
- A vertex can send a message to any other vertex in the graph, as long as it knows its unique ID
- In each superstep, all active vertices execute the same user-defined computation in parallel

- User only need to define one vertex compute function

MapReduce versus Pregel

MapReduce

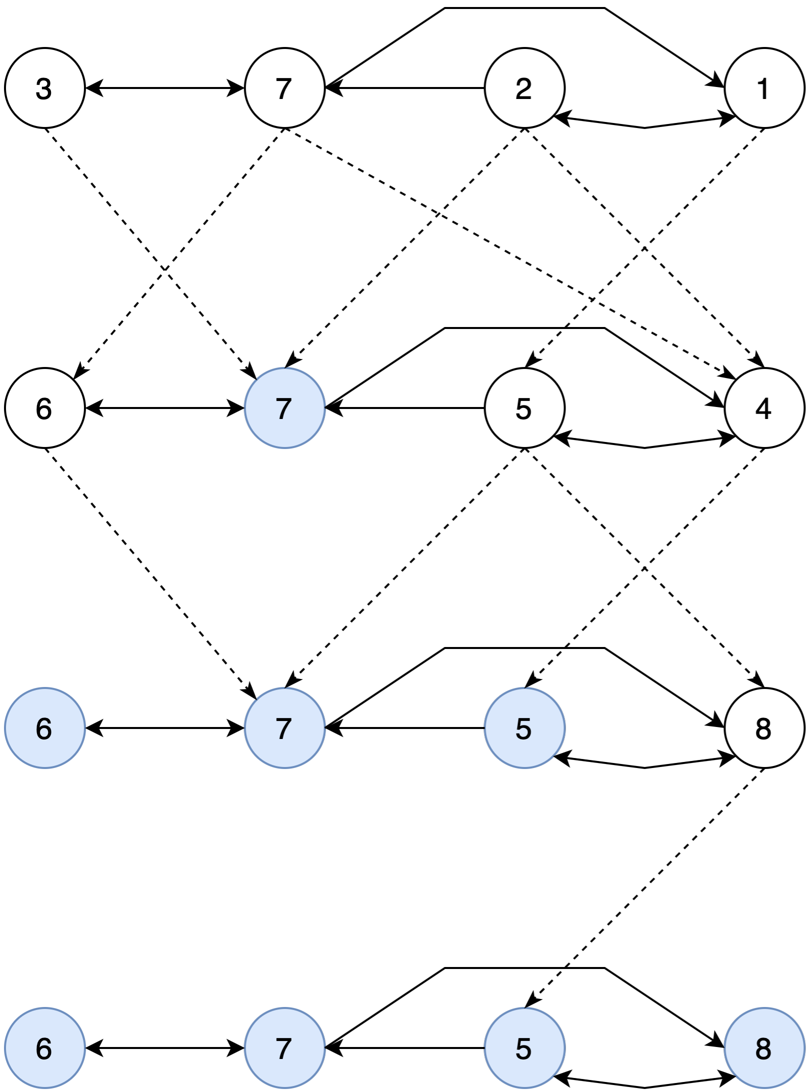
- Requires passing of entire graph topology from one iteration to the next
- Intermediate result after each iteration is stored on disk and then read again from disk
- Programmer needs to write a driver program to support iterations, and another M/R job to check for fixed point

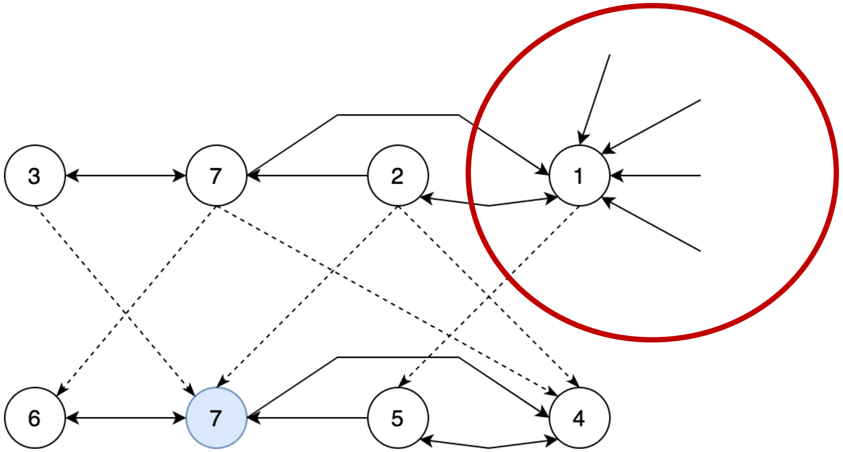
Pregel

- Graph topology is not passed across iterations, vertexes only send their state to their neighbors
- Main memory based

- Usage of supersteps and master-client architecture makes programming easy

Google Pregel

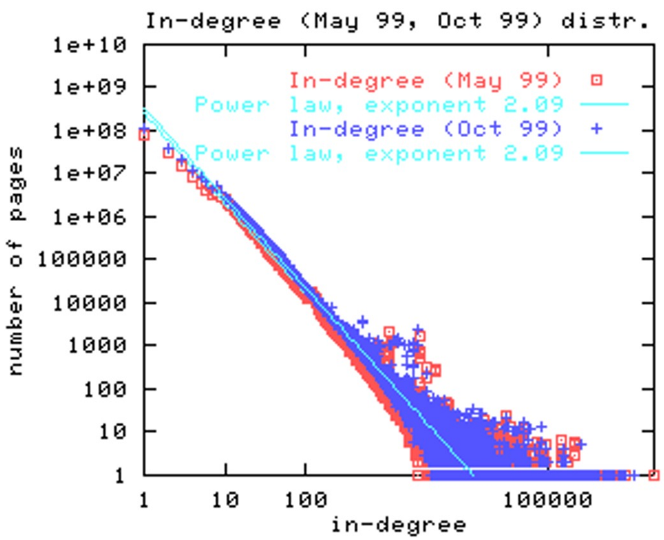


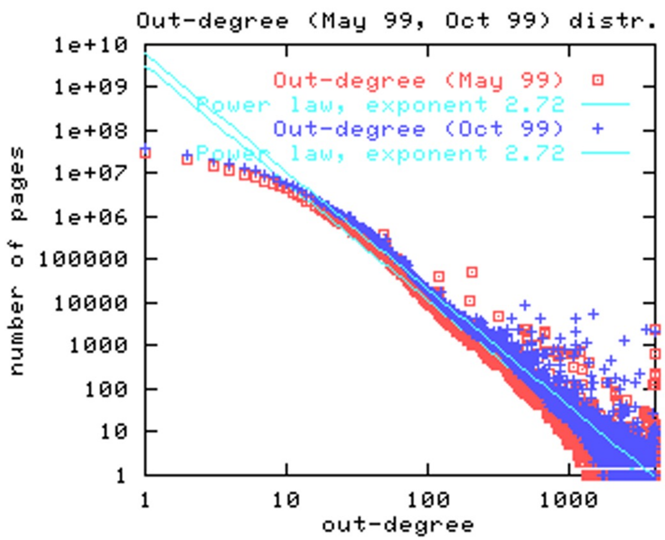


power-law degree distribution!

Google Pregel (BSP) Limitations

- In the BSP (bulk synchronous parallel) model, performance is limited by slowest worker machine
  - Many real-world graphs have power-law degree distribution, which may lead to few highly-loaded workers
  - A single vertex has more out-edges than in-edges, or vice versa





Possible optimizations to balance the workload

- Decompose the vertex program
- Sophisticated graph partitioning

- Graph-centric abstraction
- Asynchronous execution (instead of BSP)

Possible optimizations to balance the workload

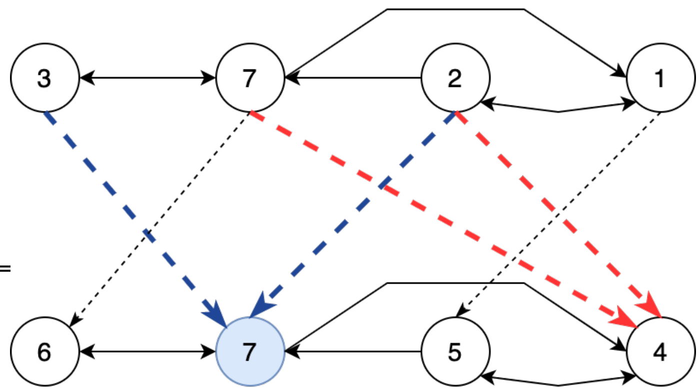
- Decompose the vertex program
- Sophisticated graph partitioning

- Graph-centric abstraction
- Asynchronous execution (instead of BSP)

Combiner

- Takes two messages and combines them into one associative, commutative function
- Can be used to aggregate messages before sending them to the worker node that has the target vertex

- Example:
  - In the vertex-centric PageRank, messages are values $m_{IN} = \left(\frac{PR_k(v_{IN})}{|Out(v_{IN})|}\right)$ of each incoming neighbor v_{IN}.
  - In the vertex function these values are summed up: $\left(\frac{PR_k(v_{IN1})}{|Out(v_{IN1})|} + \frac{PR_k(v_{IN2})}{|Out(v_{IN2})|} + \cdots\right)$
  - Parts of this sum may be computed by worker nodes that have some of the incoming neighbor vertexes



Signal/Collect Model

- Also known as Scatter-Gather Iterations, vertex-centric
- Scatter/Signaling (edge function):
    - Every edge uses the value of its source vertex to compute a message (“signal”) for the target vertex
    - Executed on the worker that has the source vertex
    - Main task: produces the messages that a vertex will send to other vertices
- Gather/Collecting (vertex function):
    - Every vertex computes its new value based on the messages received from its incoming edges
    - Executed on the worker that has the target vertex
    - Main task: updates the vertex value using received messages

Pregel vs Scatter-Gather

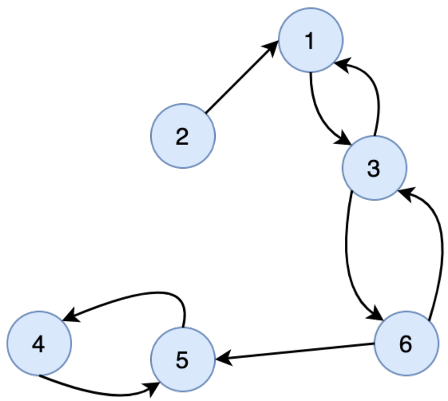
- Similarities
  - Vertex-centric
  - Pregel, Scatter-Gather, parallelism based on vertex computations
- Differences
  - In Pregel, user defines one single vertex compute function
  - In Scatter-Gather, user defines two functions
    - Scatter function for sending messages
    - Gather function for updating values
  - Scatter-Gather decouples sending messages and updating values
    - Easy to maintain

Possible optimizations to balance the workload

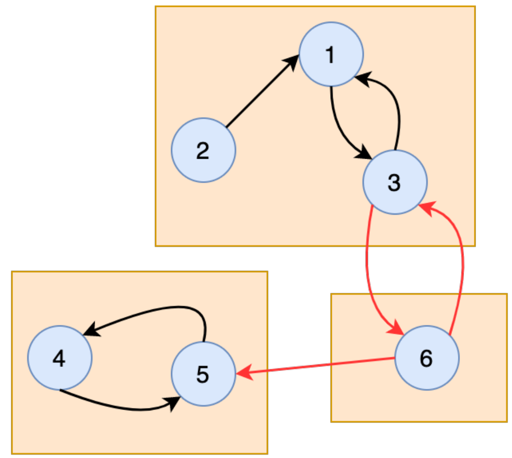
- Decompose the vertex program
- Sophisticated graph partitioning

- Graph-centric abstraction
- Asynchronous execution (instead of BSP)

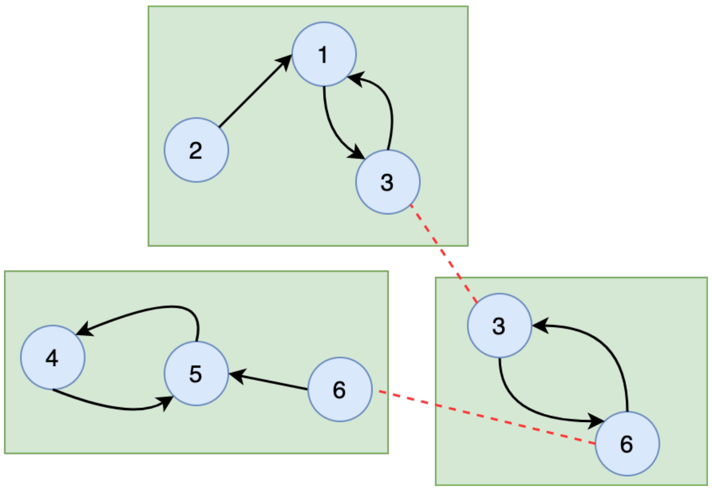
Graph Partitioning



Original graph

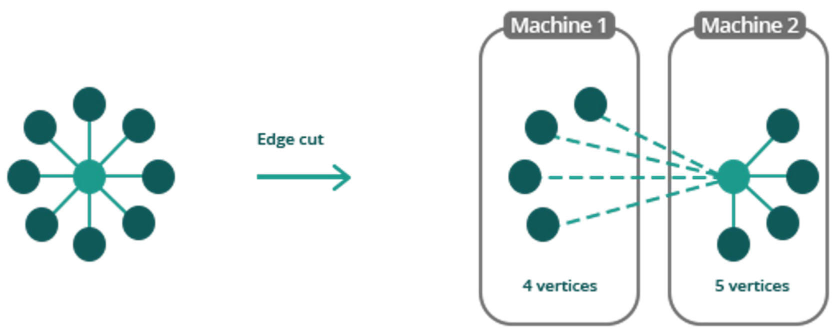


Vertex partitioning/Edge-cut



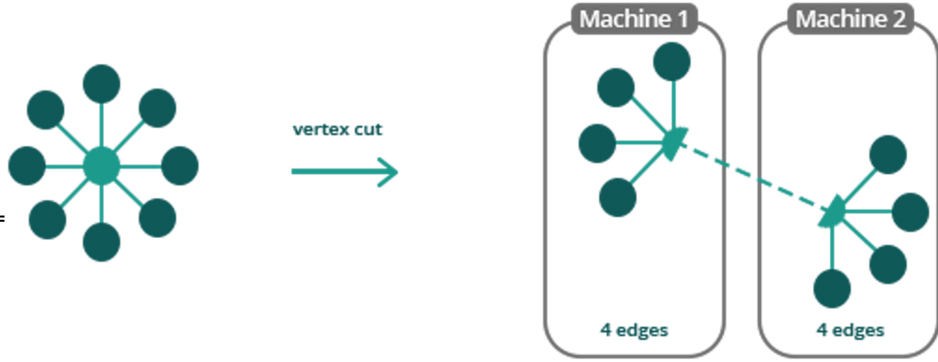
Edge partitioning/Vertex-cut

- The goals of graph partitioning
  - Load balancing, to decrease memory usage
  - Minimize cuts, to decrease communications
- Unfortunately, the problem is NP-complete
- Various heuristics and approximation algorithms



Vertex-Cut

- PowerGraph, a framework for large-scale machine learning and graph computation
- PowerGraph introduced a partitioning scheme that “cuts” vertexes such that the edges of high-degree vertexes are handled by multiple workers
  - improved work balance
- Power-law graphs (some node has a large number of edges) have good vertex-cuts
  - Communication is linear in the number of machines each vertex spans
  - Vertex-cut minimizes this number
  - Hence, reduced network traffic



Summary

- NoSQL Data Models
  - Key-value model
  - Document model
  - Wide-Column model
  - Graph Data Model
    - Graph Processing for generic graphs

www.liu.se