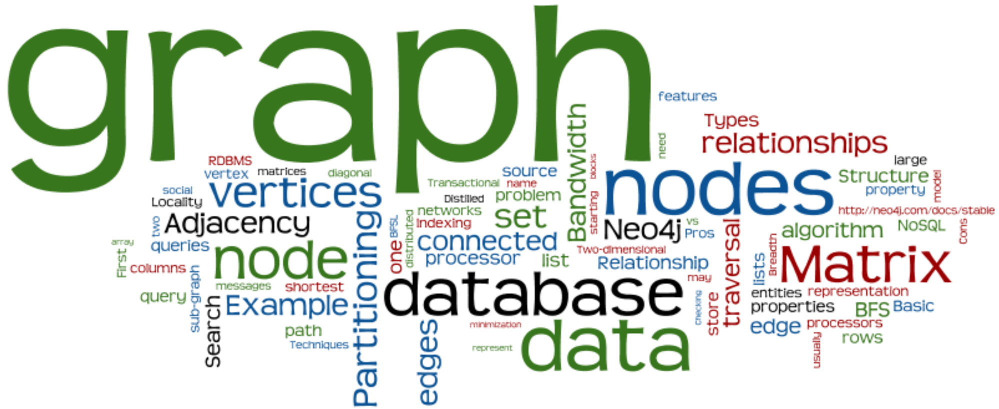
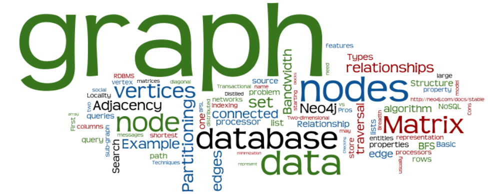


Graph Databases

Lecture 8 of *NoSQL Databases* (PA195)

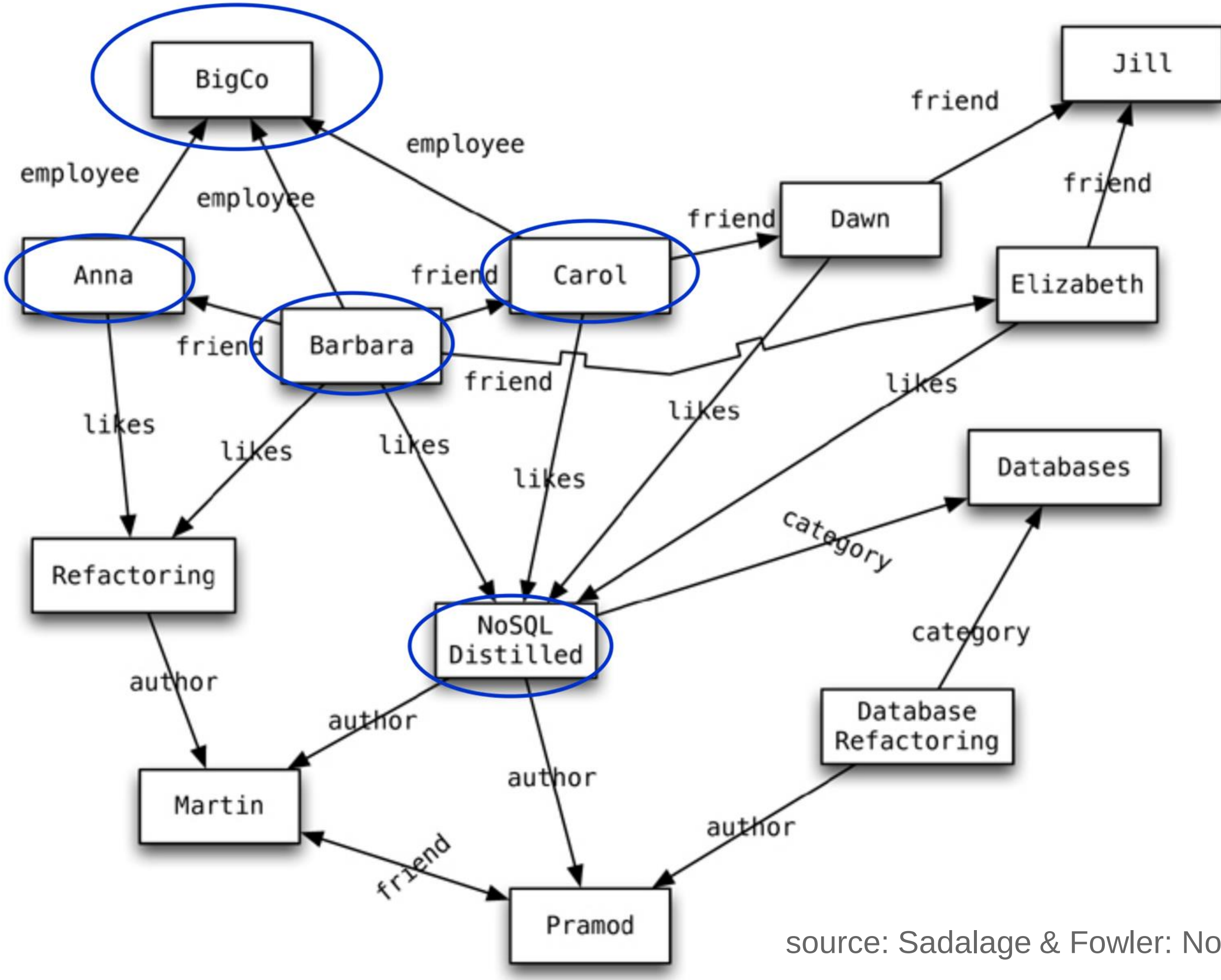
David Novak & Vlastislav Dohnal
Faculty of Informatics, Masaryk University, Brno



Agenda

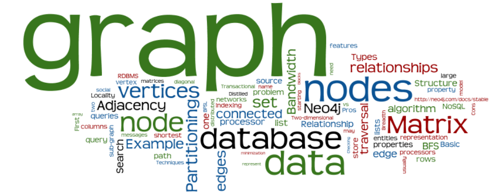
- Graph Databases: Mission, Data, Example
- A Bit of Graph Theory
  - Graph Representations
  - Algorithms: Improving Data Locality (efficient storage)
  - Graph Partitioning and Traversal Algorithms
- Graph Databases
  - Transactional databases
  - Non-transactional databases
- Neo4j
  - Basics, Native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

Graph Databases: Example



source: Sadalage & Fowler: NoSQL Distilled, 2012

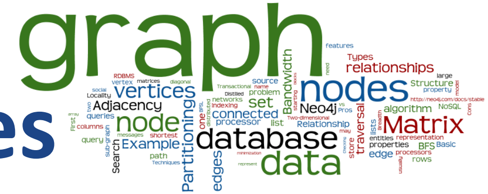
Graph Databases: Mission



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

- Nodes are organized by relationships
  - Allows finding interesting patterns
  - **Example:** Get all nodes that are “employee” of “Big Company” and that “likes” “NoSQL Distilled”

Graph Databases: Representatives



Neo4j

OrientDB

TigerGraph

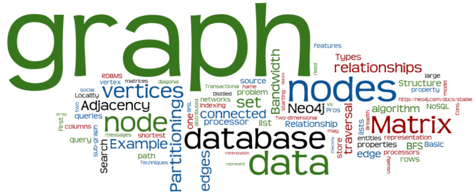
JanusGraph

APACHE GIRAPH

Azure Cosmos DB

ArangoDB

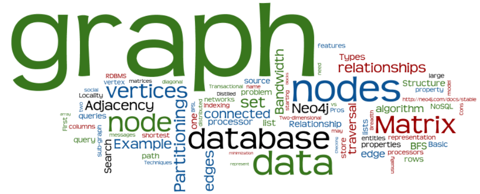
Ranked list: http://db-engines.com/en/ranking/graph+dbms



A Bit of a Theory

Basics and graph representations

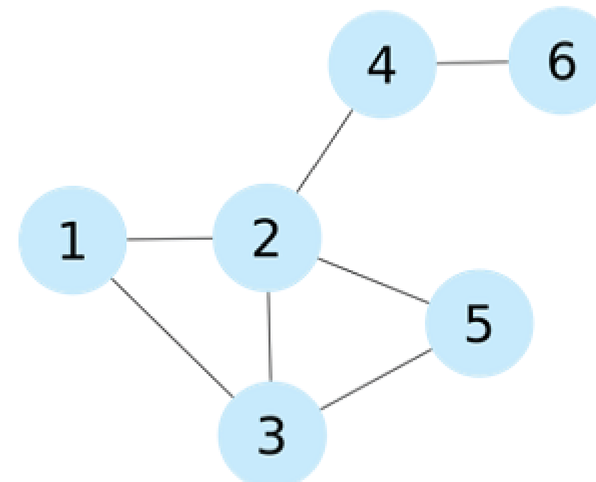
Basic Terminology



- Data: a set of entities and their relationships
  - => we need to efficiently represent graphs
- Basic operations:
  - finding the neighbors 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 (directed) edges $E = (V_1,V_2)$, $|E| = m$
- Which data structure to use?

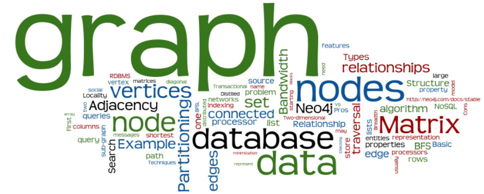
Data Structure: Adjacency Matrix

- 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

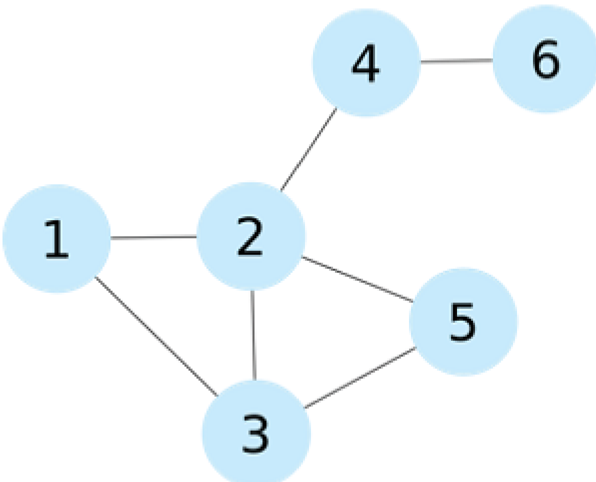


$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

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



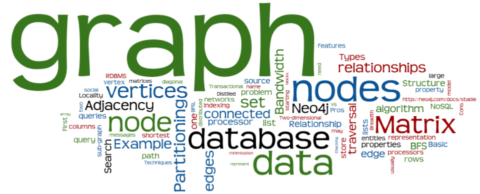
Adjacency Matrix: Properties



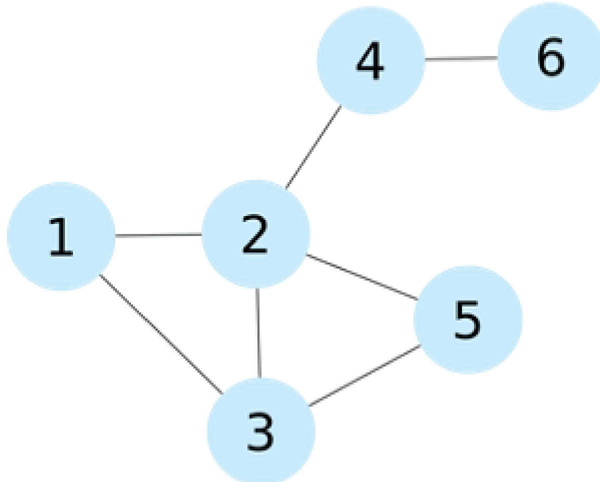
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- Pros:
  - Adding/removing edges
  - Checking if 2 nodes are connected
- Cons:
  - Quadratic space: $O(n^2)$
  - We usually have sparse graphs
  - Adding nodes is expensive
  - Retrieval of all the neighboring nodes takes linear time: $O(n)$

Data Structure: Adjacency List



- A set of lists, each enumerating neighbors of one 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
- Often compressed
  - Exploiting regularities in graphs



N1 → {N2, N3}

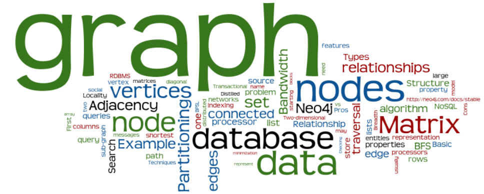
N2 → {N1, N3, N5}

N3 → {N1, N2, N5}

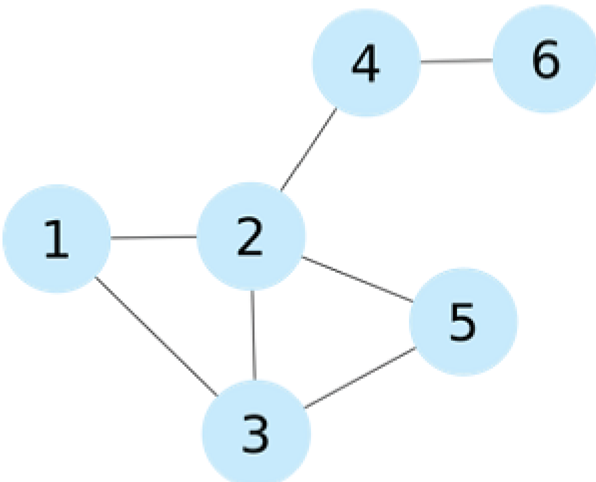
N4 → {N2, N6}

N5 → {N2, N3}

N6 → {N4}



Adjacency List: Properties



N1 → {N2, N3}

N2 → {N1, N3, N5}

N3 → {N1, N2, N5}

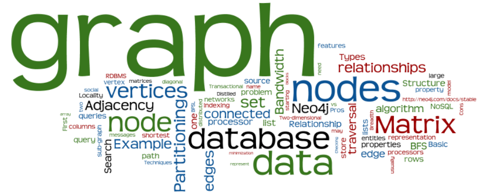
N4 → {N2, N6}

N5 → {N2, N3}

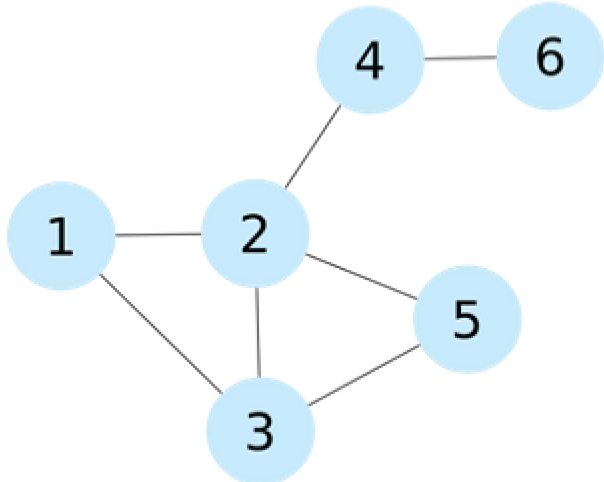
N6 → {N4}

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

Data Structure: Incidence Matrix

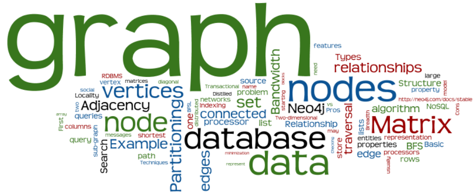


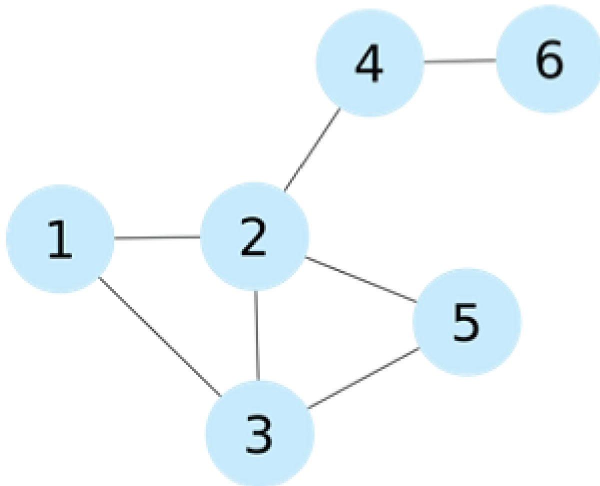
- Two-dimensional Boolean matrix of *n* rows and *m* columns
  - Each row represents a node
    - All edges that are connected to the node
  - Each column represents an edge
    - Nodes that are connected by a certain edge



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Incidence Matrix: Properties

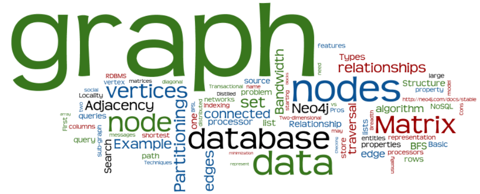




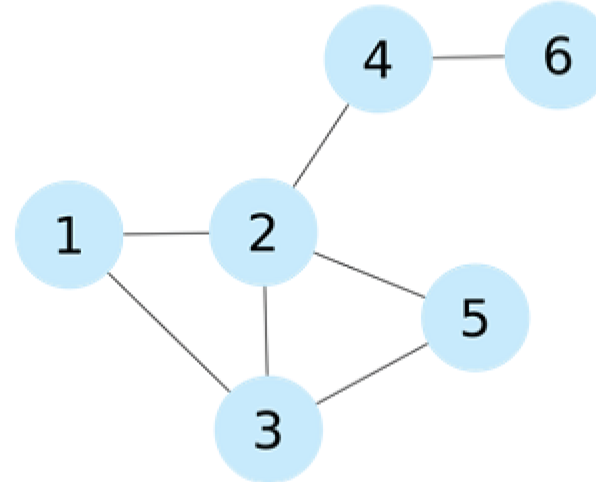
$$
\begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

- Pros:
  - Representation of hypergraphs
    - where one edge connects an arbitrary number of nodes
- Cons:
  - Requires $n \times m$ bits (for most graphs $m \ggg n$)
  - Listing neighborhood is slow

Data Structure: Laplacian Matrix

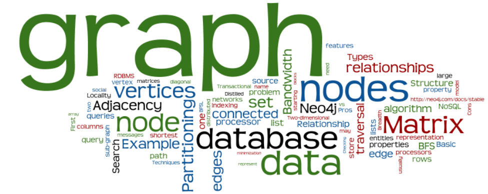


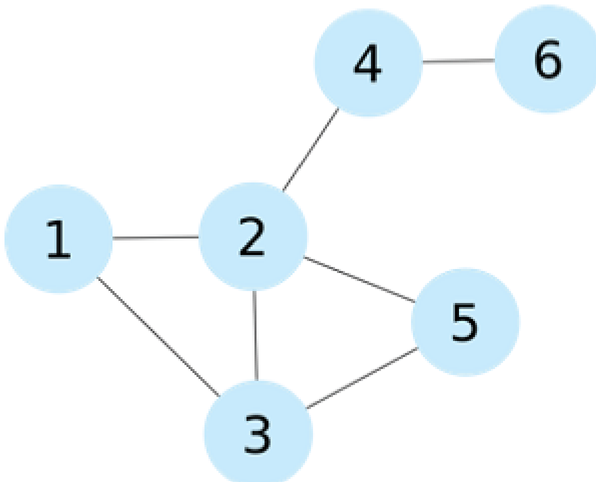
- Two-dimensional array of $n \times n$ integers
  - Similar structure as adjacency matrix
  - Diagonal of the Laplacian matrix indicates the degree of the node
  - The rest of positions are set to *-1* if the two vertices are connected, *0* otherwise



$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

Laplacian Matrix: Properties

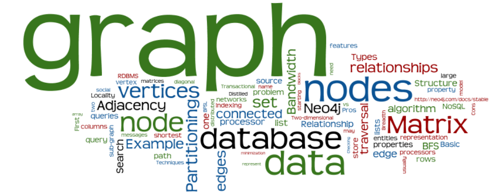




$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

All features of adjacency matrix

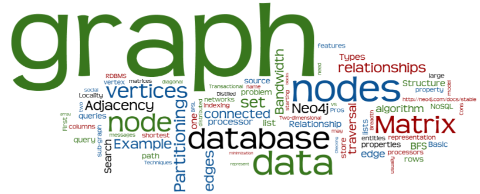
- Pros:
  - Analyzing the graph structure by means of spectral analysis
    - Calculating the number of spanning trees
    - Approximation of the sparsest cut of the graph
    - Calculate eigenvalues of the matrix
  - A good summary: Wikipedia



A Bit of a Theory

Selected graph algorithms

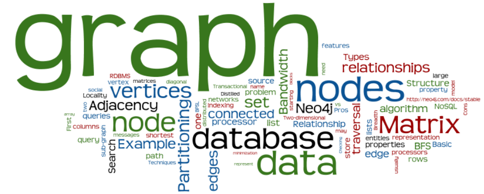
Basic Graph Algorithms



- **Access** all nodes reachable from a given source:
  - 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

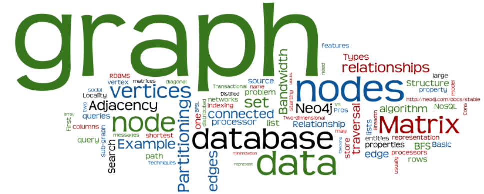
http://en.wikipedia.org/wiki/Shortest_path_problem

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, ...

Data Locality: Example

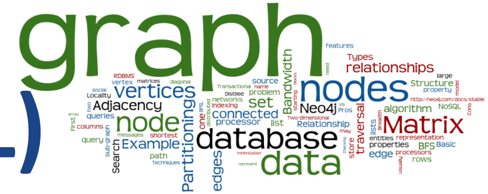


$$\begin{pmatrix} 1&0&0&0&1&0&0&0\\ 0&1&1&0&0&1&0&1\\ 0&1&1&0&1&0&0&0\\ 0&0&0&1&0&0&1&0\\ 1&0&1&0&1&0&0&0\\ 0&1&0&0&0&1&0&1\\ 0&0&0&1&0&0&1&0\\ 0&1&0&0&0&1&0&1 \end{pmatrix}$$

$$\begin{pmatrix} 1&1&0&0&0&0&0&0\\ 1&1&0&0&0&0&0&0\\ 0&0&1&1&1&0&0&0\\ 0&0&1&1&1&0&0&0\\ 0&0&1&1&1&1&0&0\\ 0&0&0&0&1&1&1&0\\ 0&0&0&0&0&1&1&1\\ 0&0&0&0&0&0&1&1 \end{pmatrix}$$

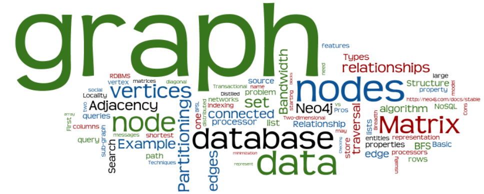
This matrix has better data locality, more efficient traversal

Breadth First Search Layout (BFSL)



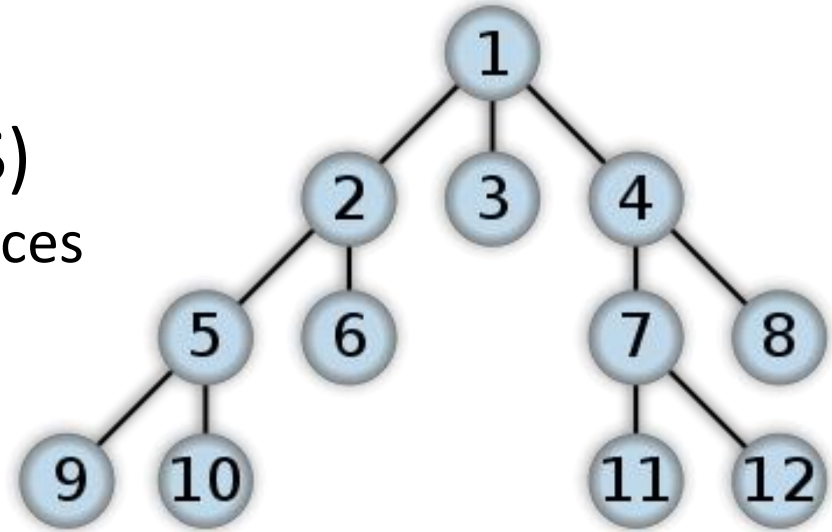
- Input: vertices of a graph
- Output: a permutation of the vertices
  - with better cache performance for graph traversals

- 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



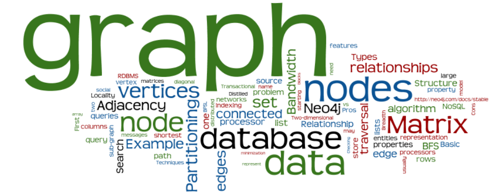
Breadth First Search Layout (2)

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

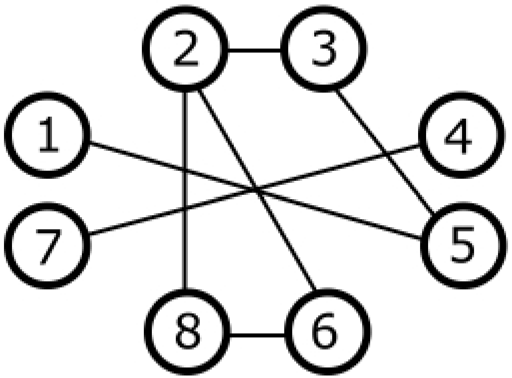


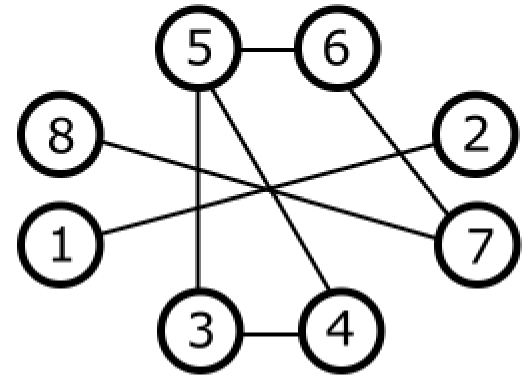
- Pros: optimal locality for traversal from the root
- Cons: starting traversal from other nodes
  - The further, the worse

Matrix Bandwidth: Motivation



- Graph represented by adjacency matrix

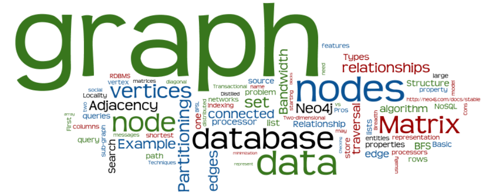




$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

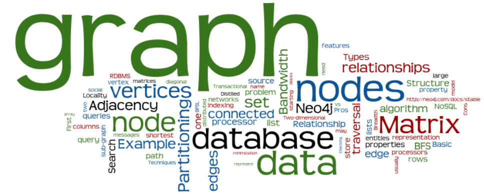
Matrix Bandwidth: Formalization



- 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

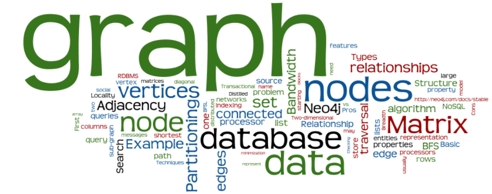
- Bandwidth minimization problem: NP hard
  - For large matrices the solutions are only approximated



A Bit of a Theory

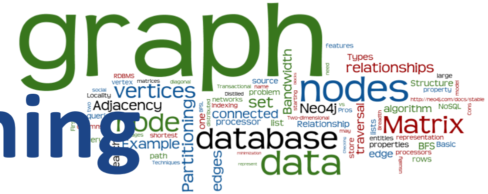
Graph partitioning

Graph Partitioning



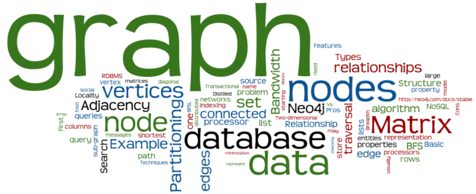
- Some graphs are too large to be fully loaded into the main memory of a single computer
  - Usage of secondary storage degrades the performance
  - Scalable solution: distribute the graph on multiple nodes
- We need to partition the graph reasonably
  - Usually for a particular (set of) operation(s)
    - The shortest path, finding frequent patterns, BFS, spanning tree search

Example: 1-Dimensional Partitioning



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

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

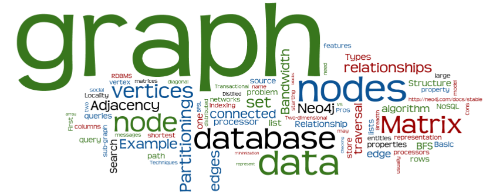


	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	0	0	0	0	0	0	1	1	0
2	0	0	1	0	0	0	0	1	0	0	0	0
3	0	1	0	0	0	0	1	1	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	1
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0	0	0	1	0
7	0	0	1	0	0	1	0	1	0	1	1	0
8	0	1	1	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1
10	1	0	0	0	0	0	1	0	0	0	1	0
11	1	0	0	0	0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

Starting BFS traversal at node 1:

1. (at black) 1 -> 10, 11
   visit green server
2. (at green) 10, 11 ->
   a. 1, back to black
   b. 6, visit red
   c. 7,9, visit blue
   d. 10, 11, myself
3. (at red) 6 -> 7
   visit blue
3. (at blue) 7,9 ->
   a. 3, back to black …
   b. 6, back to red
   c. 8 -> 2,3, back to black
   d. 10,11,12, back to green

Traversing Graph

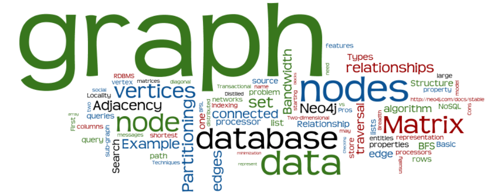


- Traversing with 1D partitioning (e.g., BFS)
  1. Each **processor** keeps information about frontier vertices
  2. ...and also list of neighboring vertices in other processors
  3. Messages are sent to other processors...

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

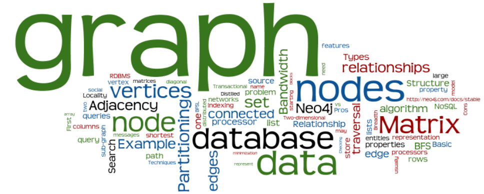
Efficient sharding of a graph is very difficult

- and thus graph DBs are often centralized



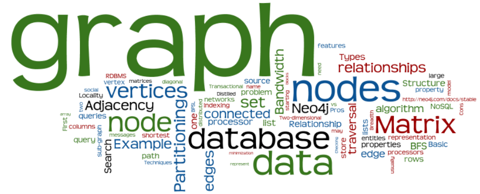
Graph Databases

Types of Graphs



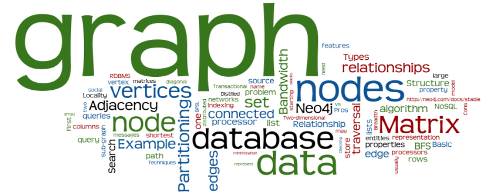
- Single-relational graphs
  - Edges are homogeneous in meaning
    - e.g., all edges represent friendship
- Multi-relational (property) graphs
  - Edges are typed or labeled
    - e.g., friendship, business, communication
  - Vertices and edges maintain a set of key/value pairs
    - Representation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

Graph Databases



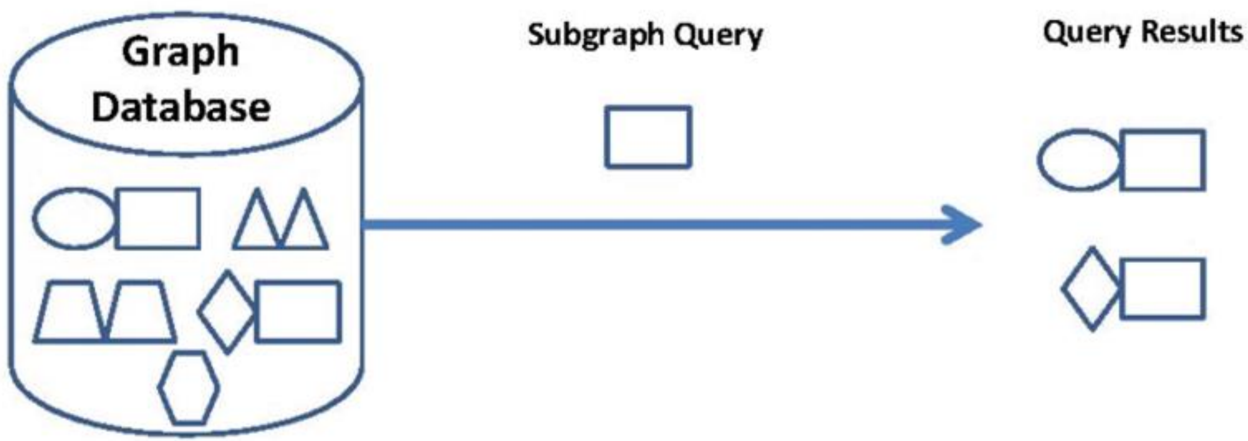
- A graph database = a set of graphs

- Types of graph databases:
    - Transactional = a 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 necessarily connected) graph
        - e.g., Web graph, social networks, …



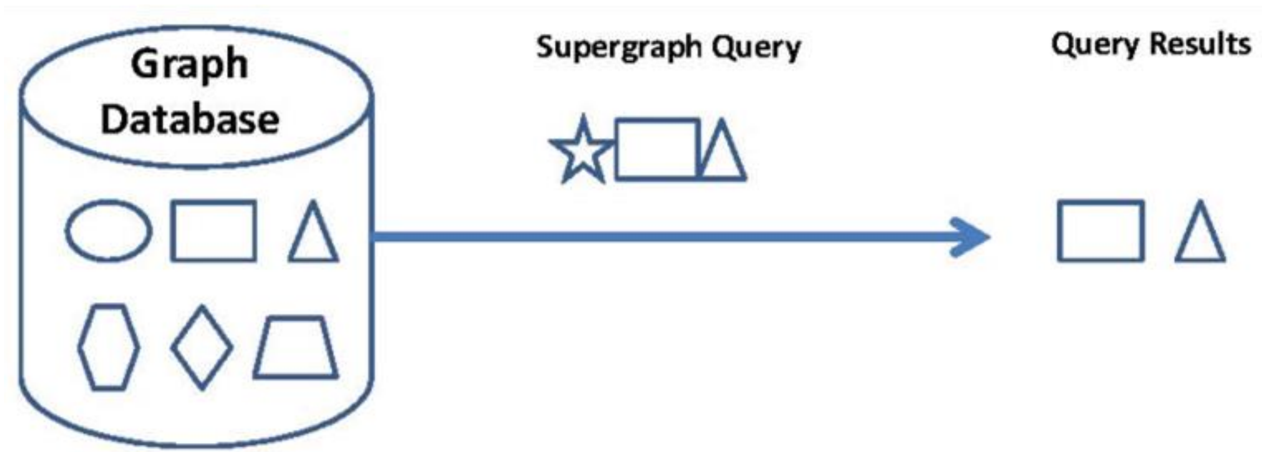
Transactional DBs: Queries

- Types of Queries
  - Subgraph queries
    - Searches 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



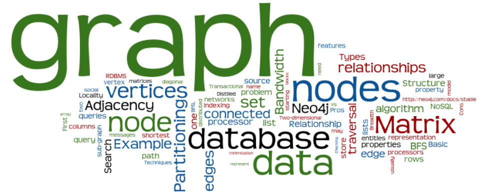
Transactional DBs: Queries (2)

- Super-graph queries
  - Search for graphs whose whole structure is contained in the query graph



- 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

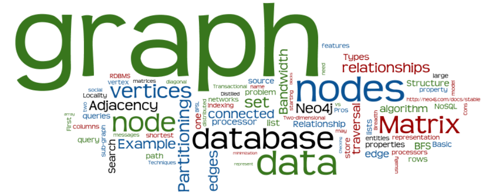
Indexing & Query Evaluation



- 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



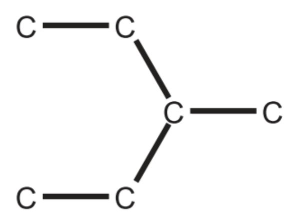
1. Mining-based Graph Indexing Techniques
    - Idea: if some features of query graph *q* do not exist in data graph *G*, then *G* cannot contain *q* as its subgraph
    - Apply graph-mining methods to extract some features (sub-structures) from the graph database members
        - e.g., frequent sub-trees,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

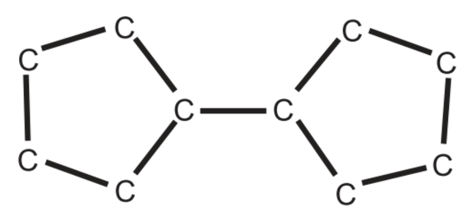
Mining-based Technique

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

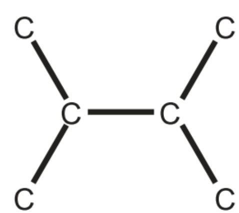
G_1



G_2



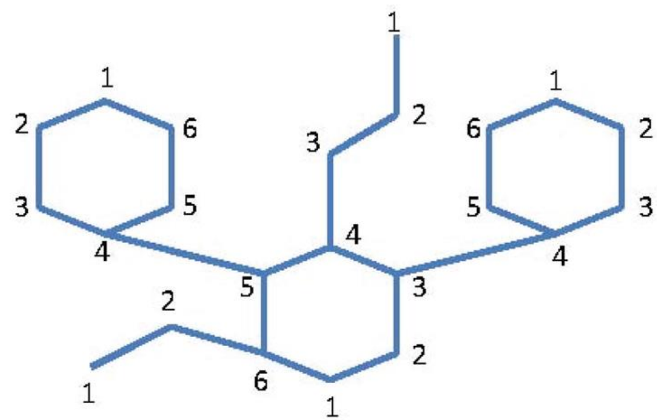
G_3

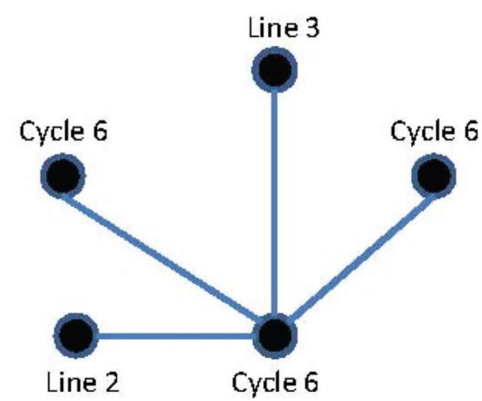


G_d

Non Mining-based Techniques

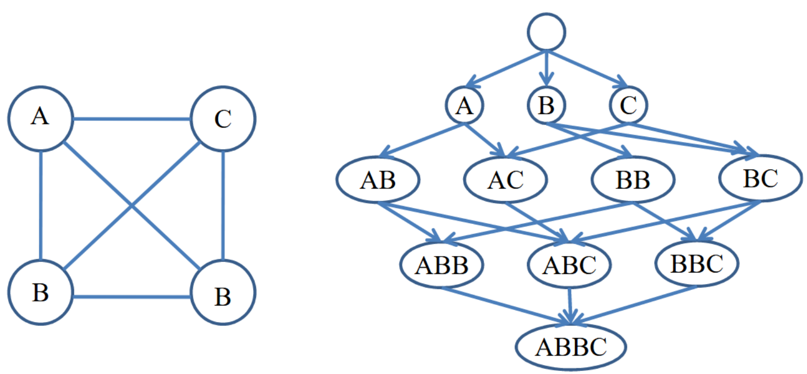
- **Example:** GString (2007)
  - Model the graphs in the context of organic chemistry using basic structures
    - Line = series of vertices connected end to end
    - Cycle = series of vertices that form a close loop
    - Star = core vertex directly connects to several vertices

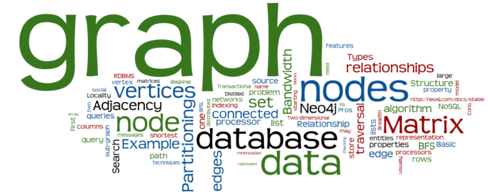




Non Mining-based Techniques

- GDIndex (2007)
  - all connected and induced subgraphs of a given graph are enumerated (at most 2^n)
  - due to isomorfisms, there much less subgraphs.
    - if all labels are identical, a complete graph of size n is decomposed into just n+1 subgraphs.

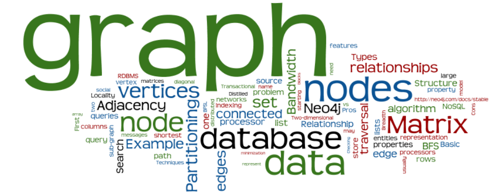




Graph Databases

Non-transactional Databases

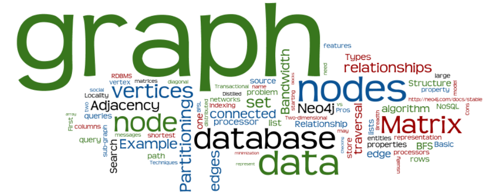
Non-transactional Databases



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

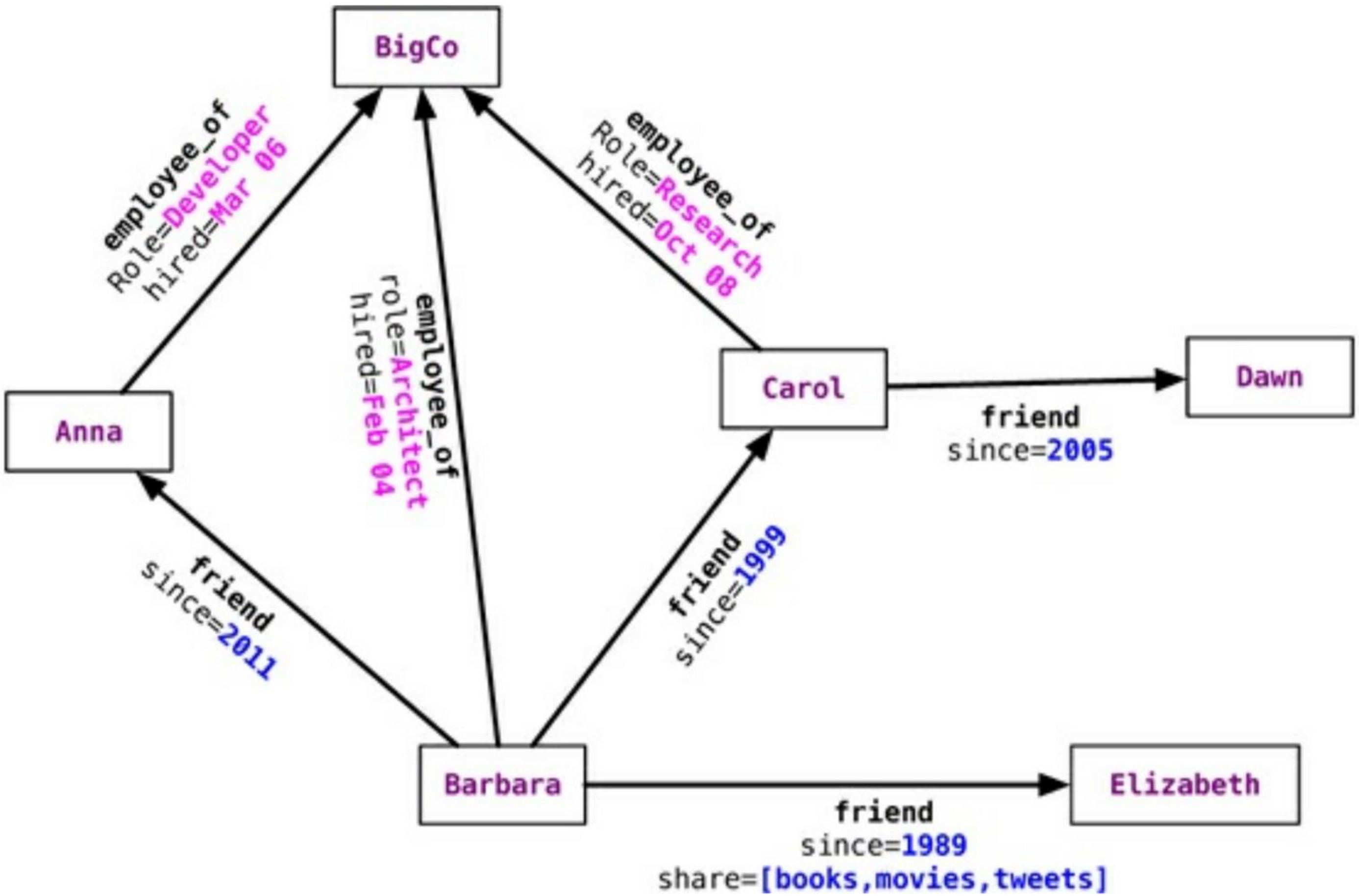
- Our example: Neo4j

Basic Characteristics



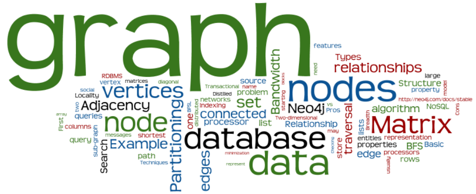
- Different types of relationships between nodes
  - To represent relationships between domain entities
  - Or to model any kind of secondary relationships
    - Category, path, time-trees, spatial relationships, …
- No limit to the number and kind of relationships
- Relationships have properties
  - E.g., since when did they become friends?

Relationship Properties: Example

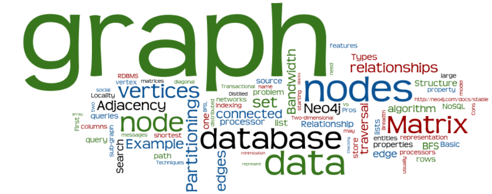


source: Sadalage & Fowler: NoSQL Distilled, 2012

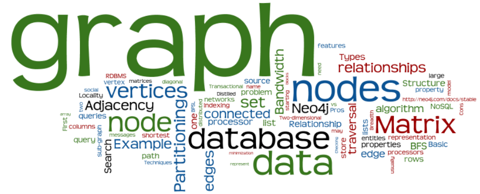
Graph DB vs. RDBMS



- RDBMS designed for a single type of relationship
  - “Who is my manager”
- Adding another relationship usually means a lot of schema changes
- 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

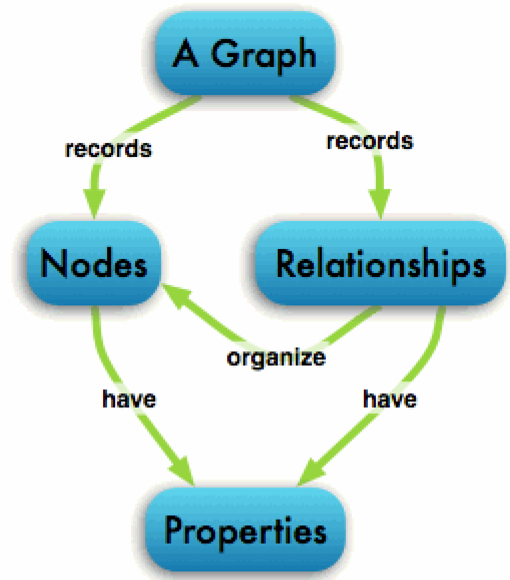


Neo4J: Basics & Concepts

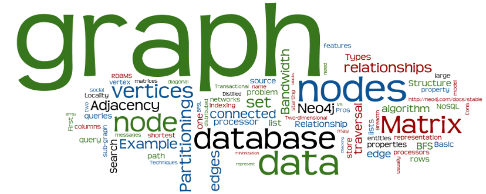


Neo4j: Basic Info

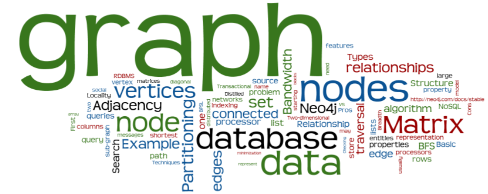
- Open source graph database
    - The most popular
- Initial release: 2007
- Written in: Java
- OS: cross-platform
- Stores data as nodes connected by directed, typed relationships
    - With properties on both nodes and relationships



Neo4j: Basic Features

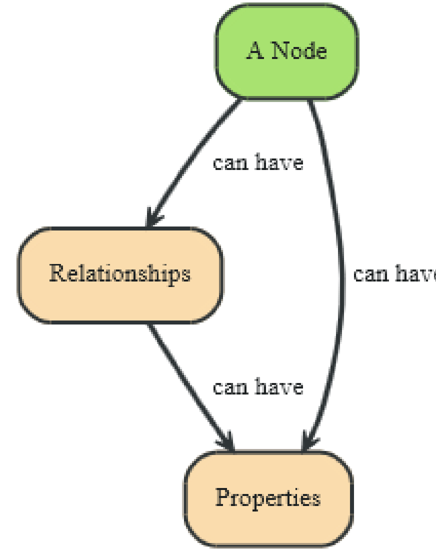


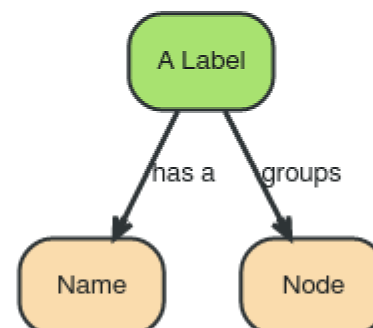
- reliable – with full ACID transactions
- durable and fast – disk-based, native storage engine
- scalable – up to several billion nodes/relationships/properties
- highly-available – when distributed (replicated)
- expressive – powerful, human readable graph query language
- fast – powerful traversal framework
- embeddable - in Java program
- accessible – simple REST interface & Java API



Data Model: Nodes

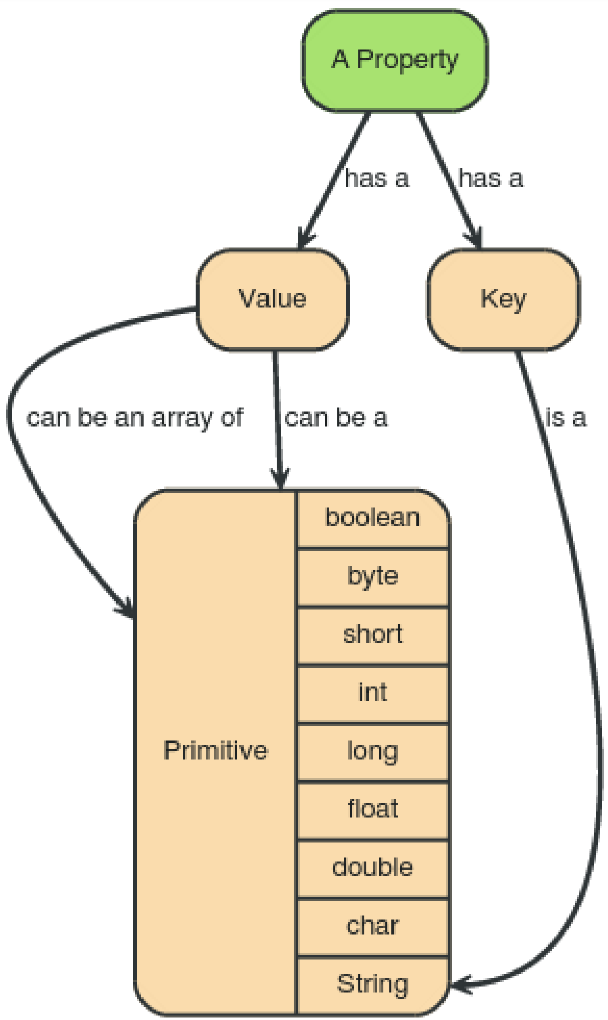
- Fundamental unit: node
- Nodes have properties
  - Key-value pairs
  - null is not a valid property value
    - nulls can be modelled by the absence of a key
- Nodes have labels
  - labels typically express "type of node"





http://db-engines.com/en/system/Neo4j

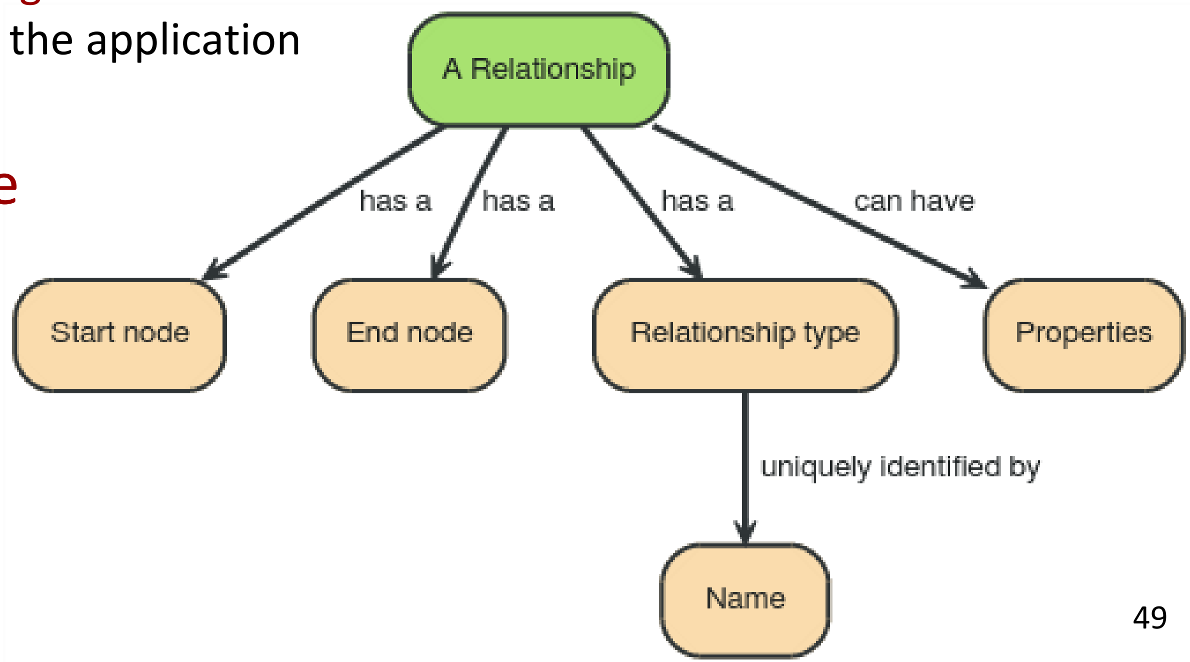
Data Model: Properties

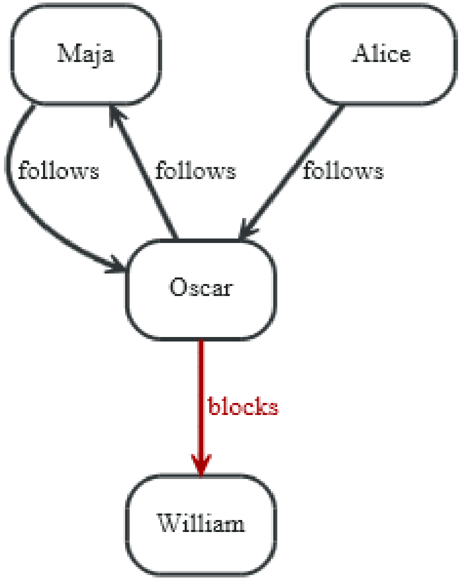


Type	Description
boolean	true/false
byte	8-bit integer
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit IEEE 754 floating-point number
double	64-bit IEEE 754 floating-point number
char	16-bit unsigned integer representing a Unicode character
String	sequence of Unicode characters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

Data Model: Relationships

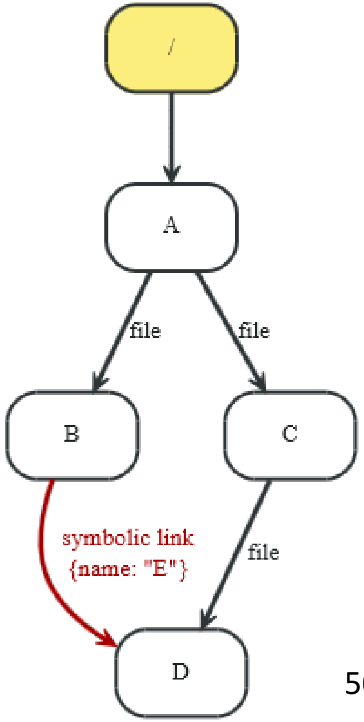
- Directed relationships (edges)
  - Incoming and outgoing edge
    - Equally efficient traversal in both directions
    - Direction can be ignored if not needed by the application
  - Always a start and an end node
    - Can be recursive

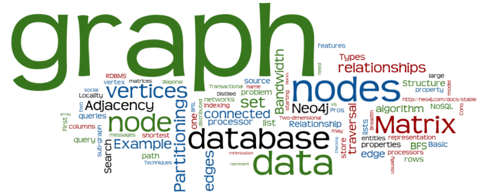




What	How
get who a person follows	outgoing *follows* relationships, depth one
get the followers of a person	incoming *follows* relationships, depth one
get who a person blocks	outgoing *blocks* relationships, depth one

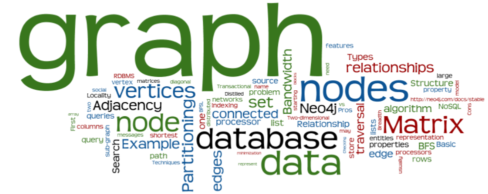
What	How
get the full path of a file	incoming *file* relationships
get all paths for a file	incoming *file* and *symbolic link* relationships
get all files in a directory	outgoing *file* and *symbolic link* relationships, depth one
get all files in a directory, excluding symbolic links	outgoing *file*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





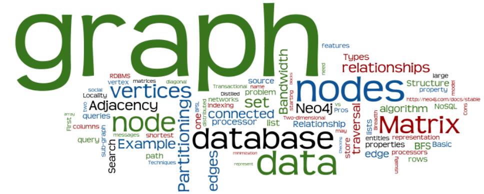
Access to Neo4j

- 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.



Neo4J: Native Java API & Graph Traversal

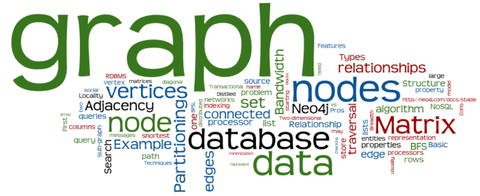
Native Java Interface: Example




```
Node irena = graphDb.createNode();
irena.setProperty("name", "Irena");
Node jirka = graphDb.createNode();
jirka.setProperty("name", "Jirka");

Relationship i2j = irena.createRelationshipTo(jirka, FRIEND);
Relationship j2i = jirka.createRelationshipTo(irena, FRIEND);

i2j.setProperty("quality", "a good one");

j2i.setProperty("since", 2003);
```

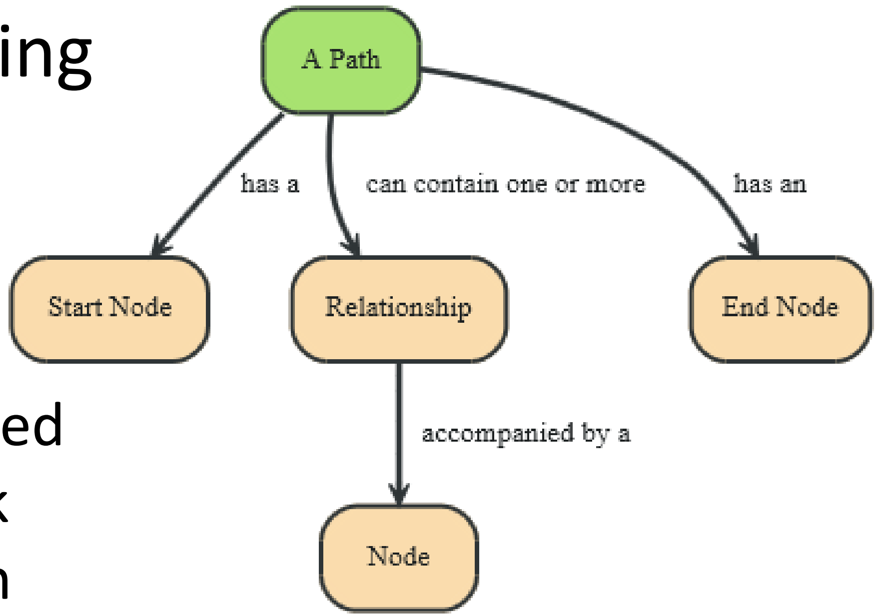
- Undirected edge:
  - Relationship between the nodes in both directions
  - **INCOMING** and **OUTGOING** relationships from a node

Data Model: Path & Traversal



- Path = specific nodes + connecting relationships
  - Path can be 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 in Java API, Cypher, Gremlin

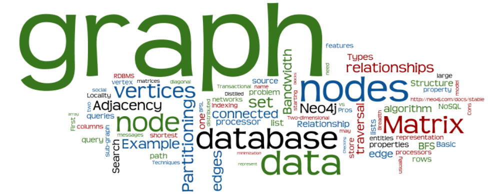


Traversal Framework

- A traversal is influenced by
  - Starting node(s) where the traversal begins
  - 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 beyond 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()`
  - Specify the relationship types 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)

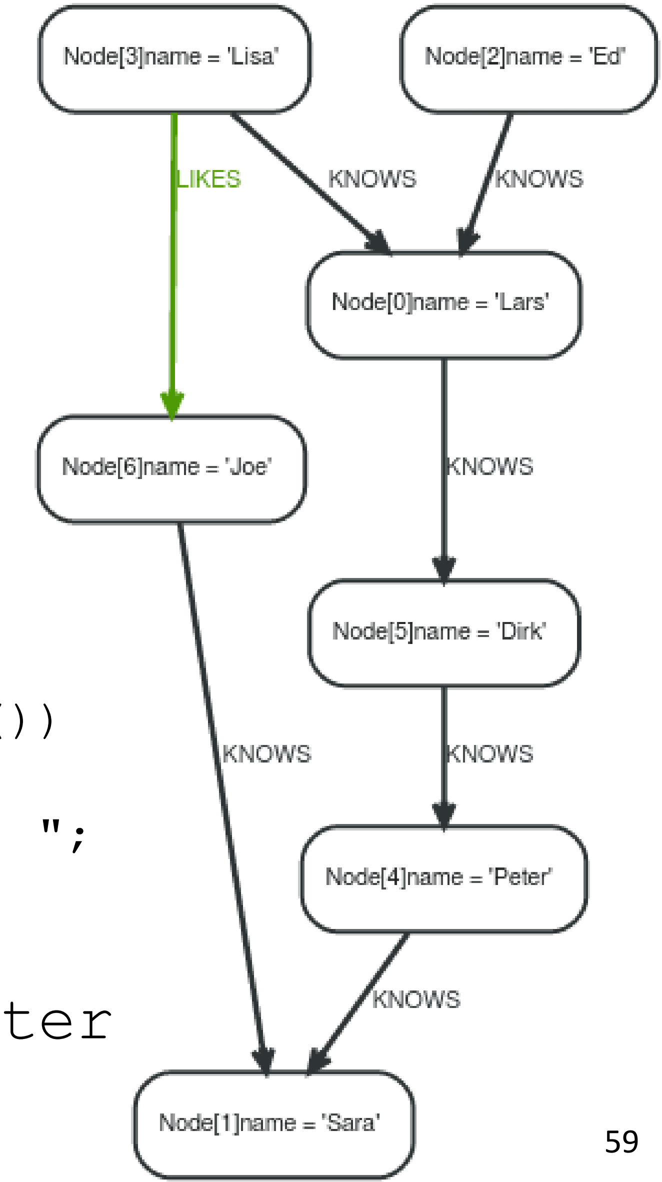
- `org.neo4j...Uniqueness`
  - Indicates under what circumstances a traversal may revisit the same position in the graph

- `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)

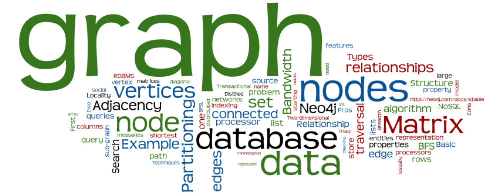
Example of Traversal

```
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`

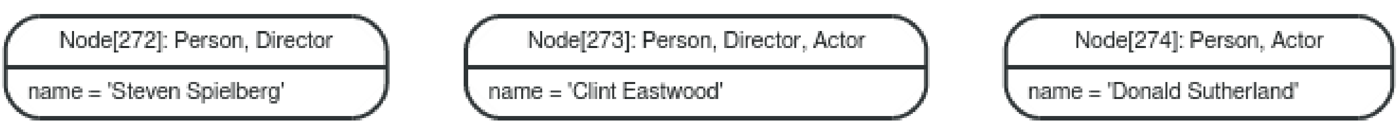


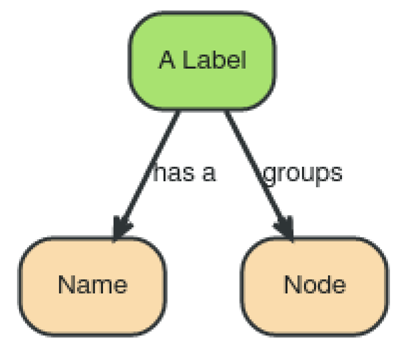
http://neo4j.com/docs/stable/tutorial-traversal-java-api.html

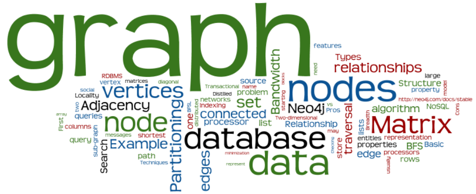


Access to Nodes

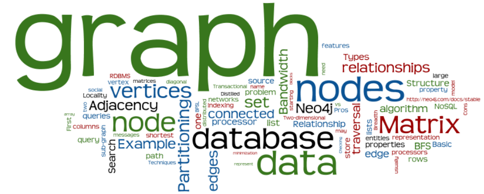
- How to get to the starting node(s) before traversal
  1. Using internal identifiers (generated IDs)
     - not recommended - Neo4j generates IDs for memory objs and reuses IDs
  2. Using properties of nodes
     - one of the properties is typically “ID” (user-specified ID)
     - recommended, properties can be indexed
       - automatic indexes
  3. Using “labels”
     - group nodes into “subsets” (named graph)
     - a node can have more than one label
       - belong to more subsets







Neo4J: Cypher Language

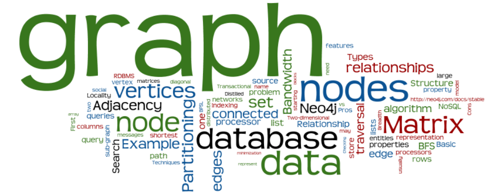


Cypher Language

- 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

http://neo4j.com/docs/stable/cypher-query-lang.html

Cypher: Clauses



- **MATCH**: The graph pattern to match
- **WHERE**: Filtering criteria
- **RETURN**: What to return
- **WITH**: Divides a query into multiple parts
  - can define starting points in the graph
- **CREATE**: Creates nodes and relationships.
- **DELETE**: Remove nodes, relationships, properties
- **SET**: Set property values

Cypher: Creating Nodes (Examples)

CREATE (n);

*(create a node, assign to var **n**)*

```
Created 1 node, returned 0 rows
```

CREATE (a: Person {name : 'David'})
RETURN a;

(create a node with label 'Person' and 'name' property 'David')

```
Created 1 node, set 1 property, returned
1 row
```

Cypher: Creating Relationships

MATCH (a {name:'John'}), (b {name:'Jack'})
CREATE a-[**r**:Friend]->b
RETURN r ;

(create a relation Friend *between John and Jack)*

```
Created 1 relationship, returned 1 row
```

MATCH (a {name:'John'}), (b {name:'Jack'})
CREATE p = a-[:Friend {name: a.name + '->' + b.name }]->b
RETURN p

(set property 'name' of the relationship)

```
Created 0 nodes, set 1 property, returned 1 row
```

Cypher: Queries

MATCH (p: Person)
WHERE p.age >= 18 **AND** p.age < 30
RETURN p.name

(return names of all adult people under 30)

MATCH (user: Person {name: 'Andres'})-[:Friend]->(follower)
RETURN user.name, follower.name

(find all ‘Friends’ of 'Andres')

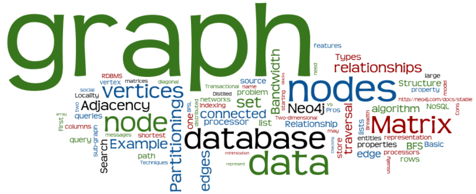
Cypher: Queries (2)

```
MATCH (andres: Person {name: 'Andres'})-[*1..3]-(node)
RETURN andres, node ;
```

(find all 'nodes' within three hops from 'Andres')

```
MATCH p=shortestPath(
  (andres:Person {name: 'Andres'})-[*]-(david {name:'David'})
)
RETURN p ;
```

(find the shortest connection between 'Andres' and 'David')



Neo4J: Behind the Scene

Neo4j Internals: Indexes

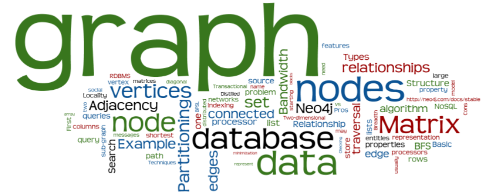
```
CREATE INDEX ON :Person(name);
```

(Create index on property name of nodes with label Person)

```
Indexes added: 1
```

- Since Neo4j v. 2, indexes are used automatically
  - Can be specified explicitly (which index to use)

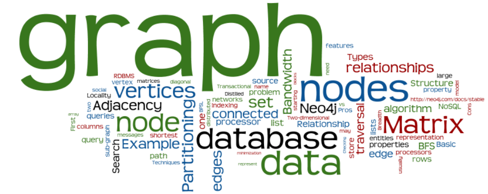
```
MATCH (n:Person)
USING INDEX n:Person(surname)
WHERE n.surname = 'Taylor'
RETURN n
```



Neo4j Internals: Transactions

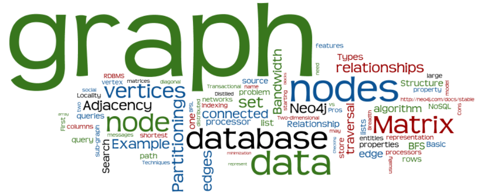
- Transactions in Neo4j
  - Support for ACID properties
  - All write operations must be performed in a transaction
  - Default transaction isolation level: Read committed
    - Operation can see the last committed value
    - Reads do not block or take any locks
    - If the same row is retrieved twice within a transaction, the values in the row CAN differ
  - Higher level of isolation can be achieved
    - By explicit acquiring the read locks

Neo4j Internals: High Availability



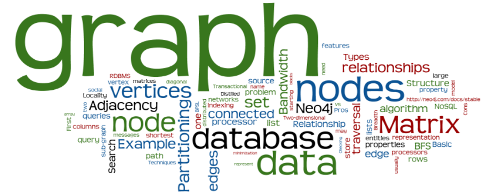
- Master-slave replication
  - Several Neo4j slave databases can be configured to be exact replicas of a single Neo4j master database

- Speed-up of read operations
  - Enables to handle more read load than a single node
- Fault-tolerance
  - In case a node becomes unavailable
- Transactions are still atomic, consistent and durable, but eventually propagated to the slaves



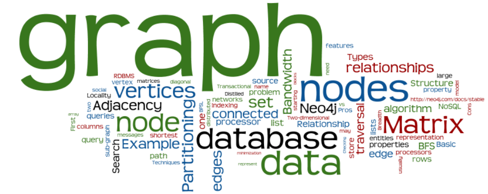
Graph Databases: When (not) to Use

Graph DBs: Suitable Use Cases



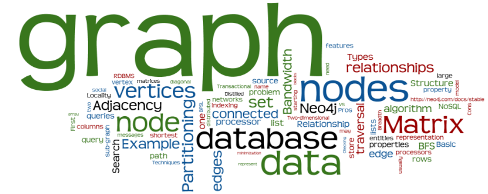
- Connected Data
  - Social networks
  - Any link-rich domain is well suited for graph databases
- Routing, Dispatch, and Location-Based Services
  - Node = location or address that has a delivery
  - Graph = nodes where a delivery has to be made
  - Relationships = distance
- Recommendation Engines
  - “your friends also bought this product”
  - “when buying this item, these others are usually bought”

Graph DBs: Modeling Issues



- Node modeling:
  - tradeoff between placing all attributes and properties in a single node,
  - and separating each attribute into an individual node.
- Relationship modeling:
  - "unlabeled" all,
    - e.g., person **connected_to** person/address/product
  - versus semantic meaning encoded labels
    - e.g., person **peters_work_colleague** person, person **peters_home_address** address

Graph DBs: When Not to Use

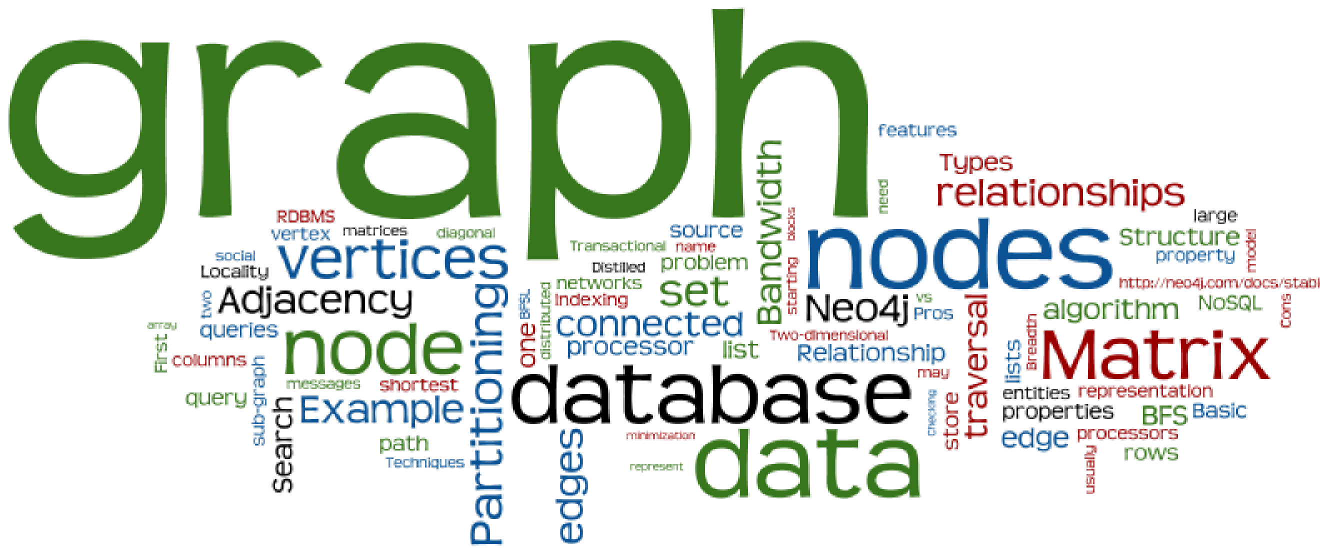


- If we want to update all or a subset of entities
  - Changing a property on many nodes is not straightforward
    - e.g., analytics solution where all entities may need to be updated with a changed property

- No BLOBs (large binary objects) in byte arrays.

- Some graph databases may be unable to handle lots of data
  - Distribution of a graph is difficult

Questions?


graph
nodes
database
data
Matrix
node
vertices
relationships
Partitioning
edges
Adjacency
Example
connected
Neo4j
set
Bandwidth
traversal
algorithm
Relationship
processor
Types
features
Structure
property
large
http://neo4j.com/docs/stable
NoSQL
representation
entities
properties
BFS
Basic
edge
processors
rows
store
lists
Breadth
Pros
vs
Two-dimensional
list
indexing
networks
problem
source
Transactional
name
Distilled
one
BFSL
distributed
RDBMS
vertex
matrices
diagonal
social
Locality
two
queries
First
columns
query
sub-graph
Search
messages
shortest
path
Techniques
minimization
represent
starting
blocks
need
model
Cons
may
checking
usually
array

References

- I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015. 288 p.
- RNDr. Irena Holubova, Ph.D. MMF UK course NDBI040: Big Data Management and NoSQL Databases
- Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications
- Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 192 p.

- http://neo4j.com/docs/stable/