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
Graph Databases: Representatives

- Neo4j
- OrientDB
- TITAN
- Apache Giraph
- 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

❖ Variants:
  ▪ (Un)directed graphs
  ▪ Weighted graphs…

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 0 \\
3 & 1 & 1 & 0 & 0 & 1 & 0 \\
4 & 0 & 1 & 0 & 0 & 0 & 1 \\
5 & 0 & 1 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]
Adjacency Matrix: Properties

❖ 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 $j$
  - $\Rightarrow$ the adjacency list of $i$ contains node $j$ and vice versa

- Often compressed
  - Exploiting regularities in graphs

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]
Adjacency List: Properties

❖ 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

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]
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
In incidence matrix, the **Pros** are:

- Can represent **hypergraphs**
  - where one edge connects an arbitrary number of nodes

The **Cons** are:

- Requires $n \times m$ bits (for most graphs $m \gg n$)

### Example

<table>
<thead>
<tr>
<th></th>
<th>$E_1$</th>
<th>$E_2$</th>
<th>$E_3$</th>
<th>$E_4$</th>
<th>$E_5$</th>
<th>$E_6$</th>
<th>$E_7$</th>
<th>$E_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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

$$
\begin{array}{ccccccc}
1 & 2 & -1 & -1 & 0 & 0 & 0 \\
2 & -1 & 4 & -1 & -1 & -1 & 0 \\
3 & -1 & -1 & 3 & 0 & -1 & 0 \\
4 & 0 & -1 & 0 & 2 & 0 & -1 \\
5 & 0 & -1 & -1 & 0 & 2 & 0 \\
6 & 0 & 0 & 0 & -1 & 0 & 1 \\
\end{array}
$$
Laplacian Matrix: Properties

All features of adjacency matrix

❖ Pros:
  ▪ Analyzing the graph structure by means of spectral analysis
  • Calculating eigenvalues of the matrix

```
  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  
```
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

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
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
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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, …
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

\[ \text{G}_1 \]
\[ \text{G}_2 \]
\[ \text{G}_3 \]
\[ \text{G}_d \]
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

![Diagram of relationship properties]

source: Sadalage & Fowler: NoSQL Distilled, 2012
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true/false</td>
</tr>
<tr>
<td>byte</td>
<td>8-bit integer</td>
</tr>
<tr>
<td>short</td>
<td>16-bit integer</td>
</tr>
<tr>
<td>int</td>
<td>32-bit integer</td>
</tr>
<tr>
<td>long</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE 754 floating-point number</td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE 754 floating-point number</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unsigned integers representing Unicode characters</td>
</tr>
<tr>
<td>String</td>
<td>sequence of Unicode characters</td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>What</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>get who a person follows</td>
<td>outgoing <em>follows</em> relationships, depth one</td>
</tr>
<tr>
<td>get the followers of a person</td>
<td>incoming <em>follows</em> relationships, depth one</td>
</tr>
<tr>
<td>get who a person blocks</td>
<td>outgoing <em>blocks</em> relationships, depth one</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>get the full path of a file</td>
<td>incoming <em>file</em> relationships</td>
</tr>
<tr>
<td>get all paths for a file</td>
<td>incoming <em>file</em> and <em>symbolic link</em> relationships</td>
</tr>
<tr>
<td>get all files in a directory</td>
<td>outgoing <em>file</em> and <em>symbolic link</em> relationships, depth one</td>
</tr>
<tr>
<td>get all files in a directory, excluding symbolic links</td>
<td>outgoing <em>file</em> relationships, depth one</td>
</tr>
<tr>
<td>get all files in a directory, recursively</td>
<td>outgoing <em>file</em> and <em>symbolic link</em> relationships</td>
</tr>
</tbody>
</table>

---

### Diagram

```
/  
  |   
  A   B
    |   
    |   
    C   D
      |   
      |   
      E
```

- A: directory
- B: file
- C: directory
- D: directory
- E: symbolic link named "F"

---

```
/  
  |   
  A   B
    |   
    |   
    C   D
      |   
      |   
      E
```

- A: directory
- B: file
- C: directory
- D: directory
- E: symbolic link named "F"
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
Traversals 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`
Traversing 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()
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(ReIs.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