Implementing Link-Prediction for Social Networks in a Database System

Sara Cohen
Dept. of Computer Science and Engineering
Hebrew University of Jerusalem
sara@cs.huji.ac.il

Netanel Cohen-Tzemach
Dept. of Computer Science and Engineering
Hebrew University of Jerusalem
nati.ct@mail.huji.ac.il

ABSTRACT

Storing and querying large social networks is a challenging problem, due both to the scale of the data, and to intricate querying requirements. One common type of query over a social network is *link prediction*, which is used to suggest new friends for existing nodes in the network. There is no gold standard metric for predicting new links. However, past work has been effective at identifying a number of metrics that work well for this problem. These metrics vastly differ one from another in their computational complexity, e.g., they may consider a small neighborhood of a node for which new links should be predicted, or they may perform random walks over the entire social network graph. This paper considers the problem of implementing metrics for link prediction in a social network over different types of database systems. We consider the use of a relational database, a key-value store and a graph database. We show the type of database system affects the ease in which link prediction may be performed. Our results are empirically validated by extensive experimentation over real social networks of varying sizes.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Experimentation, Performance

Keywords

link prediction; social networks; database backends

1. INTRODUCTION

Social networks are a huge and fast growing data source. Analyzing this fascinating and multi-faceted data is of interest, and can be of great practical use. Thus, social network analyses is used in varied fields, including sociology, psychology, political science and economics.

Given both the size of social network datasets, and the allure of their data, storing and querying social networks is a problem of growing interest within the database community. There has been recent work on designing databases for social networks [2,6,7,9]. These works mostly deal with issues such as query and update languages, but not with the nitty-gritty of actually implementing a social network database system. In [8], on the other hand, the focus is on the type of storage that should be used for a social network database system. In particular, they describe five storage system types, based on eight current open source storage systems solutions. [8] discusses the main features of each solution, in order to analyze, on paper, their application potential. However, neither [8], nor the other papers mentioned earlier, implement and experimentally test the viability of different types of storage systems for use with social network data.

Implementing a database system for social network data is a huge challenge. In particular, as [2] pointed out, standard query languages are not capable of expressing important and common social network queries. As one example, a *link prediction* [5] query asks, for a given node x, which nodes in the network that are not currently friends with x are likely to become friends with x in the future. Link prediction is the basis of friend recommendation in a social network, and is an area of active research. Another example of a query of interest over a social network dataset is that of *team formation* [3], which attempts to form teams of people who are experts in a certain area and have minimal communication cost. Neither of these queries lends itself to expression in a standard, precise, query language. Rather, advanced algorithms and metrics are required to answer such social network queries.

The unique nature of social networks, and their querying requirements begs the question: *What type of storage solution should be used as a backend for a social network database system?* This paper takes a first step towards answering this question by considering the problem of implementing link prediction within a database system. We chose to focus on link prediction for two reasons. First, link prediction is a standard operation for social networks, has been extensively studied, and thus, effective metrics of interest have already been developed for link prediction. Second, and more importantly, metrics for link prediction differ vastly one from another in their computational complexity, e.g., they may consider a small neighborhood of a node for which new links should be predicted, or they may perform random

DBSocial '13 New York, NY USA

walks over the entire social network graph. Hence, implementing several different metrics for link predication sheds light on how the performance of social network queries is affected both by the choice of a storage system, and by their own inherent intricacies. We emphasize that our focus is not on comparing the effectiveness of link prediction methods (which is the topic of previous work), but rather only on comparing the efficiency of implementation in different storage systems.

In this paper we explore three popular paradigms for storage of data—a relational database (MySQL), a key-value store (Redis) and a graph database (Neo4J), as well as seven metrics for link prediction, and compare the performance of the metrics over the different systems. We highlight where features of the database systems (or lack thereof) significantly impact performance. As such, our results are a meaningful step towards developing a storage system for social network data.

This paper is organized as follows. In Section 2 we precisely define the link prediction problem and present seven popular metrics for use in solving this problem. In Section 3, we discuss how each of the metrics was implemented over the different data storage systems. Experimental performance results appear in Section 4. Finally, Section 5 concludes. We note that all code and experiments are available online.[1]

2. LINK PREDICTION

Link prediction is a classic and well-studied problem over social networks. Formally, the *link prediction problem* is: Given a snapshot of a social network at a given point in time, predict which edges will be added to the network as it naturally evolves. This problem comes in two flavors. The *global link prediction problem* is to find the k edges most likely to be added to the network, while the *local link prediction problem* is to find the k edges most likely to be added incident to a specific node x. This paper considers the only local version of the link prediction problem, due to lack of space, and because it is perhaps more important for friend recommendation.[2] In our experiments, we usually choose k to be 100.

The ability to predict links is considered important for friend recommendation, i.e., as predicted links are likely to represent natural relationships in the network. Link prediction is also useful for inferring missing links, i.e., discovering which links are surprisingly missing (and then the social process leading to this formation can be further analyzed).

Many score functions have been developed for predicting links. In general, a score function is defined over pairs of nodes, and indicates the likelihood of a link forming between these nodes. As of now, no clear winner for the link prediction problem has been found. Instead, different score functions have proven useful over varying datasets. As we will see later, score functions may differ substantially in their ease of implementation and in their efficiency.

Below we detail the score functions considered in this paper. These particular functions were chosen for two reasons. First, they are the "bread-and-butter" of link prediction, i.e., they are widely considered and studied. Second, their implementation requires a variety of types of accesses to the social network graph (only to neighbors, iterating over paths or random walks). Hence, they can perhaps shed light on how efficient other common graph operations would be over social networks (in different system implementations).

In total, we considered seven score functions for link prediction. All of these functions are also discussed in detail in [5]. In the following, we assume that x and y are nodes in a social network. We use $N(x)$ to denote the set of all neighbors of x. Finally, we use $P^l(x, y)$ to denote the set of all paths of length l between x and y.

- **Common neighbors:** The common neighbors cneigh function measures the number of neighbors in common to x and y, i.e.,
$$\text{cneigh}(x, y) = |N(x) \cap N(y)| .$$
- **Jaccard coefficient:** The Jaccard coefficient is slightly more sophisticated than common neighbors, as it also takes into considering the total number of neighbors that x and y have. This is significant, since nodes with many neighbors are more likely to have neighbors in common. Formally, the Jaccard coefficient is defined as
$$\text{jacc}(x, y) = \frac{|N(x) \cap N(y)|}{|N(x) \cup N(y)|} .$$
- **Adamic-Adar:** The Adamic-Adar measure [1] considers nodes to be related if they have common neighbors, but weights the importance of a common neighbor z proportionately to the number of neighbors of z. This gives more weight to rarer common neighbors, i.e., to nodes z that have few neighbors themselves, as follows
$$\text{aa}(x, y) = \sum_{z \in N(x) \cap N(y)} \frac{1}{\log |N(z)|} .$$
- **Preferential attachment:** In this score function, popular nodes (i.e., those with many neighbors) are considered more likely to form relationships one with another
$$\text{pref}(x, y) = |N(x)| \times |N(y)| .$$
- **Graph distance:** Unlike all previous measures, that only directly considered the neighbors of x and y, graph distance takes into consideration the relationship between nodes that may be far away. Thus, this scoring function $\text{dist}(x, y)$ is simply the the length of the shortest path between x and y.
- **Katz measure:** The Katz measure [4] takes into consideration all paths between x and y, but these paths are weighted by their length. In general, katz is computed as
$$\text{katz}(x, y) = \sum_{l=1}^{\infty} \beta^l |P^l(x, y)| ,$$
where $0 < \beta < 1$. We used a β value of 0.1. For efficiency purposes, we only summed until paths of length 3.

[1] `http://www.cs.huji.ac.il/~sara/link-prediction.html`

[2] However, experimental results with global link prediction problem are available online, along with our code.

- **Rooted PageRank:** PageRank computes a general importance value for a node by measuring the stationary distribution weight of a node x under a random walk, where with probability $1-d$ one jumps to a random node in the graph, while with probability d one goes to a random neighbor of the current node. Rooted PageRank adapts this measure to compute the importance of a node y with relationship to a node x. Thus, $\mathrm{rPR}(x, y)$ computes the stationary distribution weight of a node y under a random walk, where with probability $1-d$ one jumps back to x, while with probability d, one proceeds as before (i.e., to a random neighbor). In our implementation we chose $d = 0.85$ as our *damping factor*.

Note that for all measures other than graph distance, a higher value for the scoring function indicates greater likelihood of link formation, while for graph distance, the opposite is true.

3. IMPLEMENTING LINK PREDICTION

We considered three types of storage systems for the social network data, over which link prediction would be implemented. First, we consider relational databases, which have the advantage of many years of research and development, and hence, are often highly efficient. Second, we considered a key-value store, as such data storage systems are growing rapidly in popularity. Finally, we considered a graph data storage system, in hopes that the fact that the data model of the storage system precisely matches that of the actual data, would speed up data processing. As specific instantiations of these types of systems, we used MySQL[3] (relational database), Redis[4] (key-value store) and Neo4J[5] (graph storage). These implementations were chosen because of their popularity, since they are open-source (allowing us to publish performance data) and because of their purported efficiency on data sets of the size considered in this paper.

REMARK 1. *As discussed later in Section 4, we only considered social networks of up to several hundred thousand nodes and several million edges. (Even for such reasonably sized social networks, link prediction can be quite time-consuming.) Thus, it was natural and possible to use a single computer to store the social network. This paper does not consider really humongous social network databases that must be distributed over large computer clusters, and hence, storage systems that were specifically designed for such data distribution were not examined.*

In this section we show how each of the link prediction metrics from Section 2 can be implemented over each of the three data storage systems considered. In each system, effort was made to take advantage of the available features in order to yield an efficient implementation.

3.1 Relational Database: MySQL

MySQL is a popular multi-platform, open-source relational database management system, sponsored by Oracle. MySQL has multiple options for configuration, and effort was made to find a configuration that would work well with the data.[6] The link prediction metrics were implemented as stored procedures, in order to avoid communication costs that would arise from a back-and-forth of computation called from within a program external to the database.

The social network was stored within a table of `edges`, i.e., each row in the table contained the identifiers of two nodes which are connected by an edge. Indices were created over each of the columns of this table. In addition, several "helper" tables were created to help speed up computation.[7] These tables were of reasonable size in relation to that of number of nodes in input, and hence, could be used in practice. In particular, we generated: (1) a `neighbors-count` table (x, c) containing the number c of neighbors per node x and (2) a `popular-nodes` table containing the 100 nodes with the largest number of neighbors. The latter table was used to compute the preferential attachment metric, as preferential attachment always prefers more popular nodes.

Many of the metrics were computed by simple SQL queries. Specifically, common neighbors, Jaccard coefficient, preferential attachment and Adamic-Adar were simple queries, (self-)joining over the tables defined above. We note that for all of these metrics other than common-neighbors, the helper tables were necessary to get reasonable performance.

The other metrics required significantly more computational effort. Graph distance was implemented as a recursive procedure starting at the node x (for which potential new neighbors should be predicted) which traversed the social network graph in a BFS fashion, using queries, until the desired number k of nodes were found. Our implementation of the Katz metric used a procedure to calculate $|P^l(x, y)|$, for all y of distance at most three from x, and stored this information in a temporary table, that was later used to compute the actual Katz metric. Finally, our implementation of rooted PageRank iteratively looped over a temporary table to compute the stationary distribution of the weights of the nodes, terminating when the algorithm converged on the top-10 scoring nodes, and their relative order.

3.2 Key-Value Store: Redis

Redis is an open-source, in-memory key-value store database system, sponsored by VMware. Intuitively, a key-value store is a huge, highly efficient hashtable. Redis allows complex data structures as values (allowing sets of values to be stored for a given key). This was useful, as it enabled us to store the adjacency list of a node as the value for its identifier. Thus, our Redis database contained, for each node identifier (i.e., key) the set of all its neighbors (i.e., value). We also created a "helper" database (i.e., key-value store) that contained the list of the 100 nodes with the most neighbors, for use in computing preferential attachment.

We note that Redis supports in-database Lua[8] scripts within its atomic operations, which help to speed up processing. As in MySQL, common neighbors, Jaccard coefficient, preferential attachment and Adamic-Adar are relatively simple to implement, and involve some traversals and comparisons over adjacency lists. Note, however, that Re-

[3] `http://www.mysql.com/`
[4] `http://redis.io/`
[5] `http://www.neo4j.org/`

[6] `http://www.oracle.com/partners/en/knowledge-zone/mysql-5-5-innodb-myisam-522945.pdf`
[7] We considered a static social network, and thus, it was not necessary to update these helper tables. In a dynamic social network, the cost of such updates would have to be considered.
[8] `http://www.lua.org/`

dis lacks a query language, and hence, the implementation of these measures requires programming effort. Graph distance, Katz and rooted PageRank were computed in a manner that was similar, conceptually, to the implementation in MySQL (i.e., BFS traversals for the first two and run until convergence for the last). However, again, the difference in the storage model, required special consideration during the implementations.

3.3 Graph Data Store: Neo4J

Neo4J is a multi-platform, open-source, graph database supported by Neo Technology. The underlying data model is a graph, and hence, precisely matches that of a social network. Neo4J supports the Cypher query language. Cypher is a declarative query language for graphs, which supports pattern matching and filtering on properties. It has a syntax that is in the spirit of SQL. For example, to find nodes of precisely distance 2 from node number 17, one can use the following query:

```
START x=node(17)
MATCH x-->y-->z
WHERE not ( x-->z )
RETURN z
```

Neo4J uses Apache Lucene[9] to index nodes and relations, and leverages all of Lucene's features (including keyword search). We used this form of indexing to optimize the computation of the preferential attachment metric. In additional to the obvious (the graph structure), we also stored the number of neighbors per node as an attribute of the node (as Neo4J does not provide an efficient way to access this data).

The common neighbors and Jaccard coefficient metrics where implemented in the natural fashion. Preferential attachment used the previously defined index. Unfortunately, due to a lack of support for mathematical functions (including the logarithm function) in Cypher, we could not implement the Adamic-Adar metric.[10] The lack of mathematical functions, also made implementing the Katz metric more difficult (due to the lack of support for exponents), and thus, the computation required computing several queries and combining the results. The graph distance metric was computed by iteratively running Cypher queries, and rooted PageRank, similarly looped until convergence.

REMARK 2. *Although this paper does not address the global link prediction problem, we note that deriving efficient implementations of such functions over Neo4J is quite difficult, as Cypher seems to have serious performance issues when non-trivial pattern queries are applied globally over a graph.*

4. EXPERIMENTS

In our experiments, we considered 10 undirected graphs. Graph sizes appear in Table 1. The smallest graph considered had approximately five thousand nodes and thirty thousand edges, while the largest had more than 350 thousand nodes and 4 million edges. Our datasets are of three types:

[9] `http://lucene.apache.org/core/`

[10] It is possible to program directly in Java to circumvent such lack of mathematical functions. However, this is quite cumbersome.

Type	Name	# Nodes	# Edges
Coauthorship	**dblp-all**	**366,600**	**4,349,796**
	dblp-2010-2012	248,695	2,589,320
	dblp-2002-2009	182,493	1,621,846
	ca-CondMat	23,133	186,936
	ca-AstroPh	18,771	396,160
	ca-HepPh	**12,006**	**237,010**
	ca-HepTh	9,875	51,971
	ca-GrQc	5,241	28,980
Email	**enron**	**36,692**	**367,662**
Online Social Network	**facebook**	**4,039**	**170,174**

Table 1: Dataset sizes.

- **Coauthorship Networks:** We extracted the social network from the DBLP, removing all nodes with less than 3 publications (to leave us with a more connected and challenging data set, as was done also in [5]). We considered both the entire network (dblp-all-core3) and subsets from particular ranges of years. We also considered collaboration networks from additional areas of science. Such datasets are named ca-s, where s is the subject matter.

- **Email Network:** The email network is defined by the Enron email communication network, released by the Federal Energy Regulatory Commission during its investigation.

- **Online Social Network:** This network consists of anonymized Facebook data that was collected from survey participants using a Facebook application.

All of the above datasets, except those of DBLP, were taken from the Stanford Network Analysis Project (SNAP).[11] While our experimentation was performed for all datasets in Table 1, we only present the results for those in bold (i.e., the entire dblp, coauthorship in high energy physics, the Enron email dataset, and that of Facebook). For the other networks, the trends were similar. As stated earlier, full experimental results are available online.[12]

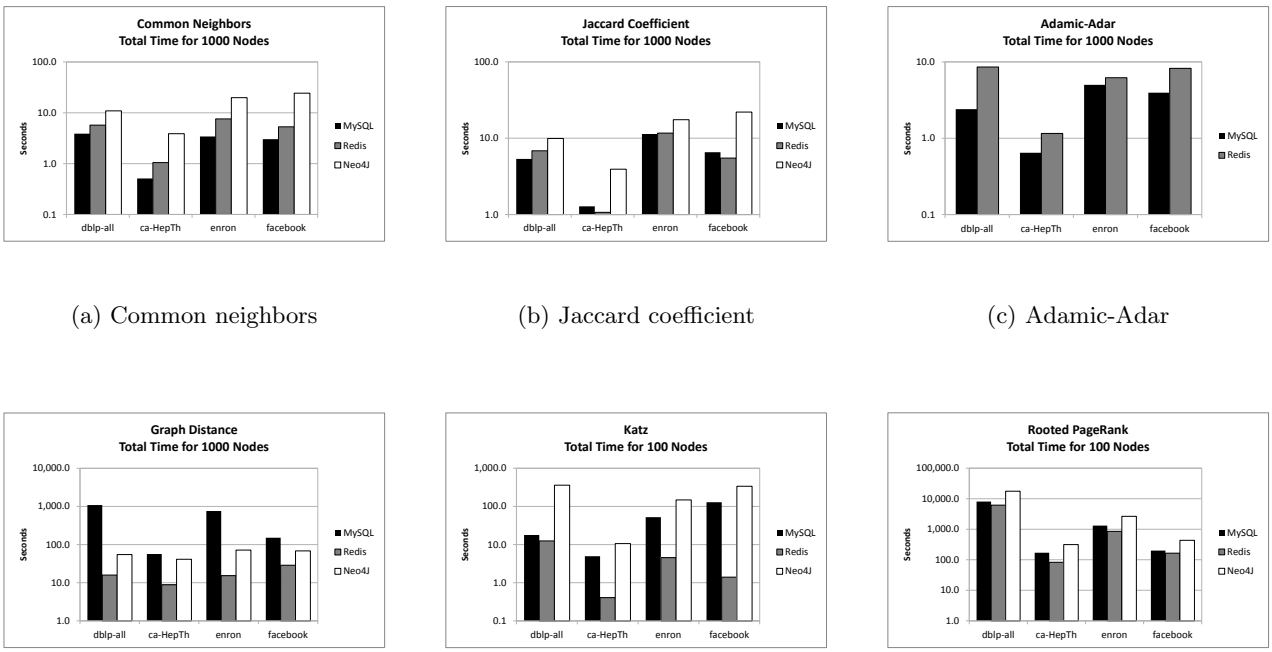
Experiments were run on a PC with an Intel Core i5 3550 processor, 24GB RAM, running the Ubuntu 12.10 64 bit operating system. We now discuss the results of our experimental evaluation, for each link prediction metric.

Common neighbors. We randomly chose 1000 nodes from the network. For each node we generated the top-100 potential new neighbors, according to the common neighbor metric. The total runtime, for each of the three database backends, appears in Figure 1a. (We do not show the average runtime in our graphs, as this is typically too small to be accurately measured.)

Observe that MySQL outperforms both Redis and Neo4J on all datsets. This is not surprising, as the common neighbors metric is implemented in MySQL as a simple join query, and as such, it benefits from the fact that relational databases

[11] http://snap.stanford.edu/

[12] `http://www.cs.huji.ac.il/~sara/link-prediction.html`



(a) Common neighbors (b) Jaccard coefficient (c) Adamic-Adar

(d) Graph distance (e) Katz measure (f) Rooted PageRank

Figure 1: Runtime for computing link prediction metrics.

are highly optimized for joins. Redis comes in at second place, and is significantly more efficient than Neo4J.

Jaccard coefficient. As for common neighbors, we randomly chose 1000 nodes from the network, and generated the top-100 potential neighbors, according to the Jaccard coefficient metric. The total time appears in Figure 1b. There is no clear winner among MySQL and Redis, as MySQL is superior on dblp-all and enron, while Redis has better time over ca-HepTh and facebook. Note that dblp-all and enron are the larger two of the four datasets. It seems that over larger datasets, the query optimizer of MySQL is significant in achieving good performance, while over smaller datasets, the overhead of using a relational database (e.g., transaction management) degrades the performance a bit, allowing Redis to have the superior runtime.

Adamic-Adar. Once again, we randomly chose 1000 nodes from the network, and generated the top-100 potential neighbors, according to the Adamic-Adar metric. As discussed earlier, this metric was not implemented in Neo4J due to the lack of support for the logarithm function. The experimental results appear in Figure 1c. MySQL outperforms Redis for this metric, as this metric is implemented as a simple query over the relational database.

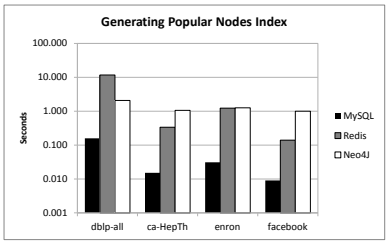
Preferential attachment. Preferential attachment favors nodes with the largest number of neighbors. Thus, our computation of preferential attachment involves first creating an index that computes the 100 nodes with the most neighbors, and then, using this index to return the nodes and compute the score. Indexing can be performed once, if the network is static, or every time preferential attachment is computed, if the network is dynamic.

In Figure 2a, we show the runtime of finding the top-100 nodes with the most neighbors. Again, MySQL outperforms the other databases. To understand why, observe that in all three databases, we store the number of neighbors per node as part of the data (explicitly in MySQL and Neo4J, and implicitly in Redis). However, only in MySQL an index can be created over this data, allowing it to be quickly accessed when we desire to find the most popular nodes. For Redis and Neo4J, it is necessary to scan the entire database, to find the most popular nodes. Redis outperforms Neo4J on the smaller datasets, while Neo4J seems to have the advantage for the largest dataset (dblp-all).

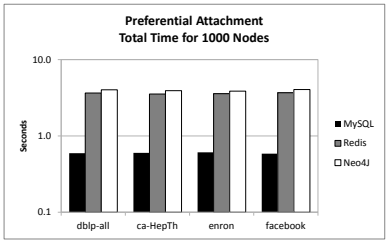
In Figure 2b, we show the runtime of using the index created, in order to return the top-100 potential neighbors, for each of the 1000 randomly chosen nodes. Once again, MySQL has the fastest performance, while there is almost no difference in performance between Redis and Neo4J.

Graph distance. As before, we chose 1000 random nodes, and looked for the top-100 nodes according to the graph distance metric. The experimental results appear in Figure 1d. For this metric MySQL is no longer the winner, and is outperformed by both other backends, with Redis being the most efficient backend. While in all three systems, graph distance is computed conceptually in the same manner, the MySQL and Neo4J implementations utilized the available query languages, which do not seem to be sufficiently optimized for recursive computations.

Katz measure. The Katz measure is significantly more time consuming to evaluate than the previously considered



(a) Popular Nodes Index



(b) Preferential attachment

Figure 2: Runtime for preferential attachment.

measures, as it takes into consideration multiple paths between nodes. Hence, in our experimentation, we chose only 100 nodes (instead of 1000 nodes), and computed the top-100 potential neighbors for these nodes. The total time appears in Figure 1e. We emphasize that while the performance seems to be similar to that of the other graphs, it is actually significant worse, as this graphs sums the time for link prediction for 100 nodes, instead of 1000 nodes.

Similarly to the graph distance metric, Redis is the most efficient implementation. In this metric, however, MySQL outperforms Neo4J. It seems likely that this is due to the fact that Cypher currently lacks support for the exponent function, thus, requiring a more cumbersome implementation of the Katz measure.

Rooted PageRank. Rooted PageRank is by far the most expensive metric to compute, as it iterates multiple times over the network, until convergence is reached. In our experimentation, as in that for the Katz measure, we randomly chose only 100 nodes. For these nodes, we returned the top-10 potential neighbors. Returning only top-10 predictions allowed us to consider the program converged earlier on, once the top-10 results did not change. The experimental results appear in Figure 1f.

Once again, for this complex measure, Redis has the best performance. However, this time MySQL comes in as a relatively close second. It seems that the inherent performance differences between Redis and MySQL systems are dwarfed by the complexity of the rooted PageRank metric, which requires a highly expensive computation over the entire social network.

5. CONCLUSION

Each of the three backends have significant advantages over the other two. MySQL is highly optimized, and performs extremely well when metrics can be written as SQL queries. Redis is very flexible, and has the performance advantage for complex metrics, as it is quite open to optimization by programmer. Neo4J, which had the worst performance in general, is the winner on implementation simplicity—the matching data model, as well as the Cypher query language, made the implementation in this system straight-forward.

As future work, we would like to develop a hybrid system, combining the advantages of all three backends. Such a system could present a graph data model to the user, as well as a Cypher-like (i.e., graph oriented) query language, while actually using both relational tables and a key-store as a backend. Redundant storage of data in both models, while dynamically choosing which implementation to use, for a given user query, could give excellent overall performance, by leveraging the strengths of both types of systems.

6. ACKNOWLEDGEMENTS

The authors were partially supported by the Israel Science Foundation (Grant 143/09) and by the Ministry of Science and Technology (Grant 3-8710).

7. REFERENCES

[1] L. Adamic and E. Adar. Friends and neighbors on the web. *Social Networks*, 25:211–230, 2001.

[2] S. Cohen, L. Ebel, and B. Kimelfeld. A social network database that learns how to answer queries. In *CIDR*, 2013.

[3] M. Kargar and A. An. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*, 2011.

[4] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, March 1953.

[5] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. In *CIKM*, 2003.

[6] M. S. Martín, C. Gutierrez, and P. T. Wood. Snql: A social networks query and transformation language. In *AMW*, 2011.

[7] R. Ronen and O. Shmueli. Soql: A language for querying and creating data in social networks. In *ICDE*, 2009.

[8] N. Ruflin, H. Burkhart, and S. Rizzotti. Social-data storage-systems. In *Databases and Social Networks*, DBSocial '11, pages 7–12, New York, NY, USA, 2011. ACM.

[9] D. F. S. Suarez. *SociQL: A Query Language for the Social Web*. PhD thesis, University of Alberta, 2011.