

# Comparative Analysis of the Performance of PostgreSQL and Neo4j Databases in the Context of Genealogical Queries

Michał Muzyka\*, Mateusz Niedźwiedź, Marek Miłośz

*Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland*

## Abstract

The modern digital age led to the creation of a field called digital genealogy. To this day there seems to be little consensus on which database approach is most suited for such environment; genealogical data remains overlooked in database research. This study examines whether the popular relational SQL database systems prove themselves to be more efficient than graph databases when operating on graph-like genealogical structures. The article analyses PostgreSQL and Neo4j database engines' efficiency, comparing them across different sizes of genealogical datasets and types of operations on artificially generated data. Results show that an optimized SQL approach is generally faster and uses less memory than the graph-based counterpart, while requiring more disk space, in all tested cases.

**Keywords:** efficiency of a query; genealogical query; relational database; non-relational database; PostgreSQL; Neo4j

\*Corresponding author

Email address: [s97681@pollub.edu.pl](mailto:s97681@pollub.edu.pl) (M. Muzyka)

Published under Creative Common License (CC BY 4.0 Int.)

## 1. Introduction

Nowadays, more and more people are interested in genealogy, which involves searching for information about family history and origins. For many, it has become an absorbing passion. One of the most important aspects of this field is documenting relationships between family members. Beginners usually intuitively, yet correctly, represent them as a graph, with people as nodes and relationships as edges. However, they typically just write them down on a piece of paper. For more experienced genealogy enthusiasts, this proves insufficient, as they collect hundreds or even thousands of records about people and constantly update them. This naturally forces them to manage their data digitally. For this purpose, genealogical programs and online services using databases were created. For software engineers, this raises questions: which databases are used by such applications and how this data is stored. Most importantly, however, the question is which type of database offers the best efficiency for genealogical queries, that is, queries concerning information about family members and their relationships.

Relational databases have dominated the market for years, but non-relational databases have also become popular. They differ in the way they store data and handle relationships. Genealogical data is structured in a hierarchical, graph-like way and often requires complex, deep queries. Database efficiency may depend on the data model and the type of query. Therefore, choosing a database for a genealogy application may not be obvious, and comparing their efficiency across different scenarios can provide new valuable insights, as related research in this area is sparse and mostly outdated.

For the purposes of this study *efficiency* was defined mostly as speed of query execution alongside the less important RAM (Random Access Memory) and disk usage, where lower values are deemed as more efficient.

## 2. Purpose and scope of the work

The purpose of this research is to determine which database system type – SQL or graph-based – will prove more efficient in context of genealogical datasets.

To achieve that goal, three different genealogical datasets were created in addition to testing application which gathered relevant results.

In essence, the study answers the following research questions:

**Q1:** Which system — PostgreSQL or Neo4j — achieves shorter response time of queries in a genealogical context?

**Q2:** How does changing the data size affect the response time of PostgreSQL and Neo4j?

**Q3:** Does the system with the shorter response time require more system resources (RAM, disk usage)?

## 3. Background

### 3.1. Modelling genealogical data and its specifics

To understand the specifics of genealogical data, it is best to start by analysing the structure of a GEDCOM (GEnealogical Data COMMunication) file, which is de facto the standard for exporting and importing data between genealogy applications [1]. It stores data common to various genealogical solutions, such as individuals and their parental and marital relationships but also information associated with them like people's names, their dates of birth, marriage and death, or the places where these events occurred.

Genealogical data has specific characteristics that must be taken into consideration when handled by databases. Although a parent-child relationship graph is commonly referred to as a *family tree*, it is in fact most often a DAG (Directed Acyclic Graph) in which the phenomenon of *pedigree collapse* frequently occurs. When related individuals have a child, this disrupts the tree structure. Next issue pertains to people who were

adopted or had children with different partners. GEDCOM format handles these situations [2] and every comprehensive application along with its database should account for these situations as well.

When it comes to SQL (Structured Query Language) relational databases, they do not naturally support hierarchical or graph structures. Nevertheless, there are certain approaches [3], the most popular of which is the *Adjacency List* with a separate table for edges [4]. Its write operations are fast but read operations are relatively slow. This method requires multiple JOIN operations or the recursive use of CTEs (Common Table Expressions), which modern databases such as PostgreSQL support [4]. Another approach is the *Closure Table* [4], which is mainly used as a precomputed structure to speed up queries. Other approaches are generally unsuitable for genealogical graphs.

Non-relational databases are more flexible and scalable than relational databases [5]. Among them, document databases such as MongoDB can be distinguished, which are a reasonable option for storing low-depth genealogical trees within a single document [5]. Another type is wide-column databases such as Cassandra [5], which are generally not well suited for hierarchical data, but are effective as an operational layer due to denormalization [6]. Key-value databases such as Redis can also be distinguished, however they are generally only suitable for storing simple key-based data [5]. The last but significant category is graph databases such as Neo4j, which naturally represent data in graph form as nodes and edges [7]. Both elements can store properties, and traversing from one node to another (i.e. *traversal*) is a native operation [8].

### 3.2. Databases used by genealogical services and programs

Analysing the genealogical services and programs, one might come to conclusion that they use both relational and non-relational databases for storing genealogical trees. However, not all information is available publicly and most of it remains speculative.

The Findmypast service acknowledges that it uses different databases including Microsoft SQL Server and MongoDB. However, PostgreSQL is their de facto standard for relational databases [9]. They also experimented with the Neo4j database, with promising results [10]. The MyHeritage service most likely uses the ScyllaDB wide-column database [11]. The FamilySearch service used an Oracle SQL database, but due to insufficient performance, they switched to a wide-column database based on Apache Cassandra [6]. The Ancestry service most likely uses one of the graph-based databases [12]. The RootsMagic program uses the SQLite 3 database engine [13]. Gramps, an open-source program, uses SQLite and PostgreSQL and its data model is based on objects such as *Person*, *Family* and *ChildRef* [14].

### 3.3. Comparative efficiency analysis of relational and non-relational databases

When talking about similar studies analysing differences between database systems, most works focus only on the *base* capabilities of the engines [15]. Although optimizations of data structures and queries were a common occurrence, extensions like for example APOC (Awesome Procedures on Cypher) [16], were rarely taken under consideration.

Another trend often found in other works is usage of multiple datasets, mainly differing in size, ranging from couple thousand up to a several million of items [17]. However, the variation in dataset sizes was not something too common, with many studies often trying different types of data structures [18] or simply relying on a single dataset [16].

The most common metric was execution time [15], or its derivatives like *throughput* [19] defined by the number of operations per unit of time. Another less common appearing metric was disk usage, describing amount of space taken by imported data, however in some cases it was the focus of the whole work [17]. The common way of gathering results was using an external application written in widely used languages such as Java [20].

Results reported in other studies are rather inconclusive. Some studies determine clear domination of the SQL-based approach in most cases [21], some implying that non-relational databases are the more efficient ones in all types of situations [16] and some reporting more mixed results [22]. Despite that, one general observation can be made: the structure of the tested data [18] and the type of query being executed [22] seem to have a significant influence on the results.

No recent reputable studies were found comparing the performance of databases for genealogical data. The latest relevant study identified was published in 2017 [23]. The authors compared database performance on genealogical DAG-structured data. PostgreSQL outperformed Neo4j and MongoDB in execution time, though the study was limited by a small dataset, did not consider measurement of resource usage and scalability, and used only two types of query scenarios.

### 3.4. Research gap

As described before, genealogical data is quite specific in its structure and because of that little research has been done comparing the performance of operations on this type of data in database engines, and the existing studies are incomplete and outdated. Graph databases appear in studies, however mainly as one of the contenders rather than the focus of the study. From this it can be concluded that a study pertaining to the genealogical prospect of databases might prove important, not only in the field of digital genealogy, but also in the field of graph databases.

## 4. Research materials and methods

### 4.1. Selected databases

For the experiment, PostgreSQL and Neo4j were selected as representative relational and non-relational databases, respectively.

The PostgreSQL database was selected due to its popularity among relational databases and high stability. It has advanced and modern features, including support for recursion in CTE expressions, which is important for hierarchical structures. An additional advantage of this database is that it is fully open source.

Neo4j was also chosen because of its popularity and large community. It is a natural choice for graph structures, where data has many connections. It has good documentation, and its query language, Cypher, is easy to understand. In addition, it features the Neo4j Browser tool for intuitive and user-friendly graph visualisation.

### 4.2. Data model

The genealogical data model prepared for the study contains the most important and common data stored in GEDCOM files. It is a compromise between implementation simplicity and the complexity of genealogical systems. On the one hand, less significant elements such as events and sources have been omitted to focus the analysis on key genealogical queries. On the other hand, the model is complex enough to handle common scenarios. The *Person* and *Family* entities can be identified here. In terms of relationships, a parent–child relationship can be distinguished with types such as *biological*, *adopted*, or *foster* as well as a partnership relationship with statuses such as e.g. *married*, *divorced*, or *partners*. The *Family* entity here is an aggregate of spouses and children. This is the same approach used in formats such as GEDCOM or in software such as e.g. Gramps, and it allows for easy retrieval of children originating from a given union. This data model was implemented for both databases (Figures 1 and 2).

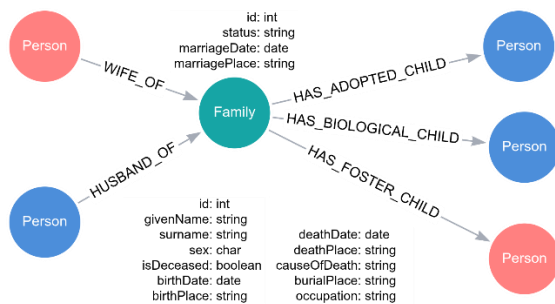


Figure 1: Graph data model for Neo4j

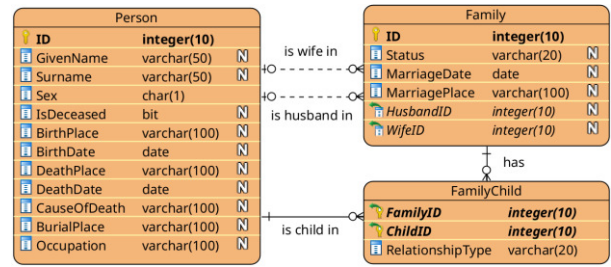


Figure 2: Physical ERD diagram for PostgreSQL

### 4.3. Dataset generation

For the purpose of the study, three synthetic datasets containing 1,000, 10,000, and 100,000 individuals were generated. Synthetic data were used due to the ability to precisely control dataset size and the number of generations, the lack of sufficiently suitable historical data, and legal constraints associated with their use. Nevertheless, the parameters were selected to roughly correspond to a typical pre-industrial period. The data were generated using a Python script in a top-down approach, i.e. from an initial group of ancestors down to their descendants. The simulation runs year by year, starting from a predefined initial year until the target population size is reached. Five regions were created. Between years 20 and 30 of the simulation, additional immigrants are introduced. Marriage age is modelled with mean ages of 22 for women and 25 for men. Partners from the same region are preferred, and pedigree collapse occurs, whereby the sum of the distances between candidates and their closest common ancestor must be at least seven. The age distribution is dominated by young individuals, with few elderly people, especially above 80 years of age. The number of children in a family is sampled from a Poisson distribution. Historical child mortality (30%) is also modelled here. First names, surnames, occupations, causes of death, and places are randomly selected from predefined name pools.

A helper script was created to identify the parameters, such as e.g., the random seed value, that generate a non-extinct population with a predefined number of individuals and generations for a given dataset (Table 1). A fixed random seed was used ensuring that the study can be replicated by other researchers. The script generates CSV (Comma-Separated Values) files with the same data for both databases, formatted for bulk import via their respective engines. It also generates statistics for the generated datasets in the form of text files and charts, such as that in Figure 3, as well as recommending parameters for individual scenarios, automatically updating them in queries and creating files with the exact expected query results for validation.

Table 1: Simulation parameters for each dataset

| Parameter                              | 1,000    | 10,000    | 100,000   |
|----------------------------------------|----------|-----------|-----------|
| Initial year                           | 1650     | 1400      | 700       |
| Seed (helper script)                   | 748,376  | 748,376   | 748,376   |
| Poisson fertility rate (helper script) | 7.10     | 4.99      | 4.53      |
| Founder couples (helper script)        | 19       | 63        | 77        |
| Immigrants per year (helper script)    | 1        | 2         | 4         |
| LCA path search depth (scenario)       | 8        | 12        | 21        |
| LCA path expected depth (scenario)     | 8        | 12        | 16        |
| Ancestor depth (scenario)              | 5        | 10        | 20        |
| Descendant depth (scenario)            | 5        | 10        | 17        |
| <b>Number of generations obtained</b>  | <b>9</b> | <b>20</b> | <b>50</b> |

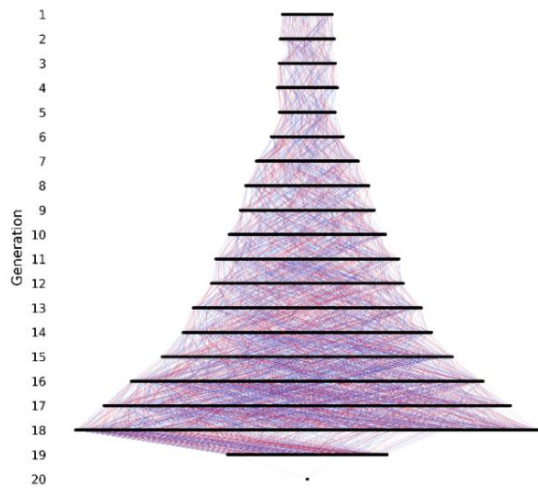


Figure 3: Parent-child relationship graph (10,000 dataset)

#### 4.4. Experimental environment

The experiments were conducted in the environment specified in Table 2.

Table 2: Experimental environment specification

| Category | Component                 | Specification                                            |
|----------|---------------------------|----------------------------------------------------------|
| Hardware | Processor                 | Intel Core i7-10700K, 3.8 GHz, up to 5.10 GHz, 16M Cache |
|          | RAM                       | Kingston FURY 32GB (2x16GB) 3600 MHz CL16                |
|          | Disk                      | Samsung 1TB M.2 PCIe NVMe 970 EVO Plus                   |
|          | Operating system          | Ubuntu Server 24.04.04 LTS                               |
| Software | Docker Engine             | 29.3.0                                                   |
|          | Container resource limits | 8 GB RAM, 8 CPUs (shared, one database active at a time) |
|          | PostgreSQL                | 17.9                                                     |
|          | Neo4j                     | 5.26.23-enterprise (LTS)                                 |
|          | .NET SDK                  | 9.0.116                                                  |
|          | BenchmarkDotNet           | 0.15.8                                                   |
|          | Python                    | 3.13.13                                                  |

The databases were run in Docker containers managed by Docker Compose. Their memory was configured to ensure optimal performance and fit within the container memory limit. Logging and other unnecessary system mechanisms were disabled in both databases to reduce the I/O overhead. Parallel processing was disabled in both database systems to compare single-threaded execution during benchmarking.

Ubuntu Server was installed on the machine in a minimal configuration without unnecessary packages. A 150 GB partition was created. Using a Bash script, the environment was prepared for reproducible and stable benchmarking. Unnecessary services, sockets, timers, and swap were disabled. The processor was set to performance mode, and Turbo Boost was enabled. Stable versions of Docker, .NET SDK, and Python were installed, and Python virtual environments were created. Container images for the databases were also downloaded and prepared.

#### 4.5. Test scenarios

For the study, nine typical and diverse test scenarios were created, covering different use cases of the genealogical system, ranging from simple filtering operations to deep graph traversals:

1. Finding the lowest common ancestor(s).
2. Filtering individuals by surname and birth year range.
3. Counting the number of births by city and quarter-century.
4. Retrieving all unique ancestors of a given person up to the Nth generation.
5. Retrieving all unique descendants of a given person up to the Nth generation.
6. Finding the person/people who became a parent at the youngest age.
7. Updating place names.
8. Adding biological parents to an adopted person and their sibling.
9. Removing the biological parents of the adopted person and their sibling.

The queries often return multiple columns. For different dataset sizes, the traversal depth of the search increases (scenarios 1, 4, and 5). The scenarios can be divided into traversal scenarios (1, 4, 5), calculation scenarios (2, 3, 6) and modification scenarios (7, 8, 9). Some of them include only biological relationships.

It was assumed that the queries in the experiment would be written in pure Cypher and SQL, without additional extensions such as APOC for Neo4j or Apache AGE for PostgreSQL. The best solutions for each database were used instead of forcing analogous implementations and execution time was the main priority. The queries were required to return exactly the same tabular or scalar result for both databases. To optimise queries, standard indexes based on primary and foreign keys, and columns used in WHERE and JOIN clauses were applied.

#### 4.6. Performance metrics

The performance metrics of the study are presented in Table 3.

Table 3: Performance metrics

| Metric                                 | Unit | Resolution |
|----------------------------------------|------|------------|
| Response time of a query / Import time | ms   | 1 ns       |
| Memory usage / Disk usage              | MB   | 4.096 KB   |

For the purposes of the study, response time of a query was chosen as the main metric of efficiency, being a commonly used metric, seen in various studies of similar kind. A secondary metric – average memory usage is present in the study; however, analysis of the results was focused primarily on the response time. For Neo4j, an additional memory usage metric is present, based on JVM (Java Virtual Machine) heap size values instead of raw memory used by the process. However, units and resolution of this metric are the same for both approaches, with results being measured during or immediately after query execution.

For import segment of the study, two metrics were determined: execution time of bash scripts responsible for importing the relevant dataset to a database and disk space taken by imported data.

#### 4.7. Test application and processing experimental data

Application testing the efficiency of the set-up databases is a program written in C# using .NET 9. Testing consisted of repeated calls to databases for execution time segments and Docker-based bash script invocations for import segment of the testing. BenchmarkDotNet library, used in the process of testing, measured the execution times of queries and bash scripts in addition to basic aggregation of the results. For RAM usage data, results were gathered from docker container *cgroups* files – providing exact data about resource usage, including memory. Additional heap memory data for Neo4j was read directly from the JVM the database ran on. Disk space usage was read using *du* – a utility program available on Linux.

The measured part of the execution of the queries involved calling previously created database drivers either fully consuming the result rows and their fields into a reusable object or fetching the number of items affected, for non-modifying and modifying queries respectively. Before and after each iteration special setup and cleanup steps were invoked (Figure 4), which stabilized the periodic memory reading, but also reversed operations for data modifying queries. Script invocations were executed similarly to queries, however no return values besides exit code were being processed.

Testing occurred for each combination of database and dataset with first couple of measurements treated as warmup and being ignored in further analysis. For import tasks first two measurements were treated as warmup, while for execution times first 10 attempts were ignored as well. After each iteration, the script pauses for two seconds to ensure that memory measurements are correct, and previously executed tasks have little influence on the readings. During and after each iteration, a periodic memory reading was conducted with a 30-millisecond interval.

Data import was performed using the COPY and neo4j-admin tools, as they were determined to be the fastest methods for importing large amounts of data into PostgreSQL and Neo4j databases, respectively.

Resulting data for both execution times and memory usage did not allow for assumption of equal variances between samples, therefore a Welch's t-test for two-sample mean comparison was used during the analysis. No outliers were identified or excluded in the gathered data for both time- and byte-based metrics.

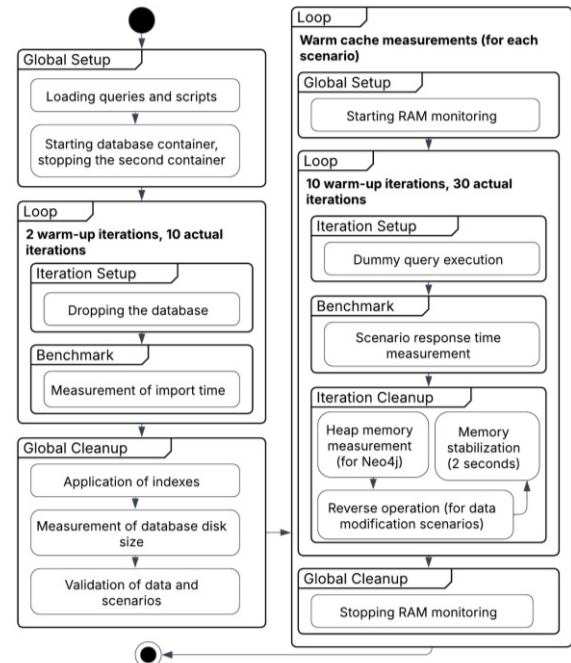


Figure 4: Measurement workflow for each combination of data size and database

## 5. Results

Results of the performance analysis are presented in three subsequent sections: import analysis, response times and memory usage. Plots regarding time-based metrics additionally display 95% confidence intervals in black.

### 5.1. Import analysis

Import times (Figure 5) for PostgreSQL were significantly faster than import times for Neo4j for all dataset sizes. Neo4j took over three seconds to import data for each dataset size, while PostgreSQL never passed a quarter of a second of import time. The import times grew with dataset size for both databases, with PostgreSQL needing more than three times more time with largest dataset than with the smallest one. Neo4j scaled relatively much better in that regard, needing only about 1.5 seconds more during the import of the largest dataset, compared with import of the smallest one – percentage difference of 37% between results for largest and smallest one.

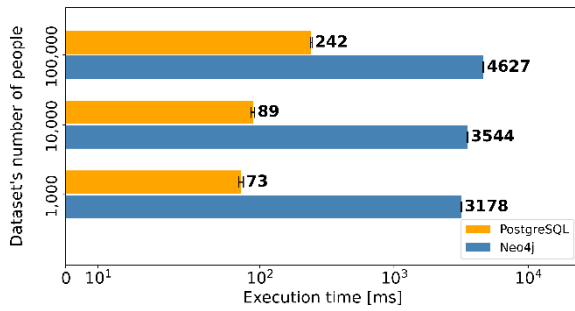


Figure 5: Average import time

Disk usage (Table 4) was lower for Neo4j in every case; however, with increasing dataset size, the percentage difference between two database systems grew smaller. The absolute difference, though, remained constant (~7 MB).

Table 4: Disk usage

| Database              | 1,000 people dataset | 10,000 people dataset | 100,000 people dataset |
|-----------------------|----------------------|-----------------------|------------------------|
| PostgreSQL            | 8.5 MB               | 12.2 MB               | 49.9 MB                |
| Neo4j                 | 1.5 MB               | 5.1 MB                | 42.9 MB                |
| Percentage difference | 140%                 | 82%                   | 15%                    |

## 5.2. Response times of the queries

Despite varying dataset sizes and queries differing from traversal to data modification, PostgreSQL proved itself to be faster (Figure 6), barring one exception. Additionally, for queries of traversal (queries 1, 4 and 5), calculation (queries 2, 3, 6) and update (query 7) types, execution times rose with the size of the dataset.

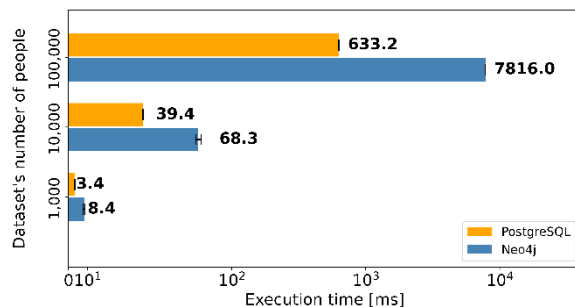


Figure 6: Average execution time for one of the traversal type queries – Descendant Retrieval

Results show that for the largest used dataset and for the calculation type query (query 6 – Youngest Parent Search) average execution times (Figure 7) for both database types were not significantly different. Similar results were observed for scenario 3; however, the analysis still showed that results for Neo4j were worse if only ever so slightly.

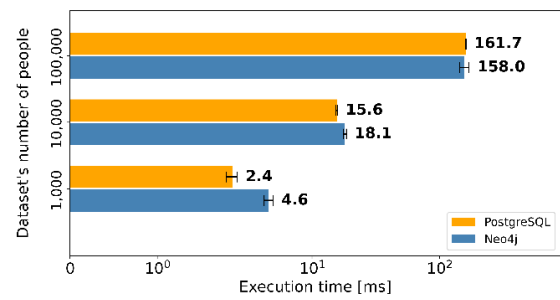


Figure 7: Average execution time for query 6 – Youngest Parent Search

Data modification query results (queries 8 and 9 – Family Insertion and Removal), while continuing the trend of faster execution times for PostgreSQL calls, suggest that there is little to no correlation between execution time of these types of operations and the current size of the dataset in the database (Figure 8).

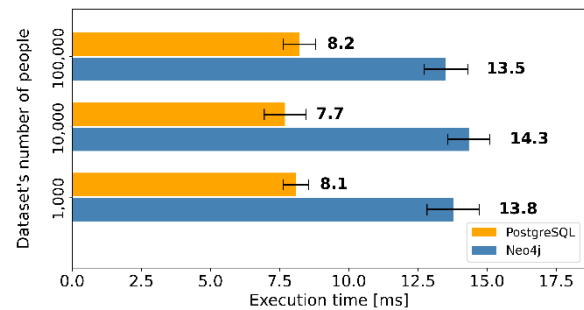


Figure 8: Average execution time for query 8 – Family Insertion

## 5.3. Memory usage

Neo4j's preallocated memory exceeds PostgreSQL dynamic memory usage in all scenarios. On average PostgreSQL never used more than 800 MB of memory, while Neo4j started with around 3,775 MB preallocated as per JVM settings. However, a comparison is still somewhat possible if based on Neo4j's heap allocation inside of JVM.

Similarly to execution time results (Figure 9), where PostgreSQL proved to be generally more efficient, memory usage of SQL-based database proved to be generally lower if comparing to average heap size of Neo4j.

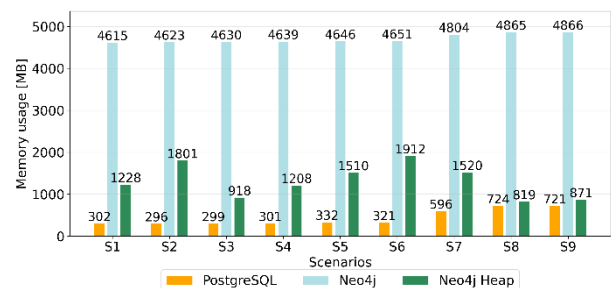


Figure 9: Average memory usage for the largest dataset

PostgreSQL's average memory usage reached its peak during data modification queries for all dataset sizes, while Neo4j's peaks differed between datasets.

## 6. Discussion

The obtained results make it possible to evaluate performance differences between the tested databases across different scenarios in terms of execution times, as well as scalability, RAM usage, and disk space consumption.

Regarding Q1, the results indicate that the PostgreSQL database generally achieved shorter response times for every type of query. The only exception is the case in scenario 6 for the 100,000-record dataset, where the times are comparable. The better performance of PostgreSQL may be due to its advanced query optimizer, as well as its more efficient use of indexes. The difference in execution times is most visible in traversal queries, especially for the largest dataset. Such queries require exploring multiple paths to the same node in order to find the shortest one. This is characteristic of genealogical data, where individuals often share common ancestors. The results suggest that queries written in pure Cypher for Neo4j handle this type of workload less efficiently. In the preparation phase, it was observed that further increasing the depth would eventually prevent the Neo4j query from being executed altogether due to exceeding transactional memory limits. As for the import time, the worse result for Neo4j is due to the inherent overhead of starting the JVM for the neo4j-admin tool.

Regarding Q2, the differing scalability behaviour across scenarios may suggest that this depends on the type of query. For traversal scenarios (scenarios 1, 4, 5), increasing query depth causes a superlinear growth in execution time in both databases. However, Neo4j with its pure Cypher appears to perform less well at larger scales. For aggregation, youngest parent search, and city name update (scenarios 3, 6, and 7), a sublinear to linear increase is observed, which appears to be due the moderate cost of this type of queries. The decreasing differences in execution time for scenarios 3 and 6 between the databases may suggest that, for very large datasets, Neo4j may match PostgreSQL and potentially even outperform it in these scenarios. For the remaining scenarios (2, 8, 9) and the data import, execution time is independent of dataset size or behaves sublinearly. This results from performing the same operation regardless of dataset size for insert and delete operations (scenarios 8 and 9) or the use of B-tree indexes for filtering operations (scenario 2). For the data import, execution time increases more gradually for Neo4j, which suggests that its main cost is the JVM overhead of the database, and the dataset growth itself has less impact on the execution time of the operation.

Regarding Q3, PostgreSQL, which generally achieves shorter response times than Neo4j, requires significantly less RAM usage in most cases, but slightly more disk space for the database. Thus, shorter response time does not necessarily imply higher resource consumption. High RAM usage in Neo4j results from its architecture, where it is recommended to allocate a fixed JVM heap size even if not fully utilized. The size of Neo4j's JVM heap memory alone, used only

as a reference value, is still higher than the total memory usage of the competing database in most cases. In contrast, PostgreSQL does not require pre-allocation and can optimize memory usage more efficiently, which significantly reduces its consumption. The percentage difference between the database disk sizes decreases with scaling. However, the absolute difference remains constant (~7 MB), which suggests a fixed structural overhead in PostgreSQL, likely resulting from its relational storage model and indexes.

The results of the study are consistent with mixed findings in the literature, where different studies report varying outcomes due to differences in tested data structures and query types. In this case, the results align with the findings where relational databases achieved better overall performance than non-relational/graph databases. Existing research in the field of genealogy is limited, outdated, and incomplete. Therefore, the study provides valuable insight into the performance of databases for this type of queries. These results may be useful for software engineers developing genealogical systems and provide a basis for future research in this area.

The results of the study should not be treated as a definitive indication of which system to choose for a genealogical system, as it contains certain limitations that require further research. First and foremost, the study did not consider the use of database extensions, e.g. the APOC plugin for Neo4j. These extensions were used by certain authors in previous studies [16] and could potentially improve performance, especially in traversal scenarios. Secondly, the measurements included driver overhead, deserialization, and data mapping in the application. A minor impact on the results may also have been introduced by the use of the intermediate *Family* node. Another commonly used model employs direct relationships between children and parents. A minimal impact may also have been caused by the simultaneous measurement of RAM during execution time measurement, as well as a limited number of warm-up and measurement iterations.

## 7. Conclusions

The aim of this study was to compare the efficiency of selected relational and non-relational databases for genealogical queries, which was successfully achieved.

The results suggest that PostgreSQL achieves shorter execution times for every type of scenario, except for scenario 6, the youngest parent search for the 100,000-record dataset, where the times are comparable. The scalability behaviour depends on the scenario. The faster PostgreSQL database uses less RAM but slightly more disk space for every dataset size.

The study provides a foundation for further research and offers valuable insights for developers of genealogy services and programs, regarding not only performance but also the data model.

The research certainly did not cover all possible aspects. Further work could include the previously mentioned examination of a model with direct relation-

ships between individuals, as well as the use of add-ons and extensions for databases. It would be valuable to investigate execution times under cold cache conditions, both with and without the use of indexes, as well as to measure the execution time of the database engine itself. The study could also be extended to include other types of relational and non-relational databases, such as MongoDB or Apache Cassandra, as well as the use of real historical data.

## References

- [1] J. T. Harviainen, B.-C. Björk, Genealogy, GEDCOM, and popularity implications, *Informaatiotutkimus* 37(3) (2018) 4–14, <https://doi.org/10.23978/inf.76066>.
- [2] The GEDCOM Standard Release 5.5.1, <https://gedcom.io/specifications/ged551.pdf>, [25.05.2026].
- [3] P. Novotný, J. Wild, The relational modeling of hierarchical data in biodiversity databases, *Database* 2024 (2024) baee107, <https://doi.org/10.1093/database/baee107>.
- [4] Storing Hierarchical Data in Relational Databases with SQL, [https://adamdjellouli.com/articles/databases\\_notes/03\\_sql/09\\_hierarchical\\_data](https://adamdjellouli.com/articles/databases_notes/03_sql/09_hierarchical_data), [25.05.2026].
- [5] W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy, B. Luo, SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review, *Big Data and Cognitive Computing* 7(2) (2023) 97, <https://doi.org/10.3390/bdcc7020097>.
- [6] FamilySearch: Huge Online Genealogical Database Driven by Cassandra, <https://www.youtube.com/watch?v=6XP62FPJUmK>, [25.05.2026].
- [7] A. Mhedhbi, P. Gupta, S. Khaliq, S. Salihoglu, A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems, In 2021 IEEE 37th International Conference on Data Engineering (ICDE) (2021) 1464–1475, <https://doi.org/10.1109/icde51399.2021.00130>.
- [8] D. A. Stumpf, Graphs for Genealogists, *Journal of Genetic Genealogy* 10(1) (2022) 101.003.
- [9] DevOps at Findmypast: Postgres Database Upgrades, <https://tech.findmypast.com/postgres-database-upgrades>, [25.05.2026].
- [10] GRAPHical Family Trees, <https://tech.findmypast.com/graphical-family-tree>, [25.05.2026].
- [11] How AI Transformed Our PHP Upgrade Journey, <https://medium.com/myheritage-engineering/how-ai-transformed-our-php-upgrade-journey-c4f96a09c840>, [25.05.2026].
- [12] Scaling Ancestry.com: Providing Billions of Personalized Hints, <https://medium.com/ancestry-product-and-technology/scaling-ancestry-com-providing-billions-of-personalized-hints-2f1866b50001>, [25.05.2026].
- [13] SQLite4RootsMagic Wiki, <https://sqlite4rootsmagic.groups.io/g/main/wiki>, [25.05.2026].
- [14] Using database API – Gramps, [https://www.gramps-project.org/wiki/index.php/Using\\_database\\_API](https://www.gramps-project.org/wiki/index.php/Using_database_API), [25.05.2026].
- [15] M. Lazarska, O. Siedlecka-Lamch, Comparative study of relational and graph databases, In 2019 IEEE 15th International Scientific Conference on Informatics (2019) 000363–000370, <https://doi.org/10.1109/Informatics47936.2019.9119303>.
- [16] K. Srivastava, D. Jain, A. Kamdar, A. Yeole, D. Shah, S. Dadheech, Adaptivity in Role-Based Access Control During Stochastic Situations: A Comprehensive Study between Graph and Relational Databases, *Journal of Computer Science* 20(12) (2024) 1744–1752, <https://doi.org/10.3844/jcssp.2024.1744.1752>.
- [17] F. Gong, Y. Ma, W. Gong, X. Li, C. Li, X. Yuan, Neo4j graph database realizes efficient storage performance of oilfield ontology, *PLOS ONE* 13(11) (2018) 1–16, <https://doi.org/10.1371/journal.pone.0207595>.
- [18] C. A. Györödi, D. V. Dumșe-Burescu, D. R. Zmaranda, R. Ș. Györödi, G. A. Gabor, G. D. Pecherle, Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application's Data Storage, *Applied Sciences* 10(23) (2020) 8524, <https://doi.org/10.3390/app10238524>.
- [19] D. Yedilkhan, A. Mukasheva, D. Bissengaliyeva, Y. Suynullayev, Performance analysis of scaling NoSQL vs SQL: A comparative study of MongoDB, Cassandra, and PostgreSQL, In 2023 IEEE International Conference on Smart Information Systems and Technologies (SIST) (2023) 479–483, <https://doi.org/10.1109/SIST58284.2023.10223568>.
- [20] V. Filatov, P. Flis, B. Pańczyk, Storage efficiency comparison of UML models in selected database technologies, *Journal of Computer Sciences Institute* 12 (2019) 193–198, <https://doi.org/10.35784/jcsi.437>.
- [21] P. C. Soares, R. Agra, F. N. B. De Souza, A. L. Queiroz, A comparative study between the performance of relational and graph databases applied to hierarchical queries, In 2023 18th Iberian Conference on Information Systems and Technologies (CISTI) (2023) 1–6, <https://doi.org/10.23919/CISTI58278.2023.10211435>.
- [22] P. Kotiranta, M. Junkkari, J. Nummenmaa, Performance of Graph and Relational Databases in Complex Queries, *Applied Sciences* 12(13) (2022) 6490, <https://doi.org/10.3390/app12136490>.
- [23] P. Setialana, T. B. Adji, I. Ardiyanto, Perbandingan Performa Relational, Document-Oriented dan Graph Database Pada Struktur Data Directed Acyclic Graph, *Jurnal Buana Informatika* 8(2) (2017) 77–86, <https://doi.org/10.24002/jbi.v8i2.1079>.