Microblogging Queries on Graph Databases: An Introspection

Oshini Goonetilleke
RMIT University, Australia
oshini.goonetilleke@rmit.edu.au

Saket Sathe
IBM Research, Australia
ssathe@au.ibm.com

Timos Sellis
RMIT University, Australia
timos.sellis@rmit.edu.au

Xiuzhen Zhang
RMIT University, Australia
xiuzhen.zhang@rmit.edu.au

ABSTRACT

Microblogging data is growing at a rapid pace. This poses new challenges to the data management systems, such as graph databases, that are typically suitable for analyzing such data. In this paper, we share our experience on executing a wide variety of micro-blogging queries on two popular graph databases: Neo4j and Sparksee. Our queries are designed to be relevant to popular applications of microblogging data. The queries are executed on a large real graph data set comprising of nearly 50 million nodes and 326 million edges.

Keywords

graph databases, twitter analytics, graph queries

1. INTRODUCTION

Growing number of applications consume data collected from microblogging websites such as Twitter. Frameworks have been developed to facilitate the different stages of analysis in the Twittersphere, typically centered around collection, querying, visualization, storage, and management of data. In [2] we review existing approaches of such frameworks and identify essential components of a solution that integrates all of these stages. Many of the existing research has focused on using RDF or relational models to represent Twitter data [2]. As one of the requirements identified in that paper we highlight the need for efficient querying and management of large collections of twitter data modeled as graphs. In this study we model the basic elements of the Twittersphere as graphs, and as a first step, determine the feasibility of running a set of proposed queries and present our experience.

Twitter data can often be modeled using a graph structure, such as a *directed multigraph*. A directed multigraph is a directed graph that allows two nodes to be connected with more than one edge. Since graph database systems support management of such types of graphs, a natural way to analyze Twitter data is by using a graph database.

Analysis of general data management queries on graph database systems have been widely reported [1, 9, 3], but none have demonstrated the feasibility of analyzing microblogging queries using such databases. It is noteworthy that most of the prior studies have focused on either executing MLDM (machine learning data mining) algorithms over large graphs [8] or on performing graph data management queries using relational databases [5]. Different from these approaches, in this paper we model Twitter data using a *directed multigraph* and study the feasibility of using a graph database to query Twitter data.

Microblogging data has unique characteristics that are not present in other graph data sets. Multigraph structure of twitter data typically contains free text (e.g, tweets) that is attached to a large number of graph nodes and the nodes and edges can have various properties defined by *key-value* pairs. This makes the analysis of such data both interesting and challenging. Armed with these observations, we have carefully chosen queries pertinent to several applications of microblogging data. For example, our queries are relevant to applications like providing friend recommendations, analyzing user influence, finding co-occurrences and shortest paths between graph nodes. In addition, we have analyzed fundamental atomic operations like selection and retrieving the neighbourhood of a node.

For executing the aforementioned queries, we have chosen two popular open-source graph database systems: Neo4j[1] and Sparksee[2]. Such systems are typically able to efficiently answer data management queries concerning attributes and relationships exploiting the structure of the graph. The goal of this paper is not to perform a full benchmark of the two systems or recommend one over the other. Instead our objective is to report our experiences working on these two graph database systems, as a way forward for us to building an analytics platform geared for well-known micro-blogging websites such as Twitter.

The rest of this section discusses related work. In Section 2 we discuss the systems (Neo4j and Sparksee) and the graph schema that we used for analyzing the dataset. Section 3 discusses the setup and dataset we used for our feasibility analysis. The queries and our experience in exe-

GRADES'15, May 31 - June 04 2015, Melbourne, VIC, Australia

DOI: http://dx.doi.org/10.1145/2764947.2764952

[1] http://www.neo4j.org/
[2] http://www.sparsity-technologies.com

cuting those queries is also discussed. A concise discussion of our experience is presented in Section 4, and a conclusion is offered in Section 5.

Related Work: Many studies have benchmarked graph database management systems in terms of computational workloads [8] or data management query workloads [9, 3, 6]. Subjective indicators such as user experience and level of support [9, 8] are also used to compare these systems. Another benchmark by Angles *et al.*[1] defines a set of generic atomic operations used in social networking applications. Ma *et al.*[5] benchmarks graph analysis queries on a relational model expressed in SQL over a schema for microblogs. We believe that graph data management systems are better equipped to test the particular type of microblogging data workloads used in this paper. Inspired by these studies, we define queries relevant to microblogging and share our introspection on executing them using graph management systems; thereby perfectly complementing these prior works.

2. PRELIMINARIES

Recall that the graph structure contained in Twitter data can be represented using a directed multigraph. Twitter has a follower-followee network, which is a directed graph of users following each other. We refer to the set of followers of user `A` as her 1-*step followers*. Similarly, the set of followers of `A`'s 1-step followers are said to be `A`'s 2-*step followers*; analogously, we can define the *n-step followers* and *n-step followees* of a user `A`.

2.1 Graph Databases

In addition to the follower network, for accurate data representation of Twitter data we need the graph systems to support two additional features: (1) labeling a node or edge, and (2) associate key-value pairs to a node or edge. Keeping these requirements in mind, we chose two leading open-source graph management systems, namely, Neo4j and Sparksee for our analysis. These systems not only support all the features needed for analyzing Twitter data, but also support a wide variety of query types and API interfaces to interact with the multigraphs.

Neo4j is a fully transactional graph management system implemented in Java. It supports a declarative query language called *Cypher*. As an example, a query that retrieves the tweets of a given user can be written in Cypher as:

```
MATCH (u:USER uid:{531})-[:POSTS]->(t:TWEET)
RETURN t.text;
```

Another method of interaction is by using its core API. The core API offers more flexibility through a *traversal framework*, which allows the user to express exactly how to retrieve the query results.

Cypher supports caching the query execution plans. This reduces the cost of re-compilation at run-time when a query with a similar execution plan is executed more than once. We have often used Cypher's profiler to observe the execution plan and determine which query plan results in the least number of database hits (db hits) and have rephrased the query for better performance. It is noteworthy that all the queries can be alternatively written using the Java API exploiting the traversal framework. However, as with any imperative approach, the performance is dependent on *how* the query is translated into a series of API calls.

Sparksee, formerly known as DEX, is a graph database management system implemented in C++. Sparksee provides APIs in many languages. We choose the Java API for our experiments. As an example, the above query can be written in Sparksee's API as:

```
int nodetype = g.findType("USER");
int attrID = g.findAttribute(nodetype, "uid");
Value attrVal = new Value();
attrVal.setInteger(531);
long input = g.findObject(attrID, attrVal);
int edgeType = g.findType("POSTS");
Objects userTweets = g.neighbors(input, edgeType,
EdgesDirection.Outgoing);
```

Sparksee queries have two primary navigation operations: `neighbours` and `explode`, which return an unordered set of unique node and edge identifiers that are adjacent to any given node ID. When translating the queries using Sparksee's API, we made use of most of the constructs provided by the developers. Instead of the traditional matrix and list based representations of graphs, Sparksee stores graphs using a compressed bitmap-based data structure [7].

2.2 Database Schema

The graph data model we use for modelling Twitter data consists of three types of nodes: `user`, `tweet` and `hashtag` (refer Figure 1). Users following each other are represented by a `follows` relationship, while posting is represented by a `posts` edge between a `user` and `tweet`. A retweet of an original tweet is denoted by a `retweets` edge, while mentions of a tweet by a particular user are captured by a `mentions` edge. If a particular tweet contains a hashtag, the `tags` edge is used for representing this information. The same schema is used when creating the databases in Neo4j and Sparksee.

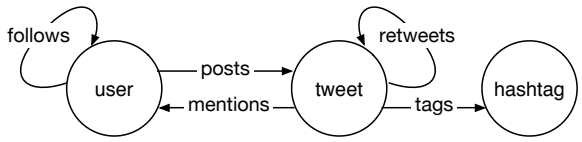


Figure 1: Data model of the schema.

3. FEASIBILITY ANALYSIS

In this section we will analyze the feasibility of executing a wide variety of relevant microblogging queries on Neo4j and Sparksee. We start by discussing the details of the data set we used (refer Section 3.1). In Section 3.2, we share our experience in importing Twitter data to both systems. Then in Section 3.3, we share our experience gathered while executing the aforementioned microblogging queries.

All the experiments were conducted on a standard Intel Core 2 Duo 3.0 GHz and 8GB of RAM with a non-SSD HDD. For Neo4j we used version 2.2.M03, and for Sparksee version 5.1 was used. The research license for Sparksee could accommodate up to 1 billion objects. We used the respective APIs in Java embedding the databases.

3.1 Data Set and Pre-processing

We use the data set crawled by Li *et.al.* in [4]. It consists of 284 million `follows` relationships among 24 million users. For a subset of 140,000 users who have at least 10 followees,

this data set contains 500 tweets per user. We only retain 200 tweets per user from this set. By processing the tweets, we reconstruct all the edges and nodes of the schema shown in Figure 1. Unfortunately, this data set does not have exact information on retweets, therefore we could not reconstruct the `retweets` edges. Although the data set is not complete with tweets of all users, it satisfies the requirement of being able to model the schema with a reasonable number of nodes and edges. A summary of the characteristics of the data set is shown in Table 1.

Table 1: Characteristics of the data set.

Node		Relationship	
user	24,789,792	follows	284,000,284
tweet	24,000,023	posts	24,000,023
hashtag	616,109	mentions	11,100,547
		tags	7,137,992
Total	49,405,924	**Total**	326,238,846

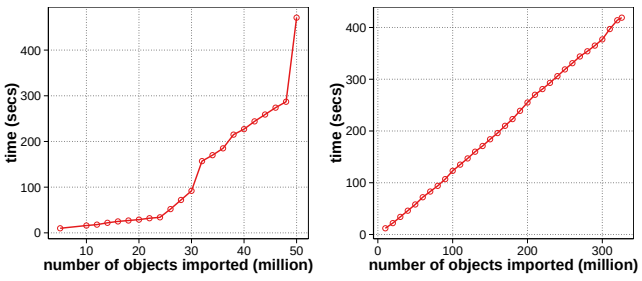
3.2 Data Ingestion

We use batch loading procedures offered by both graph systems. The same source files containing the nodes and edges were used with both databases.

3.2.1 *Neo4j*

We used the Neo4j's import tool for importing the workload. We decided to use the import tool after trying several other options. A main reason was that the tool effectively manages memory without explicit configuration. However, it cannot create indexes while importing takes place. Indices were created after the data import is complete. Neo4j's import tool writes continuously and concurrently to disk.

We plot the time taken for importing nodes and edges in Figure 2(a) and (b) respectively. Observe that the insertion of edges is smoother as compared to the nodes. The jumps in Figure 2(a) are mainly due to the time taken to flush the nodes to disk, which slightly slows down the import. After the node import is complete, Neo4j performs additional steps, for example, computing the dense nodes, before it proceeds with importing the edges. These intermediate steps require around a total of 10 minutes. Then we create indexes on all unique node identifiers, which took about 8 minutes. Since a node could be of type `user`, `tweet`, or `hashtag`, these indexes give us the flexibility to efficiently query the aforementioned node types. Overall, importing the workload required a total of 45 minutes taking 20.8 GB of disk space.



(a) Nodes (b) Edges

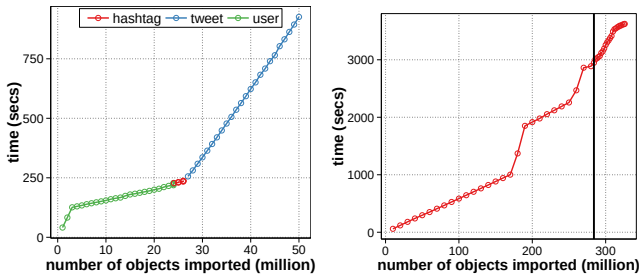
Figure 2: Import times for nodes and edges using Neo4j.

3.2.2 *Sparksee*

Sparksee scripts, which is an importing mechanism available in Sparksee, has been used to define the schema of the database. A script also specifies the IDs to be indexed and source files for loading data. Recovery and rollback features were disabled to allow faster insertions. The extent size was set to 64 KB and cache size to 5GB. With lower extent sizes, we found that the insertions are fast initially but slow down as the database size grows. Sparksee recommends to materialize neighbors during the import phase. This creates a neighbor index that can be used for faster querying. But with this option enabled, it took us a long time to import, and we aborted the import after waiting for 8 hours. With materialization turned off, Sparksee required 72 minutes taking 15.1 GB of disk space.

The load times for nodes and edges are shown in Figure 3. The three colored regions in Figure 3(a) correspond to the three node types imported with different pay loads. The import times of the nodes can be separated in three regions marked in Figure 3(a). Since the payload of the `tweet` nodes is larger as compared to the other node types, their insertion is slower. The vertical line in Figure 3(b) refers to the end of the import of `follows` edges, which make up 80% of the edges. The remaining edge types add up to only 20% of the edges. Sharp jumps in the insertion time of edges is when the cache is full and has to flush to disk, before insertions can be continued. Notice that the jumps in Figure 3(a) are bigger than the jumps in Figure 2(a). This is because Neo4j concurrently writes to disk, while Sparksee waits for the cache to be full before flushing it to disk. The plots for data ingestion are not consolidated as the batch loaders for the two database systems operate on different settings.

At the time of writing this paper, both Neo4j and Sparksee could not import additional data into an existing database (i.e. incremental loading), hence all data was loaded in one single batch.



(a) Nodes (b) Edges

Figure 3: Import times for nodes and edges using Sparksee. The vertical line in (b) refers to the end of the import of `follows` edges.

3.3 Query Processing

In this section we propose a set of relevant microblogging queries and share our experience in executing these queries on Neo4j and Sparksee. We propose a set of non-exhaustive queries that were designed, keeping in mind the typical analysis that is performed with microblogging data. We observe typical queries for general social networks as proposed by [1] and extend the query classification by introducing queries useful in the context of microblogs.

Table 2: Query workload. Experience for queries marked with (⋆) is discussed in detail.

	Category	Example	
	Q1.1	Select	All Users with a follower count greater than a user-defined threshold
	Q2.1	Adjacency (1-step)	All the followees of a given user `A`
	Q2.2	Adjacency (2-step)	All the Tweets posted by followees of `A`
	Q2.3	Adjacency (3-step)	All the hashtags used by followees of `A`
(⋆)	Q3.1	Co-occurrence	Top-n users most mentioned with user `A`
	Q3.2	Co-occurrence	Top-n most co-occurring hashtags with hashtag `H`
(⋆)	Q4.1	Recommendation	Top-n followees of `A`'s followees who `A` is not following yet
	Q4.2	Recommendation	Top-n followers of `A`'s followees who `A` is not following yet
	Q5.1	Influence (current)	Top-n users who have mentioned `A` who are followees of `A`
(⋆)	Q5.2	Influence (potential)	Top-n users who have mentioned `A` but are not direct followees of `A`
(⋆)	Q6.1	Shortest Path	Shortest path between two users where they are connected by `follows` edges

The queries are classified into six categories as shown in Table 2. For each category we use 2-3 exemplar queries for performing the analysis. The systems were left in its default configurations. We start executing a query and once the cache is warmed-up and the execution time is stabilized, we report the average execution time over 10 subsequent runs. Although we present our experience with implementing all query types on top of both systems, we do not report on the performance of all query types as Q1, Q2 are deemed to be simpler than the rest. Hence, in our discussion we focus on Q3, Q4, Q5, Q6, as indicated with a (⋆) in Table 2. As opposed to Sparksee, Neo4j uses a declarative query interface, which may involve processing overhead. Therefore, for fairness reasons, we report on the performance of these two systems separately.

Basic Queries.
We start by presenting our experience with selection and adjacency queries.

Q1 – Select queries: These queries select nodes or edges based on a predicate over one or more of their properties. Combination of selection conditions can be easily expressed in Cypher with logical operators. Sparksee does not directly support filtering on multiple predicates. Therefore, to evaluate a disjunctive or conjunctive query, we have to evaluate its predicates individually and combine the results appropriately to construct the final result.

Q2 – Adjacency queries: Adjacency queries retrieve the immediate neighborhood of a node specified with different number of hops. Adjacency queries form the basis of almost all other queries mentioned in Table 2. As shown in Table 2, we have used 1-, 2- and 3-step adjacency queries on both systems.

Advanced Queries.
Next, we discuss our experience in executing queries Q3-Q6. Queries Q3, Q4, and Q5 are top-n queries. Such queries can be easily expressed and executed in Cypher using `COUNT`, `ORDER BY` and `LIMIT` clauses. For Sparksee, a map structure is used for maintaining the required counts. These counts are then sorted to obtain the final result. Its API does not provide the functionality to limit the returned results.

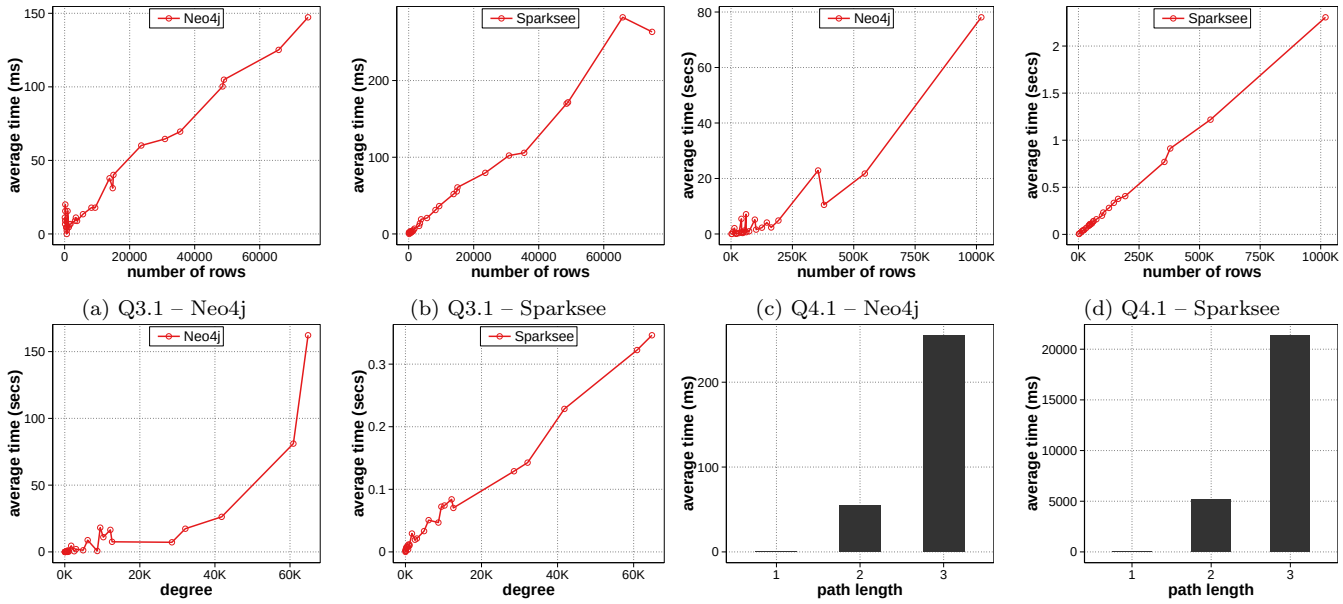
Q3 – Co-occurrence queries: Two nodes of any type are said to be *co-occurring* with each other if there is another node that connects both of them. Elements tested here are co-occuring hashtags and mentions. Co-occurrence is a special type of adjacency query useful for finding recommendations for users. Finding co-occurrences is a 2-step process. For example, in Q3.1 the steps are (1) find the users who mention a given user `A` in their tweets `T`, and (2) find other users that are mentioned in the tweets `T`. The results of the query execution for query Q3.1 are shown in Figure 4(a) and (b). They show a straightforward increasing trend. However, when the number of rows returned are low the results for both systems seem to fluctuate, but become more predictable with increase in rows returned. Perhaps this fluctuation is due to the random disk accesses that require a different portion of the graph for a new parameter.

Q4 – Recommendation queries: Recommending users to follow, often involves looking at a user's 1- and 2-step followers/followees, since recommendations are often useful when obtained from the local community. We propose two types of recommendation queries. Q4.1 finds all the 2-step followees of a given user `A`, who `A` is not following. Such followees are recommended to `A`. Another variant of this query is Q4.2 that finds 1-step followers of `A`'s 1-step followees. Only the top-n users who are followees of A's followees are considered relevant for recommendation.

Recall that for retrieving the neighbours of a particular node, Sparksee provides the `neighbours` operator. For answering Q4.1, a separate `neighbours` call has to be executed for each 1-step followee of `A`, which makes the execution of this query expensive. A separate `neighbours` call is required since we are interested in the popularity (in terms of outlinks) in addition to the identity of `A`'s 2-step followees.

The result of executing Q4.1 are shown in Figure 4(c) and (d). Finding 2-step followees results in an explosion of nodes when 1-step followees have high out-degree. This forces the systems to keep a large portion of the graph in memory. The sudden spike in the plot for Neo4j is due to the fact that, the direct degree of the node in concern is much higher even though the number of rows returned are lower. It is noteworthy that (a) Neo4j's performance degrades with a large intermediate result in memory, and (b) Sparksee is able to take advantage of the graph already in memory, as we observe less fluctuations with the output.

Q5 – Influence queries: In many use cases, it becomes



(a) Q3.1 – Neo4j (b) Q3.1 – Sparksee (c) Q4.1 – Neo4j (d) Q4.1 – Sparksee

(e) Q5.2 – Neo4j (f) Q5.2 – Sparksee (g) Q6.1 – Neo4j (h) Q6.1 – Sparksee

Figure 4: Query execution results. (a) and (b) show co-occurrence query (Q3.1), while (c) and (d) show recommendation query (Q4.1). Influence query (Q5.2) is shown in (e) and (f) and shortest path query (Q6.1) is shown in (g) and (h).

necessary to discover current and potential influence a particular user has on her community. As an example, for targeting promotions a retail store (with a Twitter account) might be interested in the community of users whom they can influence. Although there exists research on many models of influence propagation, we focus our attention to a set of intuitive queries that can be used for examining influence.

Influence queries are defined based on current and potential influence with respect to a given user. Current influence of `A` is defined as the most frequent users who mention `A` and who are already followers of `A`. Potential influence of a person is people who are most mentioning `A` without being direct followers of `A`. Both in Neo4j and Sparksee this translates to finding the users who mentioned `A`, and removing (or retaining) the users who are already following `A`.

The result of executing Q5.2 is shown in Figure 4(e) and (f). Degree of a user mention(x-axis) is defined as the number of times that user is mentioned in the collection. Notice that the degree is low, demonstrating behavior similar to that of the first portion of the plots for co-occurrence queries (refer Figure 4(a) and (b)).

Q6 – Shortest path queries: Shortest path queries find the shortest path between two given nodes in a graph. In addition to finding a path, they can also handle restrictions on the type of node that these queries can return as a part of the shortest path. An example of a shortest path query is Q6.1. Shortest path queries can be the basis of a query that needs to target a particular user or a community of users, essentially finding the degrees of separation from one person to another.

In practice, it is necessary to limit the number of returned paths and/or depth of the traversal (maximum hops) otherwise it could lead to an exhaustive search for all paths. The Cypher function `shortestPath` was used for writing the query. While the native function `SinglePairShortestPathBFS` was used for Sparksee, where maximum length of the shortest path was set to 3 hops. The average time required to compute the shortest path between two randomly selected users is shown Figure 4(g) and (h). In our experience, Neo4j seems to perform shortest path queries more efficiently.

Deriving Other Queries.
It is noteworthy that many other interesting questions can be answered by using different combinations of the aforementioned queries in Table 2. As an example, suppose user `A` is interested in a topic (represented by a hashtag `H`) and is looking for users to know more about the topic. Such a query can be answered combining other queries as follows:

1. Get all the hashtags that co-occur with the given hashtag `H` **(Q3.2)**
2. Get the most retweeted tweets mentioning those hashtags **(Q2.3)**
3. Get the original users of those retweets (given that `retweets` relationships is modeled in the database)
4. Order the users based on the shortest path length from `A` **(Q6.1)**

Our limited data set restricted us in trying more complex queries, such as the one above, even though they are interesting and useful queries for microblogging. For future work, we are exploring opportunities to expand our data set to enable more extensive evaluation.

4. DISCUSSION

Alternate Solutions: The queries in Cypher can also be written using the Neo4j API with a combination of constructs in the traversal framework and the core API. For queries that we did translate to the API, we observed a slight improvement in performance compared to the Cypher queries version. But the benefit of a declarative language is lost, if they have to be re-written using the less expressible

API from scratch for performance gains. It must also be noted that significant effort was required to translate some of the queries in the traversal description when they can be very conveniently expressed in Cypher.

We also noticed performance differences in queries (returning the same results) depending on the way they were expressed in Cypher. For example, a recommendation query can be written in three similar ways: (a) going through the follows relationships for depth 2, `[:follows*2..2]`, (b) collecting the intermediate results and checking them against the results at depth 2, and (c) expanding the `follows` relationship to depth 2 and removing the friends at depth 1. Method (b) was performing the best. Methods (a) and (b) resulted in different execution plans, however with similar number of total database accesses. It was not clear why Method (c) failed to return a result in a reasonable time.

As such, some queries had to be rephrased in order to achieve gains in performance. Ideally a query optimizer in Cypher should be converting a query plan to a consistent set of primitives at the back end. With every new release Cypher is being improved with a lot of emphasis on cost based optimizers to cater for this. While the expressiveness is a great advantage in Cypher, an optimizer must take care of converting it to an efficient plan based on the cost of alternate traversal plans. A good speedup can be achieved by specifying parameters, because it allows Cypher to cache the execution plans.

On the other hand Sparksee requires sole manipulation of mainly navigation operations (`neighbors`, `explode`) to retrieve results. Even though this gives a lot of flexibility, this might end up in a series of expensive operations as it was in the case of recommendation queries. In Sparksee, queries can also be translated to a series of traversals using the `Traversal` or `Context` classes. Our preliminary findings show that using the raw navigation operations (`neighbors` and `explode`) are slightly more efficient than expressing the query as a series of traversal operations. This is perhaps due to the overhead involved with the traversals. Sparksee can certainly also benefit from a query language to complement its current API.

Overhead for aggregate operations: For many queries we are often interested in only finding the top-n results. For example, it is not useful to show more than 5 recommended followers for a given user. Cypher performance increases with the removal of the additional burden of having to order the results by a count after the grouping. Removing ordering, deduplication and limiting the number of results returned are all factors that contribute to performance gains in Cypher. In Sparksee, in order to limit the returned results, the entire result set must be retrieved and filtered programatically to display only the top-n rows.

Problems with the cold cache: We noticed that Neo4j takes a long time to warm up the caches for a new query. This time is even larger if we do not allow the execution plans to be cached. The time taken for the first run is significant even for queries exploring a small neighborhood. It might not always be possible to allow the caches to be warmed up, if a large number of queries access the cold parts of the graph. As the degree of the source node increases, the time it takes to warm the cache dramatically increases as the system attempts to load a large portion of the graph in memory.

5. CONCLUSIONS

In this paper, we ran a set of queries with a schema suitable for microblogs. Our purpose was not to formally evaluate these two graph management systems, but was only to share our experience while running a set of queries on these two representative systems. The queries we have proposed can be easily extended to other domains where a similar schema is applicable.

The workload for algorithms such as PageRank, calculating connected components etc. although useful, have not been tested here as we believe such algorithms are better suited for distributed graph processing platforms.

As future work, we would like to investigate how the graph could be generated on-the-fly with new incoming users, tweets and follow relationships. By observing the content of the tweet and generating the remaining relationships, one can simulate the true real-time nature of microblogs. With this setting, it would be possible to test for the ability of systems to handle update workloads as well.

The studied graph management systems treat all node (and edge) types equally. They have virtually no understanding of the semantics of the nodes/edges. For example, to represent the `posts` relationship different from a `follows`, as `posts` are more dynamic in nature than `follows`. It would be an interesting extension to explore the possibility of a semantic-aware strategy to speed up the queries, and to see how semantically related nodes can be stored/partitioned when the queries are known.

6. REFERENCES

[1] R. Angles, A. Prat-Pérez, et al. Benchmarking database systems for social network applications. In *GRADES Workshop*, pages 1–7, 2013.

[2] O. Goonetilleke, T. Sellis, et al. Twitter analytics: A big data management perspective. *SIGKDD Explorations*, 16(1):11–20, 2014.

[3] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *SocialCom*, pages 708–715. IEEE, 2013.

[4] R. Li, S. Wang, et al. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *KDD*, pages 1023–1031, 2012.

[5] H. Ma, J. Wei, et al. On benchmarking online social media analytical queries. In *GRADES Workshop*, pages 1–7, 2013.

[6] P. Macko, D. Margo, et al. Performance introspection of graph databases. In *SYSTOR*, pages 1–10, 2013.

[7] N. Martínez-Bazan, M. A. Águila Lorente, et al. Efficient graph management based on bitmap indices. In *IDEAS*, pages 110–119, 2012.

[8] R. C. McColl, D. Ediger, et al. A performance evaluation of open source graph databases. In *Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, 2014.

[9] C. Vicknair, M. Macias, et al. A comparison of a graph database and a relational database: A data provenance perspective. In *Annual Southeast Regional Conference*, ACM SE '10, pages 1–6, 2010.