# Large Scale Graph Processing

**D. Tsoumakos**

Αναλυτική Μεγάλων Δεδομένων

ΠΜΣ Ψηφιακές Εφαρμογές

Department of Informatics

Ionian University

**Από παρουσίαση Sebastian Schelter,** Invited talk at GameDuell Berlin, 29th May 2012

**Από παρουσίαση μαθήματος** CS 347: Parallel and Distributed Data Management, Stanford, H.G. Molina

IONIAN UNIVERSITY

DEPARTMENT OF INFORMATICS

# Overview

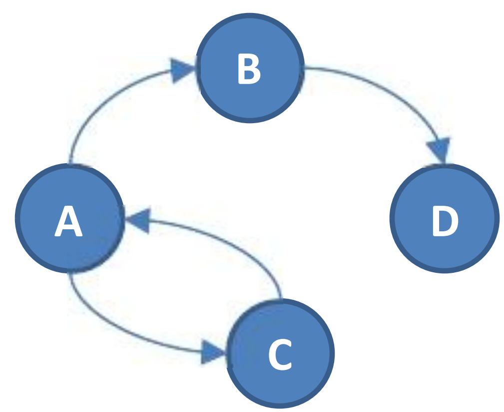
1) **Graphs**
2) Graph processing with Hadoop/MapReduce
3) Google Pregel

# Graphs

**graph**: abstract representation of a set of objects (**vertices**), where some pairs of these objects are connected by links (**edges**), which can be directed or undirected

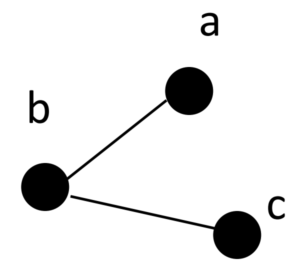
Graphs can be used to model arbitrary things like road networks, social networks, flows of goods, etc.

Majority of graph algorithms are iterative and traverse the graph in some way

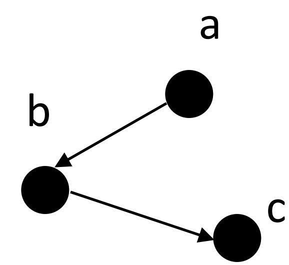


# What is a graph

- Formally: A finite graph $G(V, E)$ is a pair $(V, E)$, where $V$ is a finite set and $E$ is a *binary relation* on $V$.
  - Recall: A *relation R* between two sets $X$ and $Y$ is a subset of $X \times Y$.
  - For each selection of two distinct $V$'s, that pair of $V$'s is either in set $E$ or not in set $E$.
- The elements of the set $V$ are called *vertices* (or *nodes)* and those of set $E$ are called *edges*.
- **Undirected graph**: The edges are unordered pairs of $V$ (i.e. the binary relation is symmetric).
  - Ex: undirected G(V,E); V = {a,b,c}, E = {{a,b}, {b,c}}



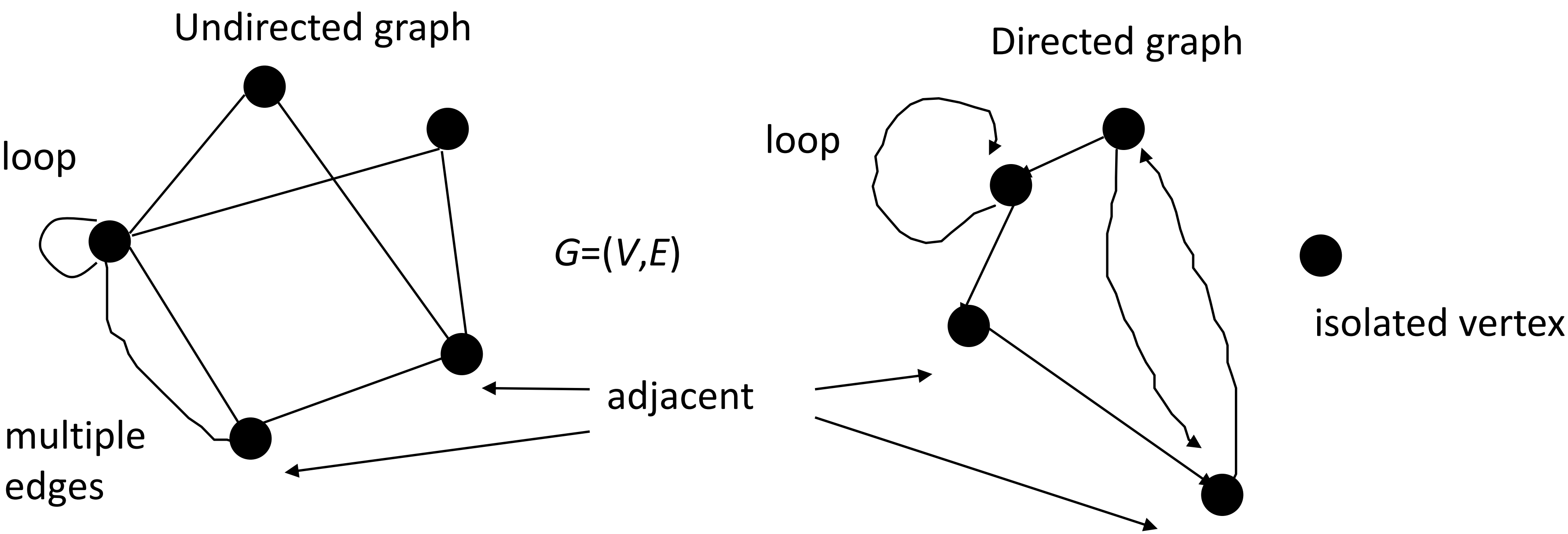

- **Directed graph** (digraph):The edges are ordered pairs of $V$ (i.e. the binary relation is not necessarily symmetric).
  - Ex: digraph G(V,E); V = {a,b,c}, E = {(a,b), (b,c)}



# Why graphs?

- Many problems can be stated in terms of a graph
- The properties of graphs are well-studied
  - Many algorithms exists to solve problems posed as graphs
  - Many problems are already known to be intractable
- By *reducing* an instance of a problem to a standard graph problem, we may be able to use well-known graph algorithms to provide an optimal solution
- Graphs are excellent structures for storing, searching, and retrieving large amounts of data
  - Graph theoretic techniques play an important role in increasing the storage/search efficiency of computational techniques.

# Basic definitions

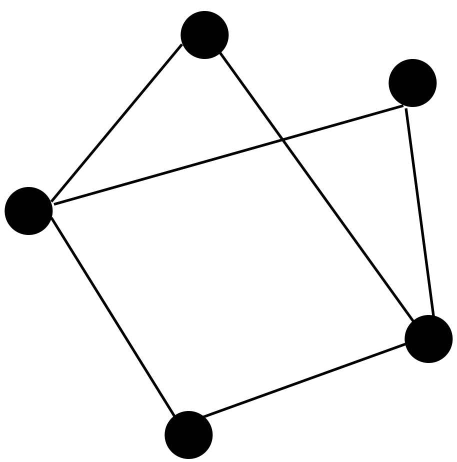


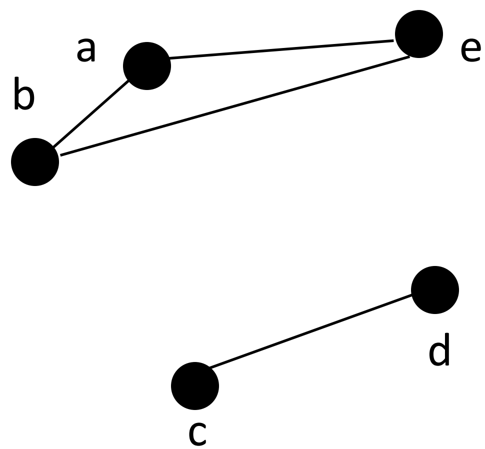

- **incidence**: an edge (directed or undirected) is incident to a vertex that is one of its end points.
- **degree** of a vertex: number of edges incident to it
  - Nodes of a digraph can also be said to have an **indegree** and an **outdegree**
- **adjacency**: two vertices connected by an edge are adjacent

# Types of graphs

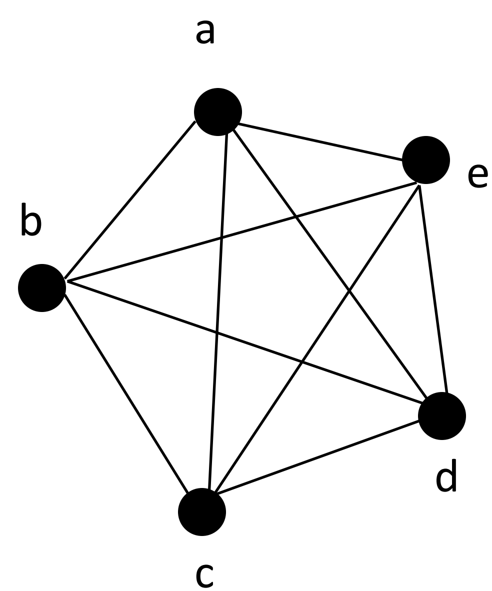
- **simple graph:** an undirected graph with no loops or multiple edges between the same two vertices
- **multi-graph:** any graph that is not simple
- **connected graph**: all vertex pairs are joined by a path
- **disconnected graph**: at least one vertex pairs is not joined by a path
- **complete graph**: all vertex pairs are adjacent
  - $K_n$: the completely connected graph with *n* vertices

Simple graph





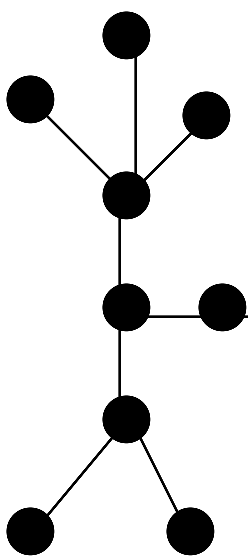

Disconnected graph with two components

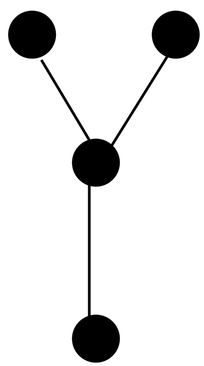


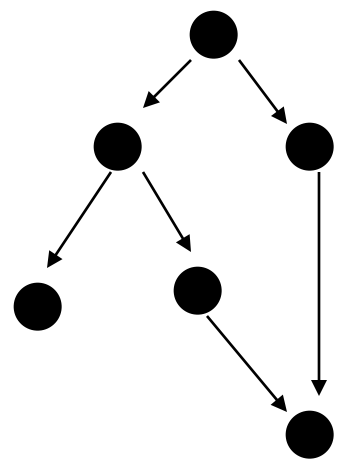

$K_5$

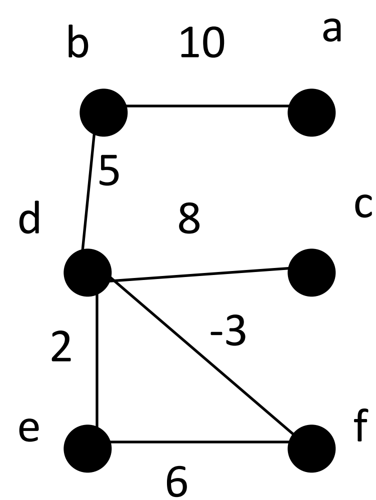
# Types of graphs (2)

- **acyclic graph** (forest): a graph with no cycles
- **tree:** a connected, acyclic graph
- **rooted tree**: a tree with a “root” or “distinguished” vertex
  - **leaves:** the terminal nodes of a rooted tree
- **directed acyclic graph** (DAG): a digraph with no cycles
- **weighted graph:** any graph with weights associated with the edges (edge-weighted) and/or the vertices (vertex-weighted)

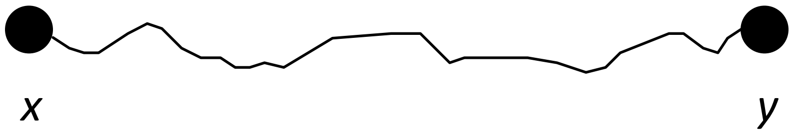


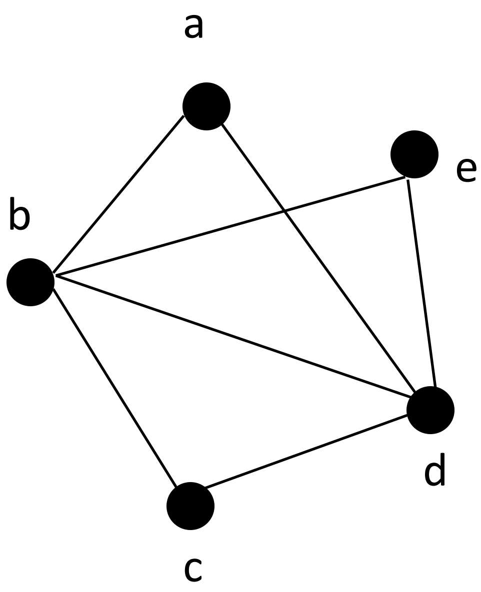






# "Travel" in graphs




**path**: no vertex can be repeated
example path: a-b-c-d-e
**trail**: no edge can be repeated
example trail: a-b-c-d-e-b-d
**walk**: no restriction
example walk: a-b-d-a-b-c

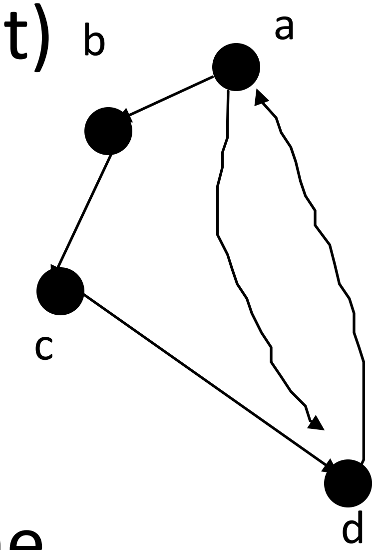
**closed:** if starting vertex is also ending vertex
**length**: number of edges in the path, trail, or walk

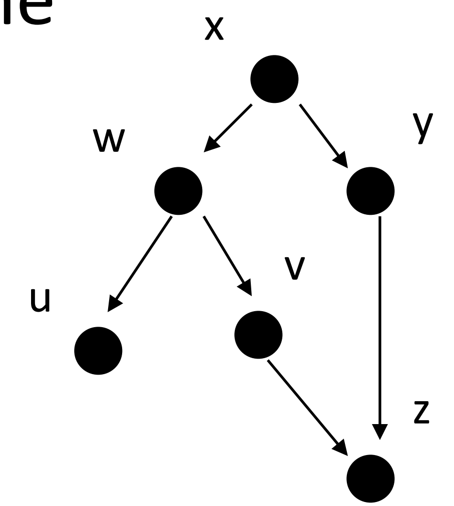
**circuit:** a closed trail (ex: a-b-c-d-b-e-d-a)
**cycle:** closed path (ex: a-b-c-d-a)

# Digraph definitions

- **for digraphs only…**
- Every edge has a **head** (starting point) and a **tail** (ending point)
- Walks, trails, and paths can only use edges in the appropriate direction
- In a DAG, every path connects an **predecessor/ancestor** (the vertex at the head of the path) to its **successor/descendents** (nodes at the tail of any path).
- **parent:** direct ancestor (one hop)
- **child:** direct descendent (one hop)
- A descendent vertex is **reachable** from any of its ancestors vertices

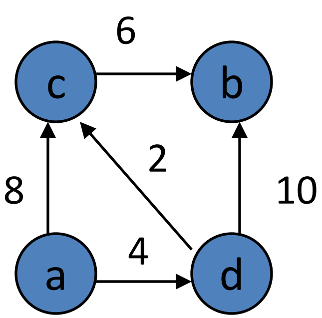
Directed graph



# Computer representation

- **adjacency matrix:** a $|V| \times |V|$ array where each cell *i,j* contains the weight of the edge between $v_i$ and $v_j$ (or 0 for no edge)
- **adjacency list:** a |V| array where each cell *i* contains a list of all vertices adjacent to $v_i$
- **incidence matrix:** a |V| by |E| array where each cell *i,j* contains a weight (or a defined constant HEAD for unweighted graphs) if the vertex i is the head of edge *j* or a constant TAIL if vertex I is the tail of edge *j*




| | a | b | c | d |
|---|---|---|---|---|
| a | | | 8 | 4 |
| b | | | | |
| c | | 6 | | |
| d | | 10 | 2 | |

adjacency matrix

| | |
|---|---|
| a | c (8), d (4) |
| b | |
| c | b (6) |
| d | c (2), b (10) |

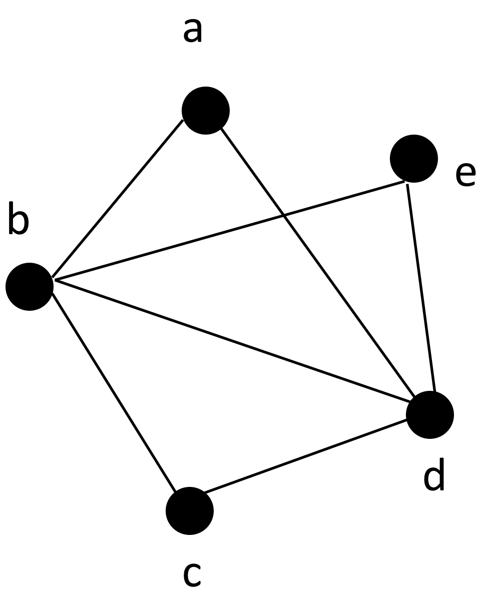
adjacency list

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| a | | 8 | | | 4 |
| b | t | | | t | |
| c | 6 | t | t | | |
| d | | | 2 | 10 | t |

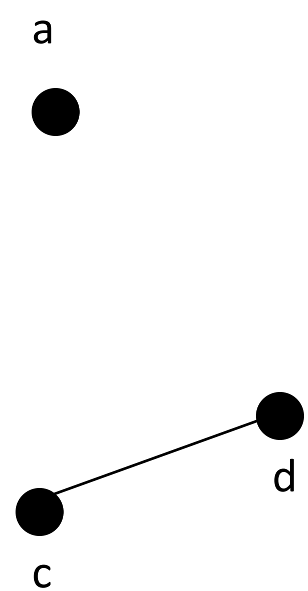
incidence matrix

# Subgraphs

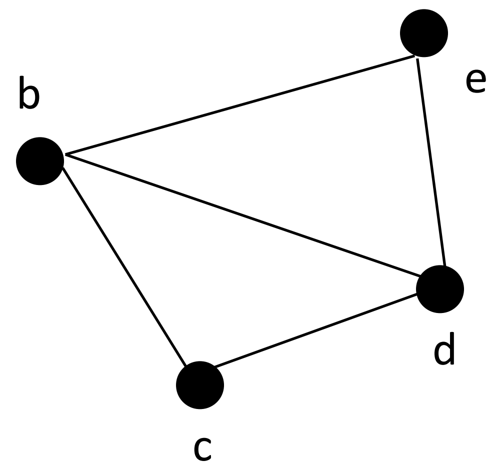
- $G'(V',E')$ is a **subgraph** of $G(V,E)$ if $V' \subseteq V$ and $E' \subseteq E$.
- **induced subgraph:** a subgraph that contains all possible edges in E that have end points of the vertices of the selected V'




G(V,E)



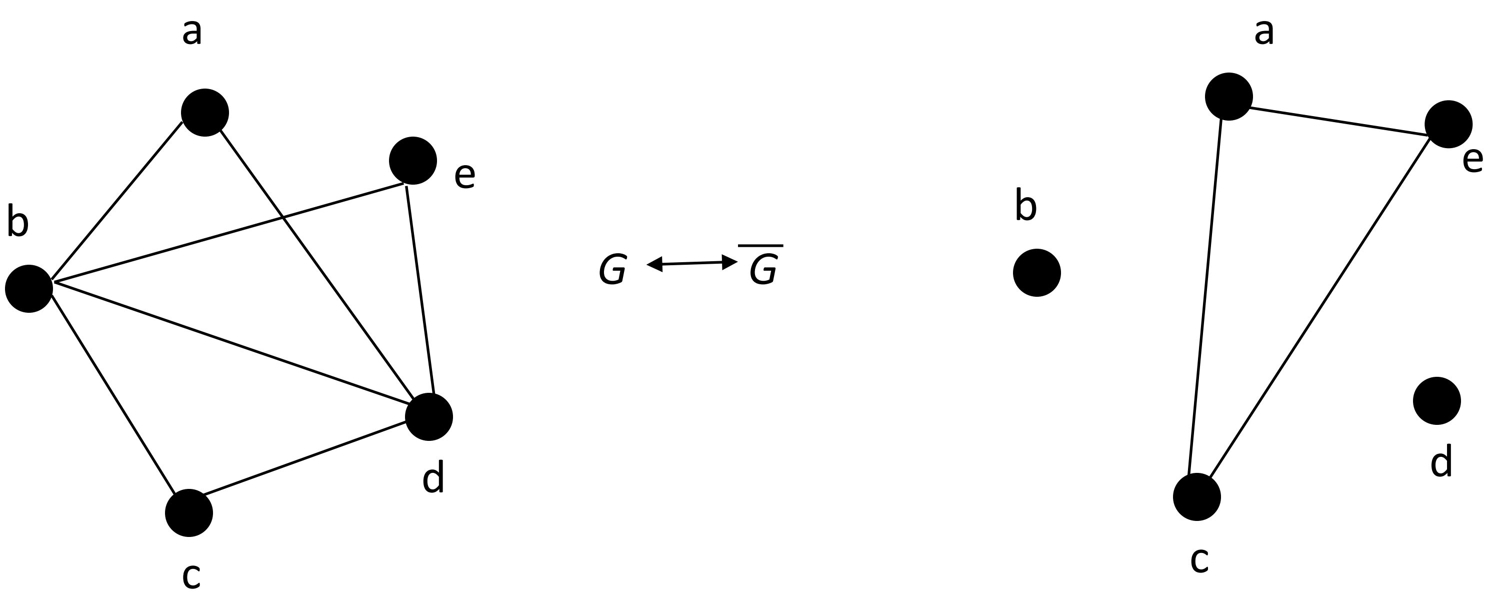

G'({a,c,d},{{c,d}})




Induced subgraph of G with V' = {b,c,d,e}

# Complement of a graph

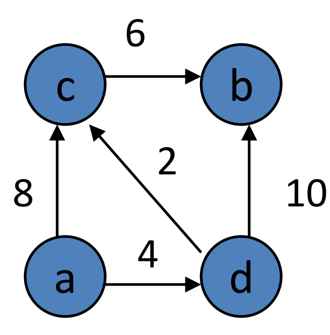
- The **complement** of a graph G (*V*,*E*) is a graph with the same vertex set, but with vertices adjacent only if they were not adjacent in *G*(*V*,*E*)



# Famous problems: Shortest path

- Consider a weighted connected directed graph with a distinguished vertex

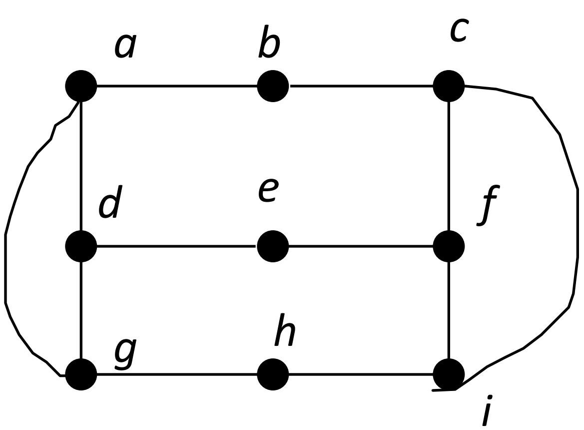
  **source:** a distinguished vertex with zero in-degree
- What is the path of total minimum weight from the source to any other vertex?
- Greedy strategy works for simple problems (no cycles, no negative weights)
- Longest path is a similar problem (complement weights)

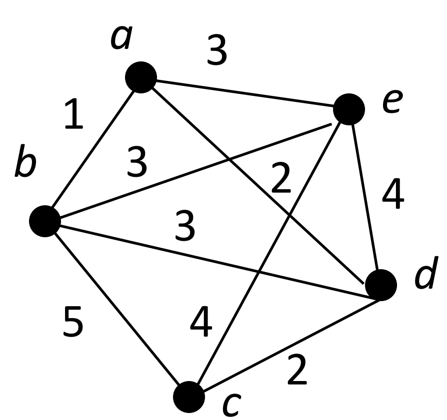


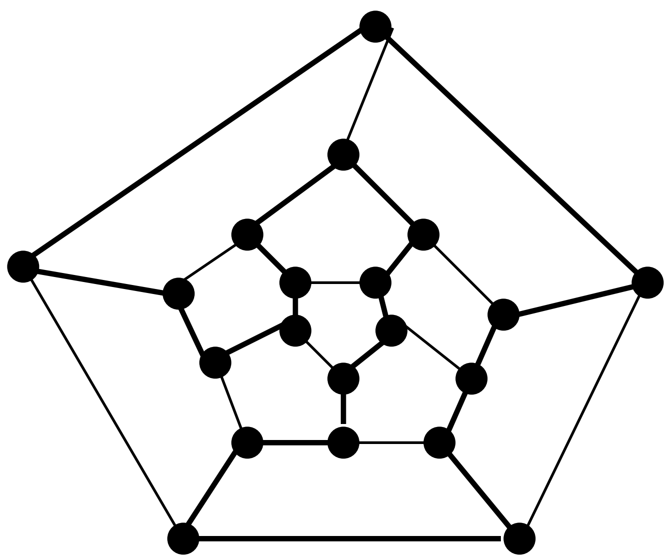
# Famous problems: Hamilton & TSP

- **Hamiltonian path:** a path through a graph which contains every vertex exactly once
- Finding a Hamiltonian path is another NP-complete problem…

- **Traveling Salesmen Problem (TSP):** find a Hamiltonian path of minimum cost

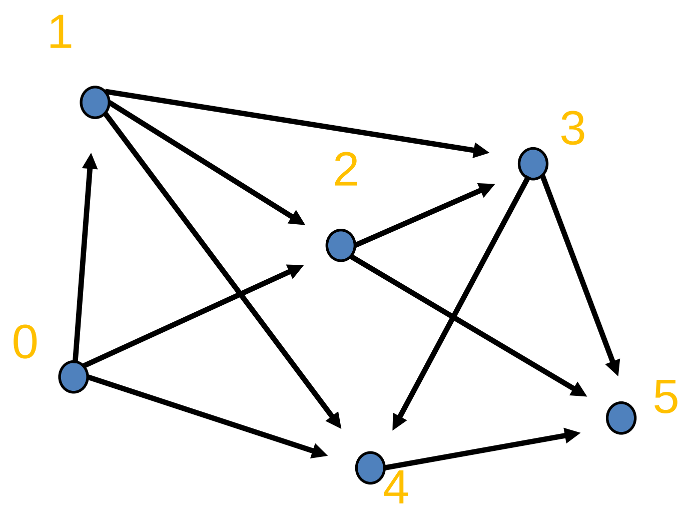


# Topological Sort

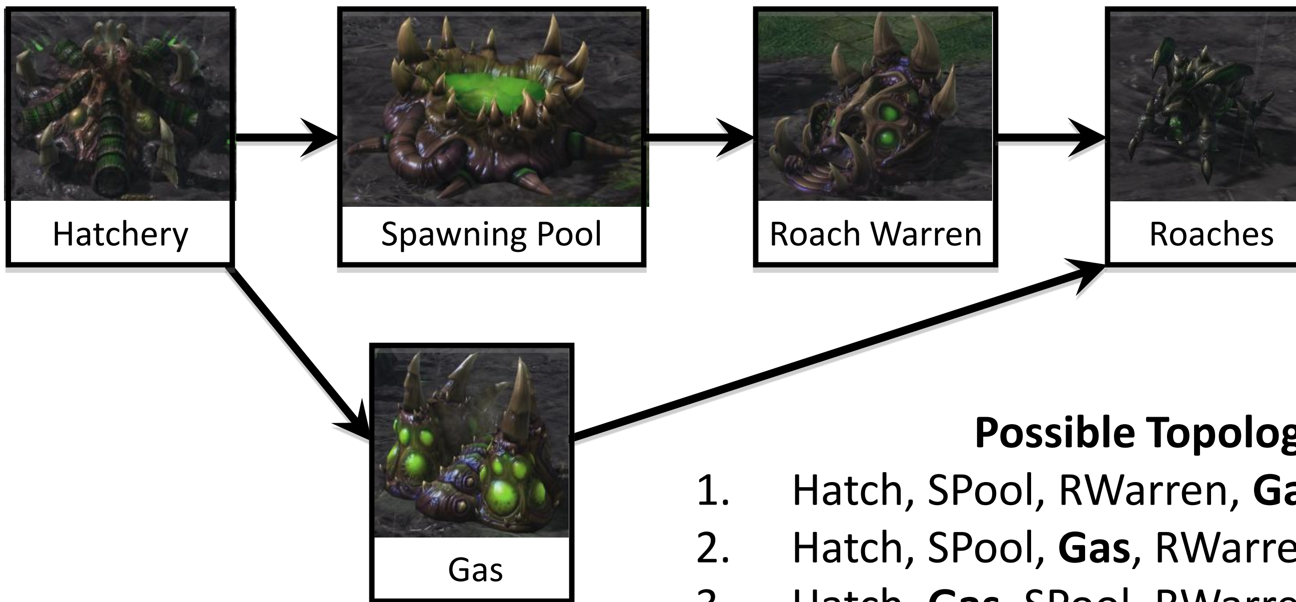
- We just computed a **topological sort** of the dag
  - This is a numbering of the vertices such that all edges go from lower- to higher-numbered vertices




- Useful in job scheduling with precedence constraints

# Example of Topological Sort

- Starcraft II build order: Roach Rush

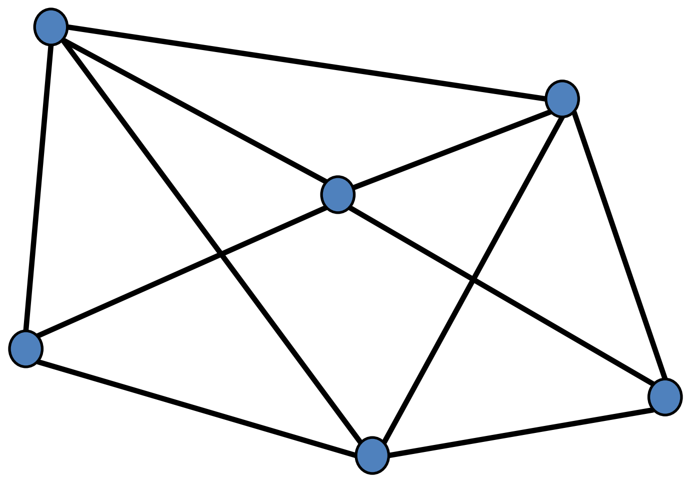



**Possible Topological Sorts**

1. Hatch, SPool, RWarren, **Gas**, Roaches
2. Hatch, SPool, **Gas**, RWarren, Roaches
3. Hatch, **Gas**, SPool, RWarren, Roaches

# Graph Coloring

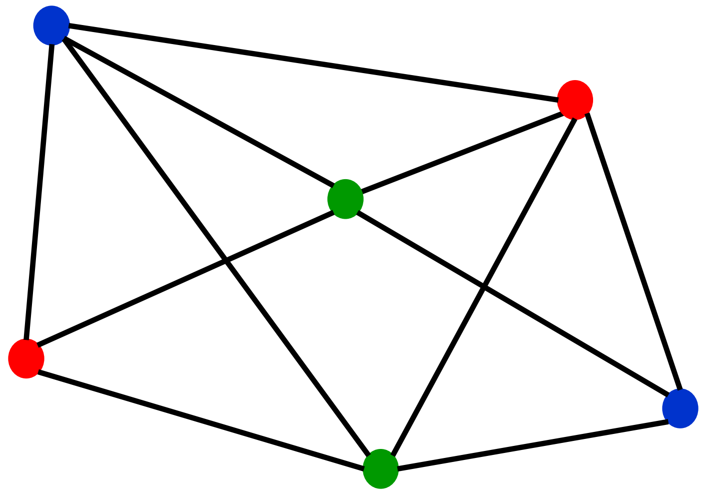
- A **coloring** of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



- **chromatic number**: the smallest number of labels for a coloring of a graph
- How many colors are needed to color this graph?

# Graph Coloring

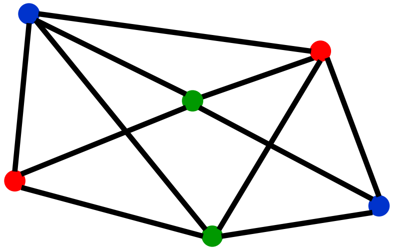
- A **coloring** of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color



- How many colors are needed to color this graph?

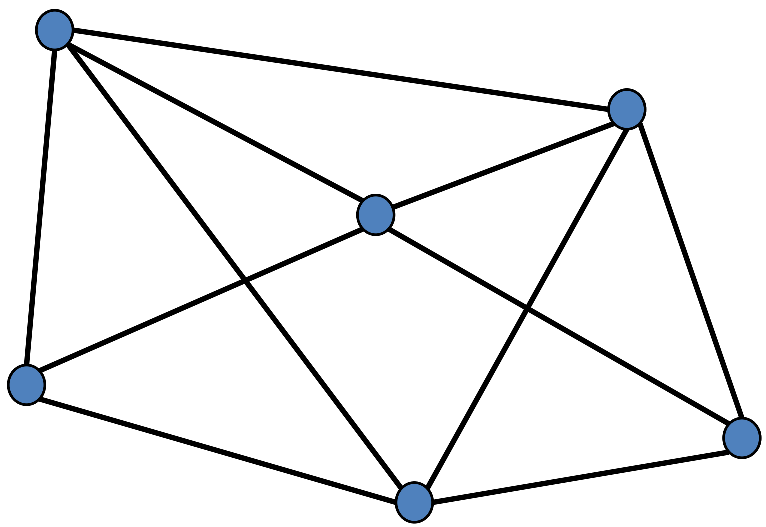
# An Application of Coloring

- Vertices are jobs
- Edge (u,v) is present if jobs u and v each require access to the same shared resource, and thus cannot execute simultaneously
- Colors are time slots to schedule the jobs
- Minimum number of colors needed to color the graph = minimum number of time slots required



# Planarity

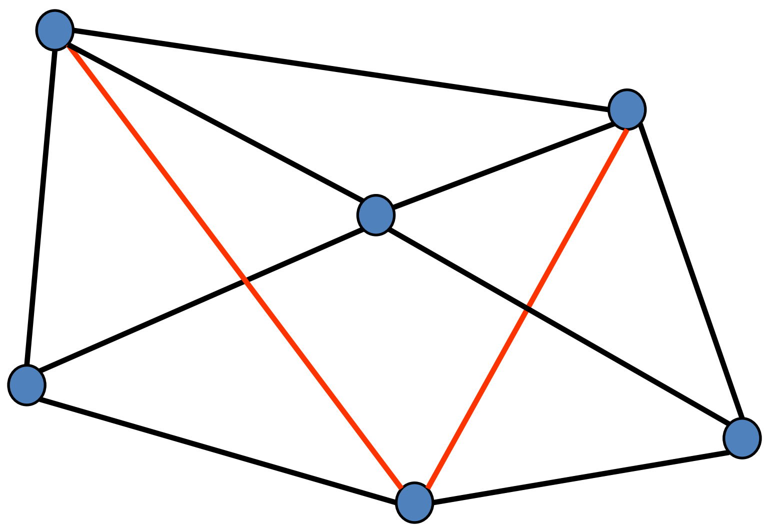
- A graph is **planar** if it can be embedded in the plane with no edges crossing



- Is this graph planar?

# Planarity

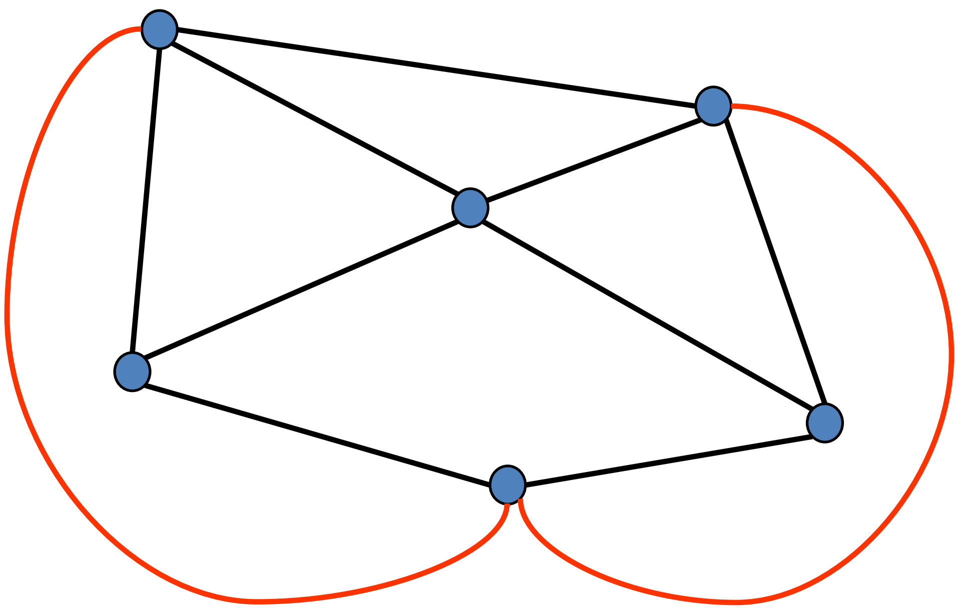
- A graph is planar if it can be embedded in the plane with no edges crossing



- Is this graph planar?
  - Yes

# Planarity

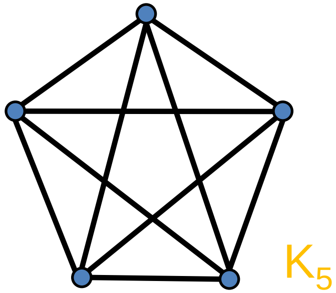
- A graph is **planar** if it can be embedded in the plane with no edges crossing

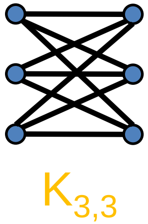


- Is this graph planar?
  - Yes

# Detecting Planarity

- Kuratowski's Theorem



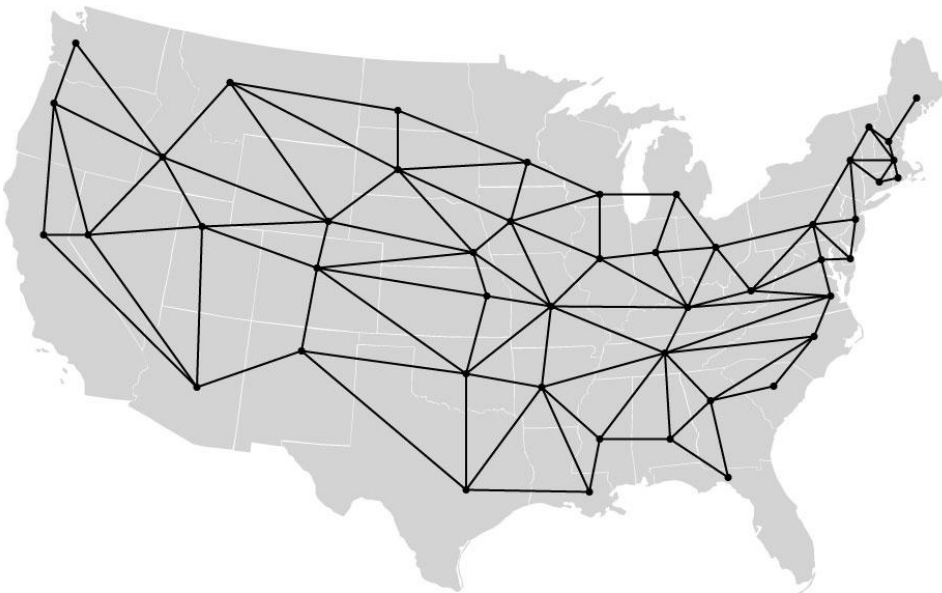



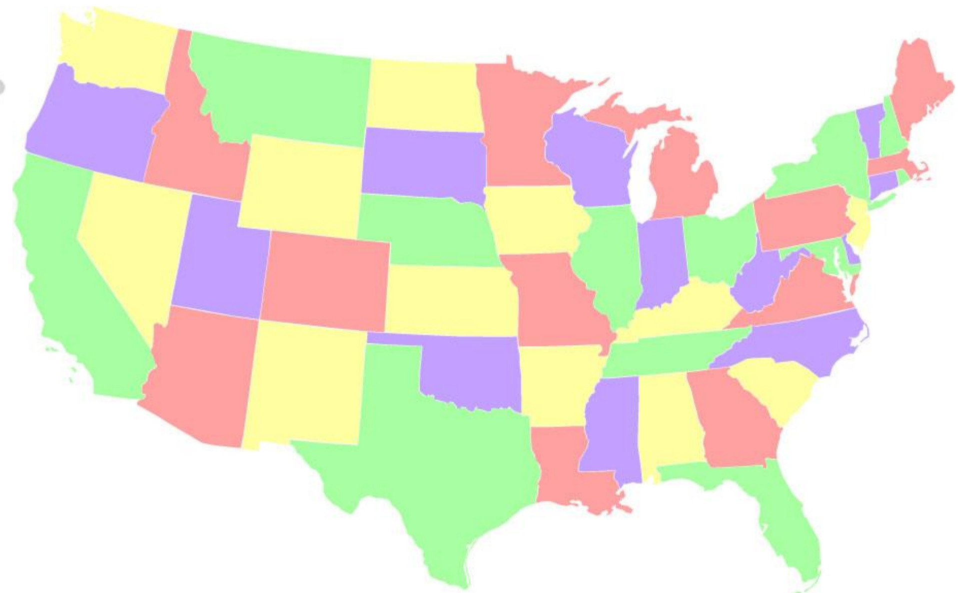

- A graph is planar if and only if it does not contain a copy of $K_5$ or $K_{3,3}$ (possibly with other nodes along the edges shown)

Four-Color Theorem:
Every planar graph is 4-colorable.

(Appel & Haken, 1976)

# Another 4-colored planar graph

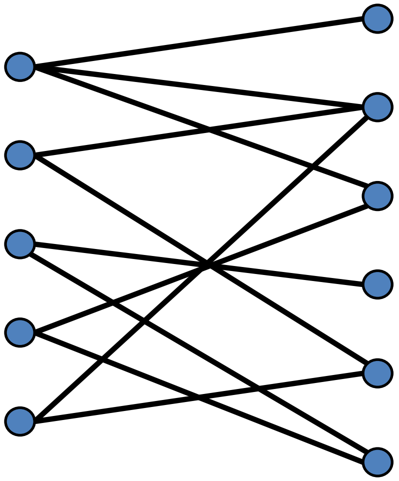




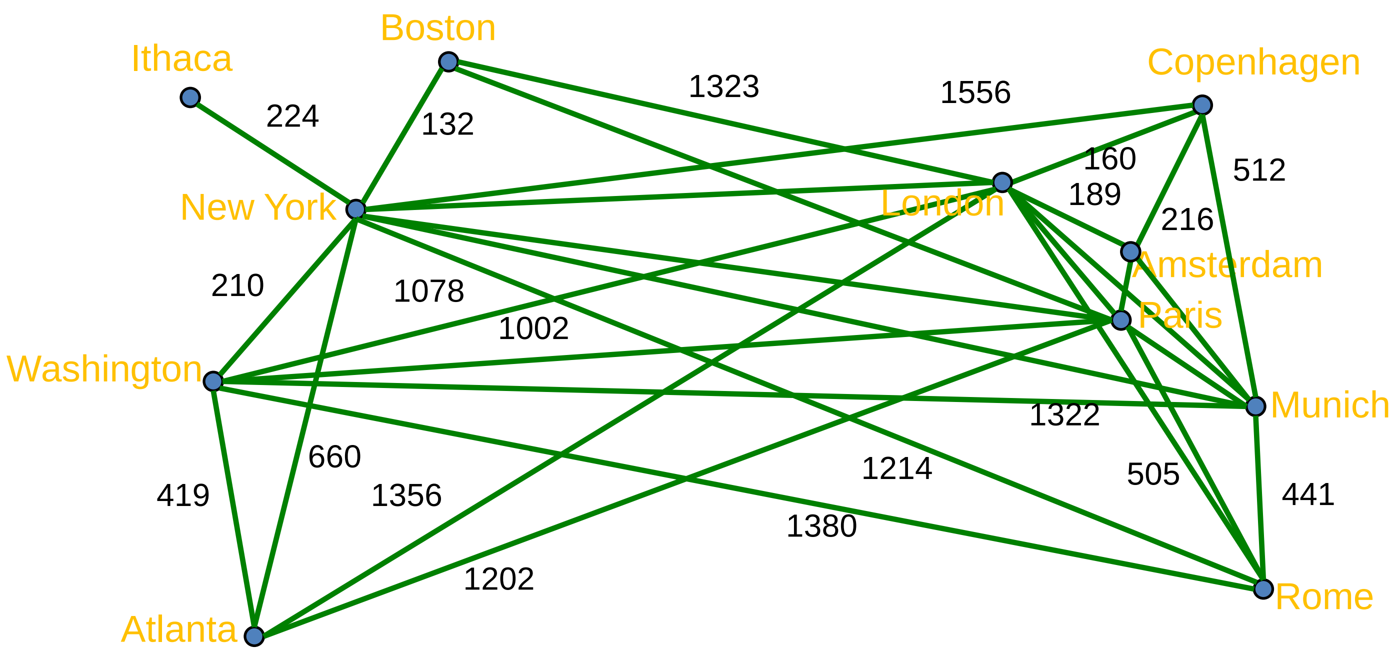
http://www.cs.cmu.edu/~bryant/boolean/maps.html

# Bipartite Graphs

- A directed or undirected graph is **bipartite** if the vertices can be partitioned into two sets such that all edges go between the two sets
- The following are equivalent
  - G is bipartite
  - G is 2-colorable
  - G has no cycles of odd length



# Traveling Salesperson




- Find a path of minimum distance that visits every city

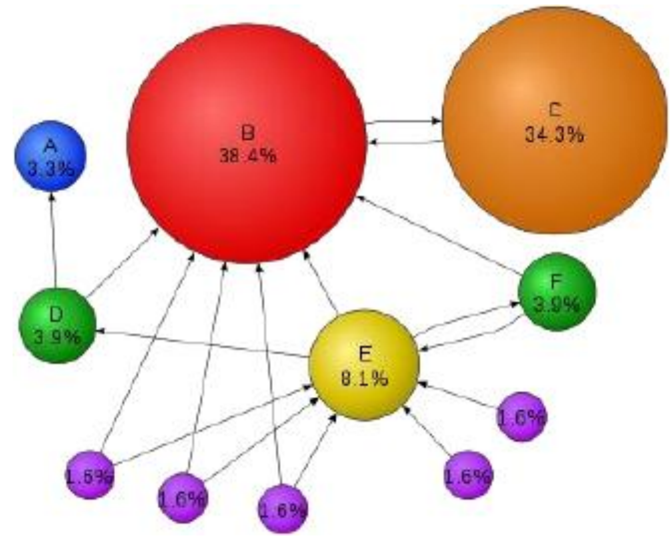
# The Web

- the World Wide Web itself can be seen as a huge graph, the so called **web graph**
  - **pages are vertices** connected by **edges that represent hyperlinks**
  - the web graph has **several billion vertices** and **several billion edges**

- the success of major internet companies such as Google is based on the ability to conduct computations on this huge graph

# Google‘s PageRank

- success factor of Google‘s search engine:
  - much **better ranking of search results**
- ranking is based on **PageRank**, a graph algorithm computing the ‚importance‘ of webpages
  - simple idea: **look at the structure of the underlying network**
  - important pages have a lot of links from other important pages




- major technical success factor of Google: ability to conduct **web scale graph processing**

# Social Networks

- on facebook, twitter, LinkedIn, etc, the users and their interactions form a **social graph**
  - **users are vertices** connected by **edges that represent some kind of interaction** such as friendship, following, business contact
- fascinating research questions:
  - what is the structure of these graphs?
  - how do they evolve over time?

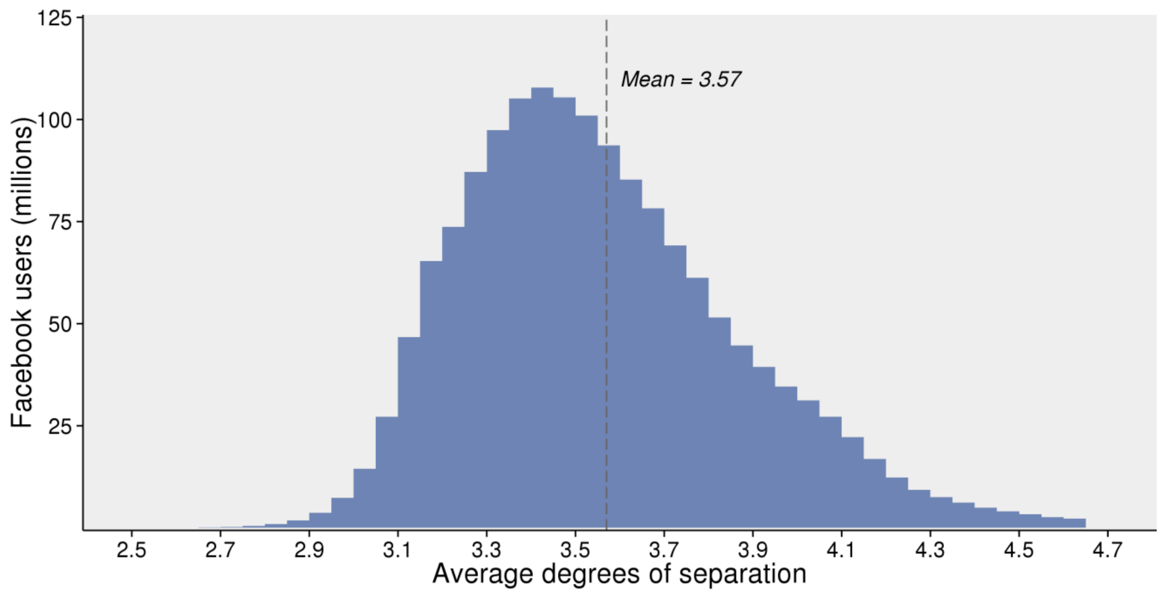
- analysis requires knowledge in both computer science and social sciences

# six degrees of separation

- **small world problem**
  - through how many social contacts do people know each other on average?
- **small world experiment** by Stanley Milgram
  - task: deliver a letter to a recipient whom you don‘t know personally
  - you may forward the letter only to persons that you know on a first-name basis
  - how many contacts does it take on average until the letter reaches the target?
- **results**
  - it took 5.5 to 6 contacts on average
  - confirmation of the popular assumption of **‚six degrees of separation‘** between humans
  - experiment criticised due to small number of participants, possibly biased selection

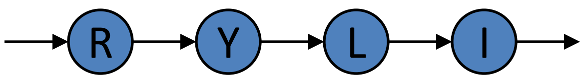
# 3.5 degrees of separation

- the small word problem as a graph problem in social network analysis
  - **what is the average distance between two users in a social graph?**
- In Feb 2016, a **world scale experiment** using the Facebook social graph
  - 1.6 billion users
  - result: average distance in Facebook is 3.57

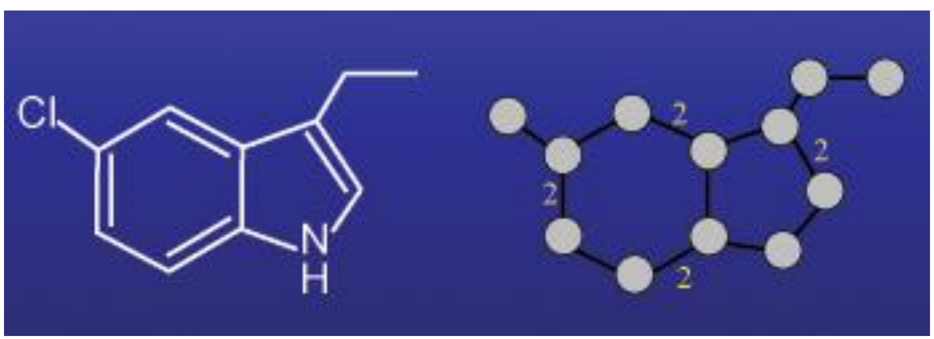


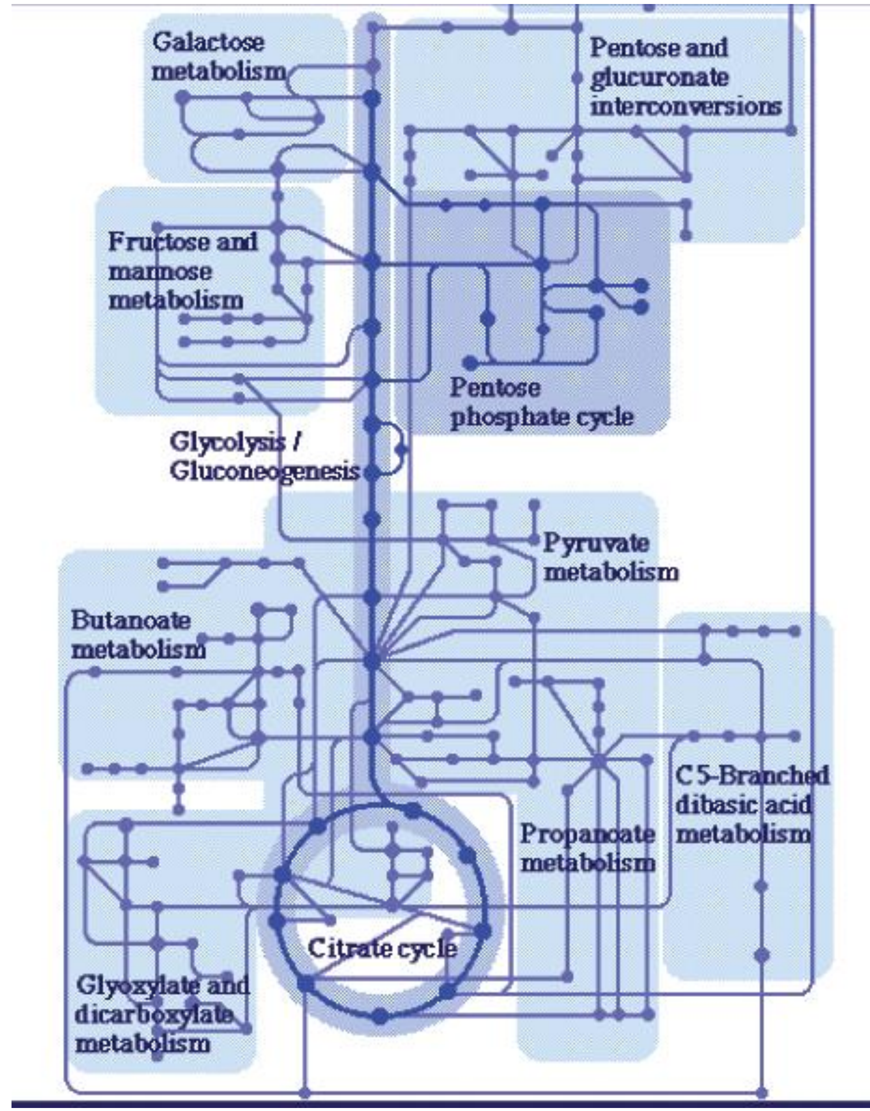
# Graphs in bioinformatics

- Sequences
  - DNA, proteins, etc.



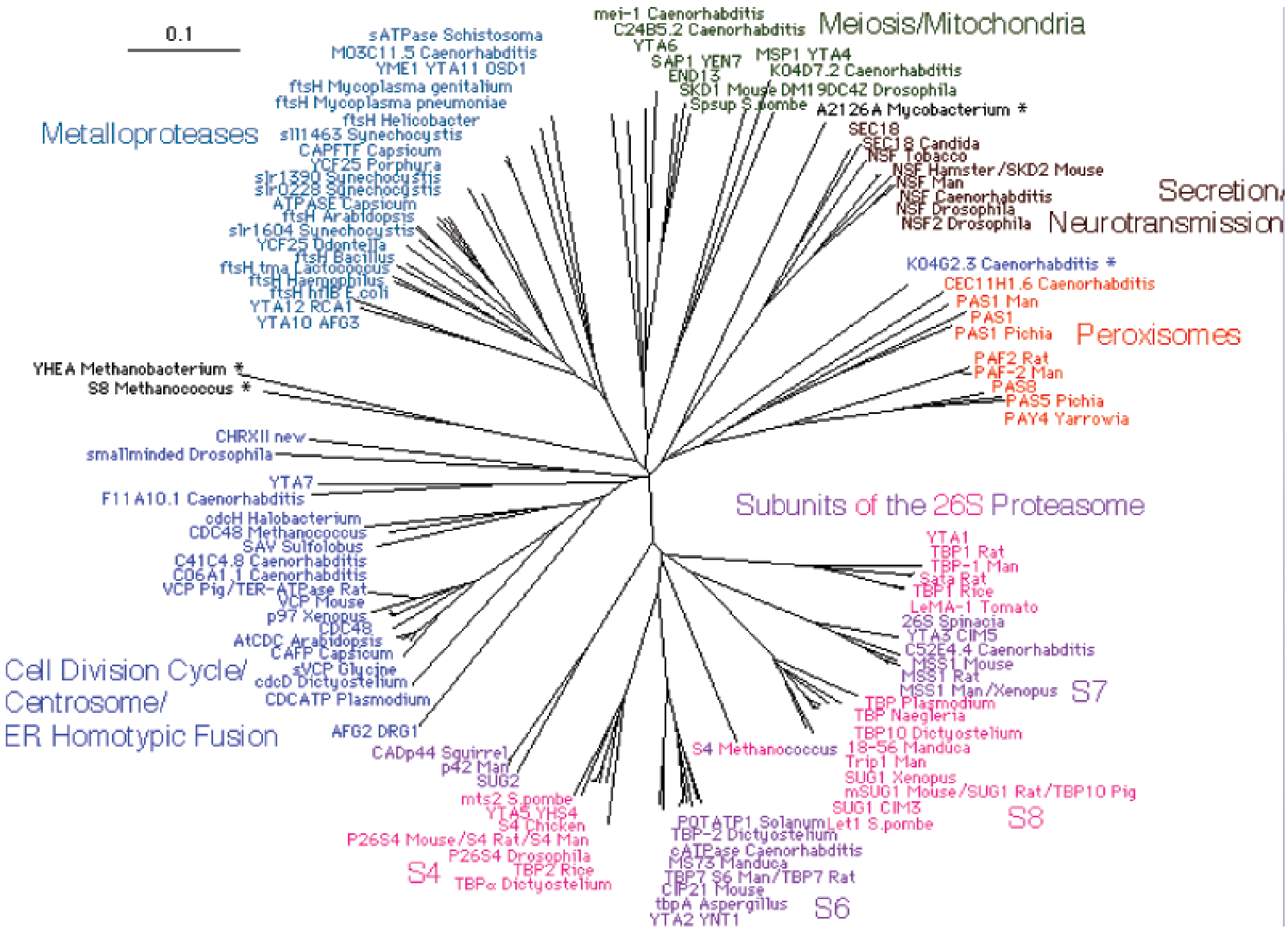

Chemical compounds




Metabolic pathways

# Graphs in bioinformatics

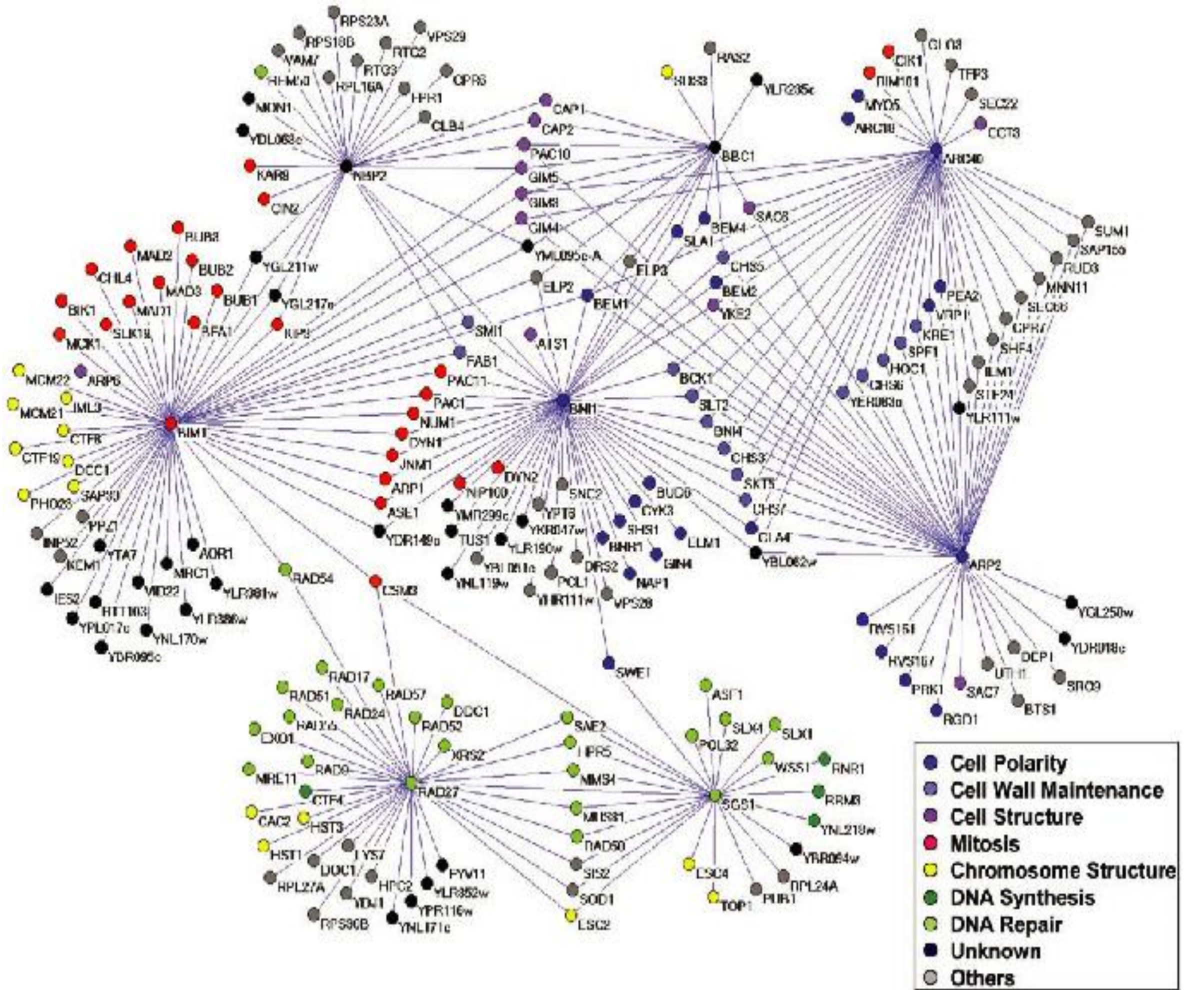



Phylogenetic trees

# Genetic interactions: synthetic lethals and suppressors

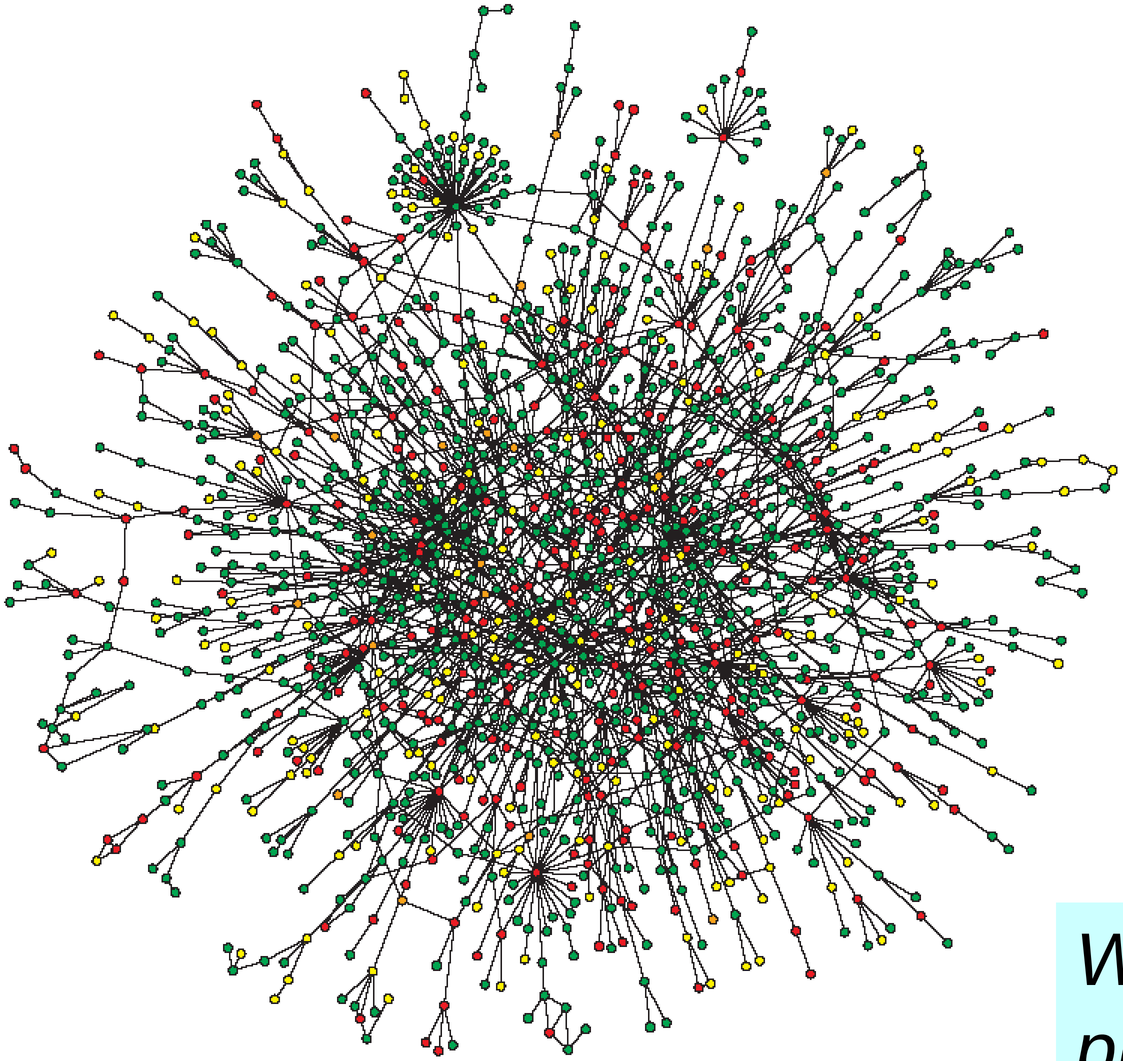
<u>Genetic Interactions:</u>

- Widespread method used by geneticists to discover pathways in yeast, fly, and worm
- Implications for drug targeting and drug development for human disease
- Thousands are now reported in literature and systematic studies
- As with other types, the number of known genetic interactions is *exponentially increasing…*




Adapted from Tong *et al.*, *Science* 2001

# Yeast protein-protein interaction network



*What are its network properties?*

# Applications of Graphs

- Communication networks; social networks
- Routing and shortest path problems
- Commodity distribution (network flow)
- Traffic control
- Resource allocation
- Numerical linear algebra (sparse matrices)
- Geometric modeling (meshes, topology, …)
- Image processing (e.g., graph cuts)
- Computer animation (e.g., motion graphs)
- Systems biology
- …

# Overview

1) Graphs
2) **Graph processing with Hadoop/MapReduce**
3) Google Pregel

# Why not use MapReduce/Hadoop?

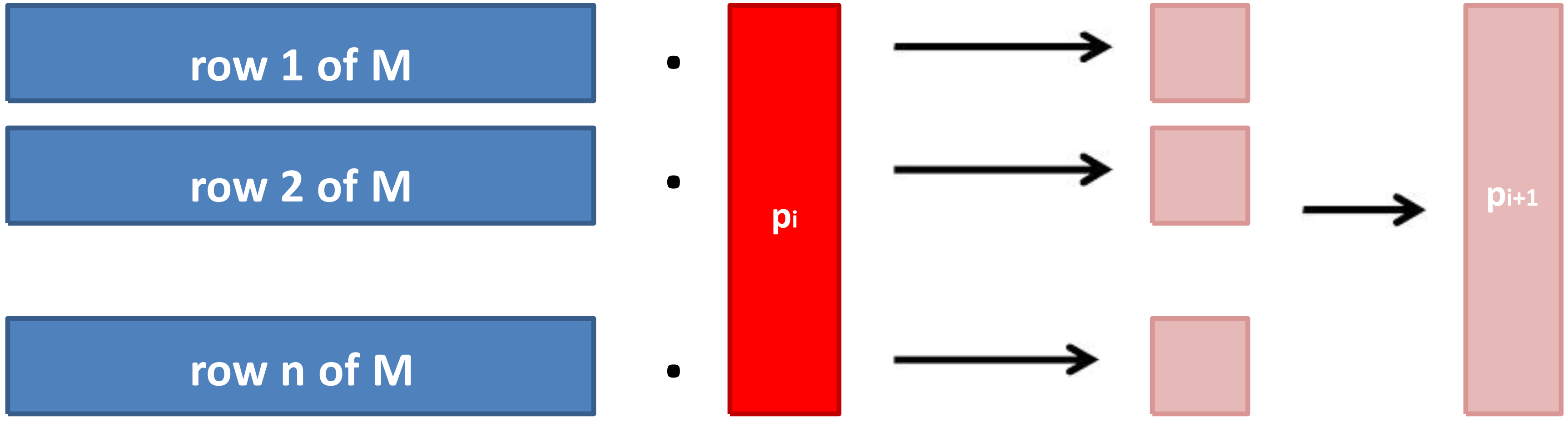
- MapReduce/Hadoop is a popular way to perform data-intensive computing, why not use it for graph processing?
- Example: **PageRank**
  - defined **recursively**
  - each vertex distributes its authority to its neighbors in equal proportions

$$p_i = \sum_{j \in \{(j,i)\}} \frac{p_j}{d_j}$$

# Textbook approach to PageRank in MapReduce

- PageRank p is the **principal eigenvector** of the Markov matrix M defined by the **transition probabilities between web pages**
- it can be obtained by iteratively multiplying an initial PageRank vector by M (**power method**)

$$p_{i+1} = Mp_i$$



# Drawbacks

- **Not intuitive**: only crazy scientists think in matrices and eigenvectors

- **Unnecessarily slow**: Each iteration is a single MapReduce job with lots of overhead
  - separately scheduled
  - the graph structure is read from disk
  - the intermediary result is written to HDFS
- **Hard to implement**: a join has to be implemented by hand, lots of work, best strategy is data dependent

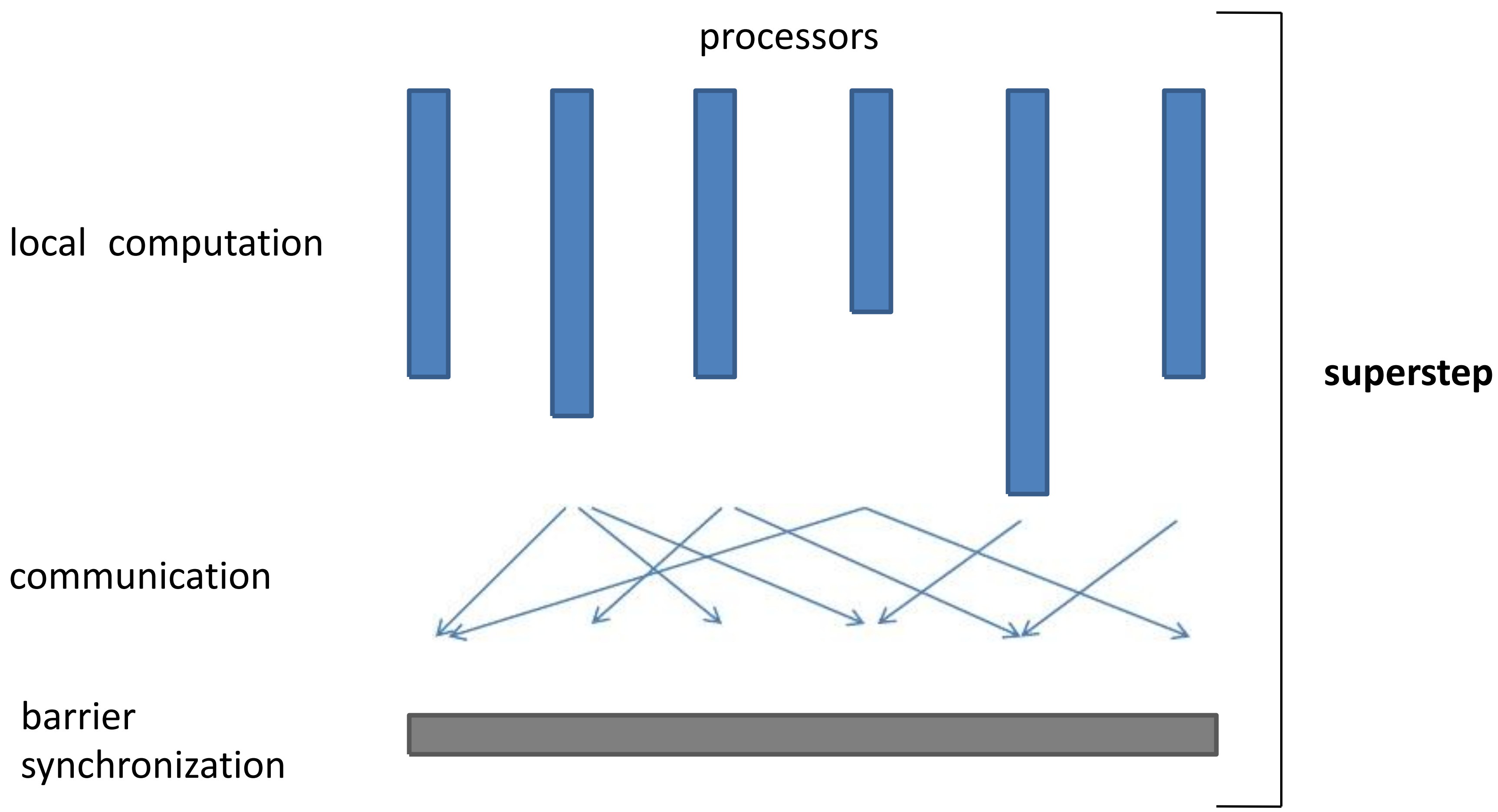
# Overview

1) Graphs
2) Graph processing with Hadoop/MapReduce
3) **Google Pregel**

# Google Pregel

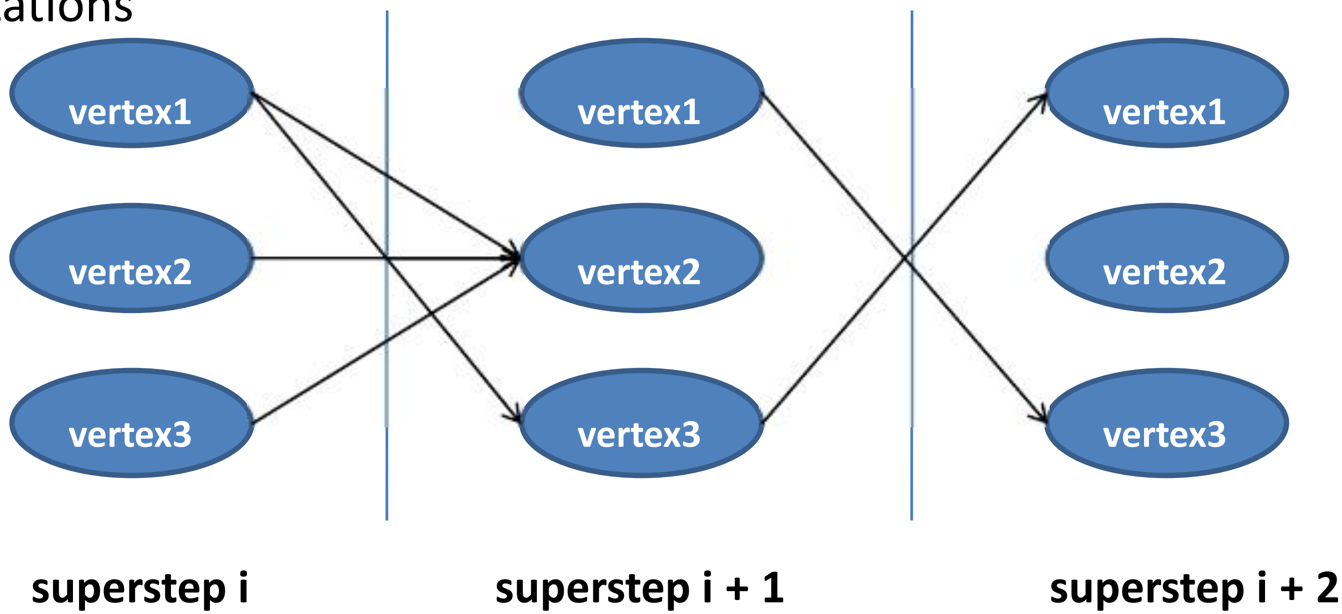
- distributed system especially developed for **large scale graph processing**
- intuitive API that let‘s you ‚**think like a vertex**‘
- **Bulk Synchronous Parallel** (BSP) as execution model
- fault tolerance by **checkpointing**

# Bulk Synchronous Parallel (BSP)

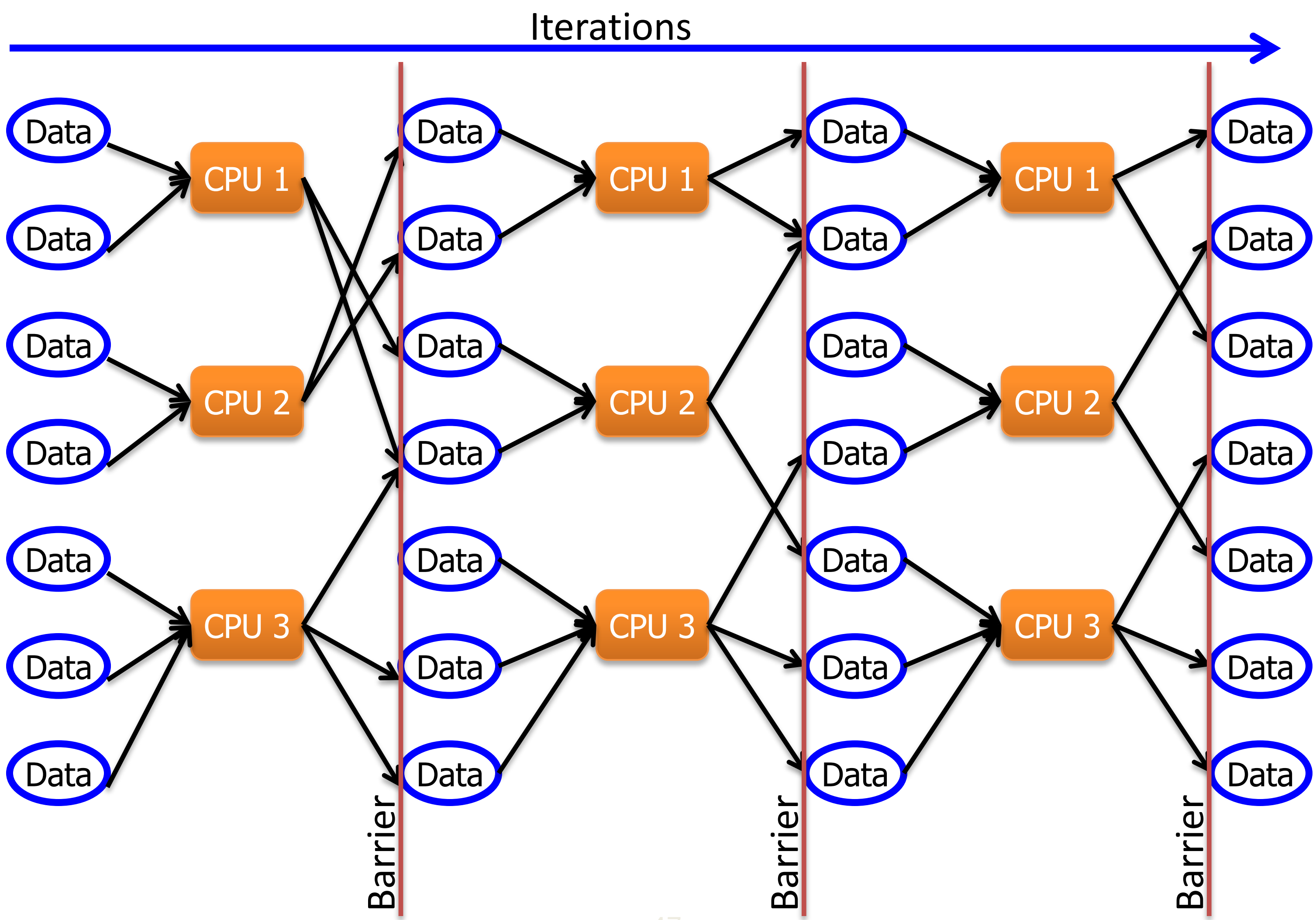


# Vertex-centric BSP

- each vertex has an **id**, a **value**, a **list of its adjacent neighbor ids** and the corresponding **edge values**
- each vertex is invoked **in each superstep**, can **recompute its value** and **send messages to other vertices**, which are **delivered over superstep barriers**
- advanced features : **termination votes**, combiners, **aggregators**, topology mutations

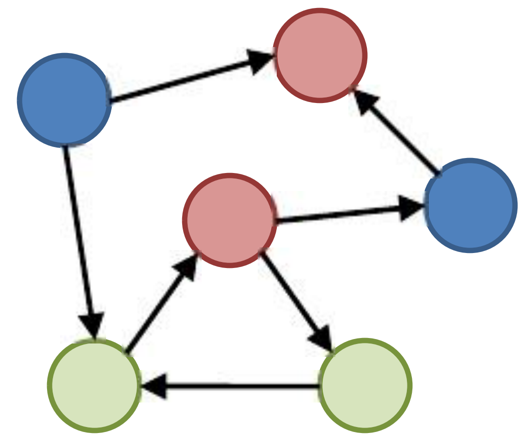


# Bulk Synchronous Parallel Model

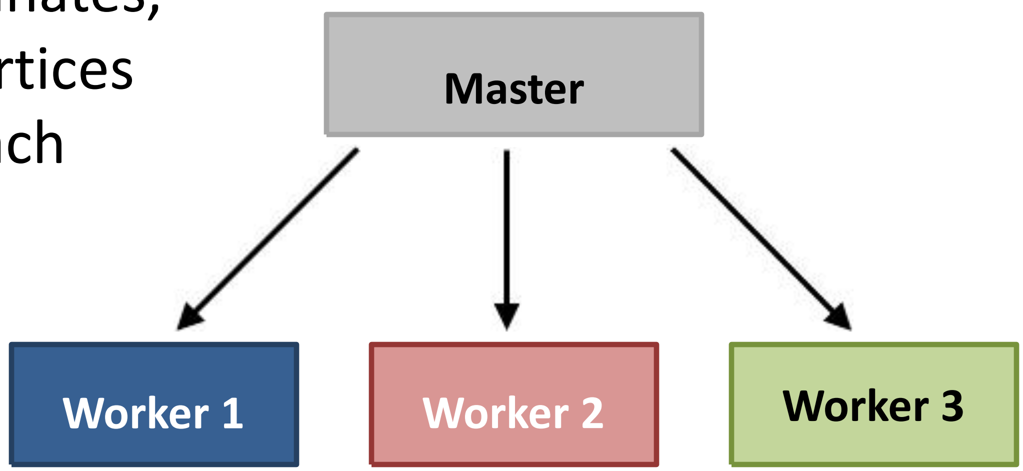


# Master-slave architecture

- **vertices are partitioned and assigned to workers**
  - default: hash-partitioning
  - custom partitioning possible

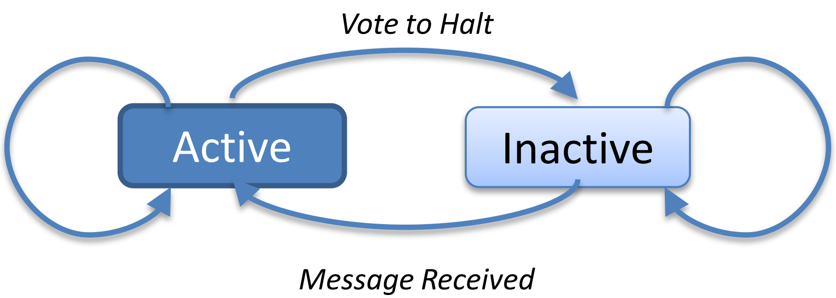


- **master** assigns and coordinates, while **workers** execute vertices and communicate with each other



# Algorithm Termination

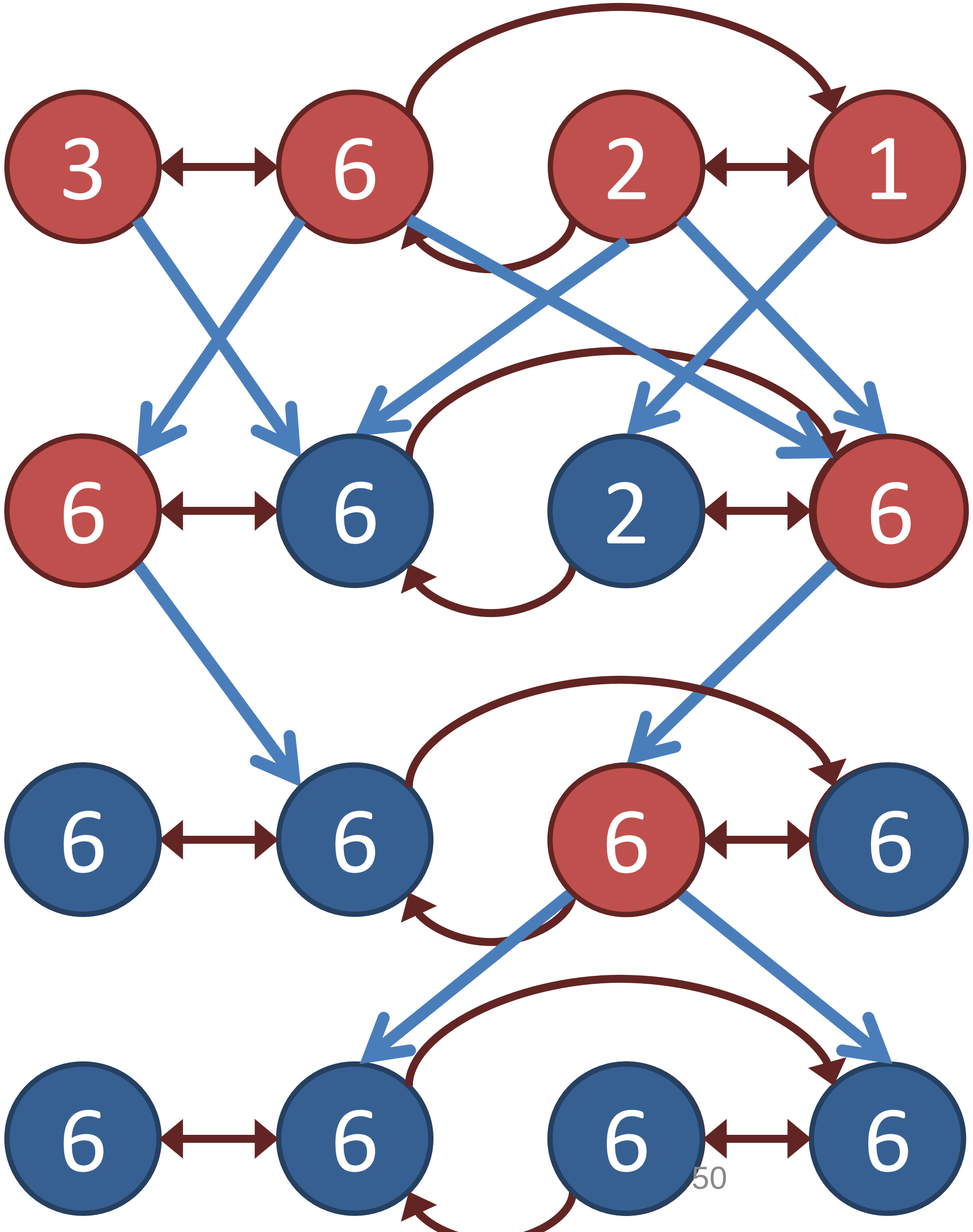
- Algorithm termination is based on every vertex voting to halt
  - In superstep 0, every vertex is active
  - All active vertices participate in the computation of any given superstep
  - A vertex deactivates itself by voting to halt and enters an inactive state
  - A vertex can return to active state if it receives an external message




Vertex State Machine

- Program terminates when all vertices are simultaneously inactive and there are no messages in transit

# Finding the Max Value in a Graph

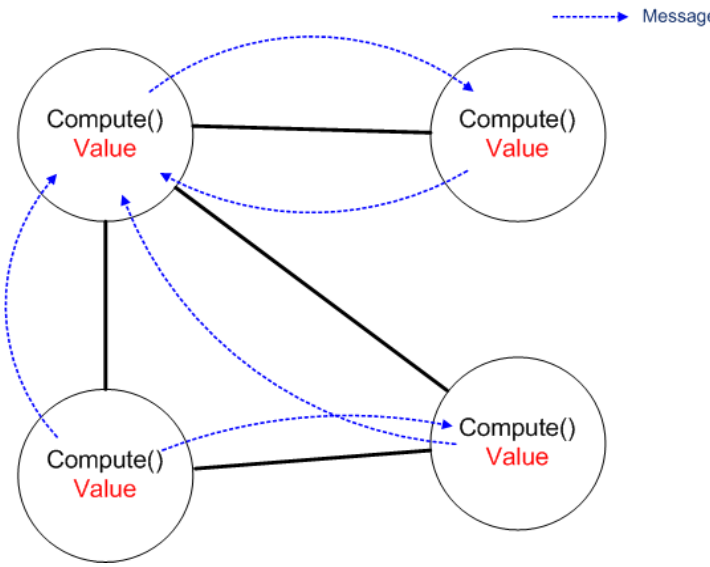



Blue Arrows are messages

Blue vertices have voted to halt

# Message Passing, Combiners, and Aggregators

- **Messages** can be passed from any vertex to any other vertex in the Graph
  - Any number of messages may be passed
  - Message order is not guaranteed
  - Messages will not be duplicated




- **Combiners** can be used to reduce the number of messages passed between supersteps

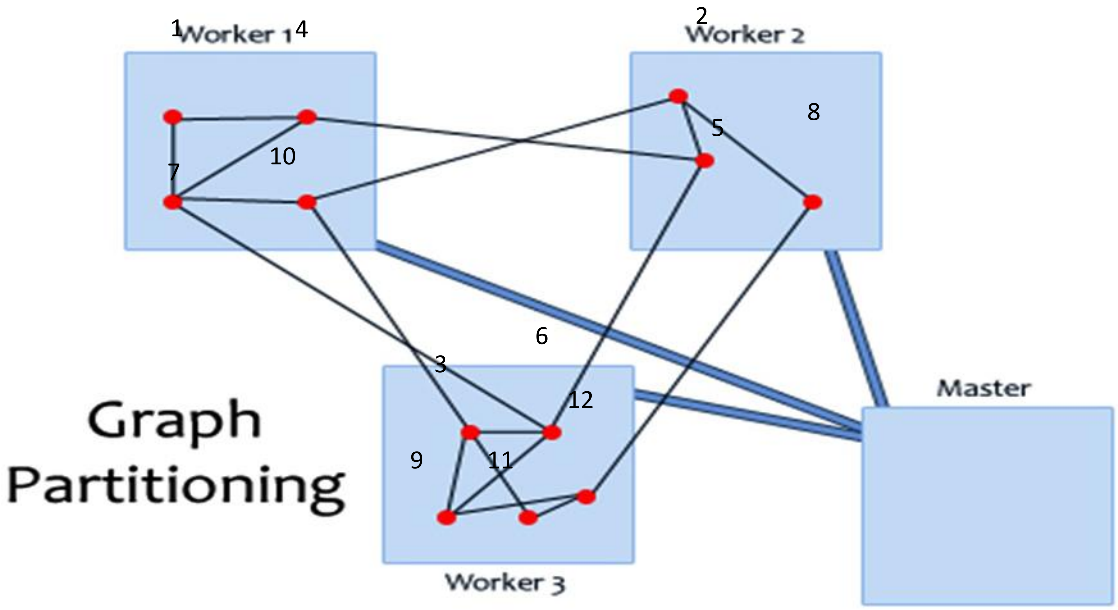
- **Aggregators** are available for reduction operations such as sum, min, max etc.

# Topology Mutations, Input and Output

- The graph structure can be modified during any superstep
  - Vertices and edges can be added or deleted
  - Conflicts are handled using partial ordering of operations
  - User-defined handlers are also available to manage conflicts
- Flexible input and output formats
  - Text File
  - Relational Database
  - Bigtable Entries
- Interpretation of input is a “pre-processing” step separate from graph computation
  - Custom formats can be created by sub-classing the Reader and Writer classes

# Graph Partitioning

- The input graph is divided into **partitions** consisting of vertices and outgoing edges
  - Default partitioning function is ***hash(ID) mod N***, where ***N*** is the # of partitions
  - It can be customized



# Execution of a Pregel Program

- **Steps of Program Execution:**

1. Copies of the program are distributed across all workers

   1.1 One copy is designated as a master

2. Master partitions the graph and assigns workers their respective partition(s) along with portions of the input
3. Master coordinates the execution of supersteps and delivers messages among vertices
4. Master calculates the number of inactive vertices after each superstep and signals workers to terminate if all vertices are inactive and no messages are in transit
5. Each worker may be instructed to save its portion of the graph

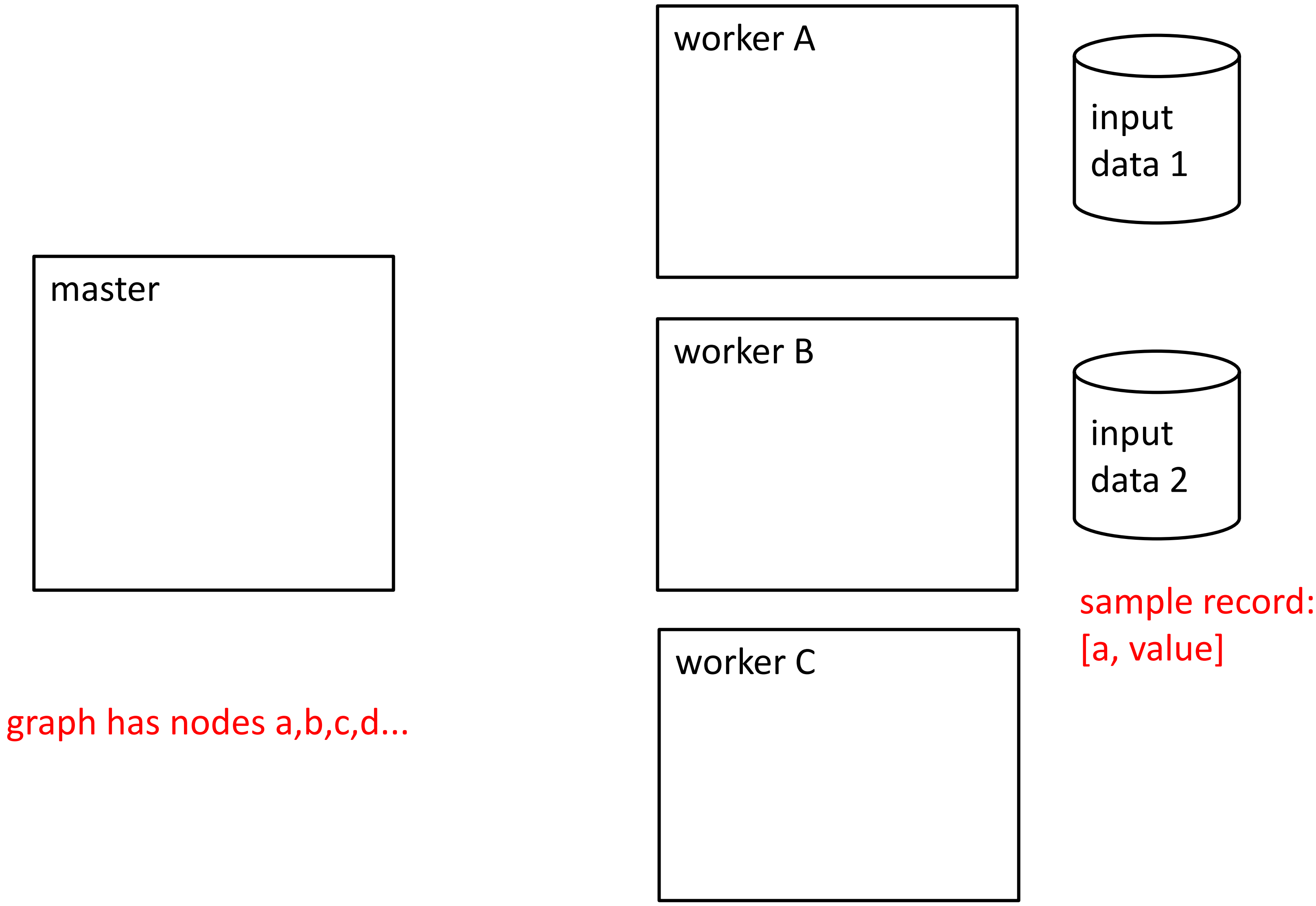
# Fault Tolerance in Pregel

- Fault tolerance is achieved through checkpointing
  - At the start of every superstep the master may instruct the workers to save the state of their partitions in a stable storage

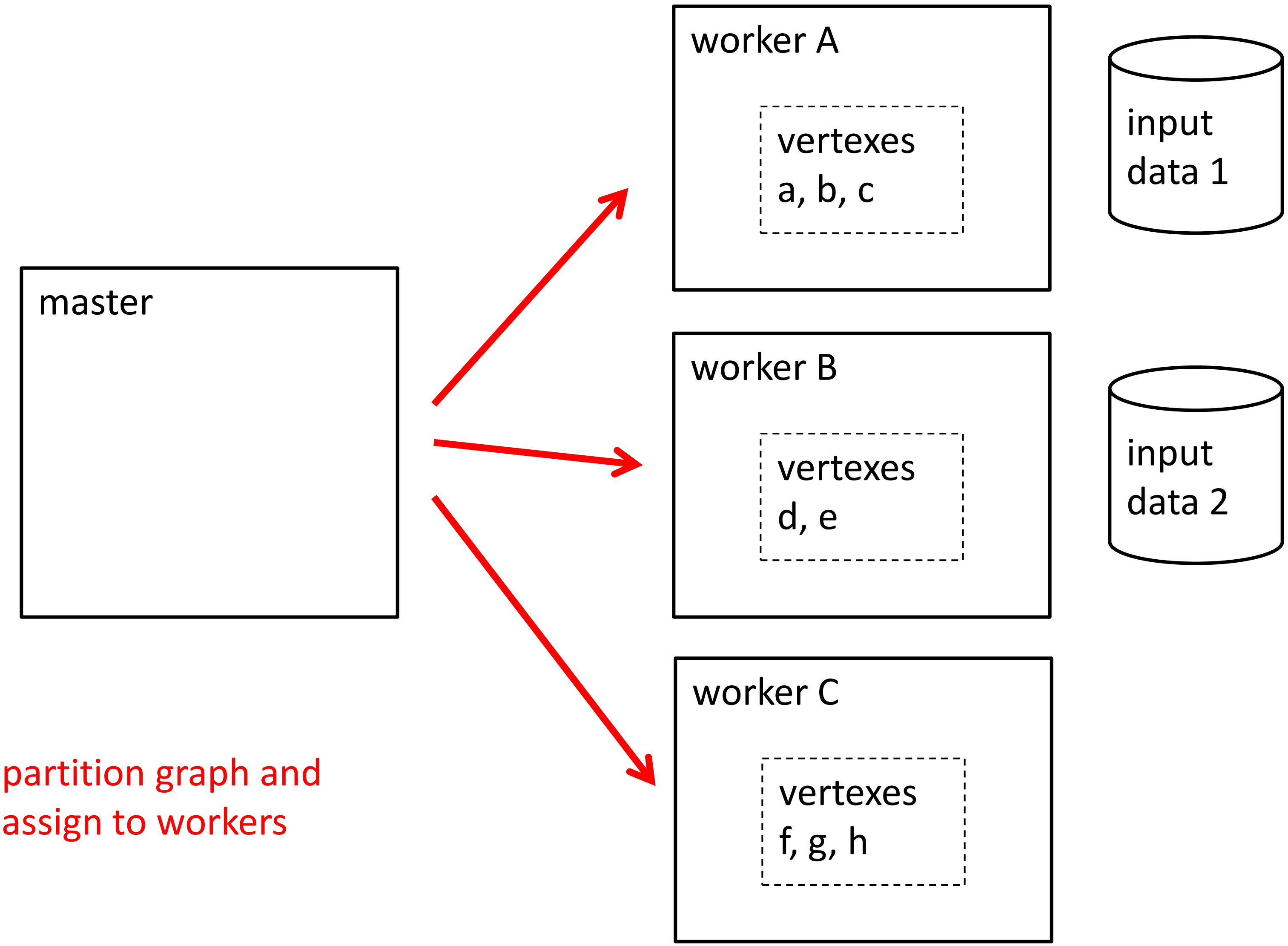
- Master uses *ping* messages to detect worker failures

- If a worker fails, the master reassigns corresponding vertices and input to another available worker and restarts the superstep
  - The available worker reloads the partition state of the failed worker from the most recent available checkpoint

# Architecture

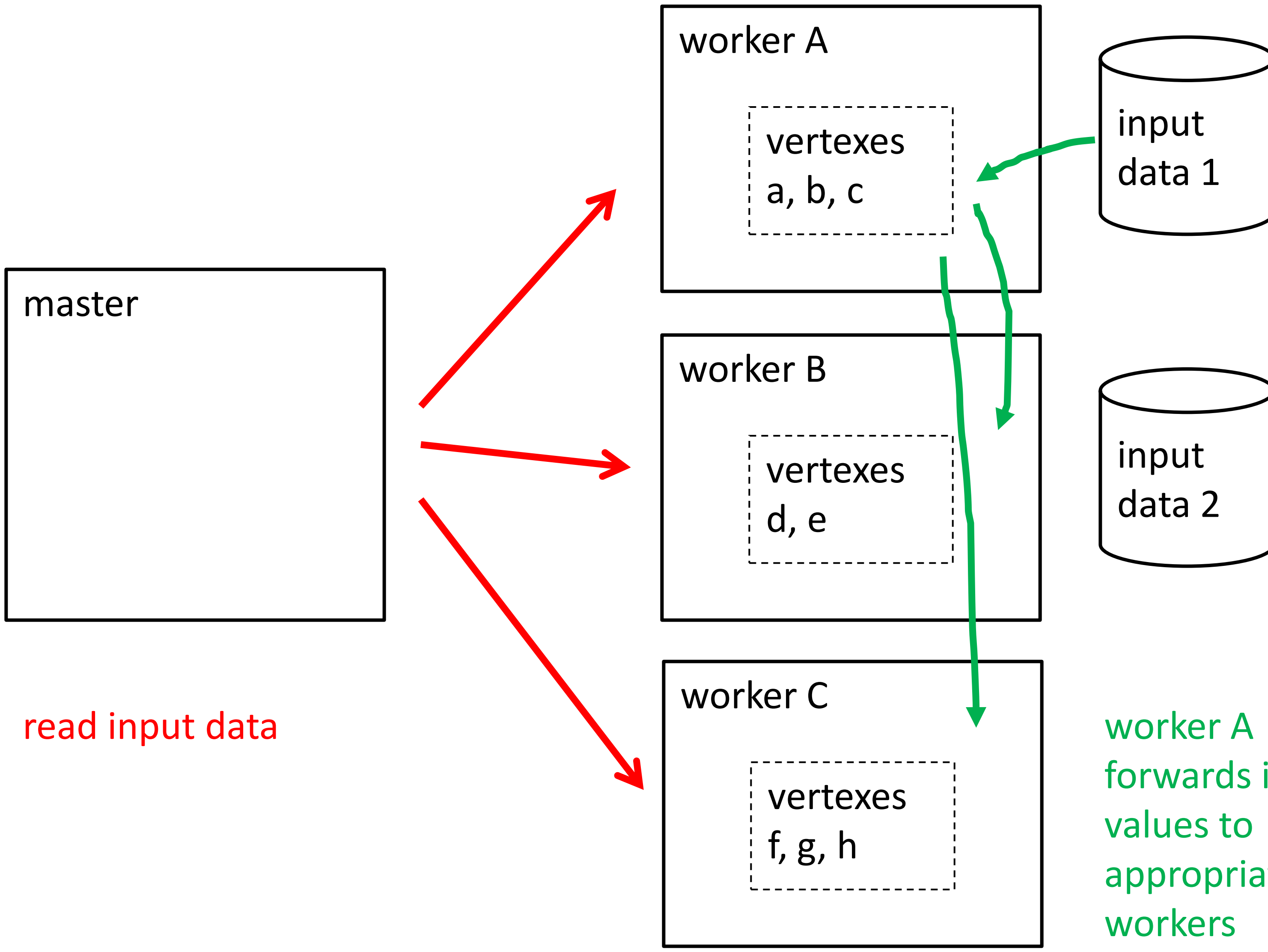


# Architecture




partition graph and
assign to workers

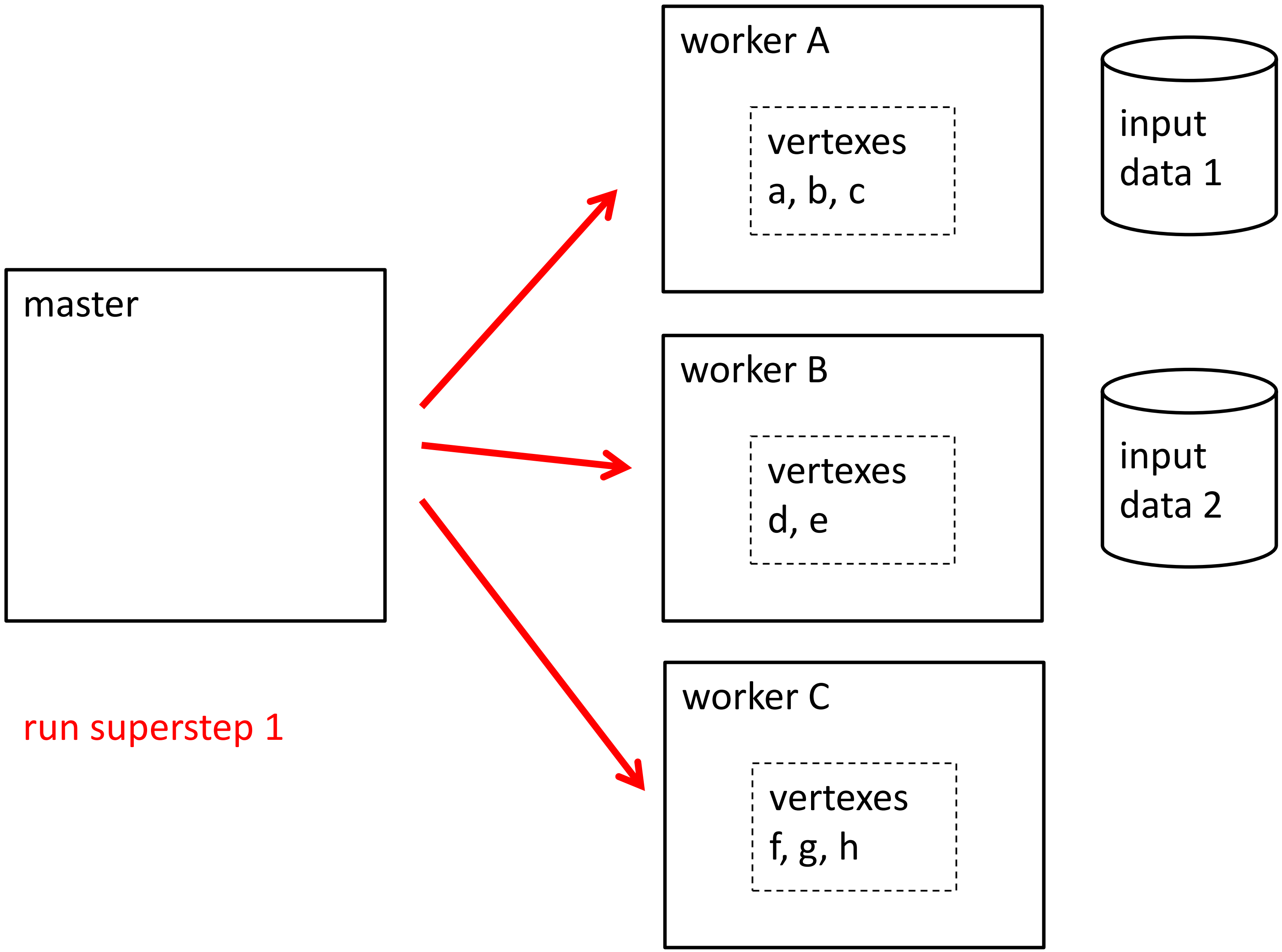
# Architecture



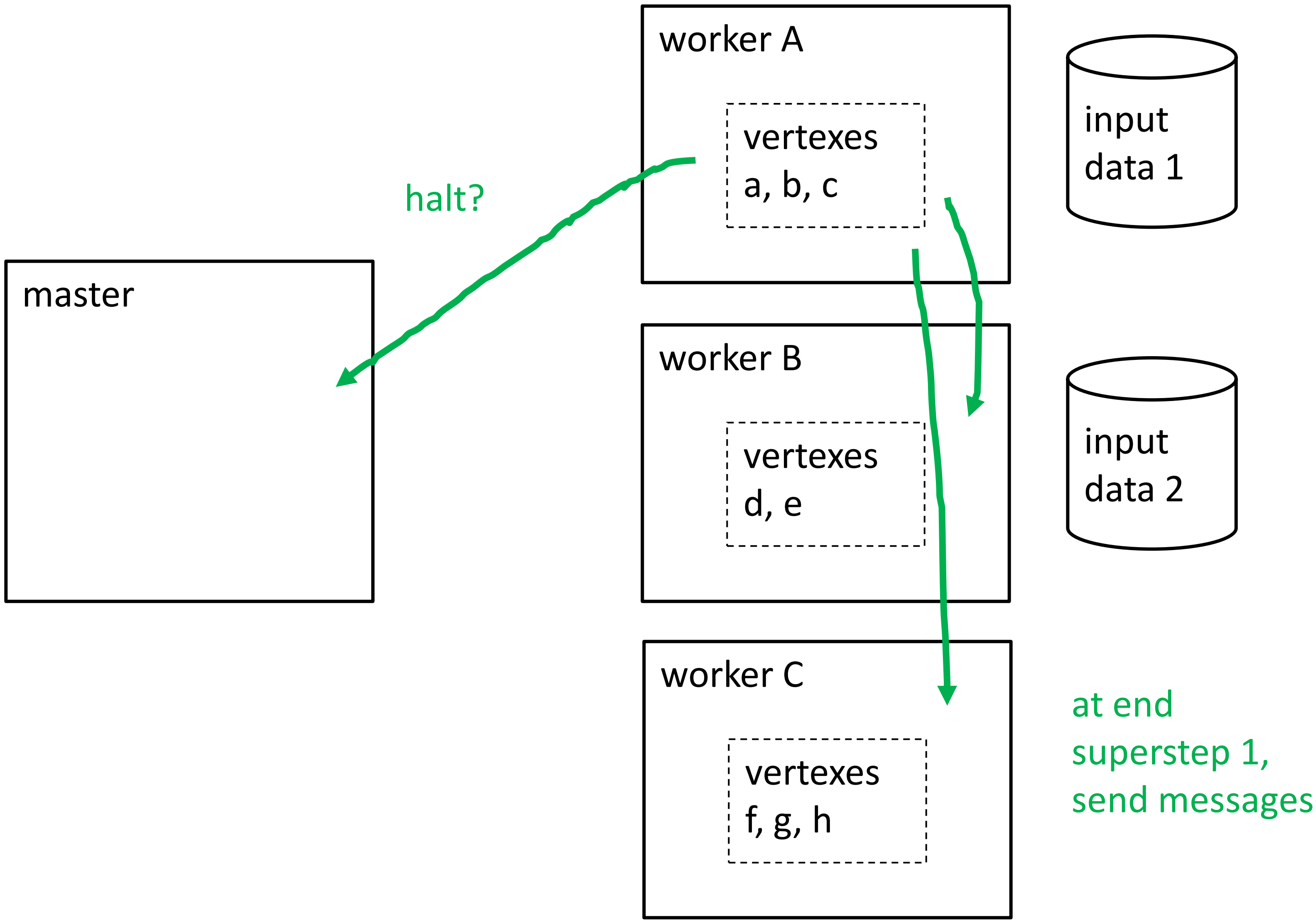

read input data

worker A forwards input values to appropriate workers

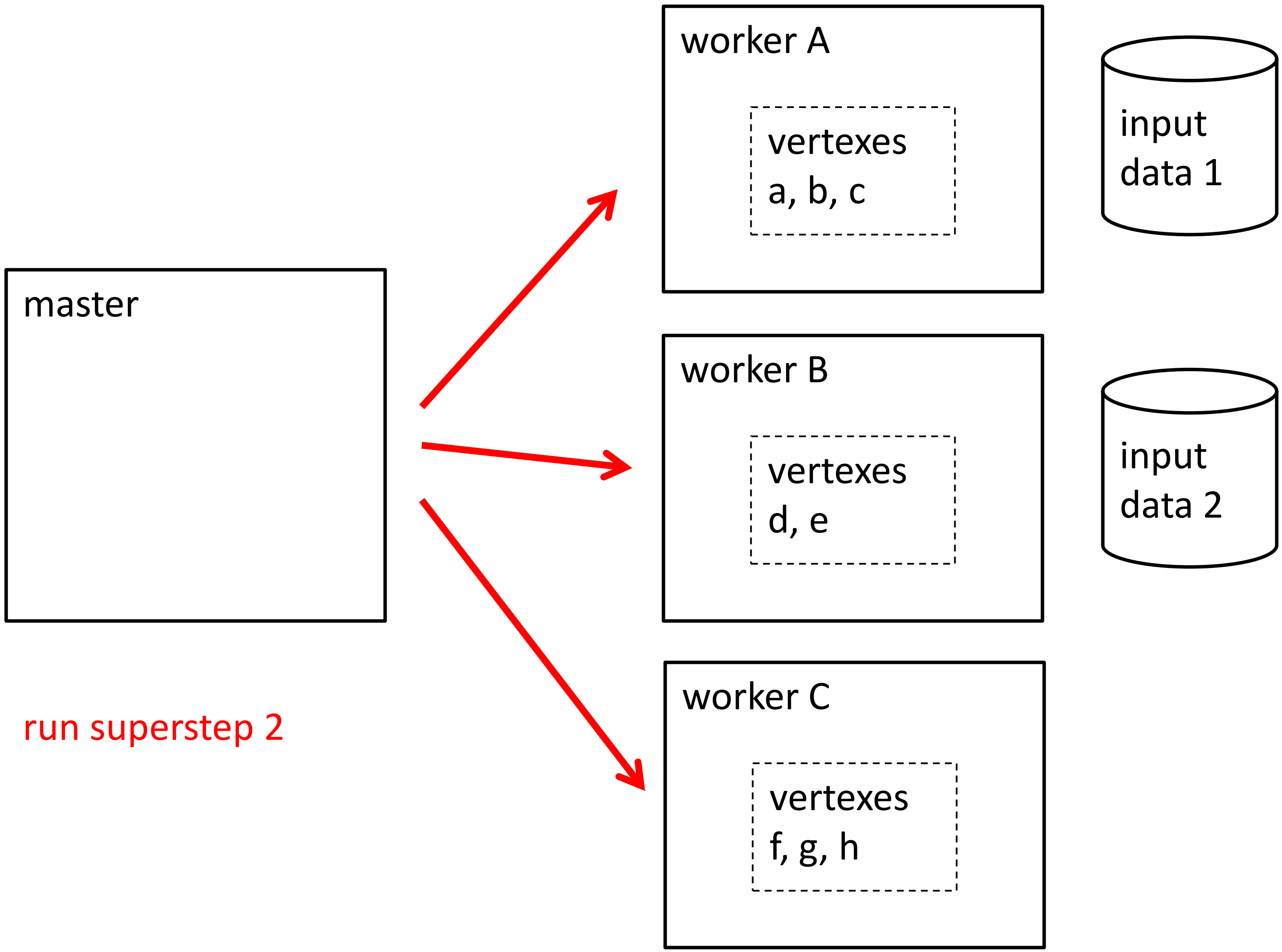
# Architecture



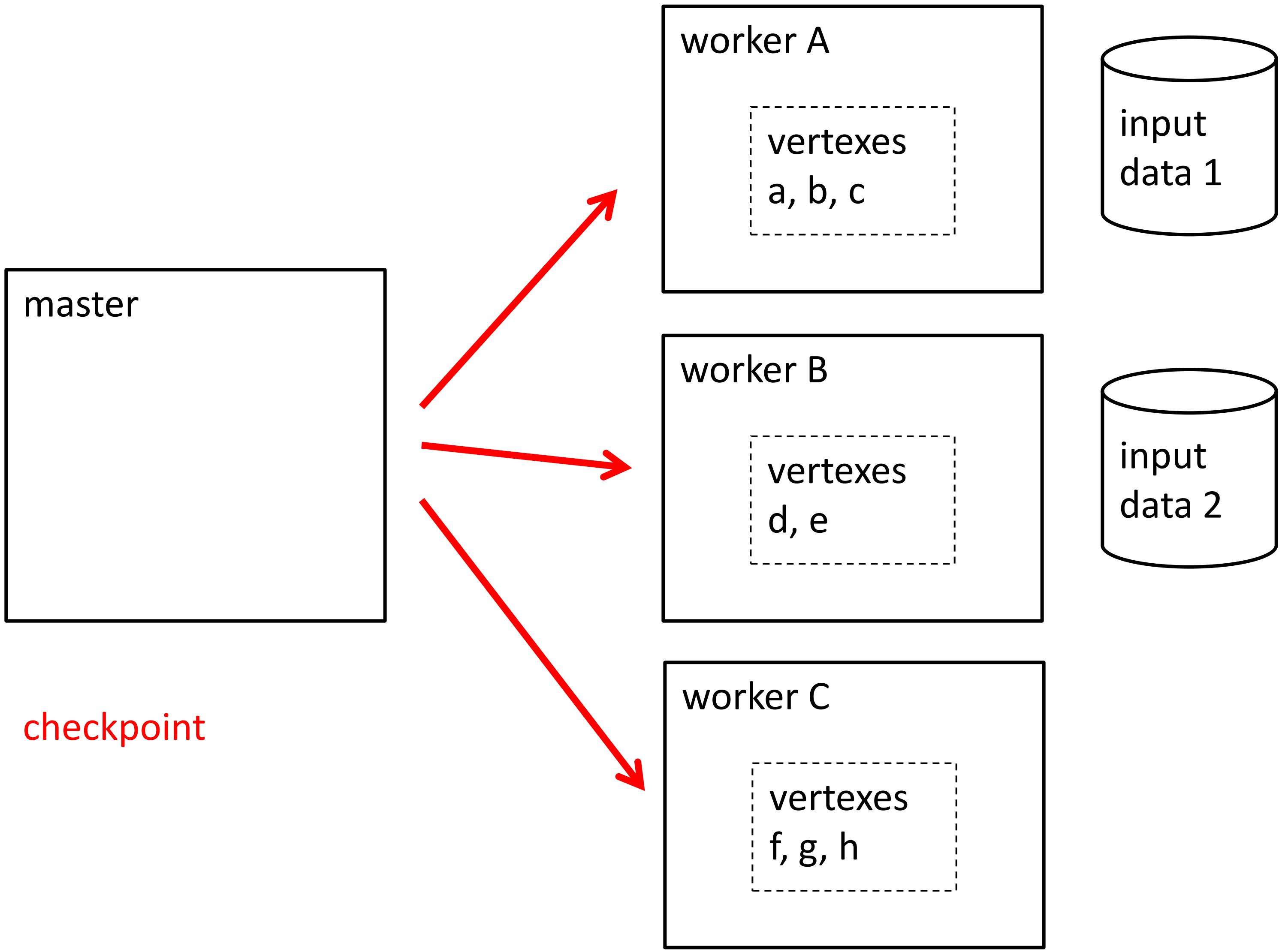
# Architecture



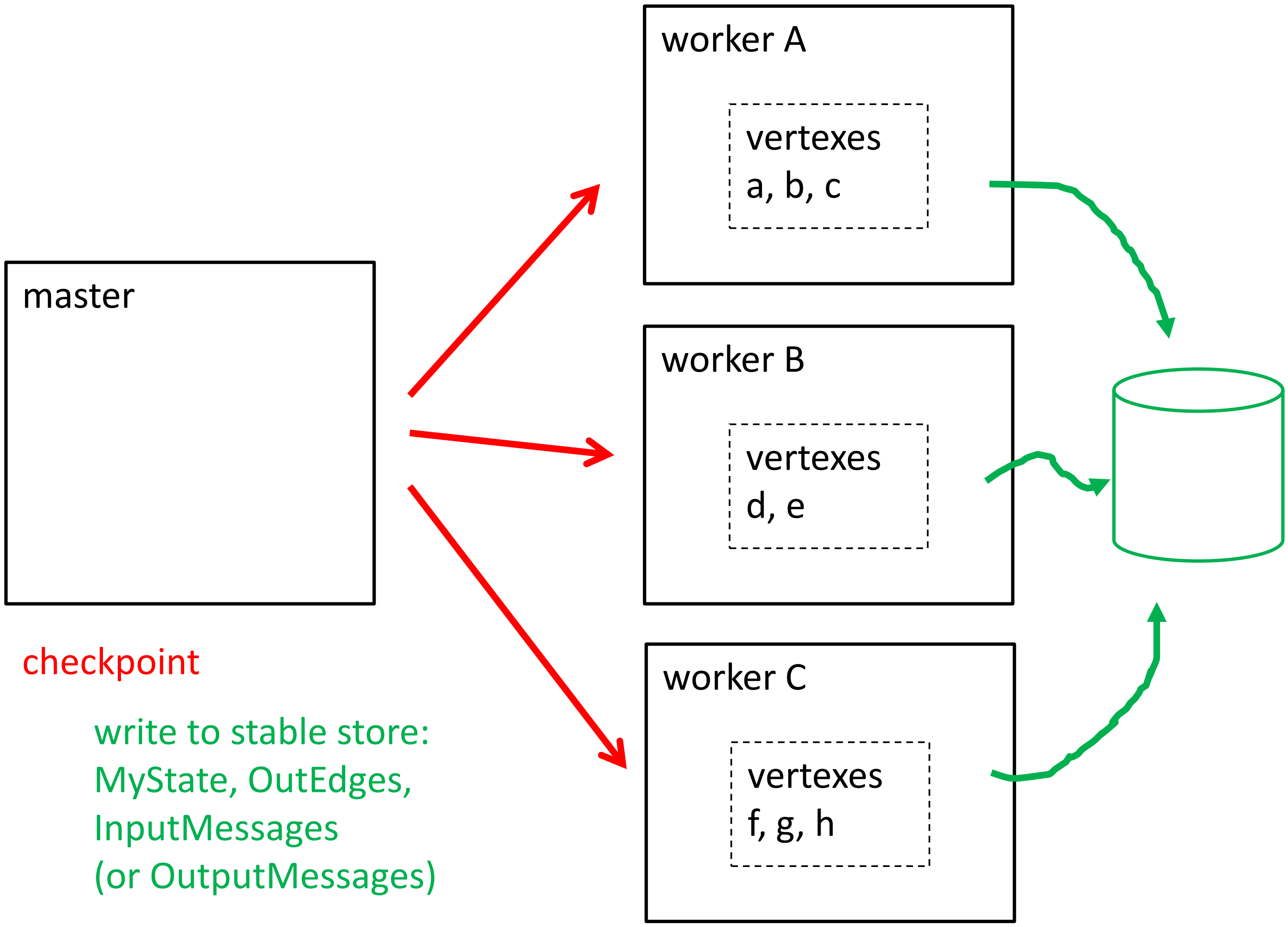
# Architecture



# Architecture



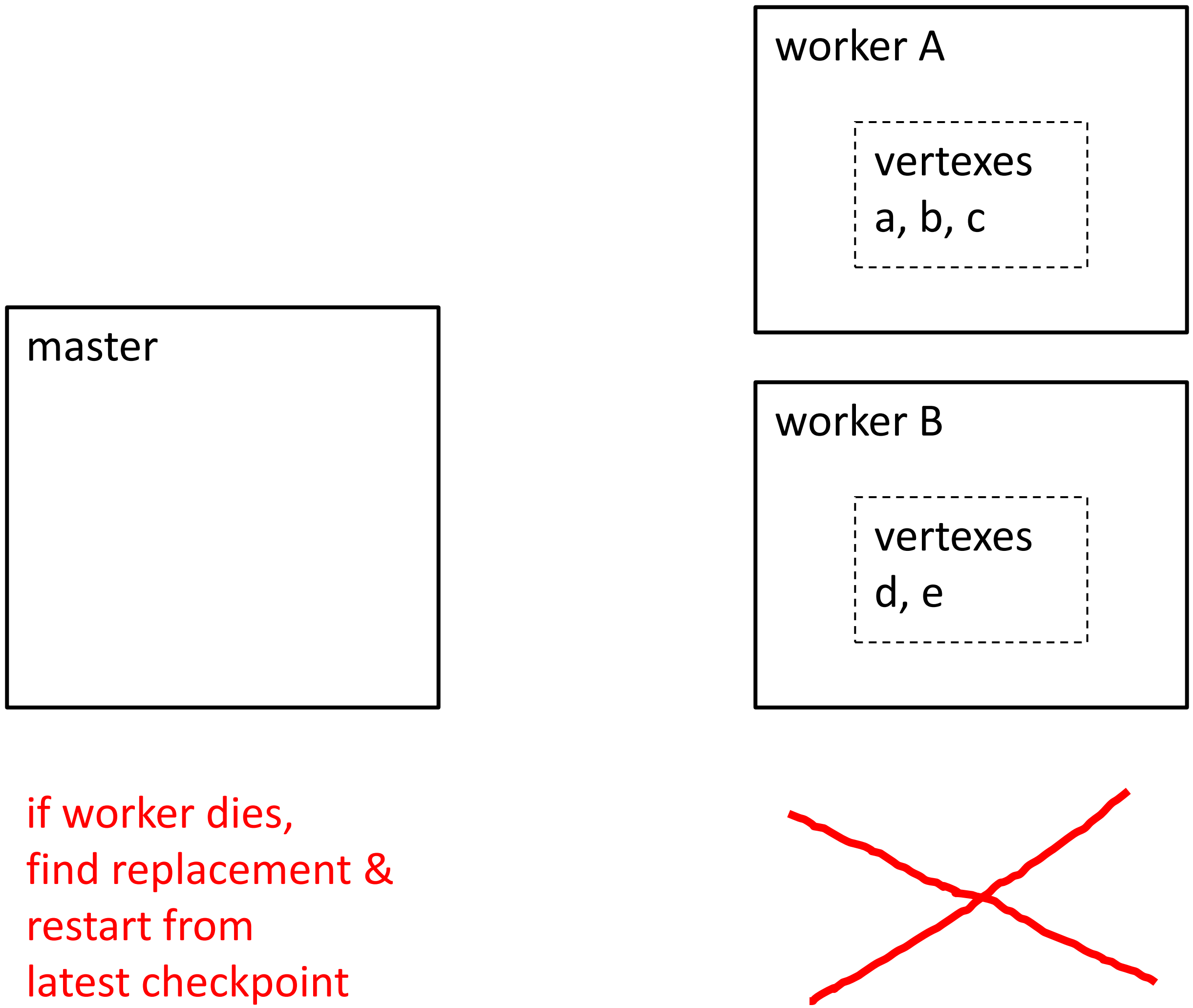
# Architecture



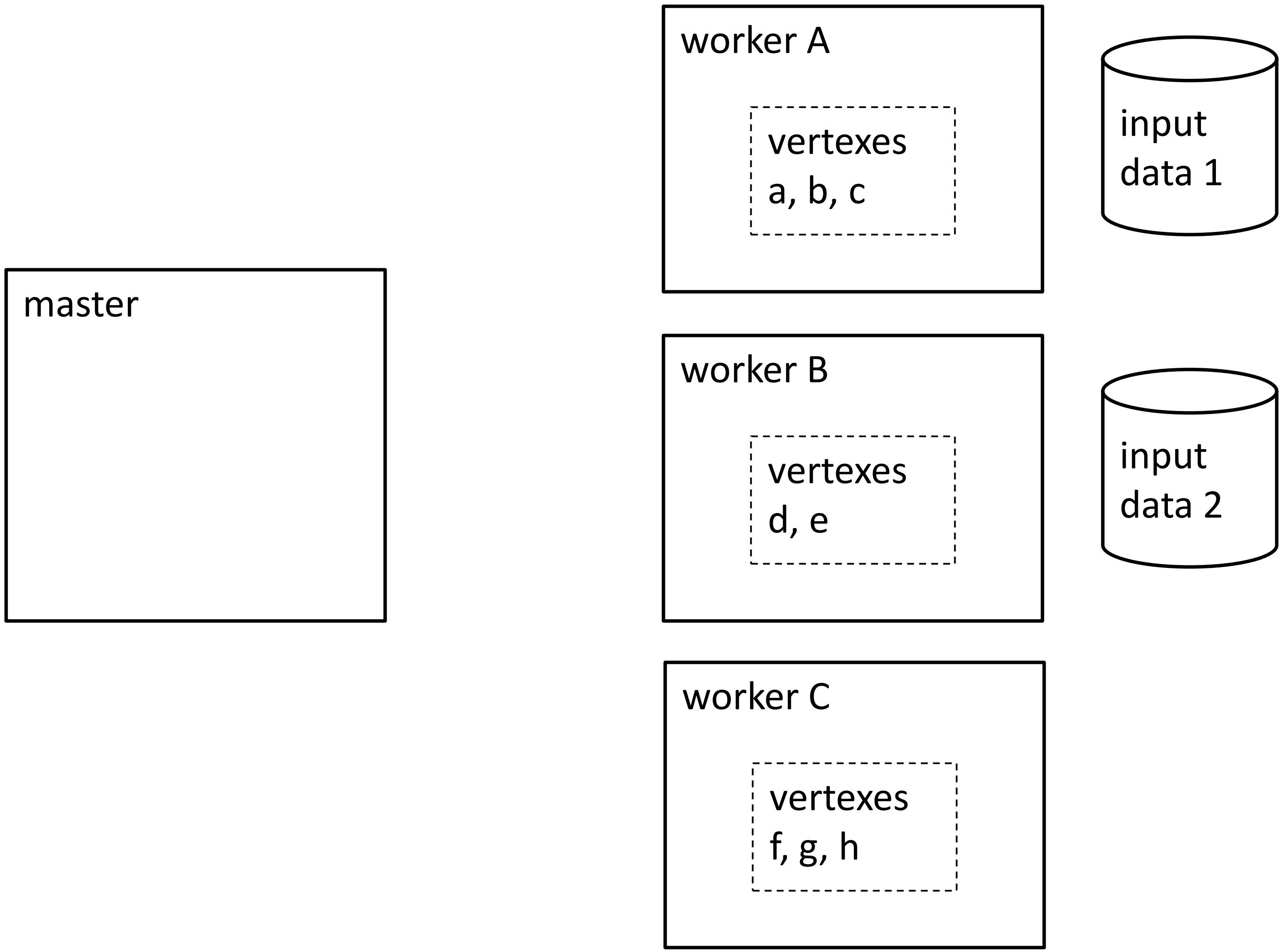

checkpoint

write to stable store:
MyState, OutEdges,
InputMessages
(or OutputMessages)

# Architecture



# Architecture



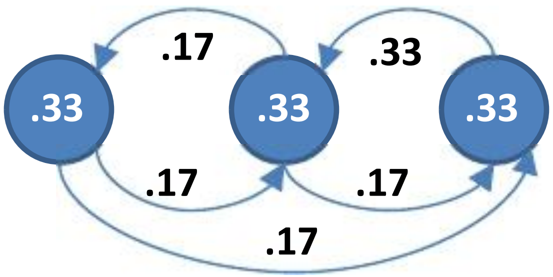
# PageRank in Pregel

```
class PageRankVertex {
 void compute(Iterator messages) {
   if (getSuperstep()> 0) {
     // recompute own PageRank from the neighborsmessages
     pageRank= sum(messages);
     setVertexValue(pageRank);
   }

   if (getSuperstep()< k) {
      // send updated PageRank to each neighbor
      sendMessageToAllNeighbors(pageRank/ getNumOutEdges());
   } else {
     voteToHalt();  // terminate
   }
}}
```

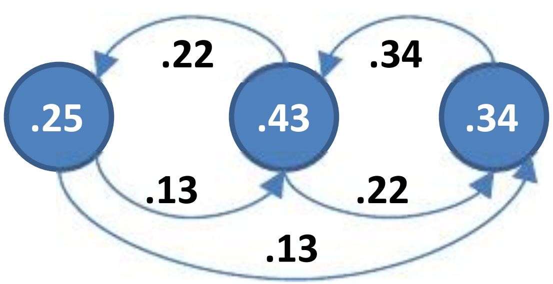
$$p_i = \sum_{j \in \{(j,i)\}} \frac{p_j}{d_j}$$

# PageRank toy example



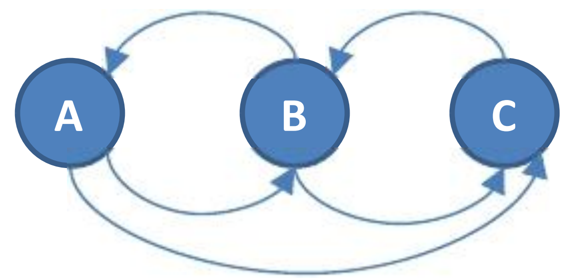

**Superstep 0**


**Superstep 1**




**Superstep 2**

**Input graph**



# Cool, where can I download it?

- Pregel is **proprietary**, but:
  - **Apache Giraph** is an open source implementation of Pregel
  - runs on **standard Hadoop infrastructure**
  - computation is executed **in memory**
  - can be a job in a **pipeline** (MapReduce, Hive)
  - uses **Apache ZooKeeper** for synchronization