International Journal of Future Generation Communication and Networking
Vol. 10, No. 2 (2017), pp. 9-20
http://dx.doi.org/10.14257/ijfgcn.2017.10.2.02

Social Recommendation using Graph Database Neo4j：Mini Blog, Twitter Social Network Graph Case Study

Enzhi Zhang, Jinan Fiaidhi and Sabah Mohammed

Department of Computer Science, Lakehead University
**955 Oliver Rd, Thunder Bay, ON P7B 5E1, Canada*
{ezhang1, jfiaidhi, sabah.mohammed}@lakeheadu.ca

Abstract

We are currently living in the age of BigData and social networking where the generated data is mainly unstructured and disorganized. This nature of such newly generated data which are growing exponentially gives importance for relationships between entities and point to the importance of using graph databases. By following the relationships between the people and properties in a meaningful manner you can determine co-occurrences, frequencies, and relevant nodes in the graph. This is the basis for many recommendation engines especially the one used for real-time recommendation engines operating on fast growing social data like Twitter. Real-time recommendation engines are key to the success of any online business. To make relevant recommendations in real time requires the ability to correlate product, customer, inventory, supplier, logistics and even social sentiment data. Moreover, a real-time recommendation engine requires the ability to instantly capture any new interests shown in the customer's' current visit – something that batch processing can't accomplish. This case study is an attempt to use a graph database Neo4j; one of the NoSQL data model, to build a mini blog prototype in order to perform efficient social recommendation. The Neo4j cypher query language is used to analyze real life social network dataset imported from twitter. The mini blog prototype has been created a web application using python flask web framework.

Keywords*: Graph database, Neo4j, Python Flask, Social recommendation*

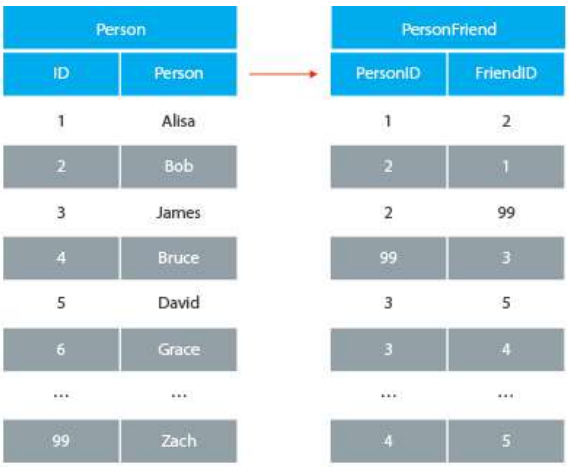
1. Introduction

We believe that any developer has met a series of very complicated design problems in the use of relational database. In relational databases data is stored as rows in a table. It looks like kind of Excel spreadsheet. Each table stores a specific category of data (*e.g.* Persons and Friends) and has a predefined list of a properties for every element in that table (*i.e.* columns). With this data representation one can generate relations (*e.g.* making the connections between Persons and their Friends). However, adding a new column in a table is something that adds new dimension of complexity. The table is going to expand both vertically and horizontally, making it really tough to manage where the majority of these fields are going to be null for most the entries. This is an inefficient use of space. More complicated relations can be created in the relational representation using the join or self-join operations between different tables[1] (see Figure 1 when applying the self-join operation on the Person table to produce a new relation of PersonFriend). In the relational database approach, friend relationship is stored using ID. In order that you look for friend relationship on the basis of this data, search all tables and then match IDs. However, relational database work well when the data is predictable and fits well into tables, columns, rows, and wherever queries are not very join-intensive. But there are rich,

[1] http://bitnine.net/rdbms-vs-graph-db/?ckattempt=1

ISSN: 2233-7857 IJFGCN

connected domains all around us including social and P2P networks, thesauri, route-planning systems, recommendation systems, collaborative filtering and the World Wide Web where the relational approach isn't so well equipped at dealing with. Given such new paradigms importance, it's really worth spending some time to find better representation of data as well as algorithms to work with them effectively. The start was the use of NOSQL wave database approaches (*e.g.* MongoDB, Cassandra, and Riak) which are designed to handle simple data. However, the most interesting applications deal with a complex, connected world. Such new relational complexity among data mandates the search and the development of new type of database that can significantly changes the standard direction taken by NOSQL. Graph databases, unlike their NOSQL and relational brethren, are designed for lightning-fast access to complex data found in social networks, recommendation engines and networked systems. A graph database[2] is a database that uses graph structures for semantic queries with nodes, edges and properties to represent and store data. A key concept of the system is the graph (or edge or relationship), which directly relates data items in the store. The relationships allow data in the store to be linked together directly, and in many cases retrieved with a single operation. Figure 2 illustrates representing the Friends relations in a graph database.



(a) The Person Relation (b) The PersonFriend Relation

Figure 1. Adding a New a New Column to a Relational Table

[2] https://en.wikipedia.org/wiki/Graph_database

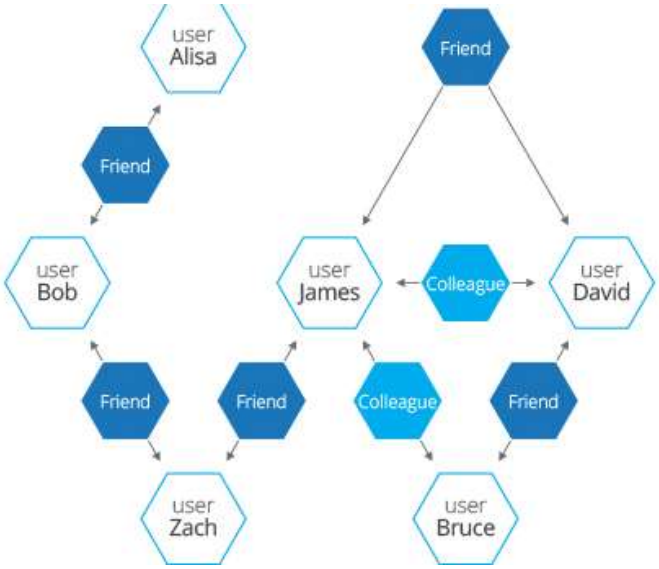


Figure 2. Representing the Friend Relations in a Graph Database

In a graph database, nodes and relationships (also called edges) are native to a graph database and this includes making sure that the graph is consistent (for example ensuring that every relationship has a start node and an end node). As a developer you can focus more on what to model and less on how to represent it. For example, to traverse a graph is to find the nodes that are directly connected to a node in the case of directed relationships. We assume the node variable as input and then the code is as easy as follows:

```
for ( Relationship rel : node.getRelationships( RelationshipTypes.FRIEND,
Direction.OUTGOING ) )
{
   Node otherNode = rel.getEndNode();
   // Here goes your code to make use of otherNode.
   // You can aggregate it or whatever you want to do.
}
```

These relationships can be uni-directional or bi-directional and can even contain properties specific to that relationship (see Figure 3).

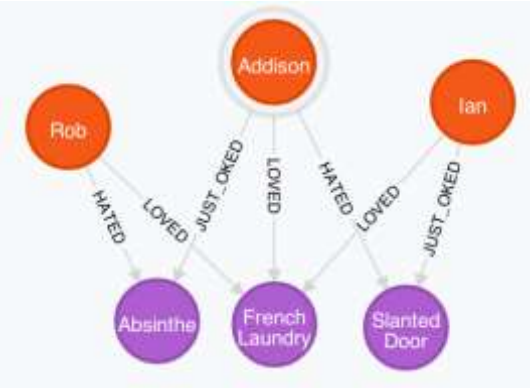


Figure 3. Graph Representation with Attributes

Comparing the performance of relational databases on graph analytics [1]. Here the leading graph database system (Neo4j [3]) compared to three relational databases as described in Figure 3: a row-oriented database (MySQL[4]), a column-oriented database (Vertica), [5] and a main-memory database (VoltDB [6]) are compared. Two queries, PageRank and Shortest Paths, on each of these systems. Considering two datasets from the Stanford large network dataset collection:

- A Facebook dataset having 4K nodes and 88K edges, and
- A Twitter dataset having 81K nodes and 1.8Medges

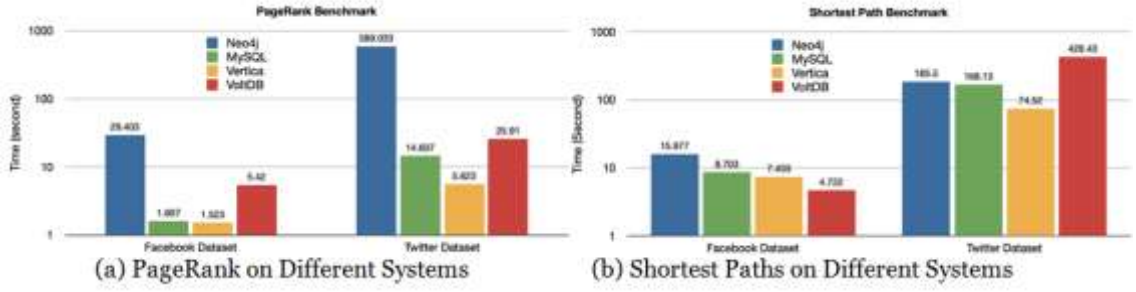


Figure 3. Comparing the Performance of Neo4j with three Relational Databases

Based on the findings from Figure 3 we can conclude that searching social networks like Twitter will far more efficient using Neo4j graph database. Twitter in particular creates a graph of Users, Tweets, Hashtags and shared Links which makes searching for relations like a local community very difficult with any relational database. Figure 4 illustrates the various relations Twitter generates when linking users and tweets.

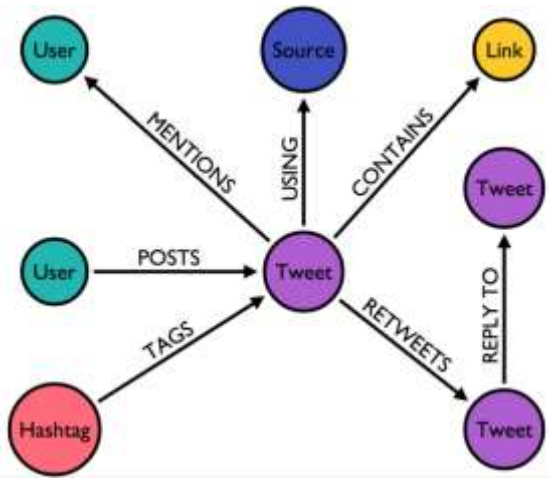


Figure 4. Representing Tweets Relations in a Graph Database

Graph databases such as Neo4j provide production grade front- or back-end social graph storage. Moreover, they offer graph analytics such as link prediction, shortest paths,

[3] https://neo4j.com/try-neo4j-sandbox/?utm_source=GPPC&gclid=CJarx6OX3NECFViewAodR5MJGA

[4] https://www.mysql.com/

[5] https://en.wikipedia.org/wiki/Vertica

[6] https://www.voltdb.com/

clustering coefficient, and minimum spanning trees, bolstering the potential of graph tools such as NetworkX or NodeXL, machine learning frameworks such as Graphlab, and distributed processing systems such as Spark [2]. Increasingly, more important information is being shared through Twitter. New opportunities arise to use this tool to detect and extract crucial information about the scope and nature of certain emerging events. A major challenge for the extraction of such event information from Twitter is represented by the unstructured and noisy nature of tweets which requires a graph based representation that can accommodates such complexity and be practical to be used for extracting such relations [3,4].

2. Developing Mini Blog Social App

The mini blog web app is built under Python Flask web framework, Flask depends on two external libraries: the Jinja2[7] template engine and the Werkzeug WSGI toolkit. Jinja2 is a modern and designer-friendly templating language for Python, modelled after Django's templates. It is fast, widely used and secure with the optional sandboxed template execution environment, it has some certain features to support the project [5,6]:

1) Built-in development server and debugger
2) Integrated unit testing support
3) RESTful request dispatching
4) Uses Jinja2 templating
5) Support for secure cookies (client side sessions)
6) 100% WSGI 1.0 compliant
7) Unicode based
8) Extensively documented.

Beyond these features, the most important part is the connection between the front-end and the graph database, which in this project will be using Neo4j, Neo4j is a high-performance, open-source NOSQL graphical database that stores structured data into graph instead of the tables. It is sponsored by Neo Technology and implemented in Java and Scala. Because of the advantages such as embedded, high-performance and lightweight, *etc*., Neo4j is getting more attention by the world.

The Neo4j developer manual [7] has indicate that "Neo4j implements the Property Graph Model efficiently down to the storage level. As opposed to graph processing or in-memory libraries, Neo4j provides full database characteristics including ACID transaction compliance, cluster support, and runtime failover, making it suitable to use graph data in production scenarios." All the data in Neo4j will be stored as an edge, a node, or a property. Each node and edge could have multiple properties. While processing a large number of complex, interconnected, low-structured data, the data traversal speed will not be influenced under this graph data model. Furthermore, Neo4j provides a unique SQL like Cypher query language, which has more straightforward syntax.

There is a list of features of Neo4j provided by the Neo4j developer manual that make Neo4j very popular among users, developers, and DBAs:

1) Materializing of relationships at creation time, resulting in no penalties for complex runtime queries.
2) Constant time traversals for relationships in the graph both in depth and in breadth due to efficient representation of nodes and relationships.
3) All relationships in Neo4j are equally important and fast, making it possible to materialize and use new relationships later on to "shortcut" and speed up the domain data when new needs arise.

[7] http://jinja.pocoo.org/

4) Compact storage and memory caching for graphs, resulting in efficient scale-up and billions of nodes in one database on moderate hardware.

Besides, Neo4j has provide powerful library Py2neo [8], which is a 3rd-party library for connecting to Neo4j from python. While the project started, the only way to communicate with a Neo4j server was over HTTP, which is impossible to dumping a lot of data to Neo4j at once. However, with the recently release of Neo4j 3, this technology now has a binary protocol that takes place over TCP, which it's now possible to dump data directly from python to the Neo4j database. Therefore, by simply import the Py2neo and set database URL to the Neo4j data browser, the mini blog will build connection n to the graph database. Same as most social network, the mini blog will support user register, login and share new post, all the HTML file will be create using Jinja2 template, the social recommendation will be demonstrating by using Neo4j CQL (Cypher query language), the Neo4j Developer Manual v3.1 is published by Neo Technology in 2016, and it provides detailed information about the CQL. Cypher is a unique query language for Neo4j which allows user for expressive and efficient querying and updating of the graph store. Cypher is a relatively simple language but very powerful. Even the most complicated database queries can be expressed easily through Cypher. This will save you from getting lost in database access and focus on your domain.

CQL has more straightforward syntax which all very complex queries in very easy manner.

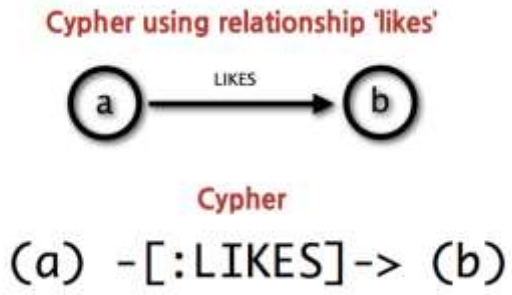


Figure 5. CQL Structure

Three recommendations will be built within the mini blog, which are the similar user by same tags, common user by the post and list most recent post by the time. And the graph data in the database will show as below:

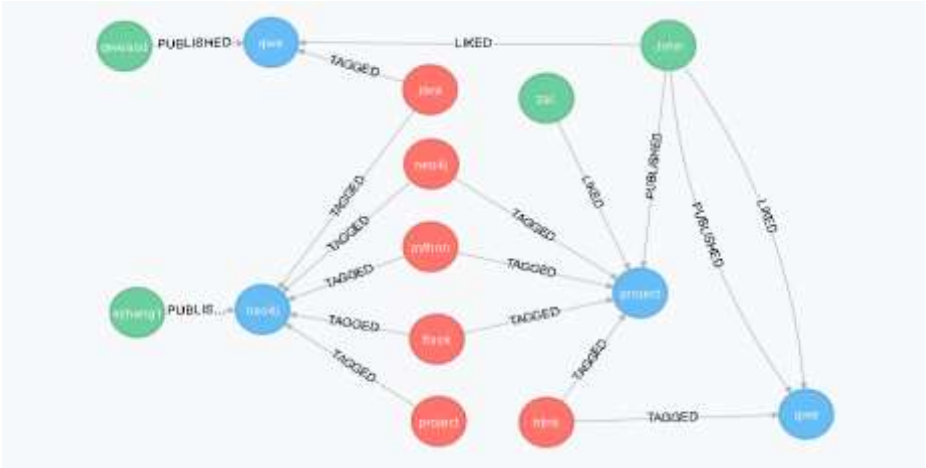


Figure 6. Mini Blog Graph Data

3. Extending the Prototype to Twitter Recommendation

In order to store the real twitter data into the graph database Neo4j, this project will use another Python Library Requests, refer to the documentation provided by Python, Requests is the only Non-GMO HTTP library for Python, safe for human consumption [6]. Requests allows user to send organic, grass-fed HTTP/1.1 requests, without the need for manual labor. There's no need to manually add query strings to your URLs depends on the will of the developers and the objectives of the project, or to form-encode your POST data. Keep-alive and HTTP connection pooling are 100% automatic, powered by urllib3, which is embedded within Requests.

The authorization is required in order to access the Twitter API, which contains the API Key and API secret, with a Twitter app account user will be allowed to generate the access token, and with the authorization this project will be able to connect to Twitter API and import data into the graph database. This project will access to the Twitter search engine and import the keyword data into Neo4j, by construct the URL to set certain rules for the data importing, the parameter of the rules will depend on the device and the user needs. The final step is to create nodes and relationships in the graph database after the scraper got a list of dictionary back from the Twitter API, in order to generate the data graph. The process flow of the method is shown as below:

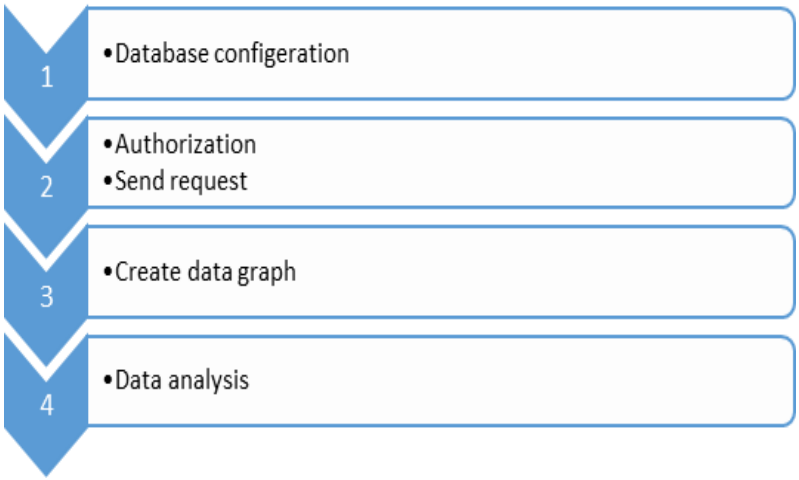


Figure 7: Process Flow of the Method

Following the authorization process, our blog allows the user to provide the arguments as a dictionary, using the params keyword argument, we could send required data in the URL's query string in order to constructing the URL, the data would be given as key/value pairs in the URL after a question mark:

```
def find_tweets(since_id):
                payload["since_id"] = since_id
                url = base_url +
"q={q}&count={count}&result_type={result_type}&lang={lang}&since_id={since_id}"
.format(**payload)
    r = requests.get(url, headers=headers)
    tweets = r.json()["statuses"]
    return tweets
```

Following that is to create graph through each of these dictionaries, by extracting the user and entities out of the dictionaries:

```
for t in tweets:
  u = t['user']
```

```
e = t['entities']
```

The graph tweets will be created first, merge function will be used to find or create the node tweet, the first argument Tweet is the node label, the id is property key and the third argument is the property value. Because every tweet need a unique id, it will give an id to the tweet, merge_one function is used just in case some tweets will show more than once, it is to avoid create two same tweets. By setting the text properties to text and push it in order to create the tweet:

```
tweet = graph.merge_one("Tweet", "id", t['id'])
tweet.properties['text'] = t['text']
tweet.push()
```

Then using merge_

one function to find or create the user node, because certainly user will tweet about something multiple times, the following code will find the user with the property username and screen_name value out of the user dictionary. Then by using relationship function to create the relationship that indicate the user post the tweet:

```
user = graph.merge_one("User", "username", u['screen_name'])
graph.create_unique(Relationship(user, "POSTS", tweet))
```

Next step will go through the entities dictionary which has a key hashtag, it will give back a list of hashtag that tag the tweet, and also use the merge_one function to avoid create the hashtag nodes more than once, find or create the hashtag with hashtag as the node label, name as the property which is equal to a lower case version of the hashtag, then create the relationship for each of the hashtag that the hashtag tags the tweet:

```
for h in e.get('hashtags', []):
    hashtag = graph.merge_one("Hashtag", "name", h['text'].lower())
    graph.create_unique(Relationship(hashtag, "TAGS", tweet))
```

Also we will capture mentions with the same function and create relationship for the mentions that the tweet mentions other users:

```
for m in e.get('user_mentions', []):
    mention = graph.merge_one("User", "username", m['screen_name'])
    graph.create_unique(Relationship(tweet, "MENTIONS", mention))
```

Second last, we will get the tweet that potentially will reply to, it will refer to the tweets that reply to other tweets, to get the reply if there are any we need to find or create the tweet by the unique id then create the relationship indicate the tweet reply to other tweet:

```
reply = t.get('in_reply_to_status_id')
if reply:
    reply_tweet = graph.merge_one("Tweet", "id", reply)
    graph.create_unique(Relationship(tweet, "REPLY_TO", reply_tweet))
```

Then if exist a tweet is a retweet of other tweet, we will also get the retweet return with the Twitter API as retweeted_status, then extract the id out and find or create the tweet by its id property, follow by create the relationship that the tweet retweet the other tweet if it exist:

```
r = t.get('retweeted_status', {})
r = r.get('id')
if r:
    retweet = graph.merge_one("Tweet", "id", r)
    graph.create_unique(Relationship(tweet, "RETWEETS", retweet))
```

Until now, the entire schema of the script has explained clearly, before start collecting data into Neo4j, we need to create some unique constrains using Py2neo create unique constrain function, the method is to make tweets unique by id, users unique by username and hashtag unique by name, it will also create an index on the node label property pair:

```
def constraint(label, property):
  if property not in graph.schema.get_uniqueness_constraints(label):
```



```
    graph.schema.create_uniqueness_constraint(label, property)
constraint("Tweet", "id")
constraint("User", "username")
constraint("Hashtag", "name")
```

Applications which process a timeline, wait some quantity of time, and then need to process new Tweets which have been added since the last time the timeline was processed can make one more optimization using the since_id parameter. So by steeing a since_id and update it in order to avoid finding same tweets is required, the last part is to find tweets by the keyword that specified in the command line, and send the list of dictionary to upload tweets, warn if there is no match tweets found and then sleep the Twitter API after each update:

```
since_id = -1
while True:
    try:
        tweets = find_tweets(sys.argv[1], since_id=since_id)
        if not tweets:
            print("No tweets found.")
            time.sleep(60)
            continue
        since_id = tweets[0].get('id')
        upload_tweets(tweets)
        print(str(len(tweets)) +  " tweets uploaded!")
        time.sleep(60)
    except Exception as e:
        print(e)
        time.sleep(60)
        continue
```

The resulted data will be store as a graph into Neo4j, the final process to the data is using CQL to create recommendation Cypher. View the database via the Neo4j data browser, and use the Cypher below to view all the data:

```
MATCH (n) RETURN n
```

The data graph is shown as below:

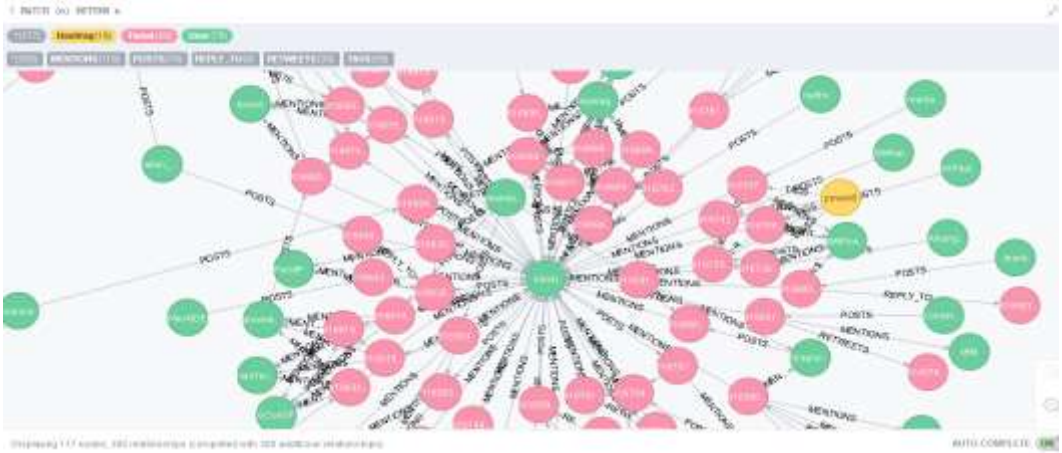


Figure 8. Twitter Graph Data

Normally, the social recommendation will base on what kind of data that user wants to get from the database, therefore the creation of the recommendation Cypher should start with asking questions, then translate the questions into Cypher, run it in the database and get the required data. For example: the question “Which has the top mentions of Neo4j?”

```
MATCH
(u:User)-[:POSTS]->(t:Tweet)-[:MENTIONS]->(m:User {username:'Neo4j'})
WHERE   u.username <> 'Neo4j'
RETURN  u.username AS username, COUNT(u.username) AS count
ORDER BY   count DESC LIMIT 10
```

username	count	u
jeanepaul	10	{username: jeanepaul}
mpyeager	3	{username: mpyeager}
importio	2	{username: importio}
sharmagourav	2	{username: sharmagourav}
FergusInLondon	1	{username: FergusInLondon}
tnarik	1	{username: tnarik}
marc_data	1	{username: marc_data}
IBMPowerSystems	1	{username: IBMPowerSystems}
jimwebber	1	{username: jimwebber}
riz_emba	1	{username: riz_emba}

Figure 9: Top Mentions of Neo4j

4. Conclusions

Due to the rapid development of the network performance, we started this case study to test the Neo4j graph database performance in both large and small range social networks. The mini blog web app indicate that the advantage of graph database is not only appears on the performance, the clear model and simple method makes the entire database extremely easy to manage. The recommendation is functional but with more entities import into the system, the power of the graph recommendation will appear better. From the Twitter data analysis, we have built social recommendations that uses the Twitter graph database we found that without joint tables the relationships and nodes traversing takes barely no time.

Acknowledgments

This case study is part from an MSc project under the supervision of Dr. J. Fiaidhi.

References

[1] S. Patil, G. Vaswani and A. Bhatia, "Graph Databases-An Overview", International Journal of Computer Science and Information Technologies, vol. 5, no. 1, **(2014)**, pp. 657-660, http://www.ijcsit.com/docs/Volume%205/vol5issue01/ijcsit20140501141.pdf
[2] Robinson, Ian, J. Webber and E. Eifrem, "Graph databases: new opportunities for connected data", O'Reilly Media, Inc., **(2015)**.
[3] Drakopoulos, Georgios, A. Kanavos and A. Tsakalidis, "Evaluating twitter influence ranking with system theory", In Proceedings of the 12th International Conference on Web Information Systems and Technologies, WEBIST, **(2016)**.
[4] Klein, Bernhard, X. Laiseca, D. C. Mansilla, D. López-de-Ipiña, and A. P. Nespral, "Detection and extracting of emergency knowledge from twitter streams", In International Conference on Ubiquitous Computing and Ambient Intelligence, **(2012)**, pp. 462-469.
[5] J. Webber, "A programmatic introduction to neo4j", In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity, **(2012)**, pp. 217-218.
[6] A. Ronacher, "Welcome to Flask", **(2015)**, https://media.readthedocs.org/pdf/flask-russian-docs/0.9/flask-russian-docs.pdf
[7] Neo Technology, The Neo4j Developer Manual v3.1, **(2017)**, https://neo4j.com/docs/developer-manual/current/
[8] Neo Technology, The Py2neo 2.0 Handbook, **(2017)**, http://py2neo.org/2.0/
[9] K. Reitz, "Requests: Http for humans", Online: http://docs. pythonrequests. org/. (24 December, 2012.) (2014).

Authors

Enzhi Zhang, he is a MSc Graduate Student at the Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada.

Jinan Fiaidhi, she is a Professor and Graduate Coordinator, Department of Computer Science, Lakehead University, Thunder Bay Ontario, Canada. Dr. Fiaidhi research is on Social Networking, Machine Learning, Big Data and Ubiquitous Computing.

Sabah Mohammed, he is a Professor with the Department of Computer Science, Lakehead University, Thunder Bay, Ontario, Canada. Dr. Mohammed research is on Health Informatics, Data Science, Web Intelligence and Big Data Security.

