Class 8: Introduction to graph databases
Using Neo4j for network analysis and visualization

Boleslaw Szymanski
Konstantin Kuzmin
Main Topics

• Overview of graph databases
• Installing and using Neo4j
• Neo4j hands-on labs
Graph Databases

• Use graph structures for semantic queries with nodes, edges, and properties to represent and store data
• Use the Property Graph Model:
  – Connected entities (nodes) can hold any number of attributes (key-value-pairs) and can be tagged with labels representing their different roles in your domain
  – Relationships provide directed, named connections between two node-entities. A relationship always has a direction, a type, a start node, and an end node.
• Well suited for semi-structured and highly connected data
• Require a new query language
Relational vs. Graph Databases

• Relational
  – Store highly structured data in tables with predetermined columns of certain types and many rows of the same type of information
  – Require developers and applications to strictly structure the data used in their applications
  – References to other rows and tables are indicated by referring to their (primary-)key attributes via foreign-key columns
  – In case of many-to-many relationships, you have to introduce a JOIN table (or junction table) that holds foreign keys of both participating tables which further increases join operation costs

• Graph
  – Relationships are first-class citizens of the graph data model
  – Each node (entity or attribute) directly and physically contains a list of relationship-records that represent its relationships to other nodes
  – The ability to pre-materialize relationships into database structures provides performances of several orders of magnitude advantage
Neo4j Graph Database

- NoSQL Graph Database
- Implemented in Java and Scala
- Open source
- Free and open-source Community edition and Enterprise editions which provide all of the functionality of the Community edition in addition to scalable clustering, fail-over, high-availability, live backups, and comprehensive monitoring.
- Full database characteristics including ACID transaction compliance, cluster support, and runtime failover
- Constant time traversals for relationships in the graph both in depth and in breadth
Cypher Query Language

- SQL-inspired language for describing patterns in graphs visually using an ASCII-art syntax
- Declarative – allows us to state **what** we want to select, insert, update or delete from our graph data without requiring us to describe exactly **how** to do it
- Contains clauses for searching for patterns, writing, updating, and deleting data
- Queries are built up using various clauses. Clauses are chained together, and they feed intermediate result sets between each other
- Cypher query gets compiled to an execution plan that can run and produce the desired result
- Statistical information about the database is kept up to date to optimize the execution plan
- Indexes on Node or Relationships properties are supported to improve the performance of the application
Neo4j API

- **REST API**
  - Designed with discoverability in mind (discover URIs where possible)
  - Stateless interactions store no client context on the server between requests
  - Supports streaming results, with better performance and lower memory overhead

- **HTTP API**
  - Transactional Cypher HTTP endpoint
  - POST to a HTTP URL to send queries, and to receive responses from Neo4j

- **Drivers**
  - The preferred way to access a Neo4j server from an application
  - Use the Bolt protocol and have uniform design and use
  - Available in four languages: C# .NET, Java, JavaScript, and Python
  - Additional community drivers for: Spring, Ruby, PHP, R, Go, Erlang / Elixir, C/C++, Clojure, Perl, Haskell
  - API is defined independently of any programming language

- **Procedures**
  - Allow Neo4j to be extended by writing custom code which can be invoked directly from Cypher
  - Written in Java and compiled into jar files
  - To call a stored procedure, use a Cypher CALL clause
Neo4j Resources

- Neo4j Web site: https://neo4j.com/
- Cypher Refcard https://neo4j.com/docs/cypher-refcard/current/
- Coursera course “Graph Analytics for Big Data” from the University of California, San Diego (https://www.coursera.org/learn/big-data-graph-analytics) has a lesson “Graph Analytics With Neo4j”
Main Topics

• Overview of graph databases
• **Installing and using Neo4j**
• Neo4j hands-on labs
Neo4j Installation

- Neo4j runs on Linux, Windows, and OS X
- A Java 8 runtime is required
- For Community Edition there are desktop installers for OS X and Windows
- Several ways to install on Linux, depending on the Linux distro (see the “Neo4j Resources” slide)
- Check the `/etc/neo4j/neo4j.conf` configuration file:
  ```
  # HTTP Connector
  dbms.connector.http.type=HTTP
  dbms.connector.http.enabled=true
  # To accept non-local HTTP connections, uncomment this line
  dbms.connector.http.address=0.0.0.0:7474
  ```
- File locations depend on the operating system, as described here: [https://neo4j.com/docs/operations-manual/current/deployment/file-locations/](https://neo4j.com/docs/operations-manual/current/deployment/file-locations/)
- Make sure you start the Neo4j server (e.g., “./bin/neo4j start” or “service neo4j start” on Linux)
Neo4j Browser

• Open the URL [http://localhost:7474](http://localhost:7474) (replace “localhost” with your server name, and 7474 with the port name as set in neo4j.conf)

• Enter the username/password (if not set, Neo4j browser will prompt you to select the username and password)

• Start working with Neo4j by entering Cypher queries and observing their results

• Save frequently used Queries to Favorites
The Structure of a Cypher Query

- Nodes are surrounded with parentheses which look like circles, e.g. (a)
- A relationship is basically an arrow --> between two nodes with additional information placed in square brackets inside of the arrow
- A query is comprised of several distinct clauses, like:
  - MATCH: The graph pattern to match. This is the most common way to get data from the graph.
  - WHERE: Not a clause in its own right, but rather part of MATCH, OPTIONAL MATCH and WITH. Adds constraints to a pattern, or filters the intermediate result passing through WITH.
  - RETURN: What to return.

MATCH (john {name: 'John'})-[[:friend]->()]-[[:friend]->(fof) RETURN john.name, fof.name
Writing Cypher Queries

- Node labels, relationship types and property names are case-sensitive in Cypher.
- `CREATE` creates nodes with labels and properties or more complex structures.
- `MERGE` matches existing or creates new nodes and patterns. This is especially useful together with uniqueness constraints.
- `DELETE` deletes nodes, relationships, or paths. Nodes can only be deleted when they have no other relationships still existing.
- `DETACH DELETE` deletes nodes and all their relationships.
- `SET` sets values to properties and add labels on nodes.
- `REMOVE` removes properties and labels on nodes.
- `ORDER BY` is a sub-clause that specifies that the output should be sorted and how.
Importing and Exporting Data

• Loading data from CSV is the most straightforward way of importing data into Neo4j
• For fast batch import of huge datasets, use the neo4j-import tool
• Lots of other tools for different data formats and database sizes
• Export data using Neo4j browser or neo4j-shell-tools
Loading Data from CSV

- Understand your graph model

\[(\text{p1}:\text{Person}\{\text{userId}:10, \text{name}:"Anne"\})-[:\text{KNOWS}]\rightarrow(\text{p2}:\text{Person}\{\text{userId}:123, \text{name}:"John"\})\]

- CSV files
  - people.csv
  - friendships.csv
    1,234 10,4893 234,1 4893,234943 234943,234 234 234943,1

- Run the following Cypher queries:
  - CREATE CONSTRAINT ON (p:Person) ASSERT p.userId IS UNIQUE;
  - LOAD CSV FROM "file:///people.csv" AS csvLine
    MERGE (p:Person \{userId: toInteger(csvLine[0]), name: csvLine[1]\});
  - USING PERIODIC COMMIT LOAD CSV FROM "file:///friendships.csv" AS csvLine
    MATCH (p1:Person \{userId: toInteger(csvLine[0])\}), (p2:Person \{userId: toInteger(csvLine[1])\})
    CREATE (p1)-[:KNOWS]->(p2);
  - CREATE INDEX ON :Person(name);

- Check the results:

\[
\text{MATCH (}:\text{Person}\{\text{name}:"Anne"\})-[:\text{KNOWS}*2..2]-(p2) \text{ RETURN p2.name, count(*) as freq ORDER BY freq DESC;}
\]
Loading Data from a Spreadsheet

- Lay out your data in a spreadsheet

- Use formulas to generate the required Cypher statements

- Collect Cypher queries and run them

- Check the results:

  MATCH (p1:Person)-[:ATTENDS]-(e:Event{name:"Meetup Malmö"})-[:ATTENDS]-(p2:Person) WHERE (p1)-[:FRIENDS_WITH]-(p2) RETURN p1, p2, e;
Loading Data from a GraphML file

- Use **neo4j-shell-tools** from [https://github.com/jexp/neo4j-shell-tools](https://github.com/jexp/neo4j-shell-tools)
- Populate the database from a GraphML file
  
  `import-graphml -i /usr/share/neo4j/import/airlines.graphml -r
  HAS_DIRECT_FLIGHTS_TO -b 20000 -c -t`

- Check the results:

  ```
  MATCH (a)--()
  WITH a.tooltip as airport, count(*) as flights
  RETURN airport, flights ORDER BY flights DESC LIMIT 10
  ```
Loading Data from an Arbitrary Format

- Write a simple piece of code to convert your file into a set of two CSV files
- Load data from the CSV file into a Neo4j database
  - CREATE CONSTRAINT ON (p:Person) ASSERT p.userId IS UNIQUE;
  - LOAD CSV WITH HEADERS FROM "file:///Wiki-Vote-nodes.csv" AS csvLine MATCH (p:Person {userId: toInt(csvLine.NodeID)}) RETURN;
  - USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM "file:///Wiki-Vote-edges.csv" AS csvLine MATCH (p1:Person {userId: toInt(csvLine.EdgeFrom)}), (p2:Person {userId: toInt(csvLine.EdgeTo)}) CREATE (p1)-[:VOTED_ON]->(p2);
  - CREATE INDEX ON :Person(name);
- Check the results:
  MATCH (p:Person)-[r]-(
  WITH p as persons, count(distinct r) as degree
  RETURN degree, count(persons) ORDER BY degree ASC

from sys import argv

def read_edge_list(filename):
    nodeset= set([])
edgelist = []
    with open(filename, 'r') as file_handle:
        for line in file_handle:
            if line[0] != '#':
                data = line.split()
                node_from = data[0]
                node_to = data[1]
                nodeset.add(node_from)
                nodeset.add(node_to)
edgelist.append([node_from, node_to])
    return nodeset, edgelist

def write_csv_nodes(nodes, file_nodes):
    with open(file_nodes, 'w') as file_handle:
        file_handle.write("NodeID\n")
        for node in nodes:
            file_handle.write('{0}\n'.format(node))

def write_csv_edges(edges, file_nodes):
    with open(file_nodes, 'w') as file_handle:
        file_handle.write("EdgeFrom,EdgeTo\n")
        for edge in edges:
            file_handle.write('{0},{1}\n'.format(edge[0], edge[1]))

script, input_file, output_file_nodes, output_file_edges = argv
nodes, edges = read_edge_list(input_file)
write_csv_nodes(nodes, output_file_nodes)
write_csv_edges(edges, output_file_edges)
Exporting Data From Neo4j

• Click the download icon on the table view of the Cypher query results
• Use neo4j-shell-tools to export results of a Cypher query to a CSV or GraphML file
• Access the graph data with Neo4j API and save it in the desired format
Analyzing Graph Data with MATLAB

- Load CSV data exported from Neo4j into MATLAB
- Use MATLAB to perform additional analysis and to draw plots
- Export analysis results and plots for publication

```matlab
filename = 'filename.csv';
M = csvread(filename,1,0);
x = M(:,1);
y = M(:,2);
plot(x,y)
f = fit(x,y,'poly2')
plot(f,x,y)
f = fit(x,y,'power1')
plot(f,x,y)
```
Accessing Neo4j Data using REST API

- Service root is the starting point to discover the REST API
  - GET http://localhost:7474/db/data/
  - Accept: application/json; charset=UTF-8

- Create node with properties
  - POST http://localhost:7474/db/data/node
  - Accept: application/json; charset=UTF-8
  - Content-Type: application/json
  - { "foo" : "bar" }

- Create relationship
  - POST http://localhost:7474/db/data/node/66/relationships
  - Accept: application/json; charset=UTF-8
  - Content-Type: application/json
  - { "to" : "http://localhost:7474/db/data/node/67", "type" : "FRIENDS_WITH" }
Using Transactional Cypher HTTP Endpoint

- Allows you to execute a series of Cypher statements within the scope of a transaction
- The transaction may be kept open across multiple HTTP requests, until the client chooses to commit or roll back
- Each HTTP request can include a list of statements
- Requests should include an Authorization header, with a value of Basic <payload>, where "payload" is a base64 encoded string of "username:password"

```python
import requests
from requests.exceptions import ConnectionError
import json

NEO4J_SERVER = 'http://ganxis.nest.rpi.edu:7474'
NEO4J_COMMIT_ENDPOINT = '/db/data/transaction/commit'
NEO4J_CREDENTIALS = 'bmVvNGo6bmVvNGo='

def execute_neo4j_cypher(url, credentials, query, parameters):
    result = None
    query_text = json.dumps(dict(statements = [dict(statement = query, parameters = parameters)]))

    headers = {'Accept' : 'application/json', 'Content-type' : 'application/json', 'Authorization:' : 'Basic ' + credentials}
    try:
        resp = requests.post(url, headers = headers, data = query_text)
        result = resp.json()
    except ConnectionError as exception:
        print exception
        # Log error
    if len(result['errors']) > 0:
        print '@@@ ERROR! Error executing Cypher query'
        # Log error
    print '@@@ ', query, '<-', parameters
    print '@@@ ' + str(result)
    return result

query = 'MERGE (p: Person {id:{userid}, name:{name}}) ON CREATE
SET p.created = timestamp() ON MATCH SET p.matched =
timestamp() RETURN p'
parameters = dict()
parameters['userid'] = 17
parameters['name'] = 'J J'
execute_neo4j_cypher(NEO4J_SERVER + NEO4J_COMMIT_ENDPOINT, NEO4J_CREDENTIALS, query, parameters)
```
Using Drivers to Access Neo4j

- Binary Bolt protocol (starting with Neo4j 3.0)
- Binary protocol is enabled in Neo4j by default and can be used in any language driver that supports it
- Native Java driver officially supported by Neo4j
- Drivers implement all low level connection and communication tasks

```java
import org.neo4j.driver.v1.*;

public class Neo4j {
    public static void javaDriverDemo() {
        Driver driver = GraphDatabase.driver("bolt://ganxis.nest.rpi.edu", "neo4j", "neo4j");
        Session session = driver.session();

        StatementResult result = session.run("MATCH (a)-[()-](b)-[()-](c)-[()-](a) WHERE a.id < b.id AND b.id < c.id RETURN DISTINCT a, b, c");
        int counter = 0;
        while (result.hasNext()) {
            counter++;
            Record record = result.next();
            System.out.println(record.get("a").get("id") + "	" + record.get("b").get("id") + "	" + record.get("c").get("id"));
        }
        System.out.println("Count: " + counter);
        session.close();
        driver.close();
    }
    public static void main(String[] args) {
        javaDriverDemo();
    }
}
```
Using Core Java API

- Native Java API performs database operations directly with Neo4j core

```java
import java.io.*;
import java.util.*;
import org.neo4j.graphdb.*;

public class Neo4j {
    public enum NodeLabels implements Label { NODE; }
    public enum EdgeLabels implements RelationshipType { CONNECTED; }
    public static void javaNativeDemo(int nodes, double p) { 
        Node node1, node2; Random randomgen = new Random();
        GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
        GraphDatabaseService db = dbFactory.newEmbeddedDatabase(new File("TestNeo4jDB"));
        try (Transaction tx = db.beginTx()) {
            for (int i = 1; i <= nodes; i++) {
                Node node = db.createNode(NodeLabels.NODE);
                node.setProperty("id", i);
            }
            for (int i = 1; i <= nodes; i++)
                for (int j = i + 1; j <= nodes; j++) {
                    if (randomgen.nextDouble() < p) {
                        node1 = db.findNode(NodeLabels.NODE, "id", i);
                        node2 = db.findNode(NodeLabels.NODE, "id", j);
                        Relationship relationship = node1.createRelationshipTo(node2, EdgeLabels.CONNECTED);
                        relationship = node2.createRelationshipTo(node1, EdgeLabels.CONNECTED);
                    }
                }
            tx.success();
        }
        db.shutdown();
    }
    public static void main(String[] args) {
        javaNativeDemo(100, 0.2);
    }
}
```
Main Topics

• Overview of graph databases
• Installing and using Neo4j
• Neo4j hands-on labs
Overview

Exercise 1

• Learn how to use Neo4j interface
• Import a network file
• Export a network file
• Visualize the graph

Exercise 2

• Learn the basics of Cypher
• Compute simple network measures
• Compute additional network measures using more advanced Cypher queries

Exercise 3

• Access Neo4j graph database using an API
• Generate a Barabási–Albert graph