Analysis and Benchmarking of Threaded Applications on MySQL and Neo4j

07.12.2021

Link to Github: https://github.com/abhaygoel1906/CS6422

Abhay Goel

Pranay Methuku

Prerna Agarwal

Introduction

Different types of data exist in the world that is used by applications. Some are about people and are used by social media applications, while some are more general, and enhance the knowledge of the individuals. The main objective of this project is to find the most optimal implementation for the back-end of a Social Media data-set. This was done by comparing two completely different approaches to store data: a Relational database, that stores data in the form of relations (attributes and tuples) and a Graph based database, that stores data in the form of Graph nodes, and connects these nodes together to show relations between them.

The goal of this project is to feed various types of such data into two database systems: Relational Databases and Graph Databases. Then, we would run some transactions on the two databases in a single-threaded and multithreaded fashion, to return a total of four observations:

1. Single-threaded RDBMS load
2. Multi-threaded RDBMS load
3. Single-threaded Graph DB load
4. Multi-threaded Graph DB load

Finally, these observations would be compared and the results would be justified. It is important to derive conclusions based on various factors (such as representation of data, and space required to store the data). This report would list the merits and demerits of both the implementations and let the user make the decision.

Infrastructure

This section discusses the platforms used during the project. We will build Java applications that would connect to the two database systems. The Java applications would execute single-threaded and multi-threaded loads on the databases and would be responsible for timing the responses. The voice of database systems is crucial as they are what the project depends on. We have chosen MySQL as the Relational Database and Neo4j as the Graph Database of choice.

The choice of the same language (Java) for both the databases ensures fair comparison, and it's libraries also provide provisions for multi-threading in both the applications. We got the data from the Tweepy API which is an Open Source API, and it contains User and Tweet data from 2013. As the data was completely random, we required several data cleaning processes, and had to alter some of it to demonstrate the purpose of this project. FInally, we compiled all our changes and prepared the data for both databases.

To summarise, these are the platforms of choice were Java, MySQL and Neo4j.

1. MySQL for Relational Database
2. Neo4j for Graph Database
3. Twitter API endpoint for the data itself
4. Java for connecting to both the databases.

Dataset

As mentioned before, the data we used was the same for both types of databases. We fetched approximately Gigabyte of raw data that was collected from the Twitter Open Source API to form our dataset. The tests were run on the same machine for the most fairness. For the tweets, we used Replab 2013 dataset, and did a random hashtag search. Some of the popular ones during that time were #Xbox, #PlayStation, #SpiderMan, etc.

The tweets that we got had the following attributes:

- Author
- Timestamp
- Mentions
- Tweet Text
- Num Likes/Retweets
- Geo location

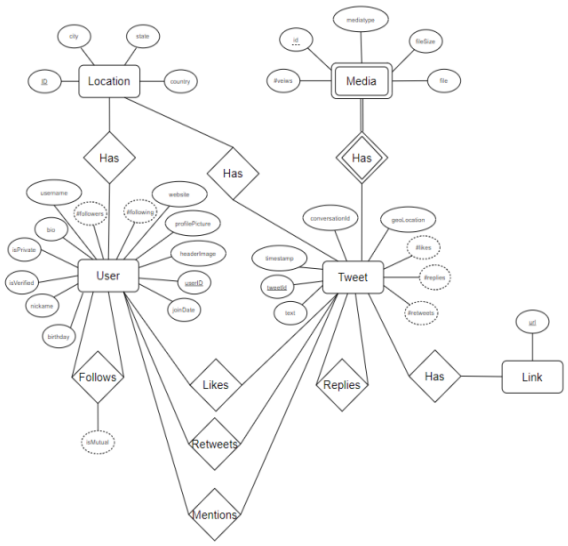
And their users had these attributes:

- Name
- Bio
- Location (if available)
- Profile picture
- Following/Follower graph

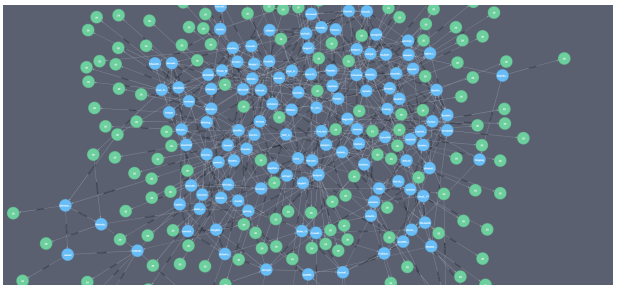
There are around 20,000 tweets, 6,500 users who have 50,000 mentions and 16,000 follows. While these records were completely random, we had to add some data of our own, and perform some data cleaning to demonstrate the project.

Design Specifications

As mentioned before, the data would consist of the mentions and followers statistics that each user received. This data would be perfect as it can reflect the performance in the graph database. The following is the Entity-Relationship diagram of the data that we used:



Below is the Neo4j and MySQL representation for the same.



```
mysql> use assign4
Database changed
mysql> select * from user;
+-------------------------+-------------+-----------+-----------+
| username                | password    | firstName | lastName  |
+-------------------------+-------------+-----------+-----------+
| 2Cool_not_todoschool    | 4242424242  | Smarty    | Pants     |
| 6400_thebestclass       | 6400640044  | Seriously | Itis      |
| abcd                    | 1234567890  | AB        | CD        |
| Aturning_Machine12      | 3333333333  | Alan      | Turing    |
| beBatman!               | 5555555555  | Bruce     | Wayne     |
| bestfriends4ever1       | 44444444433 | C3P0      | Droid     |
| bestfriends4ever2       | 44444444443 | R2D2      | Droid     |
| BuzzyAsAYellowJacket    | 1010101010  | Buzz      | Buzz      |
| coxRaycox               | 4242424242  | Ray       | Cox       |
| customer1               | 1111111111  | One       | One       |
| customer2               | 1011111111  | Two       | Two       |
| deer.john               | 22222022222 | John      | Deer      |
| doe.jane                | 22222222200 | Jane      | Doe       |
| doe.john                | 20000000002 | John      | Doe       |
| efgh                    | 1234567899  | EF        | GH        |
| EmmsBest                | 1000000011  | Emma      | Williams  |
| employee1               | 1000111111  | Employee  | Won       |
| Employeeofthemonth      | 1000011111  | Beast     | Boy       |
```

We executed our queries to generate benchmark scores by running both DBMS solutions with the data loaded into them. While there exist benchmarks that apply for both of these database types, we were able to find the merits and demerits to both these implementations, so the

users can find the one that is tailored towards their needs. We wrote the same queries for both MySQL and Neo4j in Java as it has libraries available for both Neo4j and MySQL, and ran them all on the same machine. Finally, we did a check for correctness by referring to (and verifying with) some research papers that had been conducted along similar lines.

Test Queries

As mentioned before, a set of queries was designed in order to not only benchmark the databases, but also emulate real-world usage. These queries were specifically designed so that they would be similar to the queries that are received by a Twitter database. Below are the test queries that we used for benchmarking performance in both Neo4j and MySQL.

1. Perform a query that receives all links for tweets that have under 30 likes that come from users that have over 250 followers.
2. Perform a query that receives all tweets that have more replies than likes.
3. Perform a query that receives the username of all verified users with more than 10000 followers.
4. Perform a query that returns all tweets made outside of the US with media attached to it.
5. Perform a query that receives all users within a US state that joined before 2012 with a bio that is not NULL and a profile picture.
6. Perform a query that receives all tweets that contain links with more than 100 likes.
7. Perform a query to get all tweets made outside California and Texas with media attached to them.
8. Perform a query that returns each unverified user that is followed by a verified user.
9. Perform a query that returns ALL tweets between the years 2016 and 2020.
10. Perform a query that returns each user with a username that contains the letter “a”.

These were executed one-by-one by the java code and the results were printed out (log output). The same Java code was run in multiple machines to verify for correctness.

Query Explanation

Let’s go over each query and understand it a little bit.

1. This query checks the stats of the user and then links the tweet data. It could have a higher overhead due to linking the data.
2. This query just searches for the tuple values and compares it against the threshold. Significant differences were observed probably due to the way data is stored in the databases.
3. This just checks the data of the user and returns the output. The difference between the executions is the least probably due to the simplicity of the query.
4. This query joins the Geo-data with the User’s tweets data. Not much time difference was seen in the two implementations.
5. This checks for the necessary user data and does not require any joins. Neo4j has performed the

slowest, possibly due to the non-contiguous storage of records.

6. This query analyzes the text of each entry and scans for the keywords “http”. Then it compares the amount of likes with the threshold value.
7. This again checks for the location data, but this time it is the City from where the tweets have been made, making the search to be more meticulous.
8. This query checks for the necessary attribute and a connection between the two users. A join is required in order to run it.
9. A simple query that runs between specific dates in order to see if any optimization is being done in that data structure.
10. This query focuses on searching for ‘A’ or ‘a’ anywhere in the text written. It checks how optimized String pattern matching is in the database systems.

Multi-threading

This section talks about how multi-threading was implemented in both the databases. Apparently, not much research has been done in multithreaded Neo4j queries. Also, we could not find any specific provisions for a parallel graph traversal algorithm which could be applied to the graph database and whose library would be present in Java. Hence, if we were to implement inherent multi-threading in MySQL, it would be unfair for Neo4j, and the results could be skewed.

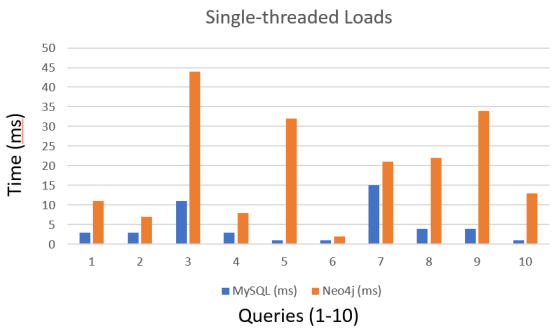
So, we went with a different approach altogether. As all the queries are only reading the data and returning the results, we decided to run the queries in a multi-threaded fashion through Java. This means that we assigned each query to a thread and then ran the code. This led to the results being returned in a random order, each result would be printed as soon as it’s process was finished. The results were similar to a multithreaded environment.

Results

The following are the results for single threaded loads on both MySQL and Neo4j.

Query	MySQL (ms)	Neo4j (ms)
1	3	11
2	3	7
3	11	44
4	3	8
5	1	32
6	1	2
7	15	21
8	4	22
9	4	34
10	1	13

Table 1: time required for Single-threaded Loads

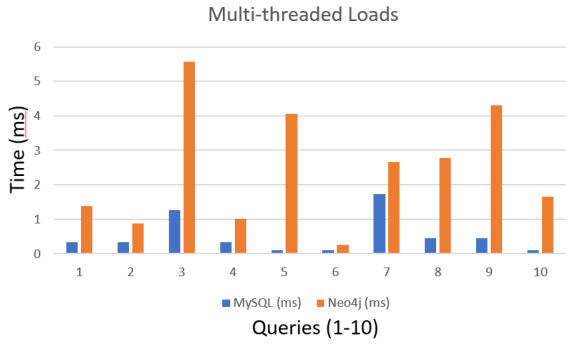


Graph 1: Single-threaded pictorial Comparison

And these are the results for multi threaded loads on both MySQL and Neo4J.

Query	MySQL (ms)	Neo4j (ms)
1	0.344828	1.392405
2	0.344828	0.886076
3	1.264368	5.56962
4	0.344828	1.012658
5	0.114943	4.050633
6	0.114943	0.253165
7	1.724138	2.658228
8	0.45977	2.78481
9	0.45977	4.303797
10	0.114943	1.64557

Table 2: time required for Multi-threaded Loads



Graph 2: Multi-threaded pictorial Comparison

Challenges

This section elucidates on all the challenges faced while building the project. Most prominent ones of those being importing data onto the two Database Systems from the API. Multiple disparities were observed while importing data, having Java as the bridge between the two. For example, the date format had to be changed in order for both the DBs to recognise it as both use a different format. Also, the data received was completely random and didn't have enough graph-connections. We had to change it manually to demonstrate the purpose of the project.

Moreover, Neo4j requires data to be saved into CSV files, in order to load that on the database. CSV files are lossy, and it required some hand-fixing for the data to be loaded as entries.

Also, the API enforces restrictions on how much data can be extracted from the website. Apparently, only 900 entries could be gathered in 15 minutes. The restrictions are also present in the Neo4j side. While loading the data, if the size exceeds the buffer size, it might lose some entries, making the process even more troublesome. These challenges limited the size of the data that we wanted to benchmark the systems with.

Conclusions

From the response time of these queries, we can determine that MySQL was faster than Neo4j each time. All queries run faster, and if we had more data, then the results would have been more prominent.

That said, Neo4j was able to represent the data in a way that was easier to understand. The nodes being connected to each other gave a good idea of the users and their followers, which was not the case with MySQL.

Check for Correctness

It was important for the results to be correct, so we ran the tests on various machines with different system

configurations and verified our results. The check for correctness was also done by referring to and confirming from some research papers written on the similar lines. In the case of single-threaded tests, the check was easy as the same queries were run in the databases too (one-by-one) manually, and the results were verified.

To check the correctness of the multi-threaded tests, we checked certain research papers based on parallel algorithms for RDBMS data traversals [8, 9], and looked up research done on Multi-threading in Neo4j in order to double-check our results.

Future Work

In the future, we would like to conduct similar tests with other database solutions like NoSQL, and between similar options (like between PostgreSQL and MySQL). These benchmarks would not only give us a better idea of how optimized some systems are, but also tell us how the Database systems interact with the Operating system and other system resources.

Moreover, we would like to see how these database systems perform under very large loads. Arguably, the loads that we used could fit into the memory, and we would like to use more data for our tests. Also, we would like to check the fault-tolerance of these systems.

References

1. Neo4j (4.3.6) download link: https://neo4j.com/download-center/#community
2. MySQL (8.0) download link: https://dev.mysql.com/downloads/workbench/
3. Miller, Justin J. "Graph database applications and concepts with Neo4j." Proceedings of the southern association for information systems conference, Atlanta, GA, USA. Vol. 2324. No. 36. 2013.
4. Cheng, Yijian, et al. "Which category is better: Benchmarking relational and graph database management systems." Data Science and Engineering 4.4 (2019): 309-322.
5. Holzschuher, F., & Peinl, R. (2014). Performance optimization for querying social network data. In EDBT/ICDT Workshops (pp. 232-239).
6. Fernandes, D., & Bernardino, J. (2018, July). Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. In Data (pp. 373-380).
7. Gupta, S. (2015). Building Web Applications with Java and Neo4j. Packt Publishing Ltd.
8. Bader, David A., and Kamesh Madduri. "Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2." 2006 International Conference on Parallel Processing (ICPP'06). IEEE, 2006.
9. Laxman Dhulipala, Guy Blelloch, Julian Shun, Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing.
10. Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, Julian Shun, Optimizing Ordered Graph Algorithms with GraphIt.