

Graph Data Management

Amol Deshpande

Associate Professor

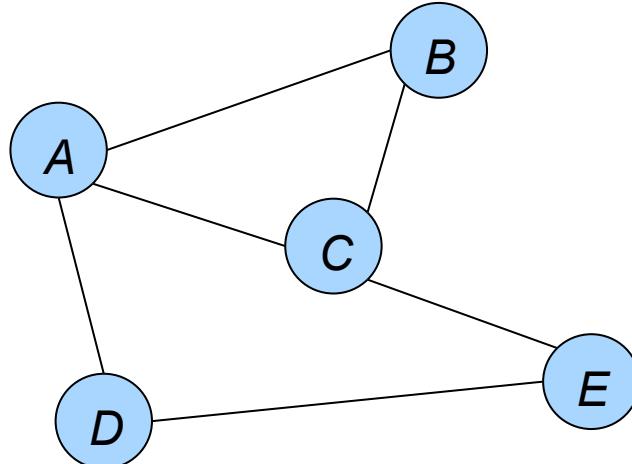
*Department of Computer Science and UMIACS
University of Maryland at College Park*

Outline

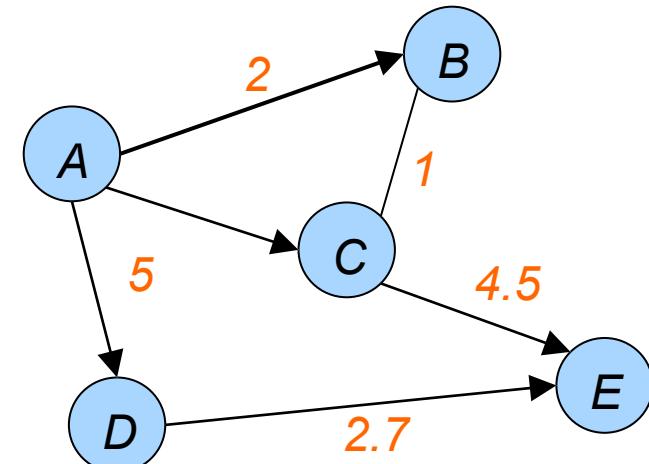
- **Background and Motivation**
- Graph Queries and Analysis Tasks
- Graph Data Management: Storage
- Graph Data Management: Processing
- What we are doing

Background: Graphs

- A *graph* captures a set of entities/objects, and interconnections between pairs of them
 - *Graphs* also often called *networks*
 - Entities/objects represented by *vertices or nodes*
 - Interconnections between pairs of vertices called *edges*
 - Also called *links, arcs, relationships*



An undirected, unweighted graph



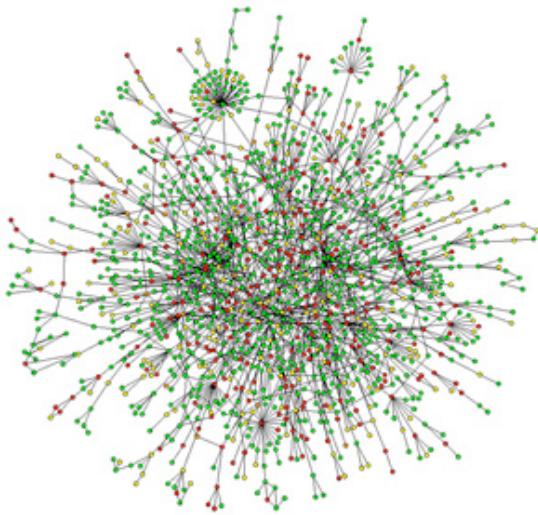
A directed, edge-weighted graph

Background: Graphs

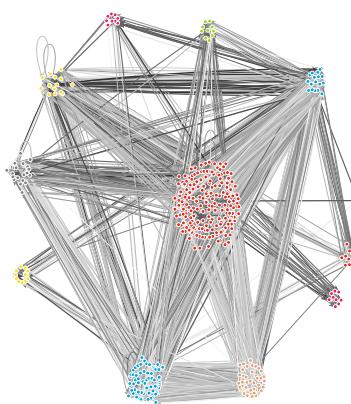
- A *graph* captures a set of entities/objects, and interconnections between pairs of them
 - *Graphs* also often called *networks*
 - Entities/objects represented by *vertices or nodes*
 - Interconnections between pairs of vertices called *edges*
 - Also called *links, arcs, relationships*
- Graph theory, graph algorithms very well studied in Computer Science
 - Not as much work on managing graph-structured data

Motivation

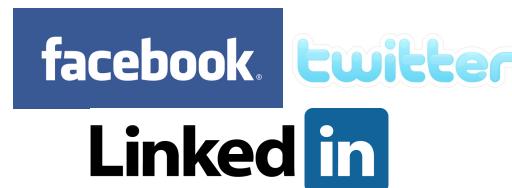
- Increasing interest in querying and reasoning about the *underlying graph structure* in a variety of disciplines



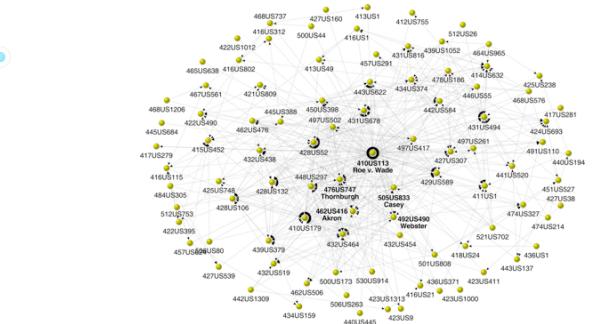
A protein-protein interaction network



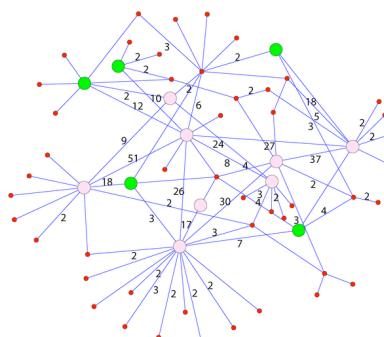
Financial transaction networks



Social networks



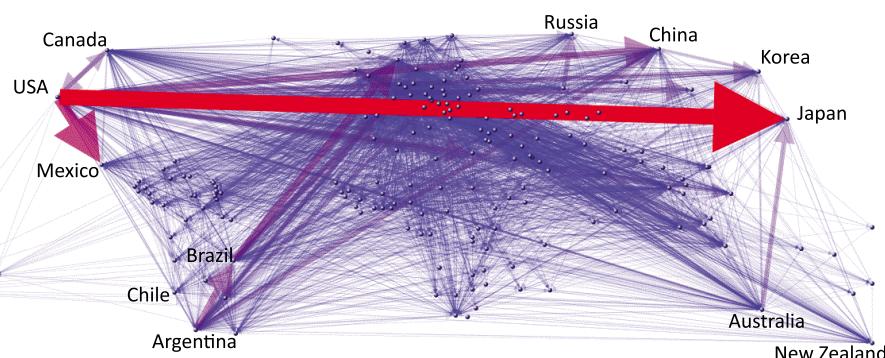
Supreme court citation network



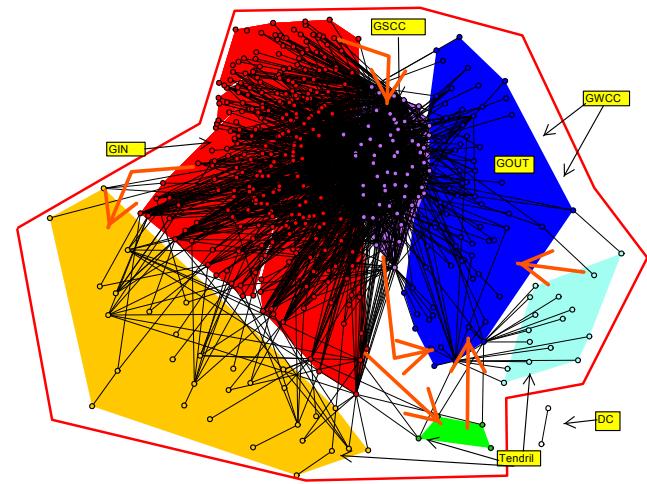
Stock Trading Networks

Motivation

- Increasing interest in querying and reasoning about *interconnected entities* in a variety of disciplines



Global virtual water trade network



Federal funds networks

Citation networks

Parcel shipment networks

Collaboration networks

Knowledge Graph

Telecommunications networks

World Wide Web

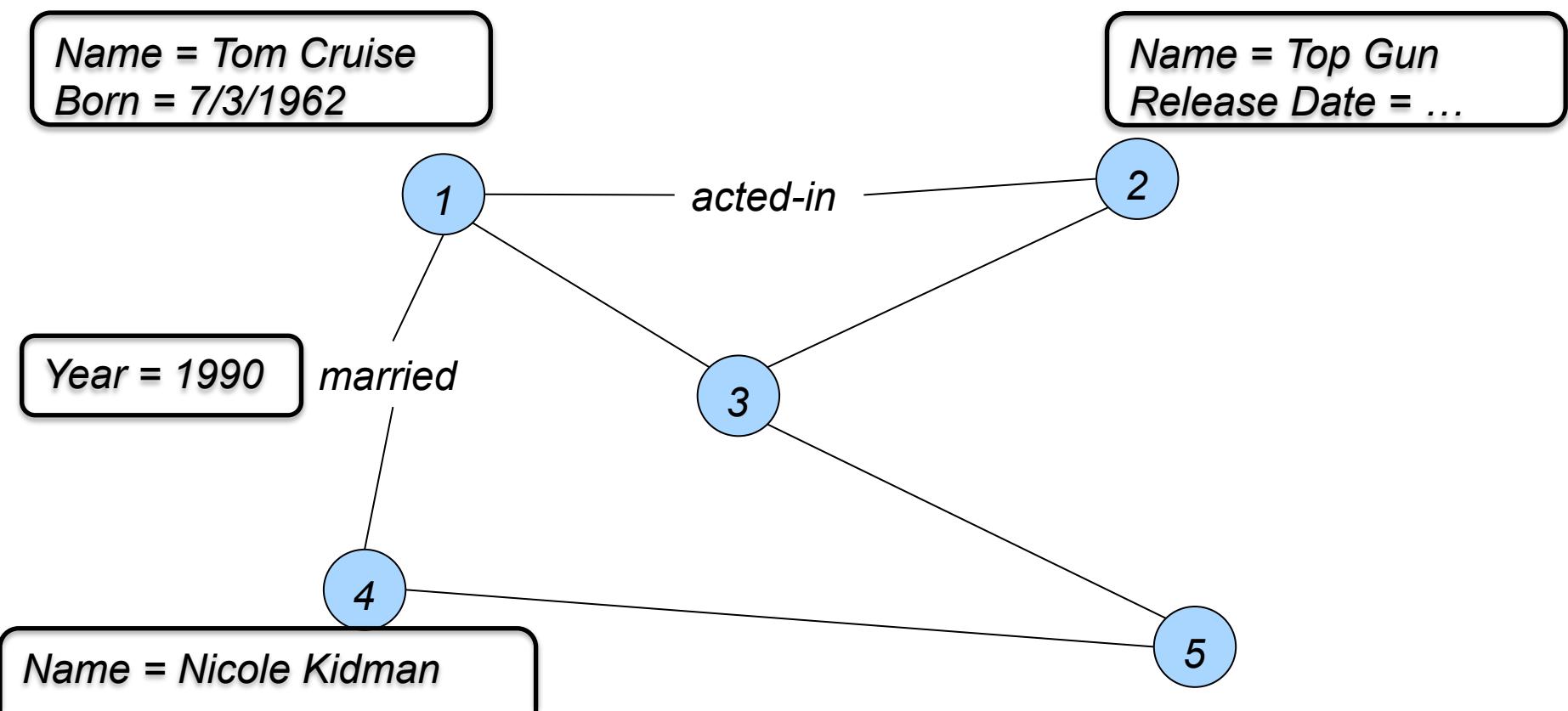
Disease transmission networks

Motivation

- Increasing interest in querying and reasoning about *interconnected entities* in a variety of disciplines
- Underlying data hasn't necessarily changed that much
 - Aside from the data volumes and easier availability
- However, several new realizations in recent years:
 - Reasoning about the graph structure can provide useful and actionable insights (*network science/complex network analysis*)
 - Lose too much information and intuitions if graph structure ignored
 - Not easy to write many natural queries or tasks using traditional tools
 - Especially relational databases like Oracle
 - Harder to efficiently process inherently graph-structured queries or complex network analysis tasks using existing tools
 - A major concern with increasingly large graphs seen in practice

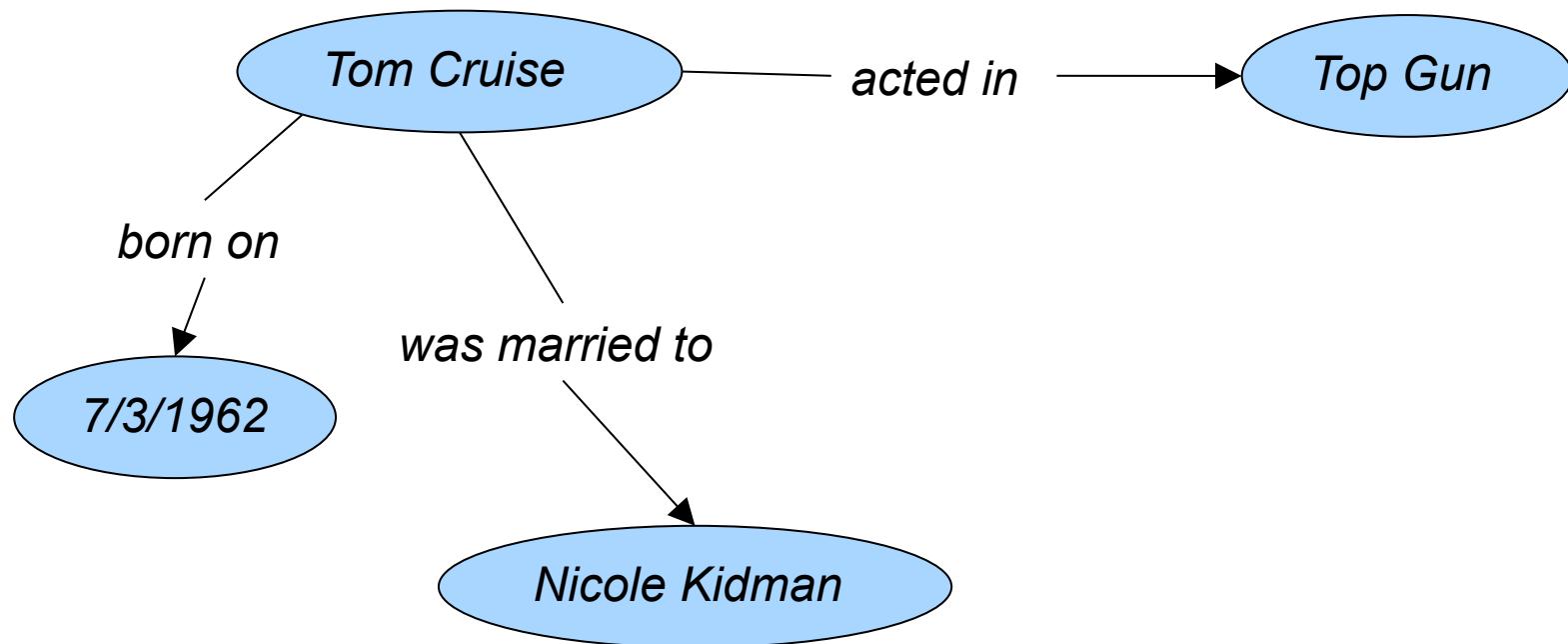
Graph Data Models: Property Graph

- Property Graphs
 - A graph model used by many open-source graph data management tools
 - In essence: a directed graph where each node and each edge may be associated with a set of *properties*



Graph Data Models: RDF

- Resource Description Framework (RDF)
 - A commonly used data model for representing *knowledge bases*
 - In essence: a directed, labeled graph
 - Each edge (called a triple): connects a “subject”, an “object”, and is associated with a “predicate”



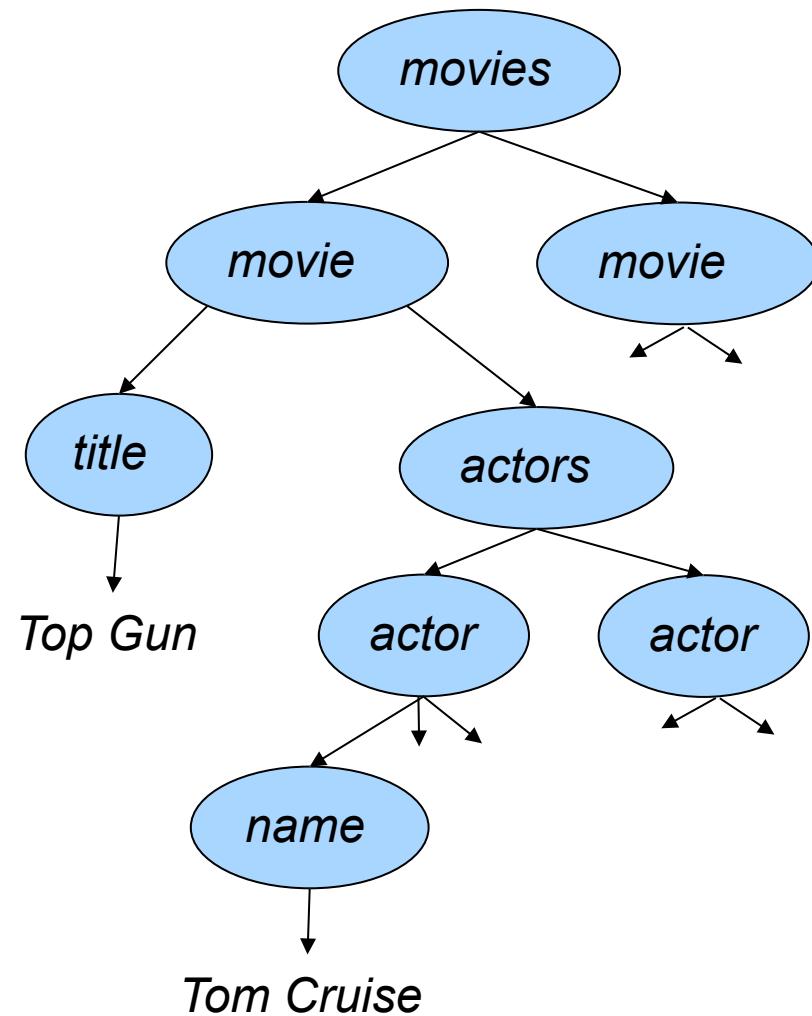
Graph Data Models: XML

- XML

- A commonly used data model for representing data without rigid structure
- In essence: a directed, labeled “tree”
- Very popular data exchange format

```
<movies>
  <movie>
    <title>Top Gun</title>
    <actors>
      <actor>
        <name>Tom Cruise</name>
        <born>7/3/1962</born>
      </actor>
      <actor>
        ...
        </actor>
      </actors>
    </movie>
  ...

```



Outline

- Background and Motivation
- Graph Queries and Analysis Tasks
- Graph Data Management: Storage
- Graph Data Management: Processing
- What we are doing

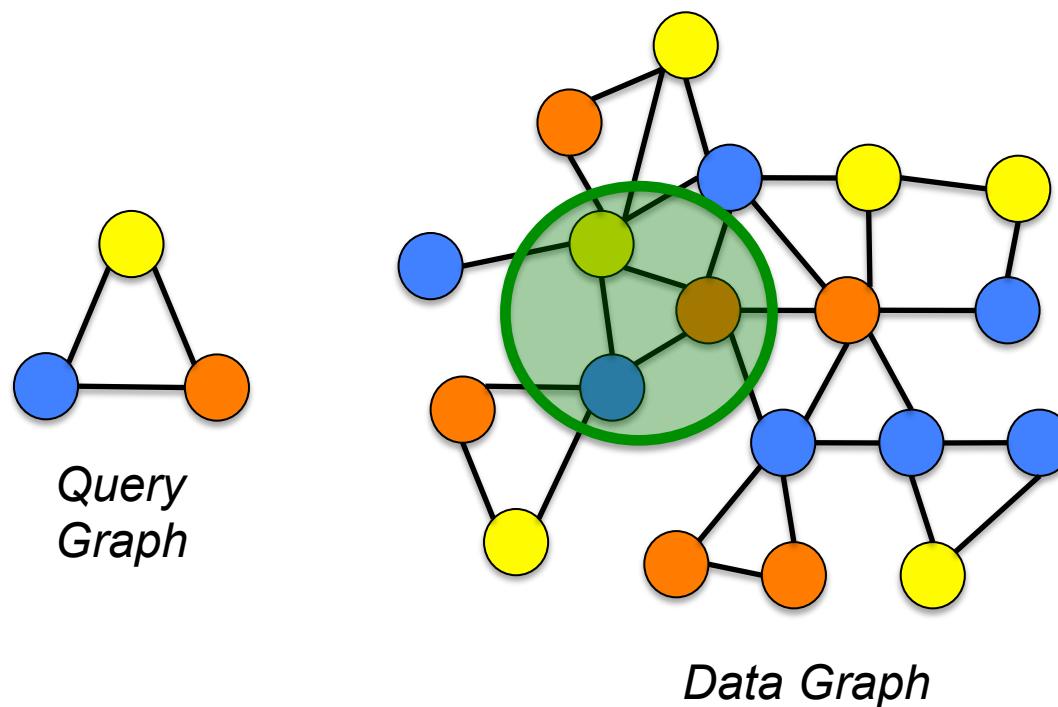
Queries vs Analysis Tasks

- **Queries:**
 - Permit focused exploration of the data
 - Result is typically a small portion of the graph (often just a node)
 - Main data management challenge:
 - Minimize the portion of the graph that is explored
 - Heavy use of “indexes” (auxiliary data structures)

- **Analysis tasks:**
 - Typically require processing the entire graph
 - Main data management challenges:
 - How to handle the large volume of data efficiently
 - How to parallelize if the data doesn’t fit in memory of a single machine

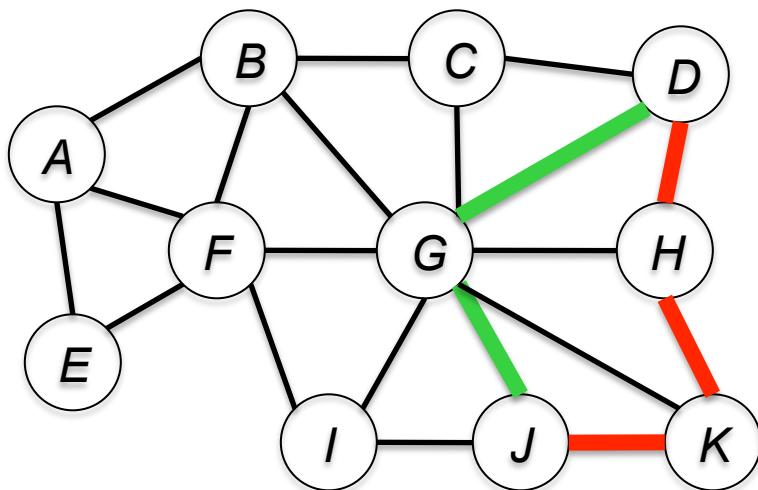
Queries: Subgraph Matching

- Given a “query” graph, find where it occurs in a given “data” graph
 - Query graph can specify restrictions on the graph structure, on values of node attributes, and so on
 - An important variation: *approximate* matching
- Alternatively, given a collection of data graphs, find the ones that contain the query graph



Queries: Connection Subgraphs

- Given a data graph and two (or more) nodes in it, find a small subgraph that best captures the relationship between the nodes
- Key question: How to define “best captures”?
 - E.g., “shortest path”: but that may not be most informative



The “red” path between D and J maybe more informative than the “green” path

Queries

- Reachability:
 - Given two nodes, is there an undirected or directed path between them?
 - ... with constraints on the types of edges that can be used?
- Shortest path:
 - Find the shortest path between two given nodes
- Keyword search:
 - Find the smallest subgraph that contains all the specified keywords
- Historical queries:
 - Given a node, find other nodes that evolved most similarly in the past
- And so on...

Graph Analysis: Centrality Measures

- Centrality measure: a measure of the relative importance of a vertex within a graph
- Many different centrality measures
 - ... that can give fairly different results

Degree centrality of a node u :

of edges incident on u

Betweenness centrality of a node u :

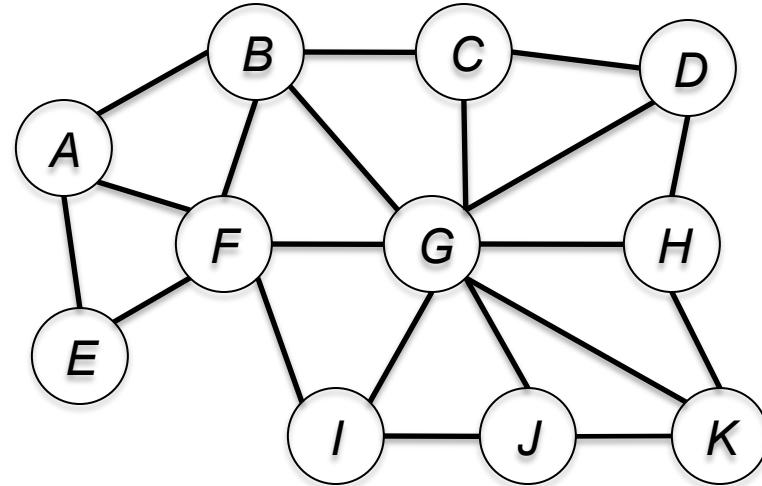
of shortest paths between pairs of vertices that go through u

Pagerank of a node u :

probability that a random surfer (who is following links randomly) ends up at node u

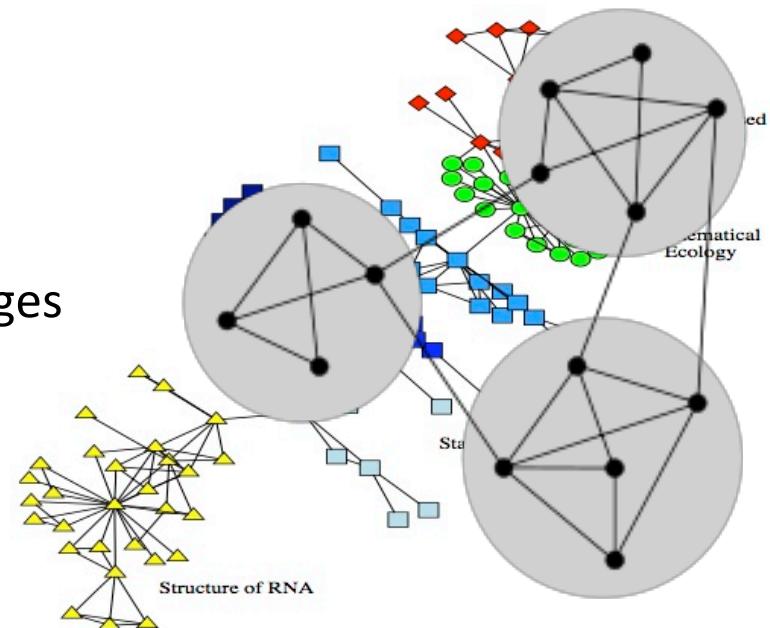
Eigenvector centrality:

Used in a recent work on analyzing Federal Funds Network



Graph Analysis: Community Detection

- Goal: partitioning the vertices into (potentially overlapping) groups based on the interconnections between them
 - Basic intuition: More connections within a community than across communities
 - Provide insights into how networks function; identify functional modules; improve performance of Web services...
- Numerous techniques proposed for community detection over the years
 - Graph partitioning-based methods
 - Maximizing some “goodness” function
 - Recursively removing high centrality edges
 - ... and so on



Graph Analysis: Models of Evolution

- Two somewhat related goals:
 - Measuring different properties of networks
 - E.g., degree distributions, diameter, clustering coefficient, ...
 - Using those to build models of how a network forms and evolves
 - To gain insights; for predictions about the future...
- Example:
 - Most real networks exhibit highly skewed degree distributions
 - *Preferential attachment model* explains that phenomenon
 - Basic idea: a new node is more likely to connect to a high-degree node than a low-degree node (“rich get richer”)
- Some other observed properties:
 - Shrinking diameters
 - Average degree in the network increases over time
 - High clustering coefficients

Outline

- Background and Motivation
- Graph Queries and Analysis Tasks
- Graph Data Management: Storage
- Graph Data Management: Processing
- What we are doing

Options for Storing Graph Data

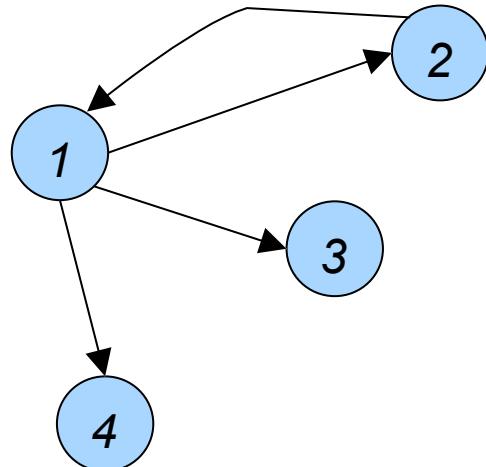
1. Use file systems
 - + Very simple, and no (practical) limits on how large a dataset to manage
 - No support for transactions; minimal functionality
2. Use a *relational* database (e.g., Oracle, IBM DB2, etc.)
 - + Mature technology – much of the data is already in them anyway
 - + All the goodies (SQL, transactions, toolchains) available
 - Almost no support for traversing the graph structure
3. Use NoSQL *key-value* stores
 - + Can handle very large datasets efficiently, in a distributed fashion
 - Minimal functionality – must build the analysis/querying tools on top
4. Use a persistent *graph database*
 - + Efficiently support *graph traversals*
 - But even the most mature products not as a mature as RDBMSs
 - Typically no declarative languages (a la SQL), so must write programs

Storage 1: File Systems

- Simplest to get started, and widely used in practice
 - Especially since the other options don't really help that much anyway for graph querying or analytics
- Many *cloud computing* programming frameworks read data from file systems
 - E.g., Hadoop Distributed File System (HDFS) used by Apache Hadoop and others
- Key disadvantages:
 - Almost no data management functionality
 - Everything from parsing to analyzing must be done by the programmer
 - No support for updates, or transactions
 - Hard to do “queries” without building auxiliary structures

Storage 2: Relational Databases

- Store the entities in a set of tables, and encode the connections between them in separate tables
 - E.g., RDF and XML data predominantly stored in relational databases
- Can use SQL to query the data, and other DBMS analytic tools
- However, no support for graph traversals
 - Must extract the relevant data, & write code to construct/process the graph
 - Can be much much slower than specialized solutions for traversal operations



Banks

ID	Name	...
1
2
3
4

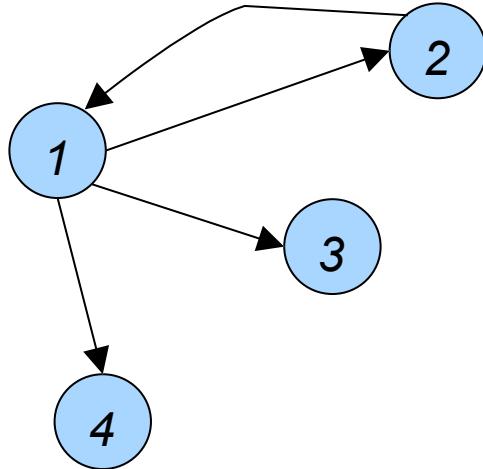
Trades

Bank1	Bank2	Date	...
1	2	...	
2	1	...	
3	1	...	
4	1	...	

Storage 3: Key-value Stores

- A recent, and wildly popular solution, to manage large datasets
 - Examples: Apache HBase, Cassandra, Amazon Dynamo, Redis
- Very basic functionality
 - $\text{Put}(k, v)$: Store the value v , and associate it with key k
 - $\text{Get}(k)$: Get the value associated with key k

One way to store it in a key-value store



Key	Value
Bank1	Information about the bank
Bank2	...
Bank1.outTrades	A list of all the trades where Bank1 is the seller
Bank1.inTrades	A list of all the trades where Bank1 is the buyer
...	...

Storage 3: Key-value Stores

- A recent, and wildly popular solution, to manage large datasets
 - Examples: Apache HBase, Cassandra, Amazon Dynamo, Redis
- Key-value stores manage the data in a distributed fashion
 - Can handle very large datasets with very low latencies
 - Underlie many Web applications (many Google products, Facebook, etc.)
- Advantages:
 - Support efficient updates (must be careful about consistency)
 - Fast retrieval → easy to traverse the graph structure
- Disadvantages:
 - Everything outside of graph traversals must be built on top

Storage 4: Specialized Graph Databases

- Built to manage and query graph-structured data
- Many built over the years, and increasing interest in recent years
 - **Neo4j**: Perhaps the most mature product out there
 - **InfiniteGraph**: Originally an object-oriented database
 - **DEX**: Quite similar to Neo4j in functionality
 - **AllegroGraph**: An RDF database
 - **HyperGraphDB**: Allows modeling hypergraphs
 - Wikipedia page on graph databases lists many more
- Key disadvantages:
 - Fairly rudimentary declarative interfaces -- most applications need to be written using programmatic interfaces
 - Or using provided toolkits/libraries

Neo4j

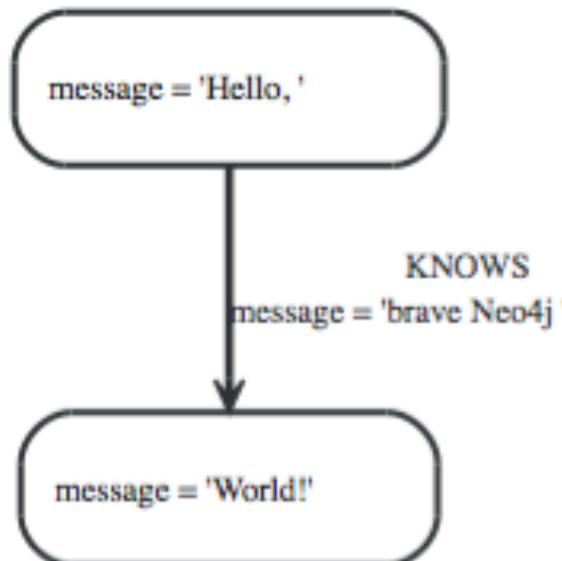
- Open-source graph database supported by Neo Technology
 - Uses the *property graph* model
 - The data stored on disks (unlike key-value stores)
 - Full ACID support (i.e., consistent and reliable updates)
 - Can scale to billions of nodes and edges
 - Supports many different APIs to access the data, and to retrieve the data
 - Highly efficient retrieval of nodes of interest through “indexing”

Neo4j

- Feature-rich Programmatic API to access a Neo4j database

```
firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = graphDb.createNode();
secondNode.setProperty( "message", "World!" );

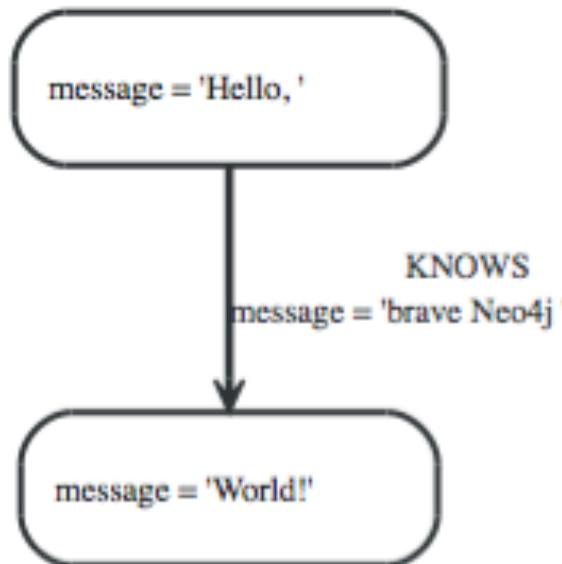
relationship = firstNode.createRelationshipTo( secondNode, RelTypes.KNOWS );
relationship.setProperty( "message", "brave Neo4j" );
```



Neo4j

- Feature-rich Programmatic API to access a Neo4j database

```
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );
```



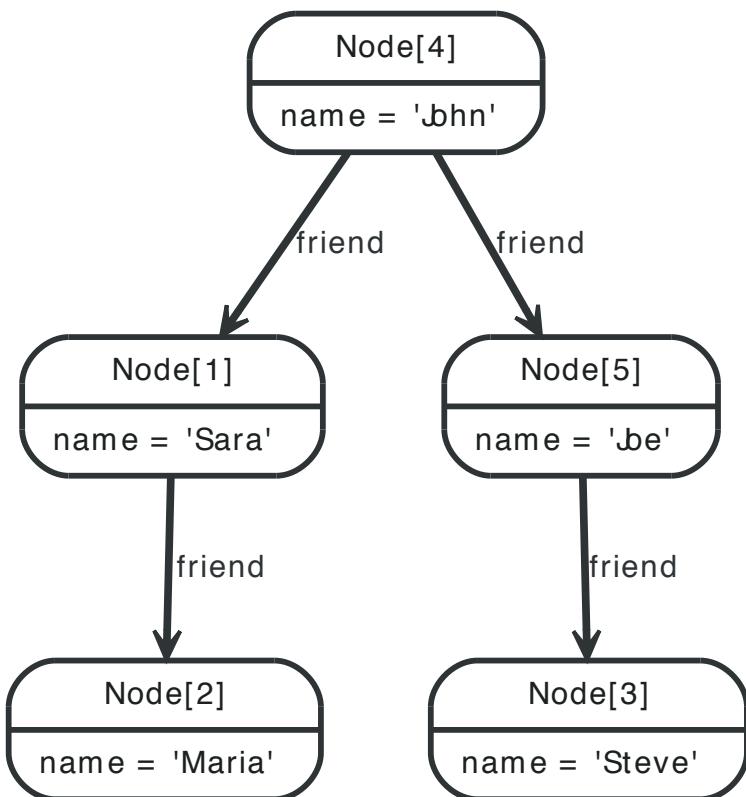
Neo4j

- Feature-rich Programmatic API to access a Neo4j database
 - Supports different types of indexes to quickly find “start” nodes
 - E.g., find the node for a particular person in a social network
 - Support different types of “traversals” to traverse the local neighborhoods

Neo4j

- Also supports a high-level language, called **cypher**, for traversing and searching

Figure 15.1. Example Graph



Finds friends of John's friends

```
START john=node:node_auto_index(name = 'John')
MATCH john-[:friend]->()-[:friend]->fof
RETURN john, fof
```

Neo4j

- Also supports a high-level language, called **cypher**, for traversing and searching
 - Can use **cypher** on its own (in a console), or in an embedded fashion (e.g., from within Java)
- Exports an API called BluePrints
 - BluePrints is analogous to JDBC for relational databases
 - Several open-source graph toolkits aim to use that API, and thus can be applied directly to data stored in Neo4j
 - E.g., Furnace, a graph algorithms package

AllegroGraph

- Aimed at Semantic Web Applications
- **Triple-store:** stores RDF assertions of the form
 - <subject, predicate, object>
 - E.g., <“sky”, “has-color”, “blue” >
- Full support for transactions, concurrency, recovery
- Several different ways to query:
 - Query patterns (specify the types of triples)
 - Has a Social Network Analysis Toolkit
 - Search methods, Centrality computations, etc.
 - Supports querying using Prolog
 - Supports SPARQL query language
 - ...

SPARQL

- Standardized RDF query language
- Basic functionality quite similar to *subgraph pattern matching*
 - But recent extensions attempt to go quite a bit beyond that

Find names and emails of every person in the dataset

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person a foaf:Person.
  ?person foaf:name ?name.
  ?person foaf:mbox ?email.
}
```

SPARQL

- Standardized RDF query language
- Basic functionality quite similar to *subgraph pattern matching*
 - But recent extensions attempt to go quite a bit beyond that

Find me all landlocked countries with a population greater than 15 million (revisited), with the highest population country first.

```
PREFIX type: <http://dbpedia.org/class/yago/>
```

```
PREFIX prop: <http://dbpedia.org/property/>
```

```
SELECT ?country_name ?population
WHERE {
    ?country a type:LandlockedCountries ;
              rdfs:label ?country_name ;
              prop:populationEstimate ?population .
    FILTER (?population > 15000000 &&
           langMatches(lang(?country_name), "EN")) .
} ORDER BY DESC(?population)
```

Outline

- Background and Motivation
- Graph Queries and Analysis Tasks
- Graph Data Management: Storage
- Graph Data Management: Processing
- What we are doing

Options for Processing Graph Data

1. Write your own programs
 - Extract the relevant data, and construct an in-memory graph
 - Different storage options help to different degrees with this
2. Write queries in a declarative language
 - Works for a very small class of graphs queries/tasks today
 - Ongoing research work (including in my group) on generalizing that
3. Use a general-purpose distributed programming framework
 - E.g.: Hadoop or MapReduce
 - Hard to program most graph analysis tasks this way
4. Use a graph-specific programming framework
 - Goal is to simplify writing graph analysis tasks, and scale them to very large volumes at the same time
 - Ongoing work – wide applicability still to be proven

Option 2: Declarative Interfaces

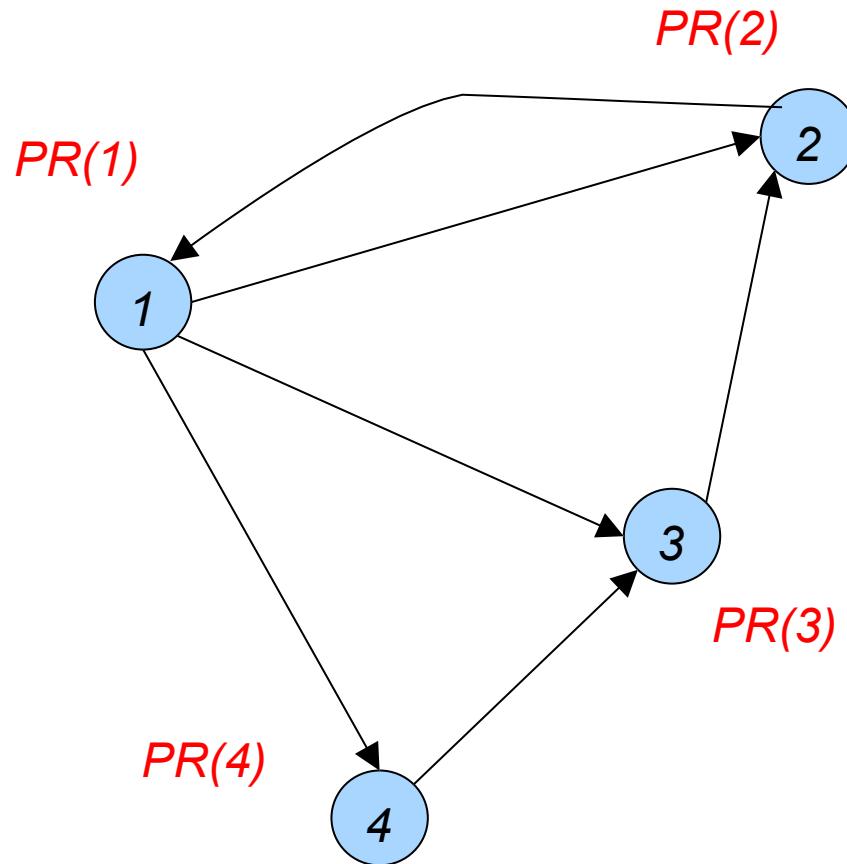
- No consensus on declarative, high-level languages (like SQL) for either querying or for analysis
 - Too much variety in the types of queries/analysis tasks
 - Makes it hard to find and exploit commonalities
- Some limited solutions:
 - XQuery for XML
 - Limited to tree-structured data
 - SPARQL for RDF
 - Standardized query language, but limited functionality
 - Cypher by Neo4j
 - Datalog-based frameworks for specifying analysis tasks
 - Mostly research prototypes, typically specific to some analysis task

Option 3: Map Reduce

- A very popular option for (batch) processing very large datasets
 - More specifically: Hadoop, the open source implementation
- Two key advantages:
 - Scalability without worrying about scheduling, distributed execution, fault tolerance, and so on...
 - Simple programming framework
- Disadvantages:
 - Hard to use this for graph analysis tasks
 - Each “traversal” effectively requires a new “map-reduce” phase
 - Map-reduce framework not ideal for large numbers of phases
- However, much work on showing how different graph analysis tasks can be done using MapReduce

Background: PageRank

- PageRank: A measure of *centrality* of a node
- $\text{PR}(\text{node})$ = Probability that a random *surfer* ends up at that node



$$\text{PR}(2) = \alpha/4 + (1 - \alpha) (\text{PR}(1)/3 + \text{PR}(3))$$

Probability of jumping to a random node

This PageRank defines a probability distribution over the nodes.
PageRank of node 3 distributed over its out-edges

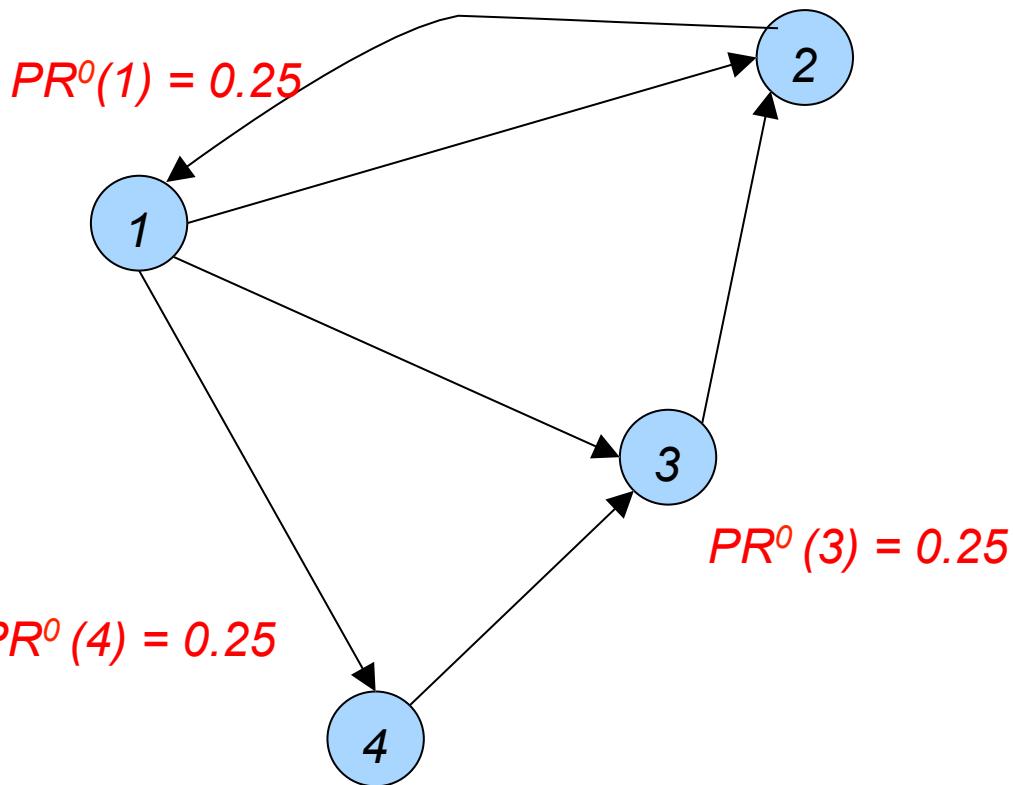
Background: PageRank

- PageRank: A measure of *centrality* of a node
- $\text{PR}(\text{node})$ = Probability that a random *surfer* ends up at that node
- Damping factor α needed to handle nodes with 0 out-degree and other special cases
 - Surfer may jump to a random page with probability α and restart
- How to compute?
 - Algebraically: using Gaussian Elimination
 - Not scalable to large graphs
 - Iteratively:
 - For the first iteration: $\text{PR}(n) = 1/N$ for all nodes
 - Repeatedly apply the formula using the $\text{PR}()$ values from the previous iteration
 - Typically 25-50 iterations enough to converge

Background: PageRank

- PageRank: A measure of *centrality* of a node
- $\text{PR}(\text{node})$ = Probability that a random *surfer* ends up at that node
- Iterative approach:

$$\text{PR}^0(2) = 0.25$$



Compute $\text{PR}^1(1), \dots$, using
 $\text{PR}^0(1), \dots$

E.g.,
 $\text{PR}^1(2) =$

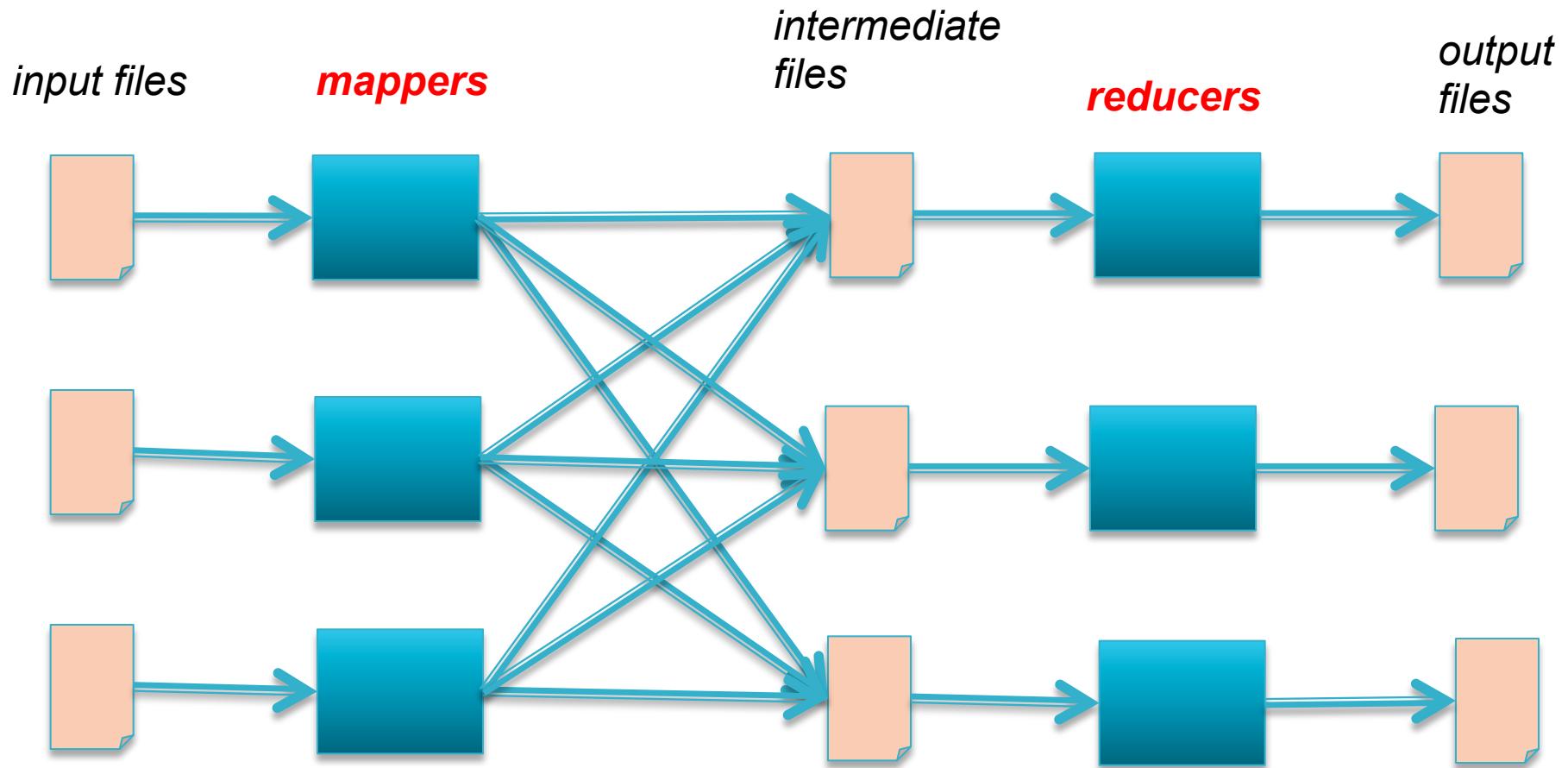
$$\alpha/4 +$$

$$(1 - \alpha) (\text{PR}^0(1)/3 + \text{PR}^0(3))$$

Option 3: Map Reduce

- Programmers write a pipeline of functions, called *map* or *reduce*
 - **map programs**
 - **inputs:** a list of “records” (record defined arbitrarily – could be images, genomes etc...)
 - **output:** for each record, produce a set of “(key, value)” pairs
 - **reduce programs**
 - **input:** a list of “(key, {values})” grouped together from the mapper
 - **output:** no specific restrictions – depends on what next
 - Both can do arbitrary computations on the input data as long as the basic structure is followed

Option 3: Map Reduce – PageRank



- All mappers run in parallel, typically on separate machines
Then all reducers run in parallel, typically on separate machines
- All the files are stored in a distributed file system

Option 3: Map Reduce – PageRank

input files

Node 1:
outEdges = 2, 3, 4
PR = 0.25

Node 2:
outEdges = 1
PR = 0.25

Node 3:
outEdges = 2
PR = 0.25

Node 4:
outEdges = 3
PR = 0.25

mappers

1, [2, 3, 4]
2, 0.25 / 3
3, 0.25 / 3
4, 0.25 / 3

3, [2]
2, 0.25

4, [3]
3, 0.25

Node 1's PR distributed over its outEdges

Node 2's PR distributed over its outEdges

Need to pass Along the graph structure

Adjacency List Representation

Option 3: Map Reduce – PageRank

mappers



1, [2, 3, 4]
2, 0.25/3
3, 0.25/3
4, 0.25/3

2, [1]
1, 0.25



3, [2]
2, 0.25



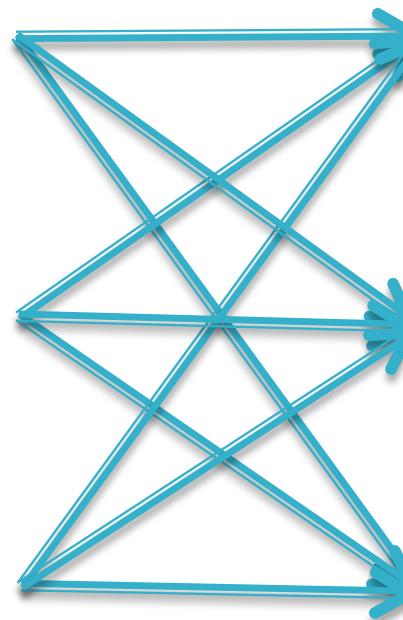
4, [3]
3, 0.25

intermediate files

1, [2, 3, 4]
2, 0.25/3
2, [1]
1, 0.25
2, 0.25

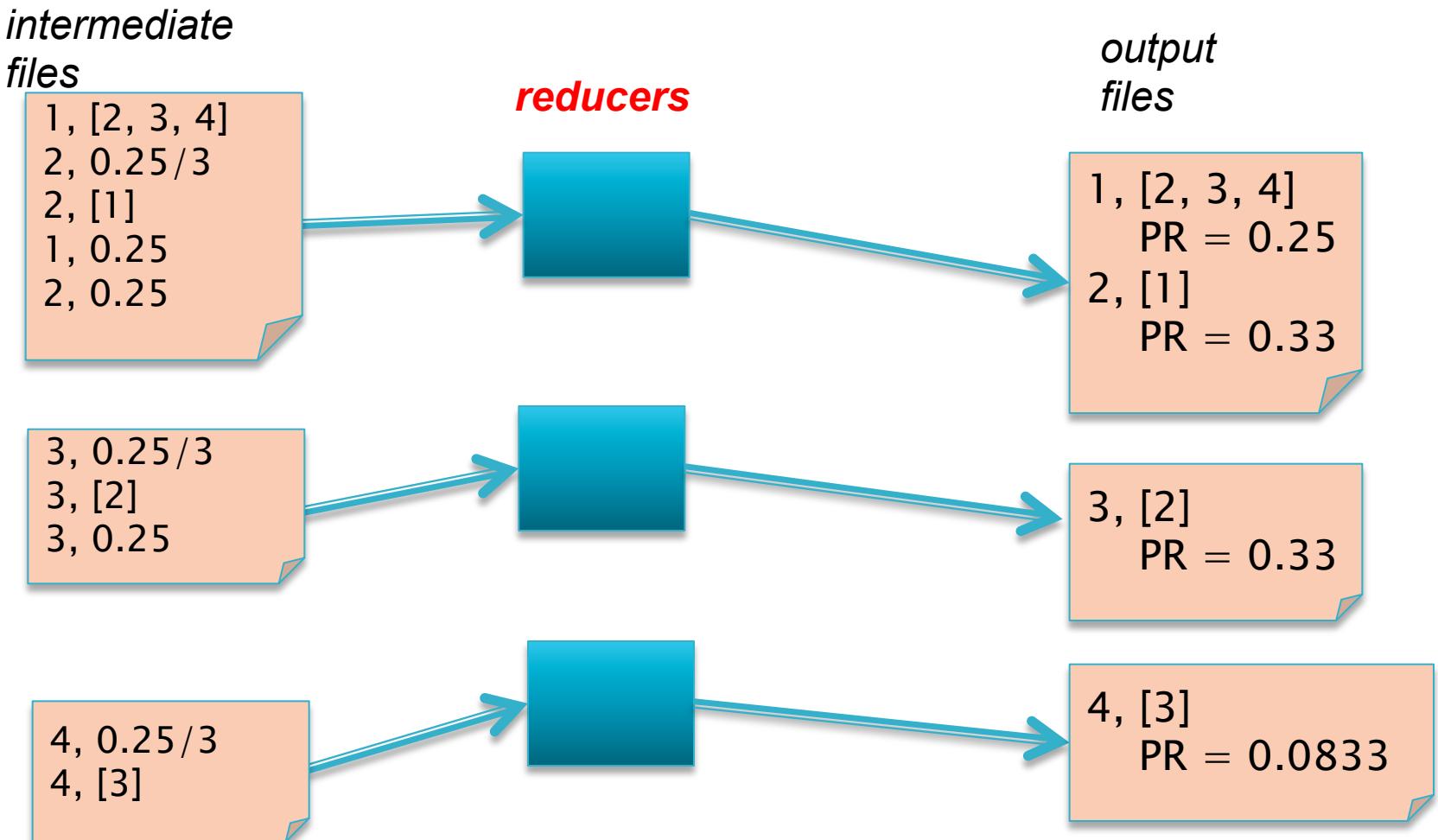
3, 0.25/3
3, [2]
3, 0.25

4, 0.25/3
4, [3]



“Shuffle” so that all the records with the same “key” end up in the same file

Option 3: Map Reduce – PageRank



Reduce:

*Compute the new PageRank (assume $\alpha = 0$)
Write out: graph structure + PageRank*

Option 3: Map Reduce – PageRank

intermediate files

1, [2, 3, 4]
2, 0.25/3
2, [1]
1, 0.25
2, 0.25

reducers

output files

1, [2, 3, 4]
PR = 0.25
2, [1]
PR = 0.33

3, 0.25/3
3, [2]
3, 0.25

3, [2]
PR = 0.33

4, 0.25/3
4, [3]

4, [3]
PR = 0.0833

REPEAT UNTIL CONVERGENCE

Option 3: Map Reduce

- A very popular option for (batch) processing very large datasets
 - More specifically: Hadoop, the open source implementation
- Two key advantages:
 - Scalability without worrying about scheduling, distributed execution, fault tolerance, and so on...
 - Simple programming framework
- Disadvantages:
 - Hard to use this for graph analysis tasks
 - Each “traversal” effectively requires a new “map-reduce” phase
 - Map-reduce framework not ideal for large numbers of phases
 - Not efficient – too much redundant work
 - In PageRank example: repeated reading and parsing of the inputs

Option 4: Graph Programming Frameworks

- Analogous frameworks proposed for analyzing large volumes of graph data
 - An attempt at addressing limitations of MapReduce
 - Most are *vertex-centric*
 - Programs written from the point of view of a vertex
 - Most based on message passing between nodes
- Pregel: original framework proposed by Google
 - Based on “Bulk Synchronous Protocol” (BSP)
- Giraph: an open-source implementation of Pregel
- GraphLab: asynchronous execution

Option 4: Pregel

- Programmers write one program: **compute()**
- Typical structure of **compute()**:
 - **Inputs**: current values associated with the node
 - **Inputs**: messages sent by the neighboring nodes
 - Do something...
 - Modify current values associated with the node (if desired)
 - **Outputs**: send messages to neighbors
- Execution framework:
 - Execute *compute()* for all the nodes in parallel
 - Synchronize (for all messages to be delivered)
 - Repeat

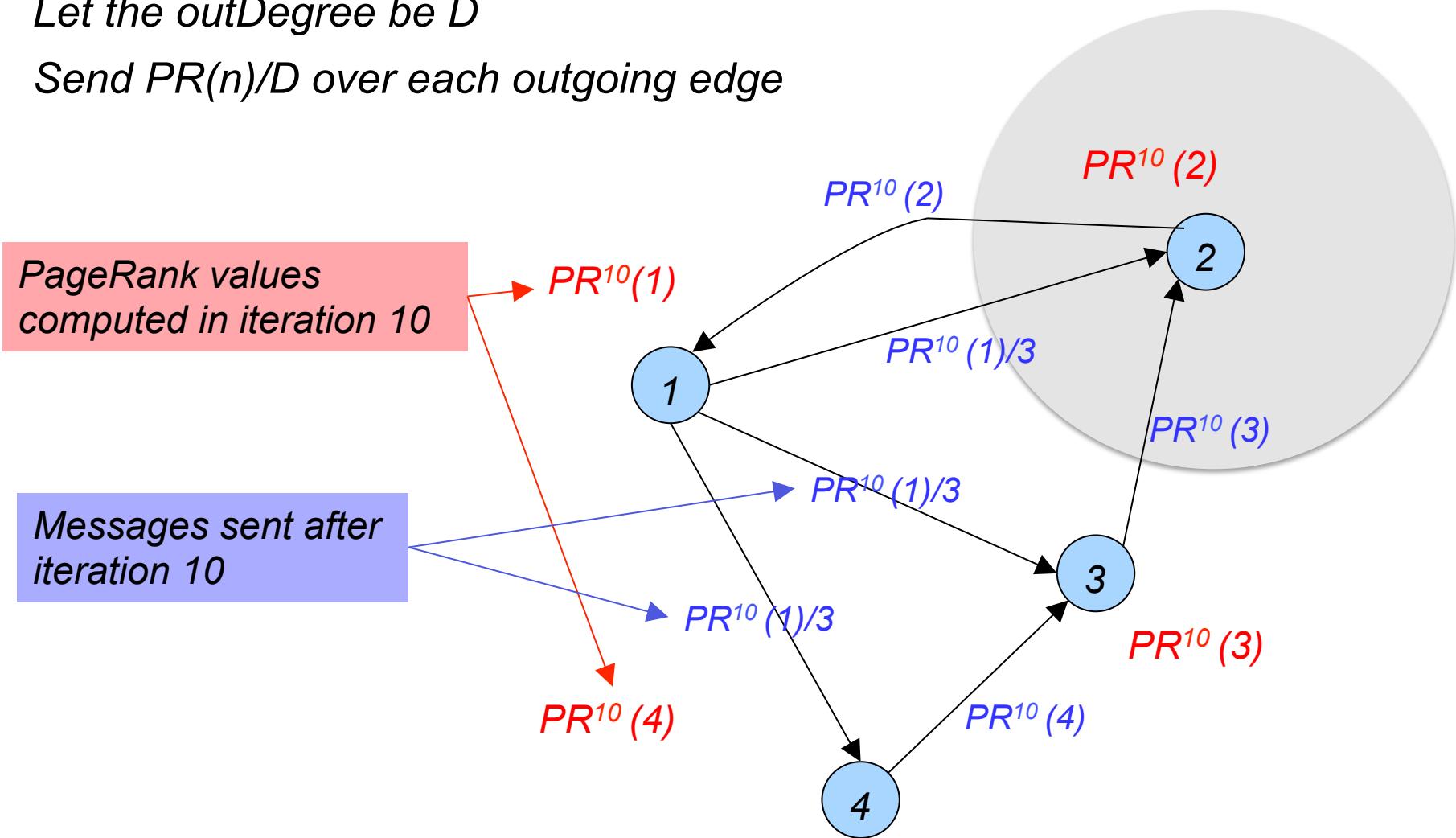
Option 4: Pregel – PageRank

Compute() at Node n:

$PR(n) = \text{sum up all the incoming weights}$

Let the *outDegree* be D

Send $PR(n)/D$ over each outgoing edge



Option 4: Graph Programming Frameworks

- Analogous frameworks proposed for analyzing large volumes of graph data
 - An attempt at addressing limitations of MapReduce
 - Most are *vertex-centric*
 - Programs written from the point of view of a vertex
 - Most based on message passing between nodes
- Vertex-centric frameworks somewhat limited and inefficient
 - Unclear how to do many complex graph analysis tasks
 - Not widely used yet