Non-Standard Database Systems

Graph Databases

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
FB Computerwissenschaften
Universität Salzburg

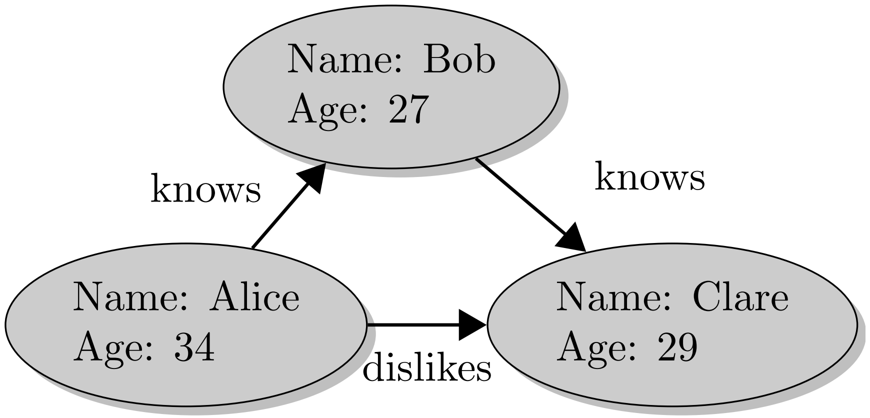
database research group
http://dbresearch.uni-salzburg.at

Sommersemester 2016

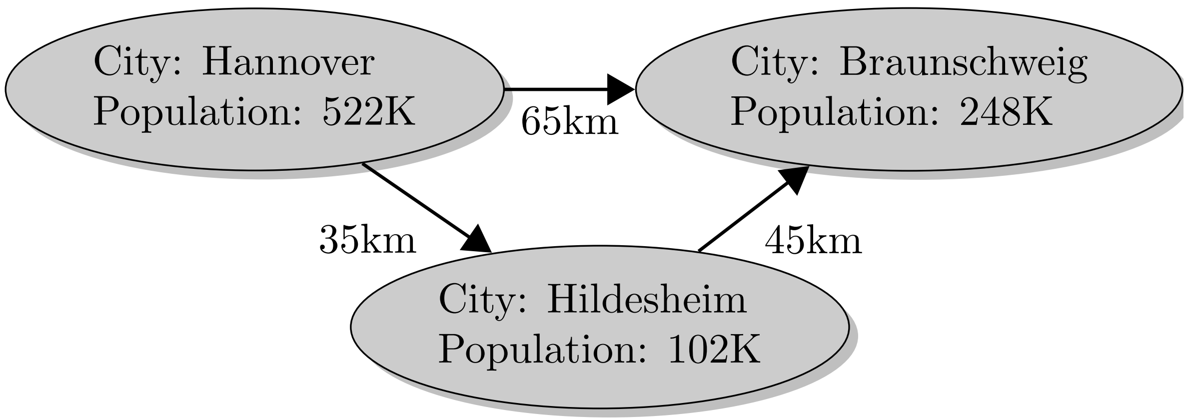
Version 9. Juni 2016

Inhalt

Storing Data in Graphs – Examples



© Lena Wiese: Advanced Data Management, DeGruyter, 2015.



© Lena Wiese: Advanced Data Management, DeGruyter, 2015.

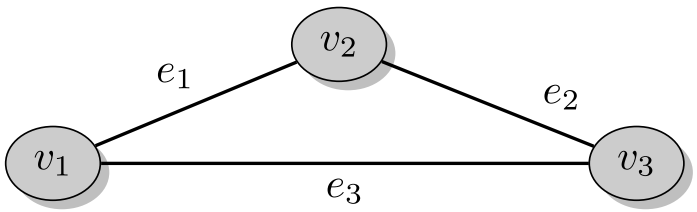
Graph Terms

- graph $G = (V, E)$
- V : set of nodes (node = vertex)
- E : set of edges
- adjacent nodes (=neighbors) are connected with an edge
- an edge is incident to a node if it is connected to the node

Different Types of Graphs

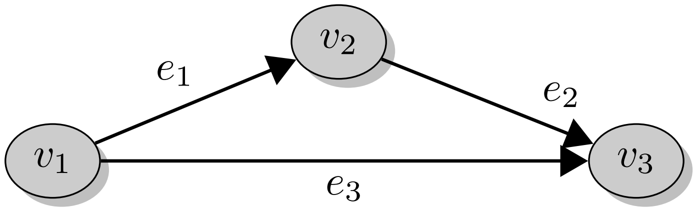
- simple undirected graph
- simple directed graph
- undirectred multi-graph
- directed multi-graph
- weighted graphs

Simple Undirected Graphs



- edges are (unordered) two-element subsets of V, e.g., $\{v_1, v_3\} = \{v_3, v_1\} \in E$
- complete graph: maximum of $\frac{n(n-1)}{2}$ edges for $n = |V|$ nodes (without self-loops)

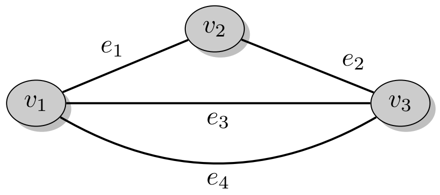
Simple Directed Graphs



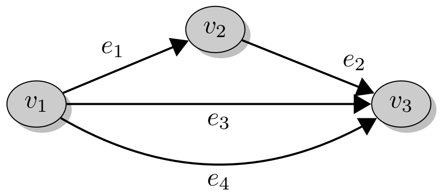
- edges $E \subseteq V \times V$ are (ordered) two-element tuples of V, e.g., $(v_1, v_3) \in E, (v_3, v_1) \notin E$
- source/tail node of an edge: outgoing (e.g., v_1 in (v_1, v_3))
- target/head node of an edge: incoming (e.g., v_3 in (v_1, v_3))
- complete graph: maximum of $n(n-1)$ edges for $n = |V|$ nodes (without self-loops)

Multigraphs

- a pair of nodes may be connected by multiple edges (in the same direction)
- undirected multigraph

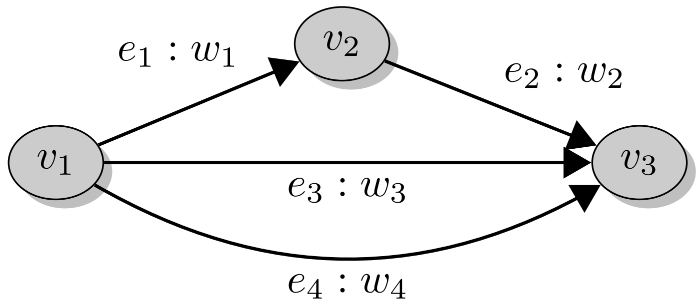


- directed multigraph



Weighted Graph

- a weight (e.g., road distance) is assigned to edges



Graph Traversals

- depth-first: visit start node, recursively traverse all un-visited neighbors in depth-first
- breath-first: visit start node (distance 0), visit all neighbors (distance 1), then all other nodes in increasing distance order
- Eulerian path/cycle: visit each edge exactly once
- Hamiltonian path/cycle: visit each vertex exactly once
- spanning tree: visit each vertex and a subset of edges such that visited vertices and edges form a tree

Graph Data Structures

- edge list
- adjacency matrix
- incidence matrix
- adjacency list
- incidence list

Edge List

- edge list follows mathematical definition: store edges E and nodes V as sets
- add/delete edge/node are efficient
- small memory
- most queries inefficient and require search among all edges:
  - find all neighbors of a node
  - find incident edges in directed graph
  - traverse a specific path

Adjacency Matrix

- matrix A of size $|V| \times |V|$
- element $a_{i,j}$ is the number of (directed) edges between v_i and v_j
- adjacency matrix for undirected graphs is symmetric
- adding/deleting nodes is problematic, adding/deleting edges is efficient
- storage size $O(|V|^2)$, large overhead if graph is sparse (small average degree, i.e., few edges per node)
- edge lookup by tail and head nodes is very efficient
- finding incident edges requires scanning matrix row or column

Incidence Matrix

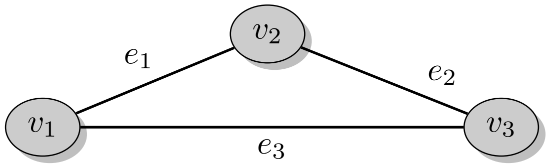
- matrix B of size $|V| \times |E|$
- element $b_{i,j}$ is 1 if edge e_i is incident to v_i (-1 for outgoing edge in directed graph)
- adding/deleting nodes/edges is problematic
- less memory than adjacency matrix for sparse graphs since no zero-only columns
- storage size may grow to $O(|V|^3)$ (since $|E| = O(|V|^2)$ in complete graph)
- checking for the existence of an edge between vertex pair is expensive
- finding incident edges requires searching matrix row
- finding the head for a given edge tail requires searching column

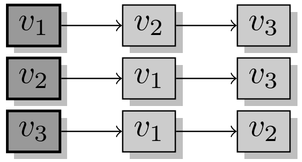
Adjacency List

- each vertex stores linked list of incident edges (outgoing edges in directed graph)
- edges are not stored explicitly
- adding/deleting nodes/edges is efficient
- finding all neighbors is efficient
- small memory
- checking existence of edge between vertex pair requires search in adjacency list
- finding incoming edges in directed graphs is inefficient (solution: forward *and* backward search adjacency list)

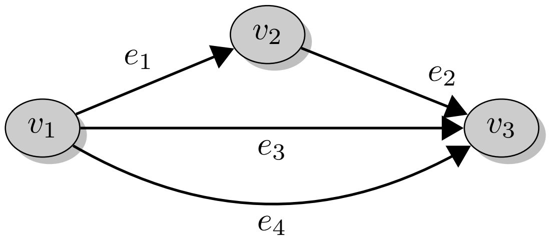
Adjacency List – Examples

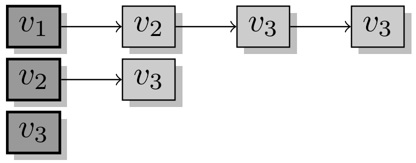
- simple, undirected graph





- directed multigraph



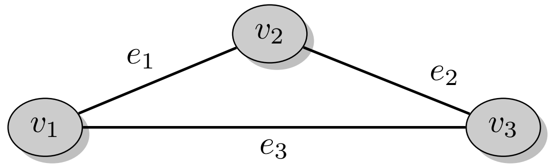


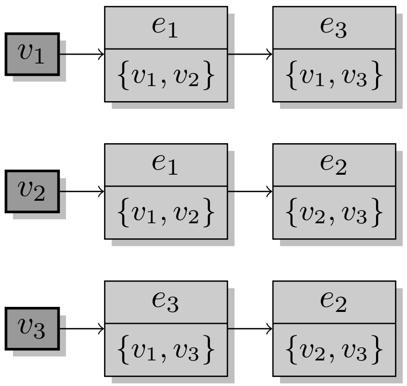
Incidence List

- each vertex stores linked list of incident edges (outgoing edges in directed graph)
- edges are listed explicitly such that information can be stored with edges
- finding all neighbors is efficient
- small memory
- checking existence of edge between vertex pair requires search in incidence list
- finding incoming edges in directed graphs is inefficient (solution: forward *and* backward search incidence list)

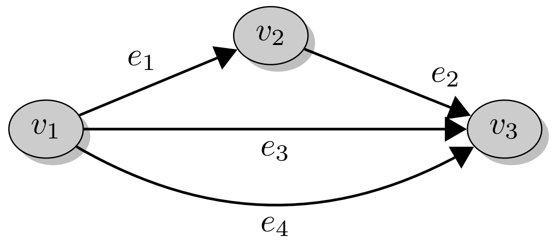
Incidence List – Examples

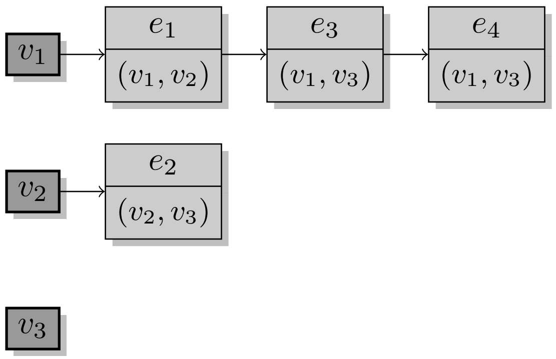
- simple, undirected graph





- directed multigraph



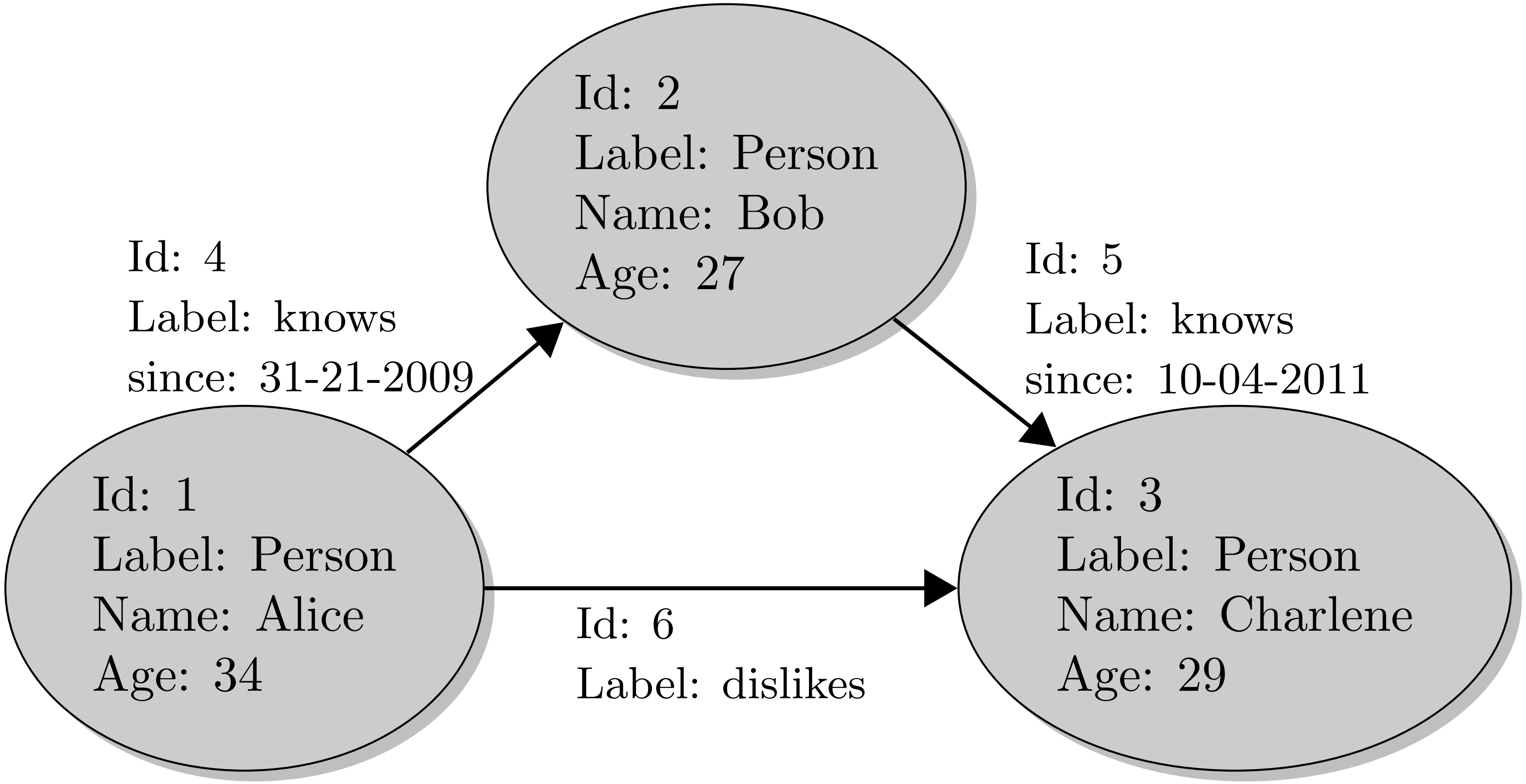


Inhalt

Property Graph Model

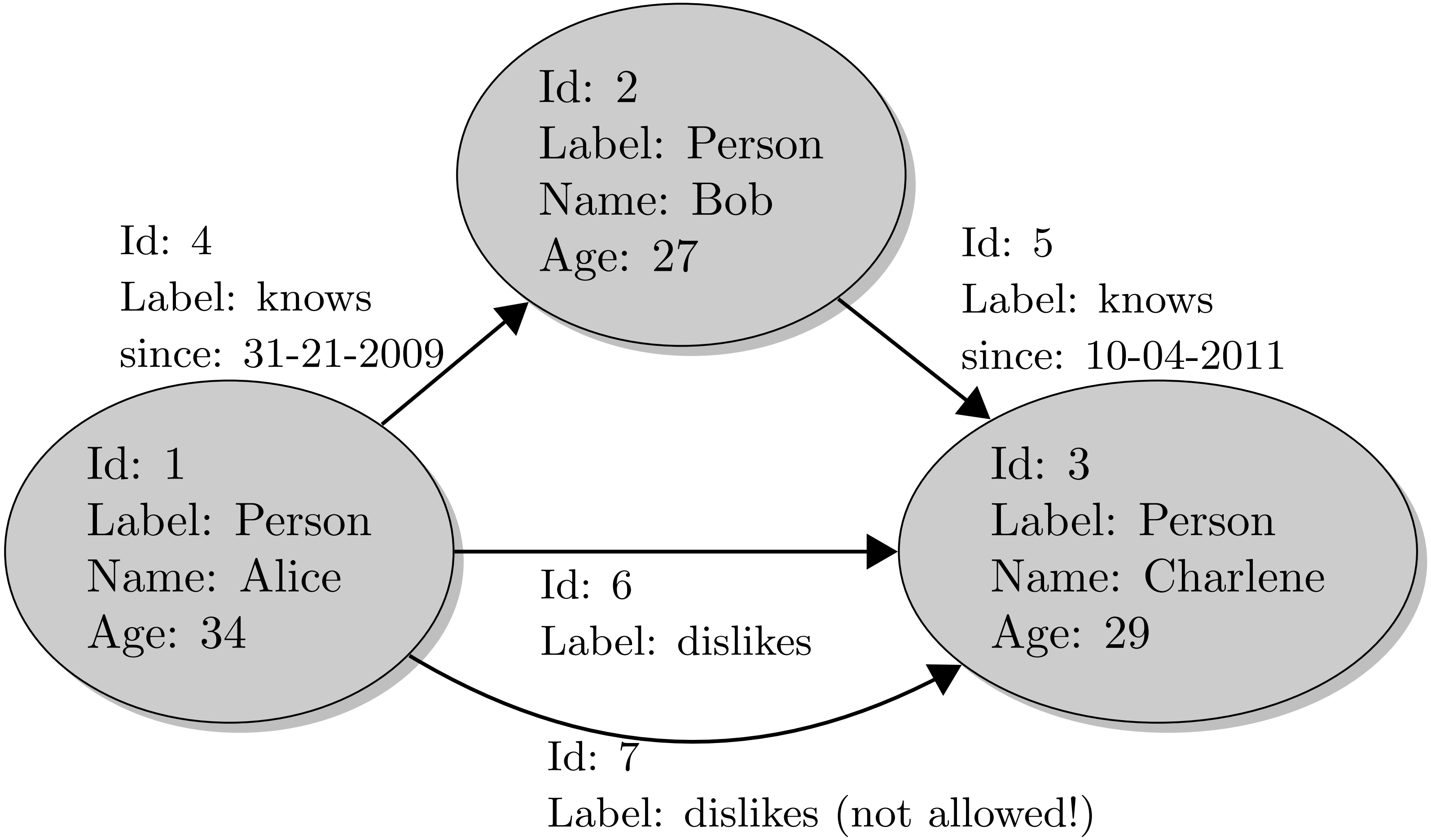
- directed, multi-relational, labeled multi-graph
- multi-relational
  - single-relational graph: only one “kind” of nodes/edges
  - multi-relational graph: nodes and edges have a type
- labels
  - node label is the node type
  - edge label is the edge type
- nodes and edges may have attributes
  - name:value pairs
  - name is the key (e.g., age)
  - value has a domain (e.g., non-negative integer)
- each node and each edge has an explicit ID
- only one edge of a specific type allowed between a given pair of nodes
- restrictions on edges can be defined (e.g., edges of type “likes” allowed only between nodes of type “person”)

Property Graph – Social Network Example



Property Graph – Social Network Example

- multiple edges between node pair only allowed if they differ by type



Storing Property Graphs in Relations

Alternative 1:

- Nodes and their attributes:
  Node(<u>NodeID</u>, NodeLabel)
  Person(<u>NodeID</u>, Name, Age)
  $\pi_{\text{NodeID}}(\text{Person}) \subseteq \pi_{\text{NodeID}}(\text{Node})$
- Edges and their attributes:
  Edge(<u>EdgeID</u>, EdgeLabel, Source, Target)
  Knows(<u>EdgeID</u>, Since)
  $\pi_{\text{EdgeID}}(\text{Knows}) \subseteq \pi_{\text{EdgeID}}(\text{Edge})$
  $\pi_{\text{Source}}(\text{Edge}) \subseteq \pi_{\text{NodeID}}(\text{Node})$
  $\pi_{\text{Target}}(\text{Edge}) \subseteq \pi_{\text{NodeID}}(\text{Node})$

Alternative 2:

- General attribute table:
  Attributes(<u>ID</u>, <u>Name</u>, Value)
  ID is edge or node ID, Name is attribute key
- problem: values may be of different type

Inhalt

Apache TinkerPop

- Java interfaces for property graphs
- Gremlin traversal language: queries over TinkerPop graphs
- TinkerPop-enabled databases implement these interfaces:[1]
  - Hadoop (Giraph) - OLAP graph processor using Giraph
  - Hadoop (Spark) - OLAP graph processor using Spark
  - Neo4j - OLTP graph database
  - Sqlg - RDBMS OLTP implementation with HSQLDB and Postresql support
  - TinkerGraph - In-memory OLTP and OLAP reference implementation
  - Titan - Distributed OLTP and OLAP graph database with BerkeleyDB, Cassandra and HBase support
  - ...
- storage backend can be substituted without changing the code

[1] see `http://tinkerpop.incubator.apache.org`

TinkerPop Structure API

- `Graph`: set of edges and vertices
- `Element`: has a label and a collection of properties
- `Vertex`: `Element` with incoming and outgoing edges
- `Edge`: `Element` with one incoming and one outgoing vertex
- `Property`: attribute key:value pair, key is of type string, `Property<V>` allows only values of type `V`
- `VertexProperty`: `Property` with a collection of key value pairs (i.e., allows for nested properties)

TinkerPop Structure API – Code Example

```
Graph g = TinkerGraph.open();
Vertex alice = g.addVertex("name", "Alice");
alice.property("age", 34);
Vertex bob = g.addVertex("name", "Bob");
alice.addEdge("knows", bob, "knows_since", 2010);
```

TinkerPop Graph Process API

- defines "traversals" in the graph
- traversal: definition of how the graph should be traversed (starting with nodes or edges)
- returns a `GraphTraversal` object (iterator)
- code example: names of all nodes that Alice knows

```
g.traversal().V().
   has("name","Alice").out("knows").values("name");
```

- Gremlin console is an interpreter for the Gremlin query language

Neo4J

- widely used graph database for property graphs
- support for ACID transactions (but eventual consistency with replicas)
- support for replication
- properties
  - Apache Lucene indices for properties
  - property names are strings
  - property values can be strings, booleans, numbers, or arrays
- CIPHER query language:

```
START alice = (people_index, name, "Alice")
MATCH (alice)-[:knows]->(aperson)
RETURN (aperson)
```

- TinkerPop enabled

Neo4J Clusters – Updates and Replication

- master node and slaves with full replication
- updates on slaves
  - slave must be up-to-date
  - acquire lock on slave and master
  - commit on master first
- replication
  - push from master to slaves
  - optimistic: commit happens before push is successful
  - eventual consistency: outdated reads on slave are possible

Neo4J Clusters – Availability

- failing nodes are detected and marked
- master fails:
  - other nodes elect new master
  - master needs quorum
  - no writes during master election
- network partitioning[2]:
  - writes only on (strict) majority partition with master
  - minority partition cannot elect a new master
  - minority partition with master cannot perform writes
  - reads are possible in any minority partition

[2] see `http://neo4j.com/resources/understanding-neo4j-scalability-white-paper/`

Resource Description Framework – RDF

- RDF stores so-called "linked data"
- RDF stores graphs as triples
  - subject (source node): string or URI
  - object (target node): string or URI
  - predicate (edge source→target): string or URI
- based on XML
- RDF databases are called "triple stores"
  - RDF3X (based on relations, joins, and B-tree indexes)
  - Blazegraph - RDF graph database with OLTP support
  - Oracle Spatial and Graph
  - ...
- common query language: SPARQL