# GRAPHS IN THE CLOUD

## CLOUD COMPUTING


Dr. Atm Shafiul Alam

a.alam@qmul.ac.uk

**Queen Mary University of London**

**School of Electronic Engineering and Computer Science**



**Thanks to Dr. Gareth Tyson and Dr. Félix Cuadrado who created the original version of these slides**

# In last weeks' sessions, we…

- Learnt about relational databases and their limitations
- ACID & CAP theory
- NoSQL databases
  - memcached
  - Cassandra

# ACID Database Properties

- **Atomicity**
  - Transactions must act as a whole, or not at all
  - No changes within  a transaction can persist if any change within the transaction fails
- **Consistency**
  - Changes made by a transaction respect all database integrity constraints and the database remains in a consistent state at the end of a transaction
- **Isolation**
  - Transactions cannot see changes made by other concurrent transactions ,
  - Concurrent transactions leave the database in the same state as if the transactions were executed serially
- **Durability**
  - Database data are persistent,
  - Changes made by a transaction that completed successfully are stored permanently

# CAP Theorem

- **Strong Consistency** - A service is considered to be ***strongly consistent*** if after an update operation of some writer all readers see his updates in some shared data source (all nodes containing replicas of a datum have the same value)
- **Availability** - A service is considered to have a ***high availability*** if it allows read and write operations a high proportion of time (even in the case of a node crash or some hardware or software parts are down due to upgrades)
- **Network partition tolerance** is the ability of a service to perform expected operations in the presence of a network partition, unless the whole network crashes . A network partition occurs if two or more “islands” of network nodes cannot connect to each other. Dynamic addition and removal of nodes is also considered to be a network partition.)

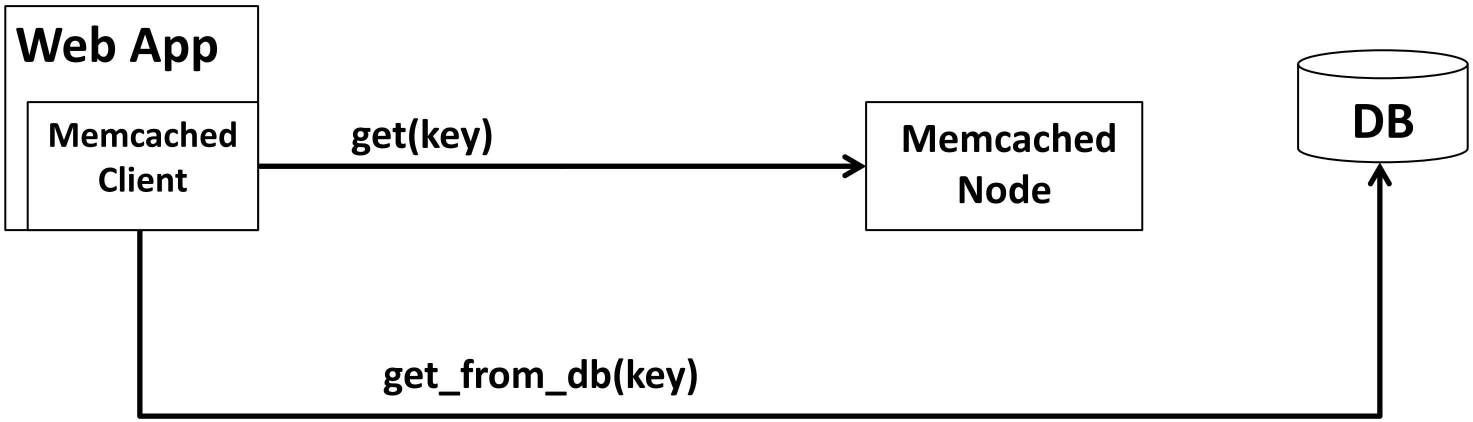
# Strong vs. Eventual Consistency

- Strong Consistency: Replicas update linearly in the same total order
  - After the update completes, any subsequent access will return the same updated value.
- Eventual Consistency: Replicas may not converge to the same total order”
  - Update is accepted by local node
  - Local node propagates update to replica nodes

# Data Partitioning and Replication

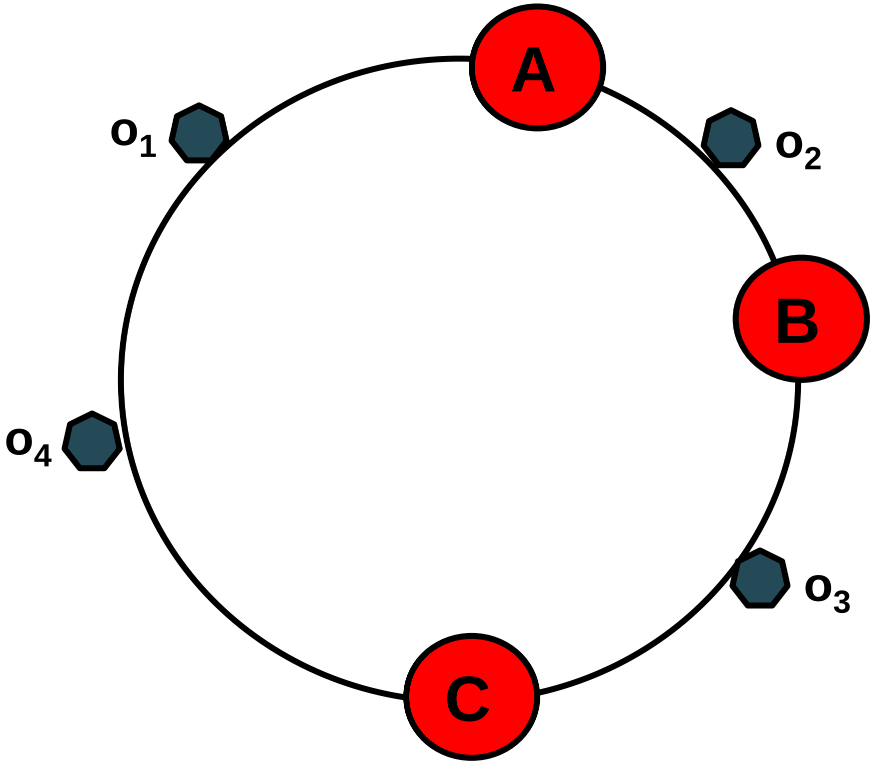
- NoSQLs relax ACID to allow better scalability, performance, etc.
- NoSQL DBs tend to maximise availability when partitioning
- There are a number of techniques to achieve data partitioning and replication:
  - Memory caches,
  - Separating reads from writes,
  - Clustering, sharding

# Memcached use with standard DB




```
public String getFoo(String fooId)
        String foo = memcached.get(fooId)
        if foo != null return foo
                foo = fetchFooFromDatabase(fooId)
                memcached.set(fooId, foo)
        return foo
        end
```

# Cassandra

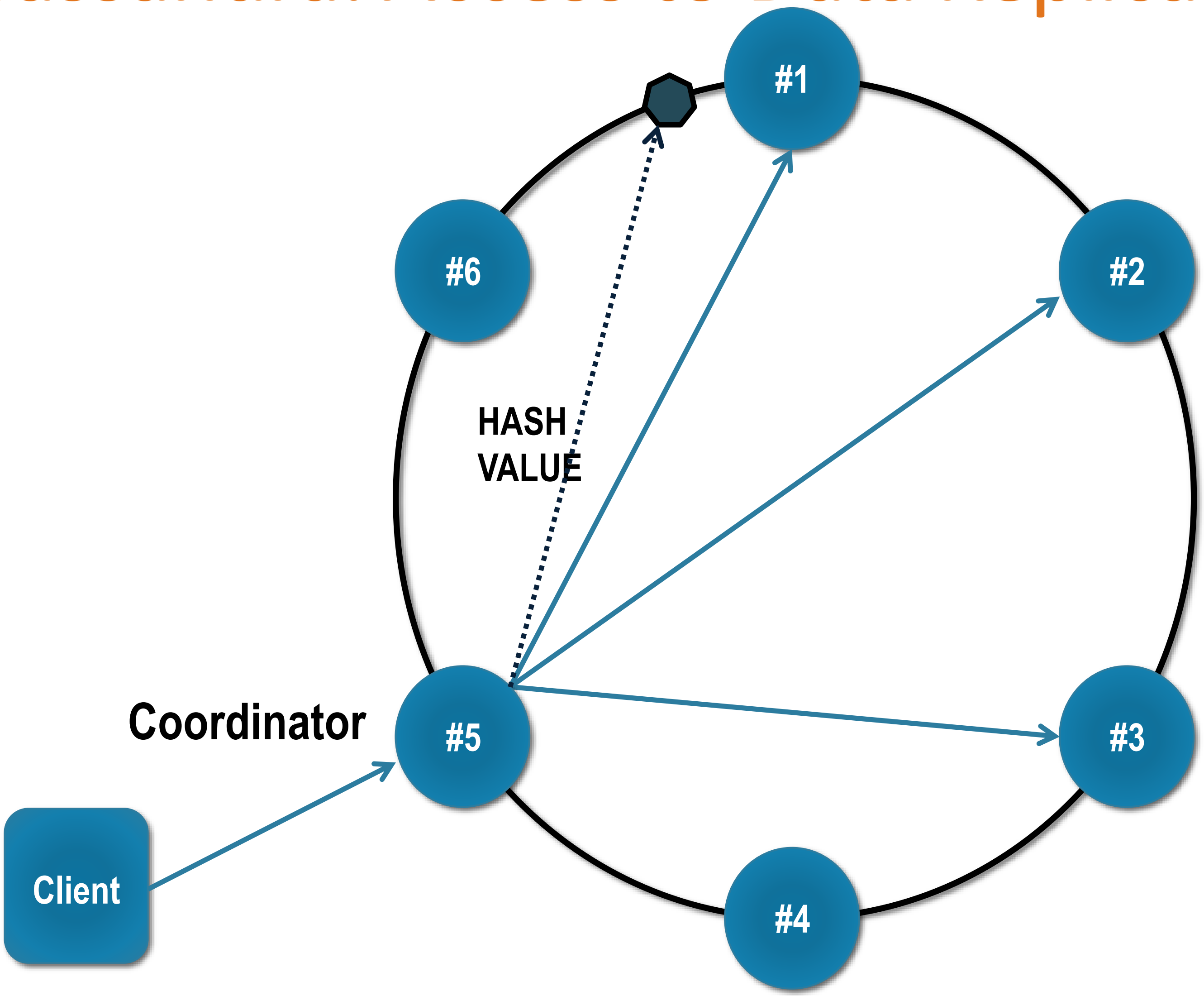



Objects $o_1$ and $o_4$ are stored on the node A

Object $o_2$ is stored on the node B

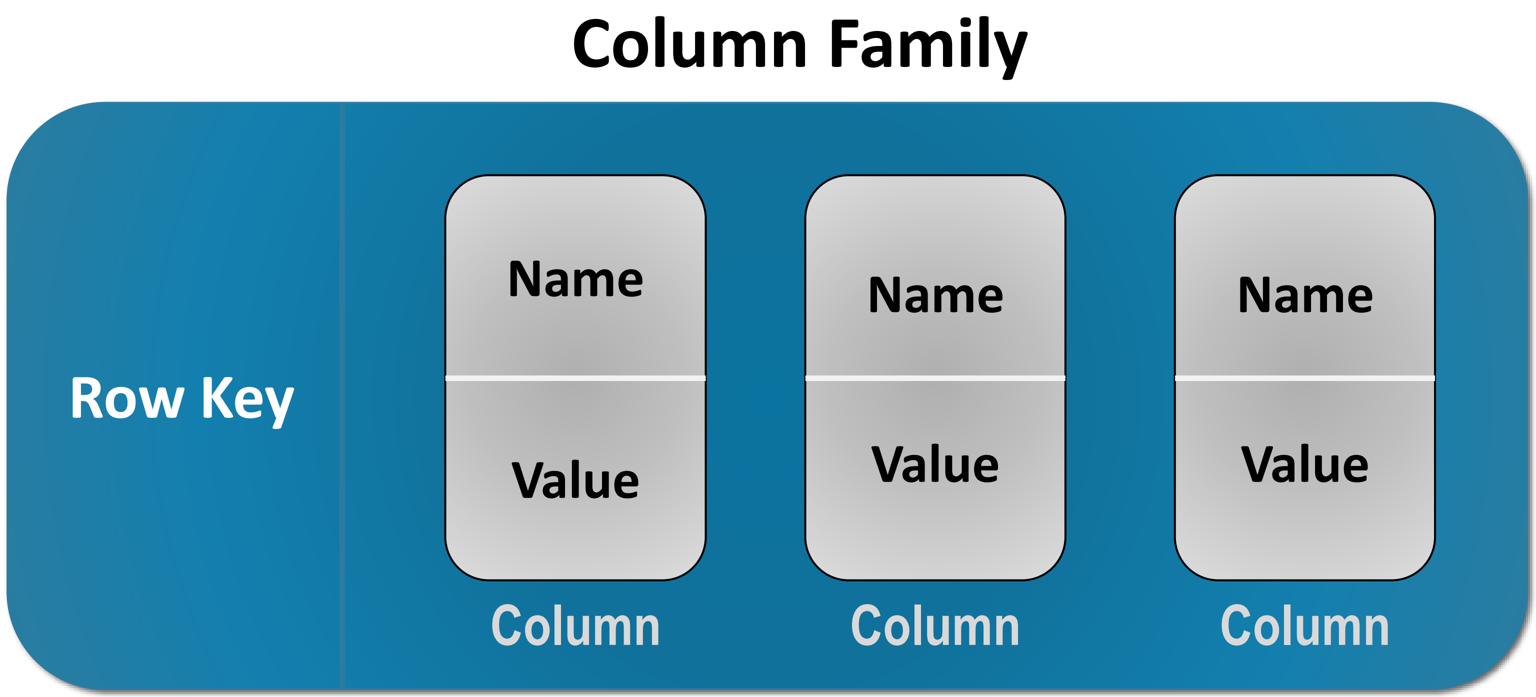
Object $o_3$ is stored on the node C

# Cassandra: Access to Data Replicas




**Replication Factor = 3**

# Cassandra Data Model (from BigTable)



# Cassandra data model

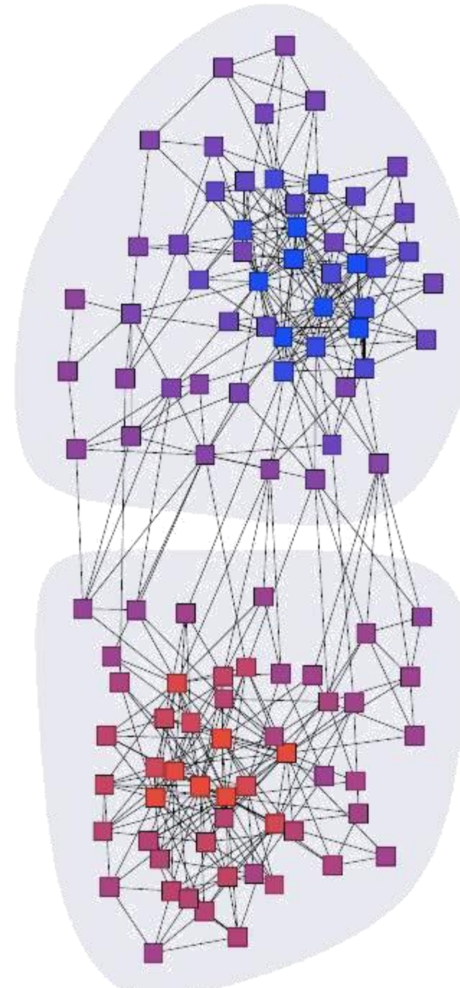
- Effectively a 3D hashtable
- A table contains a set of Column Family elements, each one has a key(rowName), and a value
- The value of a Column Family is in turn a collection of key/value elements, called column
- Different Column Family elements can (will) have different values
- No joins, limited query capabilities

# Contents

- **A world of graphs**
- Storing Graphs in the Cloud
- Processing Graphs in the Cloud

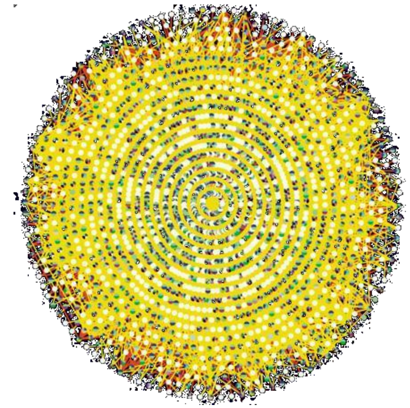
# Graph Definition

- A graph is a pair G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles

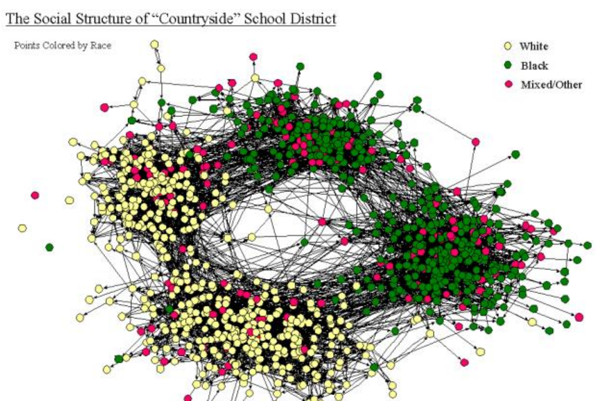


# Graphs are ubiquitous

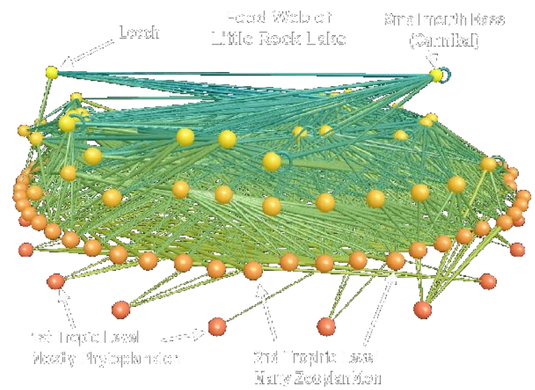
Internet Map
[lumeta.com]



Protein Interactions
[genomebiology.com]

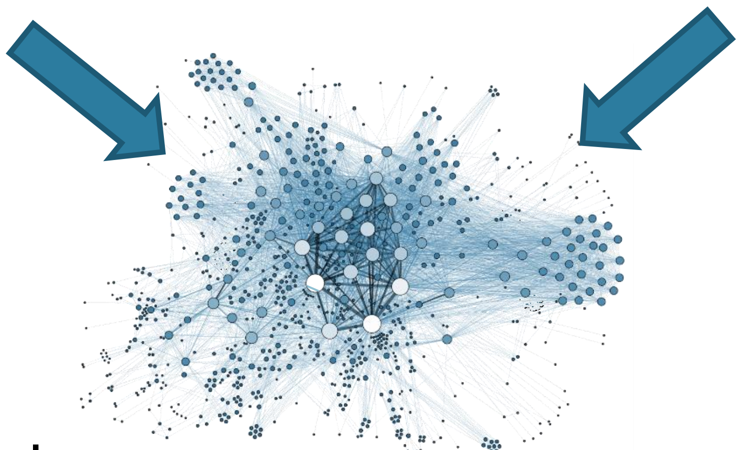



Friendship Network
[Moody '01]




Food Web [Martinez '91]

# Modeling & tracking interactions

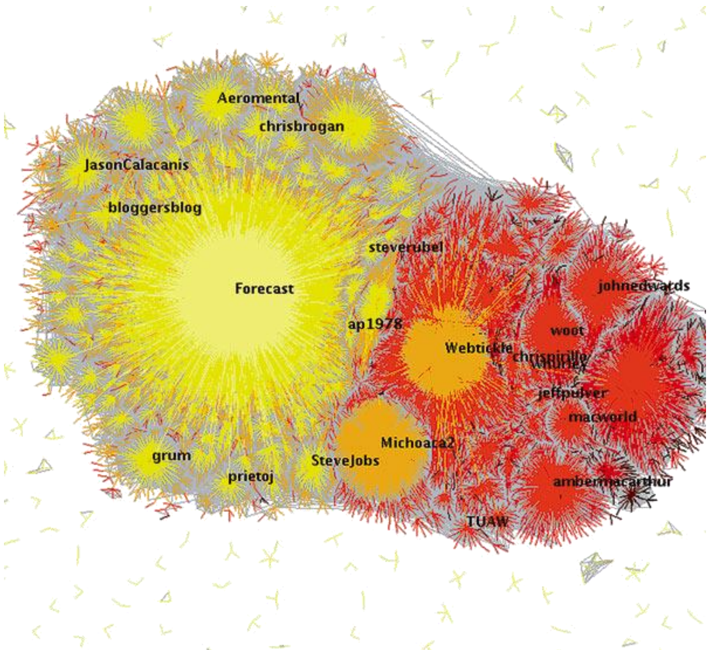


Graphs model **interactions**

Graph **size** grows with input data

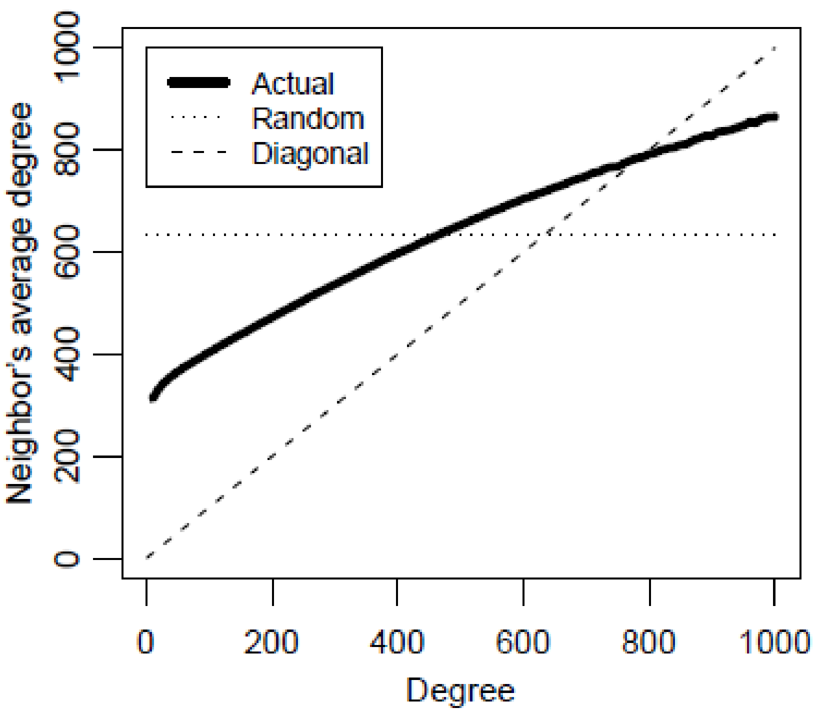
# Social graphs

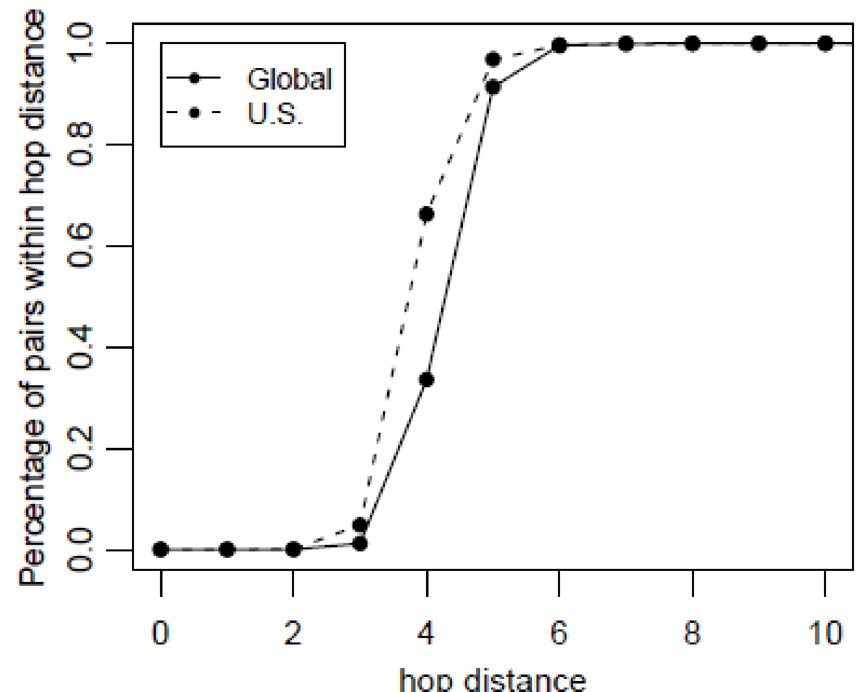
- Social media defines networks
  - Contacts
  - Messages
  - Tags
- Graph analysis allows to obtain valuable information
  - Identify leaders in a community
    - Measure influence
  - Identify “special” nodes and communities
    - Breaking up terrorist cells
    - Track spread of avian flu



# The Facebook Graph

- The power law in the degree popularity
- Replicated in many other human networks

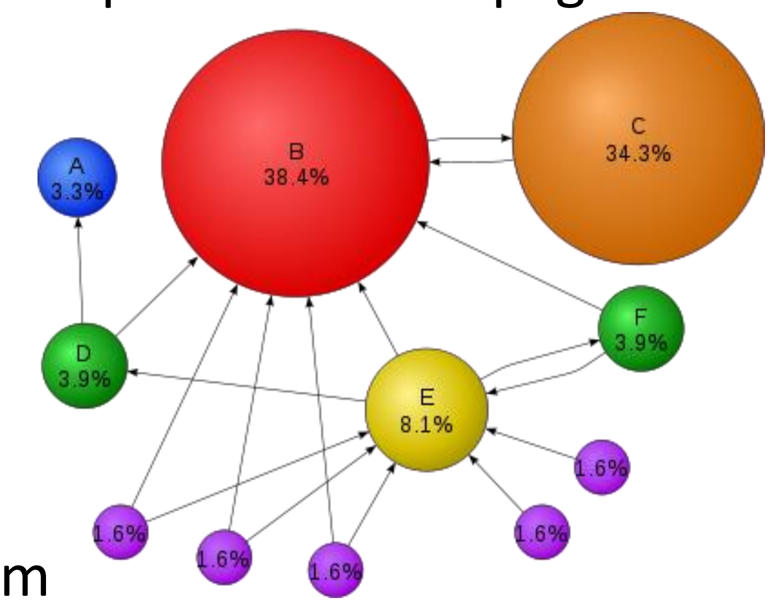



Ugander, J., Karrer, B., Backstrom, L., & Marlow, C. (2011). The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*.

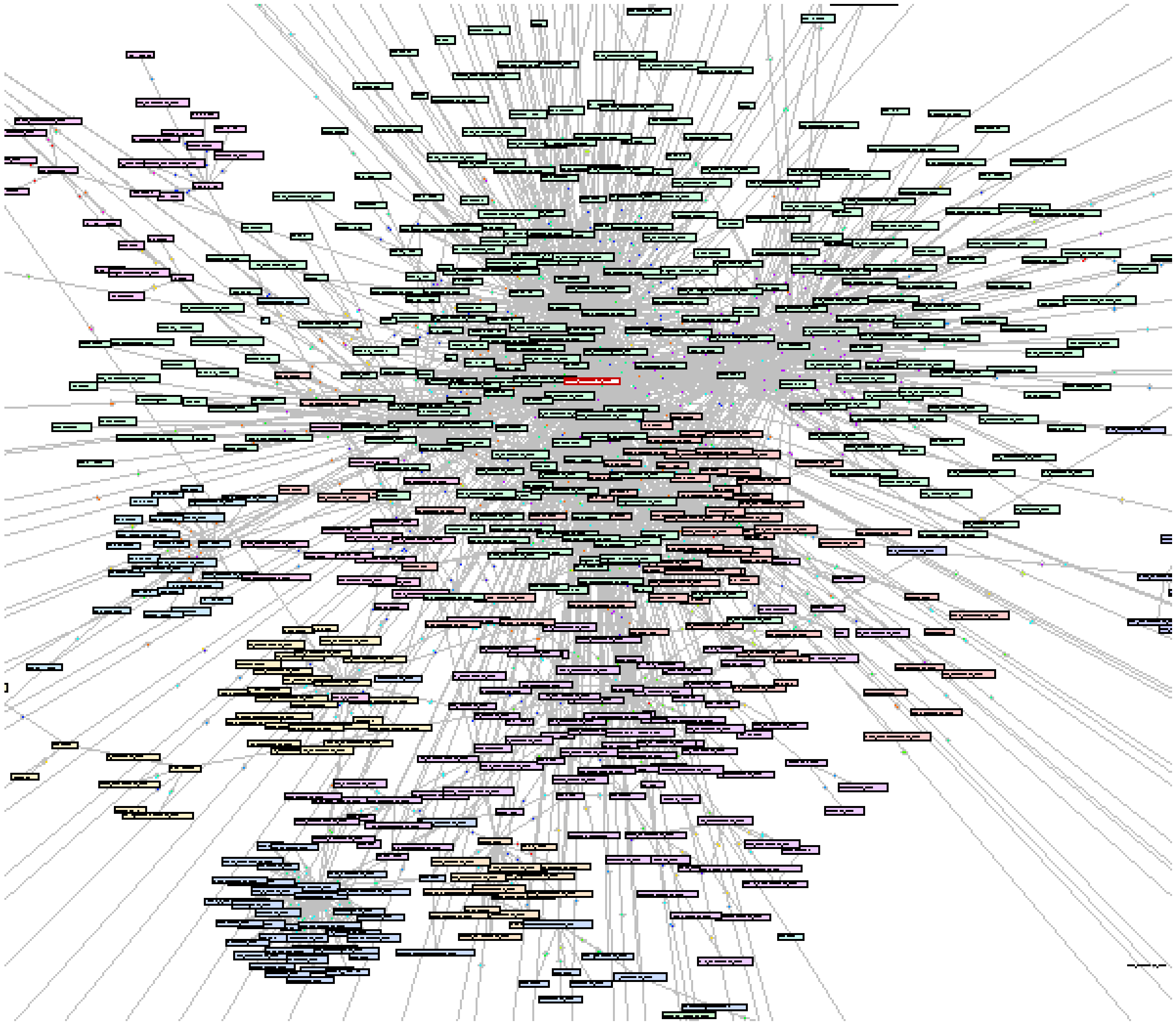
# Google's PageRank

- PageRank is a link analysis algorithm
- The rank value indicates an importance of a particular web page
- A hyperlink to a page counts as
  a vote of support
- A page that is linked to by many pages with high PageRank receives a high rank itself
- A PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank




Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank citation ranking: bringing order to the web. WWW

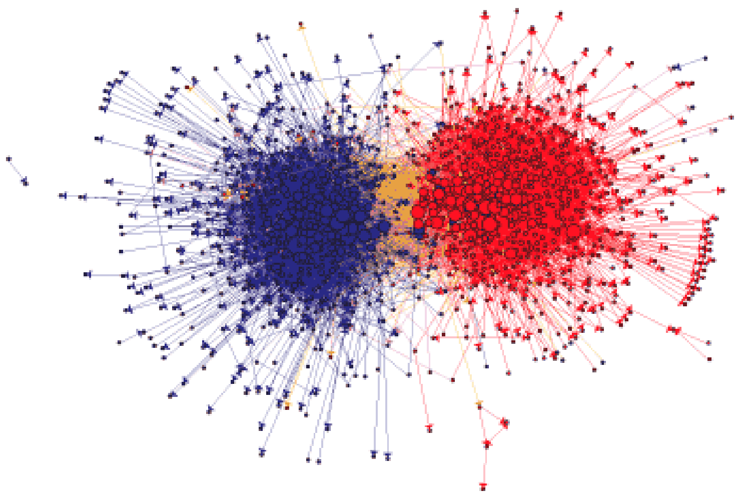
# Community detection



H., Jeffrey. "Exploring Enron: Visualizing ANLP results." *Applied Natural Language Processing*. 2004.

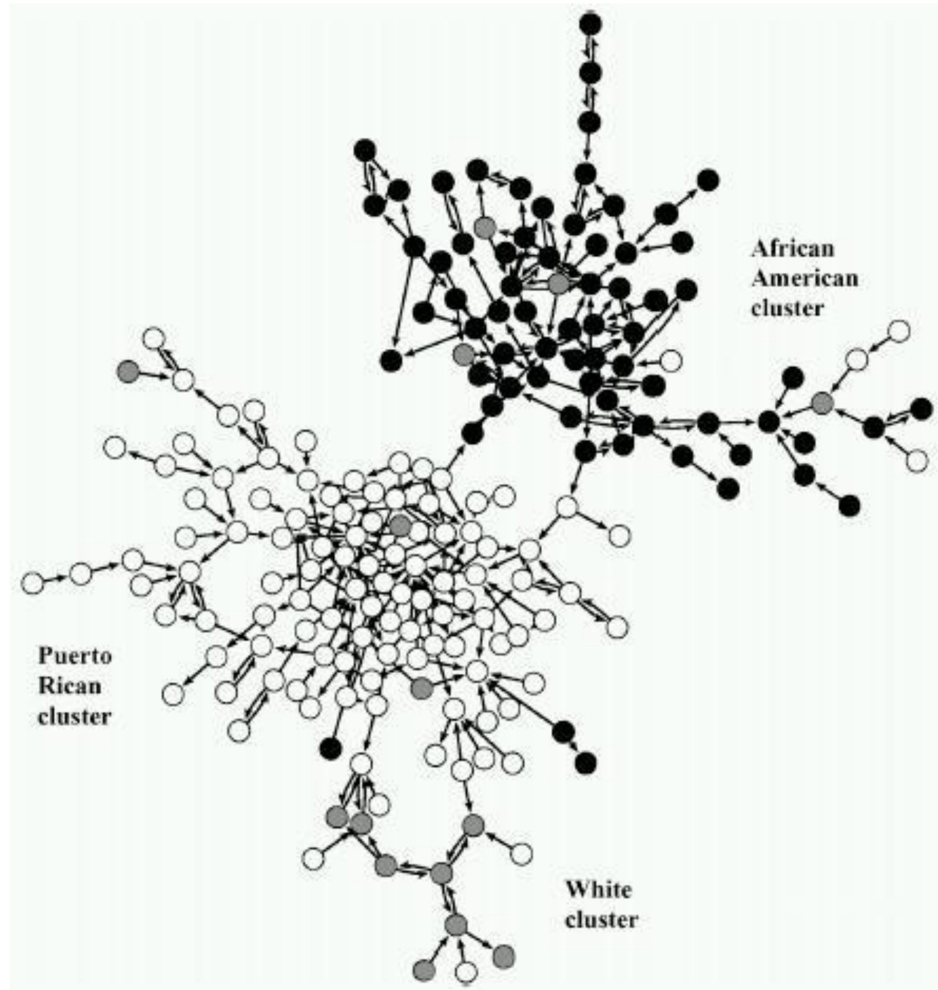
# Bipartite graphs

- Nodes only can have edges to the other part
- “Stable marriage” problem
  - Monster.com, Match.com
- Web advertising, click prediction



# Contagion / epidemic networks

- How quickly will a disease spread on this graph?




"Needle exchange" networks of drug users [Weeks et al. 2002]

# Contents

- A world of graphs
- **Storing Graphs in the Cloud**
- Processing Graphs in the Cloud

# Graph Storage

- Traditional DBs, NoSQL DBs can store graphs
- But query languages do not support native queries on graph elements (checking for relations)
- We need query languages/abstractions suitable for finding relationship patters
- Rise of graph DBs
  - Neo4j
  - Titan

# Graph Databases

- Database that uses graph structures with nodes, edges and properties to store data
- Provides index-free adjacency
  - Every node is a pointer to its adjacent element
  - Fast for following relationships
- Edges hold most of the important information and connect
  - nodes to other nodes
  - nodes to properties

# Graph queries

- List nodes/edges that have this property
- List matching subgraphs
- Can these two nodes reach each other?
- How many hops does it take for two nodes to connect?

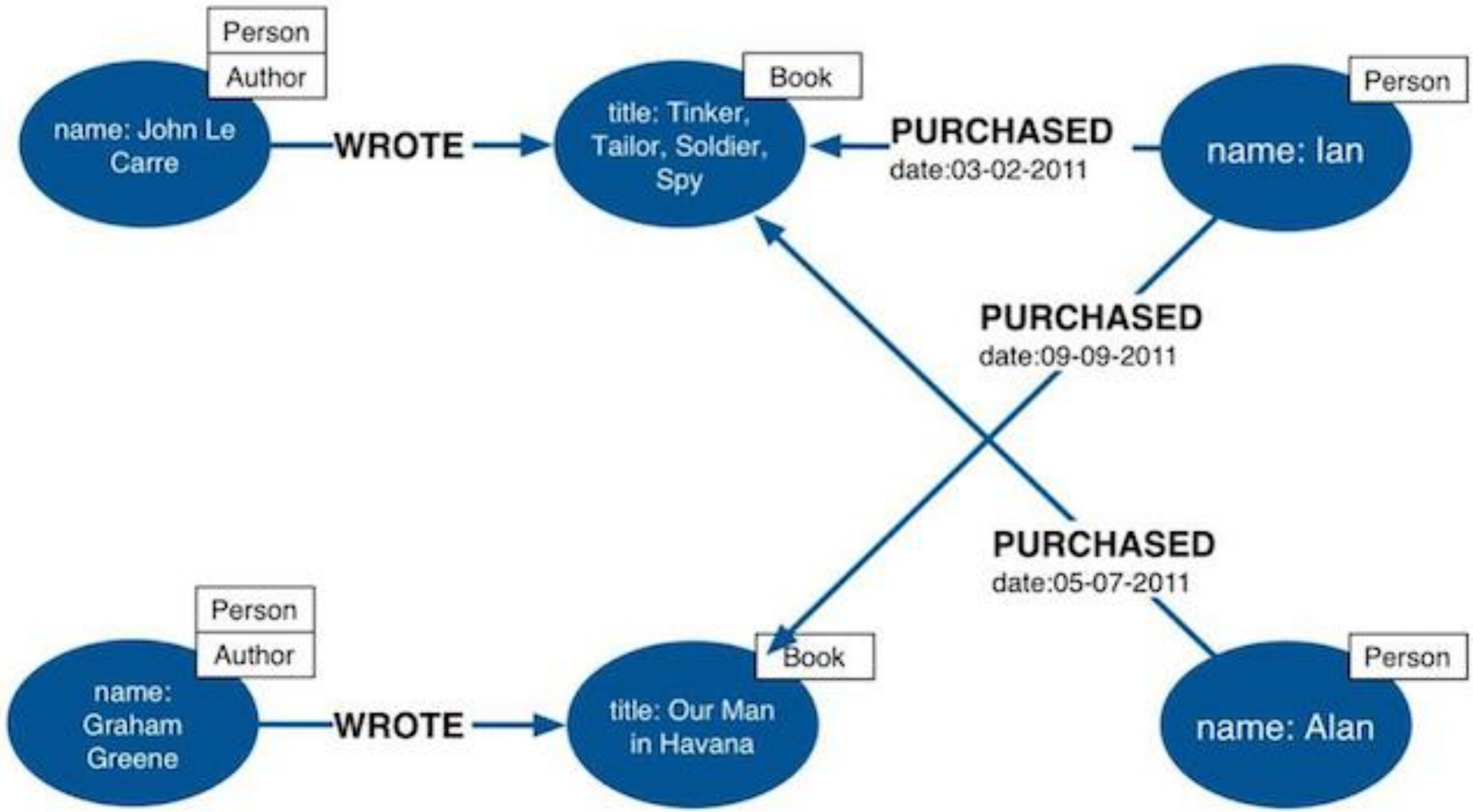
# Neo4j

- Java-based graph database
- ACID – atomic, consistent, isolated and durable for logical units of work
- Property model: powerful schema-less way to model graph-based information
- Good performance for not massive datasets
- Cypher - Graph-specific query language
  - ASCII-art syntax. Define and match patterns

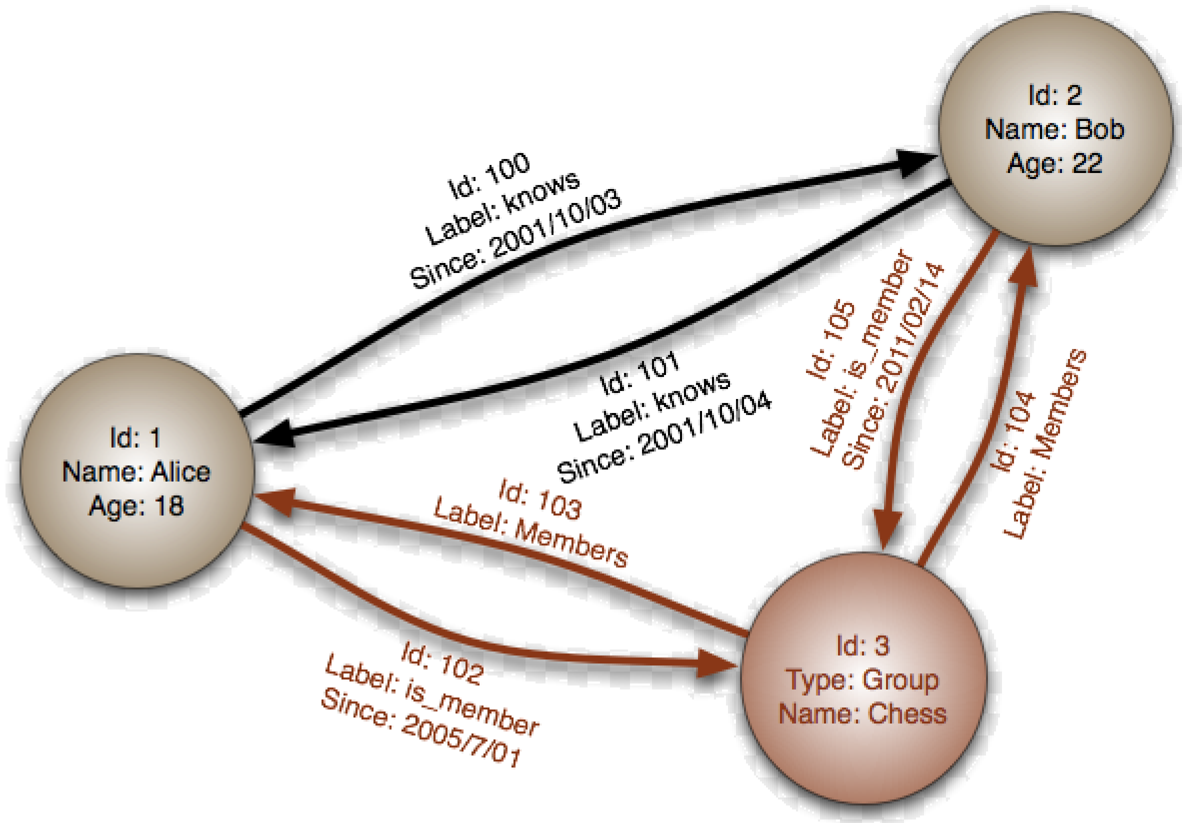
# Neo4j graph model

- A graph has nodes and edges
  - Multiple edges between nodes possible
- Nodes and edges can have properties (key-value pairs)
  - Node property: {name: “Ian”}
  - Edge property: {date: 10}
- Nodes and edges can have labels
  - Node label: Person
  - Edge label: Purchased

# The property graph model



# Property Graph



# SQL: Modeling and Querying a Graph

Task: listing of all friends-of-a-friend-of-a-friend of mine (who is obviously not a friend of mine)

| ACCOUNT_HOLDER | FRIEND_OF_HOLDER |
|---|---|
| Person001 | Person002 |
| Person001 | Person006 |
| Person002 | Person004 |
| Person004 | Person005 |
| Person006 | Person007 |
| Person007 | Person008 |

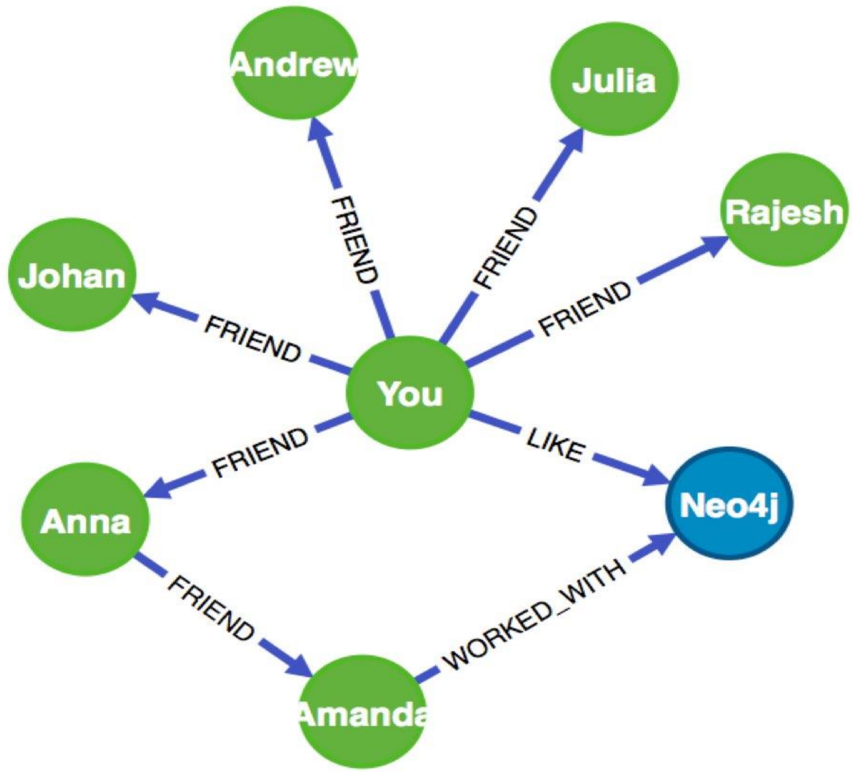
```
SQL: SELECT FRIEND_OF_HOLDER FROM TABLE_FRIENDS WHERE FRIEND_OF_HOLDER IN
(SELECT FRIEND_OF_HOLDER FROM TABLE_FRIENDS WHERE ACCOUNT_HOLDER IN
(SELECT FRIEND_OF_HOLDER FROM TABLE_FRIENDS WHERE ACCOUNT_HOLDER IN
(SELECT FRIEND_OF_HOLDER FROM TABLE_FRIENDS WHERE ACCOUNT_HOLDER = 'Person001')))
AND FRIEND_OF_HOLDER NOT IN (SELECT FRIEND_OF_HOLDER FROM TABLE_FRIENDS WHERE
ACCOUNT_HOLDER = 'Person001'); FRIEND_OF_HOLDER
```

# Cypher query language

- Query Language for Neo4j
  - Becoming standard through OpenCypher initative
- Match queries, returning all the graph elements who satisfy all the
- Expressive, does not require joining or other SQL

```
MATCH [a:Person {ID: “Person001”}]-[:IS_A_FRIEND_OF*3]->[b:Person]
WHERE NOT (a)-[:IS_A_FRIEND_OF]-(b)
RETURN DISTINCT b.id;
```

# Sample Cypher query on a graph

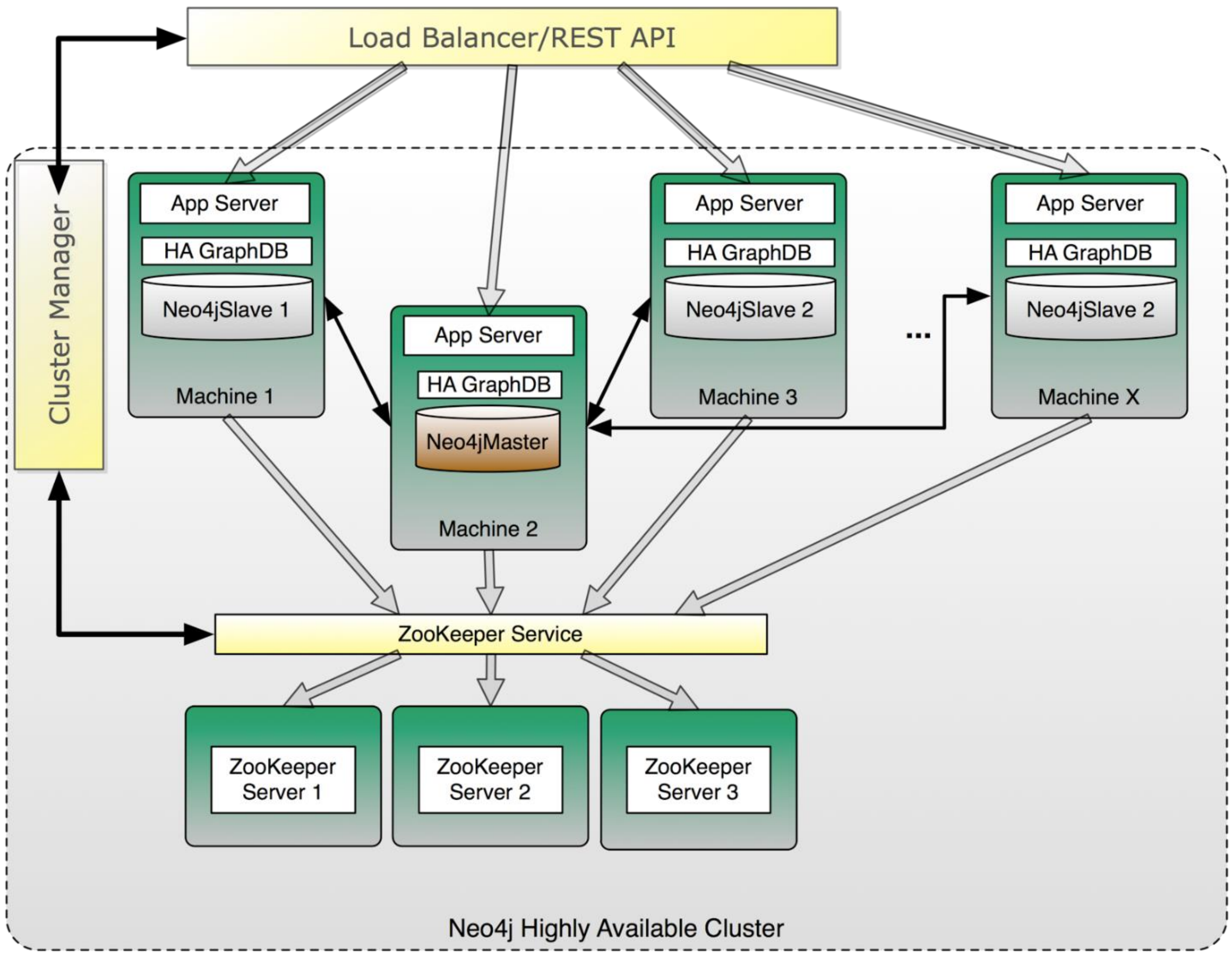



```
MATCH (you {name:"You"})
MATCH (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"})
MATCH path = shortestPath( (you)-[:FRIEND*..5]-(expert) )
RETURN db,expert,path
```

# Neo4J features

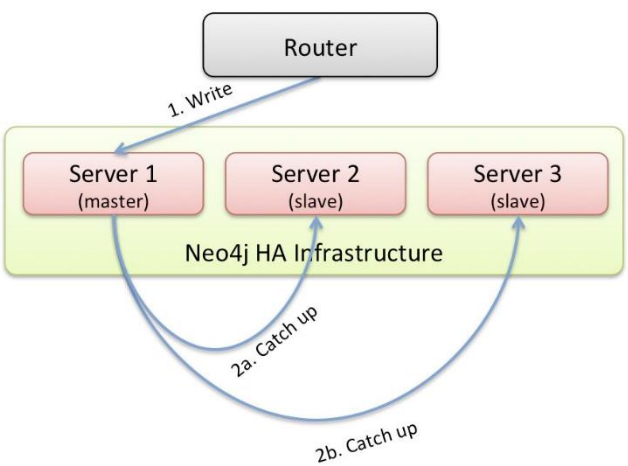
- **Consistency**
  - Since graph databases are operating on connected nodes, most graph database solutions usually do not support distributing the nodes on different servers. Within a single server, data is always consistent.
- **Transactions**
  - Neo4J is ACID-compliant. Before changing any nodes or adding any relationships to existing nodes, we have to **start a transaction**.
- **Availability**
  - Neo4J, as of version 1.8, achieves high availability by providing for **replicated slaves**. These slaves can also handle writes: When they are written to, they synchronize the write to the current master, and the write is committed first at the master and then at the slave. Other slaves will eventually get the update.

# Neo4J HA Architecture

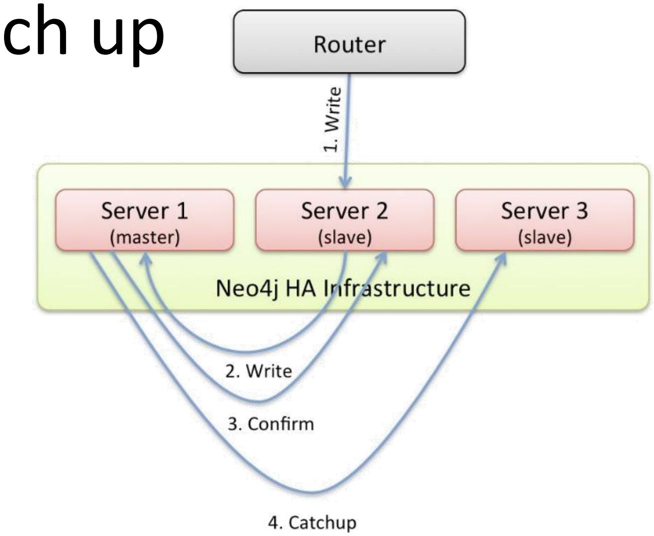


# HA Writing to a Master/ to a slave

- Writes ***to the master*** are fast
  - slaves eventually catch up




- Writes ***to a slave*** cause a synchronous transaction with the master
  - the other slaves eventually catch up

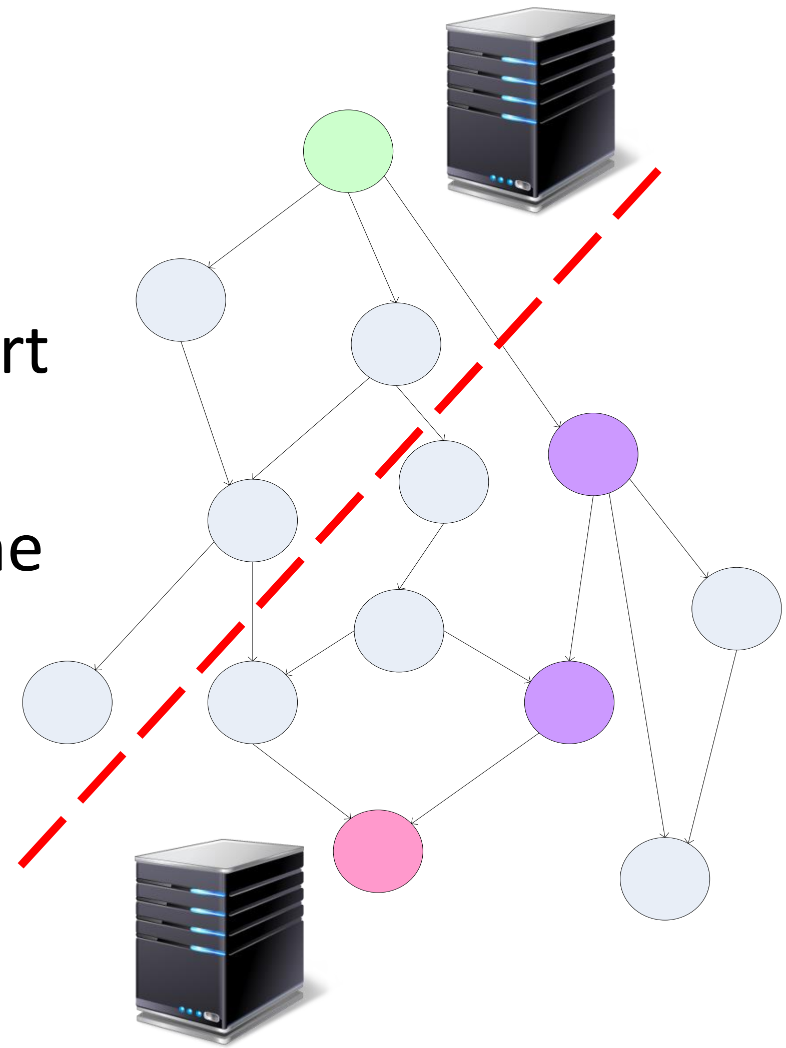


# Contents

- A world of graphs
- Storing Graphs in the Cloud
- **Processing Graphs in the Cloud**

# Sample graph problem: Finding the Shortest Path

- SSSP: The Single-Source shortest path problem.
- Common graph search application: finding the shortest path from a start node to one or more target nodes
- Commonly done on a single machine with *Dijkstra's Algorithm*



# Dijkstra's algorithm

| | |
|---|---|
| dist[s] ← 0 | (distance to source vertex is zero) |
| for all $v \in V-\{s\}$ | |
| do dist[v] ← ∞ | (set all other distances to infinity) |
| S ← ∅ | (S, the set of visited vertices is initially empty) |
| Q ← V | (Q, the queue initially contains all vertices) |
| while Q ≠ ∅ | (while the queue is not empty) |
| do u ← mindistance(Q,dist) | (select the element of Q with the min. distance) |
| S ← S ∪ {u} | (add u to list of visited vertices) |
| for all $v \in$ neighbors[u] | |
| do if dist[v] > dist[u] + w(u, v) | (if new shortest path found) |
| then d[v] ← d[u] + w(u, v) | (set new value of shortest path) |
| | (if desired, add traceback code) |
| return dist | |

# Graph representation

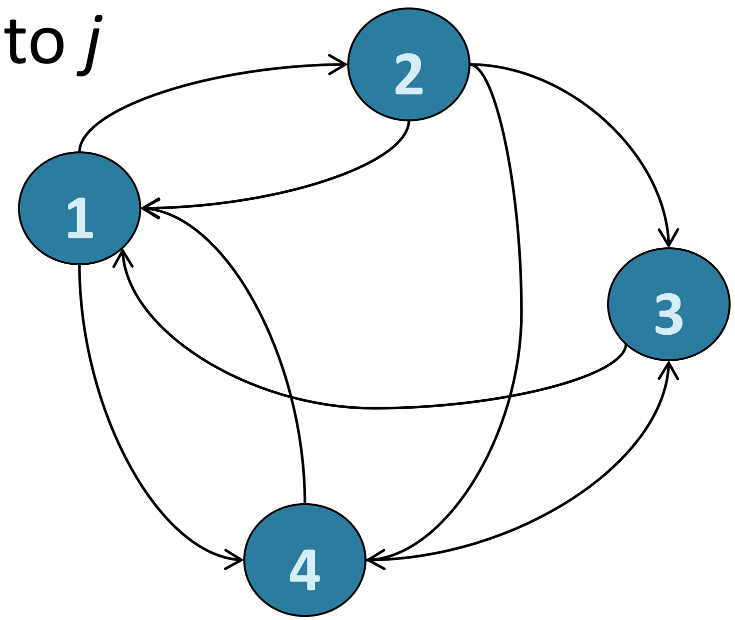
- Algorithms need to keep the information about how graph is represented
  - Structure, and attributes of edges and vertices
  - Both for serialization, and for loading in memory
  - Example: edge list
    - 1 4
    - 2 3
    - 5 2
    - 6 1
    - 1 4

# Graph Representation: Adjacency Matrices

Represent a graph as an *n* x *n* square matrix *M*

- $n = |V|$
- $M_{ij} = 1$ means a link from node *i* to *j*

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | 0 | 1 | 0 | 1 |
| **2** | 1 | 0 | 1 | 1 |
| **3** | 1 | 0 | 0 | 0 |
| **4** | 1 | 0 | 1 | 0 |



# Adjacency Matrices: Critique

- Advantages:
  - Naturally encapsulates iteration over nodes
  - Rows and columns correspond to inlinks and outlinks
- Disadvantages:
  - Sparse graphs. A graph $G = (V,E)$ is sparse if $|E|$ is much smaller than $|V|^2$.
  - Lots of zeros for sparse matrices
  - Lots of wasted space

# Graph Traversal in MapReduce

- Approach: Parallel processing of each vertex
  - Each Map/Reduce function has access to limited info
    - One node and its links
- **Iterate** executions of a MapReduce job
  - Map: compute something on each node. Potentially send information to that/other nodes that is aggregated in Reducers.
  - Reducers compute something on each node
  - The output of the reducers in iteration #n becomes the input of the mappers in iteration #n+1

# Equal Weight SSSP

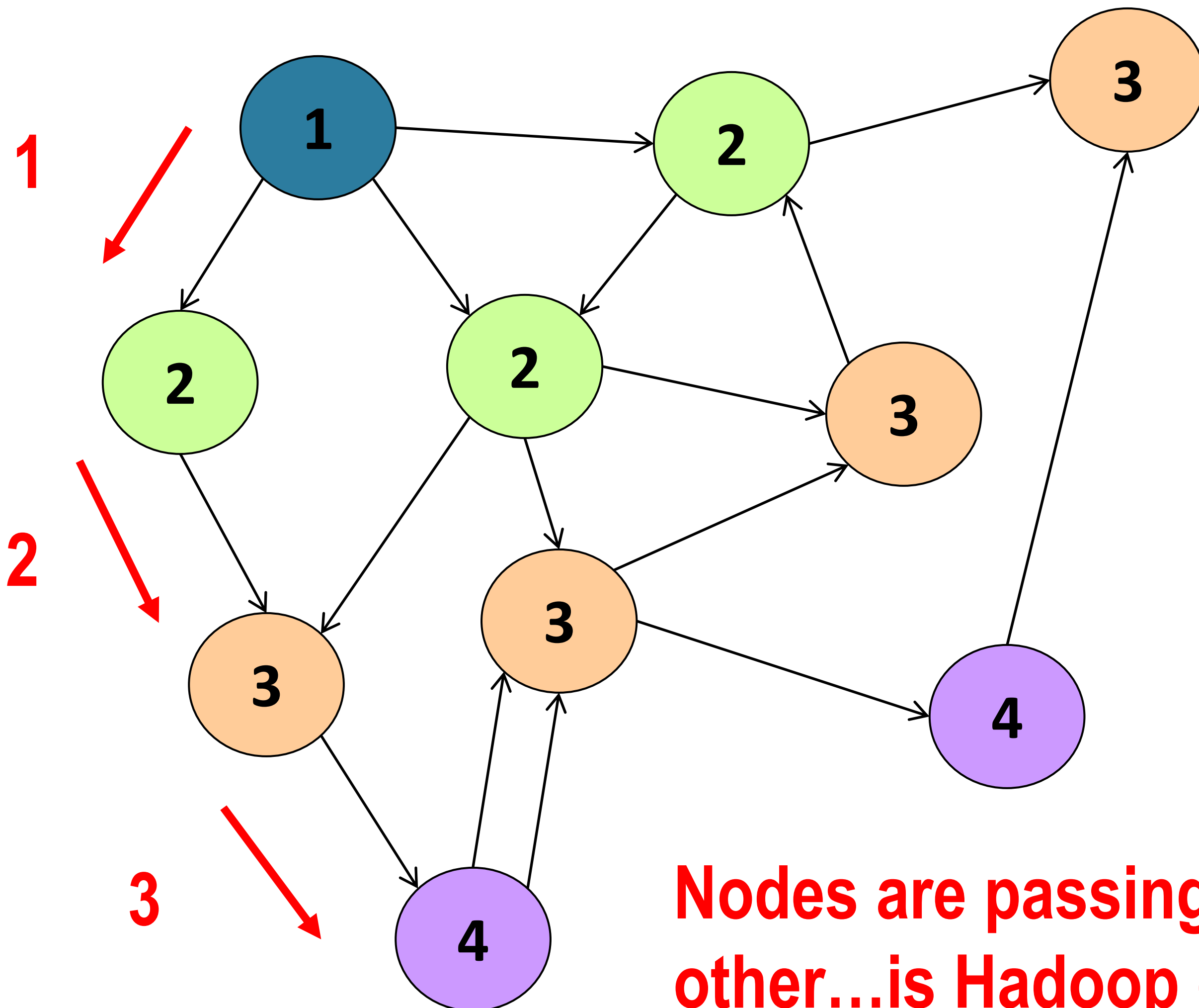
- **Problem:** from an origin node, find the **shortest distance** to every other node of the graph, with all links having the same distance
  - Also known as the “Kevin Bacon number problem”
- Small world phenomenon
  - The world is tied together by a network of personal relationships.
  - There's a popular hypothesis called *Six Degrees of Separation.*

# Finding the Shortest Path: Intuition

- Breadth-First Search (BFS) algorithm
- We can define the solution to this problem inductively:
  - `distanceTo(startNode) = 0`
  - For all nodes *n* directly reachable from startNode,
    `distanceTo(n) = 1`
  - For all nodes *n* reachable from some other set of nodes S,

    `distanceTo(n) = 1 + min(distanceTo(m), m ∈ S)`

# Visualizing Parallel BFS




**Nodes are passing data to each other…is Hadoop good at this?**

# MapReduce graph processing performance

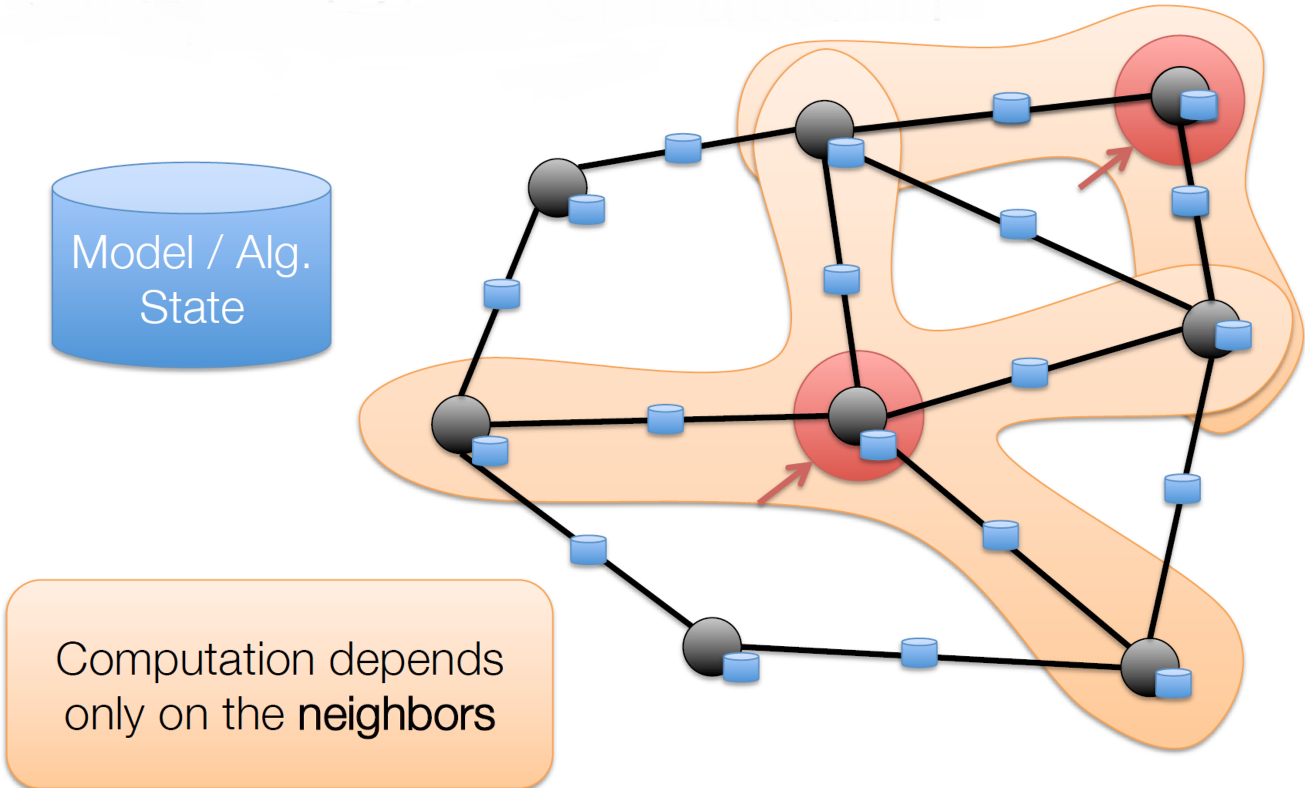
- Iterative algorithms involve HDFS writing in each step
  - Resending the graph structure in each iteration is **VERY inefficient**
- One Map task per node, and sending of messages to other nodes depending on connections results in significant communications cost.
- In-memory systems are a much better fit for this type of computation
  - Graph-specific in-memory sytsems have developed recently

# Google's Pregel model: Think like a vertex [4]

- Computation is iterative
  - Every iteration, a function that is executed at each vertex
    - Vertices can send messages to its neighbours
    - Messages arrive in the next iteration
- Computation is executed in parallel
  - Each vertex is independent from the rest in the same step
  - Messages are the synchronization mechanism

[4] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010, June). Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD*

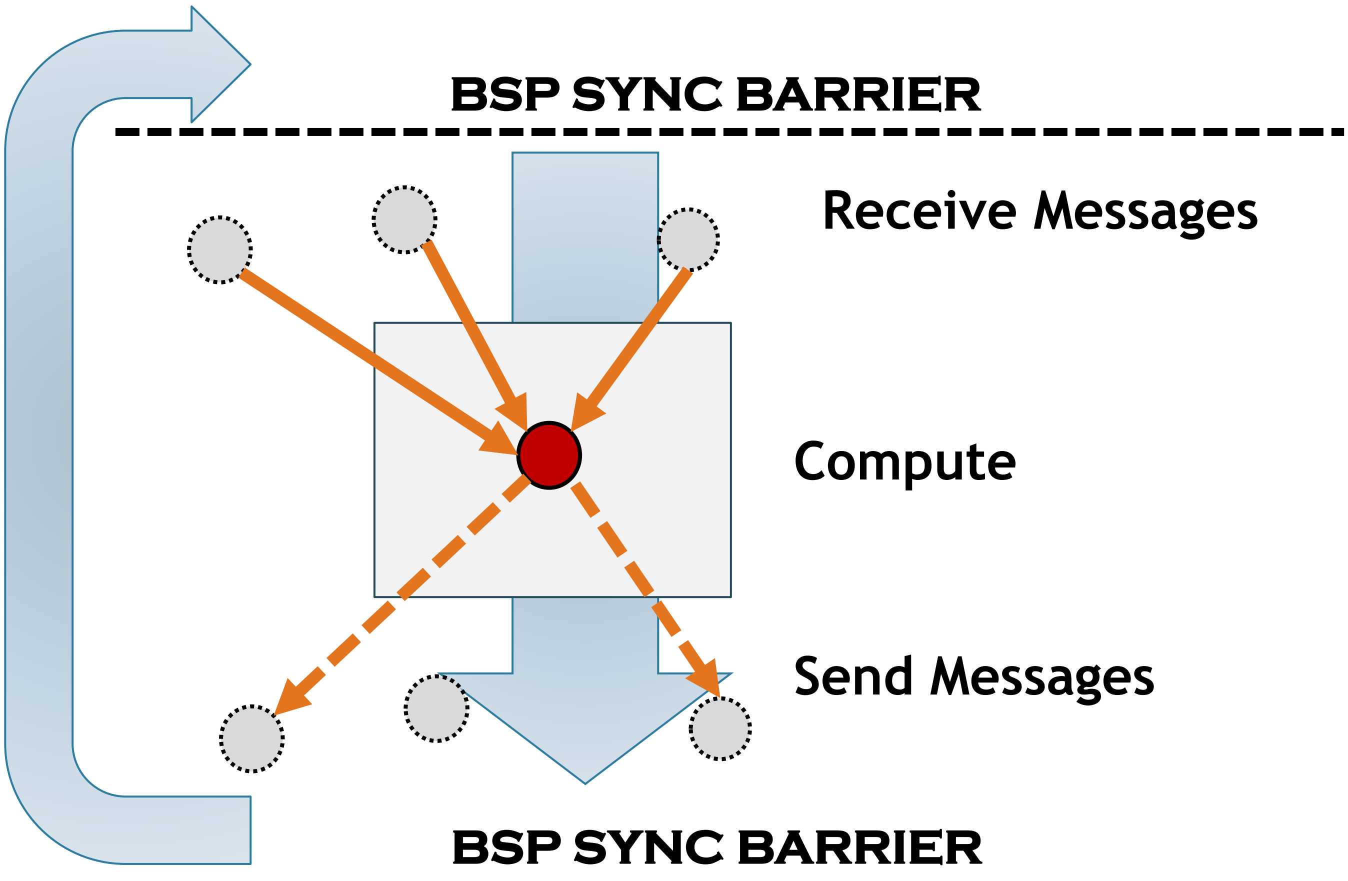
# Pregel: The world of a vertex



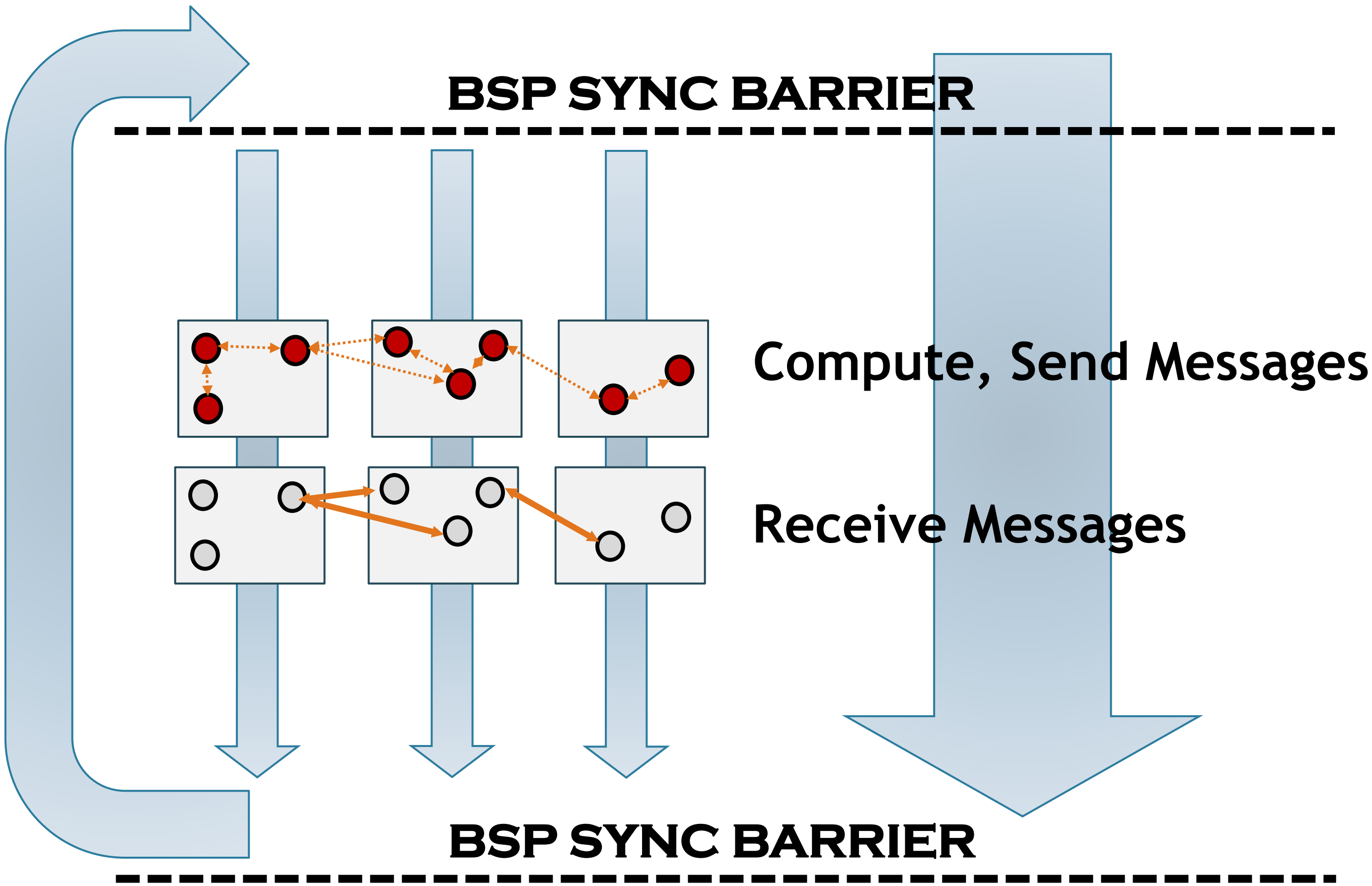
# Think like a vertex

- Parallelises computation by processing each vertex independently
  - Only using **local** information
  - Make sure neighbouring vertexes on the the **same machine**!
- Messages can be sent to other neighbours
- Great for things like:
  - Shortest path computation
  - Page Rank
  - Breadth first search

# Think like a vertex



# Pregel's node-centric processing model



# This requires **graph partitioning**

- Necessary when graph is too large to fit (or be processed) on one machine
- Split the graph across multiple machines
  - Like Hadoop splits data across multiple machines
- A **graph partition** defines which edges and verteces are allocated to which machine
- Graph partitions have impact on performance
  - There is a **large overhead** to sending messages to neighbouring vertices on **different** machines (over the network)

# Pregel-style graph processing systems

- Pregel
  - Original Google paper
- Apache Giraph
  - Java-based
- Apache Spark GraphX
  - Extension of Spark
- …Ongoing research efforts in this space

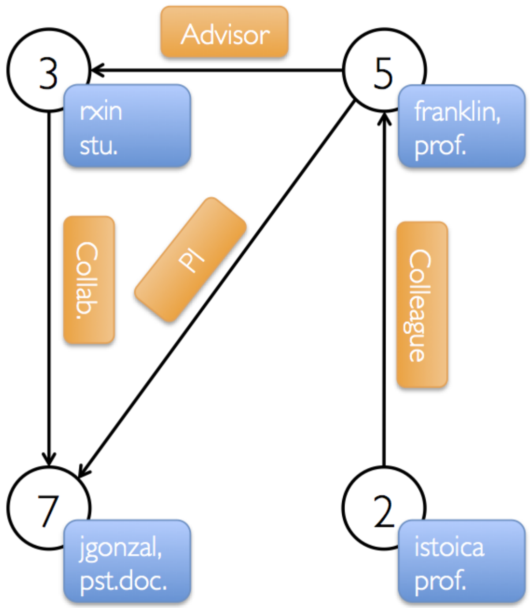
# Spark GraphX

- Spark library for graph processing
- Provides specialized RDDs for representing graph structure, as well as its information (property graphs)
- Provides methods for creating graph, transforming them, implementing multiple common graph metrics and algorithms

# GraphX Property Graphs

**class Graph**[VD, ED]
{ **val** vertices: VertexRDD[VD]
**val** edges: EdgeRDD[ED] }

Property Graph




Vertex Table

| Id | Property (V) |
|---|---|
| 3 | (rxin, student) |
| 7 | (jgonzal, postdoc) |
| 5 | (franklin, professor) |
| 2 | (istoica, professor) |

Edge Table

| SrcId | DstId | Property (E) |
|---|---|---|
| 3 | 7 | Collaborator |
| 5 | 3 | Advisor |
| 2 | 5 | Colleague |
| 5 | 7 | PI |

**val** userGraph: Graph[(String, String), String]

# GraphX RDD

- Holds graph data and provides methods
- VertexRDD[VertexId, VertexData]
  - Vertice IDs have to be longs
    - String to hash conversion
  - Holds vertex properties
- EdgeRDD [EdgeData]
  - Holds source, destination, edge properties
  - Technically a directed multigraph
- Triplets
  - Join of source vertex, destination vertex and edge

# Graph creation in GraphX

- Graphs can be created from a file - format must be a list of edges in text format. Values will be long numbers
  - GraphLoader.edgeListFile(“hdfs://........”)
- From RDDs with the right information about nodes and edges.
  - Graph.fromEdges(<<EdgeRDD>>)
  - Graph.fromEdgeTuples(<<RDD[(VertexId, VertexId)] >>)
- Possible to specify how many partitions, how the graph is partitioned

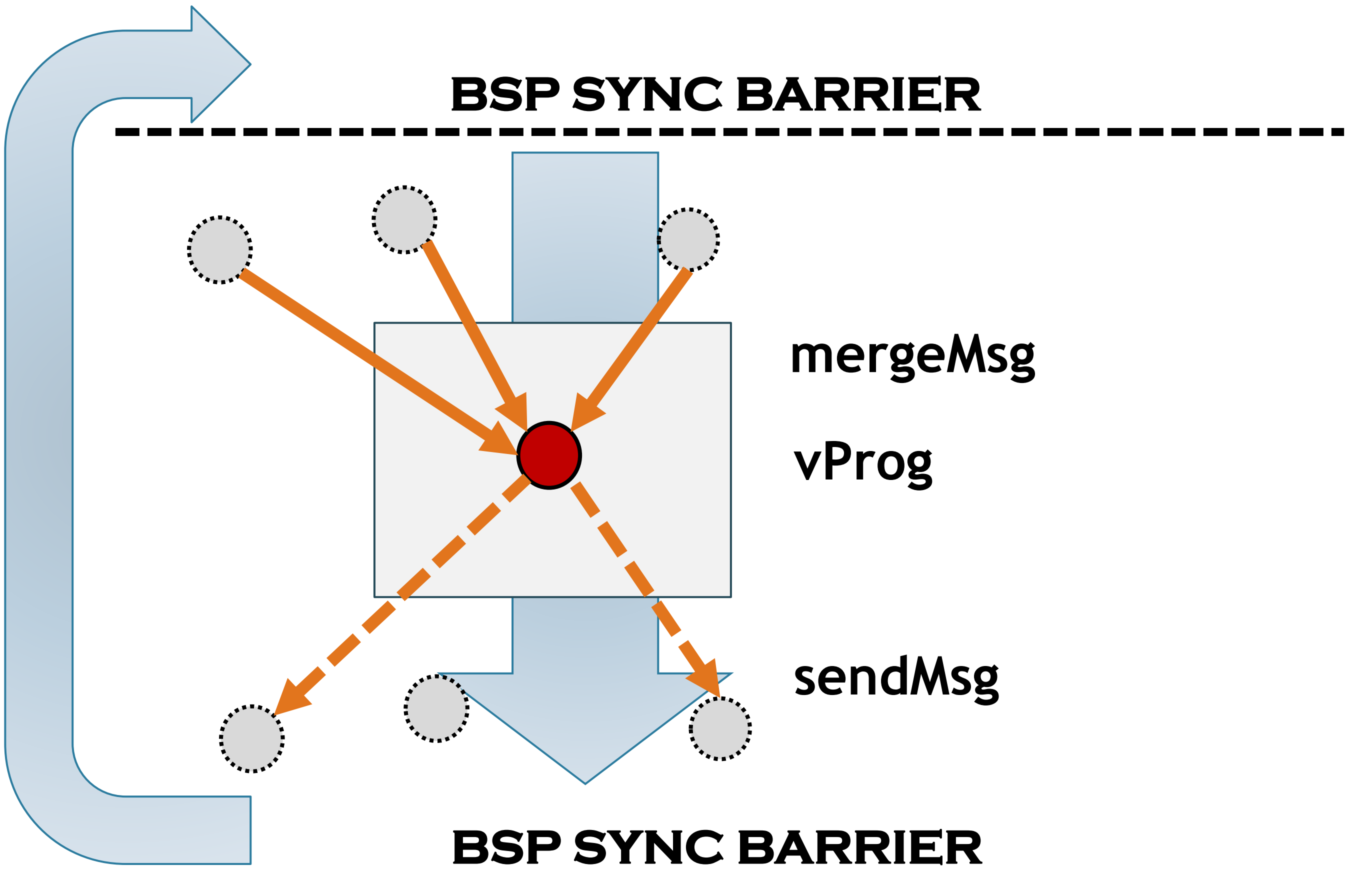
# GraphX predefined methods

- A graph RDD has multiple convenient methods that provide access to its information and implement relevant operations
- Graph.vertices, graph.edges, graph.triples
  - Access to RDDs with the property information
- Graph.degrees
  - Provides a tuple with (vertexId, degree of each vertex)
- Graph.connectedComponents
  - Obtains each of the connected components of the graph

# Pregel programming in GraphX

- GraphX provides a Pregel transformation for iterative graph traversals
- Need to specify number of iterations
- Three functions for implementing Pregel's model
  - vprog: (VertexID, VD, A) => VD
    - Updates vertex properties (VD)
  - sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)]
    - Sends messages to vertex neighbours
  - mergeMsg: (A, A) => A)
    - Combines all incoming messages into a single one
- Returns new graph with updated vertex values

# GraphX Pregel model



# GraphX Pregel model (2)