Data Management in the Cloud, Lecture 11

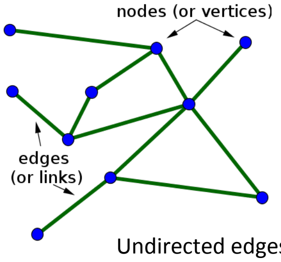
NEO4J: GRAPH DATA MODEL

Graph Data

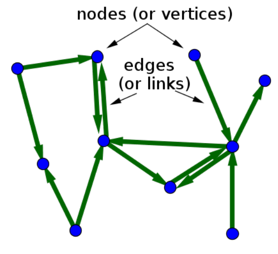
Many types of data can be represented with nodes and edges

Variations

- Edges can be directed or undirected
- Nodes and edges can have types or labels
- Nodes and edges can have attributes



Undirected edges

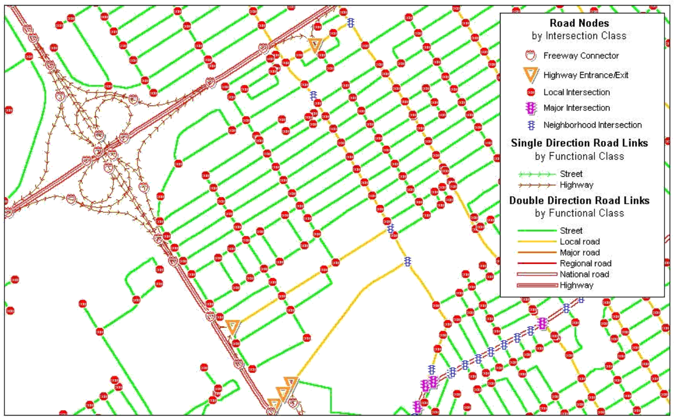


Directed edges

Credit: http://mathinsight.org/

Road Network

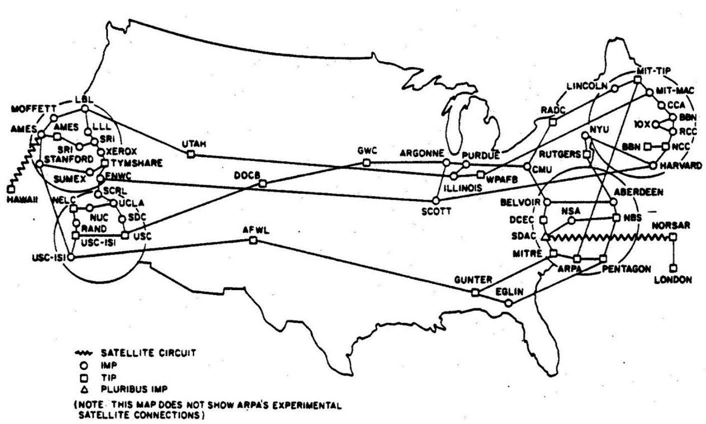
- Nodes: Intersections
- Edges: Road segments



Credit: Marius Thériault et al., Journal of Geographic Information and Decision Analysis, vol. 3, no. 1, pp. 41-55, 1999

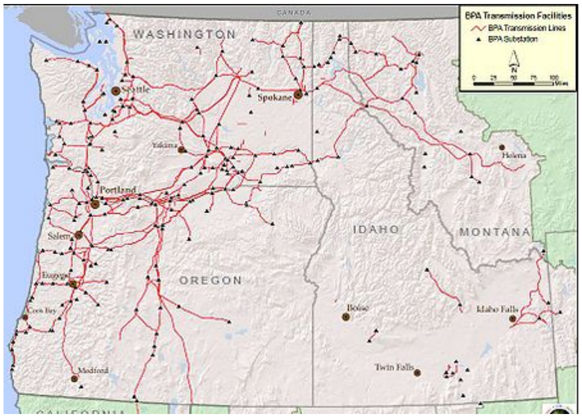
Computer Network

- Nodes: Computers
- Edges: Communication links



Power Transmission System

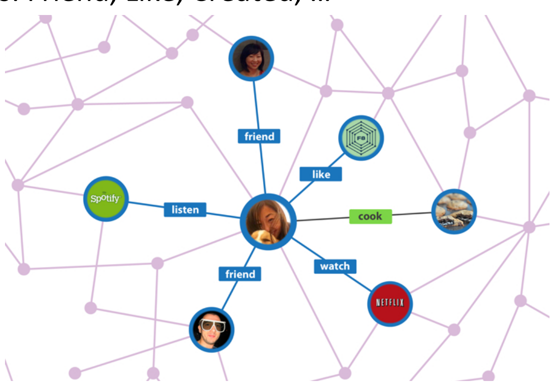
- Nodes: Substations
- Edges: Power transmission lines (possible attributes?)



Credit: http://portlandwiki.org/

Social Network

- Nodes: People, Postings
- Edges: Friend, Like, Created, …



Credit: http://mathinsight.org/

Discussion Question

Consider a graph of Twitter users (each node is a distinct user).

List some kinds of edges that might be in the graph

- Should the edge be directed or undirected?
- What attributes should the edge have?

See if you can come up with at least two kinds of edges.

7

Neo4j Nodes and Relationships

- Nodes
  - have a system-assigned id
  - can have key/value properties
  - there is a *reference node* ("starting point" into the node space)
- Relationships (Edges)
  - have a system-assigned id
  - are directed (but can be traversed in either direction)
  - have a type
  - can have key/value properties
- Key/value properties
  - values always stored as strings
  - support for basic types and arrays of basic types

8

Operations

- Nodes are managed using the **`GraphDatabaseService`** interface
  - **`createNode()`** creates and returns a new node
  - **`getNodeById(id)`** returns the node with the given id
  - **`getAllNodes()`** returns an iterator over all nodes (index is better)
- Relationships are managed using the **`Node`** interface
  - **`createRelationshipTo(target,type)`** creates and returns a relationship
  - **`getRelationships(direction,types)`** returns an iterator over a node's relationships
  - **`hasRelationship(type,direction)`** queries the existence of a certain relationship

9

Operations

- Node and relationship properties are managed using the **`PropertyContainer`** interface
  - **`setProperty(key,value)`** sets (or creates) a property
  - **`getProperty(key)`** returns a property value (or throws exception)
  - **`hasProperty(key)`** checks if a key/value property exists
  - **`removeProperty(key)`** deletes a key/value property
  - **`getPropertyKeys()`** returns all the keys of a node's properties
- Nodes and relationships are deleted using the corresponding method in the **`Node`** and **`Relationship`** interfaces

10

Example

```
GraphDatabaseService db = ...
Transaction tx = db.beginTx();
try {
   Node mike = db.createNode();
   mike.setProperty("name", "Michael");
   Node pdx = db.createNode();
   Relationship edge = mike.createRelationshipTo(pdx, LIVES_IN);
   edge.setProperty("years", new int[] { 2010, 2011, 2012 });
   for (edge: pdx.getRelationship(LIVES_IN, INCOMING)) {
      Node node = edge.getOtherNode(pdx);
   }
   tx.success();
} catch (Exception e) {
   tx.fail();
} finally {
   tx.finish();
}
```


Indexes

- Neo4j does not support any value-based retrieval of nodes and relationships without indexes
- Interface **`IndexManager`** supports the creation of node and relationship indexes
  - **`forNodes(name,configuration)`** returns (or creates) a node index
  - **`forRelationships(name,configuration)`** returns (or creates) a relationship index
- Behind the scenes, Neo4j indexes are based on Apache Lucene as an indexing service
- Values are indexed as strings by default, but a so-called *value context* can be used to support numeric indexing
- Neo4j also supports auto indexers for nodes and relationships

Node Indexes

- Index maintenance (manual)
  - **`add(node,key,value)`** indexes the given node based on the given key/value property
  - **`remove(node)`** removes all index entries for the given node
  - **`remove(node,key)`** removes all index entries for the given node with the given key
  - **`remove(node,key,value)`** removes a key/value property from the index for the given node
- Index lookups
  - **`get(key,value)`** supports equality index lookups
  - **`query(key,query)`** does a query-based index lookup for one key
  - **`query(query)`** does a query-based index lookup for arbitrary keys

Example

```
Index<Node> people = db.index().forNodes("people_idx");

// do an exact lookup
Node mike = people.get("name", "Michael").getSingle();

// do a query-based lookup for one key
for (Node node: people.query("name", "M* OR m*")) {
   System.out.println(node.getProperty("name");
}

// do a general query-based lookup
for (Node node: people.query("name:M* AND title:Mr") {
   System.out.println(node.getId());
}
```

Relationship Indexes

- Index maintenance is analogous to node indexes
- Additional index lookup functionality
  - **`get(key,value,source,target)`** does an exact lookup for the given key/value property, taking the given source and target node into account
  - **`query(key,query,source,target)`** does a query-based lookup for the given key, taking the given source and target node into account
  - **`query(query,source,target)`** does a general query-based lookup, taking the given source and target node into account
- Note: There is now schema-level indexing

Example

```
Index<Node> homes = db.index().forRelationships("homes_idx");

// do an exact lookup
Relationship r = homes.get("span", "2", mike, pdx).getSingle();

// do a query-based lookup for one key
for (Relationship r: homes.query("span", "*", mike, null)) {
   System.out.println(r.getOtherNode(mike));
}

// do a general query-based lookup
for (Relationship r:
         homes.query("type:LIVES_IN AND span:3", mike, null) {
   System.out.println(r.getOtherNode(mike));
}
```

Schema Indexing

- A more recent feature to support automatic indexing on nodes
- Depends on nodes being assigned to collections using *labels*.
  - A label can be assigned at creation time
  - `Node mike = db.createNode(Labels.ACTOR);`
- Can also create unique constraints on properties
- We'll see an example in Cypher

(*Note:* There is an automatic form of manual indexing, but it indexes a given property no matter what kind of node it belongs to – can't index **`name`** just for cities.)

17

Traversal Framework

- Neo4j provides a traversal interface to specify navigation through a graph
  - based on callbacks
  - executed lazily on demand
- Main concepts
  - **expanders** define what to traverse, typically in terms of relationships direction and type
  - the **order** guides the exploration, i.e. depth-first or breadth-first
  - **uniqueness** indicates whether nodes, relationships, or paths are visited only once or multiple times
  - an **evaluator** decides what to return and whether to stop or continue traversal beyond the current position
  - a **starting node** where the traversal will begin

18

Example: DFS in Finding Bridges

```
List<Relationship> result = ...
Set<Node> roots = ...

IndexManager manager = this.database.index();
Index<Node> dfsNodes = manager.forNodes("dfsNodes");
RelationshipIndex treeEdges = manager.forRelationships("treeEdges");

TraversalDescription traversal = new TraversalDescriptionImpl();
traversal = traversal.order(Traversal.postorderDepthFirst());
traversal = traversal.relationships(EDGE, OUTGOING);

int treeId = 0;
while (!roots.isEmpty()) {
   Node root = roots.iterator().next();
   Traverser traverser = traversal.traverse(root);
   int pos = 0;
   for (Node node : traverser.nodes()) {
      dfsNodes.add(node, P_DFSPOS, treeId + ":" + pos);
      roots.remove(node);
      pos++;
   }
   for (Relationship relationship : traverser.relationships()) {
      treeEdges.add(relationship, P_ID, relationship.getId());
   }
   result.addAll(this.tarjan(dfsNodes, treeEdges, treeId));
   treeId++;
}
```

19

Graph Algorithms

- Some common graph algorithms are directly supported
  - all **shortest paths** between two nodes up to a maximum length
  - **all paths** between two nodes up to a maximum length
  - all **simple paths** between two nodes up to a maximum length
  - **"cheapest" path** based on Dijkstra or A*
- Class **`GraphAlgoFactory`** provides methods to create **`PathFinder`**s that implement these algorithms

20

Example: Shortest Path

```
// unweighted case
PathFinder<Path> pathFinder = GraphAlgoFactory.shortestPath(
      Traversal.expanderForTypes(EDGE, OUTGOING),
      Integer.MAX_VALUE);
Path path = pathFinder.findSinglePath(source, target);
for (Node node: path.nodes()) {
   System.out.println(node);
}

// weighted case
PathFinder<WeightedPath> pathFinder = GraphAlgoFactory.dijkstra(
      Traversal.expanderForTypes(EDGE, OUTGOING), P_WEIGHT);
Path path = pathFinder.findSinglePath(source, target);
for (Relationship relationship: path.relationships()) {
   System.out.println(relationship);
}
```


Queries

- Support for the Cypher graph query language has been added to Neo4j
- Unlike the imperative graph scripting language Gremlin, Cypher is a declarative language
- Cypher is comprised of four main concepts
  - **`START`**: starting points in the graph, obtained by element IDs or via index lookups
  - **`MATCH`**: graph pattern to match, bound to the starting points
  - **`WHERE`**: filtering criteria
  - **`RETURN`**: what to return
- Implemented using the Scala programming language

Example: Director and Actor with Same Last Name in a Musical

```
MATCH (a:PERSON)-[:IS-IN]->(m:MOVIE)<-[:DIRECTS]-(b:PERSON)
WHERE a.LastName = b.LastName AND m.Genre = "musical"
RETURN a.LastName, m.Title
```

23

Example: Index Creation, Constraint

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

MATCH (p:ACTOR {name: 'Michael'}) RETURN p

CREATE CONSTRAINT ON (p:ACTOR) ASSERT p.name IS UNIQUE
```

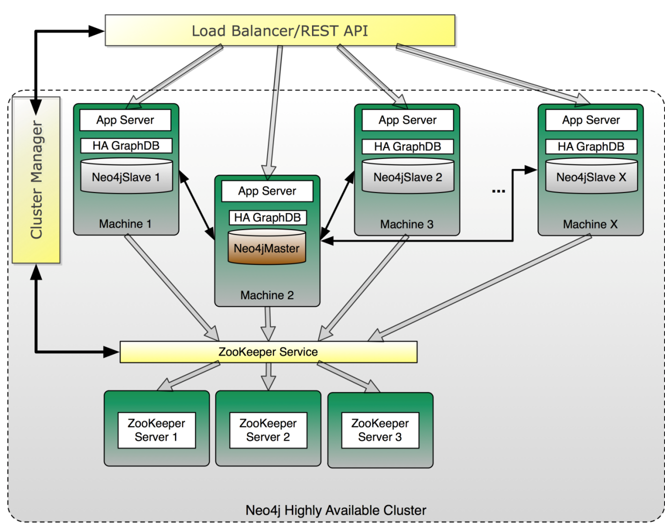
24

Deployments

- Several deployment scenarios are supported
- Embedded database
  - wraps around a local directory
  - implements the **`GraphDatabaseService`** interface
  - runs in the same process as application, i.e. no client/server overhead
- Client/server mode
  - server runs as a standalone process
  - provides Web-based administration
  - communicates with clients through REST API
- High availability setup
  - one master and multiple slaves, coordinated by ZooKeeper
  - supports fault tolerance and horizontal scaling
  - implements the **`GraphDatabaseService`** interface

25

High Availability Setup



26

High Availability Setup

- High availability
  - reads are highly available
  - updates to master are replicated asynchronously to slaves
  - updates to slaves are replicated synchronously to master
  - transactions are atomic, consistent and durable on the master, but eventually consistent on slaves
- Fault tolerance
  - depending on ZooKeeper setup, Neo4j can continue to operate from any number of machines down to a single machine
  - machines will be reconnected automatically to the cluster whenever the issue that caused the outage (network, maintenance) is resolved
  - if the master fails a new master will be elected automatically
  - if the master goes down any running write transaction will be rolled back and during master election no write can take place