ANNALS OF THE UNIVERSITY OF CRAIOVA
Series: Automation, Computers, Electronics and Mechatronics, Vol. 12 (39), No. 1, 2015

Running Complex Queries on a Graph Database: A Performance Evaluation of Neo4j

Călin Constantinov, Mihai L. Mocanu, Cosmin M. Poteraş

Department of Computers and Information Technology, University of Craiova, Romania (e-mail: constantinov.calin@ucv.ro, mmocanu@software.ucv.ro, cpoteras@software.ucv.ro).

Abstract: Computer science, by far one of the most dynamic research domains, manages to produce concepts, prototypes and paradigms at a pace that is sometimes hard to follow. Although most of these ideas set goals that are more or less realistic, some of them are able to fascinate by their potential to lead to dramatic changes. One particular case is represented today by graph databases, a type of NOSQL solution which can be applied to very power demanding tasks in data analysis frameworks. They allow for the expansion of data sources, including social media feeds, thus making complex systems such as recommendation engines much more powerful. No matter the controversies among both simple technology enthusiasts and large corporations, these databases are rapidly expanding, managing to not only generate interest, but also remarkable solutions. In this paper, we will have a deep look over the most popular graph database, Neo4j, evaluating its performance and scalability options when running very complex statistics and recommendation queries over a dataset of a significant size, containing strongly interconnected Facebook data. The evolution from a simple standalone database scenario to a Highly-Available (HA) cluster setup is described step by step by analysing the impact that each configuration change has on the system's both read and write performance. Given the positive results of this paper, we believe that graph technologies represent a very promising research domain as, considering their performance, they are likely to hold the key to building an efficient, distributed social-network graph mining framework on which data analysis jobs can be continuously run, further enhancing the data.

Keywords: Graph Database, Facebook Data, Performance Evaluation, Social Recommendation Engine, Data Analysis

1. INTRODUCTION

Although NOSQL databases faced initial reticence from developers used to the comfort and safety inspired by SQL, we are now seeing a strong raise in the popularity of some of these novel persistence solutions, as most of the large corporations are currently at least experimenting with them. Apart from the promising performance gain, some products are reaching maturity and many of the NOSQL drawbacks are being rapidly dealt with, further increasing interest. Although SQL databases still excel in many situations, the need to store very large amounts of data has revealed the difficultly of scaling these traditional solutions. Moreover, given the large number of relationships between entities, modern data is starting to have a graph-like structure. Unfortunately, SQL does not naturally support graph specific operations such as finding the shortest path between two nodes. Complex stored procedures and queries are thus needed for even the simplest tasks.

For instance, given its popularity, Facebook always represents an interesting case-study. Over the years, this platform has generated a lot of interest and has been taken as a reference for all social networks. The rate of growth has been remarkable: not only has the network seen a steep rise in the number of total active users, but also the quantity and complexity of the information that these users can share and which needs to be processed has increased. Also considering that a typical user can have a triple digit number of friends, it is easy to imagine that the data stored by Facebook is not only large and diverse, but also very strongly connected, as mentioned in paper Bronson et al. (2013). Basically, a very large portion of the world's active population is present on Facebook, creating, consuming and sharing content. This is why a lot of businesses have transited from promoting products using a website to simply managing a Facebook page, maximizing reach, strengthening relationships with clients and minimizing operational costs. The advantage of having all these kind of activities in the same place is that most of the information is semi-structured and can be easily mined and processed, given that the right tools are deployed for the job.

The need to process all this data in a reasonable amount of time has led to the evaluation of SQL alternatives. One of the most promising of them is represented by graph databases, as they naturally model social data. In this paper we aim to simulate very complex data analysis jobs

by performing socially-enhanced recommendations on a large data set, a common task in the Web 2.0 world.

Traditional recommender systems have been used by companies or shops that sell an item or a service in the attempt to cross-sell an additional product, as described by work Balabanović and Shoham (1997). There are usually a number of techniques implied: for example, if a certain product is frequently bought together with another product, it makes sense to advertise the second product along with the first one to the potential buyer. Another example would be to identify pairs of users that typically buy the same products and recommend products that just one of them has bought to the other one. Furthermore, collaborative filtering was used for predicting the way in which a certain user would evaluate a product or service, based on his similarity to other users. Modern day engines however have much more social data available for taking into consideration before performing a recommendation. The greatest challenge is to be able to run recommendation queries fast, without significant performance penalties as the size of the database grows. This is difficult to achieve using SQL, but can be attempted using a graph database such as Neo4j which promises easier scalability.

2. BACKGROUND

As mentioned in article Angles and Gutierrez (2008), alternatives for relational databases started to appear as early as the beginning of the nineties when object-oriented databases were developed for several data-intensive applications. Roughly in the same period, the first graph models were developed, but were gradually abandoned until recently. Simple record-type data having a fixed schema is no longer typical for modern applications as this can represent a major drawback in an era flooded by semi-structured information.

The need to store and process data of graph-like nature has, however, revived graph databases over the last few years, providing developers with powerful tools for capturing domain semantics within a visual data model. It is mentioned in the work that nowadays information interconnectivity and topology is at least as important as the data itself, the true value being represented by the links between entities. Moreover, contrary to SQL for which exploring the underlying web of relationships is a difficult task, this new database model comes with a much more intuitive querying language paradigm.

In the last few years, many graph database implementations have been released and choosing the right one is not always an easy task, as they also tend to evolve at a fast pace. Fortunately, many papers which try to benchmark and compare the alternatives are now available.

For instance, paper Ciglan et al. (2012) mentions that the graph data structure is an attractive abstraction for modelling real-world phenomena and insists on the importance of fairly assessing the performance of graph databases. Typically, performance is evaluated by looking over how well these databases can handle various traversal operations as this is what queries come down to in a graph. Two sets of tests are conducted, which match the types of queries that we have run over the system that we will be describing in this paper: one dealing with pure query-like traversals which have random starting points and search for related vertices and a seconds aiming to simulate whole graph traversal operations typically used more complex jobs such as computation of connected components analysis or centrality measures. In this preliminary experiment, Neo4j was tested along alternative databases, obtaining satisfactory results.

The ideal approach for traversal operations is to have the whole graph structure cached in the volatile memory as these operations are characterised by random memory accesses, which are known to be very time consuming for a persistent storage. However, as it will be later detailed, there are many cases for which the whole graph does not fit in the main memory. In such situations, some optimisations can be attempted in order to have most of the queried sub-graph cached.

Work Jouili and Vansteenberghe (2013) states that more and more companies are starting to provide services which can no longer be efficiently managed by using traditional relational databases, forcing them to seek alternative technologies. Graph databases are again mentioned as a possible solution for many types of applications and the paper aims to evaluate a number of implementations from a client's side perspective: measuring the time that a client has to wait between issuing a request and receiving a response from the database, thus including communications overhead. Neo4j is concluded to have the best overall performance, while standing out for its predictable behaviour.

As another novelty, the benchmark introduced in the work supports simulating multiple concurrent clients located on a number of distributed machines. This is something similar to what is happening in the application that we have developed.

Reference Holzschuher and Peinl (2013) provides an insight on Cypher, Neo4j's graph query language, concluding, based on performance and ease of use, that it is a promising candidate for becoming a standard. The authors experiment with a Web 2.0 inspired set of data for which it is now common to use NOSQL alternatives. Modelling this information in a relational database causes a high number of many-to-many relationships which in turn leads to a succession of very costly JOIN operations when querying the data. Graph databases are again recommended for these use-cases and an observation is made about the fact that these technologies can be placed somewhere between the SQL and NOSQL worlds as they are not simple aggregate solutions (such as Key-Value Stores, Column-Family Stores or Document Databases) and usual favour consistency and availability as opposed to partition-tolerance.

The study compares Cypher with Gremlin, an alternative graph query language which, although outperforming the former in a number of scenarios, is not preferred because of having the disadvantage of not being as maintainable and as readable. SQL is partially compared with Cypher and, as expected, performs much worse. Moreover, the experiment confirms that growing the number of entities stored in the graph database does not significantly affect performance as it does in case of an SQL solution. The reason for this is that only the local neighbourhood of a given node is traversed, no matter the size of the

whole dataset. The paper however states that it is not over-stressing highly graph-related queries such as group recommendation queries, which is something that our experiment will focus more on.

An even more thorough comparison between a graph and a relational database can be a can be found in article Batra and Tyagi (2012). Once again, it is noted that when storing social data that is expected to evolve, SQL is not an optimal approach. Although Neo4j is mentioned as not being a mature solution, it provides an easily mutable schema and outperforms MySQL in all the experiments carried out, also showing that the performance gain raises as the size of the database grows.

Scalability is further analysed in work Dominguez-Sal et al. (2010) where four of the most popular graph databases are evaluated using the HPC Scalable Graph Analysis Benchmark using a set of generated data. The tests range from measuring the edge and node insertion along with the creation of initial indexes performance to evaluating the time needed to traverse the whole graph when computing the Betweenness Centrality indicator for certain nodes. Neo4j and DEX, an alternative graph database, were able to achieve the best performance, having no problems scaling up to a dataset consisting of 1 million nodes. Additionally, the importance of having implemented an optimal caching strategy for graphs that do not fit in the main memory is also mentioned in the paper.

It is important to state that all the referenced papers work with synthetic datasets as compared to our tests which were run over real Facebook data. Moreover, older Neo4j versions are used and, as also anticipated by these studies, some of the problems identified are claimed to have been addressed in the newer releases.

Additionally, when dealing with very large datasets, horizontal scaling seems to be the way to go for increasing performance. However, an issue faced by engineers is the fact that although processing power has seen great increases in the last decades, network transfer rates fall way behind. This is why, when trying to improve parallel performance, it is important to try to limit or optimize data transfers over the network. Thus, it makes more sense to try to migrate the computational task to the data than to bring the data where the computational task is being executed. This technique is called computational steering, which our previous work Poteraş et al. (2011) describes in greater detail. A brief discussion on what can be attempted in case of our system will be detailed in a later section.

3. IMPLEMENTATION

Neo4j is currently the most popular graph database, reason for which we chose it for our implementation.

This database can be deployed in two ways:

(1) **Standalone server**: meaning that just one instance of the database is used. Two configurations are possible:
   (a) *Server mode* - as typical for any traditional database, in this configuration Neo4j runs independently of the main application. For communication, a set of REST services are exposed.
   (b) *Embedded mode* - although it only works with applications that run in a Java Virtual Machine, in this configuration Neo4j is tightly coupled to the main application. As it can be expected and also seen in our experiments, very high performance can sometimes be obtained in this scenario. However, there are also many disadvantages when sharing the volatile memory between the application and the database.

(2) **Highly-Available cluster**: allows for a number of database instances to be used together in order to improve performance. A Neo4j HA cluster can be composed of both Server-mode running Neo4j instances as well as Embedded-mode running ones, each of them operating on a different machine. Communication between instances is realized over a custom protocol that tries to eliminate as much overhead as possible. A Neo4j HA cluster has of a single master node and a number of slave nodes. By only allowing write operations to be performed through the master (even if a write request is issued to a slave node, the master will eventually be called to instrument the operation), ACID transactions can be imposed. In order to complete a write operation and guarantee consistency, at least N / 2 + 1 servers from the cluster need to be available (out of N, the total number of servers in the cluster). If this quorum is not met, the cluster functions in read-only mode. As expected, write performance can sometimes be worse on a cluster then on a single database instance, due to the cluster management overhead. Neo4j's major downside is that each node instance has a copy of the whole database, so as for now, the database does not support graph sharding. Graph sharding is a NP-complete problem which is even more difficult to solve considering that the graph is constantly evolving. Read performance can typically see near linear speedup in a cluster configuration, as long as the whole graph can be stored in-memory. Even if this is not the case, it can be assumed that, for general cases, it is faster to load portions of the missing graph from a local permanent storage device than to call for them over the network.

In a Neo4j Highly-Available cluster setup, it is typical for a Java application to issue write requests to an Embedded-Master node, while the read requests to be directed to a Server-Slave node. In case there are multiple Server-Slave nodes, a load balancer such as HAProxy can be used. An example for this setup will be detailed in a later section.

But this kind of configuration can be the starting point for a much more sophisticated graph database cluster. Although social graph sharding might be very difficult to achieve at this moment, cache sharding remains a valid approach for improving read times. In cases were the graph is very large, it might be impossible to store it entirely in the very fast, but expensive, volatile memory. This is especially true for Neo4j which employs a native graph storage. However, Neo4j's cache is filled with portions of the graph most heavily accessed, so by using an advanced load-balancing algorithm, requests could be routed to an instance of Neo4j likely that holds most of the queried graph in its cache. The trick would be to identify usage

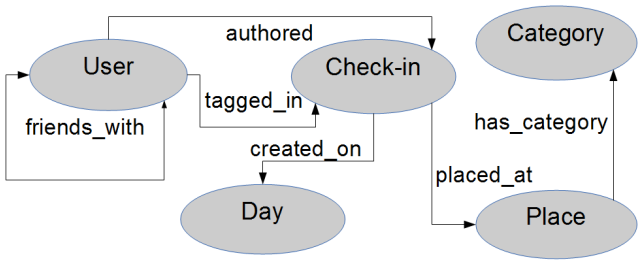

Fig. 1. Neo4j Data Model

patterns by means of statistical analysis and customize the HAProxy load balancer to forward requests to the ideal instance of the graph database. In a way, this is also a form of computational steering, as described in a previous section. This is an active research domain which could lead to building an optimal distributed social-network analysis framework, but further discussions are outside the scope of this paper.

The application aims to constantly mine Facebook information and gather as many check-ins placed in the city of Craiova as possible and use them for performing recommendations and computing statistics. As Facebook data is not generally public, user authorization tokens are required. Thus, two main modules were developed:

(1) **Core Web Module**: using Facebook Graph API 1.0 and Facebook Query Language, check-ins are gathered and stored in Neo4j. The graph data model can be seen in Fig. 1. The module also exposes a REST service that will be called by the user-facing module during Facebook login. This service saves the user's Facebook authorization token in the databases which will be then used for further gathering Facebook data. All database write requests are issued by this module. In order to guarantee that no duplicate data is stored, the module also sends a number of read requests.
(2) **User MVC Module**: a number of statistics and recommendations that are detailed bellow are available to a logged in user. Initially, a user has to login and authorise the application using his Facebook account. This module only sends read requests to the database.

A number of basic recommendations and statistics were written using Cypher and were made available to the users:

(1) **Global statistics and recommendations**:
   (a) *Total number of users / places / check-ins* - simple statistics counting the number of particular items stored in the database.
   (b) *Most popular categories* - for example: Bar, Pub, Italian Restaurant.
   (c) *Most popular places, by the number of check-ins* - the number of check-ins for each place is counted.
   (d) *Most popular places, by number of visitors* - some visitors place more than one check-in in the same location, so the number of unique visitors is counted.
   (e) *Most popular places, by the percentage of visitors that have returned at least once* - for each place, out of the total number of visitors, the percentage of visitors that has checked-in at the given location at least twice is counted.
(2) **Recommendations and statistics given a specific user**:
   (a) *Friends with the most check-ins* - the number of check-ins of each of the current user's friends is counted.
   (b) *Suggestions for new friends* - non-friends most often tagged in the same check-in as the current user are identified.
   (c) *Suggestions for new places to visit* - non-visited locations most often visited by the current user's close friends are identified. Close friends are identified by counting the number of check-ins in which both the friend and the given user are tagged in.
(3) **Recommendations and statistics given a specific place**:
   (a) *Similar places, based on their common visitors* - the number of common visitors the given place has with every other place is counted.
   (b) *Similar places, based on their common categories* - the number of common categories the given place has with every other place is counted. In case of ties, the number of check-ins placed by common visitors (as described previously) is also counted for every other place.

As it can be observed, for a number of the above queries, a graph-wide traversal will need to be performed.

For developing our platform, the following technologies were used: Java SE 7, Spring Data Neo4j 2.3.4, HAProxy 1.4.25 and Neo4j 2.0.3.

4. EXPERIMENTAL RESULTS

All tests were run 3 times for 25 minutes and started from the same dataset consisting of 21981 users, 48051 check-ins, 549 places and 76 categories, all linked by 392607 relationships. During each test, data from Facebook was further retrieved by 16 threads that operate by also making sure no duplicate information is stored in Neo4j. This means that, before saving new data, a worker thread needs to issue both read and write requests which will be considered as a whole and will be called a data persisting operation. For simulating client usage, 96 threads were constantly sending random statistics or recommendation requests (form the previously described list) to the system. The processing times presented where measured inside the main application, thus only considering network overhead inside the cluster (where applicable).

4.1 Neo4j Server vs Neo4j Embedded

This first performance comparison was conducted by using a single computer (to be further referenced as **PC1**), having the following configuration: 4th Gen Intel i7 @ 2.2 GHz, 8 GB DDR3 RAM @ 1600 MHz and HDD @ 5400 RPM. The size of our database is small enough for Neo4j to cache all of it in the computer's RAM memory, so the impact of a slower HDD is minimized for read operations. The results can be seen in Fig. 2.

The first test was run using Neo4j configured in a classic Server configuration, exposing a set of REST endpoints. For eliminating network overhead, the server was run on

Run	Time (ms) Data persist (write)			Run	Time (ms) Recommendations (read)		
	Server	Embedded			Server	Embedded	
1	9157	1027		1	991	1166	
2	10789	1039		2	998	1362	
3	11530	1267	Gain	3	812	1561	Gain
Average:	**10492**	**1111**	**+ 844 %**	**Average:**	**933**	**1363**	**- 31%**

Fig. 2. Neo4j Server vs Neo4j Embedded

the same machine as the application. The mean time for finalizing a write request (from the moment the application receives a data response from Facebook and up to the moment when the data is successfully committed to the database) was disappointing: 10492 milliseconds. Using a high level Neo4j REST API along with the HTTP overhead of each request greatly affects performance.

On the other hand, a very promising result of just 993 milliseconds for completing a recommendation request was obtained (with a very small percentage of worst performing queries timed just slightly above 2100 milliseconds), suggesting that the system is quite capable of handling a large number of traversals. However, because of the slow performing write operations, the graph did not suffer dramatic structural changes at a high pace, thus not forcing Neo4j to invalidate and refresh the content of its cache, which, in turn, makes high performing read operations possible.

A first improvement that we brought to the system was to configure Neo4j in Embedded mode and run it in the same Java Virtual Machine as our application. This allows Spring Data Neo4j to use low level (and thus fast) Neo4j Core API functions when performing write operations, as detailed in Hunger (2012). Data persisting operations now took only 1111 milliseconds resulting in an impressive 844% performance gain. However, recommendation

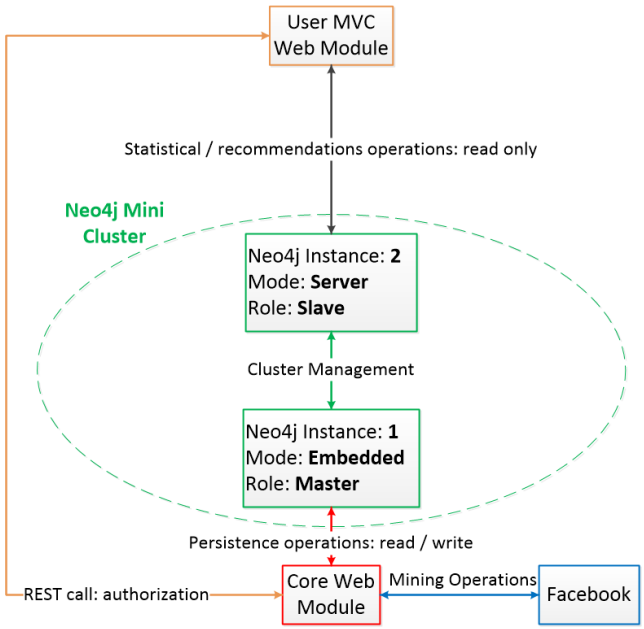

Fig. 3. Neo4j Mini-Cluster

requests now took an average of 1363 milliseconds to complete, meaning a 31% performance drop. As previously explained, because of the much higher number of write operations, this was to be expected.

4.2 Neo4j Embedded vs Neo4j Mini-Cluster

For computing recommendations that have a high probability of being relevant, a lot of data needs to be processed. This data also needs to always be up to date, so it is important for our system to quickly store new information. Although we managed to improve write times using the previous configuration, we unfortunately ended up increasing recommendation times. We decided to try and scale our system by adding an additional computer (to be further referenced as **PC2**), having the following configuration: Intel Core2Quad @ 2.83 GHz, 4 GB DDR2 RAM @ 1066 MHz and HDD @ 7200 RPM. The configuration can be seen in Fig. 3. The results can be seen in Fig. 4.

PC1 remained configured as before, with an Embedded Neo4j instance (now having the role of a HA cluster Master node) while another Neo4j Server instance was deployed as a Slave node over a Gigabit LAN on **PC2**. All recommendation requests were now sent to **PC2** that only had to handle read operations (queries). All previous tests were run again and the newly obtained results were analysed. For writing operations, a mean time of 841 milliseconds was observed, accounting for a 32% performance gain. Although **PC1** now only had to handle data persistency requests, the result is still surprising because this machine also needed to coordinate the HA database cluster. This in turn means that all data written on the Embedded instance of Neo4j needed to be replicated on the Server instance. One reason for obtaining this result is the fact that the two database instances communicate through a custom protocol that does not have as much overhead as REST calls over HTTP.

For recommendation operations, a mean time of 1186 milliseconds was obtained, meaning a 15% performance increase. However, this result is still 21% behind the result managed by our first configuration (for which, however, graph structure changes were not as frequent). Considering all the above aspects, we can conclude that, overall, this was the most performant setup of the three configurations analysed so far.

4.3 Neo4j Mini-Cluster vs Neo4j HAProxy Cluster

In attempt to further improve read operation times, we decided to enhance our mini-cluster by adding one more

Run	Time (ms) Data persist (write)			Run	Time (ms) Recommendations (read)		
	Embedded	Mini-Cluster			Embedded	Mini-Cluster	
1	1027	934		1	1166	1029	
2	1039	840		2	1362	1236	
3	1267	751	Gain	3	1561	1294	Gain
Average:	**1111**	**841**	**+ 32 %**	**Average:**	**1363**	**1186**	**+ 15%**

Fig. 4. Neo4j Embedded vs Neo4j Mini-Cluster

machine to it (to further referenced as **PC3**). This machine had the following configuration: Intel Core2Duo @ 3.16 GHz, 4 GB DDR2 RAM @ 1066 MHz and HDD @ 7200 RPM. The configuration can be seen in Fig. 5. The results can be seen in Fig. 6.

PC3 was used for running an additional Neo4j Server Slave node connected to the cluster. A HAProxy load balancer was installed on **PC2** and was used for forwarding all recommendation requests to a Slave node running on either **PC2** or **PC3**.

The tests were executed one last time, leading to the following results: for the write operations, an average execution time of 829 milliseconds was obtained, accounting for a 1% gain (which could reasonably be considered within an error margin). What is important to note is the fact that the performance of the internal communications protocol within the cluster for the Master node is not significantly impacted by the addition of an extra Slave node. Considering that the version of Neo4j used for our experiments does not support parallel write operations, it can be concluded that the only way to improve execution times would be to vertically scale **PC1**.

For recommendation requests, the new configuration managed to achieve a 38% performance gain, finalizing, on average, each request in 857 milliseconds (accounting for a

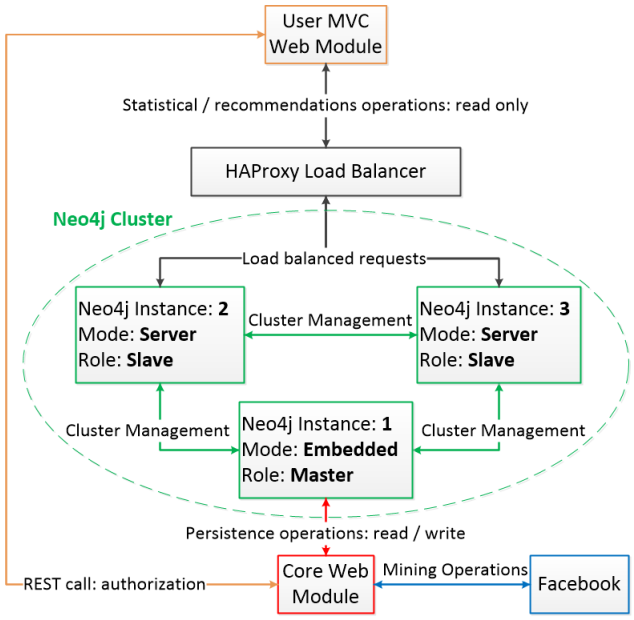

Fig. 5. Neo4j HAProxy Cluster

9% gain compared to the initial stand-alone server setup, although, as it was pointed out, this is not necessarily relevant as the database also handles much more structural changes in this final configuration). While the results are far from showing a linear performance improvement, they are still remarkable. In order for the Slave nodes to constantly catch-up with the rest of the cluster, somewhat significant computational times on **PC2** and **PC3** are used by the cluster consistency synchronization mechanisms. Near linear scaling is probably to be expected for a scenario where no data is written to the cluster. Moreover, the HAProxy load balancer was running on **PC2** meaning that the overhead for forwarding requests over the network was eliminated for this machine. Lastly, **PC3** is somewhat less powerful than **PC2**.

5. CONCLUSIONS AND FUTURE WORK

Although the initial results obtained for writing data to an ordinary Server Neo4 instance were disappointing, by progressively enhancing the system configuration, performance was greatly improved. In the end, it was shown that execution times for complex queries are quite reasonable and that the system can also be horizontally scaled without considerable effort.

Unfortunately, write operations remain the bottleneck of any Neo4j cluster configuration so future efforts could be taken in order to optimize them. An idea for a real-world application would be to store and aggregate all write requests over a period of time and perform all the operations when low incoming read request rates are detected. This is only valid for cases where postponing the write operations doesn't have a serious impact on the application.

As already mentioned, NOSQL solutions evolve very quickly. Because of this, we plan to successively upgrade to major versions of Neo4j from the last couple of years and run the initial tests again in order to assess the performance improvements of each version for our specific use-case. Moreover, we are going to further improve the cluster by adding an additional machine that will work as a dedicated load-balancer.

As query execution times were very low, as a long term focus area, the system can be used as a starting point for attempting to build a complex data analysis platform that would try and identify clusters of strongly related nodes along with their most representative entities, based on the interactions within each node cluster. Moreover, it would make sense to use this tightly interconnected clusters for

Run	Time (ms) Data persist (write)			Run	Time (ms) Recommendations (read)		
	Mini-Cluter	HAProxy Cluster			Mini-Cluster	HAProxy Cluster	
1	934	826		1	1029	986	
2	840	910		2	1236	866	
3	751	753	Gain	3	1294	721	Gain
Average:	**841**	**829**	**+ 1 %**	**Average:**	**1186**	**857**	**+ 38%**

Fig. 6. Neo4j Mini-Cluster vs Neo4j HAProxy Cluster

achieving basic cache sharding, by optimally choosing the slave instance to which to redirect requests, as previously described.

As an example, it can be observed that the friendship relationships are not annotated in any way, although it is clear that in any group of users some establish themselves as influencers while others become simple followers. These relationships can be constantly analysed and weighted. Thus communities along with their respective opinion leaders can be identified. Such information can be proven to be valuable as it would allow us to also weight individuals and their impact. This data can be used, for instance, by recommendation engines to further improve the relevance of their computed suggestions.

REFERENCES

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Comput. Surv.*, 40(1), 1:1–1:39. doi:10.1145/1322432.1322433. URL `http://doi.acm.org/10.1145/1322432.1322433`.

Balabanović, M. and Shoham, Y. (1997). Fab: Content-based, collaborative recommendation. *Commun. ACM*, 40(3), 66–72. doi:10.1145/245108.245124. URL `http://doi.acm.org/10.1145/245108.245124`.

Batra, S. and Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2).

Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y.J., and Venkataramani, V. (2013). Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 49–60. USENIX, San Jose, CA. URL `https://www.usenix.org/conference/atc13`.

Ciglan, M., Averbuch, A., and Hluchy, L. (2012). Benchmarking traversal operations over graph databases. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, 186–189. doi: 10.1109/ICDEW.2012.47.

Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazán, N., and Larriba-Pey, J.L. (2010). Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, 37–48. Springer-Verlag, Berlin, Heidelberg.

Holzschuher, F. and Peinl, R. (2013). Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, 195–204. ACM, New York, NY, USA. doi:10.1145/2457317.2457351. URL `http://doi.acm.org/10.1145/2457317.2457351`.

Hunger, M. (2012). *Good Relationships: The Spring Data NEO4J Guide Book*. InfoQ enterprise software development series. C4Media.

Jouili, S. and Vansteenberghe, V. (2013). An empirical comparison of graph databases. In *Social Computing (SocialCom), 2013 International Conference on*, 708–715. doi:10.1109/SocialCom.2013.106.

Poteraş, C., Constantinov, C., and Mocanu, M. (2011). The evolutionary design of a framework for computational steering. *Annals of the University of Craiova*, 8(2), 50–59. URL `http://ace.ucv.ro/anale/content2011vol8nr2.html`.