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. The descriptions below are mainly from the contest website: <http://www.cs.albany.edu/~sigmod14contest/task.html>

## 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.

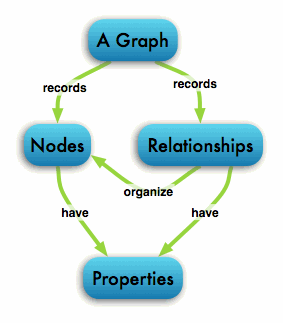
**API:** query4 (k, t)

**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).

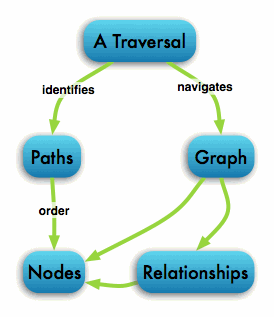
# Tools Utilized

## Neo4j

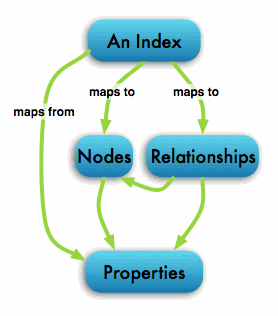
Neo4j is a highly scalable, robust (fully ACID) native graph database. Data in a graph database is stored in the form of a graph, which is the most generic form of data storage possible. The graph contains nodes and relationships (edges). The nodes represent records. The relationships describe how the 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.



The diagrams and descriptions above were mainly from the Neo4j website at <http://www.neo4j.org>

### 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. The library is self-contained and it is fairly easy to get started on it.

The core classes that are needed for the manipulation of the Neo4j database system are GraphDatabaseService, Node and Relationship.

A GraphDatabaseService object maintains a link to a Neo4j database using a root URI. The object’s methods are used to perform all operations on the database.

The Node class represents a node in the graph. Similarly, the Relationship class represents a relationship. In general, Node and Relationship objects are not directly created in the code. Instead, methods of the GraphDatabaseService object are used to create them.

# Design Principles

We used a few design principles to help guide us during this project. They are discussed below:

## Split Computation

We divided the computation for each query into a loading and a query-execution phase. Loading happens once for a given data-set. Query execution then takes place for each query that is submitted. The logic behind such a split is that if most of the computation is done during the loading stage, the cost of the loading phase can be amortized over several query-runs. If the number of query runs is reasonably high, extra computation during the loading phase will not majorly impact the total execution time for the set of queries.

Of course, if there is a change in the dataset, we would have to flush out the loaded data and load in the data for the new data-set. The assumption here is that quite a few query-runs take place over the same data-set.

Loading actually comprises of two primary phases. In the first phase, the actual data is loaded from the provided csv files into the Neo4j database. In the second phase, indices are built on relevant keys so that querying can be done very efficiently.

## Batch-Load Operations

Py2neo uses Neo4j’s RESTful web interface. As a consequence of this, individual communications with the database occurs using socket calls. This proves very expensive when there are a large number of such socket calls. The aim then is to reduce such socket calls. The main way through which this is achieved is by using the batch-load operations provided by the library.

Instead of using a separate socket call for each node, we bunch nodes and edges together and send them to Neo4j as a batch of nodes and edges. Even though the amount of information communicated is large, the number of socket calls decreases, and this results in faster communication.

## Less Dependence on the Neo4j Optimizer

One of the realizations we had early on was that Neo4j’s optimizer does not perform well on reasonably complex queries. All of the queries specified in the competition can actually be expressed as a single Cypher query. However, we found that the performance of such a query was dismal.

Neo4j, however, is able to filter out nodes and relationships based on patterns very efficiently. Once we get the filtered list of nodes and relationships, it is easy to add in more constraints and obtain the actual results.

# Important Optimizations

Below are some of the important optimizations we carried out in order to get more performance out of our system.

## Deferred Loading

This optimization actually pushes some of the loading part into the query execution phase. This seems counterintuitive, but there is a certain scenario where this proves efficient.

Some instances of a query need only part of the loaded database to generate results. For example, in query 1, we had to find the path between two persons with certain constraints. We found that in many of the given query instances, if the number of comments requirement is given as -1, we don’t need the comments database at all. The problem devolves to finding the shortest path between two person nodes.

If the number of query instances which reduce to finding the shortest path is a lot, it does not make sense to prolong the load phase by loading in the entire comments database. The comments database is huge, and expands exponentially with the number of persons in the graph. In this case, we just load in the comments needed for a particular query. This works out better because we don’t need to do the costly load of the comments database.

## Optimal Batch Size

The batch size is a trade-off. Making the batch size too small leads to many socket requests. Each time the Python program has to communicate with the database, it needs to make a socket call. Every batch submission thus needs a socket call. This leads to degradation in the performance of the system.

However, making the batch size huge is not a solution either. The problem with this is, the batch data structures take up most of the system memory and the virtual memory system starts to page stuff out to disk. Once this starts to happen, there is an enormous slowdown in the performance of the system. Furthermore, Python does not cope well with large objects in memory and that leads to further slowdown.

The aim then was to use the optimal batch size for each batch that reflects the resources of the system. We fine-tuned the system to our personal machine, but in a production system, we anticipate being able to set a global variable that is reflective of the capabilities of the machine on which the system is being run.

## Handling Special Cases

Some query instances demanded special handling. The general approach that worked for most of the cases was just not good enough for these special cases. When we felt that the performance boost was worth the extra programming effort and code bloat, we went in and tweaked the approach to take advantage of the special case.

For example, as touched upon previously, in query one, if the comments option in the query is -1, the solution devolves to finding the shortest path between two persons. In such a case, we just detected the case, and then used a shortest path algorithm directly without looking at the comments data.

In another case, for query three, the task is to find common interests among people who live in a particular geographical area. If the area is a city, for example, the number of people involved is likely to be very less. However, if the area is a continent, the number of people is enormous, and the same approach is unlikely to give an efficient result.

# The Approach

## Indexes in Neo4j

Neo4j inherently requires building indexes. This is due to the fact that it is a graph database, and so no edge or node should be loaded more than once, while loading data from a file or elsewhere. For example, consider the case when data is to be loaded from a CSV file, where each line represents an edge. If there are two lines in the file that involve a common node, say (x, y) and (y, z), Neo4j has to make sure it does not load the node y twice. The best to ensure that is to be build an index on the node identifiers, since it would save time in finding out whether or not that node is already existing in the database. Hence, the architecture of Neo4j forces the use of indexes.

These Neo4j indexes are also very useful when the actual handler for some node is required, given its ID. The actual node handler is an object of Neo4j.Node, which is necessary for representing a certain node/node category in a Cypher query. For instance, the following cypher query does gets the Person node, given the ID of the node (TODO check if this is what you wanted to convey):

"START n=node:People('id:" + str(node\_id) + "') return n"

## Query 1

### Loading data

#### Rudimentary approach

* The initial idea was to load all the data
* First, the person\_knows\_person graph is loaded into the Neo4j databae, building indexes over the person node identifiers
* Next, the comment\_hasCreator \_person graph (which stores what comment was made by whom) is loaded, building indexes over the comment node identifiers
* Finally, the comment\_replyOf \_comment graph (which stores what comments were in reply of a given comment) is loaded, building indexes over the comment node identifiers that were not created in the previous step

#### Problems

* The number of edges in the comment\_hasCreator\_person and comment\_replyOf\_comment datasets are immense and at the same time the large majority remain unused. Even batch loading that can load todo{x nodes or rels/sec on our machine}, took a long time.
* In our optimized approach for this query, the majority of the test cases (with k=-1) for this query do not use the comment nodes and relationships at all

#### Optimized approach

* The main approach is on-demand loading
* Here, all the comment nodes and relationships from comment\_ hasCreator\_person, comment\_replyOf \_comment are not loaded. Only those that are really needed are loaded. For example, while traversing some path in the person\_knows\_person graph, only those comment nodes made by the person nodes on that path are interesting

#### Tradeoff

* There is a clear tradeoff between loading all the data at the outset and on-demand loading. The approach that should be chosen would depend on the amount of queries that are expected to be made. If they are only few of them, on-demand is better, since the remaining data would be useless anyway. On the other hand, if there are many queries to be made or data has to stay for longer, the cost/time of initial loading would be amortized over all the queries and this would be less expensive than on-demand loading

### Querying data

#### Rudimentary approach

For the given start node and end node, paths between them (in the person\_knows\_person graph) are found with minimum edges in between. If the path found, it is verified that the path is indeed a frequent communication path. If yes, the path is the solution. If not, longer paths are found between those nodes that are frequent communication paths. The approach took prohibitively long amounts of time for some of the test cases, for instance where there is no frequent communication path between the given nodes

#### Problems

If the length of the shortest frequent communication path between two people is very long, this approach can take a very long time. This is mainly since the search space in which Neo4j tries to find paths increases exponentially with the addition of each edge. Additionally, in the case when there is no shortest path between the 2 nodes, Neo4j would have to look for any path of length 1,2,3,...,|V|-1, where V is a person node, only to return that no path actually exists. This is a prohibitively long search operation, and hence had to be optimized.

#### Optimized approach

A BFS traversal is performed on the person\_knows\_person graph, starting from the given start node, until the given end node is not encountered. The BFS traversal has to be such that only those outgoing edges from a node are considered that satisfy the frequent communication condition between the two nodes across that edge. If the BFS queue is empty at some point, no such path can be said to exist between the given start/end nodes.

While doing the BFS traversal, all the neighbors of the current node are obtained. For each neighbor, it is checked whether or not it's a frequent communication neighbor of the current node. To do this, we load the comment nodes for the current node and the neighbor (under the lazy-loading strategy); then check if both have made at least 'k' replies on each other's comments. If yes, the neighbor node is added to the BFS queue, otherwise not. When we find the given end node, the search terminates, and it's guaranteed that the path found from the given start node is indeed the shortest path, since unlike DFS, BFS finds the shortest path between nodes in a graph. On the other hand, if the BFS queue becomes empty at some point, we can be sure that there is no path between the nodes that satisfies the given criteria.

When k=-1, it means that an edge can be added to the path, even if the 2 people constituting the edge made no comments on each other's comments. This reduces the problem to simply finding the shortest path between the 2 input nodes, absent any constraints. This operation is very quick and inexpensive in Neo4j.

## Query 2

### Loading data

* First, the person\_knows\_person graph is loaded, indexing person nodes on a Lucene index for range queries
* Then, the person\_hasInterest\_tag graph is loaded, which involves building an index for the interest nodes, and adding an edge between a person's own node and his interest nodes

### Querying data

Person nodes in the graph are indexed using a Lucene Index for range queries, since people with birthday after a certain day are desired.

The Persons satisfying the given birthday condition are obtained by querying the Lucene index. For every Interest tag, the people among these that are associated with this particular Interest tag are obtained. A BFS traversal is performed on the Person nodes associated with each of the Interest tags, to find all the connected components in the graph induced by Person nodes with that Interest tag and edges from person\_knows\_person. The connected component with the largest size for each Interest node is stored. The top k Interest tags with the largest connected components is returned.

## Query 3

### Loading data

* First, the person\_knows\_person graph is loaded
* Then, the places nodes, that includes countries, cities and continents are loaded from place.csv
* Then, edges are added between place nodes to store what place lies within the other, from place\_isPartOf\_place file
* Then, edges between person nodes and place nodes that tell where a person lives, had studied or works are added from person\_studyAt \_organisation, person\_workAt \_organisation, organization \_isLocatedIn\_place files
* as always, an index exists for each category of nodes viz. people, place, interest tag

### Querying data

#### Rudimentary approach

* First, the nodes of all the places that exist within the given place node are found. For example, if the place given is Asia, all countries in Asia, and then all cities in those countries are queried
* Then, all people who stay in in the given place is obtained in a list P
* For each person in P, a BFS traversal is performed on the person\_knows \_person graph, up to a depth of h. With each new node being visited, the number of interest tags it has in common with the starting node is obtained. The tuple (start\_node, current\_node, common\_interests) is added to a list
* The list is sorted in descending order of the number of common interests i.e. on 3rd item of the tuples.
* The sorted lists resulting from the different BFS traversals with different starting nodes are merged into a larger sorted list and the top k tuples are picked. The couples represented by these tuples will have the maximum number of common interests, would be from the same place and would not be separated from each other by more than h edges in the person\_knows \_person graph

#### Problems

Doing a BFS traversal up to a depth of h, with each person node (satisfying place constraint) can very expensive and time consuming, especially if the value of h is large, or if the given place p happens to be a place with which many people are associated, like a continent.

#### Optimized approach

* After getting the person nodes satisfying the place constraint, for each pair of persons, the number of interest tags they have in common is found, and the tuple list (person1, person2, common\_interests) is sorted in a descending order by the 3rd attribute of the tuples.
* This sorted list of tuples is iterated over, while confirming whether or not person1, person2 in a tuple are no more than h hops away from one another on the person\_knows\_person graph. This can be performed very efficiently by using the shortest path query of Neo4j to get the minimum number of edges between the two nodes.
* k such tuples that satisfy the condition mentioned above are returned

#### Handling special cases

* While the approach works well in general, it takes a long time when the place given has a lot of people associated, for example, a continent like Asia.
* The part that takes really long is when each pair of persons has to be considered to find the number of their common interest tags. It takes long due to the fact that this operation in quadratic in the number of people living in the given place. Additionally, most of the pairs of people would have no interests in common at all, so checking all pairs is largely wasteful.
* Instead a hash table is constructed with key as the interest node and value as the list of people with that interest
* The hash table is then iterated over and the number of times a certain pair of person nodes co-occurs in the value part of the hash entry is kept track of i.e. in the people list of that interest tag.
* This ensures that the operation that was previously being done in quadratic time, can now be done in linear time i.e. in one pass of the hash-map mentioned above.

## Query 4

### Loading data

#### Rudimentary approach

* First, the person\_knows\_person graph is loaded
* Then, the interest tag nodes are loaded from forums.csv file
* Then, the edges between forum nodes and associated interest tags are loaded from forum\_hasTag\_tag file
* Finally, the edges between forum nodes and people who are its members is loaded from forum \_hasMember\_person file

#### Problems

* Just like the case of query1, the number of edges defining relationships between forum nodes and people nodes is huge and at the same time most remain unused.
* Loading all forum nodes is also unnecessary, when the nodes of use are only those that are specified in forum\_hasMember\_person and forum \_hasTag\_tag files

#### Optimized approach

* On-demand data loading is done for the data of this query, especially for creating nodes and relationships of forum\_hasMember\_person file.
* In this way of loading data, once all the forum nodes that are associated with the given interest tag are obtained, we need to load only those person-forum relationships (from forum \_hasMember\_person) that involve a valid forum node.
* This provides considerable savings in overall query time, especially for the medium sized data set.

### Querying data

#### Rudimentary approach

* All the forum nodes that are associated with the given interest tag are obtained
* All the people nodes that are members of the forum nodes found above are obtained and stored in list P
* Since the centrality of value of all the nodes in P is to be calculated, for which the number of nodes reachable from the given node and the sum of those path distances is required, the following can be done:
  + First, a BFS traversal is performed on the person \_knows\_person graph, while adding only those nodes to the BFS queue that are in P, until the queue is empty
  + While traversing a track, the number of hops at which a valid person node is encountered needs to be maintained.
  + At the end of the traversals of all nodes in P, centrality value for each node can be conveniently calculated.
  + The nodes in P are then sorted in descending order of the centrality values and the top k person nodes are returned

#### Problem

* The method takes prohibitively long amounts of time (as illustrated in the query timings section)
* A lot of rework was done in the method above. This is due to the fact that a lot of the same nodes and edges are encountered/traversed over runs of the BFS for different person nodes. At the same time doing independent BFS traversals for each node cannot be avoided, since the distances of other nodes in P need to be obtained from every other reachable node.

#### Optimized approach

* What the closeness centrality metric really needs is the number of edges (geodesic distance) existing between the various valid people nodes (that satisfy the forum-interest tag constraint) in the person\_knows \_person graph.
* This boils down the problem to finding the shortest path between all pairs of nodes in an un-weighted graph.
* The first solution that comes to mind is the All Pairs Shortest Path or Floyd-Warshall algorithm. However, since the person\_knows\_person is expected to be mostly sparse (since a lot of person nodes would not satisfy the forum-interest constraint), Dijkastra's single source shortest path algorithm is a faster alternate (although in worst case it would be outperformed by the Floyd-Warshall algorithm).
* This insight brings about notable improvements in the querying time

# Results

Please note that all times are given in seconds.

Below is the time taken for the one-time load for the small (1K) dataset. Results are contrasted against the naïve implementation.

The following numbers show the performance of query 1 for the small (1K) dataset.

|  |  |  |
| --- | --- | --- |
| **Case** | **Time (MT)** | **Time (Fin)** |
| **1** | 38.6770 | 0.1980 |
| **2** | 34.5740 | 617.0000 |
| **3** | 33.6690 | 0.0390 |
| **4** | 34.3120 | 0.0970 |
| **5** | 54.8840 | 816.9230 |
| **6** | 34.9470 | 0.0270 |
| **7** | 28.3530 | 0.0240 |
| **8** | 82.4440 | 1371.9690 |
| **9** | 31.8030 | 0.0280 |
| **10** | 46.3350 | 323.8860 |

The following numbers show the performance of query 2 for the small (1K) dataset.

|  |  |  |
| --- | --- | --- |
| **Case** | **Time (MT)** | **Time (Fin)** |
| **1** | 80.8550 | 64.0640 |
| **2** | 61.5350 | 57.0600 |
| **3** | 56.0460 | 50.8630 |
| **4** | 60.3880 | 44.5300 |
| **5** | 41.0670 | 38.1210 |
| **6** | 34.0410 | 31.4430 |
| **7** | 38.2730 | 27.0580 |
| **8** | 23.4510 | 20.4530 |
| **9** | 14.3300 | 13.7380 |
| **10** | 8.1090 | 7.9870 |

The following numbers show the performance of query 3 for the small (1K) dataset.

|  |  |  |
| --- | --- | --- |
| **Case** | **Time (MT)** | **Time (Fin)** |
| **1** | 473.3810 | 12.1390 |
| **2** | 1.9490 | 1.8880 |
| **3** | 0.4450 | 0.4450 |
| **4** | 0.3110 | 0.3300 |
| **5** | 0.1370 | 0.1430 |
| **6** | 0.1580 | 0.1330 |
| **7** | 0.1090 | 0.1250 |
| **8** | 0.0980 | 0.1080 |
| **9** | 0.0900 | 0.0840 |
| **10** | 0.1310 | 0.1370 |

The following numbers show the performance of query 4 for the small (1K) dataset.

|  |  |  |
| --- | --- | --- |
| **Case** | **Time (MT)** | **Time (Fin)** |
| **1** | 9849.7040 | 42.7480 |
| **2** | 6724.1700 | 39.9200 |
| **3** |  | 36.7740 |
| **4** |  | 35.4460 |
| **5** |  | 33.9790 |
| **6** |  | 35.0080 |
| **7** |  | 33.8510 |
| **8** |  | 32.2780 |
| **9** |  | 32.6950 |
| **10** |  | 34.7650 |

We now contrast load times between the 1K and 10K datasets for each query.

The following numbers show the performance of query 1 for the medium (10K) dataset.

|  |  |
| --- | --- |
| **Case** | **Time** |
| **1** | 1.101 |
| **2** | 164.763 |
| **3** | 0.029 |
| **4** | 0.044 |
| **5** | 0.364 |
| **6** | 0.062 |

The following numbers show the performance of query 2 for the medium (10K) dataset.

|  |  |
| --- | --- |
| **Case** | **Time** |
| **1** | 642.562 |
| **2** | 479.379 |
| **3** | 372.153 |
| **4** | 258.904 |
| **5** | 141.903 |
| **6** | 28.067 |

The following numbers show the performance of query 3 for the medium (10K) dataset.

|  |  |
| --- | --- |
| **Case** | **Time** |
| **1** | 335.686 |
| **2** | 0.184 |
| **3** | 0.092 |

The following numbers show the performance of query 4 for the medium (10K) dataset.

|  |  |
| --- | --- |
| **Case** | **Time** |
| **1** | 2293.663 |
| **2** | 2374.124 |