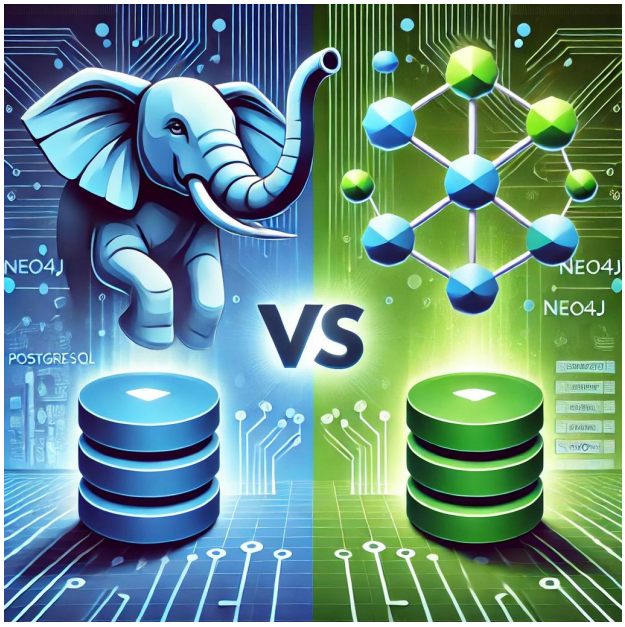
Modelação de Dados

Rafael Mira & José Castanhas
59243 & 67608

Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa

December 15, 2024

Comparative Analysis of PostgreSQL Vs Neo4j



Contents

1 Introduction

The goal of this project is to compare the performance of a relational versus a graph database with the same dataset modeled appropriately in each database. We choose to model a social network since it gives us different options to explore the dataset with diverse queries.

The initial idea for this project was to do a query driven modeling where the database schemas would be designed based on the queries we wanted to make in order to have the best performance out of the database systems. This approach had its own issues, namely, generating a realistic dataset that captures the relations of a social network is not trivial to do without previous knowledge and would require us to spend probably most of the project time just trying to understand how to generate a correct probabilistic model of a social network. We also tried to find a tool that allowed us to specify the schema and generate a realistic dataset accordingly to the provided schema, but we didn't find one that matched our needs.

Given the setbacks we faced, we choose to use a dataset with a correct probabilistic model of a social network where we don't control the generated schema. To generate the dataset, we used the LDBC SNB model, more on this in the next topic where we explain what is LDBC SNB.

2 Linked Data Benchmark Council SNB Overview

LDBC's Social Network Benchmark (LDBC SNB) is an industrial and academic initiative and its goal is to define a framework where different graph based technologies can be fairly tested and compared, that can drive the identification of systems bottlenecks and required functionalities.

While LDBC SNB provides a driver to run their custom workloads, the most important piece of software for our project is the Datagen. The Datagen is a scalable synthetic data generator based on the MapReduce paradigm, which produces networks with distributions, cardinalities and correlations that have been finely tuned to reproduce a real social network such as Facebook. It also uses real data from DBpedia to ensure that attribute values are realistic and correlated.

The generated schema represents a snapshot of the activity of a social network during a period of time. The schema includes entities such as Persons, Organizations, and Places and also models the way persons interact, by means of the friendship relations established with other persons, and the sharing of content such as messages, replies to messages and likes to messages. People form groups to talk about specific topics, which are represented as tags.

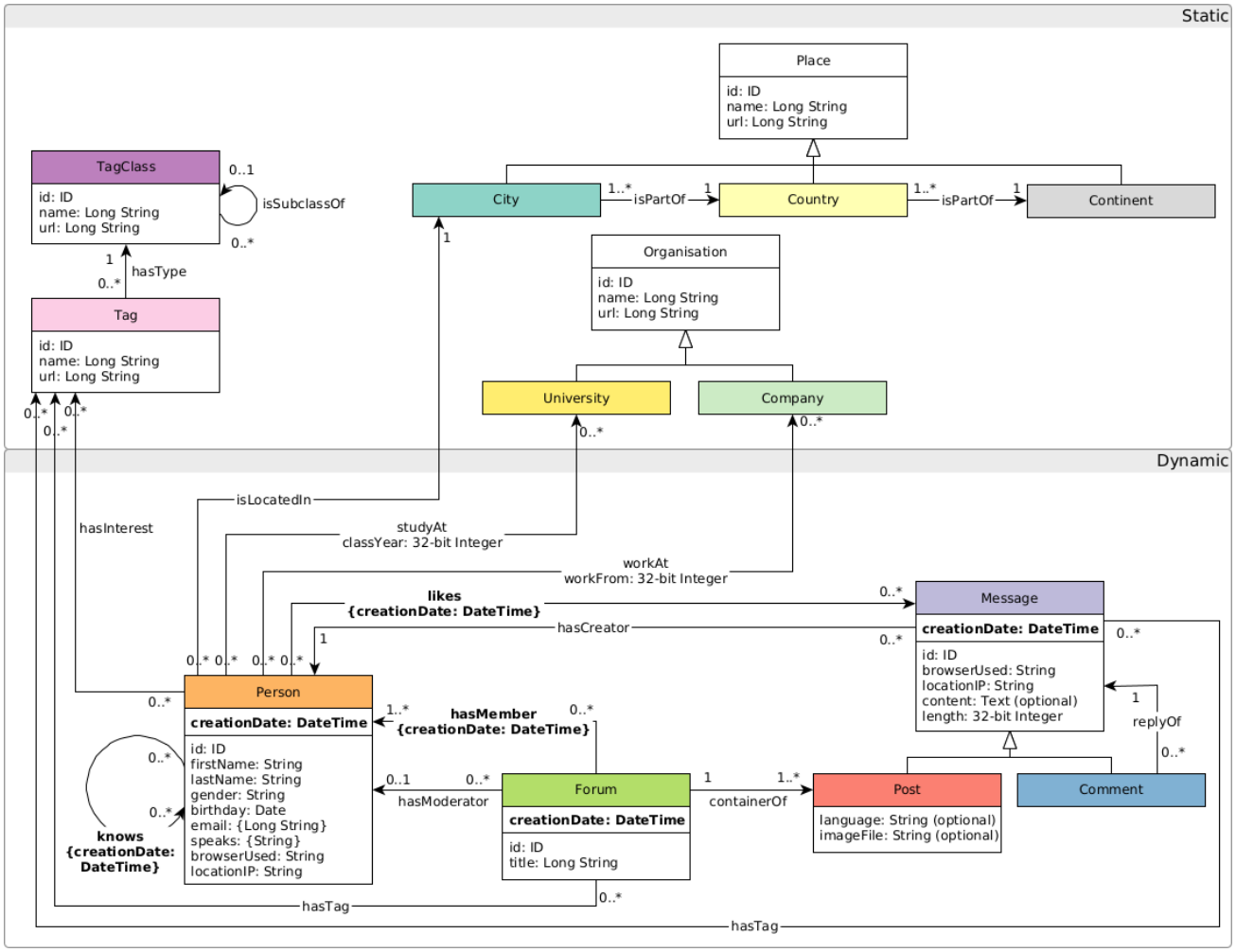


Figure 1: LDBC SNB schema [2]

3 RDBMs and GDBMs Overview

We chose PostgreSQL as our relational database to benchmark. In relational databases, data is structured in tables with rows and columns, and have a predefined schema. Relationships between tables are represented through foreign keys, and queries involving related data often require table joins. Internally, relational databases rely on B+-trees or other indexing structures (e.g. bitmaps indexes) to locate specific rows efficiently.

While relational databases are efficient for many use cases, when performing complex operations such as those involving joins with large tables, recursion, sorting operations, these can become resource-intensive as the depth or volume of the tables increases.

As the graph database, we chose Neo4j. Graph databases use a different approach, storing data as nodes, relationships, and properties. Unlike relational databases, relationships are stored as first-class entities, allowing direct traversal between connected nodes. When executing a query, a graph database typically performs a single index lookup to find the root node. From there, it uses an index-free adjacency mechanism, where each node maintains direct references to its connected nodes. This eliminates the need for computationally expensive joins, enabling the database to traverse relationships in constant time for each hop. This design is particularly advantageous for queries involving multiple hops or highly interconnected data, as the traversal is streamlined and efficient, leveraging the natural structure of the graph.

4 Methodology to perform the benchmark

4.1 Preparing the dataset

4.1.1 Generating the data

To generate the dataset we used a docker image provided by LDBC which allows specifying the scale factor and the output format for the generated CSV files. Since we used Docker, we didn't have to install any software to run the datagen, saving us a lot of time in setup and configuration. In our experiments, we used datasets of different sizes to have more comparison data. Below, we provide details about the dataset sizes and their representation in each database system.

Scale Factor	Size GB
0.3	0.3
1.0	1.0
3.0	3.0

Table 1: Scale Factor and corresponding Sizes

4.1.2 Storage size comparison for 0.3 GB dataset

Table Name	Total Size (MB)	Table Size (MB)	Index Size (MB)	Rows Count
message	119.45	100.63	18.82	874 463
forum_members	62.21	38.88	23.34	773 633
message_tags	57.75	33.75	24.00	796 077
likes	38.88	24.38	14.50	479 987
comment	31.28	20.75	10.53	489 040
post	25.06	16.75	8.31	385 423
forum_tags	8.91	5.25	3.66	120 052
has_interest	6.43	3.75	2.68	87 335
edges	5.53	5.50	0.03	91 632
knows	3.80	2.38	1.42	45 816
forum	3.65	2.82	0.83	36 505
tag	1.40	1.01	0.39	16 080
works	0.70	0.41	0.29	8 022
university	0.66	0.48	0.19	6 380
person	0.52	0.38	0.13	3 788
studies	0.30	0.16	0.14	3 038
company	0.19	0.10	0.09	1 575
city	0.16	0.08	0.08	1 343
country	0.02	0.01	0.02	111
continent	0.02	0.01	0.02	6
tag_class	0.02	0.01	0.02	71
Total	**354.43**	245.26	109.25	4 220 377

Figure 2: PostgreSQL database size for 0.3 GB dataset

Node	Count
Message, Comment	489 040
Message, Post	385 423
Forum	36 505
Tag	16 080
University	6 380
Person	3 788
Company	1 575
City	1 343
Country	111
TagClass	71
Continent	6
Total Nodes	**940 322**

Relationship	Count
TAG_OF	916 129
LOCATED_IN	886 206
POSTED_BY	874 463
MEMBER_OF	773 633
REPLY_OF	489 040
LIKED_BY	479 987
POSTED_IN	385 423
HAS_INTEREST	87 335
KNOWS	45 816
MODERATOR_OF	36 505
TAG_TYPE	16 080
WORKS_AT	8 022
STUDIES_AT	3 038
CITY_OF	1 343
COUNTRY_OF	111
SUBCLASS_OF	70
Total Relationships	**5 003 201**

Database Size
454.33 MB

Figure 3: Neo4j database size for 0.3 GB dataset

4.1.3 Storage size comparison for 1 GB dataset

Table Name	Total Size (MB)	Table Size (MB)	Index Size (MB)	Rows Count
message	397.06	335.63	61.44	2 861 962
forum_members	232.48	144.88	87.61	2 909 768
message_tags	211.87	123.75	88.12	2 928 064
likes	149.70	93.38	56.33	1 870 268
comment	111.08	73.75	37.33	1 739 438
post	73.33	49.25	24.08	1 121 226
forum_tags	24.18	14.25	9.93	328 584
edges	20.03	20.00	0.03	346 028
has_interest	17.47	10.25	7.22	238 052
knows	14.13	8.88	5.26	173 014
forum	10.33	8.13	2.20	100 827
works	1.81	1.10	0.71	22 044
tag	1.40	1.01	0.39	16 080
person	1.34	1.06	0.27	10 295
studies	0.71	0.41	0.30	8 309
university	0.66	0.48	0.19	6 380
company	0.19	0.10	0.09	1 575
city	0.16	0.08	0.08	1 343
country	0.02	0.01	0.02	111
tag_class	0.02	0.01	0.02	71
continent	0.02	0.01	0.02	6
Total	**1268.00**	886.42	381.65	14 643 811

Figure 4: PostgreSQL database size for 1 GB dataset

Node	Count
Message, Comment	1 739 438
Message, Post	1 121 226
Forum	100 827
Tag	16 080
University	6 380
Person	10 295
Company	1 575
City	1 343
Country	111
TagClass	71
Continent	6
Total Nodes	**2 997 352**

Relationship	Count
TAG_OF	3 256 648
LOCATED_IN	2 878 914
POSTED_BY	2 860 664
MEMBER_OF	2 909 768
REPLY_OF	1 739 438
LIKED_BY	1 870 268
POSTED_IN	1 121 226
HAS_INTEREST	238 052
KNOWS	173 014
MODERATOR_OF	100 827
TAG_TYPE	16 080
WORKS_AT	22 044
STUDIES_AT	8 309
CITY_OF	1 343
COUNTRY_OF	111
SUBCLASS_OF	70
Total Relationships	**17 196 776**

Database Size
1.41 GB

Figure 5: Neo4j database size for 1 GB dataset

4.1.4 Storage size comparison for 3 GB dataset

Table Name	Total Size (MB)	Table Size (MB)	Index Size (MB)	Rows Count
message	1154.00	977.63	176.38	8 214 378
forum_members	701.65	437.38	264.27	8 779 234
message_tags	655.33	382.75	272.58	9 061 103
likes	498.84	310.88	187.96	6 245 267
comment	340.32	225.75	114.57	5 343 582
post	188.38	126.75	61.63	2 873 419
edges	61.00	61.00	0.00	1 057 792
forum_tags	58.66	34.25	24.41	809 991
has_interest	43.03	25.25	17.78	589 533
knows	42.34	26.38	15.96	528 896
forum	24.44	19.13	5.31	245 524
works	4.37	2.70	1.67	54 135
person	3.15	2.56	0.59	25 066
studies	1.66	1.01	0.66	20 113
tag	1.40	1.01	0.39	16 080
university	0.66	0.48	0.19	6 380
company	0.19	0.10	0.09	1 575
city	0.16	0.08	0.08	1 343
country	0.02	0.01	0.02	111
tag_class	0.02	0.01	0.02	71
continent	0.02	0.01	0.02	6
Total	**3779.66**	2635.12	1144.59	43 870 748

Figure 6: PostgreSQL database size for 3 GB dataset

Node	Count
Message, Comment	5 343 582
Message, Post	2 873 419
Forum	245 524
Tag	16 080
University	6 380
Person	25 066
Company	1 575
City	1 343
Country	111
TagClass	71
Continent	6
Total Nodes	**8 513 157**

Relationship	Count
TAG_OF	9 870 138
LOCATED_IN	8 250 022
POSTED_BY	8 217 001
MEMBER_OF	8 780 738
REPLY_OF	5 343 582
LIKED_BY	6 244 522
POSTED_IN	2 873 419
HAS_INTEREST	589 533
KNOWS	528 896
MODERATOR_OF	245 000
TAG_TYPE	16 080
WORKS_AT	54 135
STUDIES_AT	20 113
CITY_OF	1 343
COUNTRY_OF	111
SUBCLASS_OF	70
Total Relationships	**51 034 703**

Database Size
4.44 GB

Figure 7: Neo4j database size for 3 GB dataset

4.1.5 Converting the data to our schema

Before inserting the data into the databases, we transformed the CSV files generated by LDBC Datagen to align with our database schema, ensuring a smoother and more efficient data loading process. To achieve this we utilized Python pandas library.

We converted the UML schema provided by LDBC Datagen into an ER model to offer a clearer and more intuitive representation of the schema. This ER diagram closely mirrors our relational database schema, with a few notable exceptions. For instance, attributes highlighted with a red background are not represented in our database. Specifically, the URL attributes are irrelevant to our project, as they only point to resources in DBpedia. These attributes would be more suitable in projects involving for instance RDF or OWL query languages that could leverage them effectively. In the Person entity, some multivalued attributes were excluded from the model. Including them would have required additional processing of the CSV files, which we deemed unnecessary.

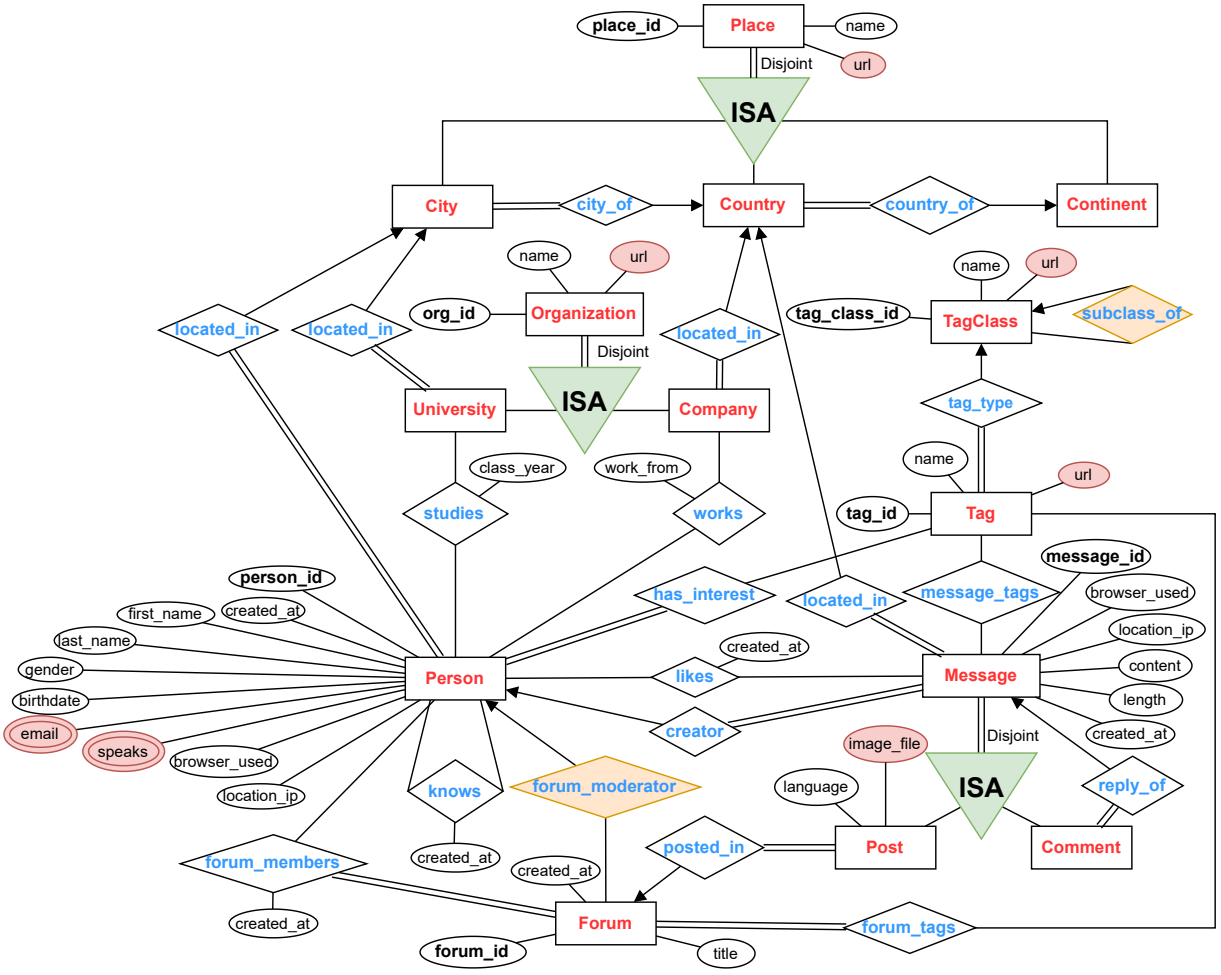


Figure 8: LDBC schema converted to ER

Another distinction are the non-mandatory many-to-one relationships, indicated by the orange background. For simplicity, we did not create separate tables for these relationships in our relational schema. Instead, we added an extra column to the "many" side of the relationship, which stores a foreign key to the "one" side, allowing it to be nullable (e.g., for the forum_moderator relationship, the forum table includes a moderator column). Had we chosen to avoid null values, we would have created separate tables for the forum_moderator and subclass_of relationships. While this approach would have resulted in a more normalized schema, we opted

for a simpler model. Based on the fact that such granular detail in the schema design was not essential to achieve the overall objectives of our project. Additionally, the simpler approach made it more straightforward to generate the corresponding dataset accurately and efficiently.

When converting the ER schema to the relational schema, there are some noteworthy details to consider. For example, we did not include a separate place table. This decision stems from the fact that the place entity represents mandatory and disjoint inheritance, meaning that a place can only be a city, country or continent, but never more than one simultaneously. Instead, we modeled this by creating separate tables for city, country, and continent, each containing the attributes of the conceptual place table. A similar approach was taken for the organization table, where the same mandatory and disjoint inheritance rules apply.

For the message table, we also encounter a case of mandatory and disjoint inheritance but, unlike other entities, there are relationships involving the top-level entity that needed to be preserved. To handle this, we retained the message table in our relational schema. The subtypes, post and comment, were modeled as separate tables that reference the message table through a foreign key, effectively representing the inheritance. In this approach, the primary key of the post and comment tables also serves as a foreign key pointing to the message table, ensuring a consistent representation of the hierarchy while maintaining the relationships associated with the parent entity.

For the graph model, we designed it by translating the ER diagram into a graph structure following the modeling techniques learned.

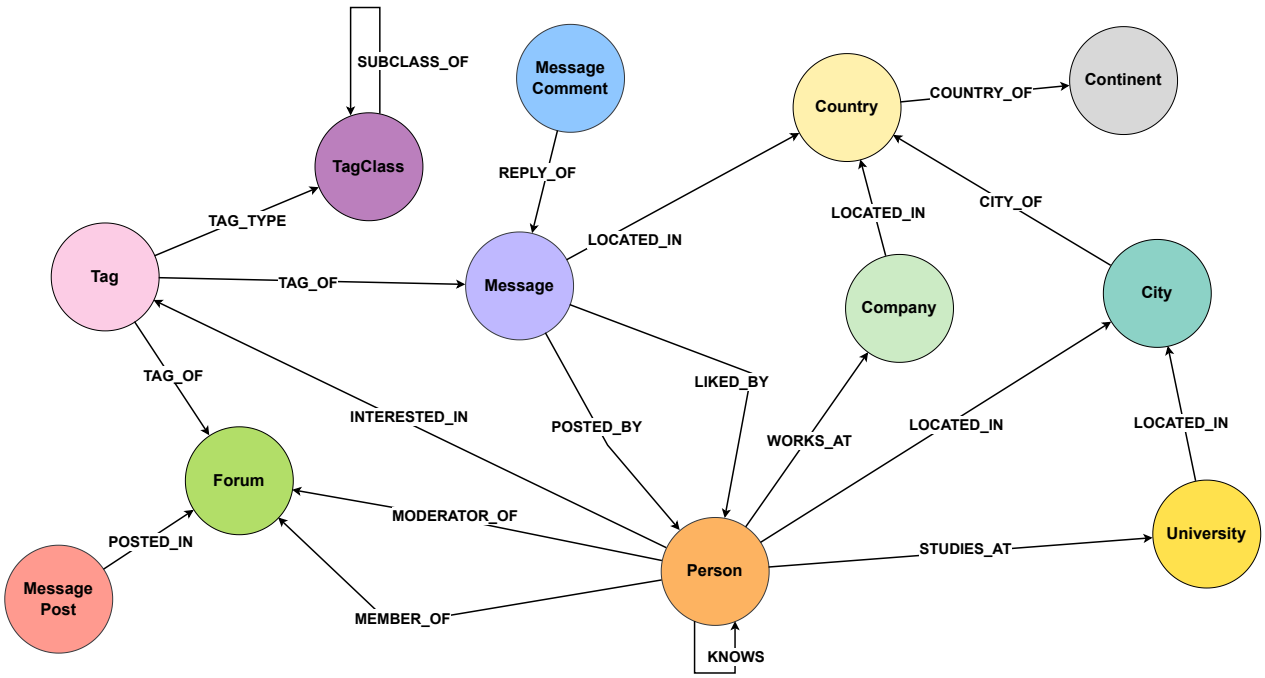


Figure 9: ER convert to Graph model

Note: The message node does not exist independently, as every message is either a post or a comment. We present this simplified schema for clarity reasons; otherwise, the relationships involving posts and comments would appear duplicated, making the graph model harder to understand. In this representation, both comment and post nodes are assigned two labels: their specific label (e.g., comment or post) and the shared parent label (message). Additionally, only post nodes are connected to the forum node, while only comment nodes have a relationship with another message node.

4.1.6 Bulk loading the data

To efficiently load the dataset into our databases, we fine-tuned the databases configuration to optimize the bulk load process, ensuring better performance during data insertion.

In PostgreSQL, we leveraged the COPY command, as the documentation recommended it as the fastest method for loading large datasets from a file. Once the data was successfully loaded, we proceeded to create the necessary constraints and indexes. Finally, we analyzed the database to update the statistics, ensuring optimal query performance and system efficiency going forward.

Parameter	**Default value**	**New value**	**Purpose**
shared_buffers	128 MB	2 GB	Caching data
work_mem	4 MB	128 MB	Sorting operations
maintenance_work_mem	64 MB	2 GB	Index creation and maintenance tasks
max_wal_size	1 GB	4 GB	Avoid frequent checkpoints
checkpoint_timeout	5 min	10 min	Reduce disk I/O
wal_buffers	-1	128 MB	Increases the WAL buffer size

Table 2: PostgreSQL load configuration

In Neo4j, we leveraged APOC procedures to optimize insertion performance, using a combination of batching and parallel insertions to achieve a significant speed improvement. This approach contrasts with PostgreSQL, where constraints are typically created after the dataset is inserted. In Neo4j, the paradigm differs due to its internal mechanics: inserting a relationship requires matching two nodes, typically using some indexed property. By creating unique constraints on the primary key of each node before data insertion, we enhance the query optimizer, allowing it to locate nodes more efficiently and significantly speeding up the insertion process.

When bulk inserting data into Neo4j, batching is absolutely essential to prevent crashes during the insertion transaction. Without batching, loading larger datasets would be infeasible due to the way memory is managed in Neo4j. Even with a substantial heap size allocated to Neo4j (e.g., 12GB), the process would still fail, resulting in a Java OutOfMemoryError. This highlights the critical importance of batching to ensure the successful handling of large-scale data loads.

Parameter	**Default value**	**New value**	**Purpose**
server.memory.heap.initial_size	512MB	4GB	Minimum heap size
server.memory.heap.max_size	512MB	8GB	Maximum heap size
server.memory.pagecache.size	-1	3GB	Caching (similar to shared_buffers)

Table 3: Neo4j load configuration

Note: In both these tables, the -1 means that the value is calculated using heuristics based on the amount of RAM and on the value of some parameters, namely, the shared_buffers parameter in the case of PostgreSQL and the heap size in the case of Neo4j.

4.2 Benchmarking Tool

To benchmark our experiments, we used JMetter. This tool allows us to automate queries, simulate user loads, and collect data on key metrics, providing a basis for comparison between Neo4j and PostgreSQL.

In all our benchmarks, we used a pre-initialized connection pool to ensure no bottleneck from threads trying to acquire a connection to either DBMS, and we also used prepared statements with dynamic inputs loaded from csv files to have a richer variety of input values in our queries.

4.3 Benchmarks specification

Our tests involved benchmarking ten distinct queries on both DBMS platforms, utilizing two different machines, various database configurations and different dataset sizes. We conducted two types of benchmarks: individual and concurrent.

For the individual tests, we conducted a thorough evaluation by running each query individually a total of sixteen times. These sixteen executions were divided into four distinct input sets[1]. The testing process followed a structured pattern: input set 1 was executed four times consecutively, followed by input set 2 executed four times, and so on for input sets 3 and 4. This design aimed to analyze how the database optimizer and planner handled caching of execution plans.

The concurrent benchmarks involved progressively increasing the number of virtual users (VUs). The test began with 4 VUs, maintaining this load for 2 minutes. Subsequently, the load was doubled to 8 VUs and held constant for another 2 minutes. This pattern continued with 16, 32, 48, and finally 64 VUs, each level sustained for 2 minutes before escalating to the next.

Figure 10: Numbers of virtual users at each phase of the concurrent benchmark

We conducted both individual and concurrent benchmarks using various database configurations. The tests were first performed with the default DBMS settings and then repeated with progressively increased memory allocations: 1 GB, 2 GB, 4 GB, and 8 GB.

[1] A tuple of inputs, e.g., (12242423, 'John', 'Germany')

4.4 Configurations

4.4.1 Machines specification

To run our experiments we used two different machines with different configurations.

Id	OS	CPU	Cores	Threads	RAM	Disk R/W speed
1	Windows 10	AMD Ryzen 5 3600X	6	12	16GB 3600 MHz	SSD
2	Windows 11	AMD Ryzen 7 7800X3D	8	16	32GB 4800 MHz	SSD

Table 4: Machines specification

4.4.2 DBMS configuration

Given that both DMBS are quite different in their internal structure, we tried balancing the tests by using similar configuration values in keys properties.

Test name	Postgres configuration		Neo4j configuration	
	Parameter	Value (MB)	Parameter	Value (MB)
Default config	shared_buffers	128	dbms.memory.pagecache.size	7680
	work_mem	4	dbms.memory.heap.initial_size	512
			dbms.memory.heap.max_size	512
Memory 1GB	shared_buffers	1024	dbms.memory.pagecache.size	1024
	work_mem	32	dbms.memory.heap.initial_size	512
			dbms.memory.heap.max_size	1024
Memory 2GB	shared_buffers	2048	dbms.memory.pagecache.size	2048
	work_mem	64	dbms.memory.heap.initial_size	1024
			dbms.memory.heap.max_size	2048
Memory 4GB	shared_buffers	4096	dbms.memory.pagecache.size	4096
	work_mem	128	dbms.memory.heap.initial_size	2048
			dbms.memory.heap.max_size	4096
Memory 8GB	shared_buffers	8192	dbms.memory.pagecache.size	4096
	work_mem	128	dbms.memory.heap.initial_size	4096
			dbms.memory.heap.max_size	8192

Table 5: Configuration used in the experiments

5 Results

5.1 Loading times

Dataset (GB)	Postgres Load (s)	Neo4j Load (s)	Postgres Constraints (s)	Neo4j Constraints (s)
0.3	9	128	5	1
1.0	33	419	21	1
3.0	101	1196	88	1

Table 6: Data Loading and Constraints Creation Times in Machine 1

Dataset (GB)	Postgres Load (s)	Neo4j Load (s)	Postgres Constraints (s)	Neo4j Constraints (s)
0.3	4	87	3	1
1.0	17	386	12	1
3.0	43	917	40	1

Table 7: Data Loading and Constraints Creation Times in Machine 2

The results clearly show that bulk inserting data into PostgreSQL was consistently faster than in Neo4j.

This difference can be attributed to several factors, such as the import tool used, whether parallelism was enabled, whether batching was applied, and the batch size chosen.

Admittedly, we did not use the most efficient method to load data into Neo4j. While the APOC procedures we used offer improvements over the standard LOAD CSV command by enabling batching and parallelism, fully optimizing data insertion in Neo4j requires additional configuration and setup steps that, given the scope of this project, were unnecessary. However, for scenarios where performance is critical or the dataset is significantly larger, the recommended approach would be to use the Neo4j Admin CLI tool for data loading.

Note 1: In the following section, we present the benchmark results. To ensure a easier comparison, some graphs have been zoomed in to maintain a consistent scale when displayed side by side. For any graph that is not fully shown within its corresponding benchmark, we provide a clickable link directing you to the complete graphs, which are included in the annex at the end of this document.

Note 2: As explained earlier, each configuration involved running four different input sets, with each input set executed consecutively four times. The same inputs were used across all configurations to ensure result correlation. The graphs are organized as follows: each group of four connected dots represents an input set, and each dot corresponds to a query that was executed.

Note 3: It is important to note that, for each run (experiment with a different configuration), the DBMS was restarted to ensure consistency across benchmarks and to evaluate the impact of starting with a cold cache. Additionally, queries were executed sequentially (e.g., query 1 was followed by query 2, and so on until query 10) without restarting the DBMS server for each query. This execution order could influence query performance, as the optimizer may benefit from previously generated execution plans.

5.2 Benchmarks

5.2.1 Query 1 - Transitive friends with a certain name (max step = 3)

Given a start Person with ID $personId, this query finds Persons with a given first name ($firstName) that the start Person is connected to by at most 3 steps via the knows relationships.

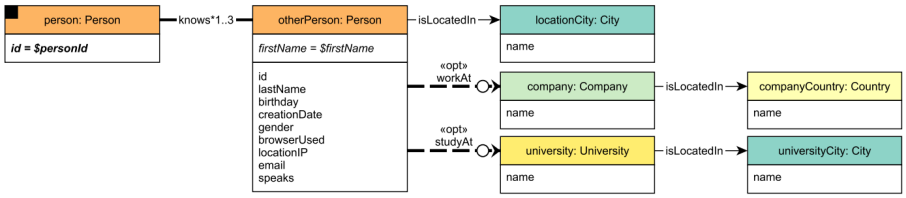


Figure 11: Visualization of the query
[1]

In PostgreSQL, we did this query recursively. We wanted to analyze how a recursive query would perform with different dataset sizes and with different step sizes. In this query, we limited the step of the recursion to 3.

Dataset 0.3 GB

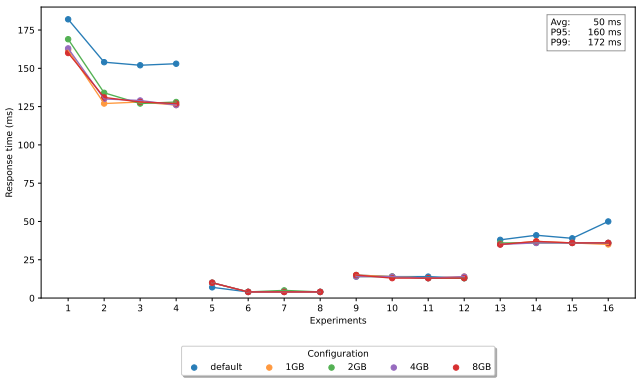


Figure 12: Machine 1 - Postgres response time

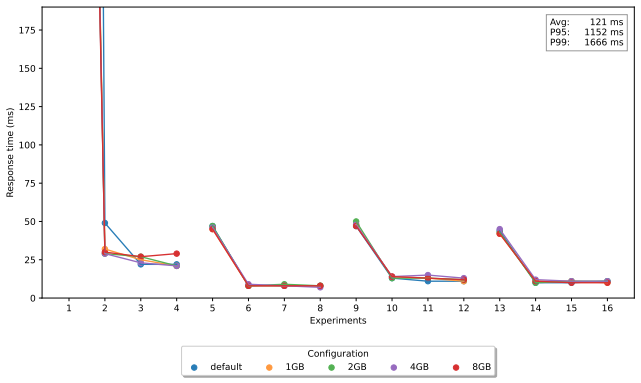


Figure 13: Machine 1 - Neo4j response time (complete graphic here)

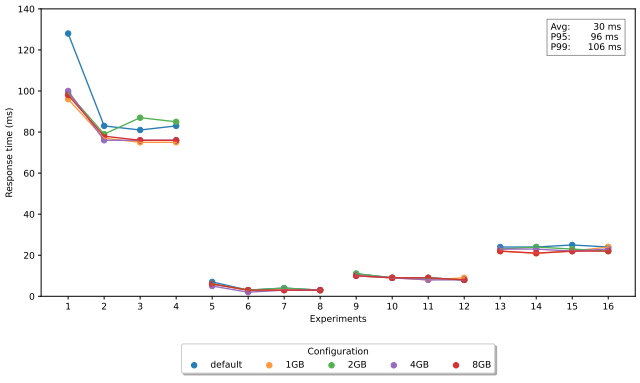


Figure 14: Machine 2 - Postgres response time

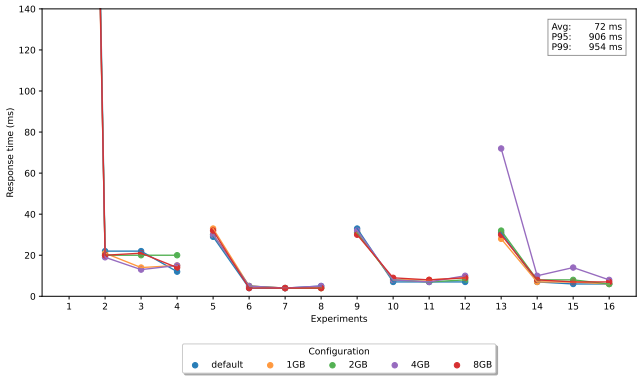


Figure 15: Machine 2 - Neo4j response time (complete graphic here)

Dataset 1 GB

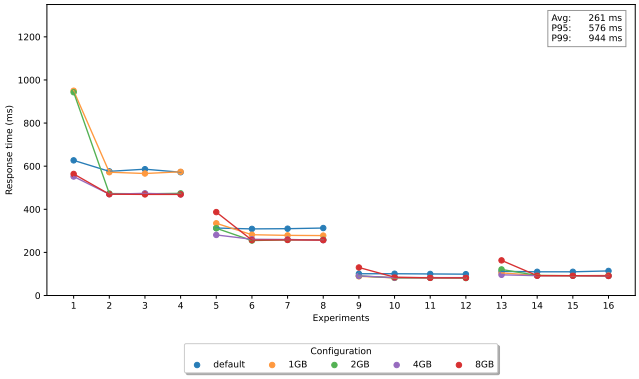


Figure 16: Machine 1 - Postgres response time

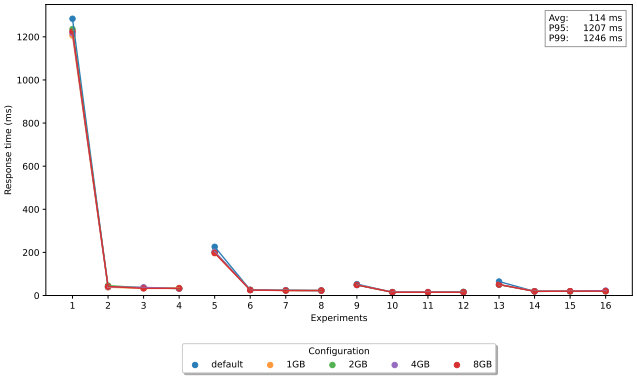


Figure 17: Machine 1 - Neo4j response time

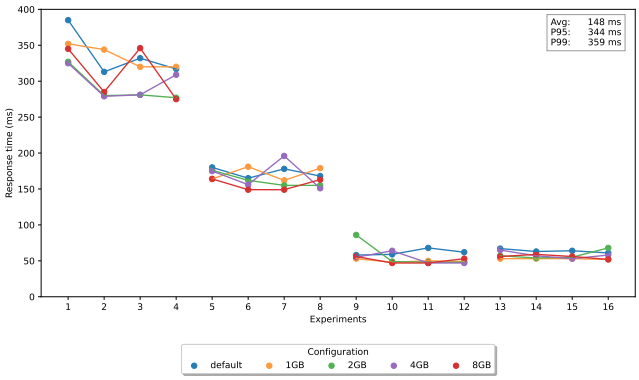


Figure 18: Machine 2 - Postgres response time

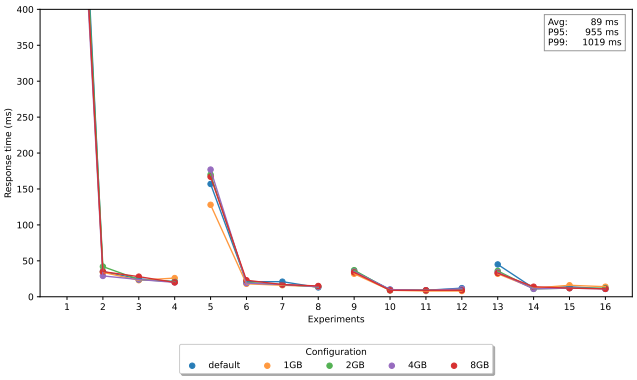


Figure 19: Machine 2 - Neo4j response time (complete graphic here)

Dataset 3 GB

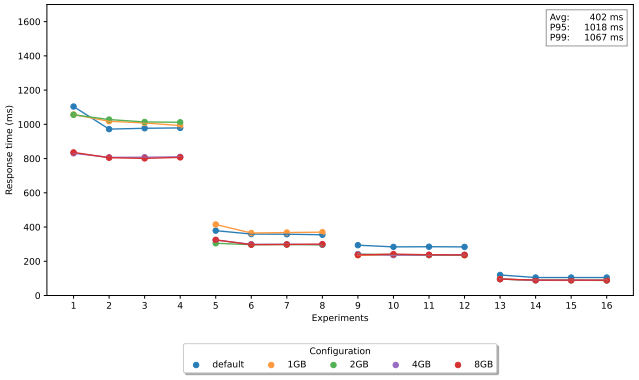


Figure 20: Machine 1 - Postgres response time

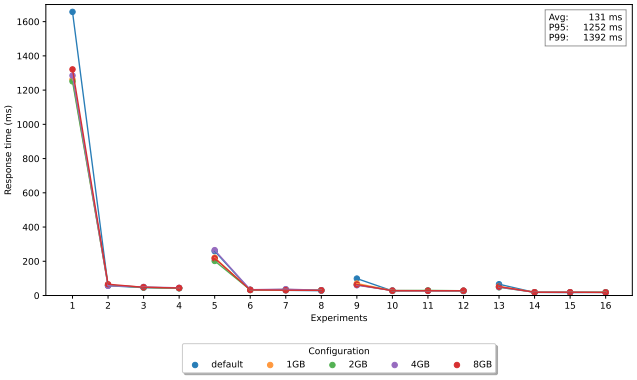


Figure 21: Machine 1 - Neo4j response time

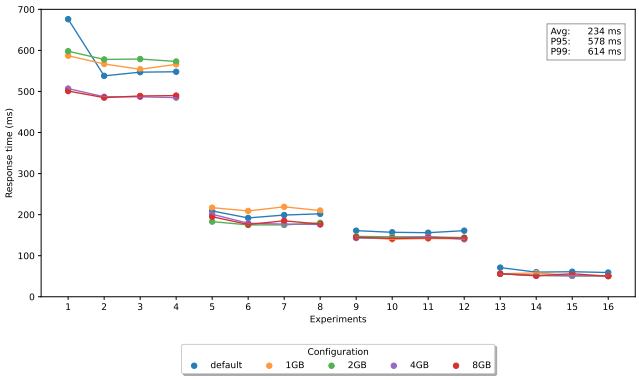


Figure 22: Machine 2 - Postgres response time

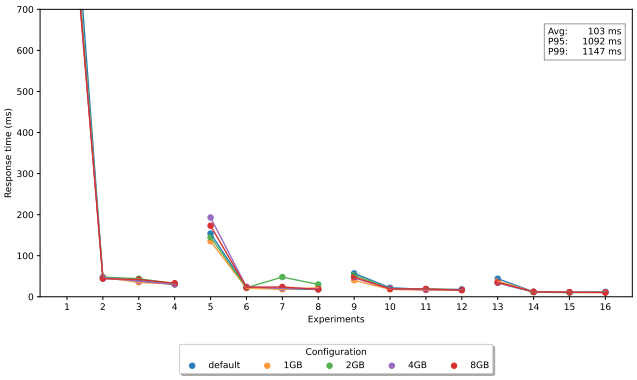


Figure 23: Machine 2 - Neo4j response time (complete graphic here)

This query reveals some noteworthy insights, particularly regarding how the two DBMS handle query caching and execution. With an empty cache, Neo4j takes significantly longer to execute the same query compared to PostgreSQL. However, once the first query is executed, Neo4j outperforms PostgreSQL by a considerable margin. While both operate within a millisecond range, Neo4j's performance advantage becomes more pronounced as the dataset size increases.

5.2.2 Query 2 - Transitive friends with a certain name (max step = 4)

This is the same query as query 1 but with a maximum step of 4 instead of 3.

Dataset 0.3 GB

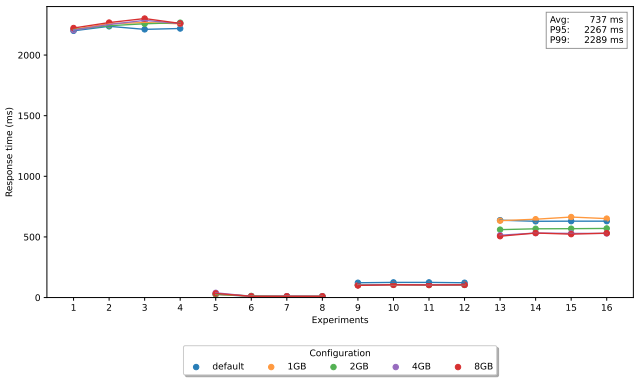


Figure 24: Machine 1 - Postgres response time

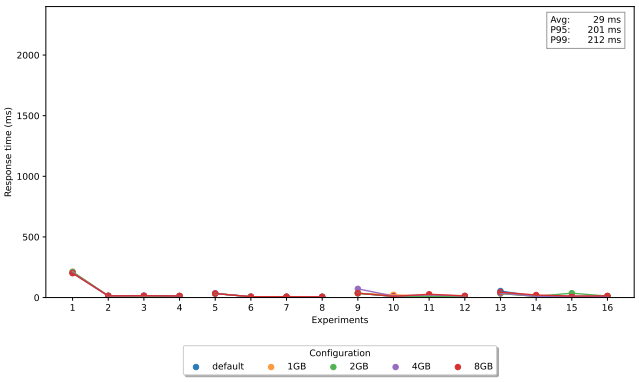


Figure 25: Machine 1 - Neo4j response time

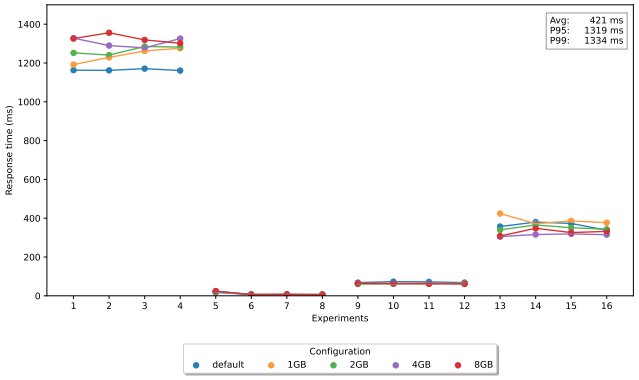


Figure 26: Machine 2 - Postgres response time

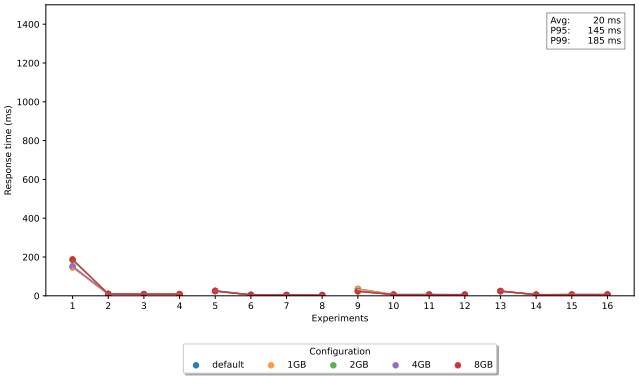


Figure 27: Machine 2 - Neo4j response time

Dataset 1 GB

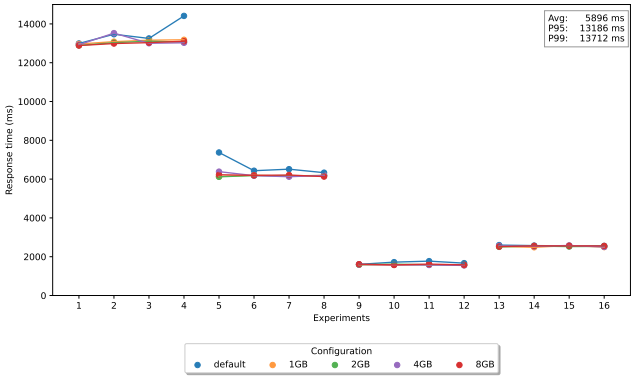


Figure 28: Machine 1 - Postgres response time

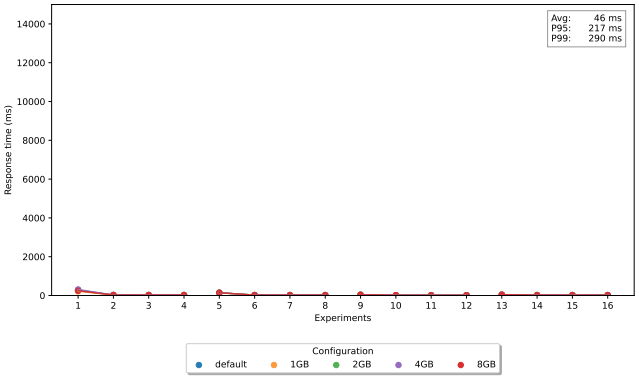


Figure 29: Machine 1 - Neo4j response time

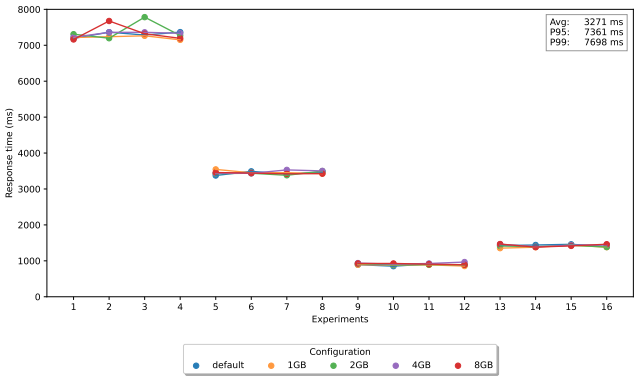


Figure 30: Machine 2 - Postgres response time

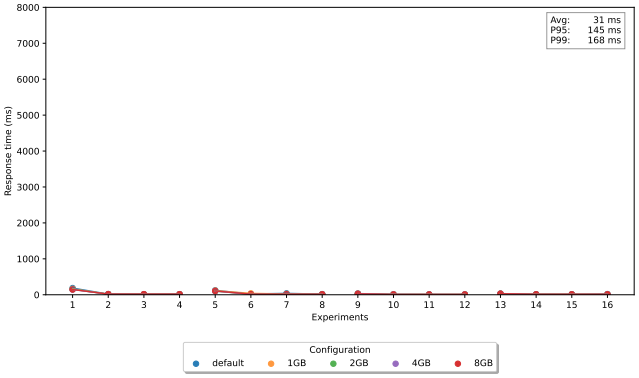


Figure 31: Machine 2 - Neo4j response time

Dataset 3 GB

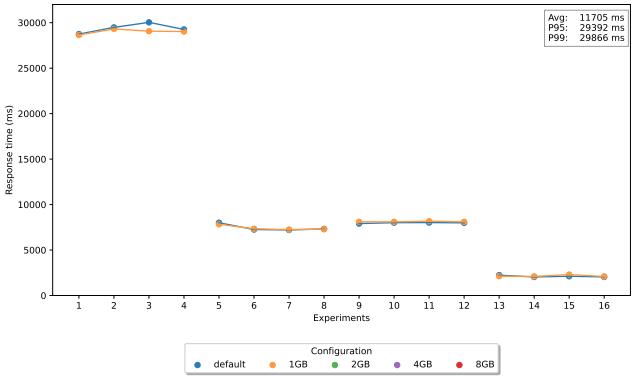


Figure 32: Machine 1 - Postgres response time

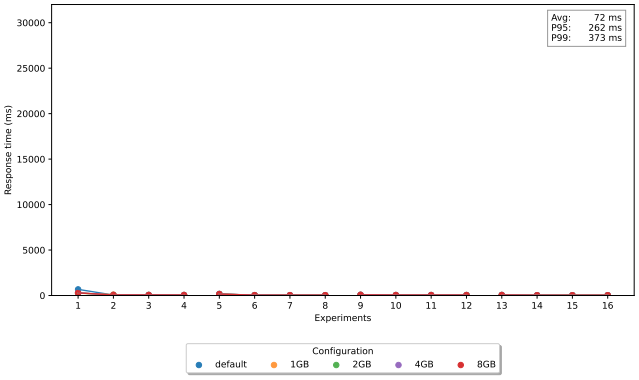


Figure 33: Machine 1 - Neo4j response time

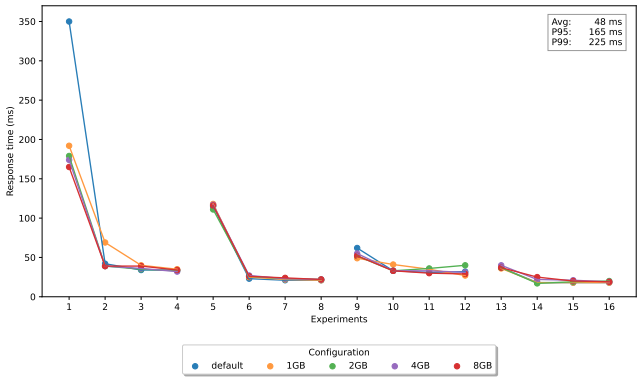


Figure 34: Machine 2 - Neo4j response time

Here, the differences between the two systems become much more apparent. For this query, we used the same inputs as in Query 1 to enable a direct comparison. Notably, Neo4j significantly benefits from previously executed queries, far more effectively than PostgreSQL. Even with a small 300 MB dataset, PostgreSQL required a few seconds to execute the first input set (a consistent pattern across all configurations). In contrast, Neo4j consistently outperformed PostgreSQL across all datasets, regardless of the configuration. In the worst cases, Neo4j's execution times were in the range of a few hundred milliseconds, while in the best cases, they were as low as 4–5 milliseconds. This demonstrates that Neo4j scales exceptionally well with increasing dataset sizes, whereas PostgreSQL struggles to handle recursive queries efficiently as the dataset size grows.

For the 3 GB dataset, we were unable to complete the PostgreSQL benchmark, as each query took an excessive amount of time, significantly delaying our benchmark automation process. This issue is further illustrated in the next query's results.

5.2.3 Query 3 - Transitive friends with a certain name (max step = 5)

This is the same query as query 2 but with a maximum step of 5 instead of 4.

Dataset 0.3 GB

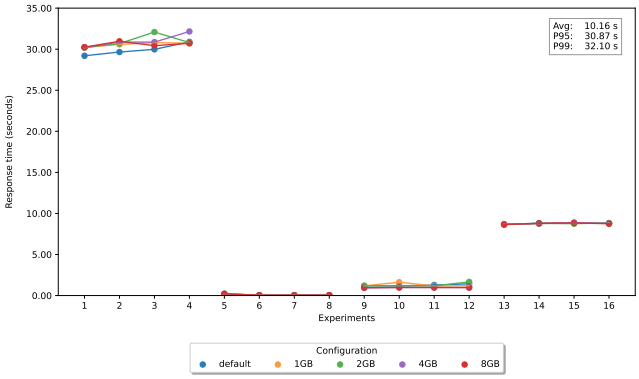


Figure 35: Machine 1 - Postgres response time

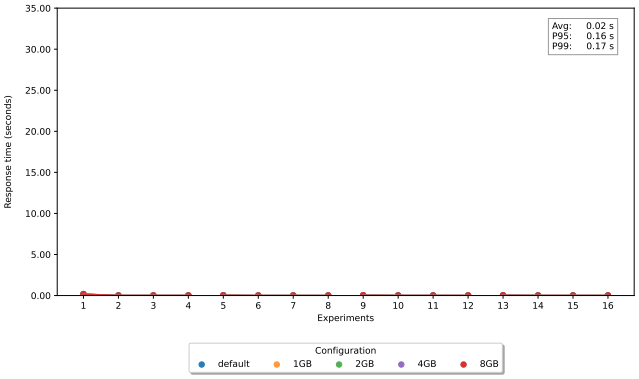


Figure 36: Machine 1 - Neo4j response time

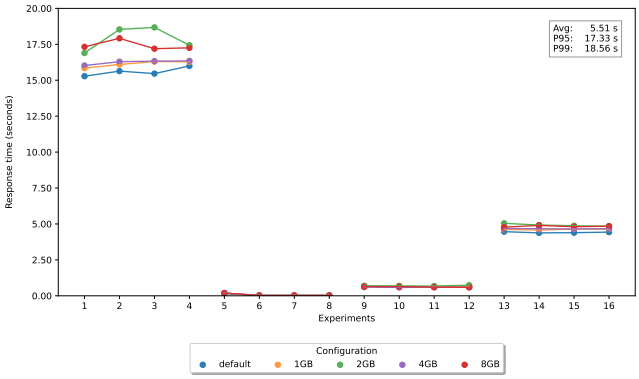


Figure 37: Machine 2 - Postgres response time

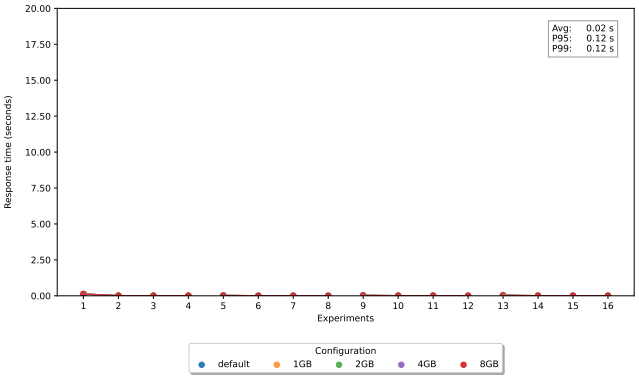


Figure 38: Machine 2 - Neo4j response time

Dataset 1 GB

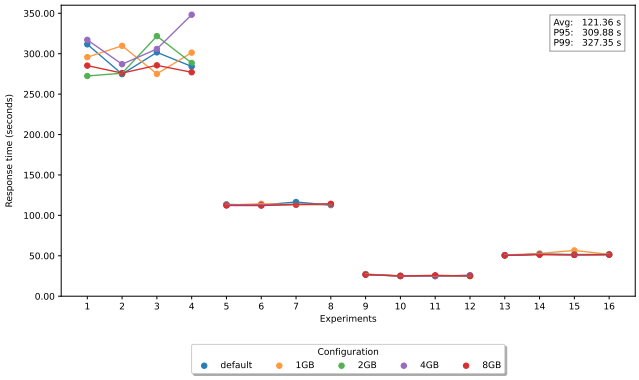


Figure 39: Machine 1 - Postgres response time

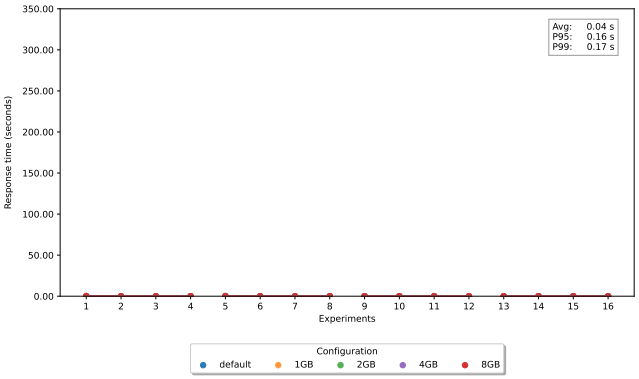


Figure 40: Machine 1 - Neo4j response time

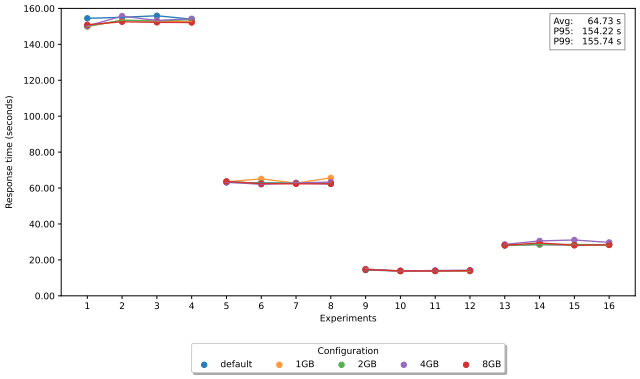


Figure 41: Machine 2 - Postgres response time

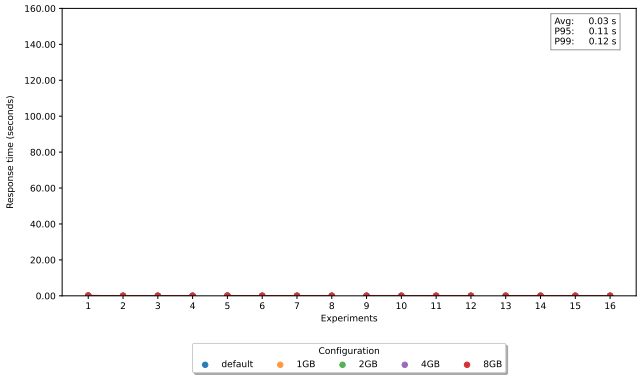


Figure 42: Machine 2 - Neo4j response time

Dataset 3 GB

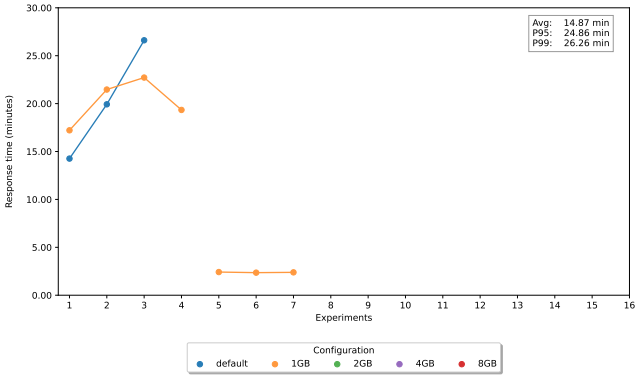


Figure 43: Machine 1 - Postgres response time

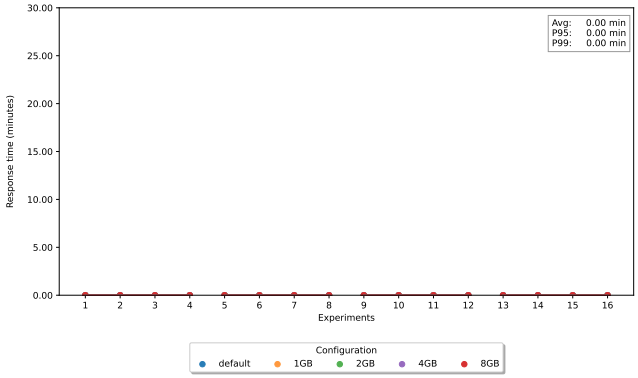


Figure 44: Machine 1 - Neo4j response time

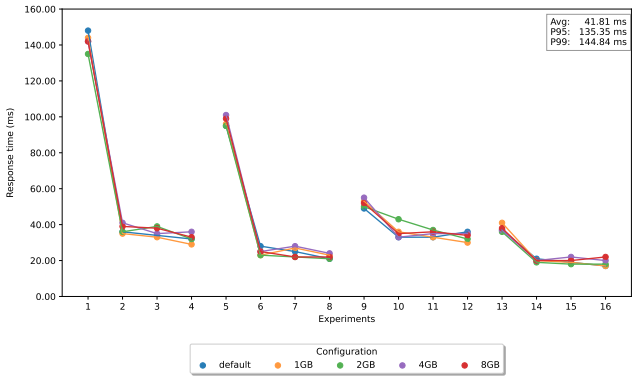


Figure 45: Machine 2 - Postgres response time

We're not going to elaborate much here... Please try to not use recursion as it kills your DBMS performance if your in the relational realm. For a query like this that involves path traversing, a graph database really shines in comparison with a realtional database.

Note: Figure 44 does not display true zero values across each iteration; rather, the recorded times were so minimal that they fell below the measurement threshold, leading to their representation as zero.

5.2.4 Query 4 - Friends and friends of friends that have been to given countries

Given a start Person with ID $personId, find Persons that are their friends and friends of friends (excluding the start Person) that have made Posts / Comments in both of the given Countries (named $countryXName and $countryYName), within [$startDate, $startDate + $durationDays). Only Persons that are foreign to these Countries are considered, that is Persons whose location Country is neither named $countryXName nor $countryYName.

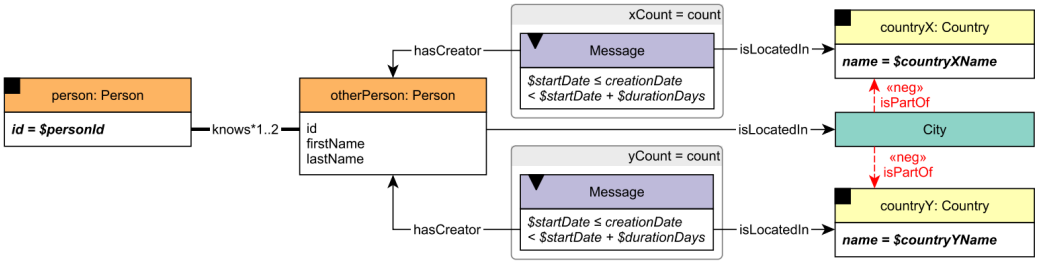


Figure 46: Visualization of the query
[1]

Dataset 0.3 GB

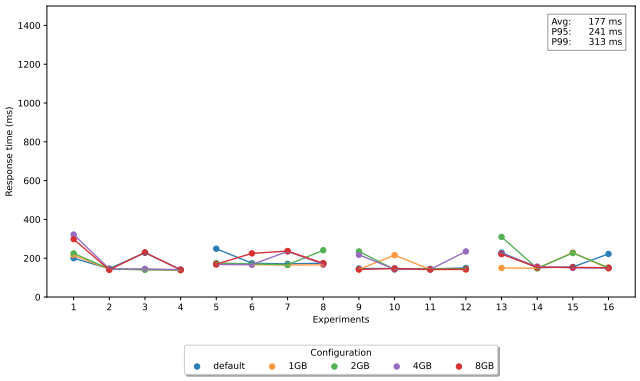


Figure 47: Machine 1 - Postgres response time

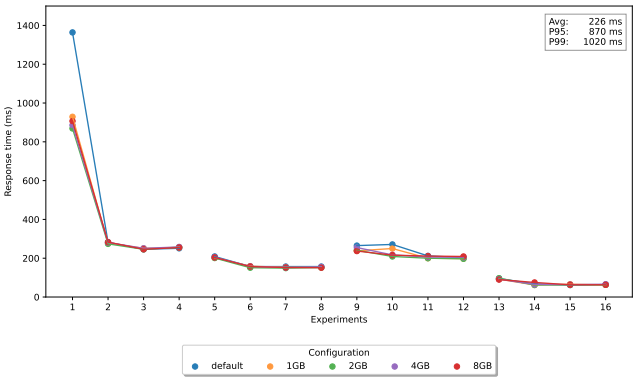


Figure 48: Machine 1 - Neo4j response time

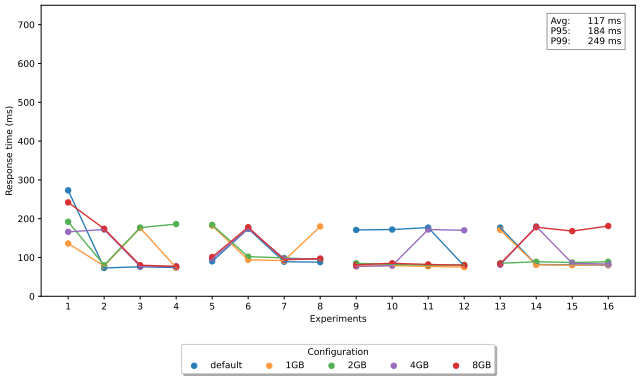


Figure 49: Machine 2 - Postgres response time

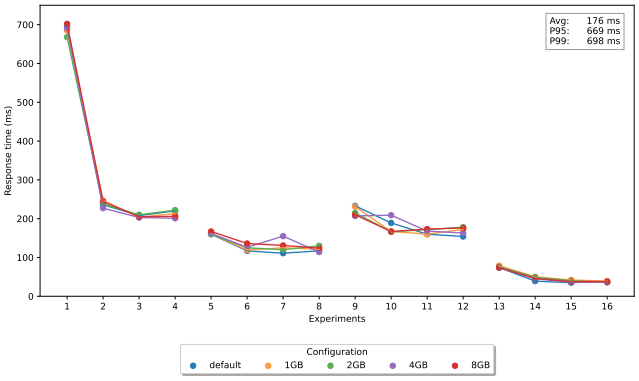


Figure 50: Machine 2 - Neo4j response time

Dataset 1 GB

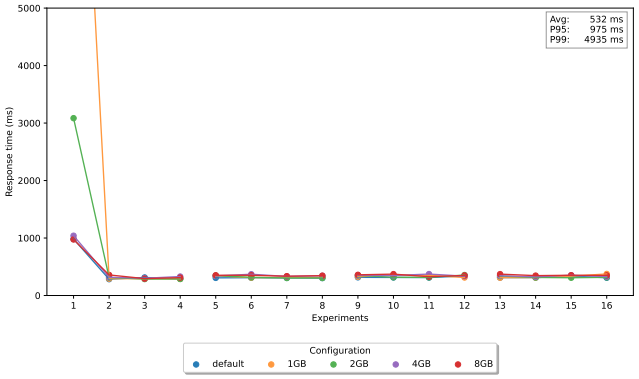


Figure 51: Machine 1 - Postgres response time (complete graphic here)

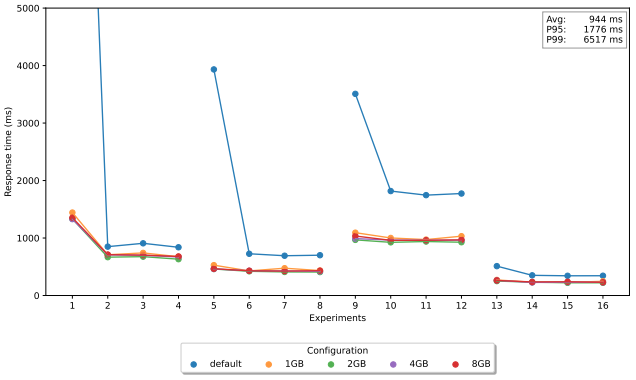


Figure 52: Machine 1 - Neo4j response time (complete graphic here)

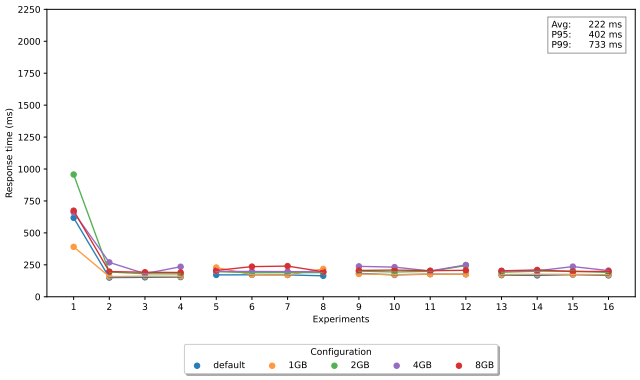


Figure 53: Machine 2 - Postgres response time

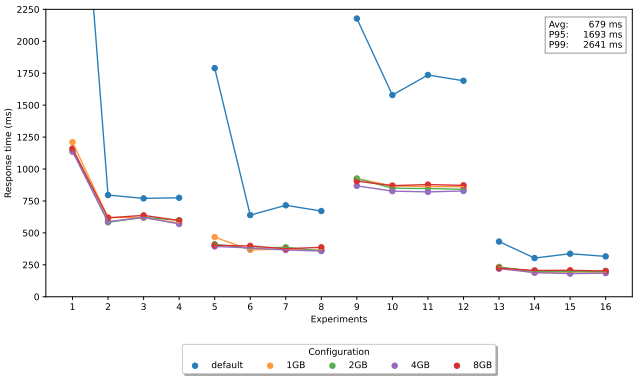


Figure 54: Machine 2 - Neo4j response time (complete graphic here)

Dataset 3 GB

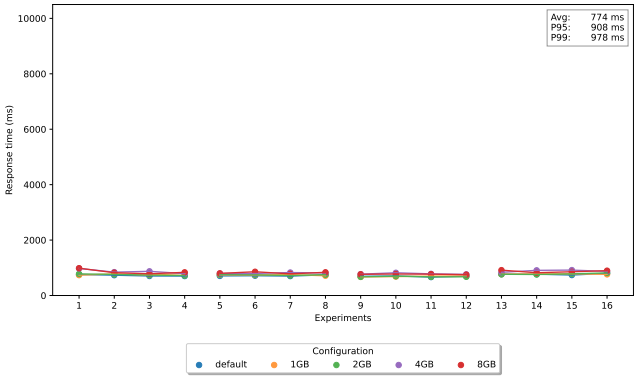


Figure 55: Machine 1 - Postgres response time

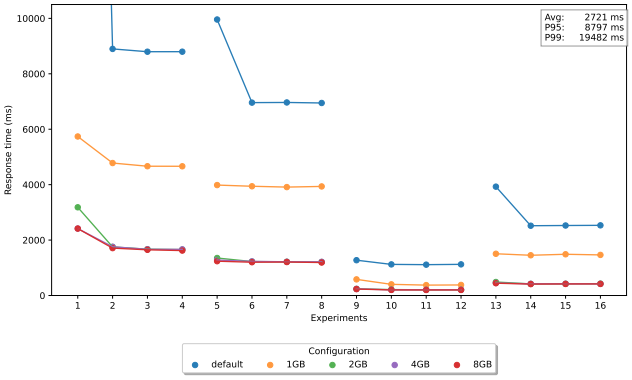


Figure 56: Machine 1 - Neo4j response time (complete graphic here)

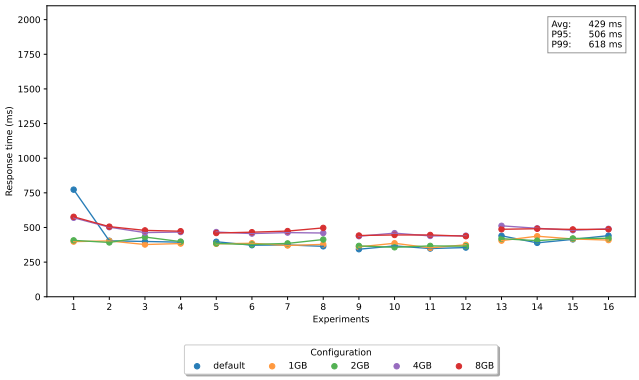


Figure 57: Machine 2 - Postgres response time

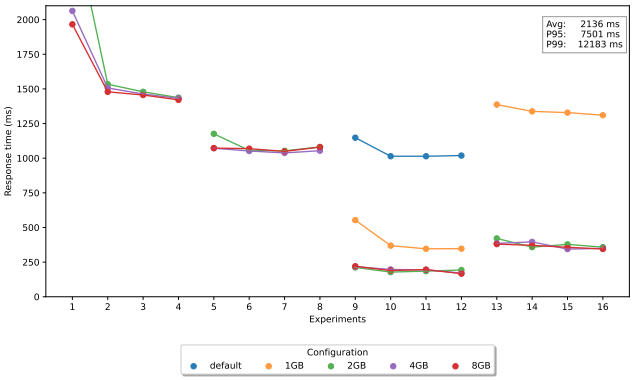


Figure 58: Machine 2 - Neo4j response time (complete graphic here)

This query reveals several interesting conclusions. PostgreSQL consistently outperformed Neo4j across all machines and dataset sizes. Notably, this is the first query where PostgreSQL demonstrates superior performance as the dataset size increases. Each metric (average, P95, and P99) shows better results with PostgreSQL, and its performance is highly consistent, remaining within a narrow range of values—excluding a few outliers in the 1 GB dataset.

Another key observation is that, as the dataset scales up, Neo4j's performance can improve significantly with optimized configurations, potentially surpassing PostgreSQL under similar conditions.

5.2.5 Query 5 - Shortest path

Given two Persons with IDs $person1Id and $person2Id, find the shortest path between these two Persons in the subgraph induced by the knows edges.

To execute this query in PostgreSQL, we utilized the PostGIS extension, which includes a sub-extension called pg_routing. This sub-extension provides various graph algorithms, such as Dijkstra and A*. For this query, we used the Dijkstra algorithm.

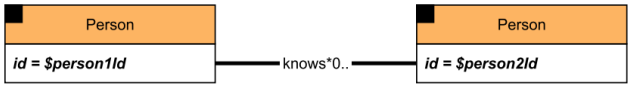


Figure 59: Visualization of the query
[1]

Dataset 0.3 GB

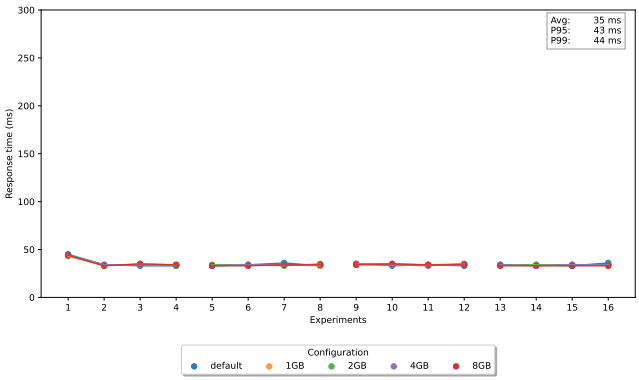


Figure 60: Machine 1 - Postgres response time

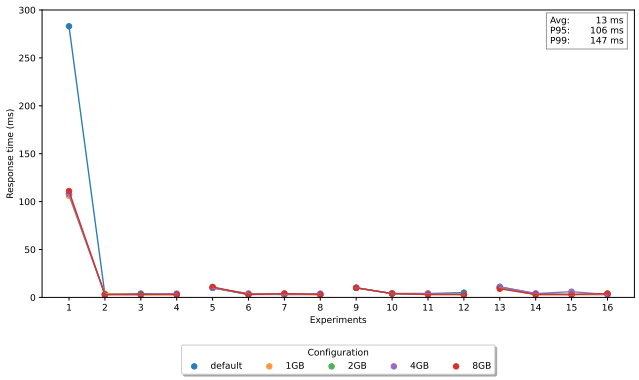


Figure 61: Machine 1 - Neo4j response time (complete graphic here)

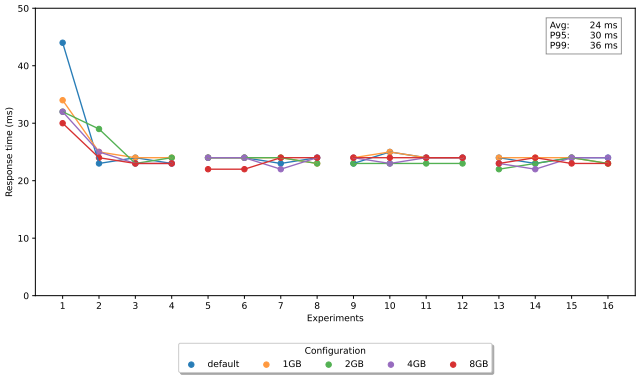


Figure 62: Machine 2 - Postgres response time

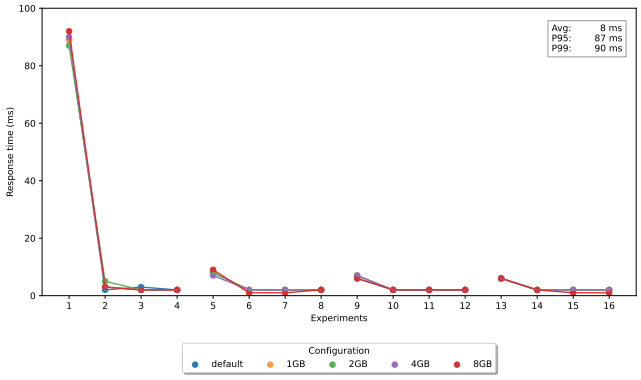


Figure 63: Machine 2 - Neo4j response time

Dataset 1 GB

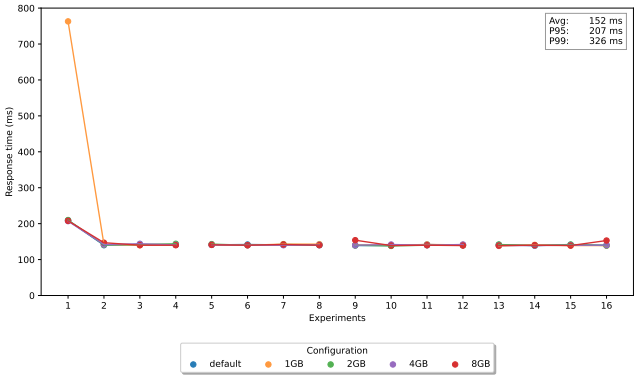


Figure 64: Machine 1 - Postgres response time

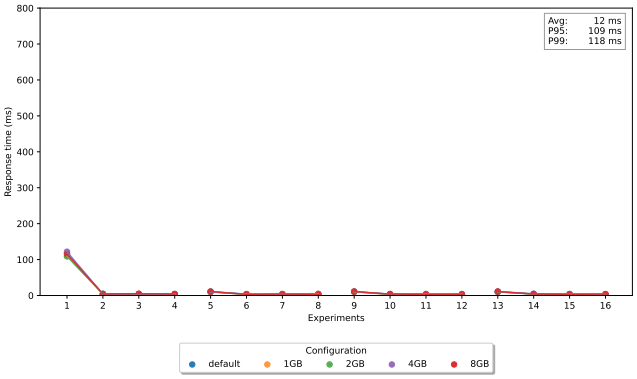


Figure 65: Machine 1 - Neo4j response time

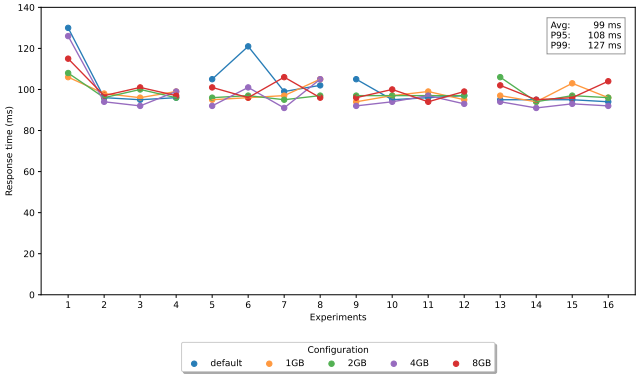


Figure 66: Machine 2 - Postgres response time

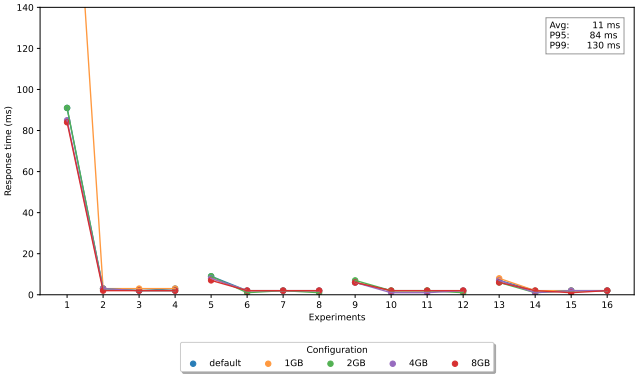


Figure 67: Machine 2 - Neo4j response time (complete graphic here)

Dataset 3 GB

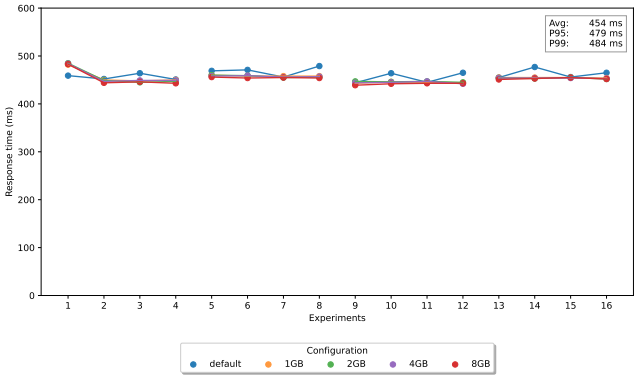


Figure 68: Machine 1 - Postgres response time

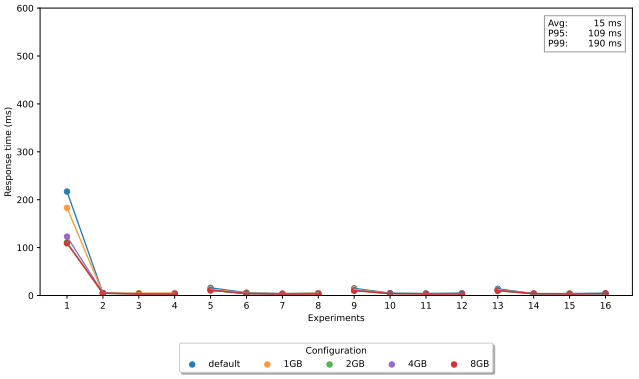


Figure 69: Machine 1 - Neo4j response time

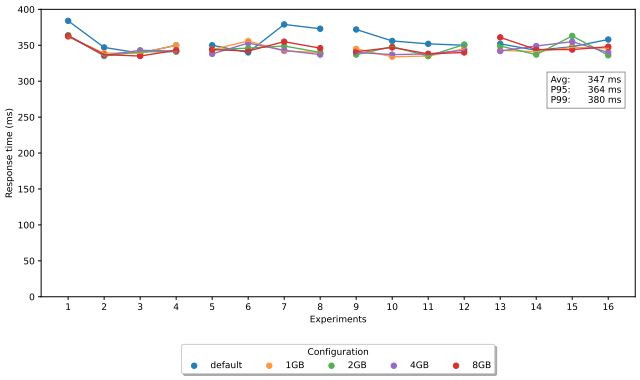


Figure 70: Machine 2 - Postgres response time

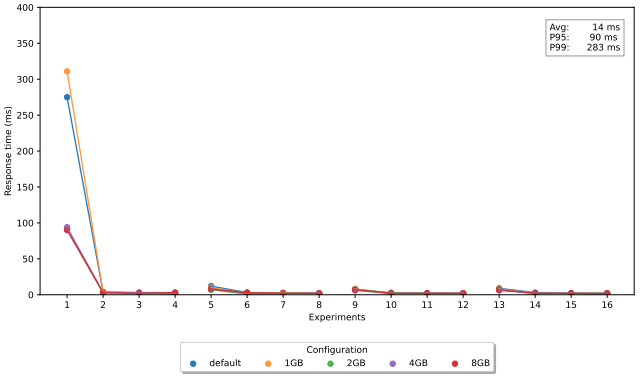


Figure 71: Machine 2 - Neo4j response time

The results show that with a small dataset, PostgreSQL can compete effectively against a native graph database. However, as the dataset size increases, the performance gap becomes more evident in favor of Neo4j. This query also highlights the efficiency of the Neo4j optimizer, which generates highly optimized execution plans after the first query is executed.

5.2.6 Query 6 - k Shortest Paths

Given two Persons with IDs $person1Id and $person2Id, find the k shortest paths between these two Persons in the subgraph induced by the knows edges. The k we used was 3.

The pg_routing extension provides a built-in algorithm to directly calculate the K shortest paths, whereas Neo4j does not offer this functionality natively. In Neo4j, the shortest path function only returns multiple paths if they have the same cost (or distance). To retrieve the 3 shortest paths, we leveraged Neo4j's Graph Data Science (GDS) library. For this query, we had to create an in-memory graph specifically for the operation. The graph creation step, which took a few hundred milliseconds, was excluded from the benchmarks to avoid inflating the response times.

Dataset 0.3 GB

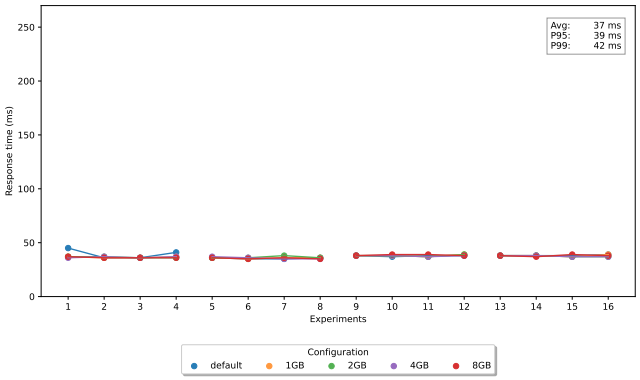


Figure 72: Machine 1 - Postgres response time

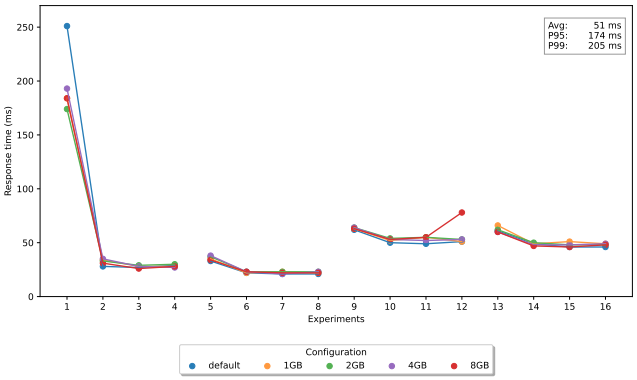


Figure 73: Machine 1 - Neo4j response time

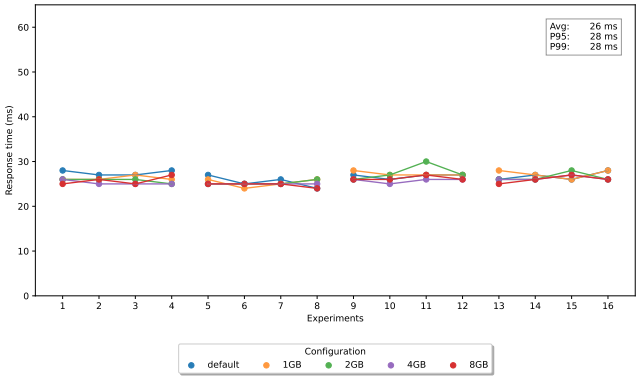


Figure 74: Machine 2 - Postgres response time

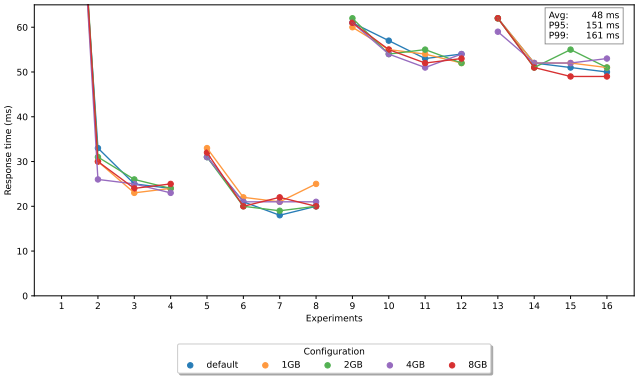


Figure 75: Machine 2 - Neo4j response time (complete graphic here)

Dataset 1 GB

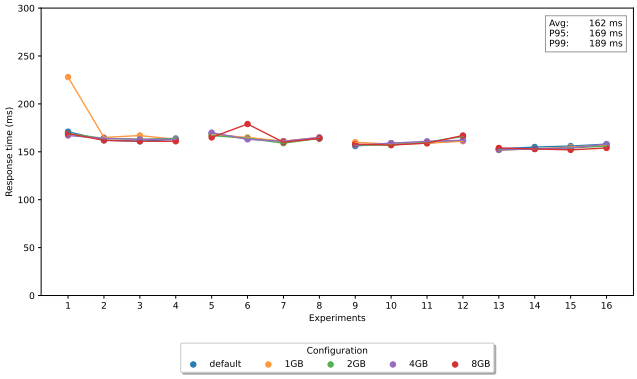


Figure 76: Machine 1 - Postgres response time

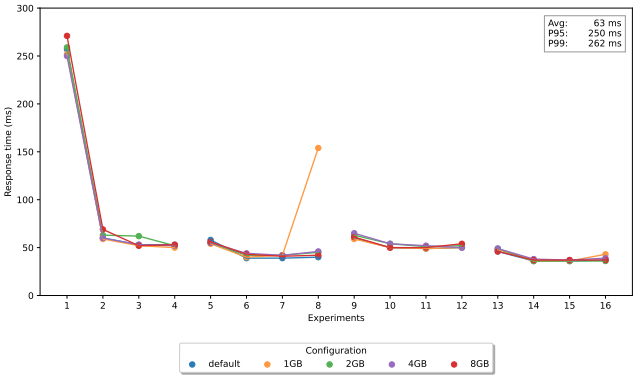


Figure 77: Machine 1 - Neo4j response time

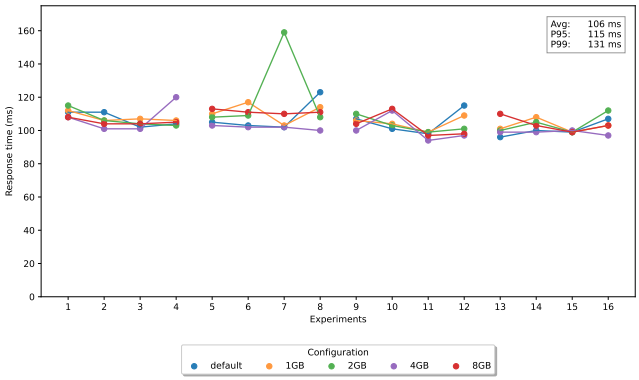


Figure 78: Machine 2 - Postgres response time

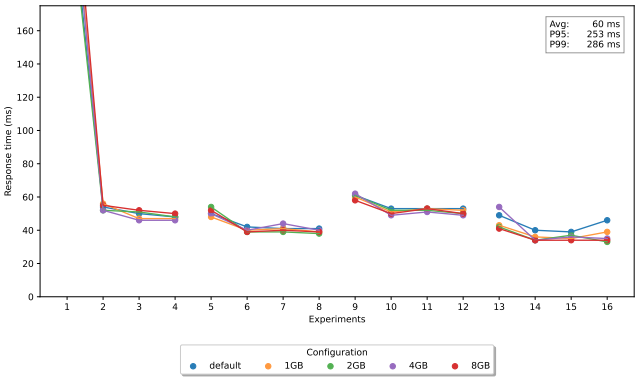


Figure 79: Machine 2 - Neo4j response time (complete graphic here)

Dataset 3 GB

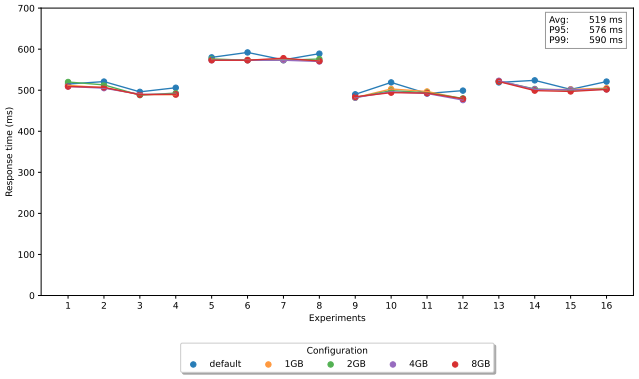


Figure 80: Machine 1 - Postgres response time

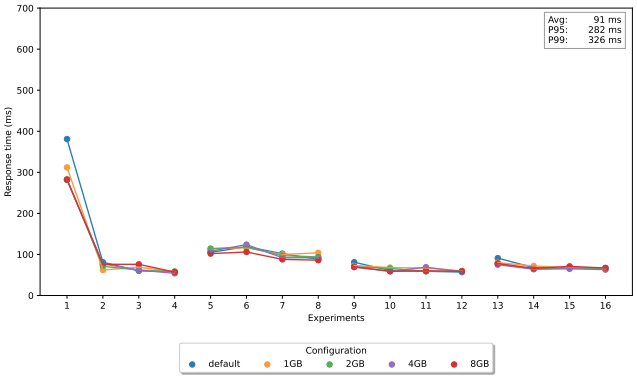


Figure 81: Machine 1 - Neo4j response time

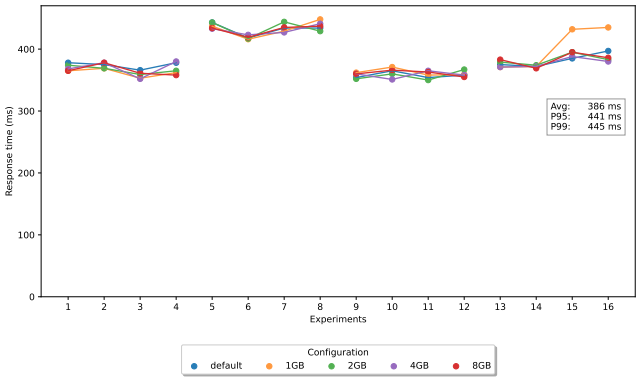


Figure 82: Machine 2 - Postgres response time

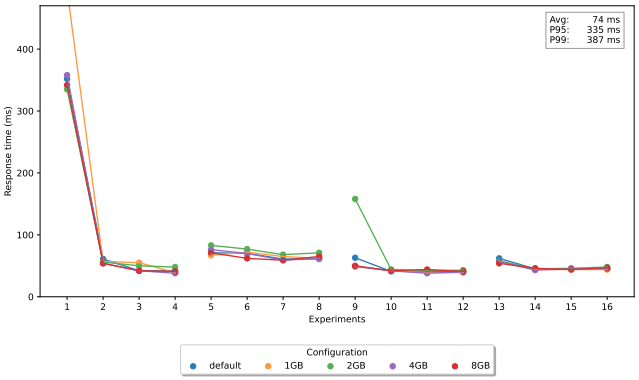


Figure 83: Machine 2 - Neo4j response time (complete graphic here)

Here, we observe PostgreSQL's consistency, maintaining response times within a narrow range. However, Neo4j once again takes the lead as the dataset size increases, demonstrating its scalability regardless of the configuration. Notably, Neo4j achieves better response times even with fewer resources, outperforming a heavily optimized PostgreSQL setup, or as we might call it, "PostgreSQL on steroids." (well relatively weak steroids but our pc's don't allow much more)

5.2.7 Query 7 - Forum of a Message

Given a Message with ID $messageId, retrieve the Forum that contains it and the Person that moderates that Forum. Since Comments are not directly contained in Forums, for Comments, return the Forum containing the original Post in the thread which the Comment is replying to.

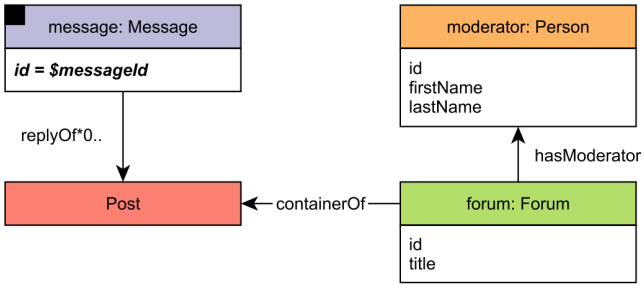


Figure 84: Visualization of the query
[1]

Dataset 0.3 GB

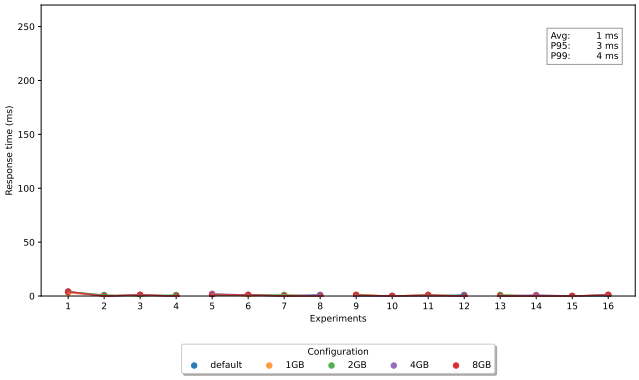


Figure 85: Machine 1 - Postgres response time

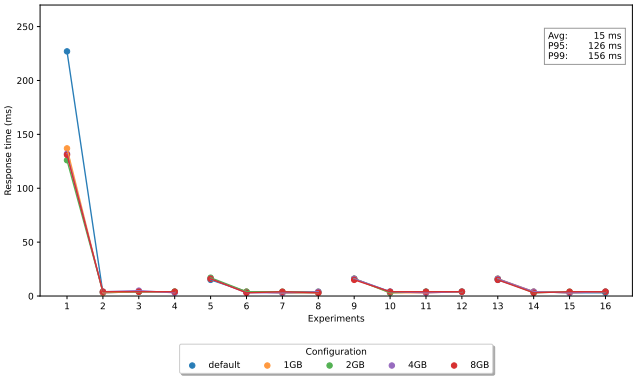


Figure 86: Machine 1 - Neo4j response time

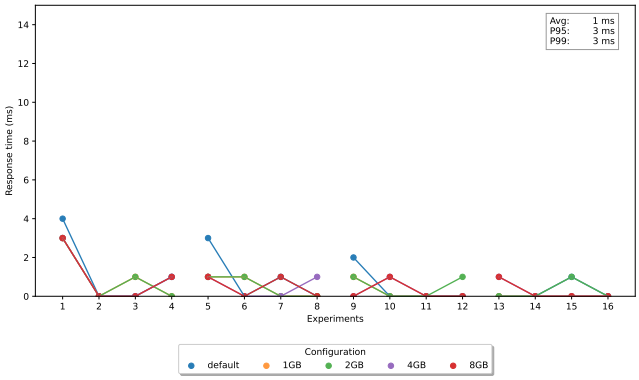


Figure 87: Machine 2 - Postgres response time

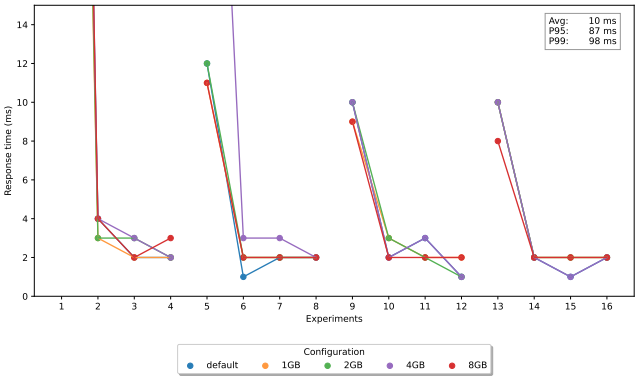


Figure 88: Machine 2 - Neo4j response time (complete graphic here)

Dataset 1 GB

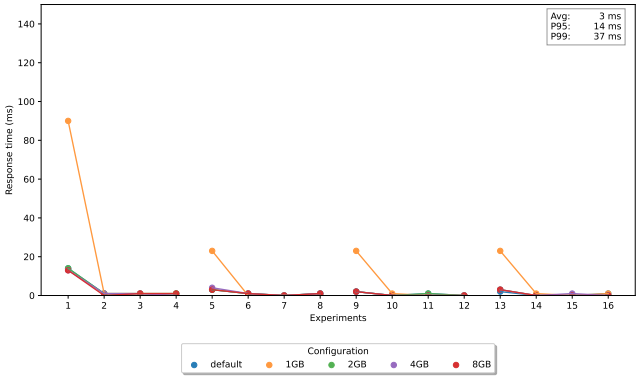


Figure 89: Machine 1 - Postgres response time

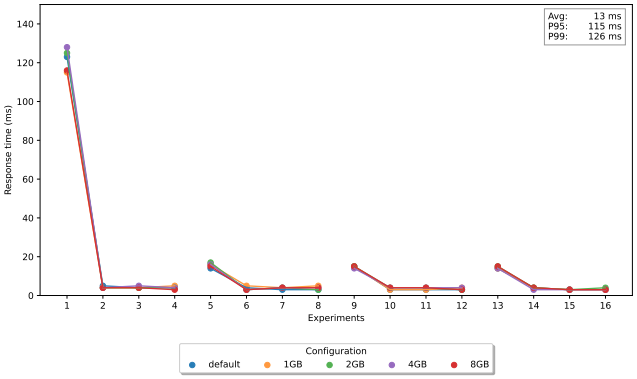


Figure 90: Machine 1 - Neo4j response time

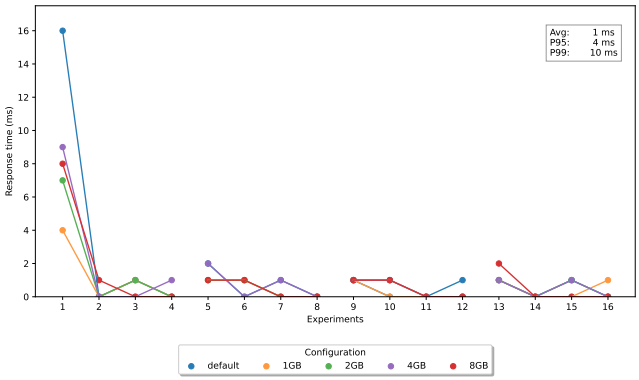


Figure 91: Machine 2 - Postgres response time

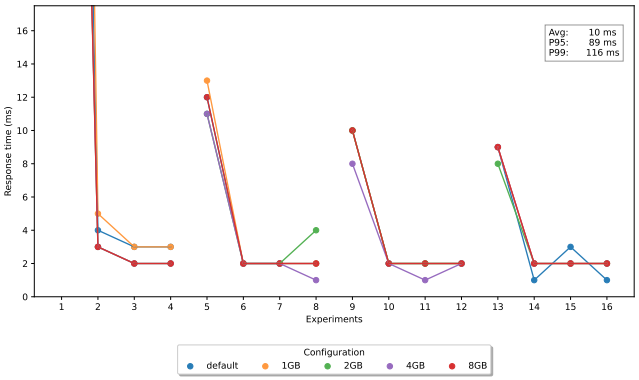


Figure 92: Machine 2 - Neo4j response time (complete graphic here)

Dataset 3 GB

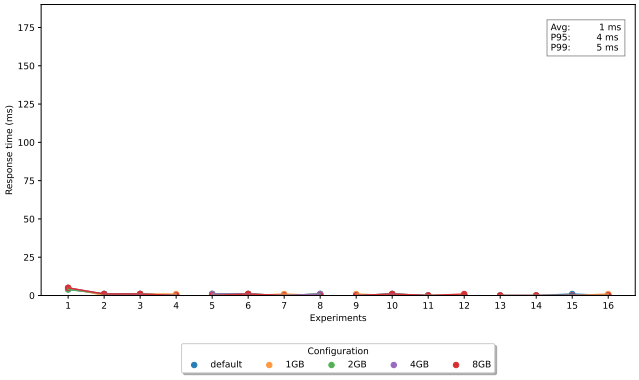


Figure 93: Machine 1 - Postgres response time

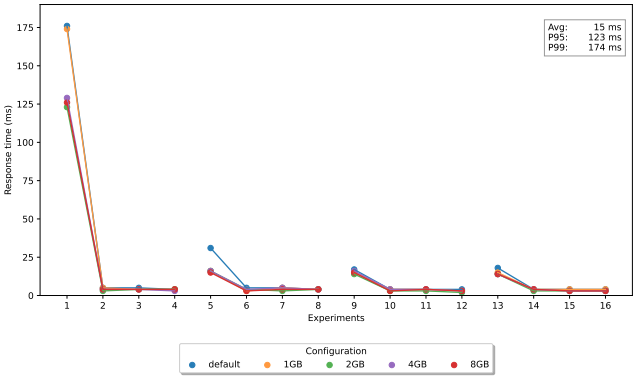


Figure 94: Machine 1 - Neo4j response time

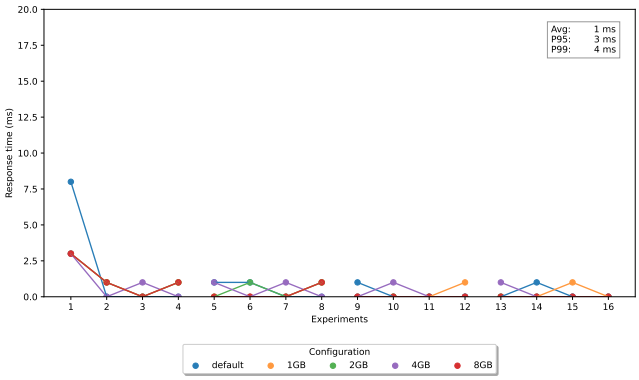


Figure 95: Machine 2 - Postgres response time

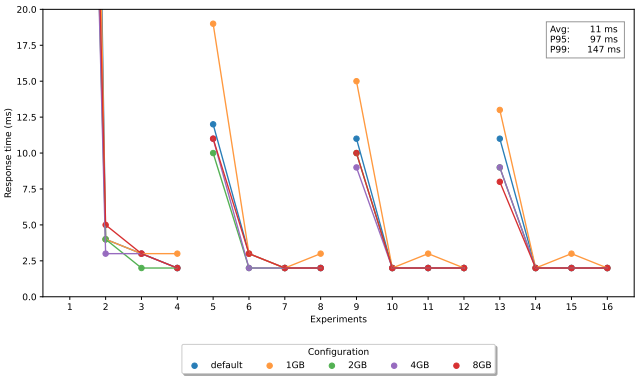


Figure 96: Machine 2 - Neo4j response time (complete graphic here)

Surprisingly, and contrary to expectations, PostgreSQL achieved a substantial performance advantage over Neo4j in a query requiring recursion to find the original post. A possible explanation for this performance gain is that the recursive query required minimal information at each step, as it only needed to "climb the tree" until reaching the root.

5.2.8 Query 8 - Recent Messages by your friends

Given a start Person with ID $personId, find the most recent Messages from all of that Person's friends (friend nodes). Only consider Messages created before the given $maxDate (excluding that day).

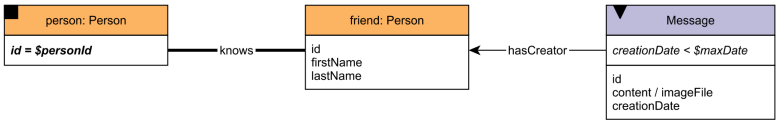


Figure 97: Visualization of the query
[1]

Dataset 0.3 GB

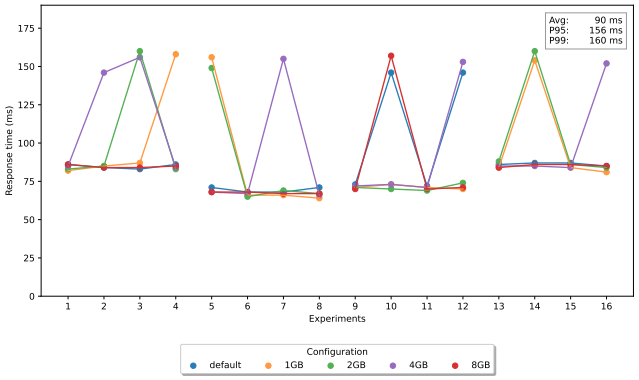


Figure 98: Machine 1 - Postgres response time

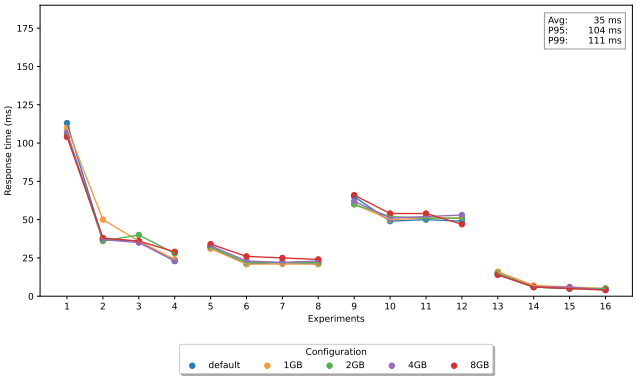


Figure 99: Machine 1 - Neo4j response time

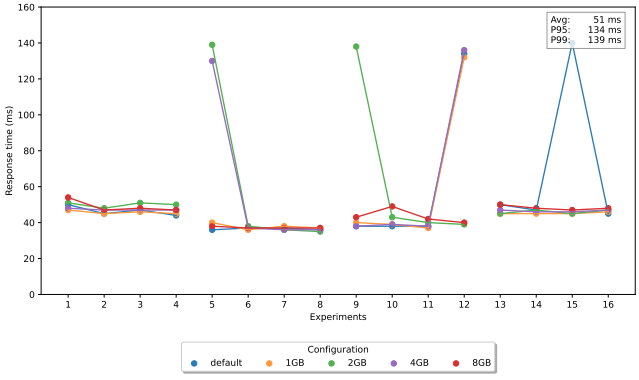


Figure 100: Machine 2 - Postgres response time

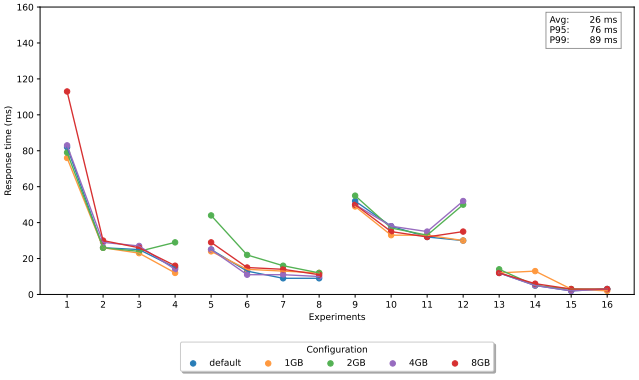


Figure 101: Machine 2 - Neo4j response time

Dataset 1 GB

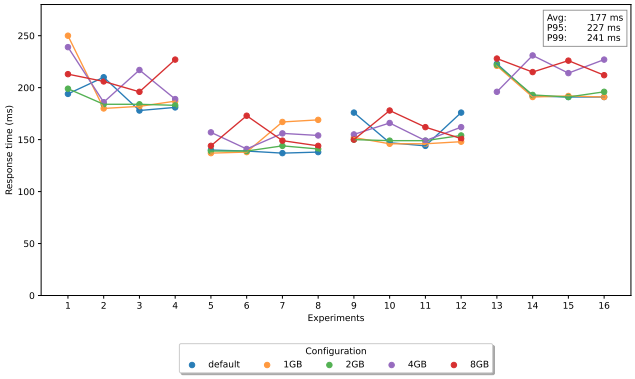


Figure 102: Machine 1 - Postgres response time

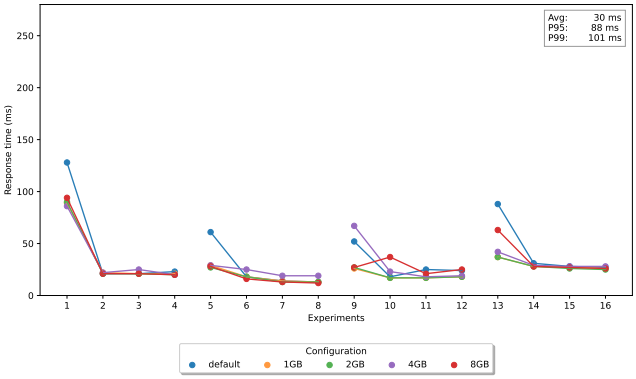


Figure 103: Machine 1 - Neo4j response time

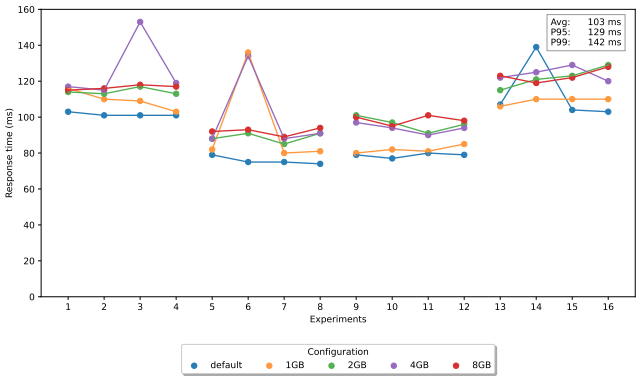


Figure 104: Machine 2 - Postgres response time

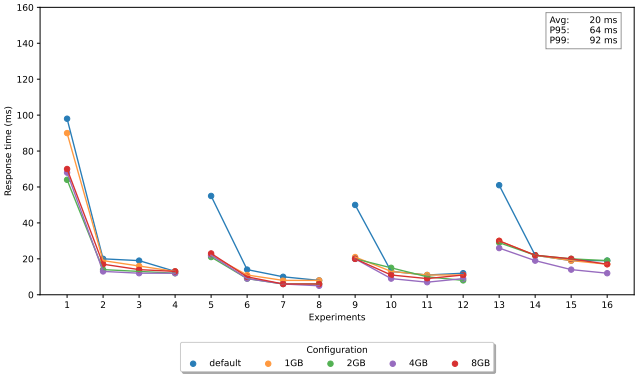


Figure 105: Machine 2 - Neo4j response time

Dataset 3 GB

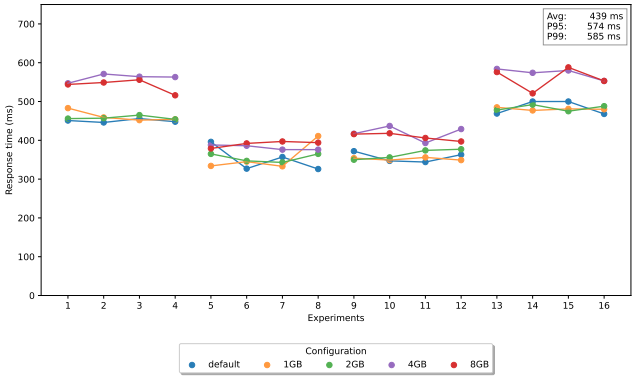


Figure 106: Machine 1 - Postgres response time

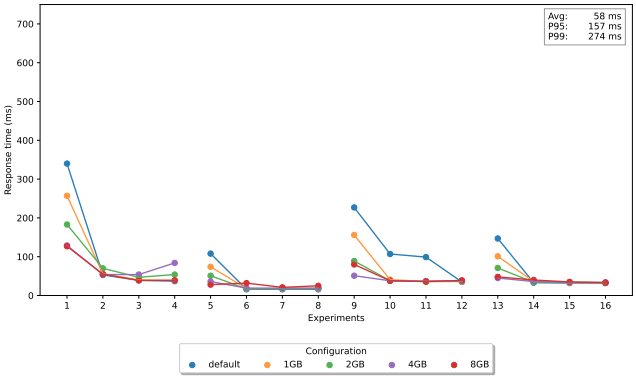


Figure 107: Machine 1 - Neo4j response time

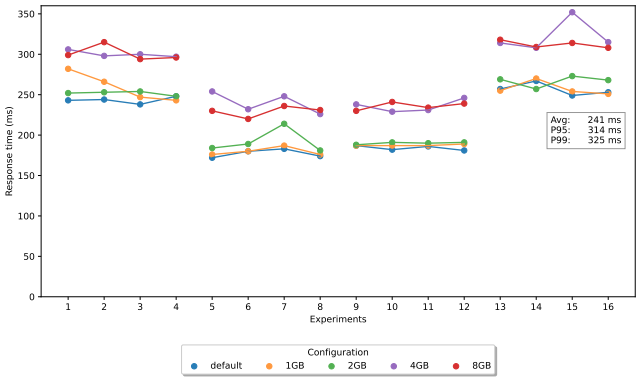


Figure 108: Machine 2 - Postgres response time

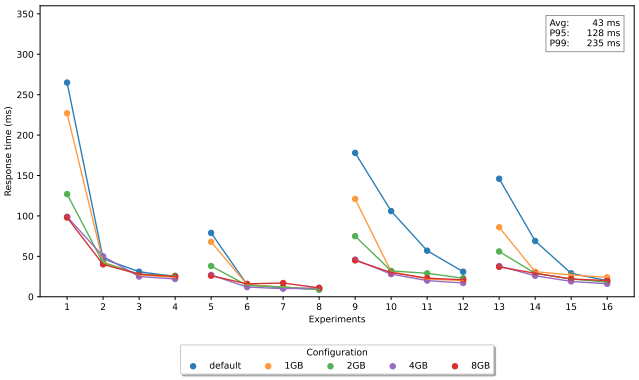


Figure 109: Machine 2 - Neo4j response time

This query was relatively straightforward to implement in both DBMSs, and the results indicate that their performance is quite similar. However, Neo4j once again takes the lead as the dataset size increases.

5.2.9 Query 9 - Friends Recommendation

Given a start Person with ID $personId, find that Person's friends of friends (foaf) – excluding the start Person and his/her immediate friends -, who were born on or after the 21st of a given $month (in any year) and before the 22nd of the following month. Calculate the similarity between each friend and the start person, where the commonInterestScore is defined as commonInterestScore = common - uncommon commonInterestScore=common - uncommon. Here, common is the number of Posts created by the friend that have a Tag the start person is interested in, and uncommon is the number of Posts created by the friend that have no Tag the start person is interested in.

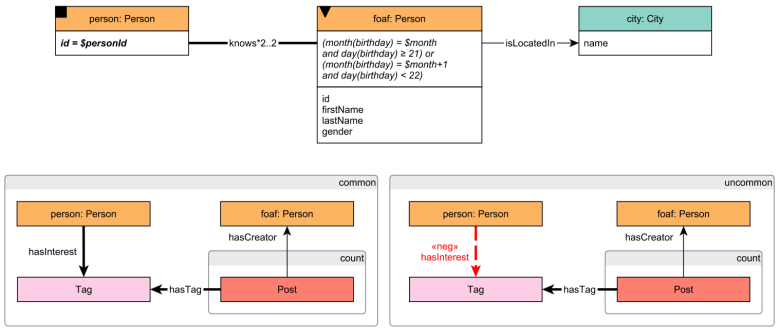


Figure 110: Visualization of the query
[1]

Dataset 0.3 GB

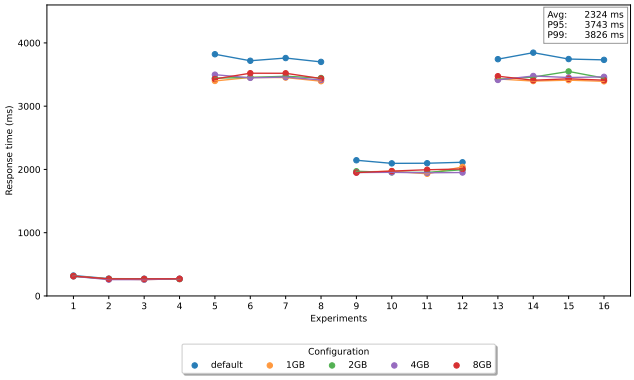


Figure 111: Machine 2 - Postgres response time

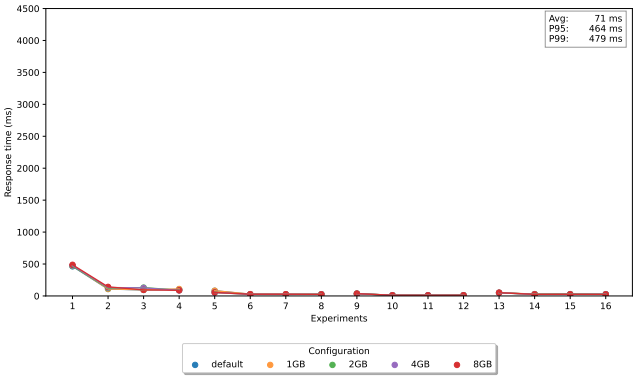


Figure 112: Machine 2 - Neo4j response time

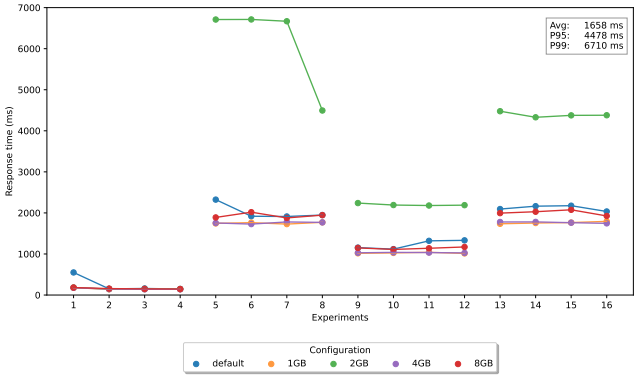


Figure 113: Machine 2 - Postgres response time

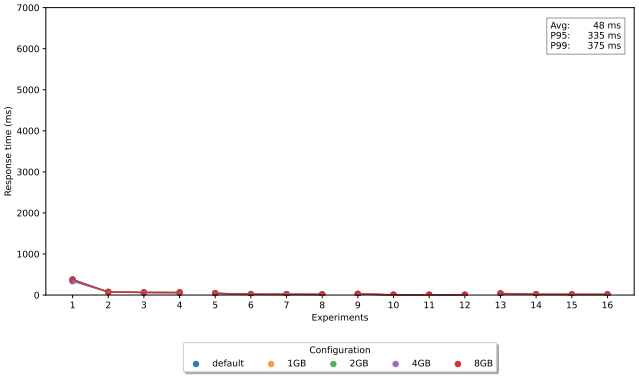


Figure 114: Machine 2 - Neo4j response time

Dataset 1 GB

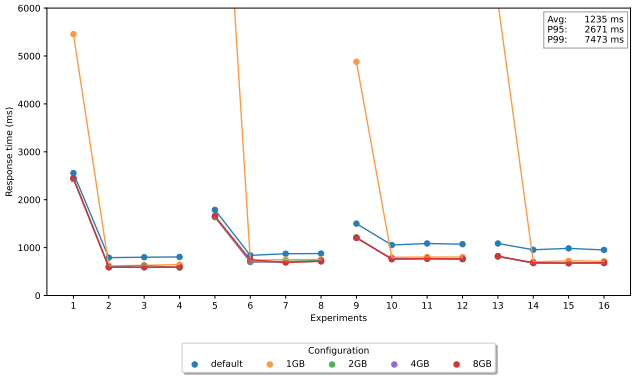


Figure 115: Machine 1 - Postgres response time (complete graphic here)

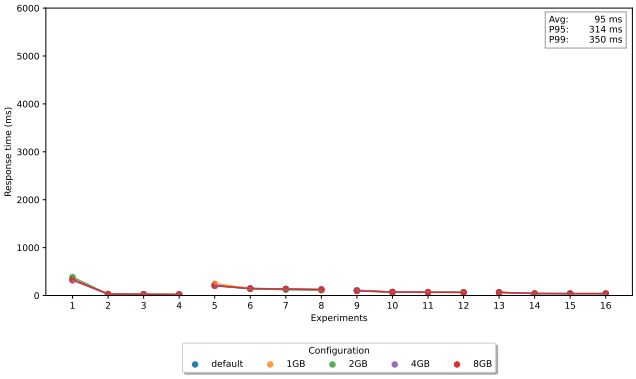


Figure 116: Machine 1 - Neo4j response time

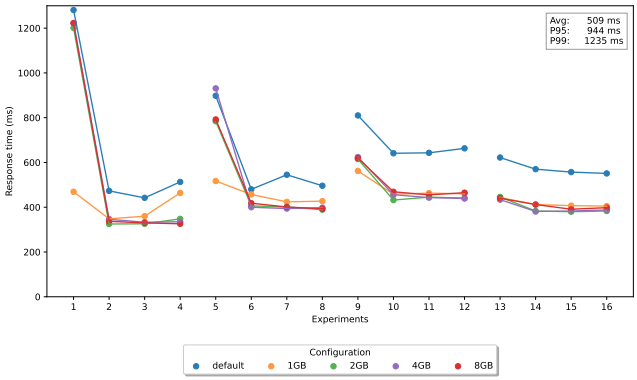


Figure 117: Machine 2 - Postgres response time

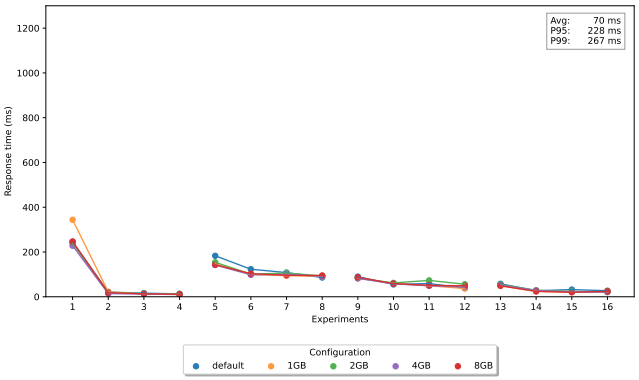


Figure 118: Machine 2 - Neo4j response time

Dataset 3 GB

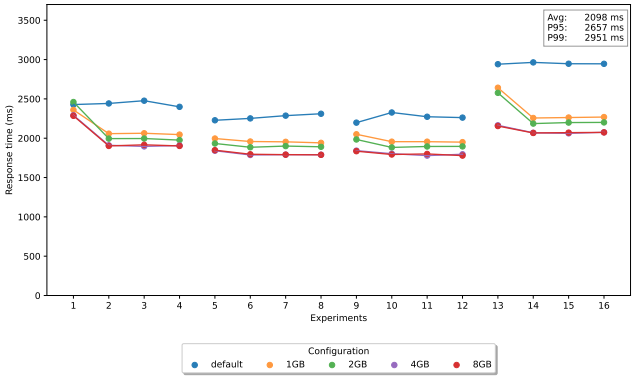


Figure 119: Machine 1 - Postgres response time

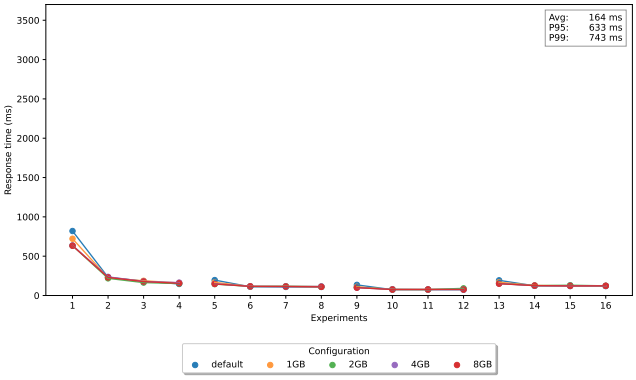


Figure 120: Machine 1 - Neo4j response time

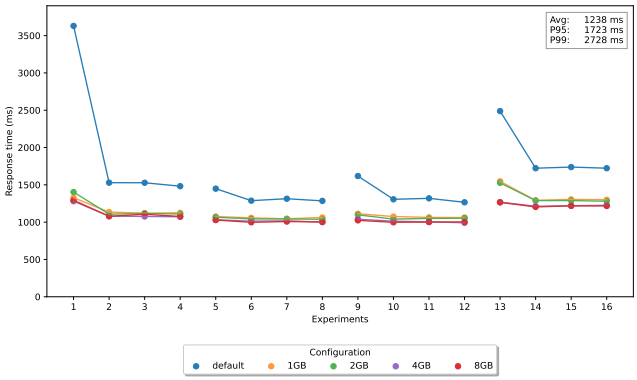


Figure 121: Machine 2 - Postgres response time

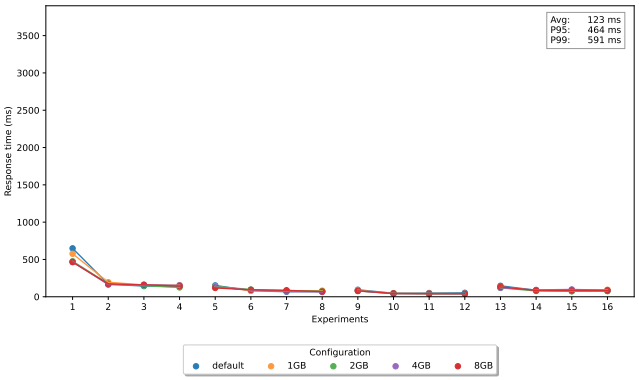


Figure 122: Machine 2 - Neo4j response time

This query stands in stark contrast to the previous one and can be considered the "final boss" of all the queries we did. It was not only challenging to write but also required significant effort in debugging and validating the results to ensure their accuracy.

The results show some fluctuation across the different input sets, which can be attributed to the variety of inputs used. However, within each input set, the performance remains relatively consistent in most cases.

This query involved numerous joins on the SQL side, highlighting PostgreSQL's struggles with performance compared to Neo4j who once again, achive better performance in every experiment. Neo4j's native graph capabilities give it a clear advantage in handling such complex queries.

One thing we didn't mention until now is that Machine 2 is significantly more powerful than Machine 1, which explains the consistently better results across all benchmarks conducted on Machine 2.

5.2.10 Query 10 - Replies of a message

Given a Message with ID $messageId, retrieve the (1-hop) Comments that reply to it. In addition, return a boolean flag knows indicating if the author of the reply (replyAuthor) knows the author of the original message (messageAuthor). If author is same as original author, return False for knows flag.

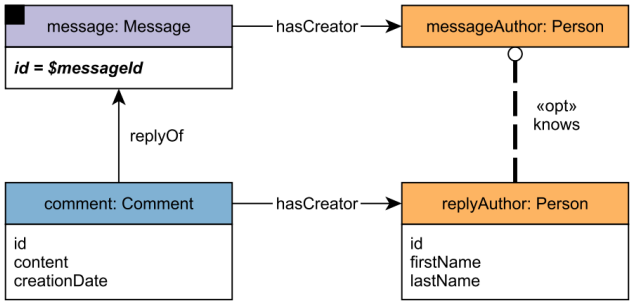


Figure 123: Visualization of the query
[1]

Dataset 0.3 GB

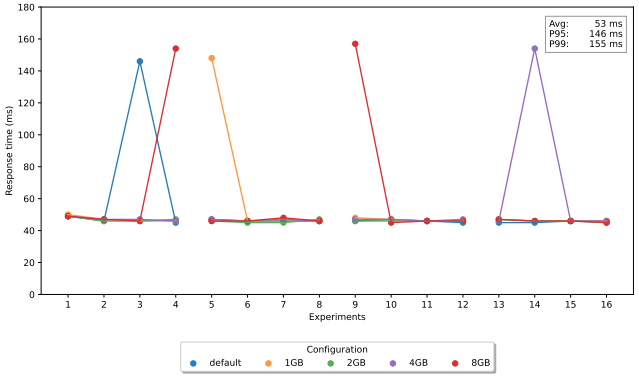


Figure 124: Machine 1 - Postgres response time

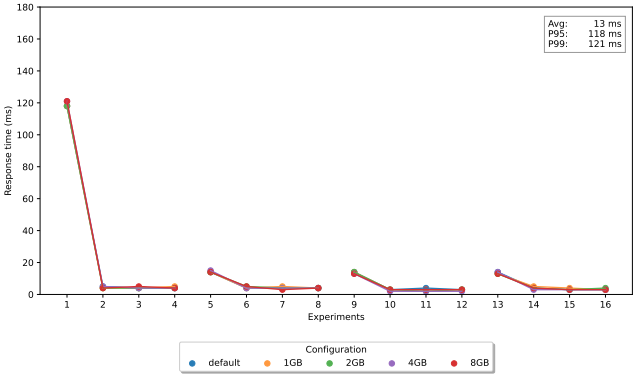


Figure 125: Machine 1 - Neo4j response time

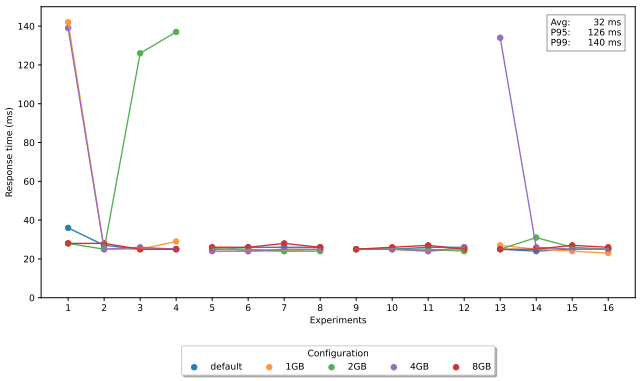


Figure 126: Machine 2 - Postgres response time

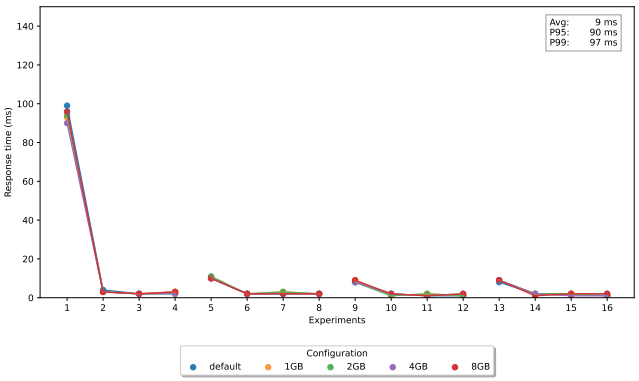


Figure 127: Machine 2 - Neo4j response time

Dataset 1 GB

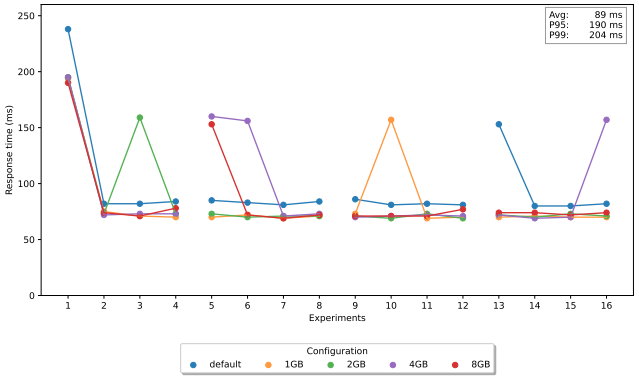


Figure 128: Machine 1 - Postgres response time

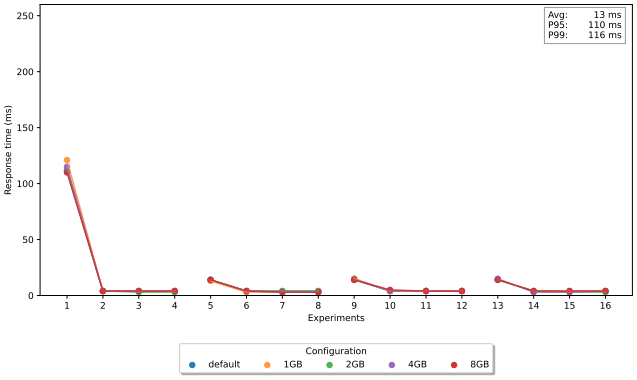


Figure 129: Machine 1 - Neo4j response time

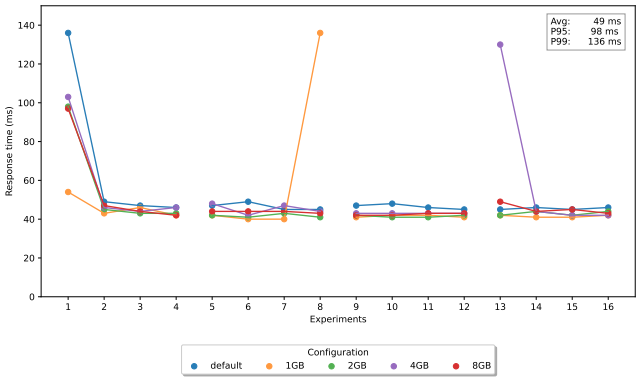


Figure 130: Machine 2 - Postgres response time

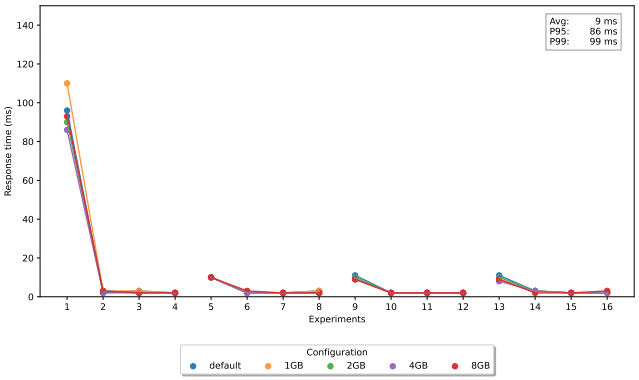


Figure 131: Machine 2 - Neo4j response time

Dataset 3 GB

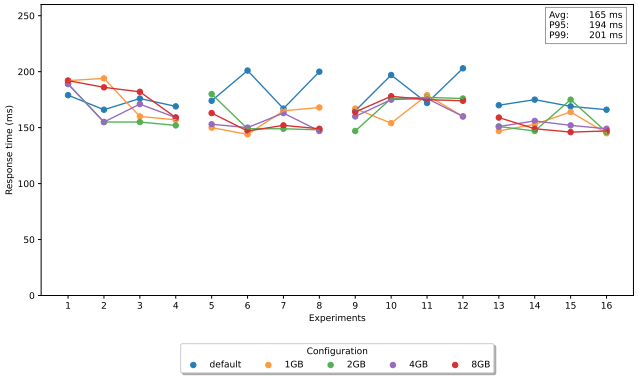


Figure 132: Machine 1 - Postgres response time

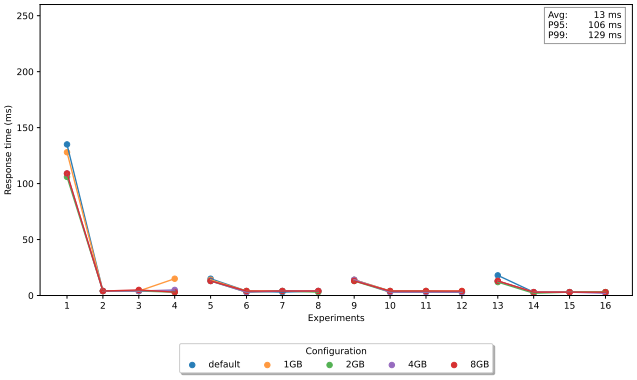


Figure 133: Machine 1 - Neo4j response time

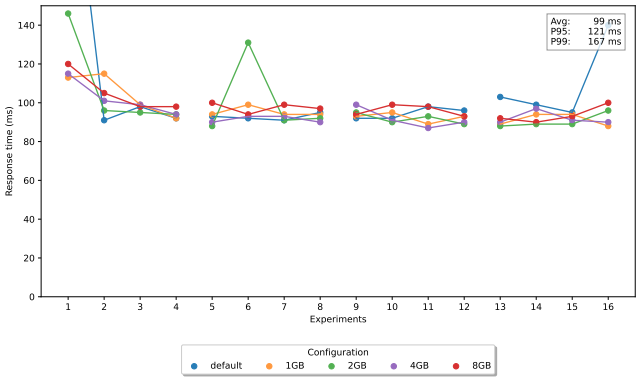


Figure 134: Machine 2 - Postgres response time

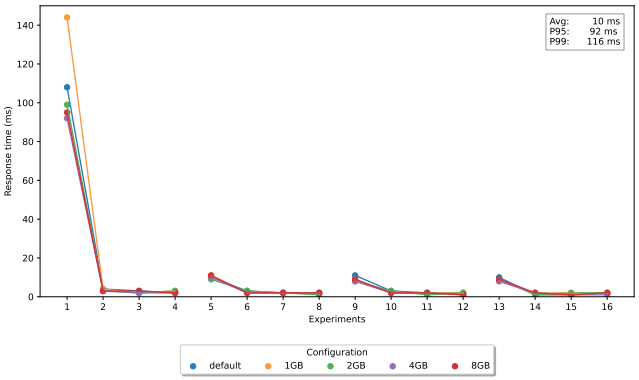


Figure 135: Machine 2 - Neo4j response time

In this query, both DBMSs demonstrated strong performance, with response times in the range of a few hundred milliseconds in the worst cases. However, Neo4j holds the advantage by achieving better overall performance, particularly after the first query of each experiment was executed.

5.2.11 Concurrent Benchmarks

In this benchmark, we executed a mix of queries to load test the DBMS over a short period. Each experiment ran for approximately 12 minutes, during which the number of virtual users was incrementally increased every 2 minutes. The graphs below illustrate the throughput over time, with the primary variable being the number of virtual users.

Dataset 0.3 GB

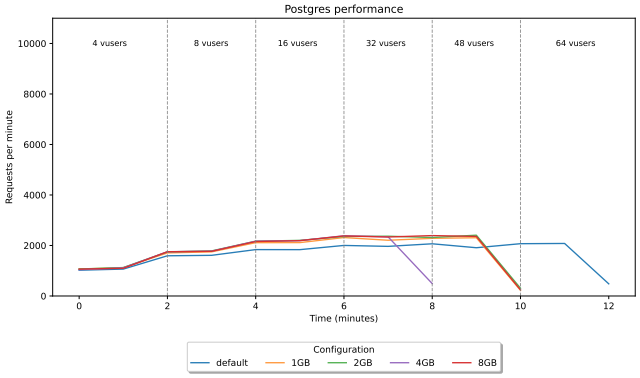


Figure 136: Machine 1 - Postgres throughput

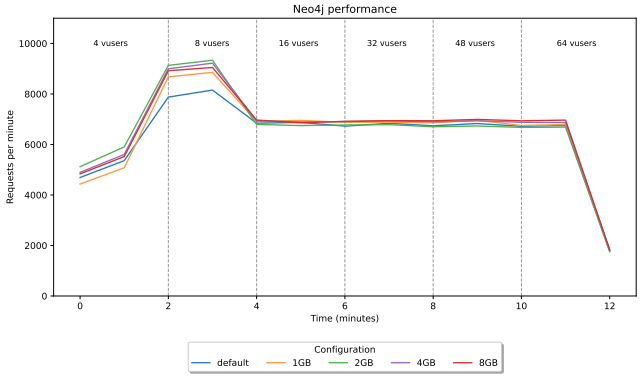


Figure 137: Machine 1 - Neo4j throughput

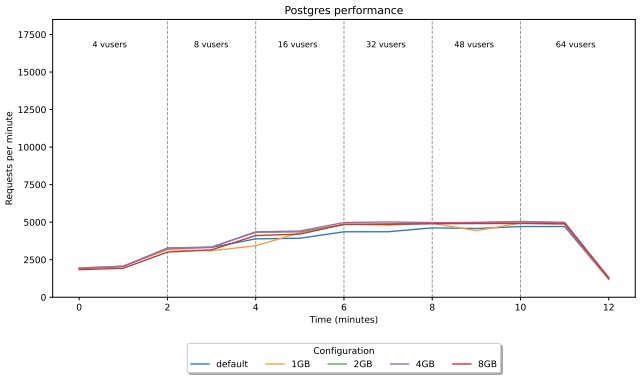


Figure 138: Machine 1 - Postgres throughput

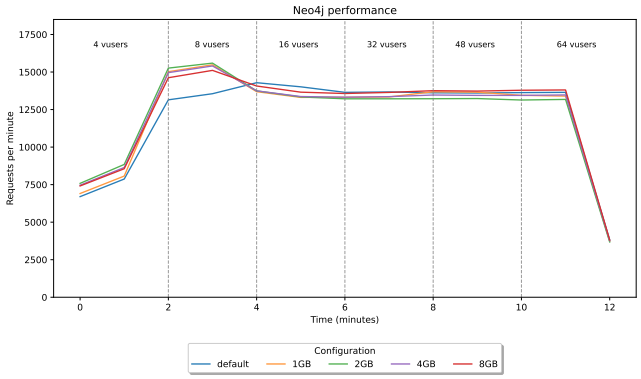


Figure 139: Machine 1 - Neo4j throughput

What we can observe is that, aside from Neo4j significantly outperforming PostgreSQL for this specific dataset and set of queries, both systems demonstrated consistency and maintained stable throughput, even under varying configurations.

Dataset 1 GB

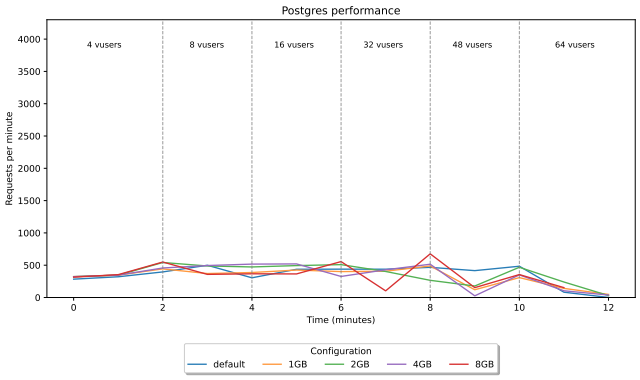


Figure 140: Machine 1 - Postgres throughput

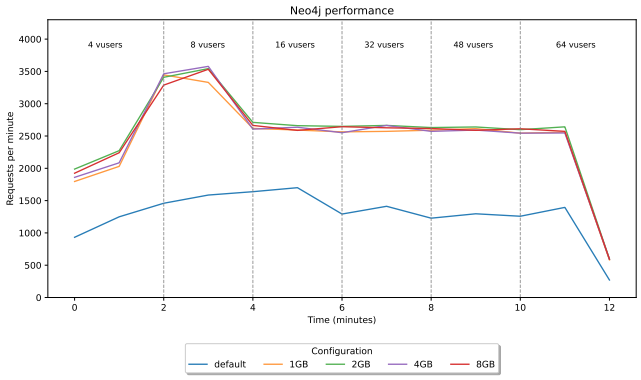


Figure 141: Machine 1 - Neo4j throughput

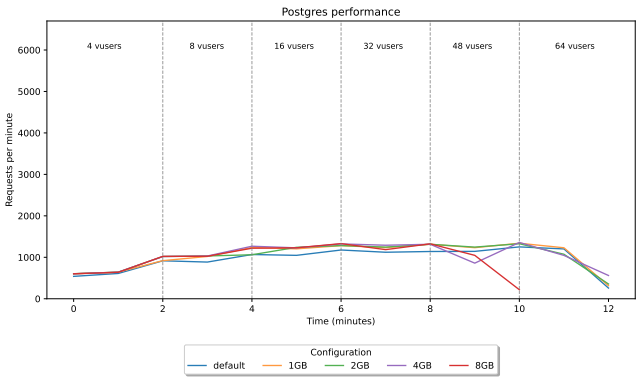


Figure 142: Machine 1 - Postgres throughput

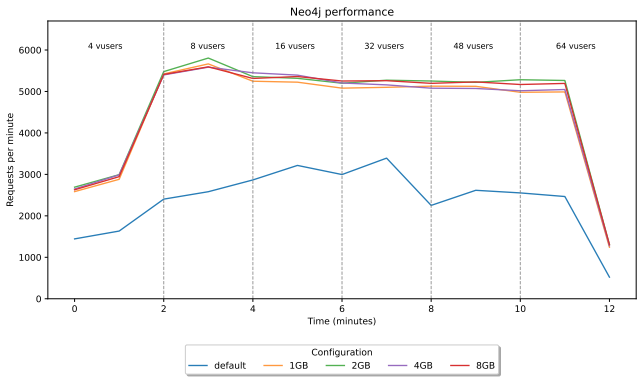


Figure 143: Machine 1 - Neo4j throughput

With a larger dataset, the performance of both systems degrades, particularly PostgreSQL. Even with a configuration allowing for increased memory usage, PostgreSQL reached points where throughput dropped nearly to zero requests answered per minute.

Neo4j, on the other hand, outperformed PostgreSQL even in its default configuration, achieving better performance than PostgreSQL with 8GB of memory. When Neo4j's configuration was optimized, it doubled its throughput and maintained steady performance, even with a high number of concurrent users.

Dataset 3 GB

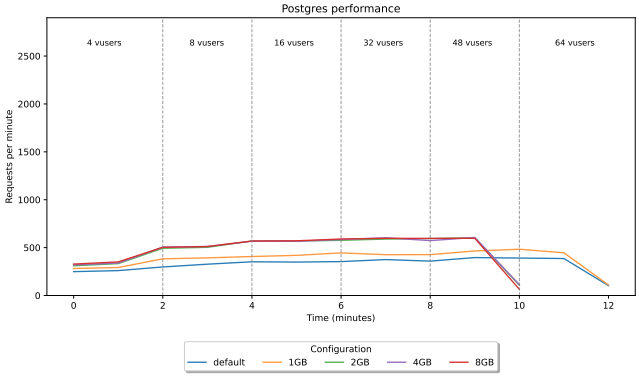


Figure 144: Machine 1 - Postgres throughput

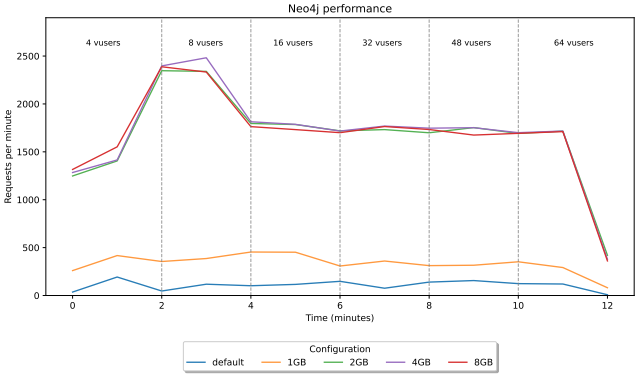


Figure 145: Machine 1 - Neo4j throughput

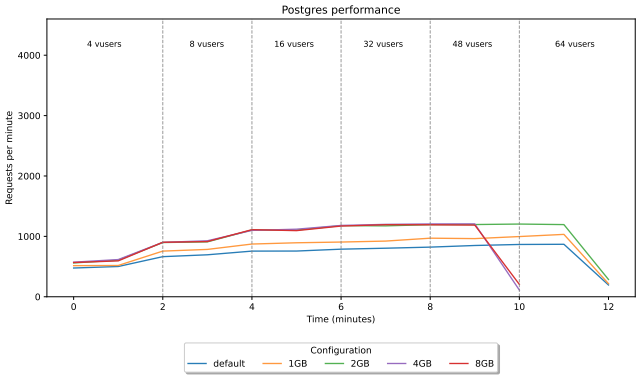


Figure 146: Machine 1 - Postgres throughput

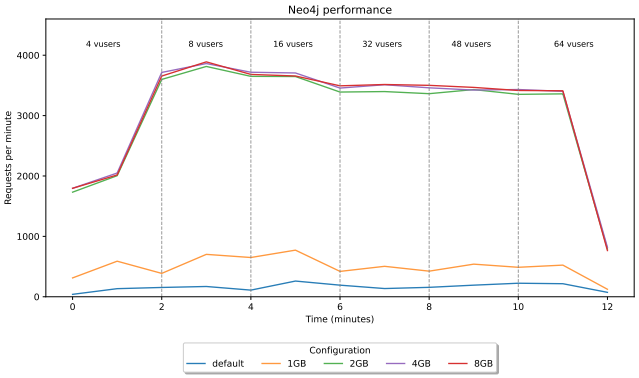


Figure 147: Machine 1 - Neo4j throughput

As the dataset grows even larger, PostgreSQL's performance remains relatively stable compared to its performance with the 1 GB dataset. However, Neo4j's performance shows noticeable degradation with the increased dataset size.

6 Conclusions

6.1 Difficulties

During our experiments, one of the challenges we encountered was identifying suitable inputs for our queries to create meaningful test cases. For example, a query involving a comment with a hierarchy of a dozen or more levels before reaching its corresponding post will likely exhibit different performance compared to a query where the comment's parent is already a post.

Another priority for us was ensuring query correctness in both DBMSs. During the creation of our queries, we occasionally observed discrepancies in the results between the two systems, even when using the same dataset. This required us to spend considerable time debugging and verifying that the queries produced consistent and accurate results across both DBMSs.

The next two points represent the areas where we invested a significant amount of time resolving issues.

One major challenge was loading the dataset into Neo4j. At first glance, this might not seem particularly difficult—and indeed, it typically isn't. However, despite thoroughly studying the Neo4j and APOC documentation and watching lectures on Neo4j's YouTube channel explaining various methods for bulk data insertion, we encountered persistent issues with one specific relationship not being created.

The issue revolved around the POSTED_IN relationship, which consistently failed to load after running our import script, even though all other relationships were correctly created. After extensive troubleshooting and experimenting with different approaches, we resolved the issue in the simplest way imaginable: by moving the POSTED_IN relationship's import statement to the end of the script. Once we reordered the script, the relationship loaded correctly.

Note: As a quick disclaimer, regardless of whether we successfully inserted the dataset or encountered issues, the nodes on which the relationship depended were always inserted prior to executing the relationship's import statement.

The final major challenge we encountered was designing a correct test plan for executing our benchmarks. Neither of us had prior experience with JMeter, so it took some time to learn and properly utilize the tool. Interestingly, the difficulties arose during the testing phase rather than during the initial learning phase.

One significant issue occurred while benchmarking PostgreSQL. During the concurrency tests, where we increased the number of virtual users, PostgreSQL retained many open processes in Windows after the test execution. Each of these processes represented a connection to PostgreSQL. The problem was that JMeter was not releasing connections quickly enough, causing our automated test script to take significantly longer than necessary.

After considerable troubleshooting, we discovered that the issue stemmed from the order in which queries were executed in our test plan. In JMeter, we defined a test plan that each newly created user would follow, and the sequence of queries had a substantial impact on the test results. If the first one or two queries executed by a user were heavy, it significantly affected the test's overall performance.

6.2 Results overview

The benchmarks conducted provided valuable insights into the performance characteristics of PostgreSQL and Neo4j under various scenarios and dataset sizes. Both systems demonstrated strengths and weaknesses depending on the query type, dataset size, and configuration.

PostgreSQL excelled in queries that required minimal recursion or relied on traditional relational operations, such as joins and filtering. Its performance remained consistent, especially for smaller datasets, with response times often falling within a predictable range. However, as dataset sizes grew and queries became more complex, PostgreSQL's performance began to degrade, particularly in scenarios involving heavy recursion or large numbers of joins. Notably, even with increased memory allocations and optimized configurations, PostgreSQL struggled with high concurrency tests, sometimes failing to maintain throughput under heavy load.

Neo4j, on the other hand, showcased its superiority in handling graph-like queries, especially those involving multi-hop relationships or recursive structures. While its initial query execution times were higher due to cold caches, Neo4j consistently outperformed PostgreSQL after the first query, leveraging its efficient graph-based optimizations and caching mechanisms. This trend became even more pronounced as dataset sizes increased, with Neo4j scaling effectively and maintaining stable performance even under heavy concurrency. With optimized configurations, Neo4j achieved significant throughput gains, outperforming PostgreSQL by a wide margin in many cases.

One of the key observations from these benchmarks is that Neo4j's native graph database architecture provides a clear advantage for graph-centric workloads, where relationships and traversal play a central role. However, PostgreSQL remains a strong contender for traditional relational queries, offering reliable performance in scenarios with smaller datasets or simpler query requirements.

To summarize, this project provided valuable learning experiences and opportunities for improvement across several areas. We deepened our understanding of Neo4j and its Cypher query language, exploring advanced features like the Graph Data Science library and optimizing configurations for better performance. Similarly, we gained insights into PostgreSQL internals, including recursive queries, memory management, and performance tuning. Working with SQL, we enhanced our ability to write and debug complex queries. Through JMeter, we learned how to design and execute effective benchmarking test plans, addressing challenges like concurrency and connection management. Overall, this project significantly improved our skills in database benchmarking and performance analysis.

Through JMeter, we learned how to design and execute effective benchmarking test plans, addressing challenges such as managing concurrency, optimizing query order, and ensuring efficient resource usage. We faced and overcame issues with connection handling, particularly in PostgreSQL, where slow connection releases initially impacted our test automation. These experiences not only enhanced our understanding of JMeter's capabilities but also provided a solid foundation for implementing load tests in similar scenarios.

Beyond technical skills, this project also strengthened our ability to work systematically, documenting observations and iterating on test setups to achieve accurate and reliable results. The knowledge gained will be invaluable for future projects requiring database performance evaluation or optimization.

7 Annex

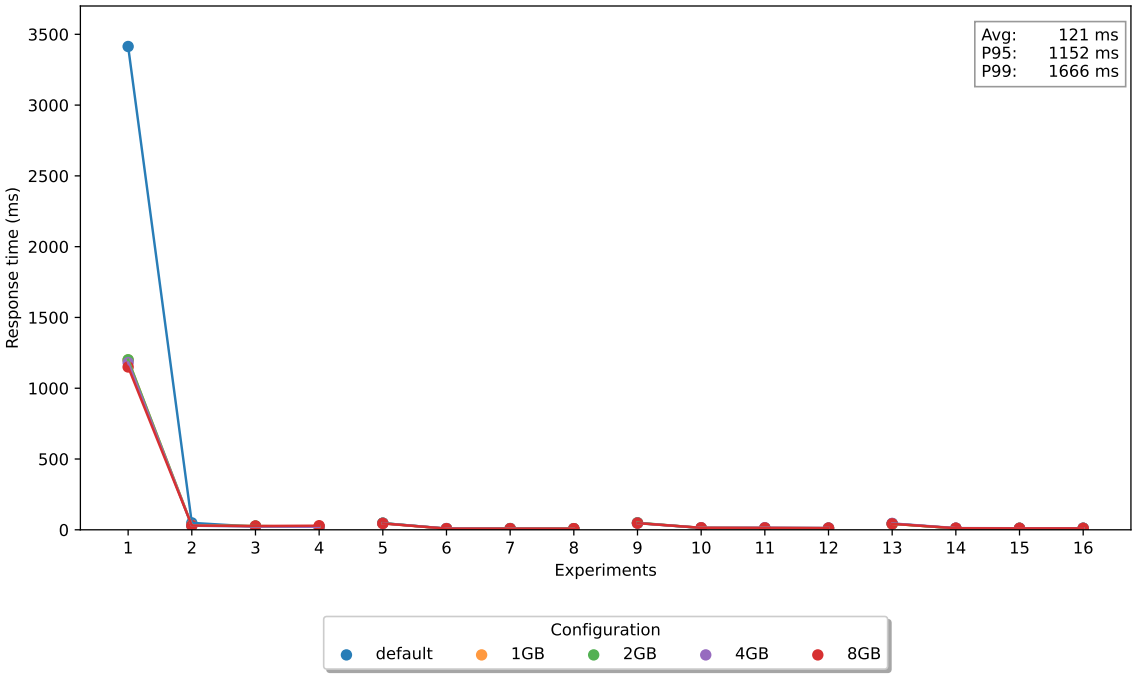


Figure 148: Complete graph for Machine 1 Neo4j dataset 0.3 GB

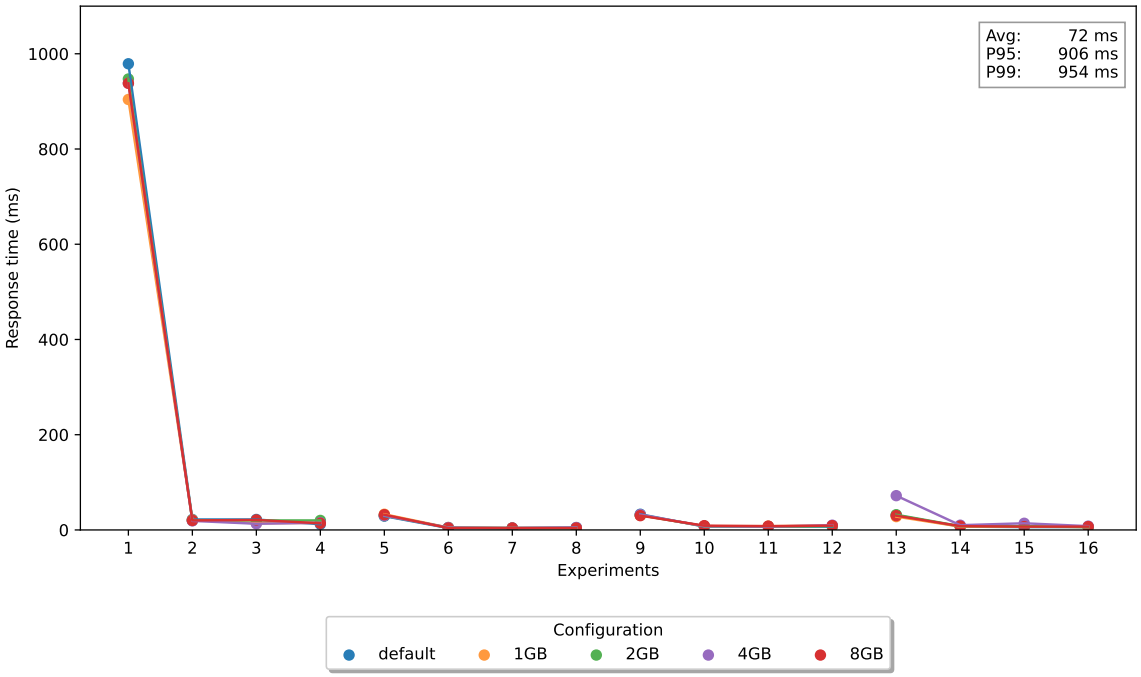


Figure 149: Complete graph for Machine 2 Neo4j dataset 0.3 GB

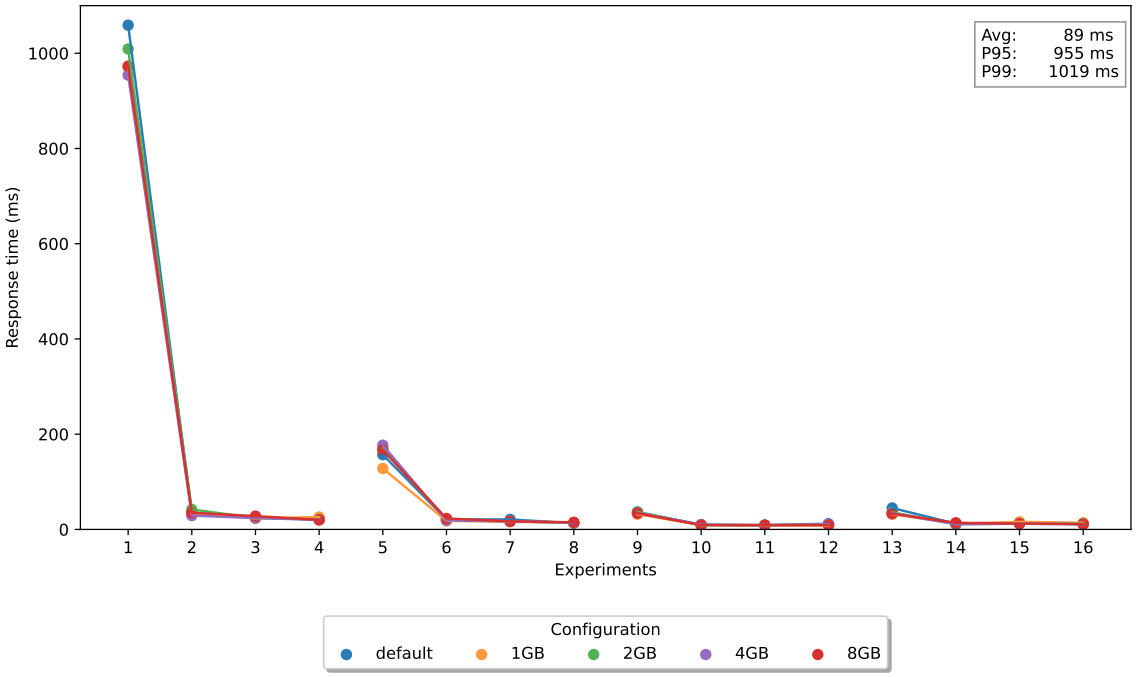


Figure 150: Complete graph for Machine 2 Neo4j dataset 1 GB

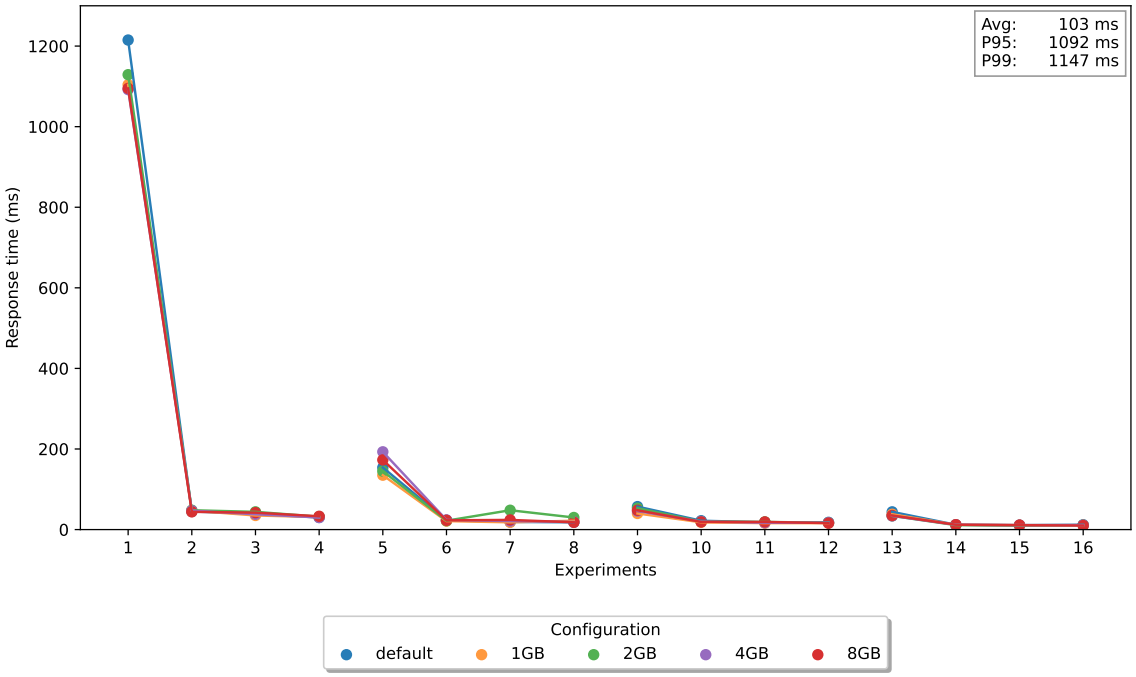


Figure 151: Complete graph for Machine 2 Neo4j dataset 3 GB

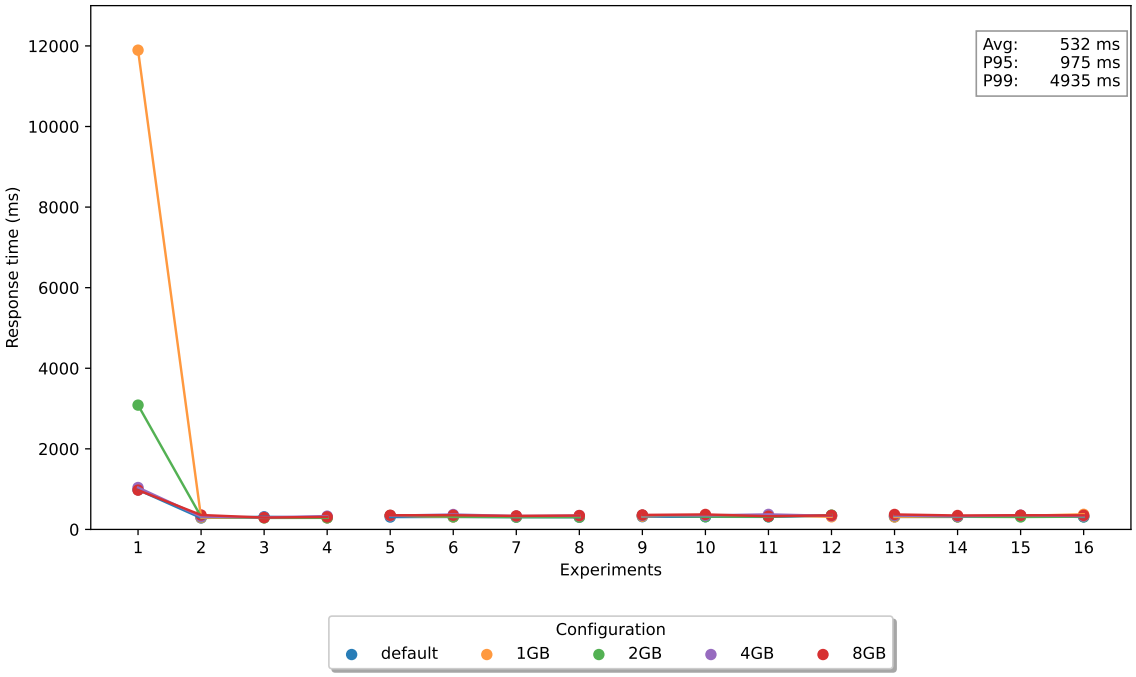


Figure 152: Complete graph for Machine 1 Postgres dataset 1 GB

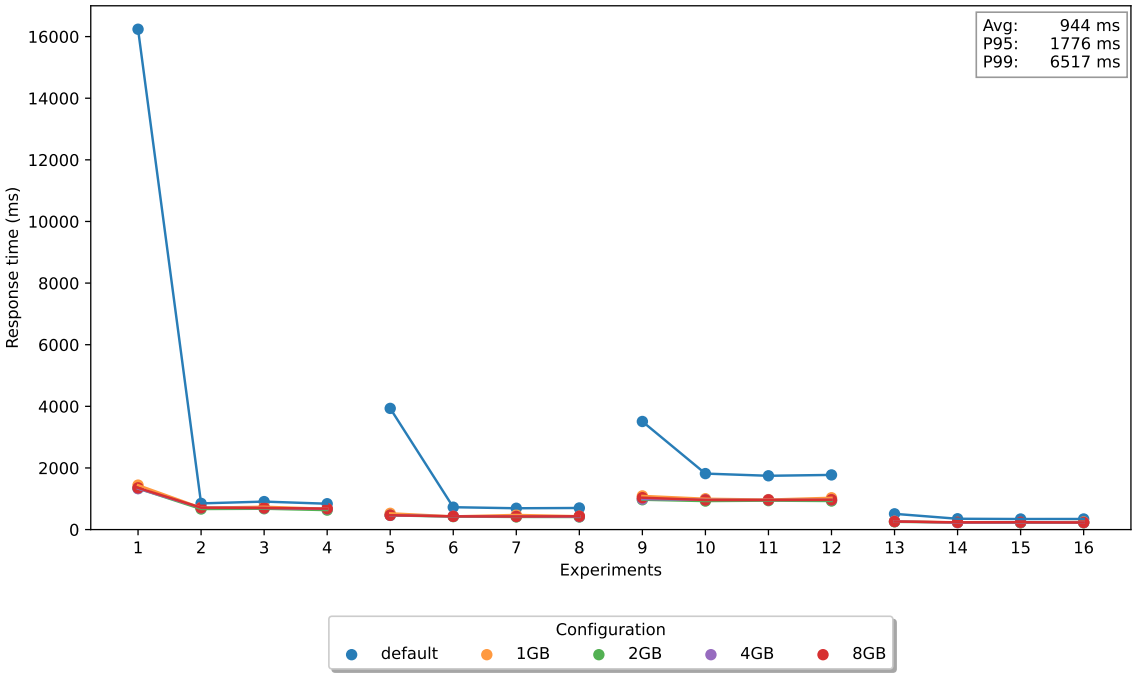


Figure 153: Complete graph for Machine 1 Neo4j dataset 1 GB

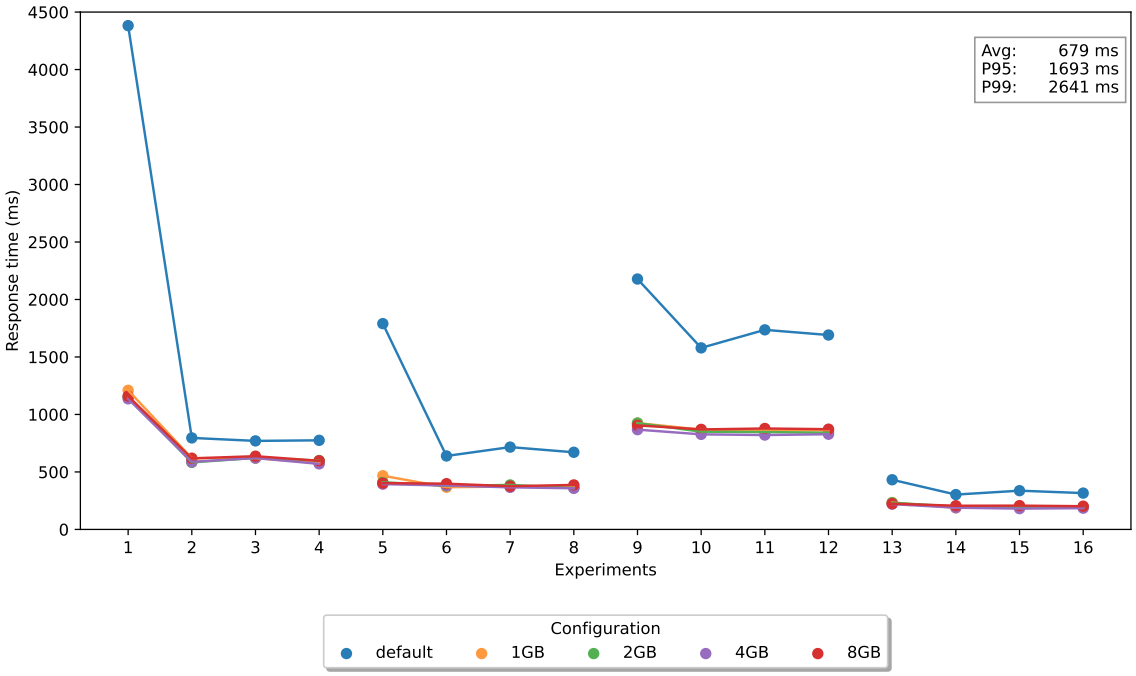


Figure 154: Complete graph for Machine 2 Neo4j dataset 1 GB

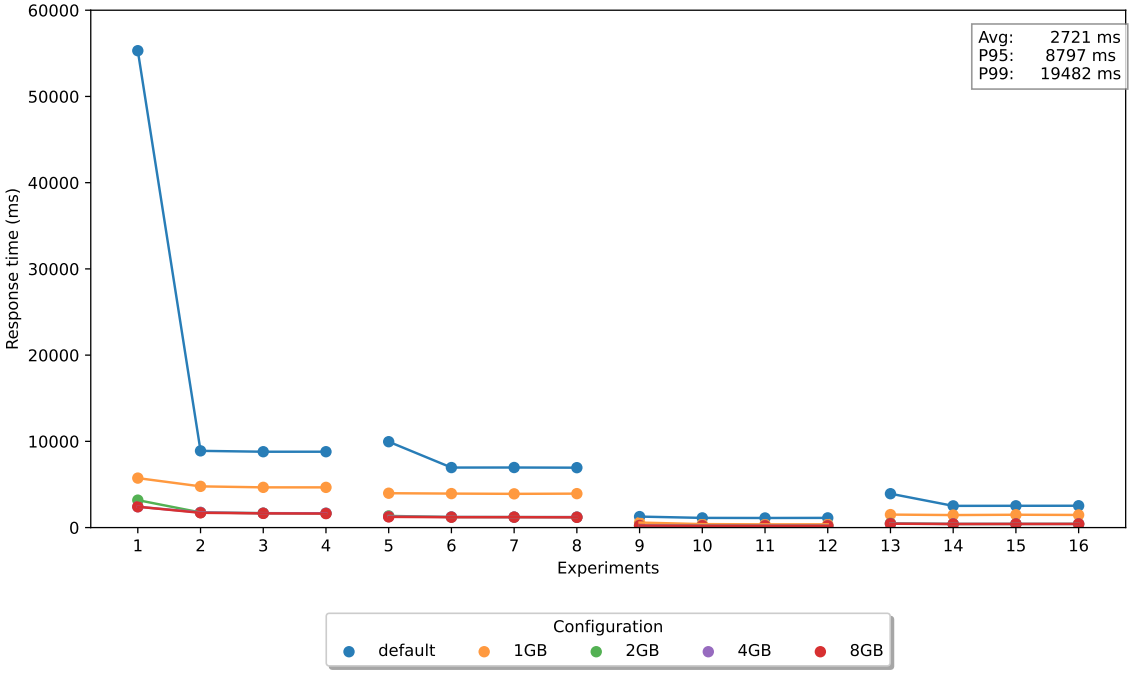


Figure 155: Complete graph for Machine 1 Neo4j dataset 1 GB

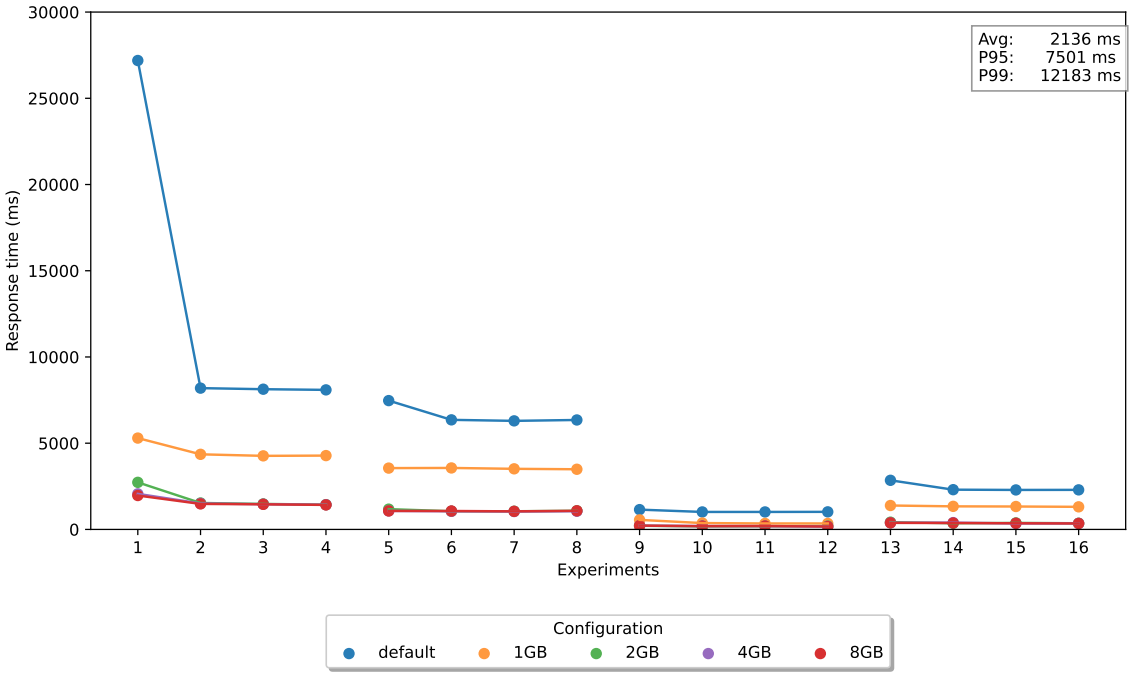


Figure 156: Complete graph for Machine 2 Neo4j dataset 1 GB

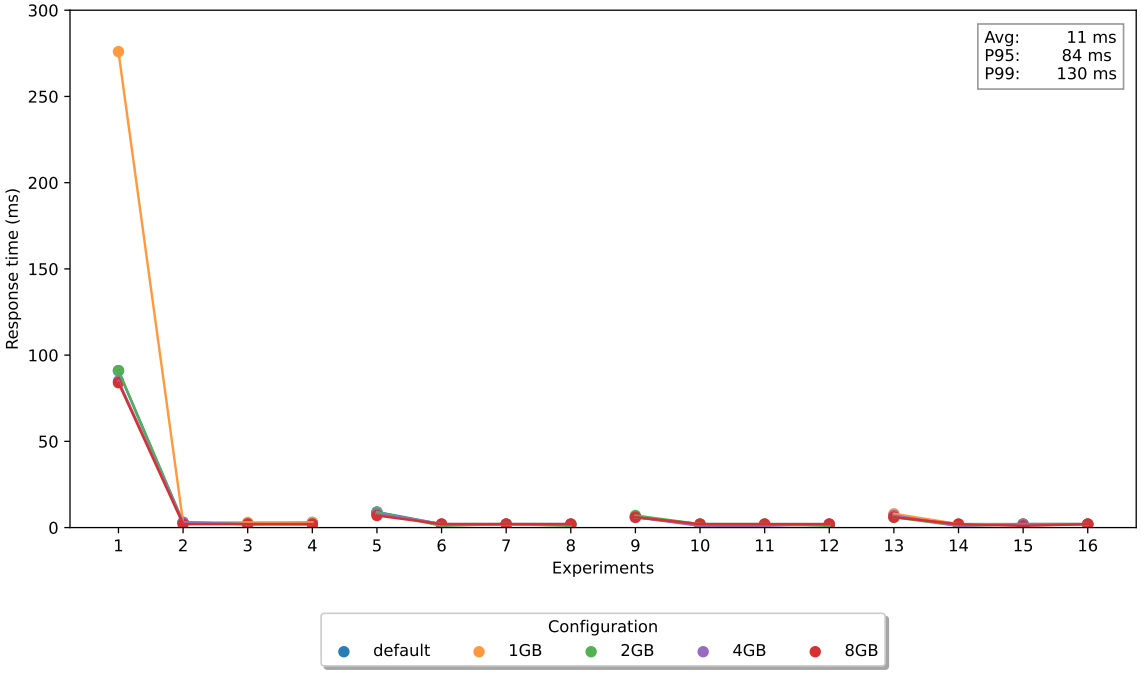


Figure 157: Complete graph for Machine 2 Neo4j dataset 1 GB

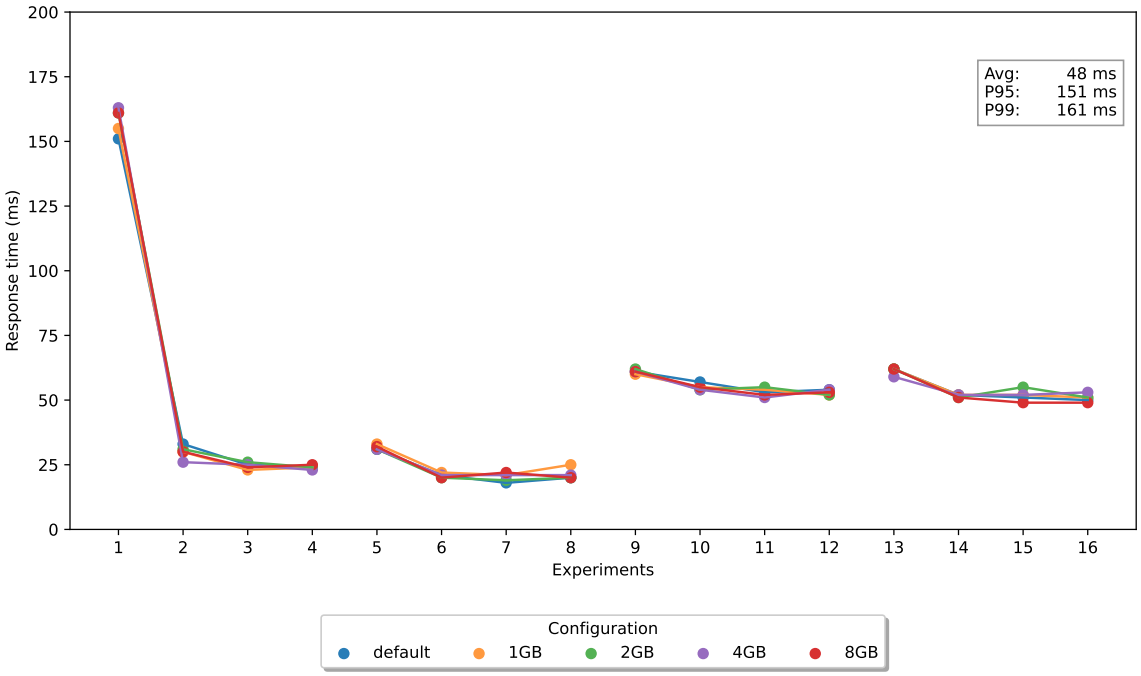


Figure 158: Complete graph for Machine 2 Neo4j dataset 0.3 GB

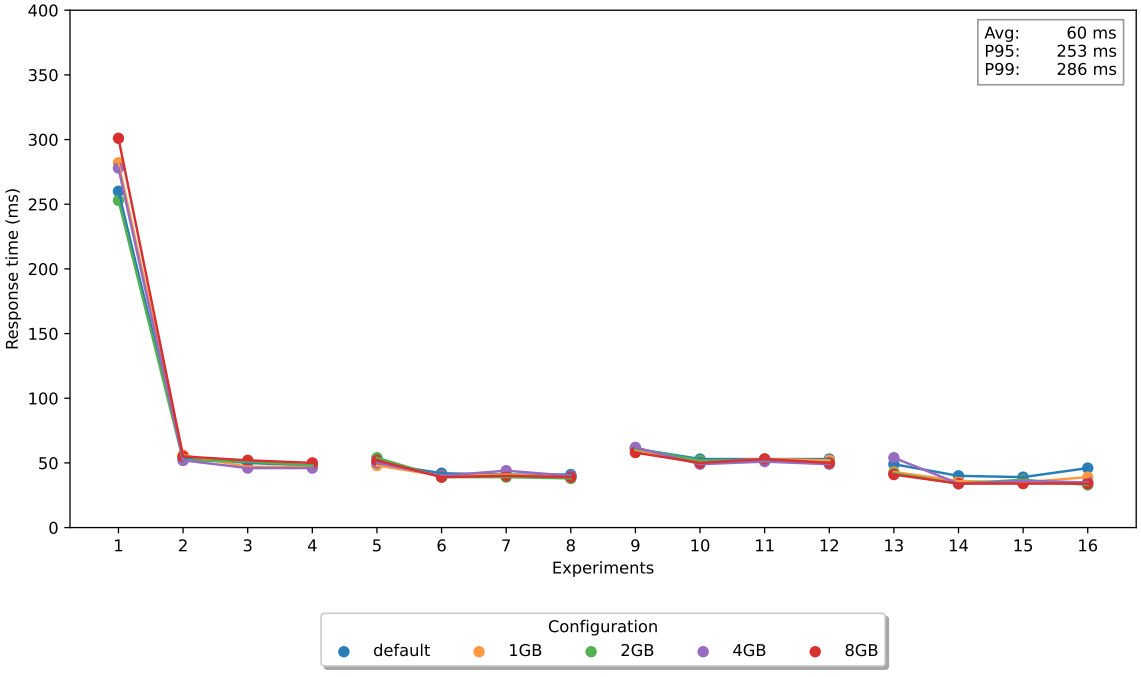


Figure 159: Complete graph for Machine 2 Neo4j dataset 1 GB

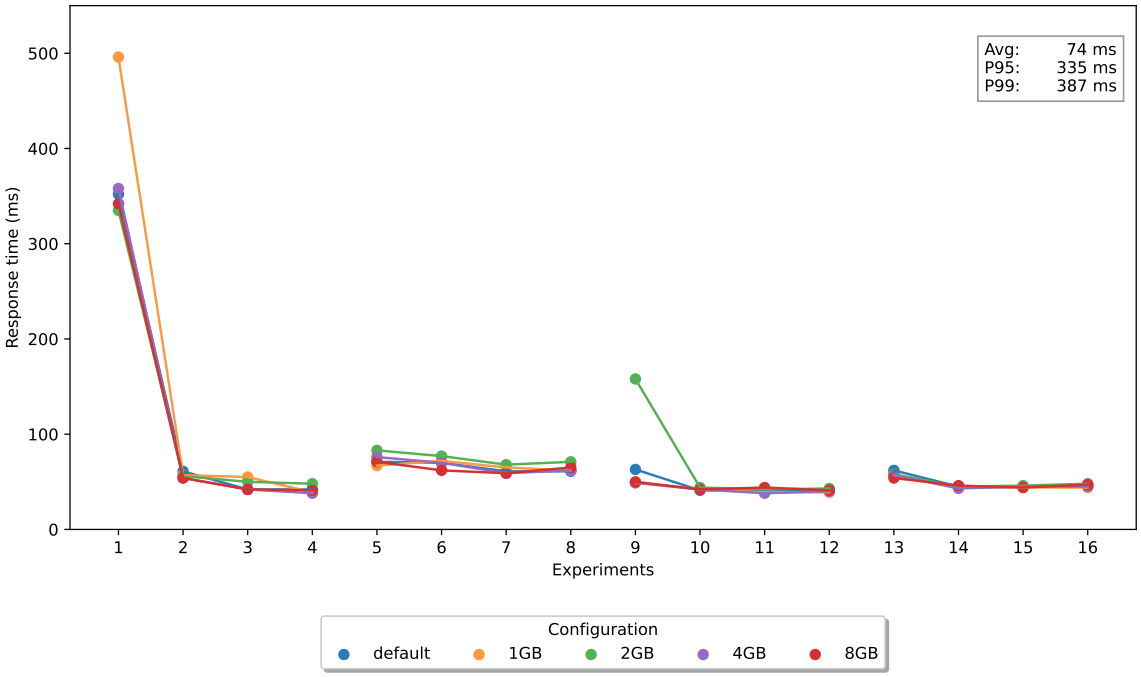


Figure 160: Complete graph for Machine 2 Neo4j dataset 3 GB

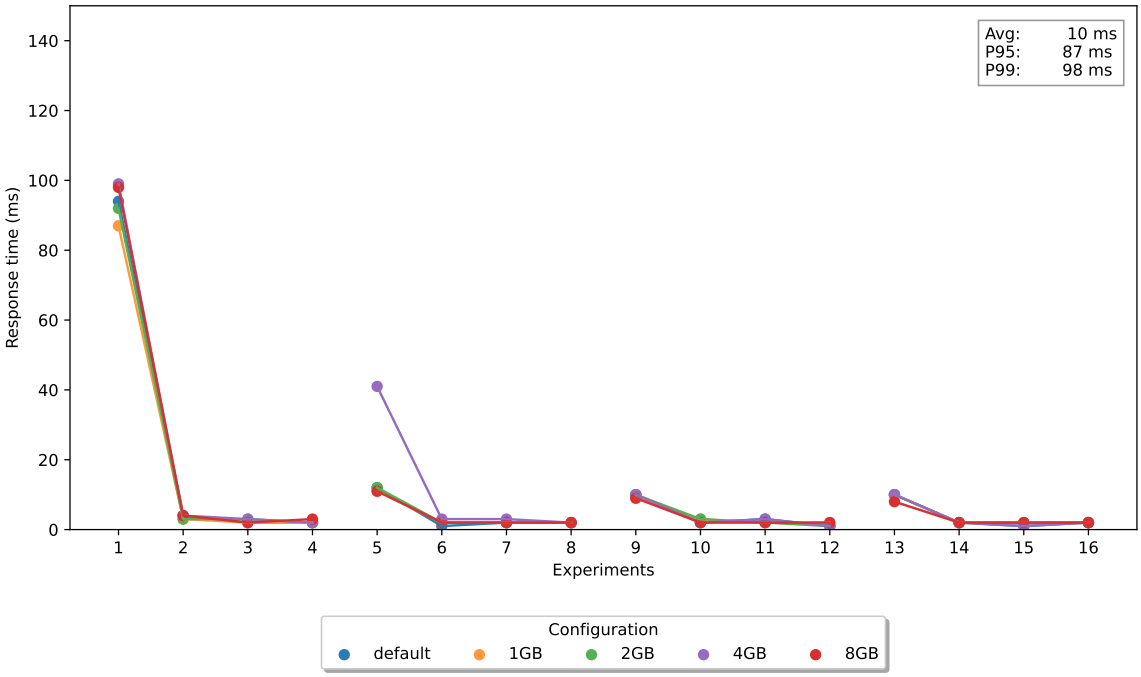


Figure 161: Complete graph for Machine 2 Neo4j dataset 0.3 GB

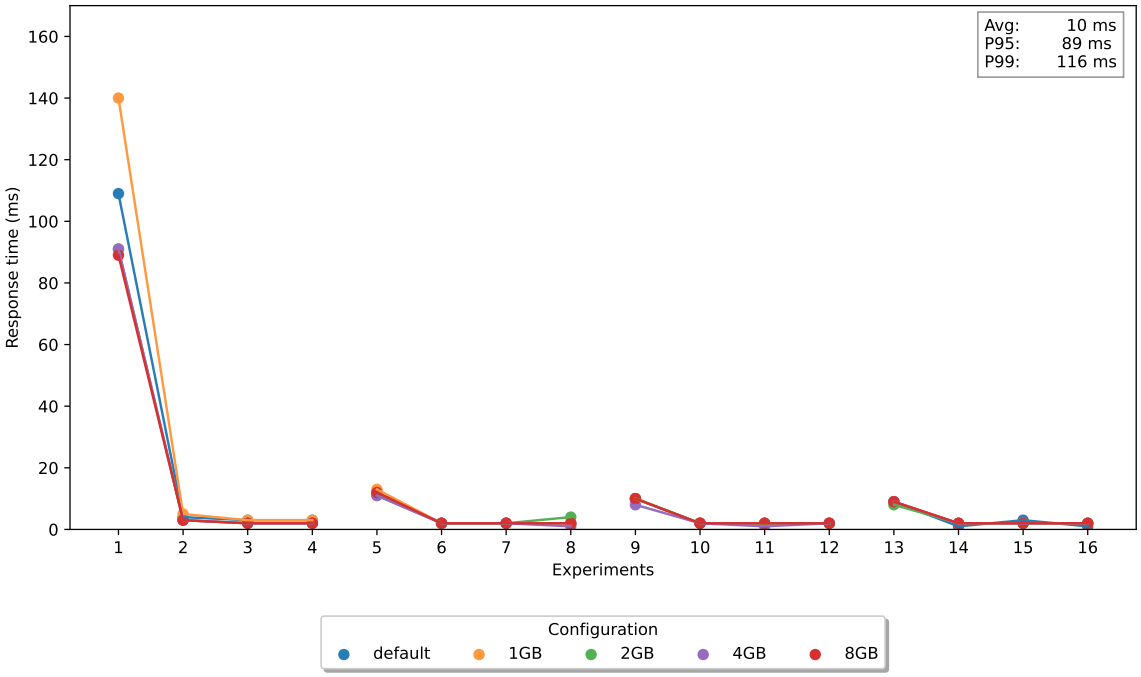


Figure 162: Complete graph for Machine 2 Neo4j dataset 1 GB

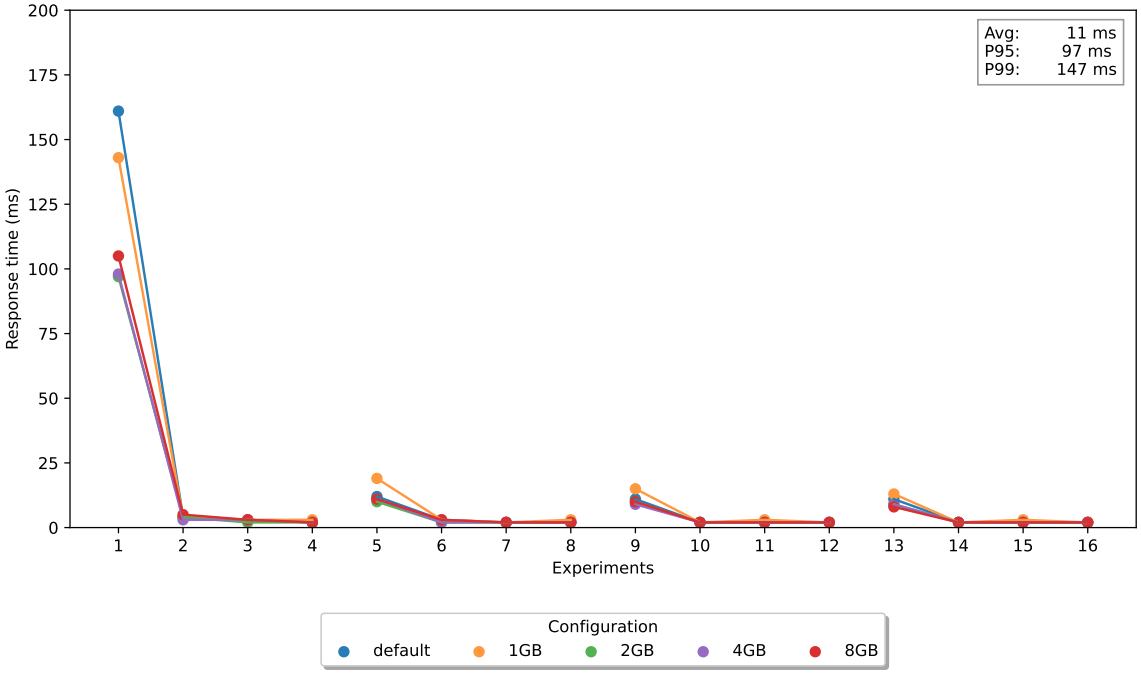


Figure 163: Complete graph for Machine 2 Neo4j dataset 3 GB

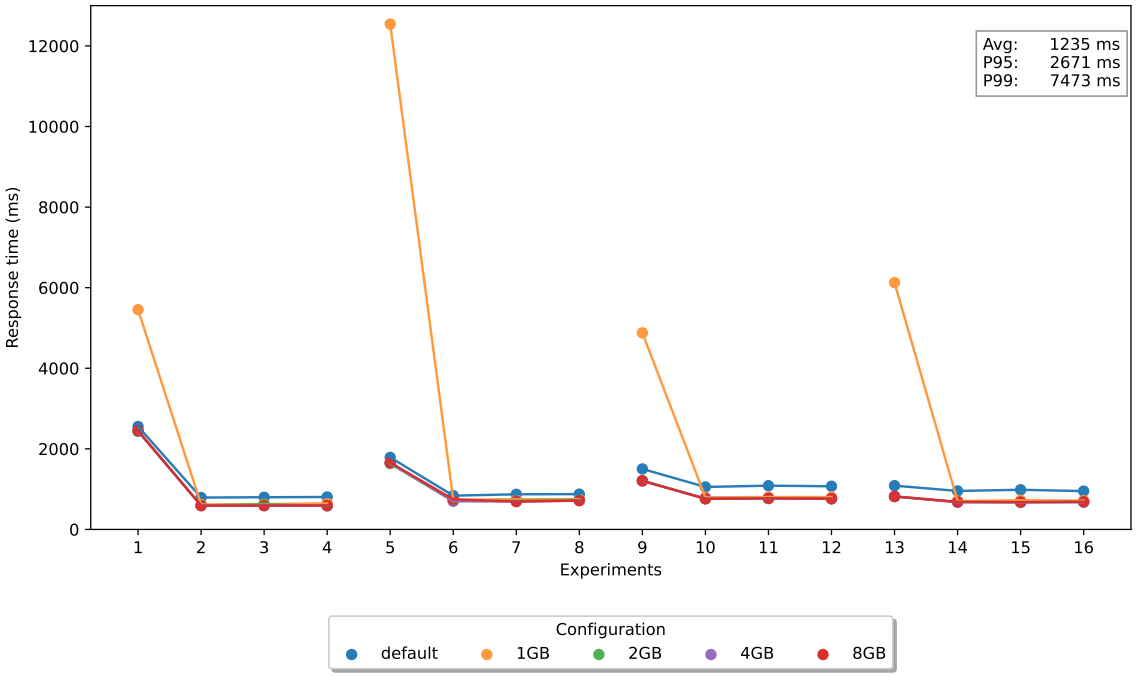


Figure 164: Complete graph for Machine 2 Neo4j dataset 3 GB

References

[1] Renzo Angles, János Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasić, Benjamin Steer, Gábor Szárnyas, and Jack Waudby. *The LDBC Social Network Benchmark*. Sept. 2024. URL: https://arxiv.org/abs/2001.02299v11.

[2] Linked Data Benchmark Council. *ldbc_snb_datagen_spark*. 2024. URL: https://github.com/ldbc/ldbc_snb_datagen_spark.

[3] PostgreSQL Global Development Group. *PostgreSQL Documentation: Populating a Database*. 2024. URL: https://www.postgresql.org/docs/current/populate.html.