Database Project Final Report

SIGMOD Programming Challenge

# Problem Statement

The task is to construct a social network analysis system. The aim is to run a set of queries as quickly as possible. There are four different kinds of queries.

## Query Type 1 (Shortest Distance Over Frequent Communication Paths)

Given two integer person ids p1 and p2, and another integer x, find the minimum number of hops between p1 and p2 in the graph induced by persons who have made more than x comments in reply to each other’s' comments, and know each other.

**API**: query1 (p1, p2, x)

**Output**: One integer (hop count) per line.

## Query Type 2 (Interests with Large Communities)

Given an integer k and a birthday d, find the k interest tags with the largest range, where the range of an interest tag is defined as the size of the largest connected component in the graph induced by persons who

* have that interest,
* were born on d or later, and
* know each other

**API**: query2 (k, d)

**Output**: Exactly k strings (separated by a space) per line. These k strings represent interest tag names, ordered by range from largest to smallest, with ties broken by lexicographical ordering.

## Query Type 3 (Socialization Suggestion)

Given an integer k, an integer maximum hop count h, and a string place name p, find the top-k similar pairs of persons based on the number of common interest tags. For each of the k pairs mentioned above, the two persons must be located in p or study or work at organizations in p. Furthermore, these two persons must be no more than h hops away from each other in the graph induced by persons and person\_knows\_person.

**API**: query3 (k, h, p)

**Output**: Exactly k pairs of person ids per line. These pairs are separated by a space and person ids are separated by the pipe character (|). For any person id p, p | p must be excluded. For any pairs p1 | p2 and p2 | p1, the second pair in lexicographical order must be excluded. These k pairs must be ordered by similarity from highest to lowest, with ties broken by lexicographical ordering.

## Query Type 4 (Most Central People)

Given an integer k and a string tag name t, find the k persons who have the highest closeness centrality values in the graph induced by persons who

* are members of forums that have tag name, and
* know each other

Here, the closeness centrality of a person p is

where r(p) is the number of vertices reachable from p (inclusive), s(p) is the sum of geodesic distances to all other reachable persons from p, and n is the number of vertices in the induced graph. When either multiplicand of the divisor is 0, the centrality is 0.

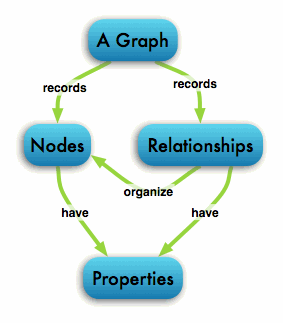
**API:** query4 (k, t)

**Output:** Exactly k person ids (separated by a space) per line. These person ids are ordered by centrality from highest to lowest, with ties broken by person id (in ascending order).

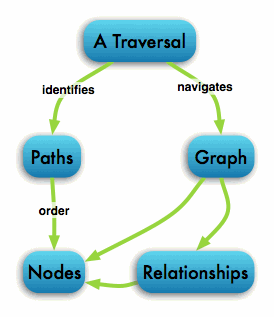
# Tools Utilized

## Neo4j

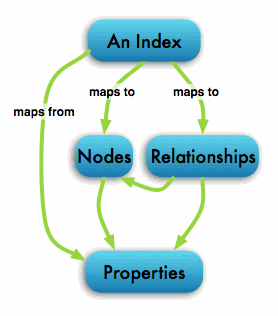
Neo4j is a highly scalable, robust (fully ACID) native graph database. Data in a graph database is stored in the form of a graph, which is the most generic form of data storage possible. The graph contains nodes and relationships (edges). The nodes represent records. The relationships describe how the nodes are connected. Each node and each relationship can be associated with a set of properties. The below graphic illustrates the concept:



Querying the graph is done using traversals. We start off with a node, and then describe the patterns of relationships and nodes that provide us with the answer we want. In essence, Neo4j identifies paths, which are orderings of nodes that satisfy the criteria we provide. The following graphic illustrates how traversals work:



It is often useful to look up nodes or relationships based on some property. Neo4j allows one to create indices on properties in order to allow fast look ups. If one specifies the value of a property, one is able to retrieve the nodes or relationships whose property holds the specified value very fast.



### Cypher

Cypher is Neo4j’s declarative graph query language. It is very expressible, and the learning curve is very small, especially for someone knowledgeable in SQL. Some of the important clauses in Cypher are described below:

* MATCH describes the graph pattern to match. This is the most common way to get data from the graph
* WHERE adds constraints to a pattern, or filters the intermediate result passing through WITH.
* RETURN describes what to return as a result of the query
* CREATE creates nodes and relationships
* DELETE deletes nodes and relationships
* SET sets values to properties and add labels on nodes
* REMOVE removes values that are set to properties and deletes labels from nodes
* MERGE matches existing or creates new nodes and patterns. This is especially useful together with uniqueness constraints

## Py2neo

Py2neo is a python library that provides access to the Neo4j graph database using its RESTful web service interface.