Performance Analysis of MySQL and Neo4J on StackOverflow User Interaction Graphs

DS220 Independent Study Final Report

Yuya Jeremy Ong
yjo5006@psu.com

1 INTRODUCTION

In data sciences, one of the most fundamental and crucial component of any data process involves the storage, retrieval, and analysis of data - databases have greatly contributed to over the past few decades. Starting with the initial relational SQL-based databases, other NoSQL based solutions such as document databases, graph stores, and key-value store based systems have also emerged as critical infrastructure for any data-driven organization. However, for each of the various solutions that exists out there, we must make smart and conscious choices on the types of systems to implement based on the types of operations we are performing on the data. Hence, depending on how our data is structured, how we plan to perform operations on the data, and in what context we plan to utilize them, the type of database solutions we choose to implement will differ greatly. For this, careful empirical analysis and use of a stringent evaluation metric would help us to make informed decisions on how to select the right tool for the appropriate task at hand.

In this study, we will empirically evaluate two different types of storage solutions: MySQL and Graph Stores (specifically using Neo4J) on a temporal graph-based data structure using dataset curated from the programmer Q&A website "StackOverflow". The objective of this empirical study is to evaluate the relative performance of the several different common database operations and evaluate several different queries on both MySQL and Neo4J data which are structured based on graph-based interaction networks. Our study aims to help understand the strengths and weaknesses of each databases and demonstrate the capabilities of each storage solution. Through this study, the objective of this work aims to help make informed decisions on whether the technology solution utilized can appropriately define the needs based on the requirements and the nature of the system we are building.

2 RELATED WORK

Performance evaluations of database systems have been both done extensively within academia and industry. In the academic setting theoretical results based on simulated queries are performed, while on the other hand, industrial data provides a much more realistic usage of the data based on battle-tested environments integrated within a product. In this section, we briefly evaluate some of the prior work that has been done with regards to the different types of performance evaluations for both MySQL and other NoSQL-based solutions, including graph-based databases.

RDBMS such as MySQL has been well studied, evaluated, and tested among various works both academically and within industry. Most notably, MySQL AB [1] provided a technical whitepaper outlining the key heuristics and evaluation framework for measuring performance of an enterprise RDBMS, along with an example performance evaluation against other types of enterprise solutions such as Oracle and DB2. Similarly, Damodaran et. al [3] also performed a benchmarking task with an MySQL RDBMS against a NoSQL based solution, MongoDB. In their work, they performed an evaluation between MySQL and MongoDB towards simple insertion and search-based operations. Similarly, Truica et. al [7] also performed a similar study with MySQL and other document based NoSQL databases in evaluating performance between various types of solutions.

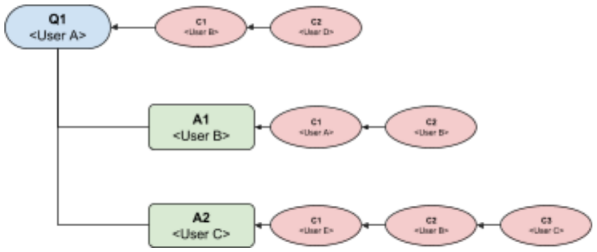
Constantinov et. al [2] performed a general evaluation of complex graph queries on Neo4J and evaluated its performance. This included both read, write, and update processes that are typically performed on any database systems. Pacaci et. al [6] evaluates the performance of various other graph-based database solutions, including Neo4J, to evaluate the performance with respect to Real-Time social Networking Applications. They also perform some evaluations against traditional RDBMS-based systems, however are much more focused on the applicability of graph-based system with a production based setting based on the query latency for real-time usage. Mpinda et. al [5] also performed a similar study with the evaluation of Neo4J and Orient-DB, which are both graph-based databases, however their analysis was focused on the improvement of performance with regards to how indexing in both solutions can provide a gained improvement in performance.

Most works listed above are based on multi-machine based systems, however, our work is only limited to a single machine based performance metric. Therefore bottlenecks of network based latency issues are eliminated from our analysis. Furthermore, when implementing our queries, we will not perform any additional optimization measures which can potentially change the performance of the queries. In our case, we will only work with the suggested optimization strategies that are performed in the background and utilize results as is for our analysis.

3 DATASET

In this project, we consider the "StackOverflow Temporal Network" dataset which was curated by Paranjape et. al [4] in their work. The dataset originates from a Q&A site called StackOverflow where users can post questions and answers to various technical programming or software development related queries. The data comprises of the entire collection of queries from the site's launch up to March 6th, 2016. A typical atomic thread (i.e. a question thread) in StackOverflow can be graphically modeled in Figure 1.

Figure 1: Atomic Structure for Post-Reply Thread in SO

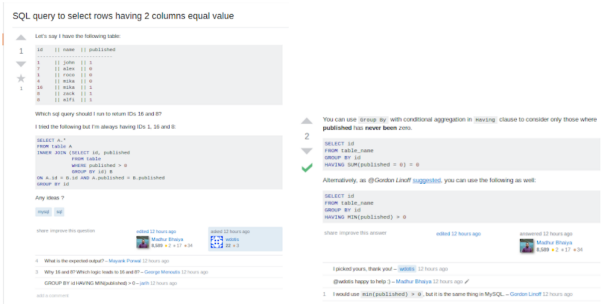


3.1 Data Structure

Unlike a traditional post-reply network thread in a forum setting, the topological structure for the thread supports for the reply of many answers to a single question. For either each question or answer, the platform allows for users to write comments - which are structured in a forward linked-list manner by temporal order. An example of such interaction is demonstrated in the screenshots shown in Figure 2.

The dataset curated by Paranjape et. al, however, only comprises of only the user-to-user based interaction within the StackOverflow site along with the timestamp of the interaction - with all other information regarding the identification of Q&A threads being completely purged from the dataset. The dataset is structured as a directed temporal graph, with each row of the data containing the source user, destination user, and the corresponding time stamp encoded in the UNIX timestamp format. The dataset comes in four types based on the interaction type: answer-question, comment-question, comment-answer, and all interactions, which is a combination of all the interaction types.

Figure 2: Sample StackOverflow Thread (Left: Question Post; Right: Answer Post)



3.2 Considerations

In the following section, we address some of the key considerations we have factored in towards deciding what dataset to perform our benchmarking on. In particular, as we are interested in graph-store database structures, we have found it appropriate to utilize the StackOverflow dataset as they already have emerging properties of network based interaction relationships. For each criteria, we consider various aspects of the graph data structure and how this can benefit the overall evaluation procedure for a database system.

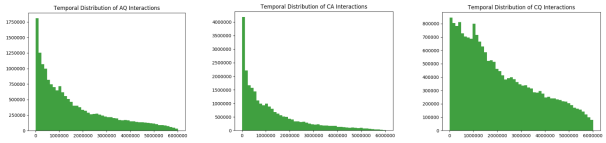
3.2.1 Size. First, the total size of the nodes (for all interactions) amounts to approximately 2.6 million unique entities within the network and the total number of directed edges being around 63.5 million edges and the undirected edges being close to approximately 36.2 million edges. We believe that this an acceptable upper-bound size for us to utilize for the performance evaluation of each database systems.

3.2.2 Temporal. For each edge in the network, we are provided with timestamps which allows us to evaluate temporal interactions and relationships between users. This allows us to further devise queries pertaining to the temporally dynamic features of the dataset. Furthermore, we partition the graph with respect its temporal distribution to obtain several different data points for performance evaluation by sub-partitioning the overall graph structure temporally.

3.2.3 Scalability. One key aspect towards evaluating the scalability of our network is through the sub-partitioning of the graph structure and evaluating the performance each of the systems based on the size and complexity of the network. Through this dataset, we are able to extract subgraphs of the overall graph through temporal partitioning. Effectively, we can perform this by computing the quartile timestamps from the given set of time ranges of the datasets and extract the respective proportions of the data. Performing this will allow us to empirically evaluate the scalability of each algorithm through different sizes and complexities of the network size - therefore providing us with a good intuition for how scalable each algorithms are with respect to the network size. Later sections of this paper will address exact values for how the data will be partitioned. One of our key objectives in this study is to assess the overall relative performance with regards to size scaling, rather than the actual dataset size individually.

In Figure 3, we provide a temporal distribution of the number of interactions (y-axis) for each given UNIX timestamp (x-axis). As observed, we can primarily see a zipf-like distribution for each of the distributions plotted, where the majority of the distribution density is centered in the left end of the plot.

Figure 3: Temporal Distribution of StackOverflow Dataset



3.2.4 Directed Graph. The curated dataset provides a directed relationship between a source and target user in terms of the interactions on StackOverflow. Given this dataset, we can query various questions regarding various interactions between the users as well as compute and derive weighted graph topologies from these directed relationships. As an example, we can consider algorithms to computing contribution ratings using the PageRank algorithm.

Table 1: Dataset Specifications

	All Interactions	Answer-Question	Comment-Question	Comment-Answer
Nodes	2,601,977	2,464,606	1,655,353	1,646,338
Temporal Edges	63,497,050	17,823,525	20,268,151	25,405,374
Static Edges	36,233,450	16,266,395	11,226,395	11,370,342
Time Span	2,774 Days	2,774 Days	2,773 Days	2,773 Days

3.2.5 Weights. By augmenting the original topology of the network, we can devise a variety of user graph structures with different weighting schemes. In particular, by evaluating the relative interactions between each pair of users, we can consider a query that can measure 'reciprocity'. For instance, given two users, A and B, we can compute each of their corresponding support by looking at the ratio of different interaction types over the total interactions between the user.

Furthermore, another type of weight based structure that we can derive from this data set is through the combination of the Comment-to-Question and Comment-To-Answer relationship as a single type of interaction. The rationale behind such grouping indicates that the interaction of commenting is based on providing some type of feedback on the question itself.

4 EVALUATION

4.1 Evaluation Tasks

To heuristically evaluate both MySQL and Neo4J, we consider preforming various benchmarks against many common database queries and tasks which are often utilized commonly in various settings - including data ingestion, simple conditional searches, and quantification of various social network based metrics (i.e. centrality). As we are performing our evaluations on both RDBMS-based and Graph Store-based systems, we appropriately consider hypothesized type of queries for both databases to evaluate which ones actually perform well under certain conditions. In particular we test our databases against both RDBMS-like queries as well as Graph Theoretic-based queries, where we respectively expect MySQL and Neo4J to perform well on based on their known database structure representations.

In our investigation for comparing between MySQL and Neo4J, we consider running the following set of queries:

- Given an arbitrary user ID, retrieve all edges where either the source or target includes that particular ID.
- Given two user IDs, find whether that edge exists in the graph and list all interactions.
- Find the user ID with the most number of interactions initialized (i.e. identify users with the most 'source' occurence).
- Compute the send-to-receive posting ratio for all users.
- Given an arbitrary user ID, compute the local clustering coefficient for that given user.
- Given two user ID, retrieve all shared users that both of these users have interacted with (either both from sending or receiving of posts).

In the queries above, we look at both operations which we believe may be more advantageous for SQL-based operations versus those that are much more appropriate for graph-based solutions. We hypothesized that queries which are defined similarly to a SQL-like structure will tend to work better for MySQL and queries which are meant for graph-based processing will work better with Neo4J. However, we will not be able to see how they actually perform until performing the queries and obtain appropriate evaluation metrics.

4.2 Metrics

To evaluate all the given queries and operations listed above, the primary metrics we will be tracking throughout our experiment will be execution time - essentially evaluating the time and size-scalability trade-off between the various queries performed on the database. In particular, when deciding which technologies to implement, one of the key aspect comes down to the optimization of our compute resources. By understanding the implications of how each operation affects the execution time, we can make better informed decisions about our choice of implementation based on the available hardware stack we have.

4.3 Environment Specifications

In this section we outline the both the hardware and software version specifications for our evaluation. We note these technical specifications in consideration of overall reproduciblity of the results presented in this project.

4.3.1 Hardware. All queries performed in this report utilized a cluster provisioned by the College of IST. The system is a symmetric multiprocessing (SMP) based machine. We were provisioned with a single compute node comprised of two dual-core Intel Xeon E5-2650 processors running at 2.2 GHz each and allocated 80 GB of additional storage and 24 GB of RAM.

4.3.2 Software. The provisioned node from the compute server runs a version of the Red Hat Linux operating system (3.10.0-862.14.4.el7.x86_64) on a x86_64 based processor. Furthermore, we were provided with Python 2.7.5, MySQL 5.5.60-MariaDB Server, and Neo4J 3.4.9 with Cypher-Shell 1.1.7. All commands and operations in this project were performed on the command line interface. For all preprocessing steps in our project, we utilized Python. For each of the respective databases, we used SQL for MySQL and Cypher for Neo4J. We did not utilize any additional libraries or APIs such as the Python wrappers for Neo4J, as this would most likely result in some overhead in the overall execution process.

4.4 Evaluation Caveats

Although we aim to make our performance analysis as fair and equally comparable as possible, there are however some key restrictions that must be considered in the following analysis of this

Table 2: Data sub-graph temporal partition results.

	Total	Partition	25%		50%		75%		Full	
			Size	T.S.	Size	T.S.	Size	T.S.	Size	T.S.
C2Q	25,405,373	6,351,343	6,351,343	1329398958	12,702,686	1374108782	19,054,029	1414991625	25,405,373	1457273420
C2A	20,268,150	5,067,037	5,067,037	1317148858	10,134,074	1357920525	15,201,111	1389990368	20,268,150	1457273420
A2Q	17,823,524	4,455,881	4,455,881	1321391443	8,911,762	1369992944	13,367,643	1412621748	17,823,524	1457266693

report. During the process of our initial work with Neo4J, one the major problems we ran into was the scalability issues with regards to large scale queries of many records - as indicated by our heuristic evaluation on the overall data write throughput during our initial preprocessing phase. More details regarding this data processing latency will be covered in later sections. This as a result already indicates one of the major problems with a graph-based solution and already raises some red flags with respect to performance. Therefore, to mitigate and account for the scalability issues in our analysis, we will have to significantly reduce and cut down our dataset for the Neo4J analysis by several factors.

For our Neo4J dataset, we will consider only up to a maximum of 10K nodes. Thus, each subset of the Neo4J graph will be based on the fractional portion of the first 10K nodes temporally. Hence, when performing our analysis for this report, we will not be able to perform a direct comparison of the time metrics. Instead, we will be characterizing our analysis based on a relative evaluations, such as how well the queries performed with respect to the other dataset type and the scalability patterns as we increase the dataset size.

5 DATA PREPROCESSING

In this section, we outline the preliminary methodology used to preprocess and setup our database for each experiments. In particular we discuss the overall method of partitioning our data, as well as how we setup and generate the corresponding database partitions for each system and the overall data ingestion pipeline for each of the respective database systems.

5.1 Format Conversion

The structure of the original dataset is comprised of a simple flat file based representation. Each record indicates a single interaction between a source user ID and target user ID (both anonymized as integer values) and a UNIX based timestamp of the interaction. The original format of the data was provided with a space delimited text file. To better ensure that we retain ACID requirements of the dataset (especially for MySQL), we introduced the use of a primary key ID for each of the interactions to ensure its uniqueness within the dataset.

We developed a simple Python script to take the original flat file and convert the dataset as a CSV (using comma delimiters) and append a integer based ID in the order that the record was presented to us from the original dataset. Hence, the final format of the csv header structure is defined as: [id, source, target, timestamp]. Furthermore, we have also noticed that the temporal timestamps in the dataset was not ordered appropriately, hence sorting based on timestamp had to be performed in order for the subgraph partition to be sensible. In the corresponding sections, we will utilize this preprocessed CSV file to import our dataset into the respective databases.

5.2 Sub-Graph Partitioning

As mentioned previously, to aid in measuring the scalability aspect of our dataset, we consider partitioning of the dataset as a temporal sub-graph of the original dataset. To perform this, we partition the original dataset as three separate subgraphs and the whole graph - each with an increase of approximately 25% of the original dataset from each sub-graph (i.e. 0.25, 0.50, 0.75, 1.0). For each of the different interaction dataset type, we consider computing the size of the partition, the number of records in each of the sub-graph, and the temporal bounds for the corresponding sub-graph. For this, we wrote a simple Python based script to compute the respective values. The results of the proposed partition are provided under Table 2. Using these predefined partition schema, we repeat our experiment setup based on re-initializing the databases with the given partition. However, this partition is only applicable under MySQL, and correspondingly we use a similar quarter-based ratio for Neo4J, except limited to 10K transactions.

5.3 MySQL Setup

The process for on-boarding our initial MySQL database comprised of setting up the initial full database table schemas, population of the data from the CSV file, and finally generating sub-graph tables correspondingly from the values proposed in Table 2.

First, we initialize the overall table structure for each of the three different types of interactions in the graph. For each table, we have the same schema where we have our integer based ID - which acts as our Primary Key, the source and target nodes, which are both integer based values, and finally the timestamp which is encoded as an integer value as well. We then wrote a SQL based script to correspondingly populate the database tables from the previously preprocessed CSV files. Below we demonstrate a sample SQL command we utilized for the data ingestion process:

```
[MySQL Data Ingestion Command]
LOAD DATA LOCAL INFILE '/data/subgraph/c2a_1.csv'
INTO TABLE c2a
COLUMNS TERMINATED BY ','
LINES TERMINATED BY '\n';
```

5.4 Neo4J Setup

The setup of the Neo4J database comprises of a Cypher based command which takes each of the generated CSV file from our preprocessing script which generates our final graph structure. The script both initializes the node object along with the establishment of the edges between the node. Within each of the nodes, we assign

a "userID" attribute which corresponds to the ID provided by the dataset. Correspondingly for the edge relationship, we map the two listed nodes with the type of response type, as either "A2Q", "C2A", "C2Q" along with a property key indicating the timestamp attribute. We correspondingly perform this processing for the dataset for each of the subset of the dataset as proposed in the previous section by utilizing the "WITH row LIMIT" command to enforce the count in the number of edges processed by our pipeline. The following example command was utilized for the Neo4J data ingestion process:

```
[Neo4J Data Ingestion Command]
PERIODIC COMMIT 100
LOAD CSV FROM 'file:///a2q_1.csv' AS row
WITH row LIMIT 2500
MERGE (u1:User {userID: row[1]})
MERGE (u2:User {userID: row[2]})
MERGE (u1)-[:A2Q {ts: row[3]}]->(u2);
```

6 BENCHMARK EVALUATION RESULTS

In this section we present the benchmark evaluation results of each of the different queries we performed for our experiment. We provide both the tabular data values along with a graphical plot for better visual representation of the overall performance. For each operation, we present first some background rationale and context around the query, the corresponding code for both MySQL and Neo4J. Additionally, we provide some commentary and interesting observations on the results per query as a form of preliminary analysis.

6.1 Data Ingestion

The first benchmark we have is based on the overall data ingestion process, as outlined in the previous section. The major difference to note in this benchmark is the size difference in the dataset we have used. While in the MySQL implementation we utilized the full set, we only used a fractional size of 10K records at most due to low throughput of data processing by the Neo4J database engine.

Based on our empirical results, we claim that one of the major difference factors we would have to consider is the data processing throughput. For this, we can evaluate the average number of records ingested by the database system by looking at the rate of the full C2Q dataset by diving the total number of records by the amount of time it took for the query to execute. For MySQL, we find that it processes approximately 310,997 records per second, while Neo4J can only process 178 records per second. Through this, we can make the justification of reducing our dataset by a fractional subset due to the infeasibility of the scope of the project - and therefore was reduced for the purposes of this investigation.

In the MySQL set, we note that the data scales proportionally in a relatively linear fashion respectively for each of the various sized subgraph dataset, as demonstrated in **Figure 4**. We also note that the scaling is also in accordance to the overall size of the different dataset (graph connection) type and scales accordingly in a linear relationship for both datasets as well, proportionately to the overall size of the dataset.

Table 3: Data Ingestion Benchmark

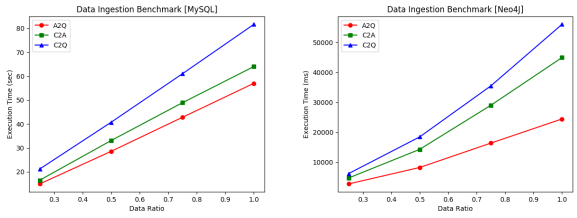
MySQL

	25%	50%	75%	Full
A2Q	14.92 s	28.55 s	42.81 s	57.03 s
C2A	16.48 s	33.08 s	48.91 s	64.04 s
C2Q	21.17 s	40.72 s	61.04 s	81.69 s

Neo4J

	25%	50%	75%	Full
A2Q	2767 ms	6261 ms	16393 ms	24450 ms
C2A	4741 ms	14310 ms	29016 ms	44934 ms
C2Q	6212 ms	18461 ms	35486 ms	56010 ms

Figure 4: Data Ingestion Benchmark Plot



6.2 Query 1: Retrieve User Interaction

The next query is evaluated based on a relatively simple search process, where given an arbitrary known user ID, we return a list of records containing any interaction with that given ID. In other words we look in the database for any edge relationship where the user ID is either in the source or target. In a real-world use-case scenario, this query would be useful in obtaining all the known interactions the user has performed in StackOverflow, where further analysis and other complex data processing can be preformed from this point.

```
[MySQL]
SELECT * FROM a2q WHERE source = 123 OR target = 123;
[Neo4J]
MATCH (:User {userID: "123"})-[r]-(n)
RETURN r, n;
```

Table 4: Query 1 Benchmark

MySQL

	25%	50%	75%	Full
A2Q	1.20 s	2.54 s	3.71 s	4.85 s
C2A	1.52 s	2.71 s	4.08 s	5.79 s
C2Q	1.81 s	3.47 s	5.24 s	6.93 s

Neo4J

	25%	50%	75%	Full
A2Q	4 ms	3 ms	2 ms	2 ms
C2A	2 ms	2 ms	3 ms	3 ms
C2Q	3 ms	4 ms	4 ms	3 ms

Figure 5: Query 1 Benchmark Plot

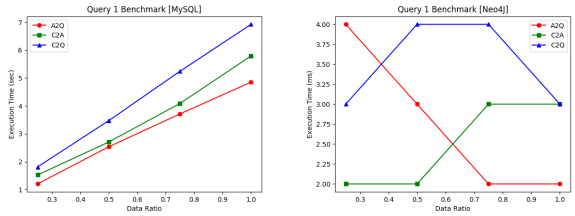
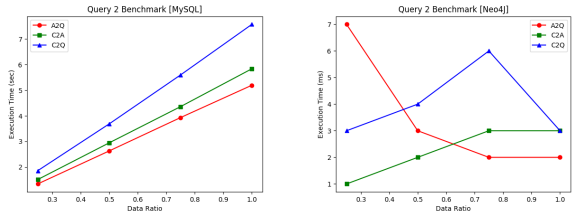


Figure 6: Query 2 Benchmark Plot



From **Figure 5**, we note that the scaling of the performance for MySQL is likewise relatively linear. However, we find that the relative performance of the Neo4J database is mostly constant with each other, as demonstrated in **Table 4**, with most of the execution time not changing too significantly as the scale of the dataset increases.

6.3 Query 2: Retrieve All Node Relations

In this evaluation, we performed another relatively simple search based query, however based on an additional constraint. Given two user IDs of interest, we query the database for the list of all related transactions which occur either between the two user nodes in both interactive directions. In this query we are given not only one single user ID, however two, which would presumably add more computational cycles for our database system to process. The use-case for such query allows us to find all historical interactions between two users, which may be helpful in listing and identifying key trends based on the retrieved data itself.

Since the direction for the interactions that are of interest are only unidirectional, we must consider queries which take into account a bidirectional relationship between the two nodes. Thus we develop our queries based on the following ways:

```
[MySQL]
SELECT * FROM c2q WHERE (source = 2199 AND target = 598740)
OR (source = 598740 AND target = 2199);
[Neo4J]
MATCH (:User {userID: "116"})-[r]-(:User {userID: "122"})
RETURN r;
```

Table 5: Query 2 Benchmark

MySQL

	25%	50%	75%	Full
A2Q	1.34 s	2.62 s	3.93 s	5.19 s
C2A	1.51 s	2.94 s	4.36 s	5.84 s
C2Q	1.85 s	3.68 s	5.60 s	7.58 s

Neo4J

	25%	50%	75%	Full
A2Q	7 ms	3 ms	3 ms	2 ms
C2A	1 ms	2 ms	3 ms	3 ms
C2Q	3 ms	4 ms	6 ms	3 ms

We have also noticed that in **Figure 6**, this query also exhibits a very similar performance pattern as to the previous query. While the MySQL database also scales through in a linear-wise fashion, the Neo4J appears to also exhibit a similar constant pattern with regards to the overall query execution performance - making it appear to run in near-constant time as well.

6.4 Query 3: Most Interaction Source

In this query, we aim to search for the user with the most interactions that the user has initialized - in other words looking at the overall occurrence count of the user in the source attributes of our database. In this query, we evaluate a query which is aimed towards graph data structure analysis of the data. While the inherent design of graph edges are built into the query language for Cypher, computing the relational edges in MySQL requires the GROUP BY pragmatics in order to derive the actual value. A use case for identifying the most number of interactions can help in scenarios where we want to find the most active user in a community base and understand their behavior towards the relationship between other users in StackOverflow.

```
[MySQL]
SELECT source, COUNT(*) AS count FROM a2q
GROUP BY source ORDER BY count DESC limit 1;
[Neo4J]
MATCH (u:User)-[r:A2Q]->()
RETURN u, COUNT(r) AS cnt
ORDER BY cnt DESC LIMIT 1;
```

Table 6: Query 3 Benchmark

MySQL

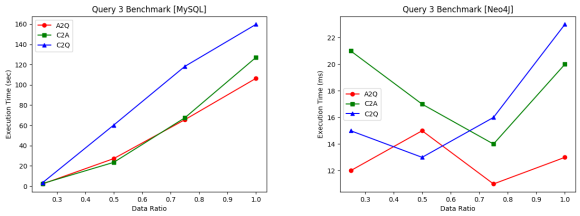
	25%	50%	75%	Full
A2Q	2.16 s	27.19 s	65.63 s	106.51 s
C2A	2.56 s	23.47 s	67.51 s	127.20 s
C2Q	3.38 s	60.29 s	118.30 s	159.81 s

Neo4J

	25%	50%	75%	Full
A2Q	12 ms	15 ms	11 ms	13 ms
C2A	21 ms	17 ms	14 ms	20 ms
C2Q	15 ms	13 ms	16 ms	23 ms

In this query, we notice a somewhat different pattern as opposed to the previous queries we have begun to execute. Although for MySQL, we also notice a similar linear trend, we see that the performance gap between the C2A and C2Q data sets are nearly identical.

Figure 7: Query 3 Benchmark Plot



However, the case for Neo4J indicates a similar trend to the previous query, as shown in **Figure 7**, where the overall performance remains relatively constant despite it's increase in node size for the most part. Though we can start to notice the sudden spike towards the full set in all datasets, which indicates the scalability for Neo4J being very superb when it comes to adding additional edges in the graph.

6.5 Query 4: Send-Receive Ratio

In this query, we evaluate the total ratio of interactions of all users in the database by diving the total number of sent responses to the number of responses received. A use case for this query allows us to gain insight on several different attributes of a given user, such as a metric for reciprocity, finding key contributors of a community, or simply evaluating competency of the developer in StackOverflow.

Similar to the previous query, we perform a GROUP BY-like operation for MySQL and a edge relation retrieval based query for Neo4J - except now the scale and the number of these calls are significantly increased since we must consider both incoming and outgoing edge relationships between the node.

```
[MySQL]
SELECT S.source, (S.SCNT / T.TCNT) AS C2Q_RATIO FROM
(SELECT source, COUNT(*) AS SCNT
FROM c2q GROUP BY source) AS S INNER JOIN
(SELECT target, COUNT(*) AS TCNT FROM c2q
GROUP BY target) AS T ON S.source = T.target;
```

```
[Neo4J]
MATCH (u:User)
MATCH (u)<-[:A2Q]-()
RETURN u, (SIZE((u)-[:A2Q]->())*1.0) /
SIZE((u)<-[:A2Q]-()) AS ratio;
```

Table 7: Query 4 Benchmark

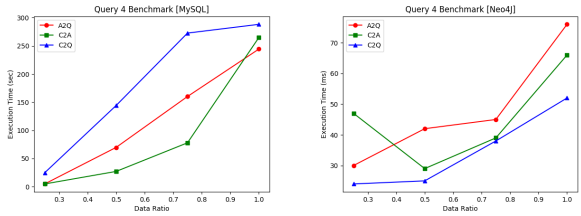
MySQL

	25%	50%	75%	Full
A2Q	4.91 s	69.74 s	160.15 s	244.36 s
C2A	4.90 s	27.10 s	77.92 s	264.92 s
C2Q	24.78 s	144.10 s	272.71 s	288.24 s

Neo4J

	25%	50%	75%	Full
A2Q	30 ms	42 ms	45 ms	76 ms
C2A	47 ms	29 ms	39 ms	66 ms
C2Q	24 ms	25 ms	38 ms	52 ms

Figure 8: Query 4 Benchmark Plot



Based on our empirical results as shown in **Table 7** and **Figure 8**, we can notice that in this query, both MySQL and Neo4J exhibit a similar linear growth with regards to performance and data scalability. However, looking closely at the metrics, we can notice that for values of C2Q are significantly higher than the other two data sets, C2A and A2Q - which is unusual based on the previous set of queries we have performed. We can assume that this may be the case as from our original distribution plot, we saw that the C2Q temporal distributions are significantly higher and therefore tends to generate a much dense graph in general.

6.6 Query 5: Local Clustering Coefficient

In this query, we have computed the clustering coefficient of a network, which is a measure used to define the degree to which the nodes in a graph tend to cluster together. In this formulation, due to the restrictive and limited nature of the scalability of the dataset, we are going to be computing only the local clustering coefficient of a given node.

Given an arbitrary user ID from the database, we can compute the clustering coefficient of the directed network by deriving two key values. First, we compute the number of edges which points towards the defined arbitrary node, denoting it as N_v. Then we compute the number of edges which are connected between the nodes that are connected to the given defined arbitrary node, denoted by K_v. Essentially by formulating this triangular count of the nodes, we can define the total ratio of these two values by the following equation:

$$CC_{Local}(v) = \frac{N_v}{K_v(K_v - 1)}$$

Therefore, our final query is broken down into computing both the values for N_v and K_v, then plugging it into the equation above to derive the final cluster coefficient value.

As demonstrated below, the complexity of the query includes various subqueries for MySQL to compute the following values for N_v and K_v. On the other hand, we have also hand coded the algorithm behind computing the clustering coefficient values for the network. Although it is known that Neo4J provides some of these algorithms out of the box, we have decided to stick to writing our own formulation of the clustering coefficient to match with the performance of the MySQL query formulation.

```
[MySQL]
SELECT Q.N_v / (Q.K_v * (Q.K_v - 1)) AS CC FROM
(SELECT COUNT(*) AS K_v, (SELECT COUNT(*) FROM
(SELECT DISTINCT source, target FROM a2q WHERE
(source IN (SELECT DISTINCT source FROM a2q
WHERE target = 22656) AND target IN
```

```
(SELECT DISTINCT source FROM a2q
WHERE target = 22656)) AND source != target)
AS TMP_B) AS N_v FROM (SELECT DISTINCT source
FROM a2q WHERE target = 22656) AS TMP_A) AS Q;
```

```
[Neo4J]
MATCH (u:User {userID: "383"}) RETURN
u, toFloat(SIZE((u)<-[:A2Q]-()) +
SIZE(()-[:A2Q]->(u))) / (SIZE((u)<-[:A2Q]-()) *
(SIZE((u)<-[:A2Q]-()) - 1)) AS CC;
```

Table 8: Query 5 Benchmark

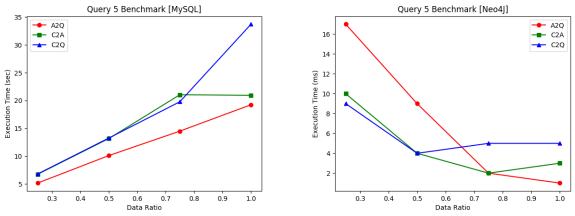
MySQL

	25%	50%	75%	Full
A2Q	5.18 s	10.11 s	14.51 s	19.25 s
C2A	6.75 s	13.16 s	21.05 s	20.94 s
C2Q	6.81 s	13.25 s	19.79 s	33.71 s

Neo4J

	25%	50%	75%	Full
A2Q	17 ms	9 ms	2 ms	1 ms
C2A	10 ms	4 ms	2 ms	3 ms
C2Q	29 ms	4 ms	5 ms	8 ms

Figure 9: Query 5 Benchmark Plot



Looking at **Figure 9**, we can observe some very interesting patterns that differ from the previous types of results that we have seen in our analysis. First, we can notice that for MySQL, the queries still scale relatively well across all proportions for all datasets. However, we find that the performance gap between all of the datasets are much more narrower, despite the different number of nodes and network topologies present in each dataset. In particular, we can observe that the values for both C2A and C2Q for the first 3/4th of the datasets have similar metrics across the board, however, diverges when it comes down to computing from the full dataset.

On the other hand, Neo4J also presents us with another interesting pattern that has never emerged in our analysis. In particular, as the dataset size decreases, the overall execution time decreases - showing an inverse relationship between one another. Furthermore, we can see another similar pattern where the times for A2Q takes longer for the first 3/4th of the data relative to the other datasets. If time persists, further analysis into this matter can help us to understand why Neo4J's benchmark demonstrates this inverse relationship with respect to the scalability aspect of this query operation.

6.7 Query 6: Mutually Shared Nodes Interactions

Another query we have investigated is the retrieval of nodes that are common to two arbitrary user IDs. This is also a very widely used and relevant query for social media network analysis, as it is often employed in the "Mutual Friends" feature list that are used to draw commonality based on the list of similar friends two people share between each other.

In this implementation of the mutual friends retrieval process, the query typically is performed in a three stage process where the first two is finding all the node relationships associated between the two arbitrarily defined user IDs and then performing an inner join between the list of user nodes to find the common set of users the two nodes share.

```
[MySQL]
SELECT DISTINCT COUNT(*) FROM ((SELECT DISTINCT target
FROM a2q WHERE source = 17034 AND (target != source))
UNION ALL (SELECT DISTINCT source FROM a2q WHERE
target = 17034 AND (target != source))) AS U1
INNER JOIN ((SELECT DISTINCT target FROM a2q WHERE
source = 22656 AND (target != source)) UNION ALL
(SELECT DISTINCT source FROM a2q WHERE target = 22656
AND (target != source))) AS U2;
```

```
[Neo4J]
MATCH (u1:User {userID: "91"})-[*..2]-(out),
(u2:User {userID: "383"})-[*..2]-(out)
RETURN u1, u2, out;
```

Table 9: Query 6 Benchmark

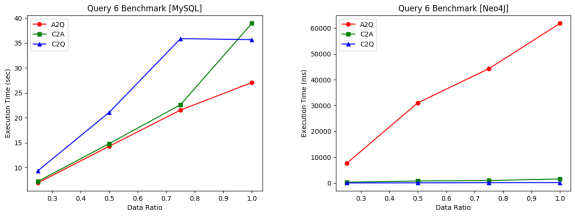
MySQL

	25%	50%	75%	Full
A2Q	6.94 s	14.29 s	21.55 s	27.07 s
C2A	7.23 s	14.82 s	22.62 s	38.94 s
C2Q	9.34 s	21.08 s	35.87 s	35.68 s

Neo4J

	25%	50%	75%	Full
A2Q	7567 ms	31042 ms	44288 ms	61829 ms
C2A	321 ms	804 ms	984 ms	1578 ms
C2Q	40 ms	74 ms	126 ms	187 ms

Figure 10: Query 6 Benchmark Plot



In this experiment, there are also several interesting points to take from these observations, as demonstrated in **Figure 10**. Similar to all the investigations performed so far, MySQL continues

to scale appropriately to the ratio size of the dataset in a linear fashion. However, we see that for Neo4J, we can observe for the first time where MySQL out performs Neo4J by a significant margin in terms of execution time, despite having a fractional node size - as demonstrated in **Table 9**. For Neo4J in particular, we see that both C2A and C2Q have constant time lookups while A2Q's time grows substantially in a linear fashion. We can attribute this growth and significant time increase potentially due to the complex topology present within the A2Q dataset in comparison to the C2A and C2Q dataset.

7 DISCUSSION

In this section, we perform a retrospective analysis of the overall results and place them in context of the bigger picture of this discussion. In particular, we focus on the various aspects between MySQL and Neo4J, not only from a performance standpoint, but also from other aspects of the database system, such as integration, scalability, and maintainability. Having these key ideas in mind would help to make better informed decisions about the process itself and can benefit users who would want to consider using either of these technologies within their platforms.

7.1 Performance Scalability

In this investigation, we have observed various key trends across most of the experiments that we have conducted with regards to the general performance differences between MySQL and Neo4J. However, the key difference when it comes down to the overall performance trade offs based on the CRUD based specifications of the operation.

First and foremost, we have discussed in previous sections that MySQL out performed Neo4J with regards to the initial setup of the database by a significant margin - through our computation of the estimated throughput of queries. We hypothesize this to be the case as internally, Neo4J utilizes extra cycle overheads to perform some type of latent indexing amongst the graph data structure, to initialize some key performance optimization for future queries. This fact is evident through most of the retrieval based queries, where the overall throughput performance of the queries are nearly constant, despite the additive increase in the number of edges appended to the graph database.

On the other hand, one of the major benefits of MySQL can be seen through its linear-like scalability across all metrics. As we have observed in all of the queries we performed, they indicated a strikingly similar type of relationship between the execution run time and the overall database size. This indicates, therefore, that the performance of a given set of queries is most highly dependent on the overall size and complexity of the network topology.

Thus, in a production based setting where we either utilize MySQL or Neo4J, we can consider these characteristics as a key advantage in how we process and utilize these solutions when integrating with a product. For MySQL, we recommend using this database system for any particular operations that have a fairly balanced set of CRUD based operations - especially when writes and updates are frequent. However, Neo4J is a solution which offers higher read performance than write performance, and is therefore better with regards to read-only based operations of the data.

In short, we can generalize that MySQL appears to have a much more balanced set of performance metrics across both the creation and read operations of the database, while Neo4J have extreme trade-offs with respect to the write and reading of the database. Of course, in our analysis we did not perform any further optimization or any additional indexing, so the time factor for those further improvements are not factored into this analysis and can be of an investigation for a future work.

7.2 Integration

Apart from the general performance evaluations of this technology, we would also like to consider the development life cycle and ease of integrating these database systems in the context of a graph-based modeling process. As RDBMS systems are known only for storing transactional data, they are not in particular adaquate for writing graph-based queries due to the semantics of the SQL being based mostly on a transactional database system. Athough, we have shown that it is not impossible to develop some of these various graph-based modeling algorithms for these types of data, we have found that the challenges in doing so comes down to the transpilation of the graph based algorithm with respect to SQL based operations such as GROUP BY and JOINs.

On the other hand, we have seen that the relative ease of Neo4J allows us to easily model graph-based data as the semantics of Cypher is conscious around the inherent graph based relationship that the underlying data structure is based on. This make the user experience (UX) of the development process very easy for the developer, as they have abstracted most of the complex underlying computing process and abstracted the commands under a graph-theoretical mindset, such as using terms like edges, nodes, and directed/undirected within their query.

8 CONCLUSION

In this investigation, we have performed a comprehensive performance analysis of the MySQL RDBMS and Neo4J system using the StackOverflow user interaction graph dataset. In particular, we have demonstrated that these database systems have various strengths and disadvantages depending on their use case and that the user who are using these technologies can choose these solutions based on the key priority their end product serves (i.e. based on the CRUD priorities). We have demonstrated some of the key trends across different scenarios of queries and drew some important ideas from this investigation with regards to the formulation of these operations.

REFERENCES

[1] 2005. *MySQL Performance Benchmarks: Measuring MySQL's Scalability and Throughput.* Technical Report. MySQL AB.

[2] Calin Constantinov, Mihai L Mocanu, and Cosmin M Poteras. [n. d.]. Running Complex Queries on a Graph Database: A Performance Evaluation of Neo4j. ([n. d.]).

[3] B Dipina Damodaran, Shirin Salim, and Surekha Marium Vargese. [n. d.]. Performance Evaluation of MySQL and MongoDB Databases. ([n. d.]).

[4] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).

[5] Steve Ataky Tsham Mpinda, Lucas Cesar Ferreira, Marcela Xavier Ribeiro, and Marilde Terezinha Prado Santos. [n. d.]. Evaluation of graph databases performance through indexing techniques. ([n. d.]).

[6] Anil Pacaci, Alice Zhou, Jimmy Lin, and M Tamer Özsu. 2017. Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications.

In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. ACM, 12.

[7] Ciprian-Octavian Truica, Florin Radulescu, Alexandru Boicea, and Ion Bucur. 2015. Performance evaluation for CRUD operations in asynchronously replicated document oriented database. In *Control Systems and Computer Science (CSCS), 2015 20th International Conference on*. IEEE, 191–196.