Performance of graph query languages: Comparison of cypher, gremlin and native access in Neo4j

Conference Paper · March 2013

DOI: 10.1145/2457317.2457351

2 authors, including:

Rene Peinl
Hof University of Applied Sciences
71 PUBLICATIONS **414** CITATIONS

SEE PROFILE

Performance of Graph Query Languages

Comparison of Cypher, Gremlin and Native Access in Neo4j

Florian Holzschuher
Institute for Information Systems (iisys)
Hof University
Alfons-Goppel-Platz 1
DE-95028 Hof, Germany
florian.holzschuher2@iisys.de

Prof. Dr. René Peinl
Institute for Information Systems (iisys)
Hof University
Alfons-Goppel-Platz 1
DE-95028 Hof, Germany
rene.peinl@iisys.de

ABSTRACT

NoSQL and especially graph databases are constantly gaining popularity among developers of Web 2.0 applications as they promise to deliver superior performance when handling highly interconnected data compared to traditional relational databases. *Apache Shindig* is the reference implementation for *OpenSocial* with its highly interconnected data model. However, the default back-end is based on a relational database. In this paper we describe our experiences with a different back-end based on the graph database *Neo4j* and compare the alternatives for querying data with each other and the *JPA*-based sample back-end running on *MySQL*. Moreover, we analyze why the different approaches often may yield such diverging results concerning throughput. The results show that the graph-based back-end can match and even outperform the traditional *JPA* implementation and that *Cypher* is a promising candidate for a standard graph query language, but still leaves room for improvements.

Categories and Subject Descriptors

H.2.3 [**Information Systems**]: Database Management, Languages [query languages]; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Graph query processing and performance optimization, Graph query processing for Social Networks

Keywords

NoSQL, graph databases, Neo4j, benchmarks

1. INTRODUCTION

Relational databases have been the means of choice for storing large amounts of structured data for applications for decades due to their high performance and ACID capabilities. With requirements changing due to transformation of the IT world caused by the social Web and cloud services, several types of NoSQL databases have emerged and are gaining popularity [19]. Among those, graph databases are especially interesting since they often offer a proper query language, which most key-value stores as well as document-oriented databases are currently missing.

Particularly Web 2.0 data in social networks is highly interconnected, e.g., networks of people, comments, ratings, tags and activities. They are forming acquaintance networks, communication networks and topic networks, just to name a few. Modeling such graphs in a relational database causes a high number of many-to-many relations. Complex join operations are needed to retrieve such data. Graph databases on the other hand are specifically designed to store such data and to deliver high performance traversing them.

To assess the suitability and performance of a graph database-based solution in a real world use case, we chose to implement different back-ends for *Apache Shindig*, the *OpenSocial* reference implementation, and measure their performance with realistic sets of data, as well as judging their practicability. Our use case considers a social Web portal within an intranet scenario, where user data is accessed in user profile pages, short messages can be sent and people's activities are shown in an *Activity Stream*.

For our effort, we chose *Neo4j*, a *Java*-based open source graph database that offers persistence, high performance, scalability, an active community and good documentation. Furthermore, two different query languages can be used to access data in *Neo4j*, *Cypher*, which is declarative and a bit similar to SQL, as well as the low level graph traversal language *Gremlin*. We compare both regarding performance, understandability and lines of code against native traversal in *Java* as well as SQL queries generated by JPA and determine how well they work with a network-connected database.

The remainder of this paper is structured as follows. We review some related work in section 2 before we explain the test setup and sample data in section 3. We move on to a comparison of query languages regarding readability and maintainability (section 4), before we present the benchmark results and anylze them in detail in section 5. Finally, we sum up our lessons learned in sections 6.

2. RELATED WORK

Our paper builds upon previous work in three main areas,

EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy

which are NoSQL databases and especially graph databases, query languages and again especially graph query languages, as well as benchmarks, especially those comparing graph databases and relational ones.

2.1 NoSQL databases

Mainly the advent of cloud computing with its large scale Web applications drove the adoption of database systems that are non-relational. The CAP theorem suggests, that databases can only perform well in two of the three areas consistency, availability and partition-tolerance [3]. While relational database systems (RDBMS) prefer consistency following the ACID principle[1] and availability using cluster solutions, they are not partition-tolerant which means their scalability is somewhat limited. On the other hand, NoSQL databases do either trade availability or consistency in favor of partition-tolerance. The former is due to blocking replication operations so that consistency can be maintained (e. g., MongoDB, Redis). The latter is due to possible inconsistencies like one client still reading old data despite another client having already updated the record on another server (e. g., CouchDB, Cassandra).

Several types of NoSQL DBMS can be distinguished [19]. Key-value stores are quite simple and implement a key to value persistent map for data indexing and retrieval (e. g., Redis and Voldemort). Wide-column stores or extensible record stores are inspired by the BigTable model of Google (e. g., Cassandra and HBase). Document stores are tailored for storing semi-structured data, like XML documents or especially JSON data[2] (e. g., MongoDB and CouchDB). Finally, graph databases are well suited to store graph-like data such as chemical structures or social network data. Examples are Allegro GraphDB and HypergraphDB. The latter ones are sometimes not counted as proper NoSQL databases, since some of them are quite similar to RDBMS and also favour consistency and availability instead of partition-tolerance [4]. However, *Neo4j* has a blocking replication mechanism that is used in the high availability cluster setup, which means it behaves like MongoDB mentioned above.

Although not covered by the CAP theorem, many NoSQL databases are not only neglecting consistency to achieve their goal, but also do not help the developer in querying the data stored in the DBMS. [17] state that this is not necessary and advise clients to chose a DBMS that offers a high level language which can still offer high performance.

2.2 Query languages

Query languages have always been key to success of database systems. The prevalence of relational database systems in the last decades is tightly coupled with the success of SQL, the structured query language [5, 18]. Soon after a new type of database system arose, a query language was invented to query data in the respective format, e. g., XPath and XQuery for XML databases [11], OQL for object-oriented databases [1] or MDX for multi-dimensional databases [15]. In case of querying RDF data, there even was a serious contest between multiple query languages like RQL, RDQL and SeRQL that competed to become the W3C standard [8], before SPARQL [14] finally arose as the winner of this battle.

Besides those RDF graph query languages, which are implemented in RDF triple stores like Jena or Allegro GraphDB, there are also proposals for other graph query languages that can be used for general purpose graph databases. GraphGrep for example, is an application independent query language based on requlars expressions [7]. GraphQL was proposed in 2008 [9] and introduced an own algrebra for graphs that is relationally complete and contained in Datalog. It was implemented in the sones GraphDB [2] which achieved some attention in Germany before the company behind it went bankrupt in 2012. [2] do also mention *Cypher*, the main query language of *Neo4j*, which is a declarative language similar to SQL.

In his comparison of *Neo4j* to Allegro, DEX, Hypergraph, sones and other graph databases, Angles stated that *Cypher* supports five out of seven types of graph queries that are relevant based on his literature review [2]. The first type are adjacency queries, which test whether two nodes are connected or are in the k-neighborhood of each other. The second type are reachability queries that test whether a node is reachable from another one and further more, which is the shortest path between them. The third type are pattern matching queries and especifically the subgraph isomorphism problem, as only this one can be solved in finite time. The final type are summarization queries that allow some kind of grouping and aggregation. [2] state that only k-neighborhood and summarization are not supported by *Neo4j*. The first is also missing in all other databases tested. The latter issue has been fixed in *Neo4j* version 1.8 which supports count, sum, max, min and avg aggregation functions [13]. Finally, *Neo4j* also provides support for *Gremlin*, a domain specific language for traversing property graphs developed within the TinkerPop open source project[3].

2.3 Benchmarking

A first comparison of *Neo4j* with *MySQL* was already published in 2010 [20]. The comparison includes benchmarks and also subjective measures like maturity of the systems, ease of programming and security. Results showed that *Neo4j* is performing quite well on string data, while being considerably slower than *MySQL* on integer data. However, the comparison was using *Neo4j* v1.0 b11, which was not surprisingly considered less mature than *MySQL* 5.1.42 and the data used was highly artificial as access to random string and integer data was benchmarked. [10] stress the importance of using realistic data in order to generate useful results and reference the TPC benchmark series as a good example. In our own comparison, we therefore put considerable effort in creating realistic data for our social Web portal scenario and are using queries that would occur in daily operation of such a portal (see section 3.2). [12] name a few requirements that a serious graph benchmark should fulfill. They argue that traversal of the graph is a key feature of graph databases and should therefore be included in the benchmark. Additionally, they discuss the influence of being able to cache the whole graph in main memory and conclude that cases should be tested, where this is not possible. On the other hand, with current server hardware it is no problem to provide 64 GB RAM or more and a typical social Web portal with a few thousand users will hardly generate such high amounts of data. Our database sizes for 2,000 and 10,000 users are only 40 MB and 200 MB respectively. Finally, they highlight the importance of using fair measures, which is why we are not overstressing highly graph-related queries

[1] atomicity, consistency, isolation, durability

[2] Javascript Object Notation

[3] http://tinkerpop.com/

such as friend-of-a-friend (FOAF) or group recommendation queries, but also use more simple queries like all attributes of the user profile. The latter ones should favor the graph DBMS less and therefore provide a fair judgement of the system and the query language.

A more recent comparison is documented in [16]. They reported query times of *Neo4j* being 2–5 times lower than *MySQL* for their 100 object data set and 15–30 times lower for their 500 objects data set. Queries executed were friendship, movie favourites of friends and actors of movies favourites of friends, representing increasingly interconnected data. Results confirmed the intuitive hypothesis that differences between *Neo4j* and *MySQL* were increasing with the number of joins or edges respectively.

Neo4j has also been compared to other graph databases in [6]. They implemented the HPC scalable graph analysis benchmark and concluded that *Neo4j*, together with DEX are the most efficient graph databases. They ran tests with 1k, 32k and 1M nodes reaching from 9k relationships to 8.4 million. Compared to that, our databases had 83.5k nodes with about 304k relationships and 350k nodes with 1.5 million relations respectively. [6] reported that they had already problems loading the data into the database as it took in some cases more than 24 hours. In our tests, *Neo4j* loaded the big data set from our XML file in less than one minute using our self-developed loader resulting in a 200 MB database file compared to 687 seconds for 32k nodes (17 MB database) and 32,094 seconds for 1M nodes (539 MB database) in the referenced source. Our *Neo4j* results are therefore within the range reported for DEX (317 seconds for 1M nodes and 893 MB database). [6] further report that in order to achieve a fair comparison, a warm up process should be conducted, which we did in our case.

3. TEST SETUP

In this section we describe in detail which software and technologies we used for our benchmark, which kind of sample data we generated and how, as well as how we proceeded to retrieve our results.

3.1 Software and Technologies

For our performance tests, we used *Neo4j* 1.8 which has a native Java API that offers direct retrieval and traversal methods as well as a traversal framework and some predefined algorithms for convenience. It is directly accessible when *Neo4j* is running in embedded mode, within the same process as the application using it. Furthermore, there is a RESTful Web service interface transferring JSON data over the network for remote operations together with wrappers for some programming languages such as *Java* and *Python*. These wrappers offer the same API as the native implementation but at the time of the benchmarks some traversal functionality was still missing.

Since version 1.8, *Cypher*, *Neo4j*'s own declarative query language, can be used for CRUD operations, even though it lacks some optimization work which is planned for the 1.9 release. Queries can be executed over both interfaces but at the moment they are mostly useful when working with a remote server as they can be remotely executed with only results being returned.

Finally, *Gremlin* from the *Tinkerpop* project is also available as a *Groovy*-based query language. It follows an imperative paradigm and is said to deliver a higher performance for very simple queries.

We used *Apache Shindig* 2.5 beta as the basis for our social Web portal, since it incorporates some major improvements regarding support of *OpenSocial* 2.0. However, it has by default a *JPA* back-end, delivering only part of the data needed for *OpenSocial* and our use case. For our measurements, only the service delivering person data was suitable. The activity service was not used as it only offers activities in a deprecated format of OpenSocial 1.0 instead of adopting the activitystrea.ms format used by OpenSocial 2.0. The implementation was slightly modified to enable us to use incomplete data, which would have caused null pointer exceptions during serialization otherwise. Moreover, we added routines to update the profile data relevant for the benchmark, which were missing at that time. *JPA* and *Hibernate* allow using a large number of relational databases. We chose *MySQL* 5.5.27 and accessed it via *Hibernate* 3.2.6. Both were running on the same virtual machine.

Besides the standard *OpenSocial* interfaces for *Shindig*, we defined additional RESTful interfaces that give access to advanced social networking features like friends of friends, shortest path over friendships and friend or group recommendation based on friends' relations and memberships.

3.2 Sample Data

To enable us to properly evaluate the back-end performance in our use case, we needed realistic sets of sample data in terms of complexity and size. For this purpose we wrote a generator that is able to create data sets with random person, organization, message and activity data based on anonymized friendship graphs and store it in an XML file.

As sources of data we used lists of common first[4] and last names[5], street names[6], and geographical data from geonames.org[7]. Furthermore we created lists of possible group names[8], interests[9], job titles and organization types. Activities and messages largely consist of randomly generated, but meaningful text with some additional meta data. The anonymized relation graph is sourced from the network data provided at *Slashdot* from the 2008 *Stanford Large Network Dataset Collection*[10].

Our sample data set contains 2011 people[11], 26,982 messages, 25,365 activities, 2000 addresses, 200 groups and 100 organizations. We also tested a larger dataset with 10,003 people and the respective amount of other nodes (see below). The data structure of our sample data is shown in tables 1 and 2. Actor, object, target and generator for activities are activity objects that only have a type and a name.

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

[4] http://german.about.com/library/blname_Boys.htm, http://german.about.com/library/blname_Girls.htm
[5] http:/de.wiktionary.org/wiki/Wiktionary:Deutsch/Liste_der_h%C3%A4ufigsten_Nachnamen_Deutschlands
[6] http://blog.christianleberfinger.de/?p=17
[7] http://download.geonames.org/export/zip/
[8] http://vereins.wikia.com/wiki/Kategorie:Vereine
[9] http://www.massmailsoftware.com/database/categories.htm
[10] http://snap.stanford.edu/data/
[11] the uneven number results from the use of real social networking data and the need for a fully connected graph

Table 1: Data for people and activities

Person	Activity
first and last name	title
birthday, age	body
gender	verb
interests (2–5)	time stamp
messages (2–25)	actor
affiliations with organizations (0–3)	object
addresses (1–2)	target
group memberships (0–5)	generator
activities (1–25)	

Table 2: Data for organizations, messages and addresses

Organization	Message	Address
name	title	street name
sector	text	house number
website	time stamp	city name
address	1–3 recipients	zip code
		state
		country
		longitude, latitude

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.

In order to test the back-ends' scalability two larger data sets were created, one with 10,003 people, about five times the normal amount and one with five times as many messages and activities per person resulting in roughly 200 MB disk space usage. Coming from real data, the set with more people differs from the first one in terms of statistics. There are now 137,000 friendships in total, with a maximum of 1,448 for one person, an average of 28, a median of 10 and a 90 % percentile of 76 friends.

Based on *Apache Shindig*'s object model, we created a graph database model for *Neo4j* and optimized it to better harness the database's capabilities. This means many entities that were integrated into others on the object level were extracted into their own nodes and connected through relationships. In total, our database model has 13 different types of nodes and 32 relation types. This only defines how our applications accesses the data since *Neo4j* is a schema-free database.

For example, in *Shindig*'s object model a person object directly contains lists of addresses, accounts and organizations, all of which are modeled using separate nodes and relations. Moreover, a person in the graph is directly linked to messages, so sender and recipients can be determined through relations instead of stored ID values among the message's properties. People are also connected among each other to model friendships and provide direct tagging support.

3.3 Procedure

The benchmark itself is separated into several suites, each responsible for running individual tests on a certain subset of data. The focus of our routines is on person and activity data as they are particularly important for our scenario. Since the implementation of the conversion between the data transfer objects delivered by the database and the object model prescribed by Shindig was a lot of work for *Cypher* and *Gremlin*, we initially ran only person queries to get an impression of the performance and then further investigate the most promising candidates. For *MySQL*, the conversion was done by Hibernate but *Shindig's* implementation of *OpenSocial* 2.0 and especially activity data was not complete, so we had to add some code here.

We used Shindig's default service interfaces to retrieve data using one of the alternative back-ends and query languages. We disabled all security checks and did not use any sorting functions. Each timed call ended with a fully converted result object, whose properties were accessed in code in order to assure their availability and trigger any lazy loading operations that might be present in the JPA back-end.

Before each run, our parser loads the sample data into the database. Neo4j's local batch mode is used due to its superior performance whereas data is inserted into *MySQL* using *JPA* operations. Since the first benchmark results already showed the superior performance of *Neo4j* and it would have required a lot more work to do a complete implementation of retrieval and storage procedures for *MySQL* this work was discontinued so only data relevant for person retrieval was actually inserted.

A warm up routine reads all data available once for every implementation, filling any caches present before every benchmark run. This should eliminate slow downs caused by cold start as this is not a typical scenario for a system that is running 24x7 and used on a day to day basis. Since our data is rather small, this means the whole data could be in memory caches after the warm up.

Before each run, a random sequence of request data is generated, in order to reduce influences of the stochastic generation process on the results.

All routines measure the time needed for several sequential requests together, as single requests often take less than a millisecond. This could have led to problems during measurement. Furthermore this already creates an average which can be useful for determining the average turnaround time. Additionally, benchmarks are executed several times with different sequences to even out possible side effects from a special set of request data.

Before and after each iteration a time stamp is acquired via the *Java* method `System.nanoTime()` and temporarily stored. Afterwards, these values are analyzed and minimum, maximum, average and deviation are computed. When all suites have finished, relative averages for all relevant tests and overall are computed. All results are stored in a CSV file and further analyzed in *LibreOffice Calc*.

3.4 Test setup

All benchmarks were executed in virtual machines once on the developer's desktop PC using Oracle VirtualBox and once on our server using Proxmox/KVM. Since there were no notable difference regarding relative speed of the alternatives we are only providing measurments made on the server. The host machine has two *AMD* 6-core *Opterons* (4234) with 3.1 GHz and 64 Gigabytes of RAM and a RAID 5 disk system with 4×1.5 TB hard disk running behind an Adaptec RAID 5405 controller. The virtual machine is using *Ubuntu Server* 12.04 LTS on an ext4 partition, with access to two CPUs and four cores each as well as 8 GB of RAM. A quick comparison showed that there are no significant differences in terms of performance compared to running the benchmarks natively.

The different suites are as follows:

- *Neo4j* embedded: benchmark with embedded *Neo4j*, native object access
- *Neo4j* REST: benchmark with a dedicated *Neo4j* server using the same methods as native access, but via RESTful Web services
- *Neo4j Cypher* embedded: benchmark with embedded *Neo4j*, *Cypher* queries
- *Neo4j Cypher* REST: benchmark with a *Neo4j* server and *Cypher* queries optimized for remote execution
- *Neo4j Gremlin* REST: benchmark with a *Neo4j* server, *Gremlin* queries for person service and a few other queries
- *MySQL JPA*: *JPA* benchmark for person service only

The different suites execute individual benchmarks, which query certain service implementations using Shindig's interfaces. The calls performed represent typical queries needed for our intranet portal scenario. They focus on the retrieval of person profiles, lists of friends, friend suggestions and groups as well as messages, activity streams and other social network information.

4. COMPARISON OF QUERY LANGUAGES

From a developer's perspective, not only performance, but also initial learning effort, code readability and maintainability are relevant when choosing a query language. In this section we give examples of the resulting code and compare it's readability and efficiency in terms of lines of code.

4.1 Native object access

Having to satisfy *Shindig*'s object interfaces, a full retrieval of an entity requires all subordinate objects to be retrieved. This means many additional nodes besides the entity's own one will be accessed through traversals. For example, to completely retrieve a person object, a total of eight additional traversals may be needed to get all these dependent objects. The same principle applies to additional tables being accessed in the relational database.

Manual native traversals always follow the same pattern. Having started the *Neo4j* instance, key-value indices are queried, mostly for people's IDs.

```
GraphDatabaseService database =
    new GraphDatabaseFactory().
        new EmbeddedDatabase("/path/db/");
Index<Node> peopleNodes =
    database.index().forNodes("people");
IndexHits<Node> matching =
    peopleNodes.get("id-key", "user-id");
```

Afterwards, the resulting nodes can be used to retrieve their attributes or to traverse the graph starting at their position. Using several of these traversal steps in succession, complex requests can be satisfied, for example activities of all friends of a person. As an example, the following code retrieves the names of a person's friends, without the necessary conversion to *Shindig* objects[12]:

[12] `RelTypes` is an enumeration of available relation types, freely definable by the programmer

```
Node person = matching.getSingle();
Iterable<Relationship> relations =
    startNode.getRelationships(
        Direction.OUTGOING, RelTypes.FRIEND);
for(Relationship rel : relations)
{
    Node friend = rel.getEndNode();
    String name = friend.getProperty("name");
}
```

4.2 Cypher

Ideally, *Cypher* queries are constant strings, so they can be cached by the database as compiled queries. When using the embedded API or a wrapper for the REST API, parameters can be passed using a *Java* `Map`. Parameters can be numbers, strings or arrays of these types. Depending on the query, results can be nodes with all attributes, individual attributes or aggregated data. Like SQL, *Cypher* is not only a query language but does also allow data manipulation like updates and deletes. As an example, the query to retrieve friend suggestions for a person can be stated as follows:

```
START person=node:people(id = {id})
MATCH person-[:FRIEND_OF]->friend-[:FRIEND_OF]
    ->friend_of_friend
WHERE not (friend_of_friend<-[:FRIEND_OF]-person)
RETURN friend_of_friend, COUNT(*)
ORDER BY COUNT(*) DESC
```

This query starts by searching the index called "people" for a node whose attribute "id" is set to the contents of the parameter "{id}". Next, friends of friends are determined using pattern matching. The "`WHERE`" clause ensures that the original person is not already friends with the friend of a friend. As a result, the query returns all nodes of unknown friends of friends and how often they were encountered during the traversal, which is also the sort criterion. Attributes could e.g. be accessed by "`friend_of_friend.id`". Limited queries would be possible using "`SKIP`" and "`LIMIT`" clauses.

4.3 Gremlin

Gremlin is a low–level graph traversal language that uses a very compact syntax and therefore is not easy to read. The code shown below represents the same friend suggestion query as in the *Cypher* example above, but without counting and sorting. The query uses an intermediate array x to store IDs of direct friends of the person, in order to exclude them from the friend suggestion list gathered in the second part of the query. The dedup function is the *Gremlin* equivalent to the distinct keyword in SQL. ID and name of the suggested friend are then returned inside the table.

```
t = new Table();
x = [];"
g.idx('persons')[[id:id_param]].
    out('FRIEND_OF').fill(x);"
g.idx('persons')[[id:id_param]].out('FRIEND_OF').
    out('FRIEND_OF').dedup().except(x).id.as('ID').
      back(1).displayName.as('name').
         table(t,['ID','name']){it}{it}.iterate();
t
```

4.4 SQL

Two aspects of the JPA/SQL implementation are relevant for our comparison. Firstly, the code that a developer has to write in order to retrieve the desired data. This is very compact, since JPA and Hibernate do all the hard work here and the developer only uses a single method call and some annotations. The second aspect is the generated SQL code. We analyzed that using a network sniffer. Unfortunately, the SQL code is too large to display here, since every single property is listed in the query in the form `SELECT persondb0.oid as oid7`. In a simplified form a friends of a person query looks like this.

```
SELECT persondb0.ID, persondb0.display_name
FROM person persondb0
WHERE persondb0.oid IN (
   SELECT frienddb2.friend_id
   from person persondb1, friend frienddb2
   WHERE persondb1.oid=frienddb2.person_id AND
       (persondb1.person_id IN (?)))
```

Data from the related tables like address, email, name, organization and so on are retrieved in separate queries, which is why the query above doesn't look very complicated. It might also be partly responsible for the measured performance, since sending multiple queries over the network is often slower than issuing one complex query (see section 5).

4.5 Discussion

As can be seen from the code examples above, the native code needed to retrieve the desired data is significantly larger than both *Cypher* and *Gremlin* queries. The *Java* code for using *JPA* is the most compact one. However, there is a second aspect that has to be taken into account. The data transfer objects returned by *Neo4j* are not yet in the desired Shindig object model and have to be converted. This is necessary in all cases but JPA, but differs regarding complexity depending on the structure of the result data. This is discussed in more detail in section 5 below.

However, from a readability and maintainability perspective, *Cypher* seems well suited since its syntax is quite easy to understand for developers familiar with SQL. For data with inherent graph-like structures, like the ones in our scenario, the code is additionally more compact and easier to read than SQL code. It therefore represents our favorite approach to access *Neo4j*, although the necessary conversion makes it much less elegant than the object-relational mapping of *JPA*.

5. BENCHMARK RESULTS

The six alternatives shown in table 3 and the following figures cannot be compared directly one on one. Queries using an embedded instance are naturally much faster than those accessing the DBMS over the network, since the overhead of the network stack is significant. Therefore, we suggest to first compare *JPA* to *Gremlin* and RESTful *Cypher*. Acknowledging that both *Neo4j* query languages outperform *JPA* for friend queries, we moved on to further compare *Gremlin* and *Cypher*. In order to learn more about the influence of the query language compared to the network stack, we finally compare RESTful *Cypher* to it's embedded version and to native object access.

5.1 Discussion

Comparing an embedded *Neo4j's* native object access with remote access over REST, the RESTful interface is significantly slower, due to it's high network overhead resulting from a high number of individual queries compared to *Cypher* or *Gremlin* so that even *JPA* is faster in most cases.

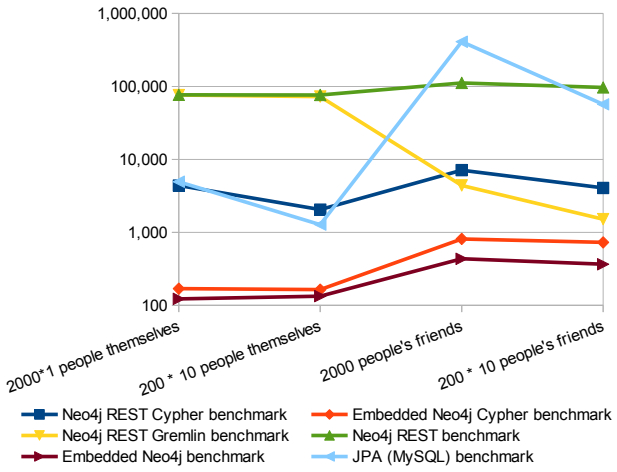

Figure 1: Results for 2,000 people in ms

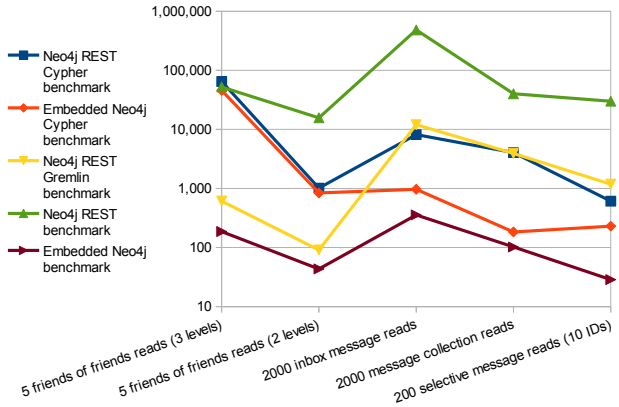

Figure 2: Results for *Gremlin* vs. *Cypher* in ms

Some tests we had prepared could not be run using the *Neo4j* REST API wrapper as we encountered problems with number conversion. Those were consequently excluded from the results presented here. In some cases, *Java* `Longs` were interpreted as `Integers` and vice versa. Since objects from the results are cast to a certain type by some routines, there were class cast exceptions at several points which could not be fixed easily.

After the first tests, we skipped writing data loaders for the rest of the JPA benchmarks and concentrated on the comparison between *Gremlin* and *Cypher* instead. Figure 2 shows the results of those additional benchmarks.

Finally, after we decided that *Cypher* is a better compromise for our work, we tried to further nail down in which areas *Cypher* significantly looses performance against native object access and in which cases performance differences are small. Those final results are shown in figure 3.

Table 3: Person benchmark for 10,000 people, average time needed, average over 10 runs

Person benchmark	Cypher REST	Cypher	Gremlin	Native REST	Native	JPA MySQL
2000 people	4,600 ms	180 ms	76,057 ms	77,041 ms	140 ms	30,010 ms
200 * 10 people	2,038 ms	173 ms	73,025 ms	76,786 ms	152 ms	6,474 ms
2000 people's friends	7,347 ms	931 ms	4,538 ms	121,471 ms	541 ms	3,558,882 ms
200 * 10 people's friends	4,381 ms	850 ms	1,694 ms	120,815 ms	496 ms	384,025 ms

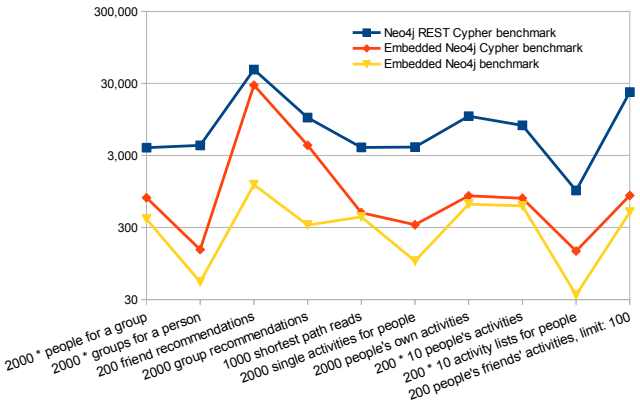

Figure 3: *Cypher* vs. native object access in ms

5.1.1 JPA vs. RESTful Cypher and Gremlin

Looking at figure 1 it can be seen that the *JPA* back-end delivers results more or less equally fast like RESTful *Cypher* for the person profile. Despite that, friends queries are more than one order of magnitude slower.

If we brake those numbers down to the time needed for a single request, the embedded *Neo4j* instance can deliver a person profile in about 0.1 milliseconds, a dedicated *Neo4j* server with *Cypher* in 4.4 milliseconds and with *JPA* it takes around 5 milliseconds. That means that all are very low and usable, especially when the software using the back-end is a web-application. On the other hand, the retrieval of someone's friends already takes about 200 milliseconds when using a local *MySQL* instance.

The advantages of *Neo4j* over *MySQL* become more prevelant, when the database gets larger. In our person benchmarks, *Neo4j* has shown a nearly constant performance when scaling from 2,000 to 10,000 people (a drop of only 3–8 %) whereas *MySQL* performance drops by factor 5 (linear) for people queries and factor 7–9 for people's friends queries.

We expected differences to become even more obvious for FOAF queries or group recommendations and therefore decided to chose *Neo4j* for our project. The huge difference between object access in the embedded instance and RESTful object access is due to the network overhead and the large amount of single queries that are issued for this query method. Every traversal leads to an own network operation in the RESTful case, so that the result is slower than even JPA in most cases.

5.1.2 Gremlin vs. Cypher

The comparison between *Gremlin* and *Cypher* in figure 1 leads to a false impression. *Gremlin* seems to be a bit faster than *Cypher* for friend queries but far slower in the people themselves queries, which is a bit suprising. The explanation for that fact is, that the performance loss is due to the conversion infrastructure, which is not optimized for *Gremlin* here so the performance is on par with the RESTful object access. Several additional network operations in order to retrieve all the person profile data are the cause. Therefore, we performed the additional tests shown in figure 2 to get a fair comparison of *Gremlin* and *Cypher*. Those results show that *Gremlin* is on par with *Cypher* in the message queries and outperforms *Cypher* in the FOAF queries. Traversal-only queries, even with multiple levels of depth, are relatively easy to write in *Gremlin*, so that the query itself seems as efficient native object access. The measured difference seems to be solely network overhead. This advantage over *Cypher* makes *Gremlin* more than an order of magnitude faster for two levels of traversal and 2,000 people and even increases with 10,000 people and a third level of traversal to two orders of magnitude. On the other hand, more complicated queries, collecting many properties, nodes and relationships at several points can be very hard to write and do not perform better than *Cypher* in our tests. *Neo4j*'s *Gremlin* plug-in supports *Groovy* scripting, which could improve on some of *Gremlin*'s weaknesses, but this was not tested for our benchmarks, since it poses and additional hurdle for learning the query language and maintaining the code.

5.1.3 Cypher vs. native object access

The last interesting comparison is between RESTful *Cypher* and *Cypher* embedded, as well as between embedded *Cypher* and native object access. We analyzed this in more depth in figure 3. Having a look at all the figures, we see that performance of *Cypher* embedded compared to native object access is between 10% and 200% worse. That seems a lot at first glance, but seems acceptable compared to the gain in readability and maintainability (see section 4.2). Analyzing these results we see that *Cypher* queries are especially fast for operations that can be formulated as one query but need several steps when using native access. An example are shortest path detections between people.

On the other hand, there are several queries, where *Cypher* performance drops to being 10-25 times slower than native object access, which doesn't seem acceptable any more. This is the case for FOAF queries and for friend and group recommendations which all require quite some graph traversal operations over multiple levels. Those show clearly limitations in either the query language that doesn't allow for early termination of those queries or a lack of optimization in the execution plan on the server. [17] says that there are five elements of a query that influence overall performance:

1. overhead resulting from the optimizer choosing an inferior execution plan

2. overhead of communicating with the DBMS

3. overhead inherent in coding in a high-level language

4. overhead for services such as concurrency control, crash recovery and data integrity

5. truly useful work, which must be performed anyway

We conclude that more complex queries which require a larger amount of pattern matching are relatively slow in *Gremlin* since there is no support for terminating a traversal early when a certain precondition is met. Our native friends of friends traversal logic collects information about which person has already been passed at which depth and thus avoids unnecessarily traversing parts of the graph several times. This is one of the examples that Stonebraker calls an inferior execution plan compared to the human programmer.

Looking at RESTful *Cypher* compared to embedded *Cypher*, we see that in most cases the difference is oscillating around one order of magnitude. This should be due to network overhead. For those queries however, where embedded *Cypher* is far outperformed by native object access, the difference between RESTful and embedded *Cypher* diminishes, which supports our interpretation of results.

However, *Cypher*'s default behavior might still be needed for other queries, like friend suggestions, so it can still be very useful. Additionally, *Neo4j*'s support for *Cypher* is still relatively young and up to version 1.8 the main development focus was on additional language constructs like aggregation. The upcoming version 1.9, which can already be tested in milestone 1 release, is the first one to focus on performance optimization of *Cypher* and will offer enhanced query speed.

5.2 Analysis

The overall low performance when retrieving friends of friends is probably caused by the highly connected set of data used. Since there are several people with far more than 100 friends, traversal with even few levels of depth requires large portions of the graph being analyzed. Yet this seems plausible for our intranet scenario, even if in real world there may be a smaller number of people in total.

In some cases we found that performance of a *Cypher* query strongly depends on the way it is formulated, although several alternatives should result in the same traversal. While knowledge about e.g., SQL is widespread, an average programmer can hardly be expected to differentiate queries with good from those with bad performance in *Cypher*. [17] states that although for SQL, early optimizers were primitive, they quickly became as good as all but the best human programmers and moreover, most clients cannot attract and retain this level of talent that outperforms the optimizer. The same might be true for *Cypher* in the near future.

5.2.1 Algorithms used

The high performance measured for shortest path searches could only be achieved by using specialized algorithms supplied by *Neo4j*'s `GraphAlgoFactory`. The concrete algorithm used to find shortest paths is not specified, but our own breadth and depth first search algorithms were more than 100-times slower. Consequently, it would be interesting to find other problems that could be solved by this high performance algorithm, e.g., *A** and *Dijkstra*. A comparison between traversals using *Neo4j*'s own traversal framework and manual traversals showed only minor differences in performance with no clear advantage for any approach. Though, using the framework can result in fewer lines of code to retrieve data which may be easier to maintain. Additionally, the application might benefit from future improvements in *Neo4j*'s own traversal framework in a database upgrade.

5.2.2 Embedded instance vs. standalone server

Despite the high performance when using an embedded *Neo4j* instance, deployment in a cloud scenario may create problems concerning scalability and reliability. An external *Neo4j* server or *Neo4j*'s high availability version would be better suited for such an environment. In that case however, performance will drop because of the additional protocol and network layer. *Neo4j* developers reckon that this might lead to a decrease in performance by about one order of magnitude, which largely is the case in our results. So such a setup would probably only be useful for large amounts of data that don't fit in the server's RAM because delay due to random disk access is expected to be even larger than one order of magnitude.

5.2.3 Scalability

For a long-term deployment it is very important to know how the database's performance will develop with growing amounts of data. A higher number of people should only take slightly longer since traversals in the graph only concern the local neighborhood of nodes, not the complete set of users. Our results confirmed that. Multiplying the number of users by five led to increase in query times of three to eight percent, which is nearly linear effort.

5.2.4 RESTful object access

Further tests showed that simply swapping the embedded implementation with the REST API wrapper causes performance to drop by a factor of about 200–300 for both native access and *Cypher* queries. However, by using optimized, remotely executed *Cypher* queries and an adjusted result conversion, performance could be raised to be only one order of magnitude worse than the embedded instance, as claimed by the *Neo4j* developers. In the embedded *Neo4j* instance, we are using single queries to retrieve data associated with a person, whereas in the remote case we are now using singular, complex queries. In this way, only a single call is needed, reducing network overhead drastically. The downside is that these queries may have to be created dynamically in code and are harder to maintain, analyze and debug.

5.2.5 ID arrays

Moreover, we noticed a rather low performance for *Cypher* queries that contained an array of IDs to look for. The array is passed as separate argument for a predefined *Cypher* query and e.g. evaluated via the clause "`WHERE message.id IN array`". While the low performance even for small lists is surprising, in case IDs are unique entities they could be more easily retrieved using an index, like it is implemented for people. But in that case, after querying the back-end would have to determine whether the entities even belong to the user, which is otherwise determined through the traversal.

5.3 Limitations

Firstly, all retrieval sequences and the underlying data were randomized. Consequently, there will probably be some variations in performance between several similar data sets as well as between individual runs. To ensure that values

generated truely represent the back-ends potential performance, we ran several iterations for each benchmark and compared them to the averages of runs with other data sets. Although they are small, there will still be fluctuations in the benchmark's results when running it several times.

Depending on the underlying graph structure, message and activity performance may degrade more rapidly with rising numbers of entries. For our benchmark, we linked each message and activty directly to a user without any order. Hence, to find certain entities, for example the newest activities, all entities have to be analyzed and sorted. But since this is probably a very common use case, all entities could be arranged into a linked list in chronological order. Using this structure, only the list's head and the requested amount of following nodes would have to be accessed.

For production environments, *Neo4j*'s server settings, *Cypher* queries and the conversion infrastructure associated with them could be further optimized. The server's optimal settings will depend on hardware and operating system used as well as dimensions of the data set. Our *Cypher* queries mostly request complete nodes and relationships, which causes a considerable network traffic overhead, due to inefficiencies in the data structure used by *Neo4j*. For each data object, a number of meta data in form of URLs is delivered back as well, leading to an overhead to payload ratio of 6-9 to one, depending on the query. This can be reduced to 11 to 10 when querying each single property of the node instead. Since this looked promising, we tested it with a single query and got the astonishing result that performance dropped by 20–40%. Therefore, we didn't investigate further. As stated before, due to lacking expert knowledge about *Cypher* and *Neo4j*, there may also be queries that deliver the same results in a shorter time. On the other hand, this also seems fair since SQL queries issued by JPA are also not manually optimized and neither was the *MySQL* installation.

Also, *Cypher* queries used for REST benchmarks and embedded benchmarks were not the same, since this would have required additional conversion routines for the different result formats used by the embedded database and the REST wrapper. The embedded *Cypher* benchmark uses several small queries, the REST *Cypher* benchmark mostly singular big queries to reduce the influence of network latency. Conversion routines take a lot of time to write, since they have to convert individual columns and collections of these big queries into *Apache Shindig*'s object model. As explained before, it strongly differs from the underlying graph model. We reckon that REST queries would not perform better on an embedded instance, since they are a lot more complex themselves and require more complex conversion routines.

Furthermore, performance may differ for the individual access options when using *Neo4j*'s advanced versions in a cluster with load balancing. Further modifications to the back-end might provide further performance improvements in this case, harnessing the capabilities of the load balancer.

6. CONCLUSIONS

We've analyzed performance and programming effort for data back-ends for Apache Shindig. Comparing the existing open source JPA back-end using *MySQL* to our own implementation with *Neo4j* and *Cypher*, we were able to achieve performance improvements of one order of magnitude for queries that span multiple tables. In our social Web portal scenario, several queries fulfill this requirement and for friend or group recommendations we expect differences to be even greater, although *Cypher* didn't perform very well itself in these tests compared to *Gremlin* and native object access. However, we expect this problem to be addressed in *Neo4j* version 1.9 which will be optimized for providing high performance *Cypher* query results. We are already cooperating with the database manufacturer Neo Technologies and provided them with our results and experiences, so that they can use them in their own tests.

Regarding the two query languages, our subjective impression was that *Cypher* feels much more natural for a developer with existing SQL skills than *Gremlin* and performs well in many cases. However, performance benefits of *Gremlin* in FOAF queries cannot be neglected. We hope for significant performance boost in this area in the next release of *Neo4j*.

Compared to native object access, *Cypher* is about two times slower, which we would be willing to accept in order to gain readability and maintainability for the code as well as some efficiency in development time. We also expect to benefit from further improvements in *Neo4j*'s query optimizer and agree with Stonebraker [17] that some overhead is unavoidable when using higher level query languages. The same was true for SQL 25 years ago and nobody today would suggest avoiding SQL and go for native access in order to gain some performance. In contrary, we are often using additional layers of indirection like JPA and Hibernate, so that even the SQL queries are not optimized any longer.

Overall, *Neo4j* can be used as a high performance replacement for relational databases, especially when handling highly interconnected data as in our social Web portal. Using an embedded *Neo4j* instance can yield very good performance when using data sets of limited size. For production environments with higher scalability requirements, *Neo4j*'s advanced versions promise high availability and scalability if needed. We will evaluate this option in our next step and are especially keen on testing the possiblity of including embedded *Neo4j* instances into a cluster. Given the results presented here, it could be a good alternative to give every instance of *Shindig* it's own embedded *Neo4j* instance and cluster those for data replication instead of using the HA proxy between the *Shindig* instances and the cluster of *Neo4j* instances, being accessed using RESTful Web services.

Another interesting approach is *Spring Data Neo4j*, the graph counterpart of object-relational mapping provided by *JPA* and *Hibernate*. It could help reducing the code necessary to convert between *Neo4j's* own data transfer objects and *Shindig's* object model, which was perceived as significant development effort in our benchmark. In some first tests, we had trouble achieving the desired results, but we will keep on investigating this alternative further.

7. REFERENCES

[1] A. M. Alashqur, S. Y. W. Su, and H. Lam. Oql: a query language for manipulating object-oriented databases. In *Proceedings of the 15th international conference on Very large data bases*, VLDB '89, pages 433–442, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[2] R. Angles. A comparison of current graph database models. In *Proceedings of the 28th IEEE International Conference on Data Engineering Workshops (ICDEW)*, pages 171–177, April 2012.

[3] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[4] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[5] E. F. Codd. Relational database: a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, Feb. 1982.

[6] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In H. Shen, J. Pei, M. Özsu, L. Zou, J. Lu, T.-W. Ling, G. Yu, Y. Zhuang, and J. Shao, editors, *Web-Age Information Management*, volume 6185 of *Lecture Notes in Computer Science*, pages 37–48. Springer Berlin Heidelberg, 2010.

[7] S. D. Giugno, R. Graphgrep: A fast and universal method for querying graphs. In *Proceedings of the 16th International Conference on Pattern Recognition*, volume 2, pages 112– 115, 2002.

[8] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of rdf query languages. In S. McIlraith, D. Plexousakis, and F. Harmelen, editors, *The Semantic Web – ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 502–517. Springer Berlin Heidelberg, 2004.

[9] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM.

[10] K. Houkjaer, K. Torp, and R. Wind. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1243–1246. VLDB Endowment, 2006.

[11] S. B. Jim Melton. *XQuery, XPath, and SQL/XML in Context*. Morgan Kaufmann, 2006.

[12] L. H. Marek Ciglan, Alex Averbuch. Benchmarking traversal operations over graph databases. In *3rd International Workshop on Graph Data Management: Techniques and Applications (GDM 2012)*, pages 186–189, Washington DC, USA, April 2012.

[13] neo technology, Inc. *The Neo4j Manual v1.8*, September 2012. http://docs.neo4j.org/chunked/stable/index.html.

[14] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *The Semantic Web - ISWC 2006*, volume 4273 of *Lecture Notes in Computer Science*, pages 30–43. Springer Berlin Heidelberg, 2006.

[15] L. Sackett. *MDX Reporting and Analytics with SAP NetWeaver BW*. SAP PRESS, 2009.

[16] C. T. Shalini Batra. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(2), May 2012.

[17] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Communications of the ACM*, 54(6):72–80, June 2011.

[18] R. P. Trueblood. *Data Mining and Statistical Analysis Using SQL*. Apress, 2008.

[19] C. Tudorica, B.G.; Bucur. A comparison between several nosql databases with comments and notes. In *10th Roedunet International Conference (RoEduNet 2011)*, Iasi, Romania, June 2011. IEEE.

[20] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 42:1–42:6, New York, NY, USA, 2010. ACM.