University of BRISTOL

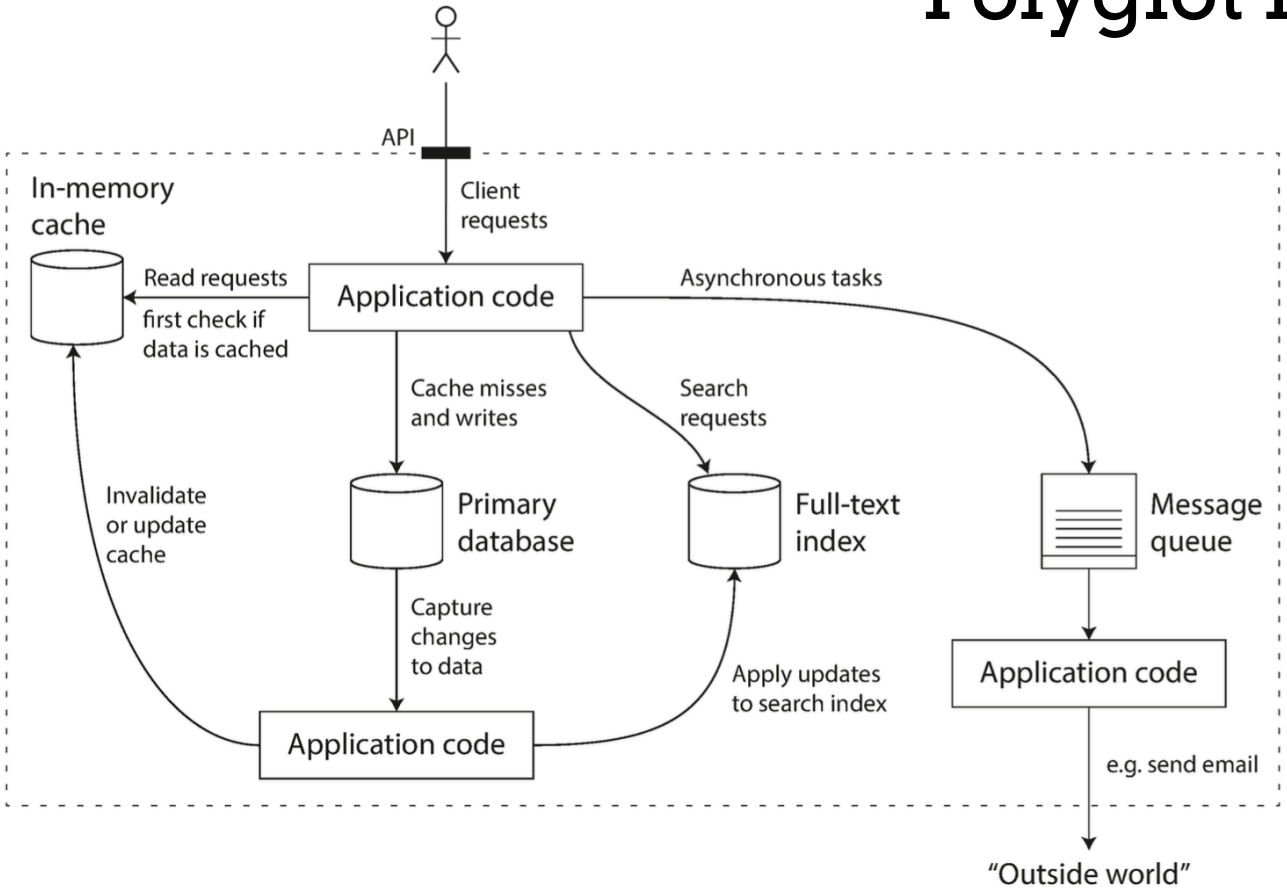
CAP Theorem, Graph DBs and Big Data Graph Analytics

Dr Dan Schien – COMSM0010

Lecture 19

bristol.ac.uk

Polyglot DB Systems



What's coming

Review

Setting the scene

- **Why NoSQL:** scale up vs scale out

Four main classes of NoSQL DB, with popular examples

- **Key-Value DBs**: DynamoDB; Memcached; Redis; Riak
- **Document DBs**: MongoDB; CouchDB
- **Column-Family DBs**: HBase; Cassandra
- **Graph DBs**: Giraph; Neo4j; Pregel ~~see next~~ this **lecture**

Choosing a NoSQL DB: which one? (And why stop at one?)

(Briefly) **NewSQL:** the future is the past

Choosing a NoSQL DB

Source: Perkin, *et al.*, 2018; Ch.9.

Review

Graph: 2D tables with rows & columns, strictly enforced data-types

Good for:

- Applications with *networks of relationships* at their core
- Social networks; bioinformatics: social med, genomics, & proteomics
- Reasoning about the graph on a single (large?) DB server/node

Less good for:

- Large-scale situations where partitioning across nodes is necessary

Outline

- CAP Theorem
- Basic Network Concepts
- Neo4J - Graph Database
- Bulk Synchronous Processing
- Big Data Graph Analytics: Pregel, Apache Giraph
- Bonus: Google’s PageRank

C, A and P

- A good cloud might seek to achieve these three things:
- Consistency (C):
  - All nodes should see the same data at the same time
- Availability (A):
  - Node failures do not prevent survivors from continuing to operate
- Partition-tolerance (P):
  - The system continues to operate despite network partitions
  - (A partition temporarily separates a system into two sub-systems that cannot communicate with each other.)

CAP Theorem

But… Eric Brewer's CAP Theorem states that a cloud service *cannot* simultaneously provide:

- Consistency
- Availability
- Partition-tolerance

There is always some kind of trade-off between these three properties.

System designers will have to decide how their system handles this trade-off.

Prof. Eric Brewer, UC Berkeley

CAP Theorem

Consider the following :

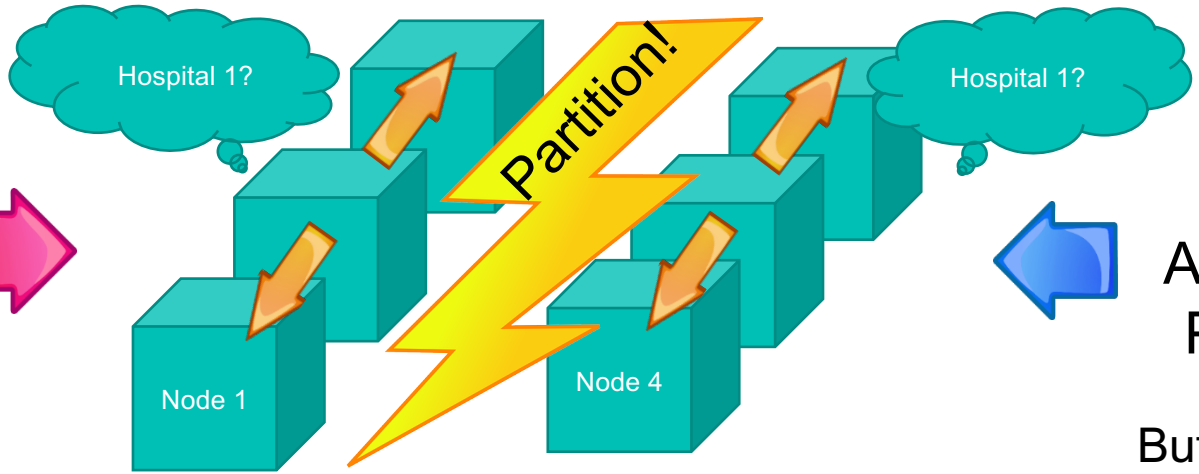
- Nodes *X* and *Y* suffer a partition: they are temporarily separated
- *X* wants to move forward with a job but gets no reply from *Y*
- *X* must either:
  - wait to hear back from *Y* (sacrificing Availability), or…
  - proceed without hearing back from *Y* (threatening Consistency), or…
  - never be partitioned from *Y* in the first place (losing Partition-tolerance)

(A formal proof of the CAP Theorem was published two years after Brewer's original lecture: Gilbert & Lynch, 2002, *ACM SIGACT News,* 33(2):51-59)

A Deadly Example

Car Accident Patient X

If we wait (we trade off A for C) our patient may die of their injuries!



A distributed system handling paramedic requests for hospital operating theatre slots

Should patient X be sent to hospital 1?

Car Accident Patient Y

But if we proceed (trade off C for A) our patient may reach a busy hospital and die!

Beyond “2 from 3”

- The CAP Theorem was sometimes understood to mean that a designer’s job was to pick “2 from 3”:
  - Pick C and P (sacrifice Availability)
  - Pick A and P (sacrifice Consistency)
  - Pick C and A (sacrifice Partition-Tolerance)

But P is *required* in the cloud; is not a choice – network faults will happen: so the “C and A” option is out.

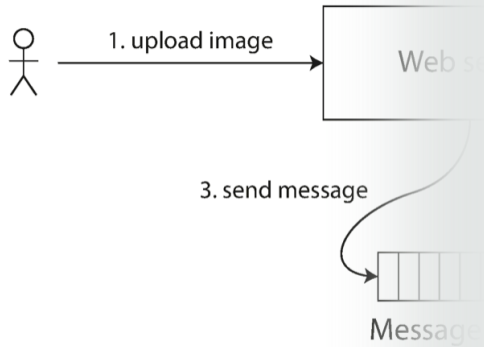
Also C, A & P are not binary concepts. Each is a matter of degree.

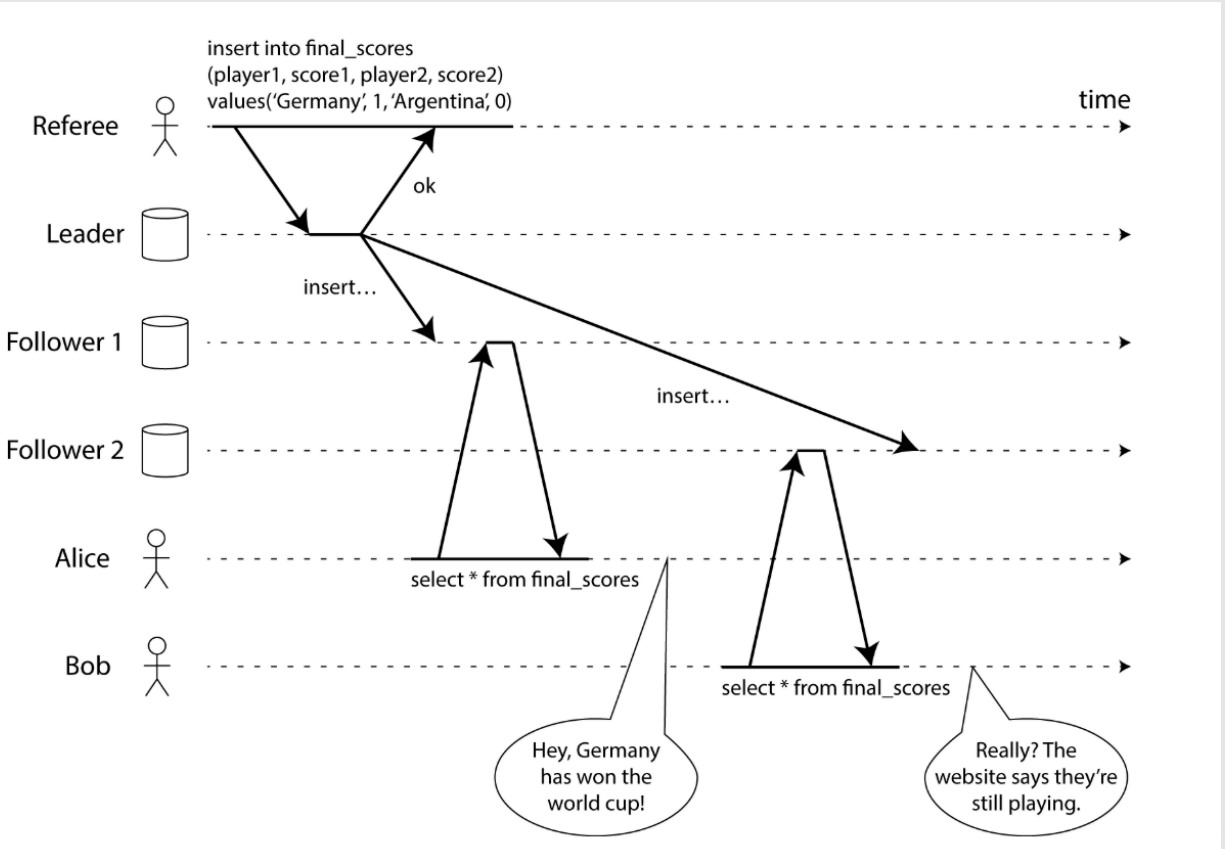
In fact CAP only rules out one corner of the design space:

- “perfect availability & consistency in the presence of partitions”

Linearizability

- Create the appearan
- Ensures a recency g
  read, all subsequen
  overwritten again.





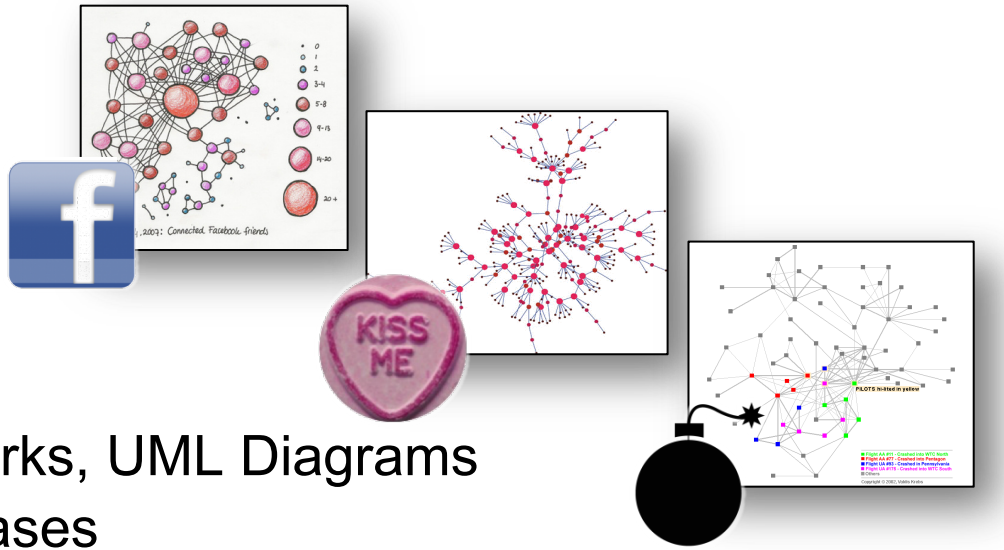
https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html

Graphs and Graph DBs

Networks/Graphs

Networks are a generic way of representing structure:

- Family Trees
- Road and Rail Networks
- Neural Networks
- Social and Sexual Networks
- Terrorist Networks

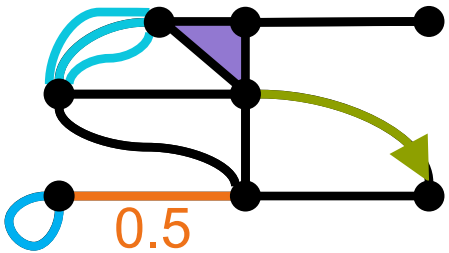


In Comp Sci:

- The Internet, Semantic Networks, UML Diagrams
- Increasingly: Data and Databases

Basics

A network (often called a *graph*), is made up of a set of individual nodes or points (often called *vertices*), where some nodes are connected by links or arcs or lines (often called *edges*).



Edges are *undirected* (2-way) or *directed* (1-way).

Edges may be *self-connections*.

Edges may be *weighted*.

Edges may even be *multi-edges* or *hyper-edges*.

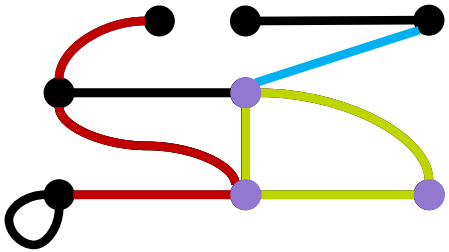
If we consider the world wide web as a graph, which of these properties do its edges possess?

Basics

A graph may be fragmented into separate *components*.

Or may be *connected*, i.e., when the number of components is 1.

A *fully-connected* or *complete* graph is one in which every pair of nodes is connected together *directly*.



A *clique* is a sub-graph that is complete.

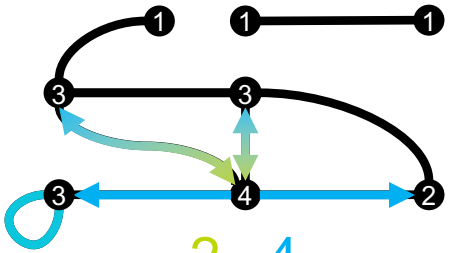
A *path* is a sequence of edges connecting a sequence of nodes. A *cycle* is a closed path.

∴ In a component there is a path between *every* pair of nodes.

The *distance* between nodes is the length of their shortest path.

Degree

The number of ways to arrive or leave a node is the node's *degree*.
This is often equivalent to counting a node's neighbours.
(But for a *self-connected* node, these two interpretations differ and typically we "count self-connections twice".)

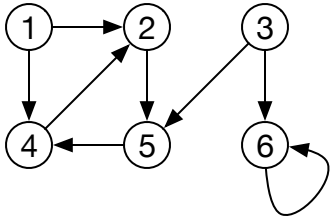


A node's *in-degree* counts ways to arrive at it
A node's *out-degree* counts ways to leave it
On an *undirected* graph in-degree=out-degree

On a directed graph *in-degree* and *out-degree* may differ.
A network where all nodes have the same degree is a *regular* graph.

Representing Graphs - Adjacancy List

- Array of lists, j in Adj[i] if edge i->j

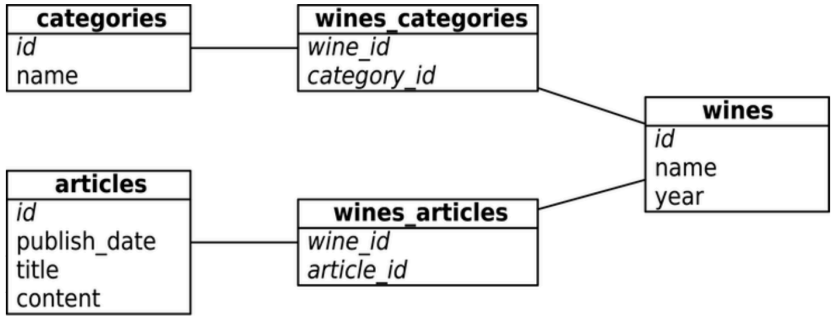


1	2	4
2	5	
3	6	5
4	2	
5	4	
6	6	

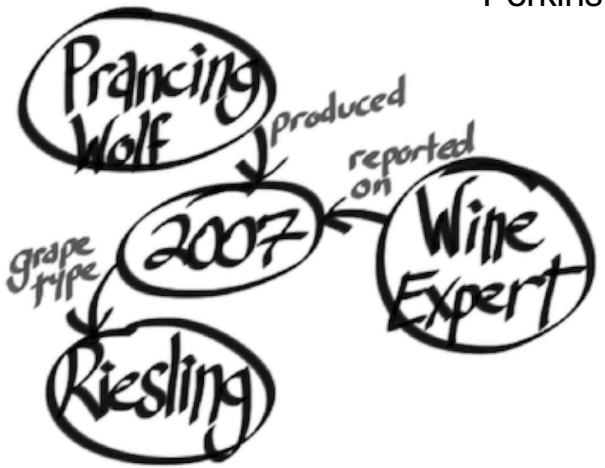
- O(n+m) space
- Look up edge O(n)

Graph DB concepts

- DB with Relationships being first-class objects
- Queries on relationships are much faster than with alternatives
  - ... that require multiple index lookups



Perkins. 2018



Neo4j Graph DB

- GraphDB (written in Java, with drivers for all languages)
- Associative data sets
  - A node with edges pointing to related nodes
  - Forming a property graph
- Clustered
- Cypher query language
- Fast on connected data
- Weaknesses
  - Sharding
  - Binary data

Cypher query language

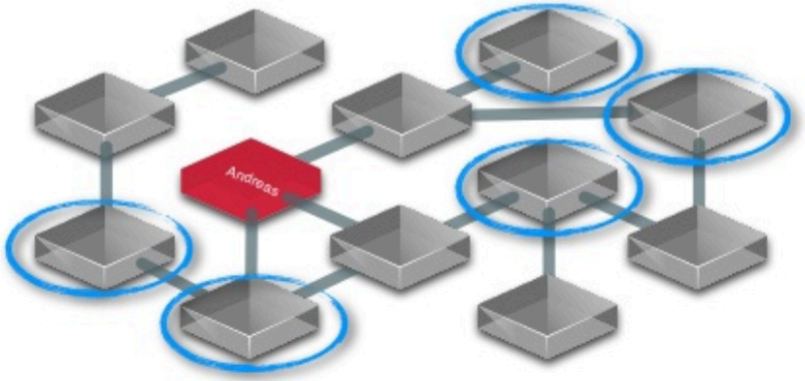
- Declarative - what, not how
- `Match` - Describe pattern
- `With` - Pipe inline processing and aggregation
- `Where` – filter
- `Sort`
- `Return` - result rows

- Parentheses for Nodes
- Arrows for relationships

  (a)-[:KNOWS]-> (b)

You traverse the graph

```
// then traverse the relationships
MATCH (me:Person {name:'Andreas'})-[:FRIEND]-(friend)
                                  -[:FRIEND]-(friend2)
RETURN friend2
```



Concrete Example


```
MATCH (:Country {name:"Sweden"})
      <-[:REGISTERED_IN]-(c:Company)
      <-[:WORKS_AT]-(p:Person:Developer)
WHERE p.age < 42
WITH c, count(p) as cnt,
     collect(p.empId) as emp_ids
WHERE cnt > 12
RETURN c.name AS company_name,
       extract(id2 in
          filter(id1 in emp_ids
             WHERE id1 =~ "...-.*")
       | substr(id2,4,size(id2)-1)]
       AS last_emp_id_digits
ORDER BY length(last_emp_id_digits) DESC
SKIP 5 LIMIT 10
```

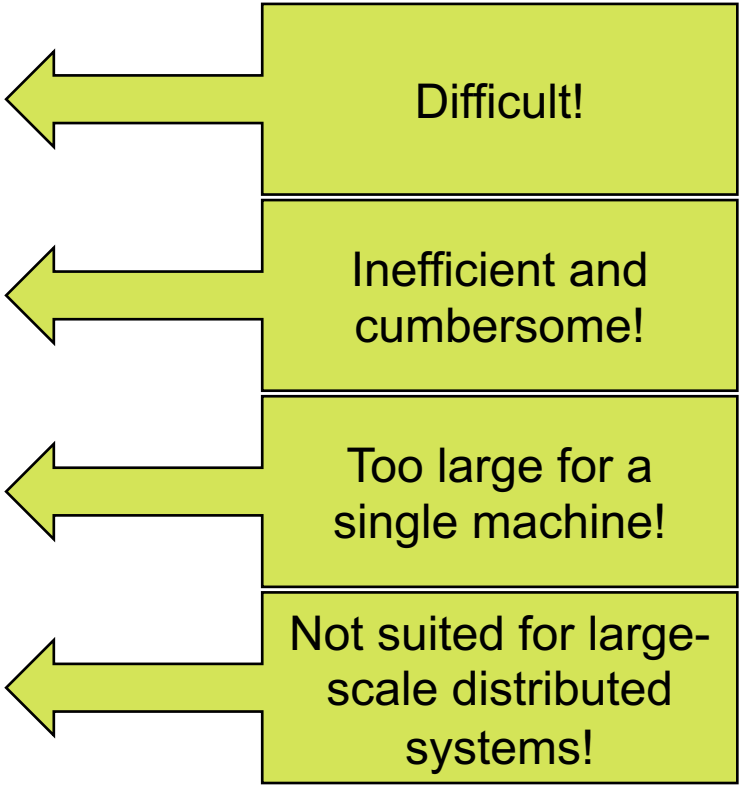
Typical Applications

- Fraud detection and analytics
- Network and database infrastructure monitoring
- Recommendation engines
- Social networks
- Knowledge graph

Big Data Graph Analytics

Graph Processing options?

- Create a custom distributed infrastructure for each new algorithm
- Rely on existing distributed computing platforms to perform MapReduce
- Use a single-computer graph algorithm library (e.g., BGL, LEDA)
- Use a parallel graph processing system (e.g., Parallel BGL, CGMGraph)



Pregel

Given the centrality of network analysis to Google's core business, their interest in efficient graph processing should be unsurprising…

Pregel is Google's framework for achieveing this.

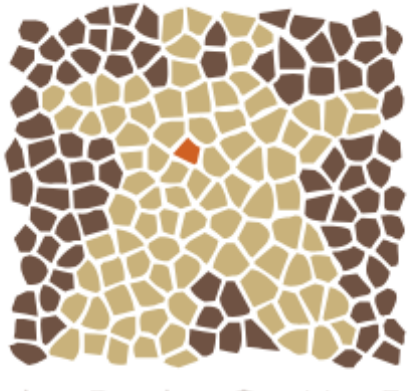
It provides:

- High scalability
- Fault-tolerance
- Flexibility in expressing arbitrary graph algorithms

It is based on Valiant's (1990) Bulk Synchronous Parallel (BSP) model.

Apache Giraph

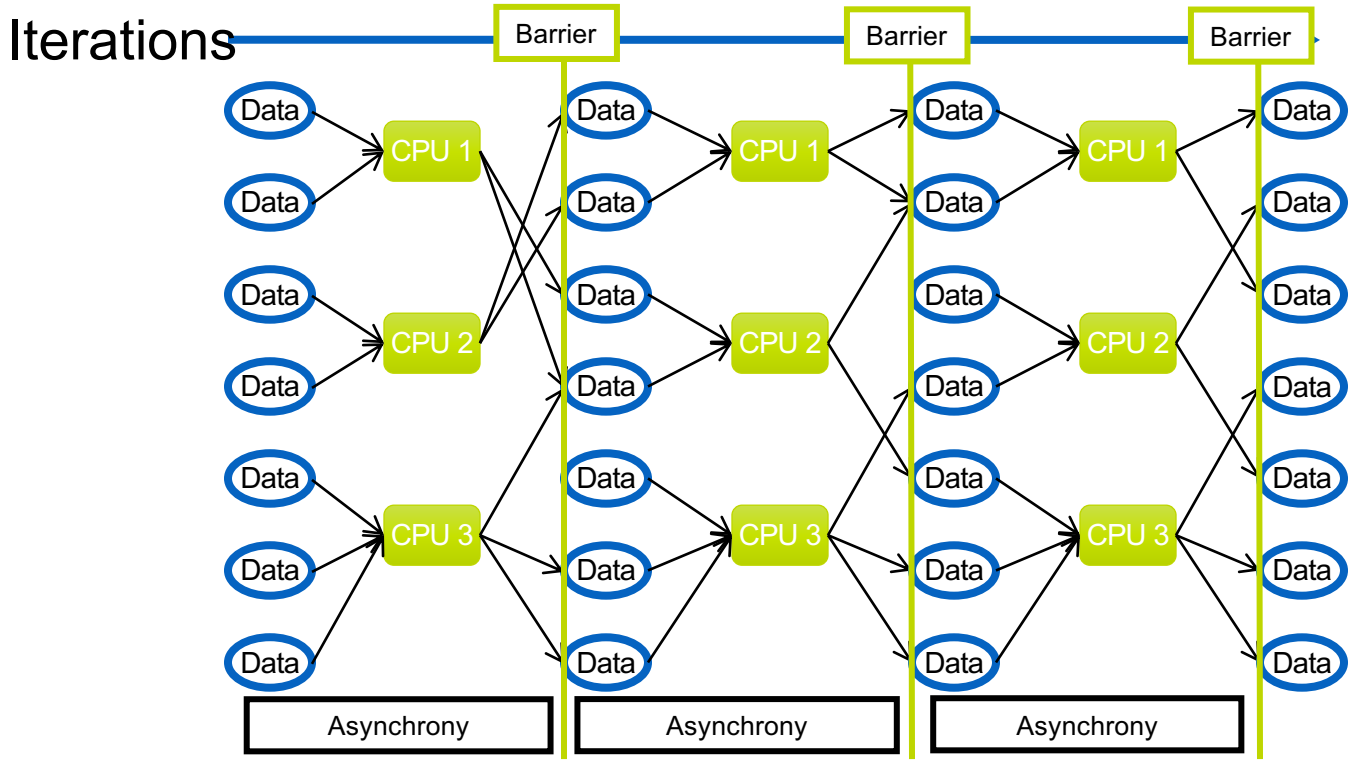
- Facebook's version of Pregel
- Based on Malewicz et. al "Pregel: A System for Large-Scale Graph Processing" (2010)
- Java
- Integrates into Hadoop ecosystem
  - Efficiently loading data from Hbase
- Like Pregel uses Bulk Synchronous Parallel (BSP)



BSP supersteps

- Algorithm model
- 3 Steps
- *Concurrent computation*: computation exclusively local to each node
- *Communication*: messaging between nodes
- *Barrier synchronisation:* checkpoints that globally block nodes before processing signalling received in a superstep

Bulk Synchronous Parallel Model

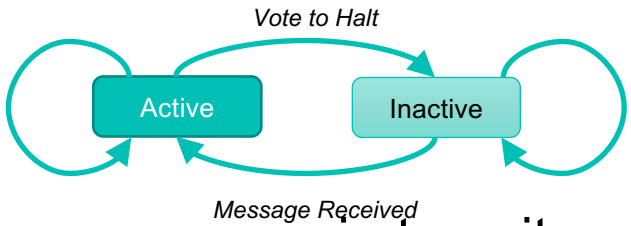


Entities and Supersteps

- Pregel operates over a directed graph:
  - Each vertex is associated with a modifiable user-defined value
  - Each edge has a value, and links a source vertex to a destination vertex
- Runs a sequence of “supersteps” analogous to MapReduce rounds
  - During each superstep, a user-defined function is executed at each vertex
  - This function can read incoming messages sent at the last superstep…
  - …and send outgoing messages that will be received at the next superstep
  - The function can modify the state of its vertex…
  - …and also modify its outgoing edges, changing the graph’s topology

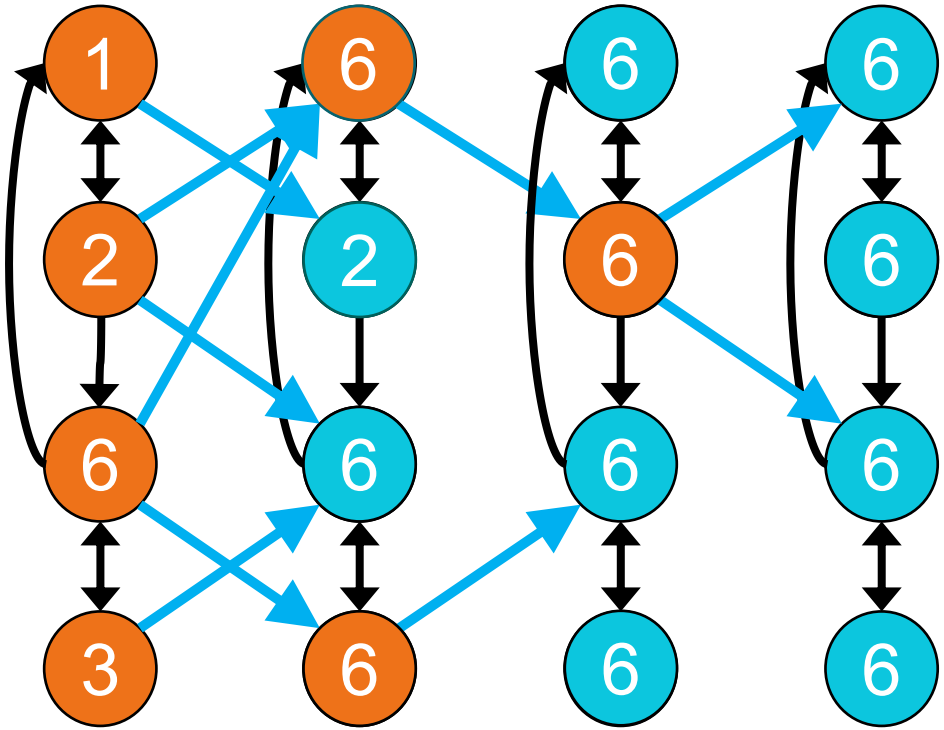
Algorithm Termination

- Algorithm termination is based on every node voting to halt
  - Initially every node is active
  - All active nodes participate in the computation of a superstep
  - A node deactivates itself by voting to halt and enters an inactive state
  - A node can return to the active state if it receives an appropriate message
- The program terminates when all nodes are simultaneously inactive and there are no messages in transit



Vertex State Machine

Agreeing on Max Value in a Graph



Four steps of Pregel algorithm execution

Nodes eventually agree on the maximum value.

Green nodes are active
Black arrows indicate connectivity
Blue arrows are messages
Red nodes have voted to halt

Pregel Code

- A Pregel program is written by subclassing the vertex class:

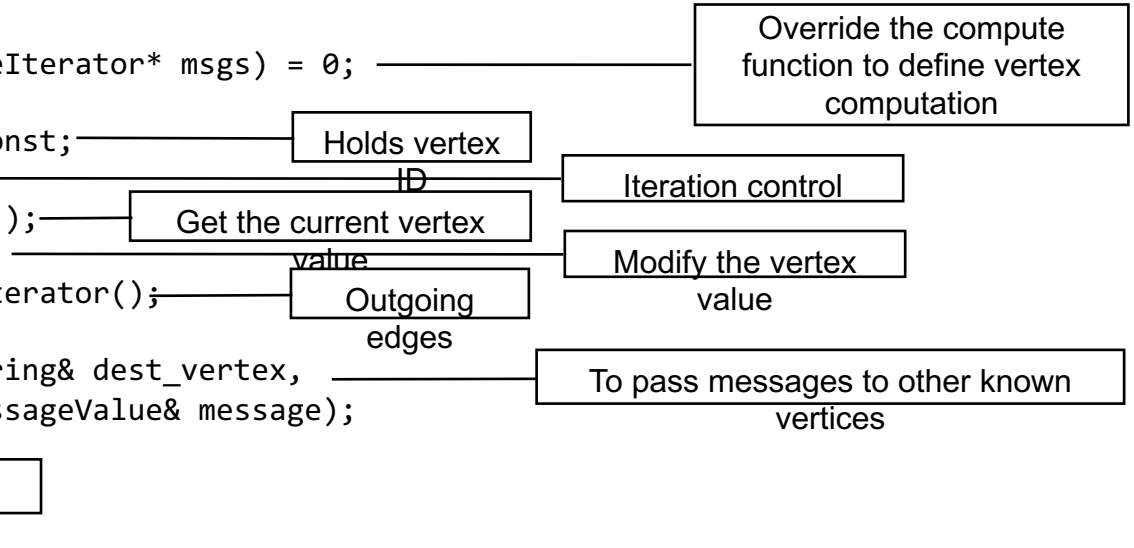
```
template <typename VertexValue, typename EdgeValue, typename MessageValue>

class Vertex {
public:
    virtual void Compute(MessageIterator* msgs) = 0;

    const string& vertex_id() const;
    int64 superstep() const;
    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                       const MessageValue& message);

    void VoteToHalt();
};
```



Example: Pregel Max Value Code

```
Class FindMaxVertex
        : public Vertex<double, void, double> {
  public:
        virtual void Compute(MessageIterator* msgs) {
                int currMax = GetValue();
                SendMessageToAllNeighbors(currMax);
                for ( ; !msgs->Done(); msgs->Next()) {
                        if (msgs->Value() > currMax)
                                currMax = msgs->Value();
                }
                if (currMax > GetValue())
                        *MutableValue() = currMax;
                else VoteToHalt();
        }
};
```

Execution of a Pregel Program

- Copies of the program are distributed across all workers
  - One copy is designated as a the master
- The master partitions the graph and assigns workers their respective partition(s) along with portions of the input
- The master coordinates execution of supersteps and all messaging
- The 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

Fault Tolerance in Pregel

- Checkpointing: At the start of a superstep the master may instruct 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
- Constraints on operation order within a superstep help prevent collisions and inconsistency, but at the expense of performance.

Pregel vs. MapReduce

- Pregel: think like a node; MapReduce: think like a graph
- Pregel: smaller messages; more asynchronous; less disk read/write
- Pregel: support for iteration; MapReduce: no iteration support
- Vertex-centric approaches: limited to graphs; MapReduce: general
- Master node bottleneck: synchronization barriers & checkpointing
- As with other cloud computing technologies, more recent (vertex/graph) approaches add improved performance, generality, etc.
  - Apache Hama, Apache Giraf, GraphLab, GRACE, GraphX in Spark, etc.

Background Resources

- Basics of Networks and Networks Science
  - Network Literacy: Essential Concepts and Core Ideas
  - Networks: Structure and Dynamics by Erzsébet Ravasz Regan
- Basics of matrices and PageRank:
  - Tanase & Rad (Cornell) provide some nice introductory material on their *The Mathematics of Web Search* webpages…
  - [3] PageRank itself
  - [2] Directed graphs and matrices
  - [1] Basics of linear algebra for matrices, eigenvalues, eigenvectors, etc.

Bonus Material

Adjacency Matrices

A network's structure can be represented by its adjacency matrix, *A*:

For row *i* and column *j*, A_{ij} represents the number of edges from node *i* to node *j*.

The sum of row *i* is the *out-degree* of *i*.

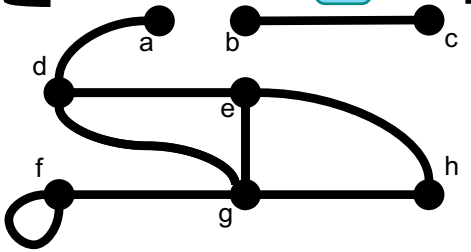
The sum of column *j* is the *in-degree* of *j*.

How can we see that this graph is undirected?

(For a weighted graph, A_{ij} represents the *weight* of the connection from *i* to *j*.)

O(n*n) memory, lookup O(1)

	a	b	c	d	e	3	g	h
a	0	0	0	1	0	0	0	0
b	0	0	1	0	0	0	0	0
c	0	1	0	0	0	0	0	0
3	1	0	0	0	1	0	1	0
e	0	0	0	1	0	0	1	1
f	0	0	0	0	0	2	1	0
g	0	0	0	1	1	1	0	1
h	0	0	0	0	1	0	1	0



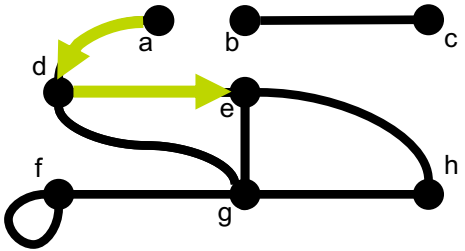
Adjacency Matrices^2

When an adjacency matrix is raised to the power 2…

	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	**1**	0	0	0	**1**	0	**1**	0
e	0	0	0	**1**	0	0	**1**	**1**
f	0	0	0	0	0	**2**	**1**	0
g	0	0	0	**1**	**1**	**1**	0	**1**
h	0	0	0	0	**1**	0	**1**	0

^{2} =

	a	b	c	d	e	f	g	h
a	**1**	0	0	0	**1**	0	**1**	0
b	0	**1**	0	0	0	0	0	0
c	0	0	**1**	0	0	0	0	0
d	0	0	0	**3**	**1**	**1**	**1**	**2**
e	**1**	0	0	**1**	**3**	**1**	**2**	**1**
f	0	0	0	**1**	**1**	**5**	**2**	**1**
g	**1**	0	0	**1**	**1**	**2**	**3**	**1**
h	0	0	0	**2**	**1**	**1**	**1**	**2**



…A_{ij} represents the number of paths of length 2 from *i* to *j*

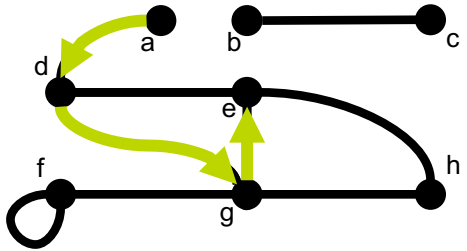
Adjacency Matrices

When an adjacency matrix is raised to the power *n*…

	a	b	c	d	e	f	g	h
a	0	0	0	1	0	0	0	0
b	0	0	1	0	0	0	0	0
c	0	1	0	0	0	0	0	0
d	1	0	0	0	1	0	1	0
e	0	0	0	1	0	0	1	1
f	0	0	0	0	0	2	1	0
g	0	0	0	1	1	1	0	1
h	0	0	0	0	1	0	1	0

^{3} =

	a	b	c	d	e	f	g	h
a	0	0	0	3	1	1	1	2
b	0	0	1	0	0	0	0	0
c	0	1	0	0	0	0	0	0
d	3	0	0	2	6	3	7	2
e	1	0	0	6	4	4	6	5
f	1	0	0	3	4	12	8	3
g	1	0	0	5	5	7	5	4
h	2	0	0	2	5	3	6	2



…A_{ij} represents the number of paths of length *n* from *i* to *j*

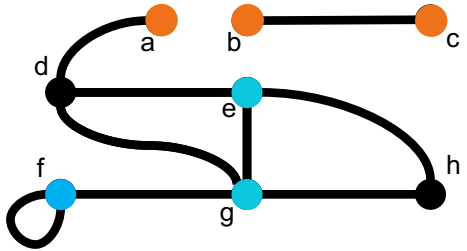
Adjacency Matrices

When an adjacency matrix is raised to the power *n*…

	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	**1**	0	0	0	**1**	0	**1**	0
e	0	0	0	**1**	0	0	**1**	**1**
f	0	0	0	0	0	**2**	**1**	0
g	0	0	0	**1**	**1**	**1**	0	**1**
h	0	0	0	0	**1**	0	**1**	0

5 =

	a	b	c	d	e	f	g	h
a	*2*	0	0	16	11	13	13	13
b	0	*0*	1	0	0	0	0	0
c	0	1	*0*	0	0	0	0	0
d	16	0	0	26	42	39	**53**	24
e	11	0	0	42	*40*	47	**52**	36
f	13	0	0	39	47	***86***	**71**	36
g	11	0	0	40	42	**59**	***54***	35
h	13	0	0	24	36	36	46	22



…*central* nodes and *loop* nodes score higher than *peripheral* nodes.

PageRank

Consider a web page, *a*:

- How important is the page relative to other pages on the web?
- If there's a link from page *b* to page *a*, then *a* must be important.
- If *b* is itself an important web page, *a* must be really important.
- If *b* only links to a few pages, *a* must be really, really important.

Can network ideas suggest a measure that captures these factors?

Sergey Brin and Larry Page showed that the answer was yes.

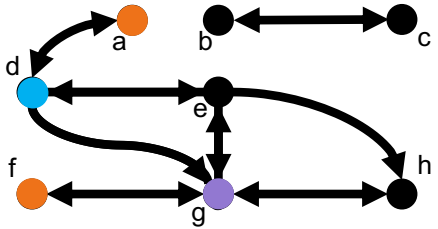
They called their measure PageRank and used it to drive a new search engine: Google.

Transition Matrix

First, build a transition matrix, M, where $M_{ij} = A_{ij} / \sum_i A_{ij}$

	a	b	c	d	e	f	g	h			
a	0	0	0	**1**	0	0	0	0	×	⅛	=
b	0	0	**1**	0	0	0	0	0		⅛	
c	0	**1**	0	0	0	0	0	0		⅛	
d	⅓	0	0	0	⅓	0	⅓	0		⅛	
e	0	0	0	⅓	0	0	⅓	⅓		⅛	
f	0	0	0	0	0	0	**1**	0		⅛	
g	0	0	0	0	⅓	⅓	0	⅓		⅛	
h	0	0	0	0	0	0	**1**	0		⅛	

0.042	a
0.125	b
0.125	c
0.167	d
0.083	e
0.042	f
0.333	g
0.083	h



Random surfers at page d, give ⅓ of their clicks to pages a, e, and g.

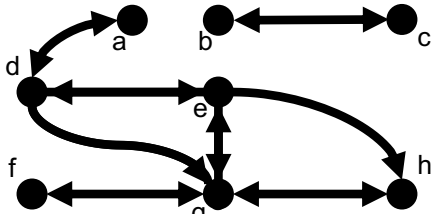
Initially ⅛ of the surfers are at each of the 8 pages, but after 1 step…
…some nodes have **more** surfers at them than **others**.

2 Steps…

If we *square* the transition matrix, we can look at 2 step paths…

	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	⅓	0	0	0	⅓	0	⅓	0
e	0	0	0	⅓	0	0	⅓	⅓
f	0	0	0	0	0	0	**1**	0
g	0	0	0	0	⅓	⅓	0	⅓
h	0	0	0	0	0	0	**1**	0

^{2} × ⅛ = 0.056 (a), 0.125 (b), 0.125 (c), 0.069 (d), 0.167 (e), 0.111 (f), 0.208 (g), 0.139 (h)



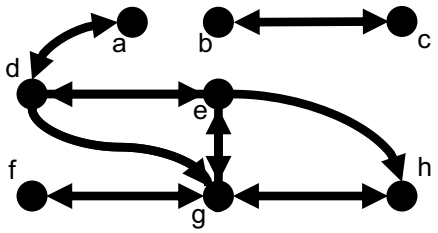
"How likely is a random surfer to be at each page after two steps…"

So to get a sense of how likely a random surfer is to be at any page, we just need to extrapolate to paths of infinite length…

3 Steps…

	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	⅓	0	0	0	⅓	0	⅓	0
e	0	0	0	⅓	0	0	⅓	⅓
f	0	0	0	0	0	0	**1**	0
g	0	0	0	0	⅓	⅓	0	⅓
h	0	0	0	0	0	0	**1**	0

$\times$ 3 $\times$ $\begin{bmatrix} ⅛ \\ ⅛ \\ ⅛ \\ ⅛ \\ ⅛ \\ ⅛ \\ ⅛ \\ ⅛ \end{bmatrix}$ = $\begin{bmatrix} 0.023 \\ 0.125 \\ 0.125 \\ 0.111 \\ 0.093 \\ 0.069 \\ 0.329 \\ 0.125 \end{bmatrix}$ a b c d e f g h

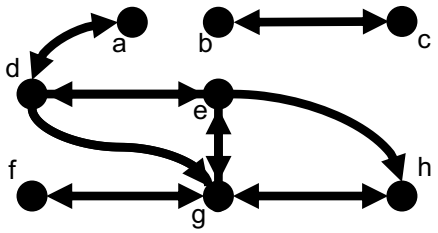


So to get a sense of how likely a random surfer is to be at any page, we just need to extrapolate to paths of infinite length…

4 Steps…

	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	⅓	0	0	0	⅓	0	⅓	0
e	0	0	0	⅓	0	0	⅓	⅓
f	0	0	0	0	0	0	**1**	0
g	0	0	0	0	⅓	⅓	0	⅓
h	0	0	0	0	0	0	**1**	0

^{4} × ⅛ = 0.037 (a), 0.125 (b), 0.125 (c), 0.054 (d), 0.147 (e), 0.110 (f), 0.262 (g), 0.140 (h)



So to get a sense of how likely a random surfer is to be at any page, we just need to extrapolate to paths of infinite length…

10 Steps…

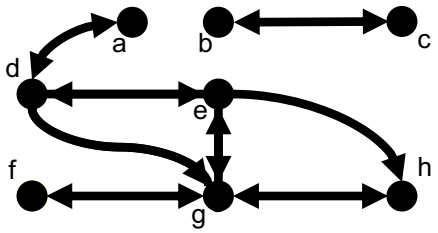
	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	⅓	0	0	0	⅓	0	⅓	0
e	0	0	0	⅓	0	0	⅓	⅓
f	0	0	0	0	0	0	**1**	0
g	0	0	0	0	⅓	⅓	0	⅓
h	0	0	0	0	0	0	**1**	0

×

1
0
⅛
⅛
⅛
⅛
⅛
⅛
⅛
⅛

=

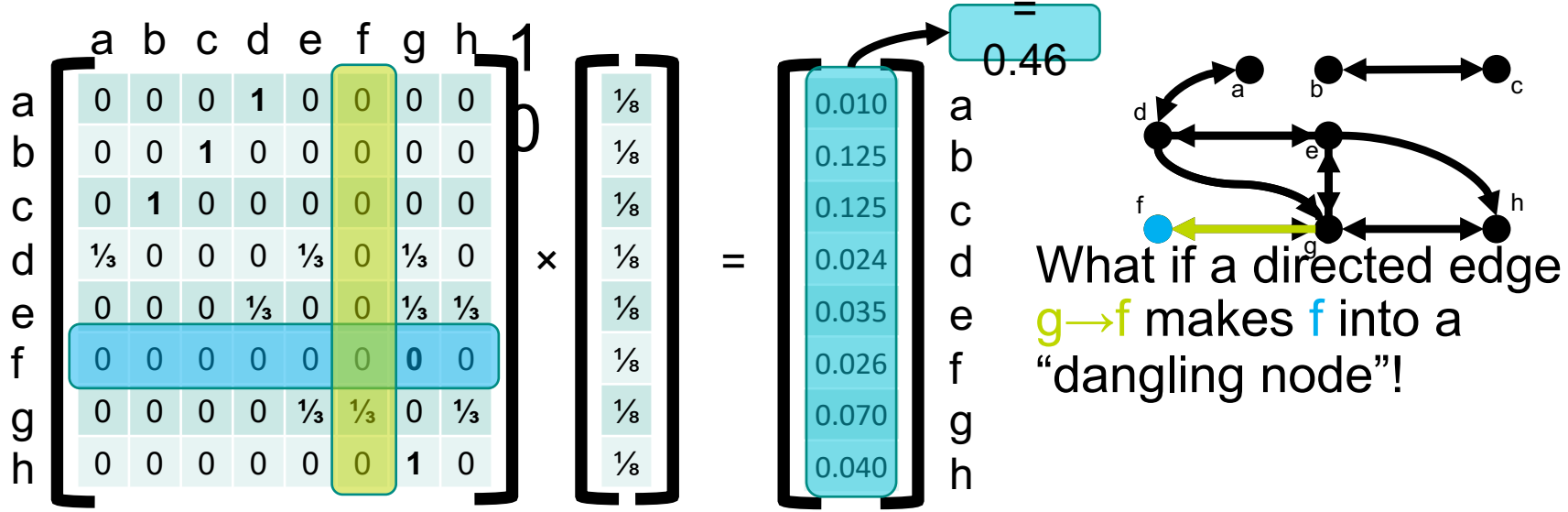
0.022	a
0.125	b
0.125	c
0.058	d
0.124	e
0.102	f
0.303	g
0.141	h



For relatively low exponents the distribution tends to converge to a good estimate of how likely a random surfer is to be at each node…

Dangling Nodes

But to make this idea work we have to deal with some problems…



What if a directed edge g→f makes f into a "dangling node"!

Dangling nodes accumulate surfers but don't redistribute them.

So, they break the random surfer model… …sum(distribution)< 1!

"Teleportation"

A fix for dangling nodes "teleports" their surfers to random pages:

	a	b	c	d	e	f	g	h
a	0	0	0	**1**	0	0	0	0
b	0	0	**1**	0	0	0	0	0
c	0	**1**	0	0	0	0	0	0
d	⅓	0	0	0	⅓	0	⅓	0
e	0	0	0	⅓	0	0	⅓	⅓
f	⅛	⅛	⅛	⅛	⅛	⅛	⅛	⅛
g	0	0	0	0	⅓	⅓	0	⅓
h	0	0	0	0	0	0	**1**	0

1
0

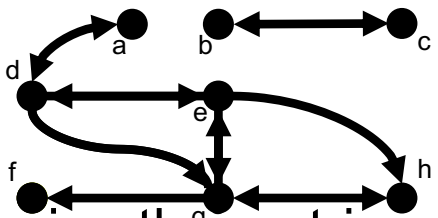
× v

a	⅛
b	⅛
c	⅛
d	⅛
e	⅛
f	⅛
g	⅛
h	⅛

=

a	0.058	
b	0.172	
c	0.172	
d	0.109	
e	0.111	
f	0.080	
g	0.186	
h	0.112	

= 1.00



Damping the matrix:

- $M'_{ij} = (1\text{-}p)A_{ij} + p.v_i$
- damping factor p=0.15

More generally, "damping" imposes transitions between all nodes.

- A low probability that a surfer jumps to a randomly chosen page.

Damping

All “off-network” transitions now occur with a small probability: ε

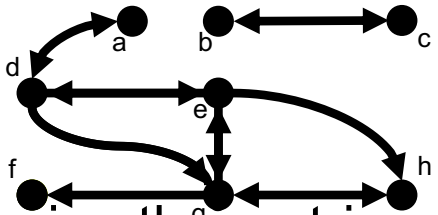
	a	b	c	d	e	f	g	h
a	ε	ε	ε	**1′**	ε	ε	ε	ε
b	ε	ε	**1′**	ε	ε	ε	ε	ε
c	ε	**1′**	ε	ε	ε	ε	ε	ε
d	⅓′	ε	ε	ε	⅓′	ε	⅓′	ε
e	ε	ε	ε	⅓′	ε	ε	⅓′	⅓′
f	⅛′	⅛′	⅛′	⅛′	⅛′	⅛′	⅛′	⅛′
g	ε	ε	ε	ε	⅓′	⅓′	ε	⅓′
h	ε	ε	ε	ε	ε	ε	**1′**	ε

×

1	*v*
0	⅛
0	⅛
0	⅛
0	⅛
0	⅛
0	⅛
0	⅛
0	⅛

=

0.058	a
0.172	b
0.172	c
0.109	d
0.111	e
0.080	f
0.186	g
0.112	h



Damping the matrix:

- $M'_{ij} = (1\text{-}p)A_{ij} + p.v_i$
- damping factor p=0.15

“On-network” transitions are scaled down (′), re-normalising M'.

Eigenvalues and Eigenvectors

Teleportation & damping fix dangling nodes & connect the graph, ensuring that the resulting matrix, M', has some nice properties:

- It is subject to the *Perron-Frobenius Theorem* which ensures that:

  The PageRank vector, v^*, for a web graph with transition matrix M, and damping factor p, is the unique probabilistic *eigenvector* of the damped, dangle-less matrix M', corresponding to the *eigenvalue* 1.
- This means that we can calculate v^* directly by solving: $M' \cdot v^* = v^*$
- Unfortunately, for a graph the size of the web, this is intractable.
- So, we have to approximate v^* by calculating $v \cdot M'^n$ as before.
- Luckily M' is *sparse*, meaning $v \cdot M'^n$ converges quickly (i.e., low n).

Example: Pregel PageRank

```
Class PageRankVertex
        : public Vertex<double, void, double> {
  public:
        virtual void Compute(MessageIterator* msgs) {
                if (superstep() >= 1) {
                        sum := 0;
                        for ( ; !msgs->Done(); msgs->Next()) {
                                sum += msgs->Value();
                }
                *MutableValue() := p/NumVertices() + (1-p)*sum;
                if (superstep() < 30) {
                        n := GetOutEdgeIterator().size();
                        sendMessageToAllNeighbors(GetValue() / n);
                } else { VoteToHalt(); }
        }
};
```

Sum contributions from neigbours

Damp the sum and add

30 is maximum

N = no. neighbours linked to

Divide PageRank amongst neighbours

Thanks!