Clustering People in Social Network Based on Graph Databases

Wilhelm Magnusson, Joel Gärtner, Yiming Fan,
Sepideh Shamaei, and Mahmut Uğur Taş

KTH Royal Institute of Technology,
SE-10044, Stockholm, Sweden
{wilhelmm,jgartner,yimingf,shamaei,mutas}@kth.se

Abstract. We have found efficient ways of structuring and searching through clusters of Facebook event data in order to find connections between people with similar interests. This was done by fetching data through Facebooks public API and testing and optimising the data structure for the algorithms described in the paper GraphSystems-tutorial. The output has DBMS form that can find people based on how similar their interests are using a graph database and a small sample size of peoples event data from Facebook.

1 Introduction

The everyday use of social network has grown explosively. Therefore, it is an interesting mission for data analysers to retrieve useful facts from the massive data. Traditional RDBMS are relatively ad-hoc and computation consuming, and they have been replaced by light-weight and efficient `NoSQL` database systems. One widely adapted database system, the graph database, could represent and store massive data with the structure of nodes and edges. This hierarchical structure allows retrievals which are relatively rapid, comparing to conventional database systems. Its performance surpasses most conventional database systems when handling with graph-like queries. [3]

Clustering users is an important task in social network data mining. Users may be recognised with respect to their ages, locations, occupations, marital status, and/or interests. Some machine learning methods, for example, Self-Organizing Maps (SOM) are proved to efficiently cluster people at various scales. Our project group, on the other hand, has implemented some clustering method based on graph queries. It utilises the advantages of graph data structure among various users, and has a relatively fair scoring scale. We also implemented a comparing group with the form of conventional RDBMS, in order to compare their performance.

2 Method & Implementation

Facebook data

First of all, data was required to be fetched with the Facebook API. This was done with the help of a `python` module called `facepy` which allowed calls to the API from within `python`. In order to get interesting data from this, the solution was to search for events located within a 20-km radius from a point in Stockholm. The persons used in the experiment was then the persons which attended these events and the only events used were the ones found in this way.

As a result, we have fetched 2000 events, 73981 persons and in total 140713 attending relationships in the databases. This means that for the fetched data each person was in average in about two of the events which were fetched.

Graph database

The graph database used was the community edition of `Neo4j`. In order to make queries and make graphs in `Neo4j` the `cypher` language is used. This was used to load the data and create a graph which had two types of nodes, namely *events* and *persons*. A relation from persons to events called *Attending* was also created for each persons to all events they attended. With this graph in place it was, once getting familiar with the language, relatively easy to do the desired query for a person with a given *id*. For the person with *id* 123 this was done with the command

```
MATCH (p:Person {id : 123})-->(e:Event),(e)<--(q:Person)
RETURN q,Count(q) AS nr ORDER BY nr DESC;
```

which simply finds all persons which the person has been on events with and orders them by how many events they have attended together.

SQL

An equivalent query was implemented in `sql` in order to get a comparison with a traditional RDBMS. In `sql` the data was loaded into three tables, *persons*, *events* and *attending* where the *attending* table was a many-to-many relation between *persons* and *events*. In order to construct the query it was simply to do a join between two copies of the *attending* table over equal event *id*s. By selecting all entries in this joined table for which one of the copies of *attending* has a specific user *id* we will all persons who have attended events with that persons. By simply counting the number of occurrences of a specific person this will give the same result as from the graph database.

3 Results

Query results

Figure 1 shows an example for how the graph looked in the web-interface. In this specific graph it shows all events the person in red have attended as well as all persons who attended these events, together with all the relations. Some comparisons were however performed to ensure that the data from the different queries was the same.

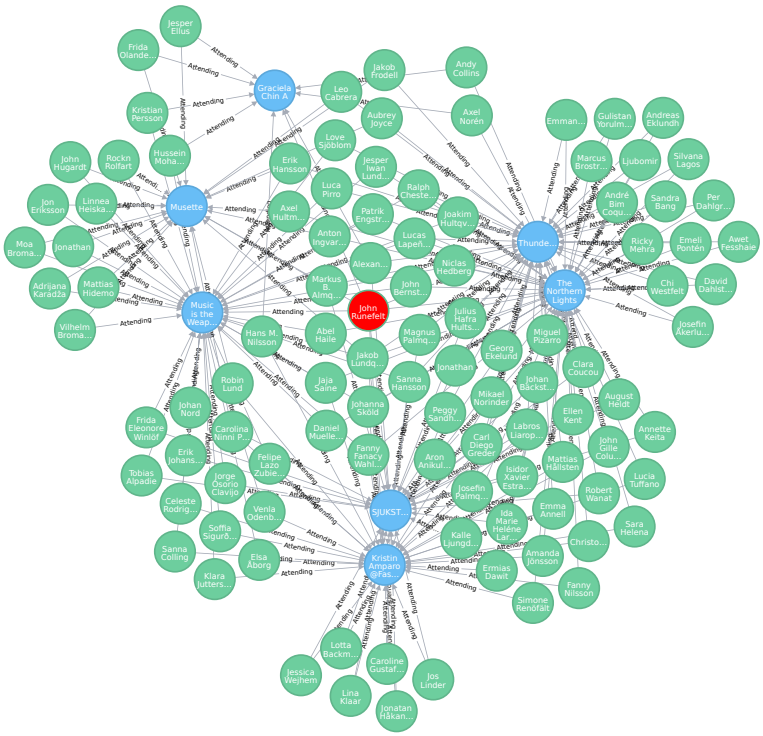

Fig. 1. An example of the resulting graph

Performance

Some tests were performed on the performance of the implementations in the different database structures. Initial testing was performed using the `Neo4j` web-interface and direct commands to an `sqlite` database. This gave the result that the graph database was quite slower than the standard relational database. A possible factor in this was however probably some added overhead in the web-interface and the time spent was still mostly used for the output of the results.

In order to get better results a Java-program was developed which used `sqlite` and `Neo4j Java` API:s. This program could then run the queries and time them in different ways.

Trying this it however became clear that there was no obvious way to objectively measure the time for running the query. One reason for this was that the queries were constructed so that they picked out N persons and found the persons they had been on events with. The problem was then that the N persons which `Neo4j` picked and the N persons which `sql` picked were not the same. In order to still get interesting queries several queries were performed. We first did the "native" query which resulted in the different databases getting different results as they selected different persons initially. For the rest of the queries we instead first selected the users from `sql` and saved them in a list in Java. This list was used mainly for a loop where each person selected was iterated over and then sent to a query for the respective languages. It was also used directly with the `Neo4j` API as it was possible to supply iterable `Java` objects to it which would result in a collection which was usable in the `cypher` language. This was used to directly insert the list of selected users into the `cypher` query. The run-times for these queries for different values of N is shown in table 1. Worth noting is that in all runs of the native version the `Neo4j` query resulted in more result lines which is shown in table 2. The running times are rounded to the nearest millisecond but in truth the running time varied by a lot more than that so the numbers are only indicators of behaviour and not average run times over multiple attempts.

N	1	10	100	300
`Neo4j` native	1416	585	900	2685
SQL native	213	482	549	859
`Neo4j` loop	76	1644	4535	5829
SQL loop	207	1885	18510	54713
`Neo4j` Java collection	1143	3243	22937	65709

Table 1. Running times in ms for different N

N	1	10	100	300
`Neo4j`	7232	30266	52115	281550
SQL	170	9102	71090	194833

Table 2. Number result lines for different N for native approach

4 Discussion

Surprisingly, the results above slightly object to what `Neo4j`'s official website says:

depth	Neo4j	RDBMS
1	....	.....
2	0.01	0.016
3	0.168	30.267
4	1.359	1543.505
5	2.132	unfinished

Table 3. Performance difference (in seconds) on multiple join query test. Claims from the book *Neo4j in Action*, chapter 1, table 2-1

However, we are not the only few who are curious about the difference between the claim and the result. Jörg Baach [2] has doubted, that to achieve the dominating performance of Neo4j over mysql, one must set his/her operating environment carefully. However, environment issue could not confine a web developer in performance that much, or it disobeys the principle of reusability. Consider a project whose speed is stuck just by changing another computer. Maybe Neo4j is quicker under some certain test cases, but that is not the real use-case.

So could we end up discussing and say, "gee, graph database sucks, let's roll back to traditional database"? The answer is far from true. In Jouili and Vansteenberghe's research [3] we found out, that Neo4j shares the similar performance with other graph database systems, such as DEX and Titan, in read-only workload, but would have a sharp performance degrade on read-write workload. Perhaps Neo4j is old-fashioned, high environment-demanding and slow, but we could have plenty of alternatives.

5 Summary

We presented Facebook friends clustering, using graph database system and traditional relational database. However, this real-life test case did not make a testimony that graph database is somehow great. This may due to incorrect computer configuration or bad computer environment, but considering the not-so-different personal-use computer capability, we're afraid that the very thing left to blame could be the improper choice of the graph database system. In the future we could do more test on different graph database systems to test their performance.

References

1. Vukotic, A., Partner, J., Watt, N., Abedrabbo, T., Fox, D.: Neo4j in Action. Manning, Greenwich (2014)
2. Baach, J.: neo4j performance compared to mysql, http://baach.de/Members/jhb/neo4j-performance-compared-to-mysql
3. Jouili, S., Vansteenberghe, V.: An empirical comparison of graph databases. In: Social Computing (SocialCom), 2013 International Conference on, pp. 708–715. IEEE Press, Alexandria (2013)