An Exploration into the Performance of NoSQL Databases, Neo4j and MongoDB, alongside Oracle Database

Scott Maner, Muhammad Rana

Abstract—**We explore situations in which NoSQL data models such as the Document-Oriented model and the Property Graph model may demonstrate how they are better suited at times when dealing with data with high interconnectedness, such as a social network, and instances in which they are not. Also we will attempt to highlight how a traditional relational database model may prove unfeasible in the same context. We go about attempting this objective by endeavoring to provide a concrete idea of what types of queries might prove highly efficient within certain database management systems as well as possibly highlight at what data-size the choice of database model becomes crucial to successfully querying and modeling a dataset.**

I. Introduction

Relational database models (RDBM) have garnered the majority of the market for well over 40 years. With flexibility and the need to handle Big Data becoming a commonplace requirement as storage has become cheaper and data continues to grow more complex, demand has continued to rise for a less rigid form of data modeling. These requirements often struggle within the confinements associated with RDBMs [1]. Confinements brought about by the data are stored within the table, where data is contained in predefined categories or columns. For the user to successfully access this data, rows must be able to be uniquely identified by some data contained in a column, which is referred to as a primary key. This form of representation of data works quite well with structured data, but once we break from this relational databases can experience a dramatic increase in complexity required to query and even maintain data.

A. Document-Oriented Database

MongoDB, a document-oriented database, operates based on creating JavaScript Object Notation (JSON) documents and performing the fundamental operations of storing and retrieving based on key-value pairs. With this type of structuring, many of the restrictions that accompany RDBMs no longer apply[2]. It is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use. It specifically can handle the task of accessing Big Data. At a certain data size (over 50GB) MongoDB surpasses relational databases by a considerable margin, as RDBMs suffer with reading and writing operations[3].

B. Graph Database

For data that is highly interconnected it is often painful attempting to fit said data into a strict data model like a RDBM, which require the data to be modeled in a way that might not best represent the data. It was because of this contortion that Graph Databases were created. Data, such as social networks, conform naturally to a graph model, which is a data structure made of vertices's connected to one another by edges. These edges are made between two vertices's whom share some common relation, for instance, a person on Facebook may be linked to another through a shared friend or some mutual interest. This is the concept that Neo4j, the management system we have employed to contain our datasets, was founded on. With Neo4j we gain the added benefit of using their query language, Cypher, which offers effective and often time concise querying of our Neo4j database.

II. Related Works

As NoSQL has increased its presence in the overall marketplace of Database models, so have the studies into their overall performance [3]. There are primarily 3 types of mainstream NoSQL databases. 1. Key-Value Database 2. Column Oriented Database 3. Document-Oriented Database. A lot of research has gone into studying their popularities and functionalities[3][4].

The field has also experienced quite a notable number of researches comparing one of the most popular document oriented database MongoDB against traditional relational databases like MySQL and Oracle on the ground of CRUD *(Create, replace, Update, Delete)* operations. Aghi et al. have done the CRUD comparison between MySQL and MongoDB with three different sizes of datasets including 10, 400 & 2000 rows[2].

Cornelia et al. have also done the same types of comparison between these two databases with a relatively larger dataset of 5000 rows[5]. However, this is still not large enough for understanding the difference between their performances on the primitive operations at million scale.

Boicea et al. have conducted their CRUD comparisons between Oracle and MongoDB with a larger dataset than the above two cited papers of one million records[6]. Their paper gave a primitive idea of how these two databases can interact with a much larger dataset.

Li et al. have published a comparison paper comparing a multitude of NoSQL databases[7]. They conducted experiments to compare the performance between: MongoDB, RavenDB, CouchDB, Cassandra, Hypertable, Couchbase and MS SQL Express. However, the depth of their comparisons is limited to basic CRUD operations.

We have also seen a lot of research on explaining graph database architectures, their types, querying the graph databases on different ways, and most importantly comparing graph databases against relational databases.

Miller has made a comparison between relational database systems (Oracle, MySQL) and graph databases (Neo4J) on the aspects of data structures, data model features, and query facilities[8]. This paper also depicted the inherent and contemporary limitations of current offerings.

Holzschuher et al. described their experiences using Cypher (Neo4j's query language) and Gremlin (the general purpose graph query/traversal language) for the graph database Neo4j and Java as alternatives for querying data with MySQL[9]. They have considered performance and usability from a developer's perspective and their results showed that Cypher is a good query language in terms of code readability and has a moderate overhead for most queries (20–200%).

On another paper Marek et al. wrote their research results after testing five graph databases, namely: Neo4J, DEX, OrientDB, Native RDF repository (NativeSail) and a research prototype SGDB[10]. They benchmarked their tests on mainly Graph traversal.

Besides these above papers the Wisal et al. paper has also shown us the performance comparison between relational and NoSQL graph databases on the ground of basic queries[11]. They have done their researches both on local and remote system and with up to 860,000 of records.

III. Implementation

In this section we will largely focus on details of our query selection, how each individual query was implemented within each environment, and why they were chosen. We will also delve into our database environment and any performance tuning that was done and briefly elaborate on the makeup of our two data sets provided by SNAP[12].

A. Optimization

As a precursor to our implementation the following steps were taken to set up our respective environment. For Neo4j, as recommended, we increased the page cache size (1.2GB) and heap size (4GB). Increasing our heap size was mandatory as certain operations could induce failures. Creation of indexes within Neo4j were limited to our primary key, user_id. Exceptions for this exist only in a specific query to be discussed in our Text Search subsection. For making MongoDB's text search faster we have used custom indexes. One more exception on MongoDB was allowing disk usage on the sort operation that was needed for doing our top x percent query. Oracle at this time is largely unchanged.

B. SNAP Dataset

The following social network datasets were chosen from SNAP: Facebook, Pokec, and a reduced version of the original Pokec data we will henceforth refer to as Pokeclite. These datasets are rather primitive, containing a main friend table and a corresponding relationship table containing a friendslist. For reasons of simplicity, we have condensed our two Pokec datasets down to five columns of data and added to Facebook's dataset to match, both contain: user_id, completion_percentage, public, age, region and gender. Table I provides details regarding the size of our datasets.

Contents	Facebook	Pokeclite	Pokec
Friends	4,049	400,000	1,632,803
Friendships	88,234	9,905,062	30,622,564

Table I: Size Of Data

C. Query Selection

The key objective behind our query selection was to formulate queries we hypothesized would appear native to a graph style database, while not appearing unfriendly to a document style database. We also desired queries that would appear better suited for a traditional RDBM in an attempt to create a fair comparison.

Our selection of queries which we believed Neo4j would excel are based on exploiting the paradigm of a graph database, which entails nodes connected by some shared relationship. In our case, these are profiles sharing a friendship with each other. The selected queries include: N-Hops, which involves finding friends a certain n distance from the search node; and Mutual Friends which searches for friends in common from two given root nodes. These queries would consist of costly join operations and nested sub-queries in a traditional relational database. Next, we have our queries which we believed are more well suited for Oracle and MongoDB such as a percentile query based on some calculation of an attribute in which a selected top x percent would be returned. Another operation we have chosen is a full Text Search, which requires reading through all the information contained in our friend node to find a desired text.

There was also a desire to explore into the performance associated with aggregation functions, group by, and order by operations. The performance of this type of query, which we will denote as group by after aggregation (GBAA), was rather uncertain towards which model would perform well. Lastly, we continued the idea of exploring aggregation functions with our most diverse friend query, the goal of which is to find the profile whom has friends from the most distinct regions.

1) N-Hops: As mentioned, Neo4j's Graph Database model consists of a series of nodes representing an entity, some object which contains some given data, and edges, which represent some relationship between entities. The notion of creating a query that traverses these relationships to some N-th depth should appear native to this type of model. The overall length of the query, shown in figure II, in relation to MongoDB and Oracle should elicit this idea. Neo4j's variable length relationship feature allows for the use of the simple *[:FRIENDSHIP*..n]* notation, where *n* represents the distance away from the search node. All that is required is to substitute the desired number of hops with *n*.

Oracle, on the other-hand, has to complete a series of joins and nested loops which can be quite expensive to perform. By extending the number of hops to do, you have to extend the number of joins needed to complete the query. MongoDB follows the same line as Oracle, requiring a series of self joins. For simplicity of the queries we haven't considered more than 2-depth queries in MongoDB.

Environment	Query

Oracle	select count(*)
	from FRIEND
	where user_id in (select Distinct friend_id
	from friendlist
	where friendlist.user_id in(select friend_id
	from friendlist
	where friendlist.user_id = *x*));

MongoDB	db.FRIEND.aggregate([{ $lookup: { from: "FRIEND", localField: "friend_id", foreignField: "user_id", as: "friends_of_friends" } }, { $match: "user_id" : *x* }])
Neo4j	Match p = (f:FRIEND user_id: "*x*") [:FRIENDSHIP*..2]->(:Friends) return count(p)

Table II: 2-Hops

2) Mutual Friends (MF): When searching for mutual friends we can use a particular pattern in Neo4j called *collaborative filtering*, as seen in Table III, to create a concise query which returns the desired result. Oracle's query has grown a decent margin larger by comparison though it can still be thought of as a concise query in which we use the *in* operator to return mutual friends. Oracle must again leverage those expensive joins and nested query operations to return the desired result. On MongoDB we first fetched the friends data of the first person and stored those in a variable. Later we used the stored data to search for possible matches denoting a mutual friend.

Environment	Query
Oracle	select count(*) from (select friend_id from friendlist where user_id = *x*) temp where temp.friend_id in (select friendlist.friend_id from friendlist where friendlist.user_id = *y*);
MongoDB	var cursor1 = db.FRIEND.find(user_id : *x*).toArray().map(function(u) { return u.friend_id ; }) db.FRIEND.aggregate([{ $match: { $and: [{ user_id: {x} }, { friend_id : {$in: cursor1}}] } }])
Neo4j	Match(f:Frienduser_id:"*x*")- [:FRIENDSHIP]->()<-[:FRIENDSHIP]- (b:Frienduser_id:"*y*") return count(*)

Table III: Mutual Friends

3) Text Search(TS): With Text Search we should gain insight on the run-times of each respective database by creating queries that iterate through all attributes contained in our main friend table, which is what our Text Search query demands. The actual implementation of our queries were rather straight forward in the section. Neo4j allows the use of a *contains* operator which we can leverage against all of our data-fields. Oracle is much the same, as values in the improper format must first be cast but can then simply be compared using the *LIKE* operator. MongoDB query for the text search operation was almost similar to Oracle and Neo4j, using a basic *find* operation with the help of the *or* operator for looking through all the columns.

Out of curiosity we also included an extra test, based on the idea that a database administrator may not want incur the cost related to creating duplicate references to all columns in a particular table, as this can slow certain operations in a given environment and certainly increase database size. As such, we will be creating a search in which we have indexes created only for *user_id*, and another test in which we have created indexes on all six columns contained in our Friend's tables.

4) Top x Percent: Neo4j has access to all the common aggregation functions most databases give

Environment	Query

Oracle	SELECT count(*)
	FROM FRIEND
	WHERE region || to_char(completion_percentage) || to_char(user_id) || to_char(public_account) || to_char(age) || to_char(gender) LIKE '%*x*%';

MongoDB	db.pokecProfiles.find($or: [region: /mesto/ , gender : /mesto/ , age : /mesto/ , completion_percentage : /mesto/ , public : /mesto/ , user_id : /mesto/]).count()
Neo4j	match (f:Friends) where f.AGE contains "*x*" or toString(f.completion_percentage) contains "*x*" or f.gender contains "*x*" or f.public contains "*x*" or f.region contains "*x*" or f.user_id contains "*x*" return count(f)

Table IV: Text Search

access to, but does lack some unique operations that other more seasoned query languages, such as *NTILE* or *TOP x PERCENT* in Microsoft SQL Server, provided. Instead, we leveraged Neo4j's *percentileDisc* aggregation function, which would return the *x*-th percentile desired by the user. For Oracle we instead employed the *NTILE* operator mentioned above. With this operation we can assign identifiers, that represent a specific percentile, to our data based on our newly created *NTILE* column. All that is left to is select the desired percentile.

For supporting the same functionality in MongoDB we needed to do a $sort operation. MongoDB failed to perform the sort operation without using the disk on the Pokec dataset. Permission of disk usage by the *"allowDiskUse: true"* command had to be given to facilitate this.

Environment	Query
Oracle	with x_percent as (select user_id, completion_percentage, NTILE(*x*) over (order by completion_percentage ASC) as percentile from friend) select count(*) from x_percent where percentile = *x*;
MongoDB	db.pokecProfiles.aggregate([{$sort: {completion_percentage:-1}}, {$limit:.1*db.pokecProfiles.count()}], {allowDiskUse: true})

Neo4j	match(t:Friends)
	with percentileDisc(t.completion_percentage, *x*) as res
	match(f:Friends)
	where res < f.completion_percentage
	return count(f);

Table V: Top *x* Percent

5) Group By After Aggregation: In this specific section we tried to explore how our datasets within their respective database models would perform on a combination of group by, order by, and aggregation operations. Specifically we formulated queries which will return all distinct regions sorted in descending order according to the average profile completion percentage and the total number of friends within a region. MongoDB, Oracle and Neo4j have their dedicated *avg* function for computing the averages. Oracle and Neo4j performed the ordering using their *order by*, whereas MongoDB did this using it's native *$sort* method. While Neo4j does not have the explicit *group by* keyword, it can be simulated by requesting specific return fields.

6) Most Diverse Friend: This section informed us in regards to how our databases could perform when computing complex hybrid grouping, sorting, and aggregation operations. For these examples we have specified our query to return the most diverse friend i.e. we find the profile which has the highest

Environment	Query
Oracle	select region,avg(completion_percentage) as avgPercent, count(user_id) as userCount from friend group by region order by userCount desc,avgPercent desc;
MongoDB	db.pokecProfiles.aggregate(["$group" : _id:"$region", count:$sum:1, avgCompletionPercentage: $avg: "$completion_percentage" , $sort:"count":1, avgCompletionPercentage:1])
Neo4j	match (f:Friends) with f return distinct f.region, count(f), avg(f.completion_percentage) order by avg(f.completion_percentage) desc, count(f) desc

Table VI: Group By After Aggregation

Environment	Query
Oracle	select t.user_id,(select count (distinct region) from friend where user_id in (select friend_id from friend ,friendlist where friend.user_id = t.user_id and friend.user_id = friendlist.user_id)) as distinctRegion from friend t order by distinctRegion desc FETCH NEXT 1 ROWS ONLY;

MongoDB	db.miniPokecEdges.aggregate([{
	$lookup: { from: "miniPokecProfiles",
	localField: "friend_id",
	foreignField: "user_id",
	as: "friends_info"
	} }, { "$group" : {_id:"$user_id", uniqueValues: {$addToSet:
	"$friends_info.region"} , count:{$sum:1}}
	}, { $project: {
	"size": {
	$size: "$uniqueValues"
	} } }, {$sort:{size:-1}}
	])

Neo4j	match (f:Friends)-[r:FRIENDSHIP]->(c:Friends) with count(distinct c.region) as regns, f as friend order by regns desc return friend.user_id,regns limit 1

Table VII: Most Diverse

number of friends whom live in distinct regions. Neo4j's *with* keyword allowed us manipulate data before we pass it to the rest of the query. In our case, we computed the number of friends per distinct region and rename this field as *regn*. From this point it becomes a trivial *order by* operation on our new *regn* field, afterwards we return our first entry.

The formulated query to compute our desired results in Oracle can be thought of as quite an involved query, featuring two sub-queries which must be computed for every *user_id*. This means all 1.6 million profiles we have in our pokec data must perform this operation. Once we perform this costly operation, we continue with an order by clause before we finally fetch our first row.

MongoDB occupies a large space for this query as is was required to go through a series of *$lookup, $group, $addToSet, $project and $sort* operations inside the aggregate method. We have used *$addToSet* in MongoDB for finding the distinct region which is equivalent to the *distinct* operator in sql.

IV. Results

All unique implementations were conducted ten times once they had been cached on the local system. We performed a preliminary test on the insertion times for all of our data into their respective environments, shown in table VIII. Though this was not a chief concern for us in our paper, the results appear very impressive for Neo4j and MongoDB, as they performed the operations even on the largest dataset in under five minutes. This is especially impressive when comparing to Oracle's insertion times. Some values were not recording due to limitations or excess time required, these shall be denoted by *x*.

DB	**Facebook**	**Pokeclite**	**Pokec**
Neo4j	3s	16s	<3min
Oracle	<1min	<14min	>40min
Mongo	<1s	<2min	<5min

Table VIII: Insertion Times

We suspected, by and large, that the results pertaining to our Facebook dataset regardless of environment would perform similarly, due to the size of our dataset. Results provided in Table IX show this to largely to be true, though we found it curious that Oracle performs slightly better with 3-Hops with Facebook's dataset, the operation which we hypothesized would be dominated by Neo4j. Where we begin to see large disparities in our results is, of course, our Pokec datasets. We can immediately see that our Neo4j query increased trivially, from single digit milliseconds with 2-Hop to double digit milliseconds with 3-Hop, when compared with the increase in dataset size. While we only compile data for 2-Hops for MongoDB, we can see that it performed quite well even on our largest dataset.

Oracle's performance decreased exponentially from a few milliseconds on our Facebook dataset to several seconds on our largest Pokec dataset. This high execution cost continued with our Mutual Friend query, as Oracle again had to conduct joins with our massive friend-list table. Neo4j, leveraging how it models data to perform this operation quite handily on both datasets, boasts 5ms average on the largest Pokec dataset. MongoDB performed this operation equally as well shadowing Neo4j's results.

X Percent is our final query, represented in our Tables IX-XI. This operation was the first that did not require costly joins on Oracles part. We previously hypothesized this section would lean towards Oracle and MongoDB as it appeared to be an operation not well suited for Neo4j. Additionally, as Oracle is by far the more seasoned database environment, one could plausibly hypothesize it would perform some optimization that would outperform the other two. The results instead showed that Oracle largely performed similar to our newer databases, coming in last on our small dataset and in the middle the remainder.

DB	**2-Hops**	**3-Hops**	**MF**	***x* *Percent***
Neo4j	1.6ms	4.5ms	2.9 ms	7.8ms
Oracle	3.6ms	4.3ms	2ms	12.1ms
Mongo	12.67ms	x	2.56ms	6.3ms

Table IX: Facebook Results Part 1

DB	**2-Hops**	**3-Hops**	**MF**	***x* *Percent***
Neo4j	2.63ms	35.6ms	4ms	661ms
Oracle	1s	1.89s	395ms	379.4ms
Mongo	36.4ms	x	4.86ms	445.7ms

Table X: Pokeclite Results Part 1

DB	**2-Hops**	**3-Hops**	**MF**	***x* *Percent***
Neo4j	5ms	51ms	5ms	2.3s
Oracle	9.6s	17.3s	8.26s	2.28s
Mongo	37.4ms	x	4.67ms	1.93s

Table XI: Pokec Results Part 1

DB	**Text Search (TS)**	**TS (Indexed)**	**GBAA**	**Most Diverse**
Neo4j	7.5ms	7.75ms	7.2ms	79ms
Oracle	5.9ms	5.8ms	10.9ms	11.9s
Mongo	6.67ms	2ms	24.11ms	6.5s

Table XII: Facebook Results Part 2

Data collected while running our Text Search with indexes limited to only the *user_id* field

DB	Text Search (TS)	TS (Indexed)	GBAA	Most Diverse
Neo4j	1.26s	1.1s	645ms	18.4s
Oracle	474ms	499ms	198ms	x
Mongo	764ms	53ms	520ms	730s

Table XIII: Pokeclite Results Part 2

showed that it was clear that Neo4j under-performs in this field, performing the operation at nearly double the time of MongoDB. Once sampling commenced on our indexed Text Search queries, results showed MongoDB well outperforming our other two once indexes were added. Specifically, when tests were performed on our Pokec datasets MongoDB was shown to be several magnitudes faster, performing the operation well below half a second, while Neo4j consistently performed worse than the others.

Results for GBAA on our Facebook data, which appear to be largely misleading, depict that on a small scale MongoDB performs at almost double the time. Once we begin to look at our larger datasets we see Neo4j actually performing worse than the others. Oracle, as we hypothesized earlier, performs the best for this type of query.

The Most Diverse proved to be quite a computational intensive query, though possibly not fully optimized in MongoDB and certainly not in respect to Oracle. As a result, only Neo4j was able to successfully complete this task of returning the most diverse friend on all datasets. MongoDB was able to successfully return results up to our Pokeclite dataset, but with astronomical time required. Lastly, Oracle was only able to return results when querying Facebook's dataset.

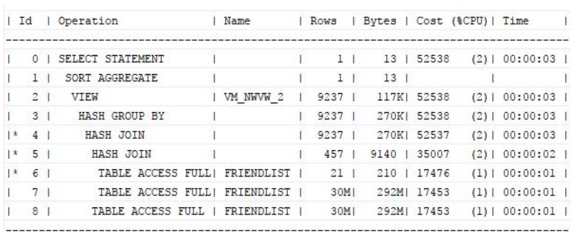
DB	Text Search (TS)	TS (Indexed)	GBAA	Most Diverse
Neo4j	5.39s	4.7s	2.39s	43.5s
Oracle	2.04s	1.93s	1.4s	x
Mongo	3.1s	206ms	2.03s	x

Table XIV: Pokec Results Part 2

V. Conclusion

As predicted, Neo4j outperformed both of the other databases on the queries where there were more relationships between nodes, such as N-hops and Mutual Friends, no matter the size of the dataset. Oracle, on the other hand showed how poorly it can perform on a large highly connected dataset like Pokec. This exponential increase is directly correlated to the repeated join and order by operations that are required to complete our 3-Hop query. As seen in Figure 1, we can directly link the cost of our repeated join operations to our decrease in performance. We now do feel that when using a small dataset, like our Facebook dataset, it is largely not relevant which database model is chosen. We highlighted this in our results section when Oracle performed all queries on our small dataset quite adequately, often only being slightly slower in almost every case except 3-Hops where it performed marginally better than Neo4j. All of this despite many of our queries being costly in Oracle.Though, once the dataset became larger, it became the norm for there to be a high deviation between queries with identical purposes within the different database models.

Figure 1: 3-Hop Pokec Execution Plan

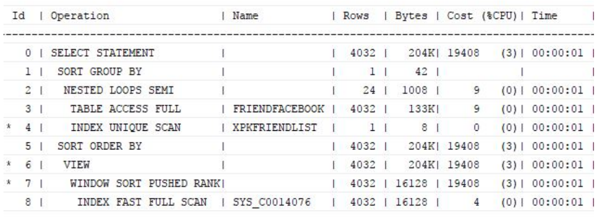


```
| Id  | Operation              | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |             |     1 |    13 | 52538   (2)| 00:00:03 |
|   1 |  SORT AGGREGATE        |             |     1 |    13 |            |          |
|   2 |   VIEW                 | VM_NWVW_2   |  9237 |  117K| 52538   (2)| 00:00:03 |
|   3 |    HASH GROUP BY       |             |  9237 |  270K| 52538   (2)| 00:00:03 |
|*  4 |     HASH JOIN          |             |  9237 |  270K| 52537   (2)| 00:00:03 |
|*  5 |      HASH JOIN         |             |   457 |  9140 | 35007   (2)| 00:00:02 |
|*  6 |       TABLE ACCESS FULL| FRIENDLIST  |    21 |   210 | 17476   (1)| 00:00:01 |
|   7 |       TABLE ACCESS FULL| FRIENDLIST  |   30M|  292M| 17453   (1)| 00:00:01 |
|   8 |      TABLE ACCESS FULL | FRIENDLIST  |   30M|  292M| 17453   (1)| 00:00:01 |
--------------------------------------------------------------------------------------
```

We hypothesized at the beginning of our paper that Neo4j would not be well suited for cases of sort type operations like x percent or ranking. However, our results showed much the opposite, as Neo4j performed adequately compared to Oracle and MongoDB.

Out of the few queries we hypothesized to perform poorly with regard to Neo4j, Text Search was the only one to conform to this, performing poorly throughout our tests no matter if indexes were employed or not. Oracle, while not performing well when indexes were used compared to MongoDB, performed the best in the environment in which indexes were restricted to only our *user_id* field. Once indexes were used liberally, MongoDB became the clear champion of this operation by performing on average only slightly over 200 milliseconds on the large Pokec dataset. To contrast, Neo4j performed the same operation in well over four seconds.

Figure 2: Most Diverse Execution Plan



```
 Id  | Operation                | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
   0 | SELECT STATEMENT         |                 |  4032 |  204K| 19408   (3)| 00:00:01 |
   1 |  SORT GROUP BY           |                 |     1 |    42 |            |          |
   2 |   NESTED LOOPS SEMI      |                 |    24 |  1008 |     9   (0)| 00:00:01 |
   3 |    TABLE ACCESS FULL     | FRIENDFACEBOOK |  4032 |  133K|     9   (0)| 00:00:01 |
*  4 |    INDEX UNIQUE SCAN     | XPKFRIENDLIST   |     1 |     8 |     0   (0)| 00:00:01 |
   5 |  SORT ORDER BY           |                 |  4032 |  204K| 19408   (3)| 00:00:01 |
*  6 |   VIEW                   |                 |  4032 |  204K| 19408   (3)| 00:00:01 |
*  7 |    WINDOW SORT PUSHED RANK|                 |  4032 | 16128 | 19408   (3)| 00:00:01 |
   8 |     INDEX FAST FULL SCAN | SYS_C0014076    |  4032 | 16128 |     4   (0)| 00:00:01 |
```

This brings up a curious anomaly that may hopefully draw the readers attention, as it certainly garnered ours. The data regarding text search, depicted from tables XII-XIV, showed that Neo4j produced almost identical run-times regardless of the number of indexes used. While a similar situation took place with Oracle, this situation was curious in Neo4j, as it increased the overall database size by approximately 12 percent with no noticeable gain in performance.

We would be remiss in not mentioning that within the Oracle 12c release a new feature called Oracle Text was also made available. Oracle Text provides added strategies for keyword searching and many other features which might have offered an increased performance while querying for text [13]. Time was a limiting factor in not including this feature, though future works may be done on this topic to provide updated results.

Exploring into the area of aggregation functions, group by and order by operations exhibited that overall there is not a large disparity between the performance in our selected query environments specifically in the context of our GBAA query. Overall, Oracle did excel in this area.

We mentioned in our results that only Neo4j was able to successfully complete its prescribed query for all datasets. We acknowledge this may be due to a lack of query optimization. For Oracle, once we analyze our execution plan provided in Figure 2, it becomes apparent why this query fails to complete. We repeatedly perform costly operations which require full scans of our respective dataset. The cost of this query becomes too much even in our Pokeclite dataset.

We have provided results to lend support to some of our hypothesized claims involving query performance favoring Neo4j, such as N-Hops and Mutual Friend. As well as queries we felt would be unfriendly towards Neo4j, such as our Text Search, which indeed performed the worst out of our sampled databases. Though certain queries did end up providing surprising results, for instance, our x percent query was performed quite well by not only Oracle but also Neo4j. An area in which we made no claims, aggregation and group by operations, provided data that showed in the instance of our GBAA query, Oracle outperforms the others. Yet, to contrast our first query with our second, Most Diverse, Neo4j was the only database able to complete the operation on all datasets, while Oracle was only able to successfully perform the operation on our smallest dataset.

VI. Hardware

Processor: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
Physical Memory (RAM): 16.0 GB
Hard Drive: 500GB 7200 RPM 16MB Cache SATA 6.0Gb/s 3.5"

VII. Current Versions

Neo4j 3.3.0
Oracle 12c
MongoDB 3.4

References

[1] N. Leavitt, "Will nosql databases live up to their promise?" *Computer*, vol. 43, 2010.

[2] R. Aghi, S. Mehta, R. P. Chauhan, S. Chaudhary, and N. Bohra, "A comprehensive comparison of sql and mongodb databases," 2015.

[3] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," *2011 6th International Conference on Pervasive Computing and Applications*, pp. 363–366, 2011.

[4] V. Abramova, J. Bernardino, and P. Furtado, "Experimental evaluation of nosql databases," *International Journal of Database Management Systems*, vol. 6, no. 3, p. 1, 2014.

[5] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah, "A comparative study: Mongodb vs. mysql," in *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, June 2015, pp. 1–6.

[6] A. Boicea, F. Radulescu, and L. Ioana Agapin, "Mongodb vs oracle – database comparison," pp. 330–335, 09 2012.

[7] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," pp. 15–19, 08 2013.

[8] J. J. Miller, "Graph database applications and concepts with neo4j."

[9] F. Holzschuher and R. Peinl, "Querying a graph database – language selection and performance considerations," vol. 82, no. 1, Part A, 2016, pp. 45 – 68, special Issue on Query Answering on Graph-Structured Data.

[10] M. Ciglan, A. Averbuch, and L. Hluchy, "Benchmarking traversal operations over graph databases," in *2012 IEEE 28th International Conference on Data Engineering Workshops*, April 2012, pp. 186–189.

[11] W. Khan and W. Shahzad, "Predictive performance comparison analysis of relational & nosql graph databases," *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, vol. 8, no. 5, pp. 523–530, 2017.

[12]

[13] "Database installation and administration guide," Jan 2017. [Online]. Available: https://docs.oracle.com/database/121/DFSIG/oracle-text.htmDFSIG269