Performance optimization for querying social network data

Florian Holzschuher
iisys – Hof University
Alfons-Goppel-Platz 1,
95183 Hof, Germany
+49 9281 409 6214
florian.holzschuher@iisys.de

René Peinl
iisys – Hof University
Alfons-Goppel-Platz 1
95183 Hof, Germany
+49 9281 409 4820
rene.peinl@iisys.de

ABSTRACT
In this paper, we report about benchmark experiments and results from optimizing database connectivity for querying social networking data from Apache Shindig in a Neo4j database. We built on our experiments from [1] and tried to improve performance of the current RESTful http connection in comparison to JDBC in order to fully utilize performance benefits of the graph database compared to relational database management systems. We implemented a database driver based on WebSockets. We found that BSON is a better data transfer format than JSON and compression increases performance in some settings while decreasing it in others. Multiple WebSocket connections are needed to scale to a high number of client requests and fully utilize database performance. Multi-threading is another key factor for scalability. Implementing a kind of stored procedure, we were able to further increase throughput and decrease response times.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems - Query processing

General Terms
Performance, Experimentation.

Keywords
Graph query processing, social networks, performance optimization, WebSocket, graph database

1. INTRODUCTION
Graph databases are a viable alternative to relational systems and perform especially well in domains like chemistry, biology and social networking [9]. In [1] we proved Neo4j to be a superior database backend for Apache Shindig compared to the existing JPA backend and MySQL. However, it seemed that RESTful http connections between client and server perform much worse than the TCP-based, permanent JDBC connection for JPA. RESTful http is a common choice for a NoSQL database, since it facilitates access from all programming languages that are able to use http and you don't have to provide drivers for every single language. CouchDB, Riak and AllegroGraph are examples of NoSQL databases using REST as their primary interface (see nosql-database.org). On the other hand, there had to be good reasons for computer engineers some ten years ago to put considerable efforts into connection pooling and similar optimization strategies for JDBC and other database connectivity technology [12, 13, 14]. Therefore, we decided to proceed with our performance analysis and investigate different options for connecting the Neo4j backend to Shindig with a WebSocket-based driver (see Figure 1).

(c) 2014, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference (March 28, 2014, Athens, Greece) on CEUR-WS.org (ISSN 1613-0073).
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

Our goal was to identify performance tuning factors for the graph database connection, while keeping the graph query language itself stable.

The remainder of the paper is structured as follows. We first discuss related work, especially other benchmarking approaches for graph databases of the last two years. Then we present the benchmark setup, discuss the relationship to previous results and compare performance of our WebSocket approach to embedded Neo4j and Cypher over RESTful http. We continue exploring the impact of different data transport formats and compression on performance and perform a detailed analysis of time measurements. As a last step, we present results from multi-threading, clustering and multiple connections before discussing limitations, future work and finishing with a conclusion.

2. Benchmark setup
Sample data and queries were the same as in our first published benchmarks in [1]. To briefly sum up, our sample data set covers a typical Web 2.0 intranet social networking portal and contains 2011 people, 26,982 messages, 25,365 activities, 2000 addresses, 200 groups and 100 organizations. The XML file generated is 45 MB in size and contains 1.5 million lines of text. Parsed into Neo4j, this set generates around 83,500 nodes and about 304,000 relationships, consuming just over 40 MB of disk space. On average, a person has 25 friends, at least 1 and a maximum of 667 resulting in about 25,000 friendship relations in total. 90% of people have less than 65 friends whereas the median is at 12 friends. We did not use the larger datasets with more people, activities and messages used in [1], since our tests generated enough data already and no significant differences were expected. We used the same 19 queries as in [1]. They are described briefly and categorized in the appendix. Due to space restrictions, we limited the diagrams to a subset.

In contrast to our first paper [1] we did not use VMs but physical hardware this time. The client with Apache Shindig was running on a server with AMD Opteron 870 CPUs (2 GHz) with 8 cores altogether and 32GB DDR RAM (400 MHz). Neo4j was running on one to five nodes with Intel Xeon X5355 CPUs (2,66 GHz) with 8 cores altogether and 32GB DDR2 RAM (667 MHz). All servers had a Gigabit network connection and a RAID 0 hard disk, but benchmarks ran completely in memory due to a warm-up that filled the caches. Monitoring data confirmed that there was less than one disk I/O operation per second in all benchmarks.

Our software consists of a client and a server part with a WebSocket connection in between (see Figure 1). Our benchmark client is based on Apache Shindig 2.5u1, the OpenSocial reference implementation, and creates the queries (step 1). This step also includes serialization of the query. This serialized query is

then transferred to the server using a pre-existing WebSocket connection (step 2). The connection is permanent and does not have to be established and closed for every single query, which is a major improvement compared to RESTful http. The server receives the query, deserializes it and executes it against the embedded Neo4j server (step 3). The driver (server part) and Neo4j database are running within the same process.

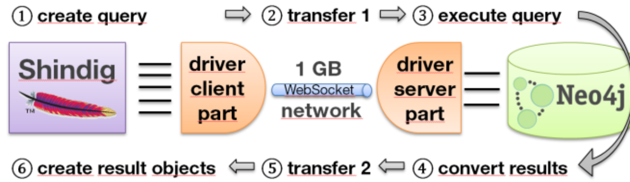

Figure 1: process of query execution

Once results from Neo4j are available, they are converted into a transferable structure (step 4). This step does not include serialization, but mainly consists of fetching additional data, since Neo4j always uses lazy loading of results. We are forcing the database to load all data, before transferring it to the client. Results are then serialized and transferred back over the network (step 5). These two steps were initially performed by the WebSocket library in one call that does both transmission and serialization using a previously defined converter class. We separated this later on in order to make better use of multi-threading for serialization. Finally, results are converted from the system independent transfer format to Shindig's object structure (step 6).

We were using Neo4j 1.9.4 as a graph database, OpenJDK 7u25 as a runtime environment, Glassfish Tyrus WebSocket library 1.2.1, json.org 20090211 and MongoDB BSON serializer 2.11.2. All servers were running Ubuntu 12.04 LTS 64 bit.

We measured response times with System.nanoTime(). "On modern hardware and operating systems, it can deliver accuracy and precision in the microsecond range. Conclusion: for benchmarking, always use System.nanoTime …" [9]. This is important, since many of our measured values are in the range of one millisecond or even below. One reason for moving from VMs to physical servers is the reliability of this measuring instrument, which only seems given for physical hardware. In order to collect network load, CPU load and memory usage, we used Monitorix[1] and modified it in order to increase time resolution from one minute to five seconds, since some of our tests only ran for two minutes.

3. Related work

In [1], we reported about our data generator for social networking data and performance comparison of several query languages.

Although we did not pay so much attention to correlation, our graph data generator follows a similar approach as [17]. We used dictionaries with real names, geographies, friendship networks and groups [1]. However, we did not intend to create big data at TB scale, but concentrate on intranet scenarios, i.e. medium to large organizations with a few thousand employees.

Although we did not crawl the data, but used the Stanford Large Network Dataset Collection as a basis instead, we followed a procedure similar to [7]. We also created a subset from a larger amount of available data that has no references regarding friendship or authorship pointing to entities outside the dataset. In addition to tweets (which we call messages), we are using activities like "Person *x* commented document *y* in System *z*", or "Person *a* rated activity *b* with three stars in System *c*". We also coincidentally use the same number of queries (19) for benchmarking, although those of [7] are analytical queries, whereas our own are operational queries used in Shindig. They are for example used to display the user profile of a person, display an activity stream or suggest interesting colleagues for "friendship" formation. [7] roughly classify queries in three categories, i.e. "*social network queries*, *timeline queries*, and *hotspot queries*". We had something similar and titled our query categories after the respective Shindig services *group*, *person*, *message*, *activity* and *graph service*. Graph service might be a misleading name, since all our queries are graph-oriented. These queries are for friend-of-a-friend (foaf) display, detection of shortest path between two people as well as friend and group suggestions. However, this doesn't seem to describe queries well enough. [2] go further than that as part of the LDBC project and perform classification based on query characteristics. They introduce the categories *select*, *adjacency*, *reachability*, *summarization* and *pattern matching*. This seems to enhance traceability and we therefore tried to categorize our queries in the same manner (see appendix). However, assignment is not always clear since several queries have more than one of the properties described in one category. [3] also classify queries regarding basic operations involved and name especially a) point reads (based on primary key), b) CRUD operations based on primary key, c) association range queries for ID, type and timestamp range ordered from latest to oldest and d) association count queries, e.g. number of friends. Our own query mix includes (a), e.g. selective message read, (c), e.g. people's friends activities and (d), although we usually fetch friend count together with top x friends. We do not benchmark write, update or delete operations. The most extensive classification is suggested in [5]. The authors present a multi-dimensional classification scheme describing the starting point (scope), reach (radius) and result of a query. We classified our queries regarding those criteria in order to make them more traceable. The result can be seen in the appendix.

We already reviewed some older graph benchmarks in [1]. [2] also benchmark Neo4j and conclude that it performs well, although a bit slower than Dex and usage of Cypher would be a viable option since it scales similarly well as the native API.

Another benchmark comparison between Neo4j and Dex is reported in [10], but they mainly use micro operations like "get vertex" or "get edge" instead of more complex queries. They found out that Neo4j scales very well for in-memory graphs, which is the case in our benchmark, but significantly loses performance when reading from disk and especially writing due to ACID transaction guarantees. They further mention that access of properties is considerably slower than access of vertices and edges for both Neo4j and Dex. We are accessing both vertices and properties in our benchmark.

[6] focus their benchmark on graph traversal operations and force the systems to perform disk I/O due to limited memory resources. They also use a graph data generator (LFR) and compare Dex, Neo4j, and four other systems. Neo4j performed well in breadth-first search with response times that are quite stable at less than 7,000 ms up to network sizes of 100,000 vertices, whereas Dex needed 15,000 ms for 10,000 vertices already. Computation of connected components on the other hand is scaling much worse, since response times increase dramatically for network sizes larger than 40,000 vertices.

A last study dealing with performance of graph databases in general and Neo4j in particular analyzes a special kind of data, namely social networking data varying over time [4]. They present a

[1] http://www.monitorix.org/

detailed data model and test ten queries. Neo4j performs well in eight of those and needs around 2,300 ms, due to a highly connected graph with up to 20,000 edges per vertex. These results are similar to ours, although distribution is even more extreme, since our slowest queries ran over 10 seconds and fastest under one ms.

4. Comparison Cypher REST vs. Cypher WS

We first discuss differences between new results for Cypher over RESTful http and embedded benchmarks compared to the previous ones published in [1], before presenting the improvements of our WebSocket implementation.

4.1 Relation to previous results

Although we tried to modify as few things as possible, the update from Neo4j 1.8 to 1.9 together with moving from VMs to physical servers influenced results. Keeping that in mind when comparing them, we still see Cypher performance improvements claimed by the vendor and anticipated in our previous paper [1] in the embedded benchmarks. In Figure 2, we show the results of the new benchmarks in relation to the respective previous ones (100%). Cypher needs only 57% of the time for running our queries compared to the results previously published. Native implementation also gains in most cases, although there are a few exceptions, where performance loss of 13% and 29% respectively can be noticed. The median is still 69%, which means that native implementation is roughly 30% faster than before.

For Cypher, results get even better, since embedded Cypher outperforms native implementation in our tests with multiple threads already at 16 threads with 1343 requests per second and reaches a maximum of 1370 req/sec with 64 threads, whereas native implementation reaches its maximum at 1323 req/sec with 128 threads. This is especially interesting, since native implementation is more than 30% faster for single threads.

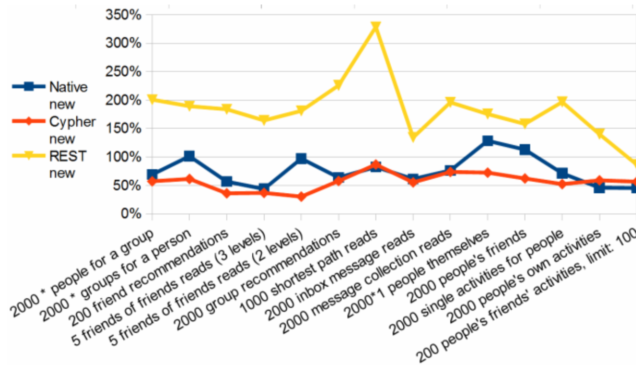

Figure 2: comparison of new results with previous ones

The downside is that although Cypher performs better than before, the connection over RESTful http got worse by nearly factor two. This further encouraged us to move on with our own connection library that should provide significant performance benefits. Part of the performance loss could be caused by the slower physical network connection between servers compared to the purely virtual connection on the VMs in our previous experiments, although our Gbps network connection's capacity was never fully utilized.

4.2 WebSocket performance

In this section, we discuss performance of Cypher queries over WebSocket with JSON (Cypher JSON) compared to Cypher over RESTful http with JSON (Cypher REST). We also introduced a kind of stored procedure, where the client only calls the procedure by name and passes parameters along (Native JSON). On the server, a native procedure is then executed. This is a higher implementation effort, but might prove worthwhile for single queries like foaf where Cypher still does not perform very well. Stored procedures using predefined Cypher queries could hardly be used due to the dynamic nature of Shindig requests. Therefore, cypher-rs[2], a Neo4j server extension for stored Cypher queries was no option. Figure 3 shows the results in relation to http performance (100%). Absolute query times for Cypher REST lie in between two and 20 seconds or in between four and 100 ms broken down to single retrievals. However, there are three exceptions, namely friend recommendations and both friend of a friend tests (foaf). Due to Cypher language constraints or inefficient implementation these queries take up to 426 seconds or between 367 and 21,330 ms on the single query level. We therefore consider them to be spikes and discuss them separately. These spikes were already present in our last test [1], although there were some performance improvements for Cypher (see section 4.1).

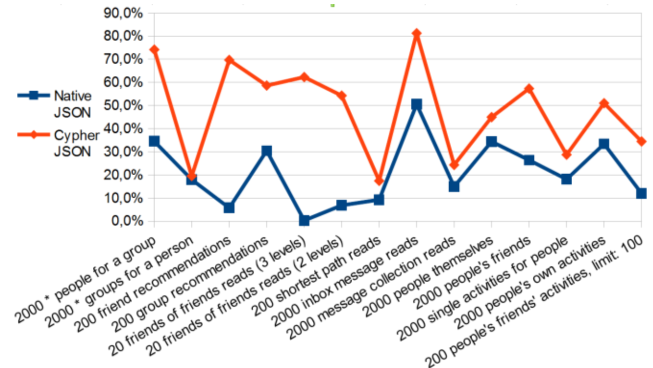

Figure 3: comparison of WebSocket and REST performance

Cypher JSON is faster than Cypher REST in all cases. Performance gains range from 83% for shortest path until 9% for inbox message reads and average at 40-50%. This is fairly good, since we were only enhancing the network connection between client and server and not the queries. However, we were inspired by the idea of stored procedures in relational database systems as mentioned above and wanted to further explore such a possibility in Neo4j. On the server, you can register your own Java implementations of such queries using Google Guice's injection mechanisms without recompiling the project. The afore mentioned spikes are good candidates for those stored procedures and the native JSON line in Figure 3 shows that performance in these cases is about ten times higher than RESTful http and even 200 times higher for foaf with three levels. A mix of normal Cypher for most of the queries, with these three "stored procedures" increased performance gains from about factor two to factor nine, which seems impressive and worth the effort.

5. Performance tuning

After these initial results, where only network connection was changed from pure http to WebSocket, we decided to investigate the impact of different choices regarding data transfer format (5.1), compression (5.2) as well as conversion from Neo4js' object model into Shindigs object model (5.3).

5.1 JSON vs. BSON

For the primary transfer format, we considered binary JSON (BSON) as an alternative to JSON, since it is type safe and we didn't need the better compatibility of JSON, since we control

[2] https://github.com/jexp/cypher-rs

both ends of the connection. Contrary to intuition, the binary BSON format produces slightly larger objects than the text-based JSON due to additional metadata regarding data types. Figure 4 shows the relative response times of our queries with BSON serialization in relation to its JSON counterparts (100%).

Native implementation gains more from using BSON instead of JSON although Cypher usually produces larger result sets and the full query in Cypher language has to be transferred instead of the name of the "stored procedure" together with parameter values in the native implementation. Therefore, one would expect that the serialization format has a larger impact there. However, native implementation is much faster than Cypher and therefore the relative impact of serialization benefits is much higher. The median for performance improvements of BSON is 44% for native implementation and 36% for Cypher. Single queries are up to three times faster for native implementation. We therefore consider BSON to be the better choice and concentrate our further results on this option.

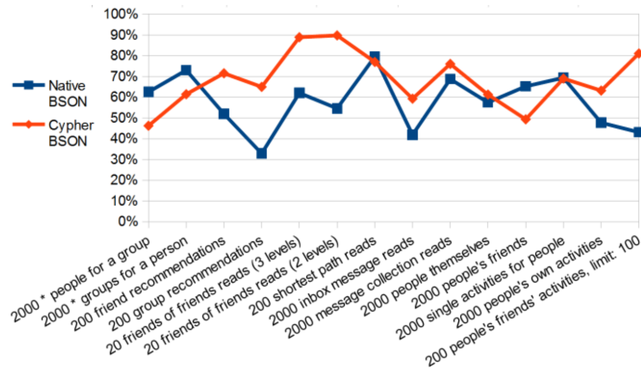

Figure 4: response time of BSON in relation to JSON

5.2 Compression

Since we are optimizing network transmission and conversion infrastructure, compression could be an interesting option. However, our WebSocket implementation does already transmit data much more efficient than the original RESTful http connection. In one of our scenarios, we measured 70 MByte/sec network load on the client for RESTful http, whereas our WebSocket implementation needed 17.4 MByte/sec only and achieved 3.8 times the number of requests/sec (see section 7.1), so that it is factor 15 more efficient. We tested the built-in Java zlib deflate algorithm with fast and best compression settings. With fast compression we were able to reduce network bandwidth usage further to 5.4 MByte/sec – again factor 3 more efficient. Best compression only marginally further reduced network load to 4 MByte/sec, but at the expense of slower operation and higher CPU load.

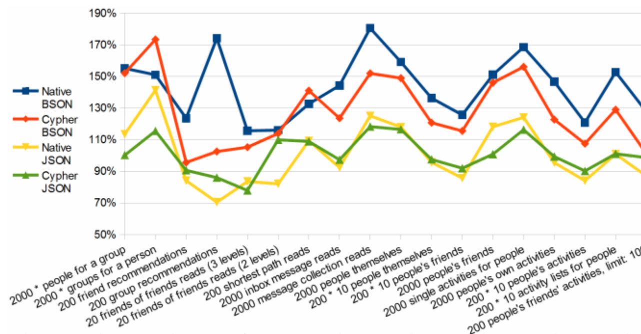

Figure 5: response time of fast in relation to no compression

Figure 5 depicts relative response times for single threaded queries and fast compression compared to no compression (100%). For BSON, fast compression achieved significantly slower answer times (30% slower for Cypher, 50% for native). In the JSON case some lower answer times and some higher times compensate more or less for each other for fast compression. Best compression performed 10% worse on average.

Astonishingly, compression is still useful in some scenarios with high throughput, although the Gbps network card is far from saturated, so that network traffic shouldn't be the bottleneck. We assume that the single threaded WebSocket implementation is the limiting factor (see section 7.3). Summed up, for 128 threads on the client and 8 threads on the server, fast compression achieved 64% more throughput and even best compression was 45% faster than no compression (see section 7 below).

5.3 Object model conversion and serialization

One annoying thing about querying the database is type conversion. With JDBC, there is a similar problem with dates and times (java.util.Calendar vs. java.sql.Date or java.sql.Timestamp). In our case the problems were arrays and lists. Neo4j internally uses arrays for multi-value properties. Both Shindig and the BSON serializer however, use java lists exclusively, although data is transferred as a BSON array. The same applies to JSON. Therefore, we had to convert at multiple points from lists to arrays and back, which is an annoying overhead, although handled by a single call of List.toArray() and Arrays.asList() respectively.

Furthermore, we identified serialization and deserialization as a potential performance bottleneck. In our initial tests, we let Tyrus (the WebSocket library from Glassfish) handle serialization of Java objects, since it conveniently performs it as part of the transfer and only requires a serializer class for the desired message format to be registered. However, this process is largely single threaded and therefore posed a limitation for scaling as soon as we introduced multiple client threads. Therefore, we chose to handle serialization ourselves before transmitting raw data using Tyrus in order to make it multi-threaded. Each client thread is creating its own converter objects due to thread safety. We considered using a pool of conversion objects instead of constantly instantiating new ones, but haven't implemented it yet.

Although we are not able to present any reliable numbers on that particular implementation detail, we are sure that it is the foundation for scaling to a large number of client requests.

6. Detailed performance analysis

After completion of the experiments described above, it seemed that uncompressed BSON was the best option and we could not further enhance performance of the connection. However, we did not have the impression to fully understand which components represented the bottleneck limiting throughput, since neither CPU nor network were used to full capacity. Therefore, we tried to investigate further and come up with more detailed time measurements.

Single processing steps shown in Figure 1 could not be logged consistently in all detail. However, we managed to capture times for steps 1+2+6 (client processing time), steps 3+4 (server processing time) and step 5 (network time). Figure 6 shows results for some of our queries and compares JSON with BSON serialization as well as fast compression and no compression. Relative times are depicted in bar charts, whereas absolute times are shown in milliseconds. Numbers above the bars are total times for the individual type of query.

Time reductions for BSON compared to JSON are achieved in both serialization and deserialization, which is included in the client and the network times depicted in Figure 6. Server times are only marginally reduced, since only deserializing the query is part of this time measure, which requires less effort than deserializing the response, which is up to 500 kB in size for friend recommendation and up to 200 kB for FOAF responses.

Compression is slower in all three parts for BSON and is especially striking in the network times that nearly double due to compression of the server response that is included in this time. Faster transfer of the reduced message size is negligible in relation to compression time when using a Gbps network connection.

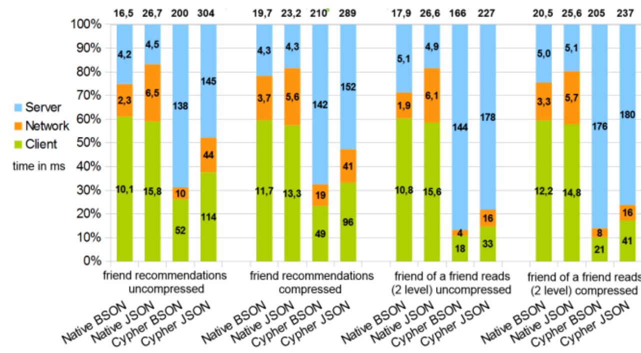

Figure 6: processing times split into client, server and net

7. Scaling tests

Having seen results for single threads, we additionally tested how threading on client and server impacted the results. We switched benchmarking measurements from time per request to requests per second. In order to better capture real relations, we limited queries to those that were not identified as spikes before (see section 4.2).

7.1 Threading

Initially, we used a single thread only in order to understand performance impacts of different options. Now, we wanted to explore the scalability with multiple threads. On the client side, we used 4, 16, 64 and 128 threads. On the server side we used powers of two up to 16. Figure 7 shows the results without compression. Numbers along the x-axis represent server threads.

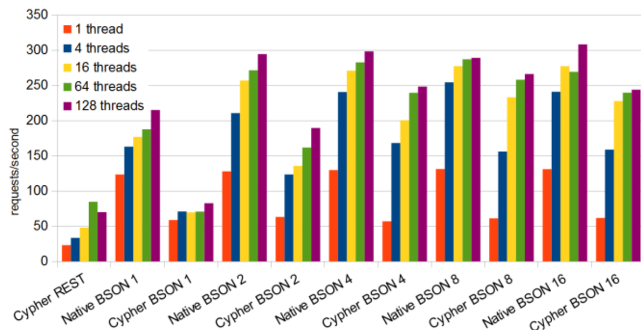

Figure 7: results with multi-threading in requests per sec.

For our native implementation the server didn't scale very well: only 37% for the step from one to two server threads and only 3% more for 16 threads. Cypher scaled better and gained an astonishing 229% for two threads but only additional 31% for the step to 4 threads. Afterwards, it gains only 7% for eight threads and even loses 8% for 16 threads. Since the server has 8 cores and neither CPU nor network are saturated, we suspected the comparably slow memory of the client and the single threaded WebSocket implementation to be the bottleneck. The good thing about the different scalability of native and Cypher implementations is that Cypher BSON reaches 86% of the maximum throughput and outperforms the old Cypher REST by more than factor three. This performance gain is in a similar order of magnitude as reported in [11].

Although we assumed that compression could not provide performance improvements in our setup, we included it in our threading benchmarks. This led to surprising results, since fast compression already proved to be superior to no compression with a single server thread and reached about 66% higher performance for eight server and 128 client threads (see Figure 8). Scaling results per se are quite similar to uncompressed results, but at a slightly higher level. Native implementation only scales well up to two server threads and only marginally gains for more threads. Cypher performance improves by a surprising 112% for two threads and an additional 57% and 50% for four and eight threads. 16 threads did not increase throughput further, which is not surprising due to the fact that the server only has eight cores.

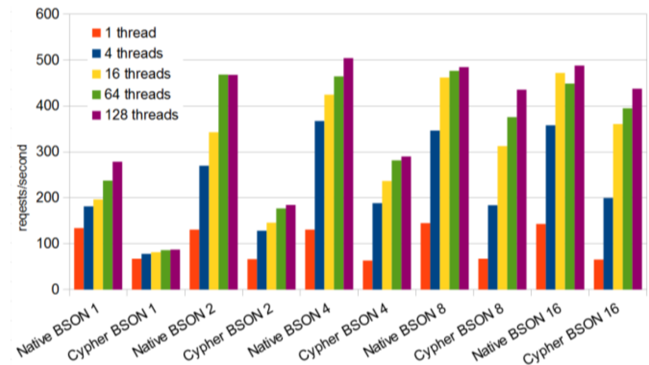

Figure 8: results with multi-threading and fast compression

7.2 Cluster

The next step was to move from a single server to use multiple Neo4j servers in a cluster. We implemented a cluster-enabled client driver that distributes requests evenly with a round robin algorithm. It works similarly to C-JDBC [12]. Based on our scaling tests (see section above) we configured the server to run with eight threads, which means one per core. This setup scales relatively well from one to three nodes, as can be seen in Figure 9.

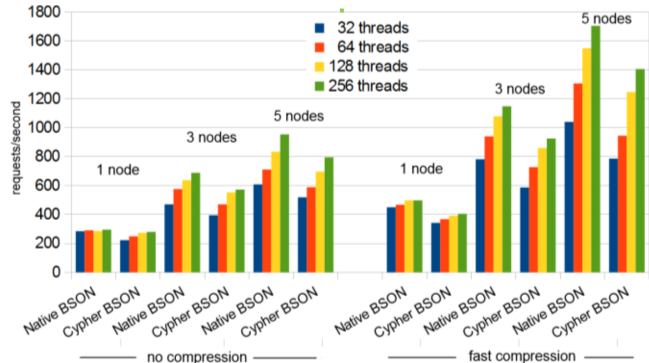

Figure 9: requests/sec with different cluster setups

Native implementation gains 133%, Cypher does not scale quite as well and reaches 105%. That still seems acceptable compared to the increase in hardware resources of 200%. Although exact numbers are not visible very well in the 3D diagrams presented in [11], it seems that our performance gains lie between MySQL and PostgreSQL with C-JDBC [11]. Performance gains for further

increasing hardware resources to five nodes are considerably lower with 39%, so that native implementation reaches 223% of single node throughput and Cypher 185%.

Compression turns out to be an important option in this scenario as well. Even with one node, throughput is between 43% (Cypher) and 74% (native) higher with fast compression than without compression. Maximum throughput with five nodes and 256 client threads is even 77% (Cypher) and 79% (native) higher than that without compression. This is mainly due to better scaling from three to five nodes. Whereas performance gains for moving from one to three nodes is similar for fast compression to no compression (131% for native and 129% for Cypher), the step from three to five nodes leads to another 48% (native) and 52% (Cypher) performance improvement for fast compression, which is significantly higher than the 39% for no compression.

7.3 Multiple WebSocket connections

We finally tried to use multiple WebSocket connections (conns) between client and server to get either CPU or network fully utilized. We directly aimed for eight connections and skipped tests for two and four, but varied the number of server threads per connection. Figure 10 shows results from one server node, 128 client threads, the depicted number of connections and server threads per connection. Results are shown for Cypher BSON and Native BSON without compression (left bar group) and fast compression (right bar group).

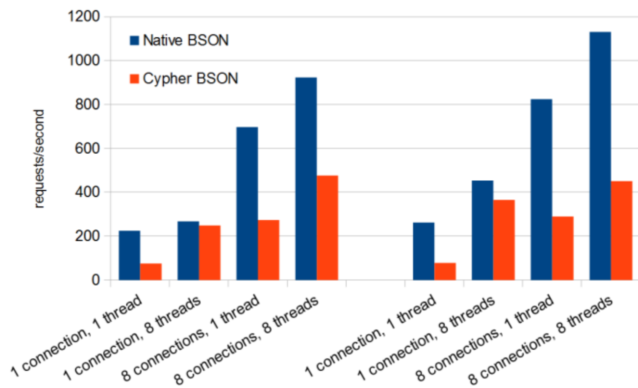

Figure 10: requests/sec with multiple WebSocket connections

It turned out, that this was indeed the limiting factor and none of our previous measurements had revealed that. Results for native implementation without compression, eight connections and eight server threads per connection outperforms both three node (45% higher throughput) and even five node cluster results (11% higher throughput) with one connection and otherwise identical settings. For fast compression at least the three node cluster is beaten by 5%. Unfortunately, Cypher does not benefit equally from multiple connections. Where native implementation gains impressive 245% when moving from one to 8 connections with 8 server threads (uncompressed), Cypher gains only 91% and therefore reaches only 52% of native throughput. This is due to CPU usage, as shown in Figure 11.

Small usage spikes at the beginning represent warm-up procedure. Then native tests starting briefly before 16:30 are utilizing all eight CPUs at roughly 60% with spikes up to 75%. Then again a warm-up is run and Cypher tests start around 16:32:30. CPU load is near 100% there. Looking at single CPU cores, we see that all except one core are saturated at 100% and the last one at 90%. This state is nearly reached for native implementation with fast compression, where global CPU load averages at 95%. Network load is not the limit. It reaches 50 MByte/sec without compression and 12 MByte/sec for fast compression. We did not manage to run all the multi-connection tests in the cluster, but gave the most promising constellation a shot and achieved **2544 req/sec** for 3 nodes Native BSON and fast compression (2489 req/sec for 5 nodes) and 1824 req/sec for 5 nodes with Cypher BSON (1266 req/sec for 3 nodes). Results without compression were lower.

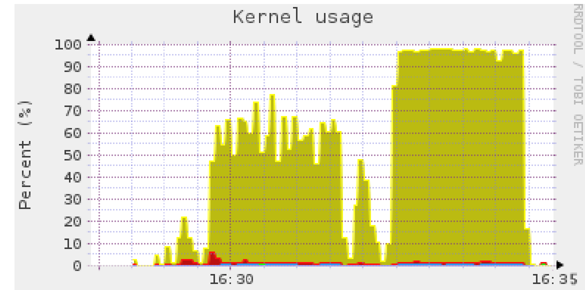

Figure 11: global kernel usage on the server (Monitorix)

That means that Cypher scales nearly linearly with the number of nodes (300% nodes => 281% performance, 500% nodes => 404% performance) with multiple WebSocket connections. This is better than the results for a cluster with a single WebSocket connection per server (see Table 1). Native implementation, on the other hand, reaches a limit at the three node cluster setup (225% performance) and does not scale further (220% for 5 nodes). The client was not able to issue more requests.

Table 1: throughput for cluster with single and multiple conns

	Native	Cypher
3 nodes, single conn	1.148 r/s (231%)	926 r/s (229%)
3 nodes, 8 conns	2.544 r/s (225%)	1.266 r/s (281%)
5 nodes, single conn	1.705 r/s (342%)	1.405 r/s (347%)
5 nodes, 8 conns	2.489 r/s (220%)	1.824 r/s (404%)

8. Limitations

Our benchmark queries are directly derived from Apache Shindig. However, some of them are only present in our extended version of Shindig that supports friend and group recommendations as well as shortest path analysis like many other social networks provide them (e.g. Xing). The single queries are not weighted based on frequency of use in normal user scenarios.

Furthermore, we did not make any efforts to optimize RESTful http, so that it uses one thread per request, which leads to some overheads. We did not consider using an asychronous event-driven implementation instead [16] that might positively impact performance when carefully tuned [15]. We didn't test alternative JSON serialization libraries, although we were pointed to an interesting resource lately, showing very fast implementations[3].

We also did not test all alternatives with multiple WebSocket connections, since some early tests with this feature proved unsuccessful so that we did not consider it for inclusion up to a few days before submission. When we remembered to retest it with our other improvements and physical machines it turned out to make a big difference. Therefore, we could only conduct the few tests discussed here instead of the full suite. Finally, it would have been desirable to have different hardware options in order to gain further insight into how different CPU speeds and cores affect

[3] https://github.com/eishay/jvm-serializers/wiki

overall performance or whether speed of system memory really is a limiting factor in some tests.

9. Future work

In parallel to our work described here, Neo Technology is finishing work on Neo4j 2.0. Keen on any enhancements the new version is providing, we did a preliminary test with Neo4j 2.0 M06 dating from 15th of October 2013. We found that due to introduction of multiversion concurrency control (MVCC), neither the native implementation nor Cypher queries ran without changes. We had to make significant changes and to introduce at least one transaction for every query. This leads to decreased performance in most of our tests. Native implementation loses the most with 19% in embedded and 30% in WebSocket BSON benchmark. For Cypher there have been further language optimizations so that some of the Cypher tests gained performance, most notably foaf (20-50%) as well as friend and group recommendation (20%-30%). Still, overall performance decreased here as well losing 12% in embedded and WebSocket and 7% in our REST benchmark. These results have to be interpreted with caution, since we did not use all new query features and did not optimize our queries and algorithms for the new version. All we did were changes to get queries running. There is e.g. a whole new transactional http API that finally gets rid of the superfluous URLs that were delivered with every result and led to the tremendous overhead reported in [1]. We have not investigated this new endpoint yet.

Being confident that we optimized the network connection quite well, we plan to further explore end-to-end performance of Shindig together with the database and use jMeter to query the Shindig REST API instead of using our own benchmarking tool. It could be the case that Shindig is not able to benefit from our optimizations due to own internal inefficiencies. Another influence could come from switching from running a standalone client and server to running them in Apache Tomcat and Glassfish.

We also plan to use much higher volumes of data, so that Neo4j has to access disks, instead of caching all data in main memory.

Finally, we aim at including benchmarks for writing data to Neo4j, since all our current queries are read-only. Neo4j provides single master replication within all nodes of an enterprise cluster, which we were using in our cluster tests. It will be interesting to see how well write operations scale and how fast replication between nodes really is.

10. Conclusion

We've presented results from optimizing the database connection to Neo4j for querying social networking data from Apache Shindig. We've thoroughly analyzed several options for the transfer including JSON vs. BSON as a data transfer format, different compression options, multiple WebSocket connections as well as multi-threading on client- and server-side for achieving the highest possible throughput. We then went from a single server database to a cluster of three and five nodes in order to analyze scalability. Results show that BSON is more efficient than JSON, especially regarding (de-)serialization. Compression reduces network load significantly and performs better for a high number of client requests. Multiple WebSocket connections increase maximum throughput significantly (up to 245%).

The database cluster is able to increase throughput and achieves a maximum of 181% performance increase for 200% additional server resources with Cypher. Scaling to five servers did not result in better throughput for native implementation. It might be the case that the client was the limiting factor there. Cypher however, was able to gain an additional 44% of performance compared to three nodes which lead to a 304% performance increase altogether in comparison to a single node. Table 2 summarizes results of our test in relation to Cypher over RESTful http. Values in parentheses represent the performance relative to REST.

Summed up, we can state that it is worthwhile to pay attention to network connectivity between application server and database server. We are convinced that many NoSQL databases are experiencing similar problems causing them to not fully expose their internal performance over a standard RESTful http interface. It would be interesting to measure performance of our approach compared to RexPro and Rexster from the tinkerpop project, that aim at providing a kind of JDBC-like standardization to graph databases. With systematic analysis and consequent enhancement of our database driver, we were able to increase speed by factor 5.6 for Cypher and up to 13.3 when using our "stored procedures" that ran native queries on the database server. When considering extreme graph queries like three level foaf, performance differences are even higher.

Table 2. Summary of results

	REST	Native BSON	Cypher BSON
single thread response time	100%	17%	38%
128 threads throughput (r/s)	70	215 (3.1x)	83 (1.2 x)
Max 1 node throughput (r/s)	85	1131 (13.3x)	476 (5.6x)
Global max throughput (r/s)	85	2544 (29.9x)	1824 (21.5x)

Although the benchmark is specific to Apache Shindig, the Neo4j driver is generic and can be used in any project. As an intended side effect of our efforts, open source projects with a more liberal license like APL v2 can now use GPL v3 licensed Neo4j, without fear of the viral GPL, since a pure network connection separates client and server part of our driver, so that GPL does not affect client code. This is another major improvement compared to our efforts in [1]. Therefore, we hope that our Neo4j backend will soon become the default for Apache Shindig.

11. REFERENCES

[1] Holzschuher, F., and Peinl, R. 2013. *Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j*. Joint EDBT/ICDT Workshop GraphQ 2013. Genoa, Italy. 22.03.2013. 195-204

[2] Angles, R., Prat-Pérez, A., Dominguez-Sal, D., and Larriba-Pey, J. L. (2013) *Benchmarking database systems for social network applications*. 1st International Workshop on Graph Data Management Experiences and Systems. ACM. 15

[3] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. (2013) *LinkBench: a Database Benchmark Based on the Facebook Social Graph*. ACM SIGMOD, June 2013. 1185–1196

[4] Cattuto, C., Quaggiotto, M., Panisson, A., and Averbuch, A. (2013) *Time-varying social networks in a graph database: a Neo4j use case*. In 1st International Workshop on Graph Data Management Experiences and Systems. ACM. 11

[5] Grossniklaus, M., Leone, S., and Zäschke, T. (2013) *Towards a benchmark for graph data management and processing*. Technical Report KN-2013-DBIS-01, University of Konstanz, Department of Computer and Information Science

[6] Ciglan, M., Averbuch, A., and Hluchy, L. (2012) *Benchmarking traversal operations over graph databases.* In Data Engineering Workshops (ICDEW 2012). 186-189. IEEE.

[7] Ma, H., Wei, J., Qian, W., Yu, C., Xia, F., & Zhou, A. (2013) *On benchmarking online social media analytical queries*. In 1st International Workshop on Graph Data Management Experiences and Systems. ACM. 10

[8] Boyer, B. (2008) *Robust Java benchmarking, Part 1- Understand the pitfalls of benchmarking Java code*. http://www.ibm.com/developerworks/java/library/j-benchmark1/index.html

[9] Miller, J. J. (2013). *Graph Database Applications and Concepts with Neo4j*. Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th, 2013.

[10] Macko, P., Margo, D., and Seltzer, M. (2013). *Performance introspection of graph databases*. In Proceedings of the 6th International Systems and Storage Conference. ACM. 18

[11] Unde, P., Vin, H., Natu, M., Kulkarni, V., Thomas, D., Vasudevan, S., and Pathak, R. (2012) *Architecting the Database Access for a IT Infrastructure and Data Center Monitoring tool*. In IEEE 28th International Conference on Data Engineering Workshops (ICDEW), 2012. 351-354

[12] Cecchet, E. (2004) *C-JDBC: a Middleware Framework for Database Clustering*. IEEE Data Engineering. Bulletin. 27(2), 19-26.

[13] Karlsson, M., Moore, K. E., Hagersten, E., & Wood, D. A. (2003). *Memory system behavior of Java-based middleware*. In 9th int. Symposion on High-Performance Computer Architecture,. HPCA-9 2003. (pp. 217-228). IEEE.

[14] Li, Y., & Lü, K. (2000). *Performance issues of a Web database*. In Database and Expert Systems Applications (pp. 825-834). Springer Berlin Heidelberg.

[15] Malkowski, S., Jayasinghe, D., Hedwig, M., Park, J., Kanemasa, Y., & Pu, C. (2010). *Empirical analysis of database server scalability using an n-tier benchmark with read-intensive workload*. In Proceedings of the 2010 ACM Symposium on Applied Computing. 1680-1687.

[16] Harji, A. S., Buhr, P. A., & Brecht, T. (2012). Comparing high-performance multi-core web-server architectures. 5th Annual International Systems and Storage Conference (p. 2).

[17] Pham, M.-D., Boncz, P. & Erling, O. (2012) S3G2: A Scalable Structure-Correlated Social Graph Generator. 4th TPC Technology Conference, Istanbul, Turkey, 27.08.2012

Appendix: Query classification

Query	Description	Scope	Radius	Result	Type
2000 * people for a group	Group x => members	node	neighbors	nodes	select
2000 * groups for a person	Person x => group membership	node	neighbors	nodes	select
200 friend recommendations	Person x => friend => friend[not x’s friend] sort by friends in common	node	neighbors	nodes	pattern matching
200 group recommendations	Person x => friend => group[not x’s group] sort by num friends with this group	node	neighbors	nodes	pattern matching
20 friends of friends reads (3 levels)	Person x => friend => friend => friend [not x’s friend]	node	neighbors	nodes	pattern matching
20 friends of friends reads (2 levels)	Person x => friend => friend [not x’s friend]	node	neighbors	nodes	pattern matching
200 shortest path reads	Person x, y => shortest path between	path	path	subgraph	reachability
2000 inbox message reads	Person x => message collection[inbox] => messages	node	neighbors	nodes	select
2000 message collection reads	Person x => message collections incl. num messages per collection	node	neighbors	nodes	summarization
2000 people themselves (profile page)	Person x => all first level properties => some second level properties	node	neighbors	subgraph	adjacency
200 * 10 people themselves	Person a, b, …, j => all first level properties => some second level properties	subgraph	neighbors	subgraph	adjacency
200 * 10 people's friends	Person a, b, …, j => friends	subgraph	neighbors	nodes	adjacency
2000 people's friends	Person x => friends	node	neighbors	nodes	adjacency
2000 single activities for people	Person x => activities[a]	node	neighbors	nodes	select
2000 people's own activities	Person x => activities	node	neighbors	nodes	adjacency
200 * 10 people's activities	Person a, b, …, j => activities	subgraph	neighbors	nodes	adjacency
200 * 10 activity lists for people	Person x => activities[a, b, …, j]	node	neighbors	nodes	select
200 people's friends' activities, limit: 100	Person x => friends => activities => sort by created descending [1..100]	path	neighbors	nodes	adjacency