NoSQL Graph Databases

Problem Set #4 is graded

Problem Set #6 is done, you will all get 100!

Agenda

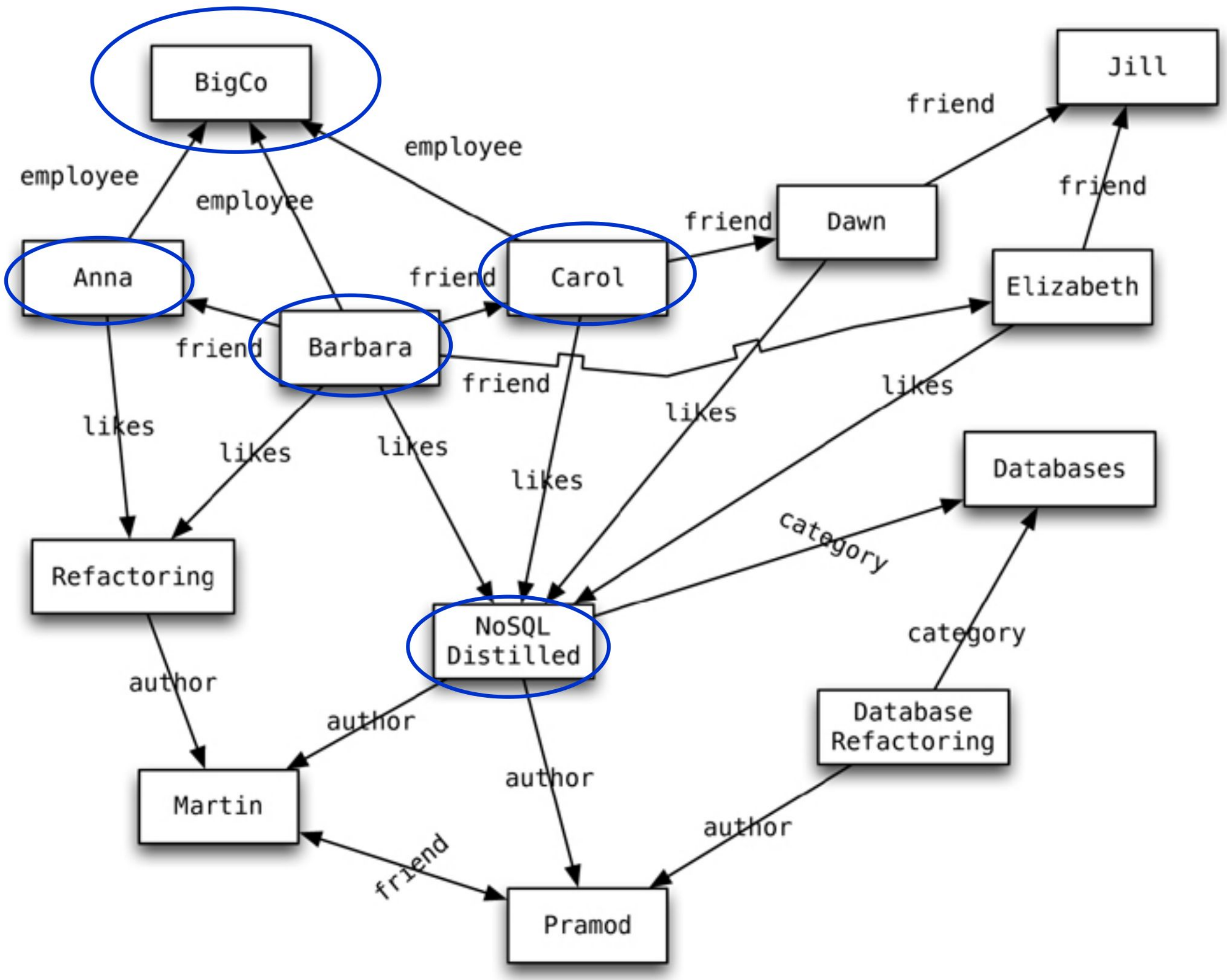
- Graph Databases: Mission, Data, Example
- A Bit of Graph Theory
  - Graph Representations
  - Algorithms: Improving Data Locality (efficient storage)
  - Graph Partitioning and Traversal Algorithms
- Graph Databases
  - Transactional databases
  - Non-transactional databases
- Neo4j
  - Basics, Native Java API, Cypher, Behind the Scene

Graph Databases: Concept

- To store entities and relationships between them
  - Nodes are instances of objects
  - Nodes have properties, e.g., name
  - Edges connect nodes and are directed
  - Edges have types (e.g., likes, friend, …)

- Nodes are organized by relationships
  - Allow to find interesting patterns
  - **example:** Get all nodes that are “employee” of “Big Company” and that “likes” “NoSQL Distilled”

Graph Databases: Example



source: Sadalage & Fowler: NoSQL Distilled, 2012

Graph Databases: Representatives

Neo4j

OrientDB

APACHE GIRAPH

TITAN

InfiniteGraph

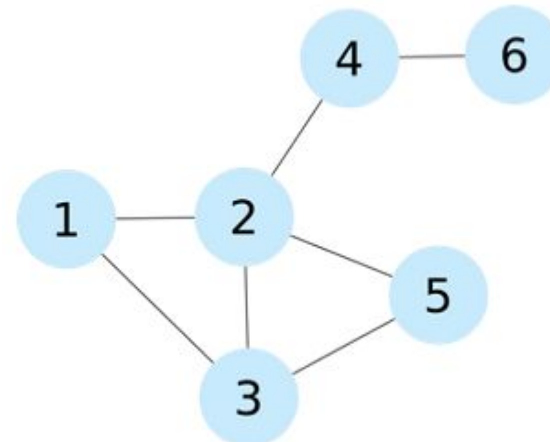
Graph Database Basics

- Data: a set of entities and their relationships
  - => we need to efficiently represent graphs
- Basic operations:
  - finding the neighbours of a node,
  - checking if two nodes are connected by an edge,
  - updating the graph structure, …
  - => we need efficient graph operations

- Graph $G = (V, E)$ is usually modelled as
  - set of nodes (vertices) V, $|V| = n$
  - set of edges E, $|E| = m$
- Which data structure to use?

Data Structure: Adjacency Matrix

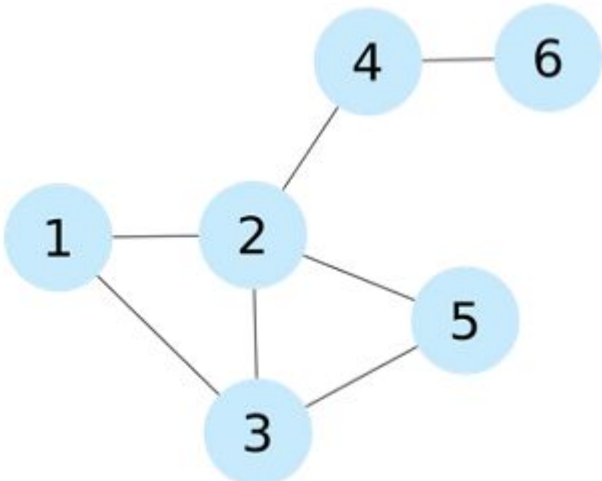
- Two-dimensional array A of $n \times n$ Boolean values
  - Indexes of the array = node identifiers of the graph
  - Boolean value A_{ij} indicates whether nodes i, j are connected



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	1	0
3	1	1	0	0	1	0
4	0	1	0	0	0	1
5	0	1	1	0	0	0
6	0	0	0	1	0	0

- Variants:
  - (Un)directed graphs
  - Weighted graphs…

Adjacency Matrix: Properties

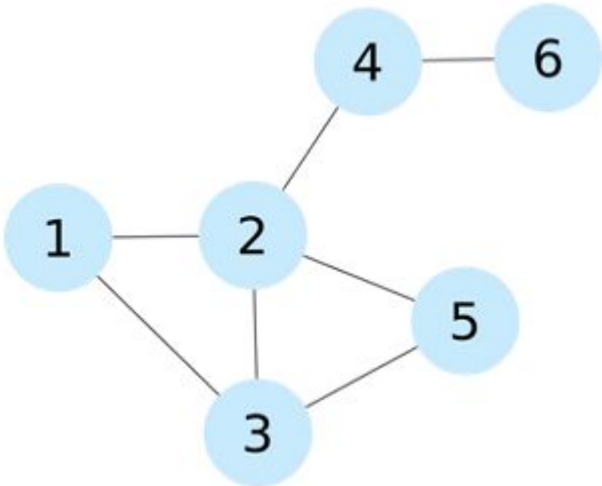


	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	1	0
3	1	1	0	0	1	0
4	0	1	0	0	0	1
5	0	1	1	0	0	0
6	0	0	0	1	0	0

- Pros:
  - Adding/removing edges
  - Checking if 2 nodes are connected
- Cons:
  - Quadratic space: $O(n^2)$
  - Sparse graphs (mostly 0s) are common
  - Adding nodes is expensive
  - Retrieval the neighbouring nodes takes linear time: $O(n)$

Data Structure: Adjacency List

- A dictionary or list of lists, describing the neighbours of the key or indexed node
  - Vector of *n* pointers to adjacency lists

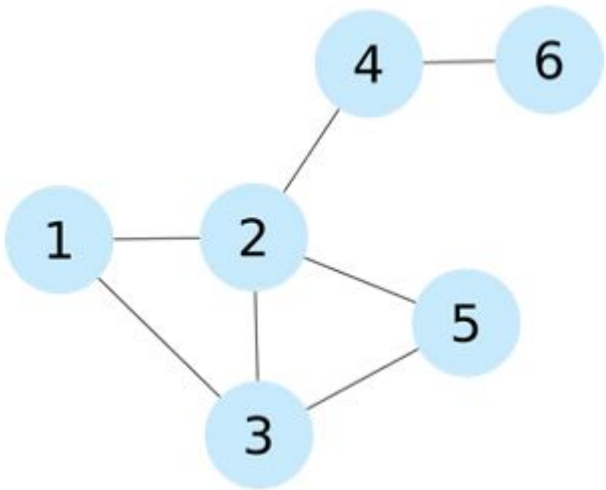


- Undirected graph:
  - An edge connects nodes *i* and *j*
  - => the adjacency list of *i* contains node *j* and vice versa

```
Neighbors[1] = [2,3]
Neighbors[2] = [1,3,5]
Neighbors[3] = [1,2,5]
Neighbors[4] = [2,6]
Neighbors[5] = [2,3]
Neighbors[6] = [4]
```

- Often compressed
  - Exploiting regularities in graphs

Adjacency List: Properties

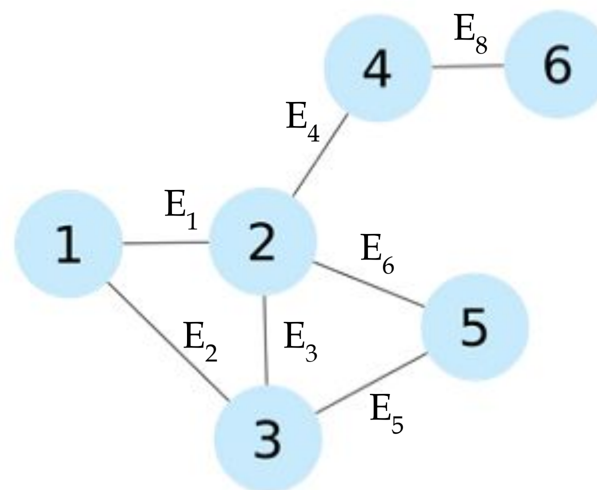



```
Neighbors[1] = [2,3]
Neighbors[2] = [1,3,5]
Neighbors[3] = [1,2,5]
Neighbors[4] = [2,6]
Neighbors[5] = [2,3]
Neighbors[6] = [4]
```

- Pros:
  - Getting the neighbours of a node
  - Cheap addition of nodes
  - More compact representation of sparse graphs
- Cons:
  - Checking if an edge exists between two nodes
    - Optimization: sorted lists => logarithmic scan, but also logarithmic insertion

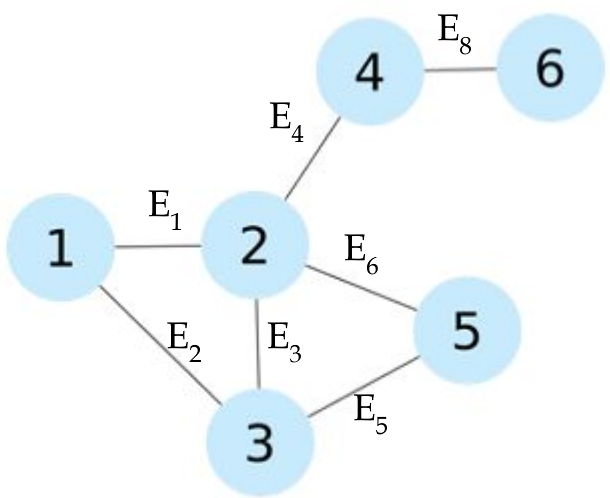
Data Structure: Incidence Matrix

- Two-dimensional Boolean matrix of *n* rows and *m* columns
  - A column represents an edge
    - Nodes that are connected by a certain edge
  - A row represents a node
    - All edges that are connected to the node



	E_1	E_2	E_3	E_4	E_5	E_7	E_8
1	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0
3	0	1	1	0	0	1	0
4	0	0	0	1	0	0	1
5	0	0	0	0	1	1	0
6	0	0	0	0	0	0	1

Incidence Matrix: Properties

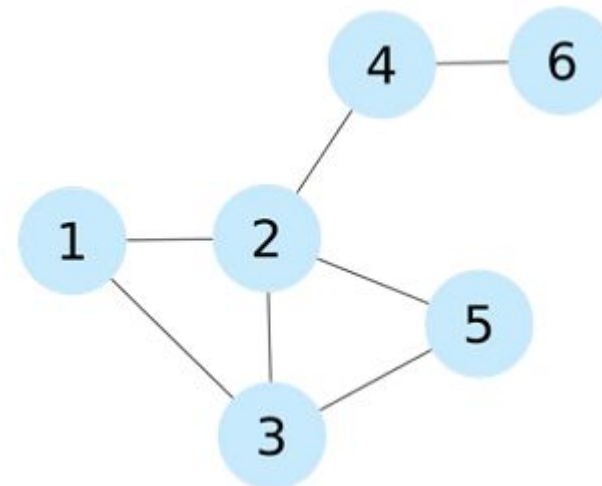


	E_1	E_2	E_3	E_4	E_5	E_7	E_8
1	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0
3	0	1	1	0	0	1	0
4	0	0	0	1	0	0	1
5	0	0	0	0	1	1	0
6	0	0	0	0	0	0	1

- Pros:
  - Can represent hypergraphs
    - where one edge connects an arbitrary number of nodes
- Cons:
  - Requires $n \times m$ bits (for most graphs $m >> n$)

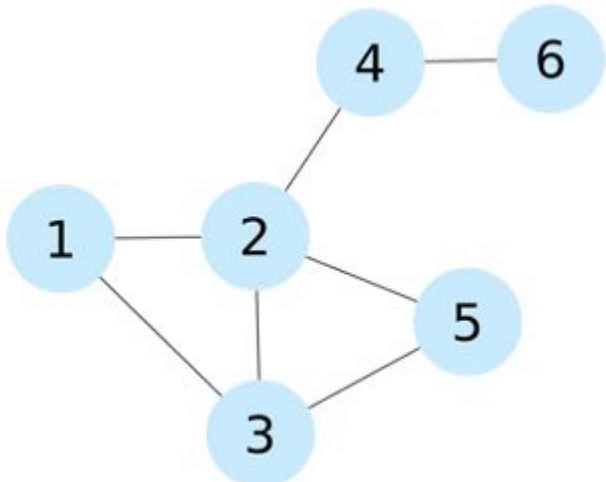
Data Structure: Laplacian Matrix

- Two-dimensional array of $n \times n$ integers
  - Similar structure to adjacency matrix
  - Diagonal of the Laplacian matrix indicates the degree of the node
  - L_{ij} is set to *-1* if the two vertices i and j are connected, *0* otherwise



	1	2	3	4	5	6
1	2	-1	-1	0	0	0
2	-1	4	-1	-1	-1	0
3	-1	-1	3	0	-1	0
4	0	-1	0	2	0	-1
5	0	-1	-1	0	2	0
6	0	0	0	-1	0	1

Laplacian Matrix: Properties



	1	2	3	4	5	6
1	2	-1	-1	0	0	0
2	-1	4	-1	-1	-1	0
3	-1	-1	3	0	-1	0
4	0	-1	0	2	0	-1
5	0	-1	-1	0	2	0
6	0	0	0	-1	0	1

All features of adjacency matrix

- Pros:
  - Analyzing the graph structure by means of spectral analysis
    - Calculating eigenvalues of the matrix

Basic Graph Algorithms

- Visiting all nodes:
  - Breadth-first Search (BFS)
  - Depth-first Search (DFS)
- Shortest path between two nodes
- Single-source shortest path problem
  - BFS (unweighted),
  - Dijkstra (nonnegative weights),
  - Bellman-Ford algorithm
- All-pairs shortest path problem
  - Floyd-Warshall algorithm

http://en.wikipedia.org/wiki/Shortest_path_problem

Improving Data Locality

- Performance of the read/write operations
  - Depends also on physical organization of the data
  - Objective: Achieve the best "data locality"
- Spatial locality:
  - if a data item has been accessed, the nearby data items are likely to be accessed in the following computations
    - e.g., during graph traversal
- Strategy:
  - in graph adjacency matrix representation, exchange rows and columns to improve the disk cache hit ratio
  - Specific methods: BFSL, Bandwidth of a Matrix, ...

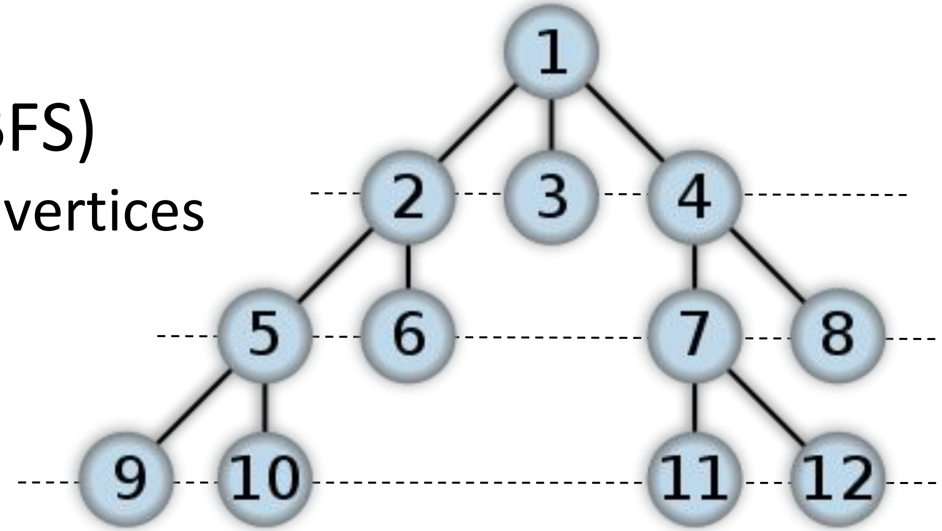
Breadth First Search Layout (BFSL)

- Input: vertices of a graph
- Output: a permutation of the vertices
  - with better cache performance for graph traversals

- BFSL algorithm:
  1. Select a node (at random, the origin of the traversal)
  2. Traverse the graph using the BFS alg.
     - generating a list of vertex identifiers in the order they are visited
  3. Take the generated list as the new vertices permutation

Breadth First Search Layout (2)

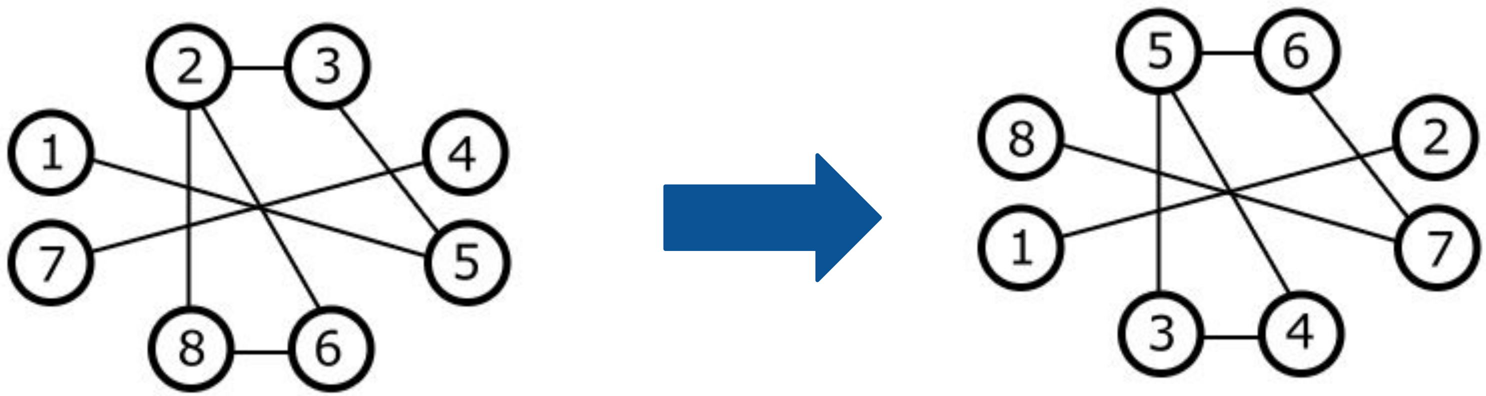
- Let us recall:
  Breadth First Search (BFS)
  - FIFO queue of frontier vertices



- Pros: optimal when starting from the same node
- Cons: starting from other nodes
  - The further, the worse

Matrix Bandwidth: Motivation

- Graph represented by adjacency matrix



1	0	0	0	1	0	0	0
0	1	1	0	0	1	0	1
0	1	1	0	1	0	0	0
0	0	0	1	0	0	1	0
1	0	1	0	1	0	0	0
0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	0
0	1	0	0	0	1	0	1

1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	0	0
0	0	1	1	1	1	0	0
0	0	0	0	1	1	1	0
0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	1

Matrix Bandwidth: Formalization

- The minimum bandwidth problem
  - Bandwidth of a row in a matrix = the maximum distance between nonzero elements, where one is left of the diagonal and the other is right of the diagonal
  - Bandwidth of a matrix = maximum bandwidth of its rows

- Low bandwidth matrices are more cache friendly
  - Non zero elements (edges) clustered about the diagonal

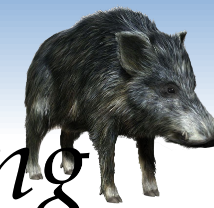
- Bandwidth minimization problem: NP hard
  - For large matrices the solutions are only approximated

Graph Partitioning

- Some graphs are too large to be fully loaded into the main memory of a single computer
  - Usage of secondary storage degrades the performance
  - Scalable solution: distribute the graph on multiple nodes

- We need to partition the graph reasonably
  - Usually for a particular (set of) operation(s)
    - The shortest path, finding frequent patterns, BFS, spanning tree search

- This is difficult and graph DB are often centralized



Example: 1-Dimensional Partitioning

- Aim: partitioning the graph to solve BFS efficiently
  - Distributed into shared-nothing parallel system
  - Partitioning of the adjacency matrix

- 1D partitioning:
  - Matrix rows are randomly assigned to the *P* nodes (processors) in the system
  - Each vertex and the edges emanating from it are owned by one processor

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	1	0
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0	1	1	0	1	1	0	0
12	0	0	0	1	1	0	0	0	1	0	0	0

One-Dimensional Partitioning: BFS

- BSF with 1D partitioning
  1. Each **processor** has a set of vertices *F* (FIFO)
  2. The lists of neighbors of the vertices in *F* forms a set of neighbouring vertices *N*
     - Some owned by the current processor, some by others
  3. Messages are sent to all other processors… etc.

- 1D partitioning leads to high messaging
  - => 2D-partitioning of adjacency matrix
  - … lower messaging but still very demanding

Efficient sharding of a graph can be difficult

Types of Graph Databases

- **Single-relational graphs**
  - Edges are homogeneous in meaning
    - e.g., all edges represent friendship
- **Multi-relational (property) graphs**
  - Edges are labeled by type
    - e.g., friendship, business, communication
  - Vertices and edges maintain a set of key/value pairs
    - Representation of non-graphical data (properties)
    - e.g., name of a vertex, the weight of an edge

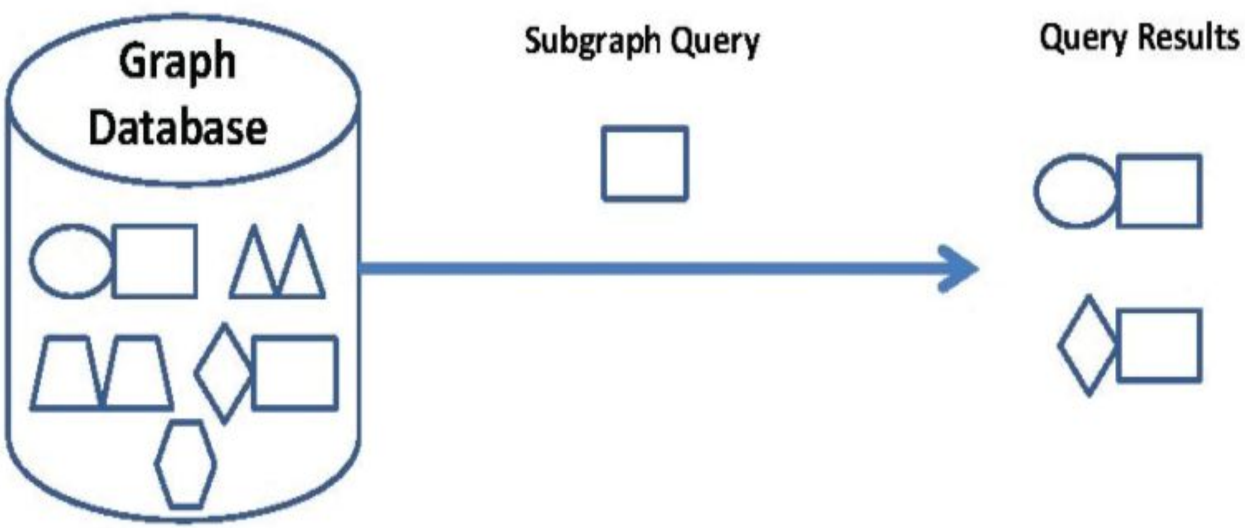
Graph Databases

- A graph database = a set of graphs

- Types of graph databases:
  - Transactional = large set of small graphs
    - e.g., chemical compounds, biological pathways, …
    - Searching for graphs that match the query
  - Non-transactional = few numbers of very large graphs
    - or one huge (not connected) graph
    - e.g., Web graph, social networks, …

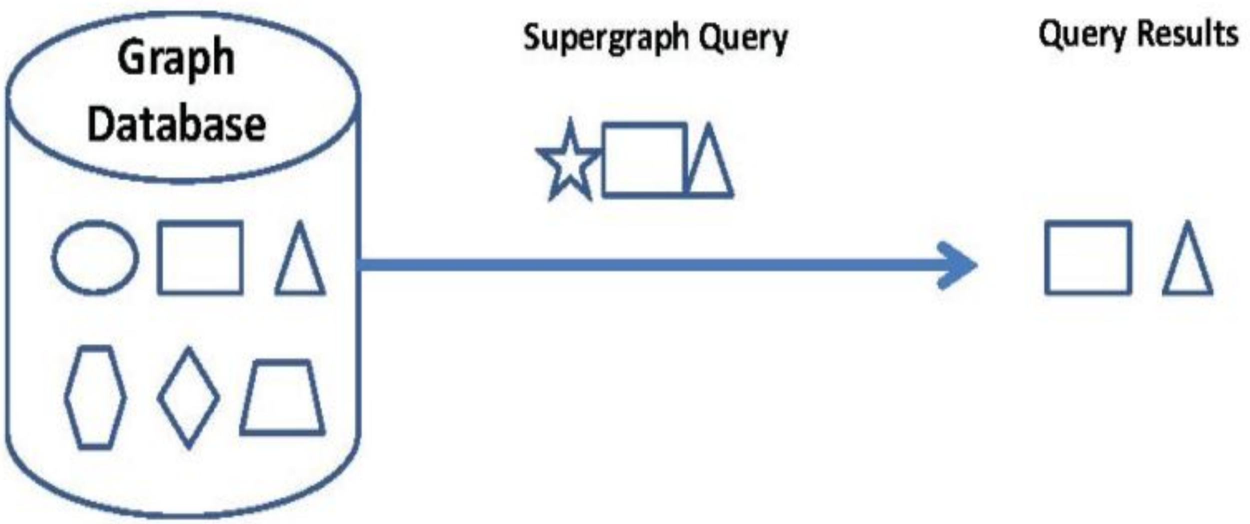
Transactional DBs: Queries

- Types of Queries
  - Subgraph queries
    - Search for a specific pattern in the graph database
    - Query = a small graph or a graph, where some parts are uncertain
      - e.g., vertices with wildcard labels
    - More general type: allow sub-graph isomorphism



Transactional DBs: Queries (2)

- Super-graph queries
  - Search for the graph database members whose whole structure is contained in the input query



- Similarity (approximate matching) queries
  - Finds graphs which are similar to a given query graph
    - but not necessarily isomorphic
  - Key question: how to measure the similarity

Indexing & Query Evaluation

- **Extract** certain **characteristics** from each graph
  - And **index** these characteristics for each $G_1, \ldots, G_n$

- **Query** evaluation in transactional graph DB
  1. Extraction of the **characteristics** from **query** graph q
  2. **Filter** the database (index) and identify a **candidate** set
     - Subset of the $G_1, \ldots, G_n$ graphs that should contain the answer
  3. **Refinement** - check all candidate graphs

Subgraph Query Processing

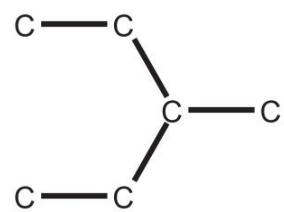
1. Mining-based Graph Indexing Techniques
   - Idea: if some features of query graph *q* do not exist in data graph *G*, then *G* cannot contain *q* as its subgraph
   - Apply graph-mining methods to extract some features (sub-structures) from the graph database members
     - e.g., frequent sub-trees, frequent sub-graphs
   - An inverted index is created for each feature

2. Non Mining-Based Graph Indexing Techniques
   - Indexing of the whole constructs of the graph database
     - Instead of indexing only some selected features

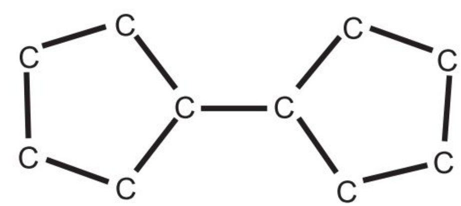
Mining-based Technique

- Example method: GIndex [2004]
  - Indexing "frequent discriminative graphs"
  - Build inverted index for selected discriminative subgraphs

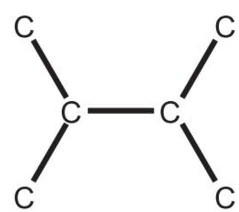
G_1



G_2



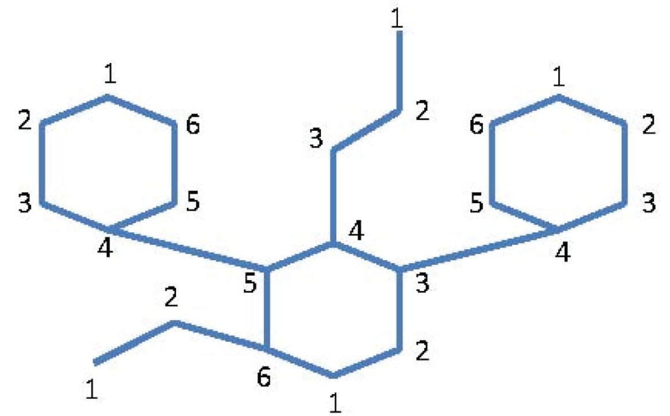
G_3

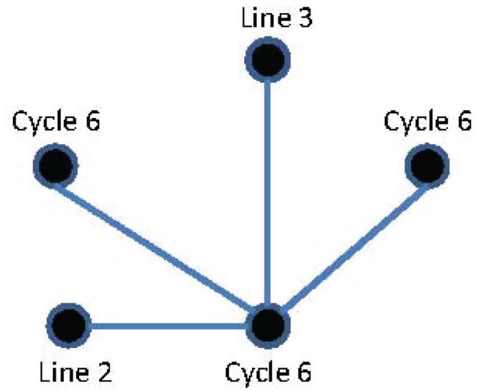


G_d

Non Mining-based Techniques

- **Example:** GString (2007)
  - Model the graphs in the context of organic chemistry using basic structures
    - Line = series of vertices connected end to end
    - Cycle = series of vertices that form a close loop
    - Star = core vertex directly connects to several vertices





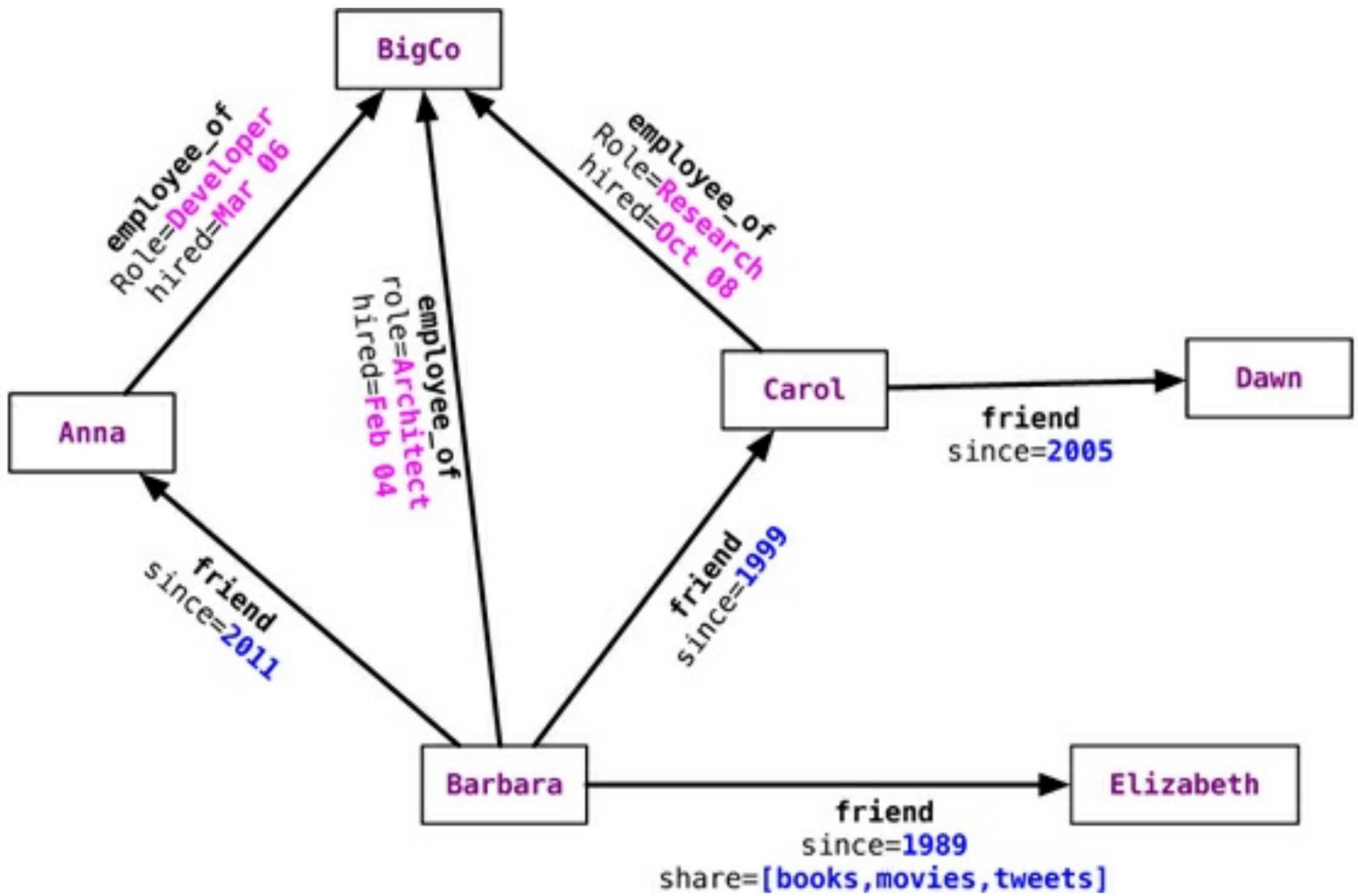
Non-transactional Graph Databases

- A few very large graphs
  - e.g., Web graph, social networks, …
- Queries:
  - Nodes/edges with properties
  - Neighboring nodes/edges
  - Paths (all, shortest, etc.)

Basic Characteristics

- Different types of relationships between nodes
  - To represent relationships between domain entities
  - Or to model any kind of secondary relationships
    - Category, path, time-trees, spatial relationships, …
- No limit to the number and kind of relationships
- Relationships have: type, start node, end node, own properties
  - e.g., “since when” did they become friends

Relationship Properties: Example

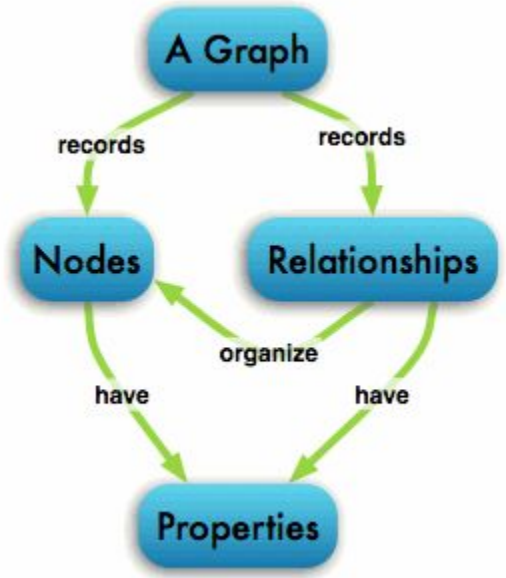


Graph DB vs. RDBMS

- RDBMS designed for a single type of relationship
  - "Think org charts"
  - Who works for who
  - Who is our lowest level common manager
- Adding a new relationship implies schema changes
  - New tables with foreign keys referencing other tables
- In RDBMS we model the graph beforehand based on the traversal we want
  - If the traversal changes, the data will have to change
  - Graph DBs: the relationship is not calculated but persisted

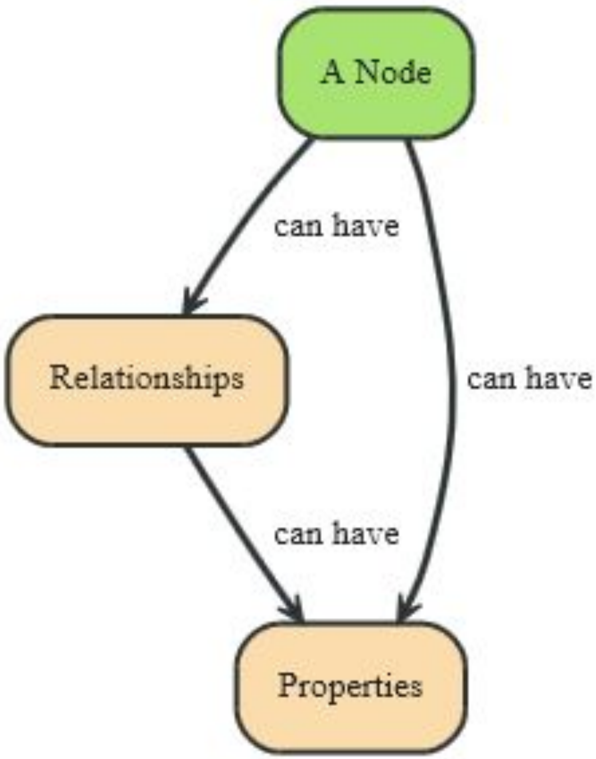
Neo4j: An exemplar Graph database

- Open source graph database
  - The most popular
- Initial release: 2007
- Written in: Java
- OS: cross-platform
- Stores data as nodes connected by directed, typed relationships
  - With properties on both
  - Called the "property graph"



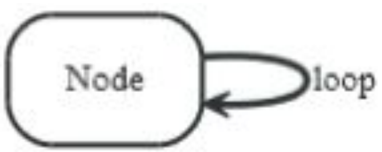
Neo4j: Data Model

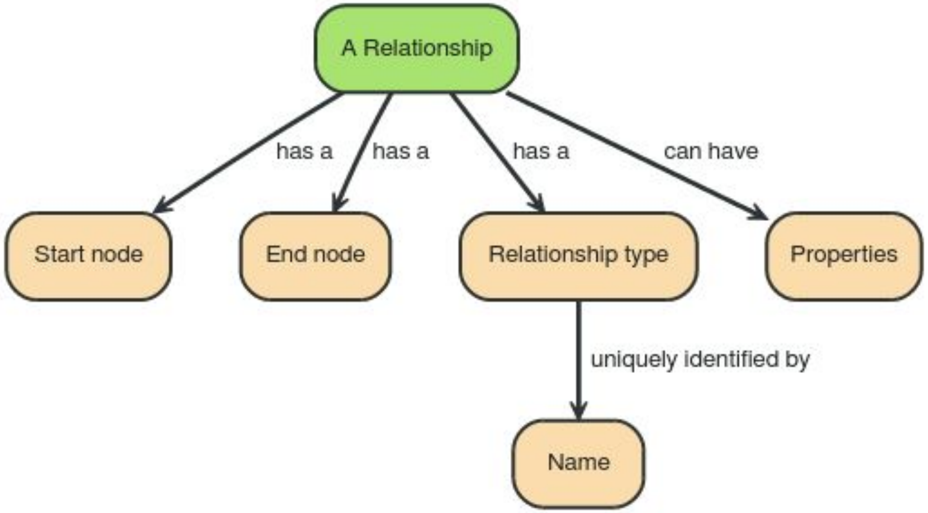
- Fundamental units: nodes + relationships
- Both can contain properties
  - Key-value pairs
  - Value can be of primitive type or an array of primitive type
  - null is not a valid property value
    - nulls can be modelled by the absence of a key



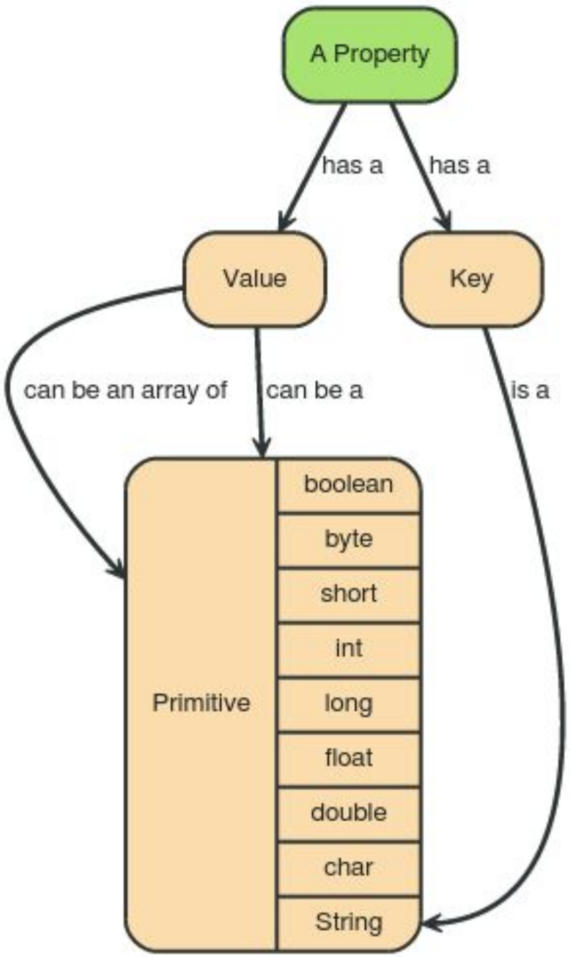
Data Model: Relationships

- Directed relationships (edges)
  - Incoming and outgoing edge
    - Equally efficient traversal in both directions
    - Direction can be ignored
      if not needed by the application
  - Always a start
    and an end node
    - Can be recursive



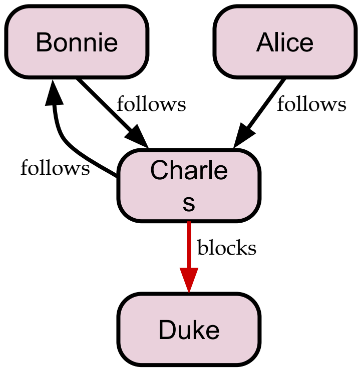


Data Model: Properties



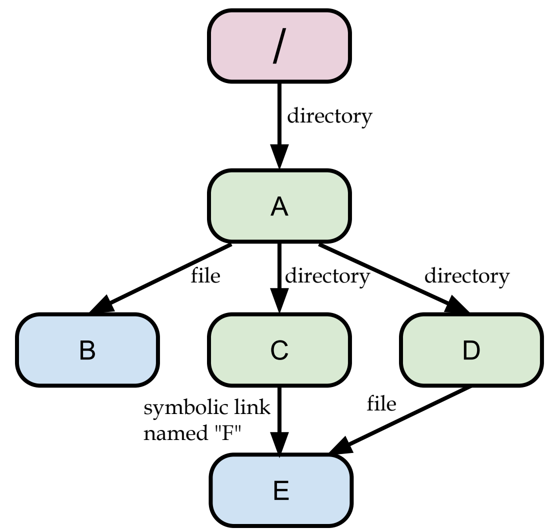
Type	Description
boolean	true/false
byte	8-bit integer
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit IEEE 754 floating-point number
double	64-bit IEEE 754 floating-point number
char	16-bit unsigned integers representing Unicode characters
String	sequence of Unicode characters

Examples



What	How
get who a person follows	outgoing *follows* relationships, depth one
get the followers of a person	incoming *follows* relationships, depth one
get who a person blocks	outgoing *blocks* relationships, depth one

What	How
get the full path of a file	incoming *file* relationships
get all paths for a file	incoming *file* and *symbolic link* relationships
get all files in a directory	outgoing *file* and *symbolic link* relationships, depth one
get all files in a directory, excluding symbolic links	outgoing *file* relationships, depth one
get all files in a directory, recursively	outgoing *file* and *symbolic link* relationships



Access to Neo4j

- Embedded database in Java system
- Language-specific connectors
  - Libraries to connect to a running Neo4j server
- Cypher query language
  - Standard language to query graph data
- HTTP REST API
- Gremlin graph traversal language (plugin)
- etc.

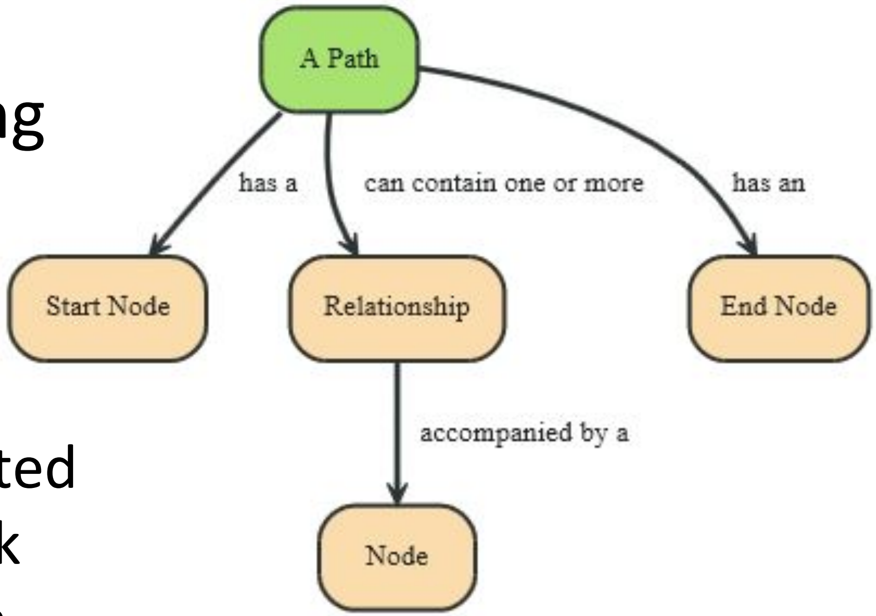
Native Java Interface: Example

```
Node alice = graphDb.createNode();
alice.setProperty("name", "Alice");
Node bonnie = graphDb.createNode();
bonnie.setProperty("name", "Bonnie");

Relationship a2b = alice.createRelationshipTo(bonnie,
FRIEND);
Relationship b2a = bonnie.createRelationshipTo(alice,
FRIEND);

a2b.setProperty("quality", "a good one");
b2a.setProperty("since", 2003);
```

- Undirected edge:
  - Relationship between the nodes in both directions
  - **INCOMING** and **OUTGOING** relationships from a node

Data Model: Traversal + Path

- **Path** = one or more nodes + connecting relationships
  - Typically retrieved as a result of a query or a traversal



- **Traversing a graph** = visiting its nodes, following relationships according to some rules
  - Typically, a subgraph is visited
  - Neo4j: Traversal framework + Java API, Cypher, Gremlin

Traversal Framework

- A traversal is influenced by
  - Starting node(s) where the traversal will begin
  - Expanders – define what to traverse
    - i.e., relationship direction and type
  - Order – depth-first / breadth-first
  - Uniqueness – visit nodes (relationships, paths) only once
  - Evaluator – what to return and whether to stop or continue traversal beyond a current position

Traversal = TraversalDescription + starting node(s)

Traversal Framework – Java API

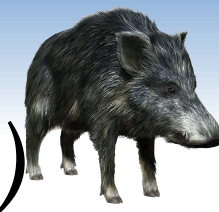
- `org.neo4j...TraversalDescription`
  - The main interface for defining traversals
    - Can specify branch ordering `breadthFirst()` / `depthFirst()`

- `.relationships()`
  - Adds the relationship type to traverse
    - e.g., traverse only edge types: FRIEND, RELATIVE
    - Empty (default) = traverse all relationships
  - Can also specify direction
    - `Direction.BOTH`
    - `Direction.INCOMING`
    - `Direction.OUTGOING`

Traversal Framework – Java API (2)

- `org.neo4j...Evaluator`
  - Used for deciding at each node: should the traversal continue, and should the node be included in the result
    - `INCLUDE_AND_CONTINUE`: Include this node in the result and continue the traversal
    - `INCLUDE_AND_PRUNE`: Include this node, do not continue traversal
    - `EXCLUDE_AND_CONTINUE`: Exclude this node, but continue traversal
    - `EXCLUDE_AND_PRUNE`: Exclude this node and do not continue
  - Pre-defined evaluators:
    - `Evaluators.toDepth(int depth)` / `Evaluators.fromDepth(int depth)`,
    - `Evaluators.excludeStartPosition()`
    - …

Traversal Framework – Java API (3)

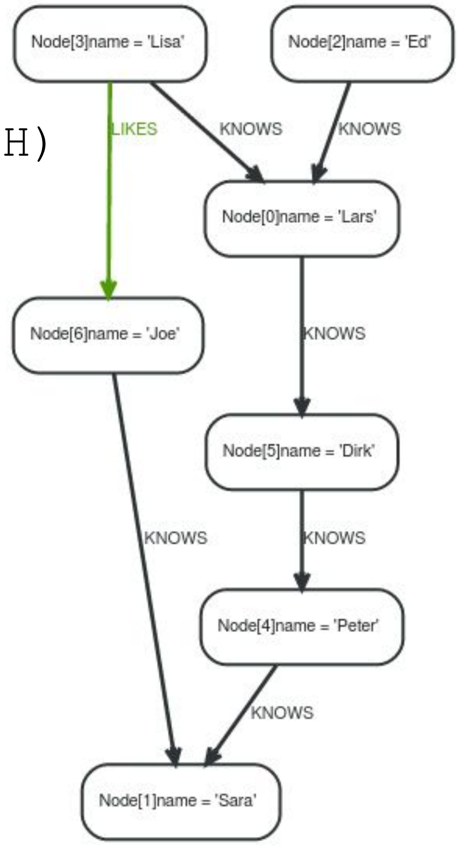


- `org.neo4j...Uniqueness`
  - Can be supplied to the TraversalDescription
  - Indicates under what circumstances a traversal may revisit the same position in the graph

- `Traverser`
  - Starts actual traversal given a TraversalDescription and starting node(s)
  - Returns an iterator over “steps” in the traversal
    - Steps can be: Path (default), Node, Relationship
  - The graph is actually traversed “lazily” (on request)

Example of Traversal

```
TraversalDescription desc =
  db.traversalDescription()
    .depthFirst()
    .relationships(Rels.KNOWS, Direction.BOTH)
    .evaluator(Evaluators.toDepth(3));

// node is 'Ed' (Node[2])
for (Node n : desc.traverse(node).nodes()) {
    output += n.getProperty("name") + ", ";
}
```




```
Output:  Ed, Lars, Lisa, Dirk, Peter,
```

Cypher Language

- Neo4j graph query language
  - For querying and updating
- Declarative – we say what we want
  - Not how to get it
  - Not necessary to express traversals
- Human-readable
- Inspired by SQL and SPARQL
- Still growing = syntax changes are often

Graph Database Summary

- Graph databases excel when objects are "indirectly" related to each other. Friends of friends, Cousins, your boss's boss's boss.

- Graph databases are suited for finding "structural patterns" in data.
  - If "X" buys "A", "B", "C" are they likely to buy "D"?

- When entites and their relationships are clustered

Next Time

- We finish up

- Alternate Final time:
  - You must have a documented conflict!
  - 9am on 12/9

- Remaining grading issues
  - See me next Tuesday