A Comprehensive Analysis of Relational and Graph Models

Ana Milroy - 55505 and Federica Di Filippo - 68911

Nova School of Science and Technology, Lisbon, Portugal

Abstract Relational and graph databases find widespread applications across various contexts. This paper aims to delve into their uses, advantages, and drawbacks, drawing insights from literature reviews and practical experiments. The experiments yielded insights into the influence of the quantity of tables involved and the incorporation of aggregations in queries, employing both SQL and Cypher.

Keywords: Realtional database · Graph database · performance.

1 Introduction

In an era dominated by big data, data analytics, and business intelligence, effective database management plays a crucial role in both technical business operations and research. Database management has been a subject of active research for many years, yielding various types of database management systems. Among these, the Relational Database Management System (RDBMS) has stood out as the most widely used in academic and industrial settings.

However, in recent times, there has been a growing interest in graph databases. This renewed attention can be attributed to the inherent properties of graphs, which naturally represent strong connectivity within data structures.

In this paper we will provide insights into both relational and graph models, analyzing their strengths, their shortcomings, and their uses. Through this work, our goal is to facilitate the decision-making process for determining the most suitable option in various database scenarios.

1.1 Paper structure

This paper undertakes a thorough exploration of relational and graph databases, analysing their advantages, disadvantages, and respective applications. Starting with an introductory chapter outlining the problem and overarching goals, subsequent sections delve into the specifics. Chapters 2 and 3 elucidate the intricacies of relational and graph databases, highlighting their strengths and weaknesses. In Chapter 4 we present a comparison, focusing on aspects such as data modeling, query performance, and scalability. Chapter 5 establishes a set of criteria for selecting the appropriate database based on factors such as data nature, query complexity, and number of relationships. In Chapter 6, a specific dataset is implemented in PostgreSQL and Neo4j, and queries are tested. The following chapter, Chapter 7, discloses the results and provides an analysis, offering insights into the databases' real-world performance. Finally, in Chapter 8, the paper concludes by synthesizing the experiment-derived findings on database use and what factors impact performance the most.

2 Relational Databases

Relational databases are a type of database management system (DBMS) that organizes and stores data in a structured format, making it easy to manage and retrieve information. The relational model was first introduced by Codd, E. in [1]. The fundamental concept behind relational databases is the use of tables to store data, where each table consists of rows and columns. These tables are interconnected through predefined relationships, forming a logical and efficient framework for storing and managing data [1].

In a relational database, each table represents an entity, such as customers, products, or orders, and each row in the table represents a specific instance or record of that entity. Columns in the table define the attributes or properties of the entity, such as a customer's name, address, or order date. The relationships between tables are established through keys—typically, a primary key in one table corresponds to a foreign key in another, creating links between related data [2] [15].

Codd, E. also defined 12 essential rules that an ideal relational model should follow [15] [17]. The rules stated are as follows:

- **Information rule:** All information should be stored in a tabular format and all data must be a value of a table's cell.
- **Guaranteed access rule:** Every data element must be accessible logically using only combinations of table names, primary keys, and attribute names.
- **Systematic treatment of NULL values:** NULL values should be supported and treated uniformly.
- **Active online catalog:** A description of the database should be available online for authorized users.
- **Comprehensive data sub-language rule:** A database should support a clearly defined language to define and view the database, manipulate the data, and restrict some data values to maintain integrity.
- **View updating rule:** All updatable views should be able to be updated by users.
- **High-level insert, update, and delete rule:** All data must be able to be updated, inserted, and deleted.
- **Physical data independence:** Data must be independent of the application accessing the data, meaning that changes in how data is stored or retrieved should not affect how the data is accessed.
- **Logical data independence:** An application's view and access to the data should not be dependent on the storage format of the data.

- **Integrity independence:** Constraints on user input should exist to maintain data integrity.
- **Distribution independence:** The database design should allow for a distribution of data across multiple machines without affecting the accessing application.
- **Non-subversion rule:** If a low-level interface that provides access to records exists then it must not be able to bypass security and integrity constraints

Relational databases are widely used in various applications and industries as they provide a solid foundation for handling complex relationships between different types of data and are essential for applications ranging from enterprise-level systems to web-based applications. Popular relational database management systems include MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server, and SQLite, each offering its own features and advantages for different use cases.

2.1 Strengths

One key advantage of relational databases lies in their adept enforcement of data integrity and consistency [19]. Beyond the structural definitions provided by constraints such as primary keys, foreign keys, not NULL constraints, and unique constraints, relational databases align seamlessly with the fundamental ACID principles (Atomicity, Consistency, Isolation, Durability). These principles fortify the database's robustness: atomicity ensures indivisible and complete transactions, consistency is maintained through constraints preventing anomalies, isolation prevents interference between concurrent transactions, and durability guarantees persistence even in the face of system failures. This integration of constraints and ACID principles establishes a reliable foundation for managing data with precision, reliability, and resilience against potential disruptions [17].

Another main advantage of utilizing relational databases is the incorporation of the Structured Query Language (SQL), a standardized language that earned recognition as an American National Standards Institute (ANSI) standard in 1986. SQL enjoys universal support across popular relational database engines, contributing to a seamless adaptability to various systems. The adoption of this ubiquitous language translates into a low learning curve for users, given its widespread use and standardization [9]. As an established language with extensive documentation, robust backing from parent companies, and a large user community, SQL offers a tried-and-tested framework. Its versatility extends to multiple facets of database management, encompassing tasks such as adding, updating, and deleting rows, retrieving data, and executing aggregations and arithmetic operations [9].

Relational models exhibit vertical scalability, a key attribute enabling them to efficiently accommodate increased workloads and larger datasets by enhancing the resources of a single server. This scalability approach involves upgrading the existing hardware components, such as CPU, RAM, or storage, providing a straightforward solution for applications experiencing growth in demand. The ability to vertically scale allows relational databases to handle expanding data volumes and transactional loads without necessitating the complexity of distributed systems [4].

Relational databases prioritize data storage efficiency through the application of normalization techniques. These techniques involve breaking down tables into smaller, related tables, reducing data redundancy, and optimizing storage space. By eliminating duplicate data, relational models enhance efficiency and maintain data consistency. Additionally, support for indexing mechanisms further contributes to storage optimization by expediting data retrieval operations. The emphasis on storage efficiency is particularly beneficial for applications dealing with large datasets, ensuring streamlined storage management and retrieval processes [4].

Relational databases excel in performing aggregations, a critical aspect for analytical processing and generating summarized insights from extensive datasets. The structured nature of relational data and the expressive power of the SQL language facilitate efficient grouping, filtering, and computation of aggregate functions [18]. Operations like GROUP BY in SQL enable the consolidation of data based on specific criteria, allowing for the extraction of meaningful summaries. The inherent support for aggregations positions relational models as robust solutions for applications that require advanced analytics, reporting, and the derivation of key insights from complex datasets[4].

Relational models present advantages when used for the management of distributed data. Its decomposition flexibility allows the effective distribution of large databases across multiple sites, facilitating both the task of distributing databases and the logical decomposition of relational commands. The recomposition power inherent in the relational model enables the seamless combination of data returned from various sites into a unified result. Economically, the model reduces transmission costs significantly by employing basic relational operators and sending concise commands across the network. Finally, distribution independence ensures the protection of users by allowing the smooth transition from a non-distributed version to a distributed one without necessitating changes to application programs. Collectively, these characteristics highlight the relational model's effectiveness when querying, manipulating, and controlling distributed data [17].

2.2 Weaknesses

While SQL serves as a powerful language, the absence of native support for recursion stands out as a notable limitation. This deficiency restricts the language's expressive capabilities and poses challenges when dealing with hierarchical or recursive data structures [3].

Scalability issues emerge as a significant concern, particularly when relational databases contend with large datasets. SQL queries, if not carefully optimized, can strain system and server resources in extensive systems, leading to potential bottlenecks, database locking, and, in severe cases, data loss. The inherent complexity of queries, often involving intricate joins and aggregations, amplifies the risk of slow data accessibility and suboptimal performance, especially in the context of substantial data volumes [4].

The fixed and predefined nature of a relational database's schema adds another layer of constraint. While the structured schema provides stability, it makes adapting to changes a costly and intricate process. Altering the schema to accommodate evolving requirements or incorporating new data types becomes challenging, impeding the database's flexibility and extensibility [5] [20].

Moreover, relational models exhibit certain inadequacies in handling big data scenarios. Their architecture, optimized for structured data and relational queries, may struggle to efficiently process and manage the vast and unstructured datasets characteristic of big data environments. The rigid schema and normalization principles, which contribute to data integrity in structured scenarios, may become cumbersome when dealing with the diverse and dynamic nature of big data [20].

In summary, while relational models offer robust capabilities, their limitations in areas such as recursion, scalability, adaptability to schema changes, and handling big data underscore the need for careful consideration and alternative approaches in specific data management contexts.

3 Graph Databases

Graph databases are a type of database management system designed to efficiently store and retrieve interconnected data represented in the form of a graph structure. Unlike traditional relational databases mentioned above, which rely on tables and relationships, graph databases use graph structures composed of nodes, edges, and properties to model and store information [5].

The commonly used model for graphs in the context of graph databases is the (labeled) property graph model. In this model, a graph is composed by interconnected entities, referred to as nodes, which can have various properties expressed as key-value pairs. Nodes and edges can be assigned labels to represent their different roles in the application. Relationships in this model establish directed and semantically significant connections between two nodes. In every relationship, there is a specified direction, along with a designated starting node and ending node. Both nodes and relationships can have any number of properties, with relationships often having quantitative attributes. Unique identifiers define each node and edge, allowing for efficient navigation of relationships regardless of their direction [13].

Entities, nodes, and relationships constitute the foundational elements of graph databases, proving particularly effective in applications accentuating data interconnectivity, such as social networks and recommendation systems[11].

A salient feature intrinsic to graph databases is the principle of index-free adjacency, denoting that each node maintains direct references to its neighboring nodes. This facilitates efficient local graph queries, diverging significantly from the methodologies of traditional relational databases that often rely on intricate join operations, particularly in the context of complicated data relationships [10].

In contrast to the rigidity inherent in traditional relational databases, graph databases, exemplified by Neo4j, are deliberately optimized for graph structures rather than conventional tabular formats. This optimization results in heightened expressiveness, scalability to accommodate extensive datasets comprising billions of nodes, and a concentrated focus on operations intrinsic to graph structures[11].

Cypher, a declarative query language, has emerged as a prominent tool for interacting with graph databases. It is tailored for property graphs, offering robust capabilities for both querying and modifying data, along with the specification of schema definitions. Notably, Cypher operates on the basis of linear queries, transforming a property graph input into a tabular output. The core principle of Cypher queries revolves around pattern matching, where patterns are visually expressed like (a)-[r]->(b). What sets Cypher apart is its intentional similarity to SQL, designed to facilitate a smooth transition for users familiar with the latter[16].

Graph databases demonstrate distinct advantages in scenarios involving highly connected data. Their flexibility is underscored by the capacity to dynamically modify schemas without inferring existing functionalities. Performance enhancements are particularly notable as query complexity escalates, with the efficiency of graph queries, especially in traversing relationships, surpassing that of traditional relational databases [12].

When considering the scalability of graph databases, the importance of *Sharding*, also known as Graph Partitioning, becomes crucial. Unlike simpler data structures in other NoSQL databases, the complex interconnections within graph data pose distinctive challenges. *Sharding* emerges as a usefull approach to efficiently distribute graph data across multiple machines. However, accomplishing this task is notably complex due to the intricate nature of relationships within the graph. [10].

3.1 Strengths

Graph database models find application in domains where understanding data interconnectivity or topology is crucial, often being of equal importance to the data itself. In such contexts, where data and relationships hold equal significance, employing graphs as a modelling tool offers several advantages:

- **Natural data modelling:** Graph structures, being visible to users, offer an intuitive way to manage application data, especially in areas such as hypertext or geographic data. The benefit arises from the capability of graphs to contain comprehensive information about an entity within a single node, revealing related information through interconnected links [5].
- **Direct Querying of Graph Structure:** Queries have the ability to directly use the graph structure, exploiting particular graph operations within the query language algebra [5]. These operations encompass tasks like discovering the shortest routes or identifying specific sub-graphs, thereby improving the efficiency and accuracy of retrieving data[13].

- **Flexibility:** The schema and structure of graph models can be dynamically adjusted based on application needs. Data analysts have the freedom to add or modify existing graph structures without disrupting existing functions, eliminating the need for pre-modeling domains.
- **Performance:** Graph databases performance improve as the complexity of relationships increase, in contrast to relational database models, that experience decreasing efficiency. This is particularly evident in querying relationships, where graph database performance scales by several orders of magnitude. Notably, this performance remains consistent even as the volume of graph data increases.
- **Efficiency:** Graph queries outperform relational databases in report generation by being more concise and efficient. Through the use of linked nodes, graph technologies simplify the traversal of joins or relationships, and the advantage lies in the persistence of relationship calculations within the database, resulting in faster query times [14].

These advantages underscore the value of graph analytics, providing adaptability, enhanced performance in handling relationships, and overall efficiency in data retrieval and analysis.

3.2 Weaknesses

Despite lots of research and real-world use of graph data management, there are still some tough problems that affect how graph databases work. Additionally, specific issues arise when dealing with Big Analytics.

- **Data Partitioning:** Many graph databases do not facilitate data partitioning and distribution across a computer network, a critical element for achieving horizontal scalability. Partitioning graphs without requiring multiple accesses for most queries remains a difficult challenge.
- **Model Restrictions:** Graph databases have limitations on data schema and constraints definitions, leading to potential data inconsistencies. Furthermore, the graph model itself may have restrictions, as seen in cases like Neo4j nodes unable to reference themselves directly.
- **Graph Derivation:** Effectively pulling out a graph or a set of graphs from data stores that aren't inherently graphical can be a bit tricky. Graph analytic systems usually work better when you already have a graph, but making one might mean bringing together and merging info from different non-graph sources.
- **High Cost Storage:** Real-world graphs are dynamic and generate data rapidly. Storing historical traces compactly while enabling efficient execution of point queries and global or neighbourhood-centric analysis tasks is a challenge, especially at the scale of Big Analytics. Loading entire historical snapshots into memory for global analytics adds another layer of complexity.
- **Complex Analysis Algorithms:** Real-world situations require graph algorithms that are more intricate than basic k-hop queries. So, graph databases

should be able to handle analytical queries that go beyond straightforward small k-hop queries.
- **Heterogeneous and Uncertain Graph Data:** Effectively querying or analyzing Big Graphs requires automated methods to handle heterogeneity, incompleteness, and inconsistency between different datasets. Semantically integrating diverse Big Graph datasets poses a considerable challenge in this context [10].

4 Comparing relational and graph databases

In the preceding sections, we explored the fundamental characteristics of graph databases and relational databases, recognizing them as two separate categories of database management systems, each designed to address specific challenges in how data is modeled and the requirements for making queries. As we delve into this chapter, our objective is to conduct a comprehensive comparison between these two database types.

Data Modeling:

- Relational Databases: data is structured by organizing it into tables, each with predefined columns and data types. Relationships between different sets of data are represented using foreign keys, which serve as connectors between the tables. This approach is most effective when dealing with structured, tabular data that has clear and well-defined relationships among different elements.
- Graph Databases: data is depicted as a network consisting of nodes, edges, and properties. Nodes symbolize distinct entities, edges signify the relationships between these entities, and properties offer supplementary details. This model is particularly well-suited for situations where relationships among entities are intricate, dynamic, and play a crucial role in the functionality of the application.

Query Language:

- Relational Databases: Usually, SQL (Structured Query Language) is employed for the retrieval and management of data. Queries often require joins and can become intricate when dealing with highly interconnected data.
- Graph Databases: Graph databases utilize dedicated query languages such as Cypher, specifically crafted for articulating patterns and relationships within graphs. The inherent strength lies in the ability to traverse relationships, providing an intuitive approach for certain types of queries.

Performance:

- Relational Databases: Effective for handling intricate queries that involve multiple aggregations, but performance issues may arise when managing data that is deeply nested or highly interconnected.

- Graph Databases: Excelling in navigating and querying relationships, graph databases prove to be efficient for specific types of queries. They can surpass relational databases, especially in situations where relationships take center stage.

Use Cases:

- Relational Databases: Ideal for applications dealing with clearly defined and structured data with fixed relationships. Frequently employed in traditional business applications, financial systems, and data warehousing.
- Graph Databases: Well-suited for applications where a thorough understanding and effective querying of relationships are essential. Widely utilized in domains such as social networks, recommendation engines, fraud detection, and network analysis.

Scalability:

- Relational Databases: A common strategy involves vertical scalability, which entails adding more resources to a single machine. Alternatively, horizontal scalability may be achieved through techniques such as sharding and partitioning.
- Graph Databases: Effective in situations involving highly interconnected data, graph databases can achieve horizontal scalability by distributing nodes and relationships across multiple servers.

Flexibility:

- Relational Databases: Schemas are rigid and changes may be challenging, especially in large systems. Well-suited for scenarios where data structure is unlikely to change frequently.
- Graph Databases: More flexible, allowing for dynamic changes to the data model. Adaptable to evolving requirements, making them suitable for agile development.

In summary, the choice between a graph database and a relational database depends on the nature of the data and the specific requirements of the application. Relational databases are excellent for structured, tabular data with well-defined relationships, while graph databases shine in scenarios where the relationships between entities are dynamic, complex, and central to the application's functionality.

5 Selecting the appropriate database

According to our research, no type of database is intrinsically better in all scenarios. Different use cases and applications require different characteristics in terms of data storage, access, and manipulation. In this chapter, we will analyze and compile the scenarios in which each type of database is more suitable. The gathered information can be found in Table 1. Following this analysis, we will provide a synthesized way of selecting the appropriate database system for a given problem.

Usage scenario	**Relational database**	**Graph database**	**Reference**
Information about data inter-connectivity or topology is important		✓	[5], [4]
Simple structured data	✓		[5], [20]
Database reliability/ACID principles are important	✓		[9],[4], [20]
Requires frequent schema changes		✓	[9], [5]
Need for a high level of support (from parent company and users)		✓	[9]
Database requires distribution	✓		[9], [17], [10]
Unstructured data		✓	[10], [5]
Graph operations are required		✓	[5], [10], [9]
Data represents complex interconnected networks		✓	[5], [9]
Large volumes of data		✓	[21], [5]
Transactional applications	✓		[20], [4]
Many queries involving aggregations are required	✓		[4], [18]

Table 1. Use case scenarios

This summary provides information on what type of database to choose when only one aspect is considered, however, the real challenge is selecting a database when multiple conflicting aspects are present. If, for example, a scenario requires frequent schema changes but ACID principles are crucial, then a careful analysis is required.

First and foremost, it is imperative to clearly define all the requirements of the database along with its intended use. After this step, the selection of a database can be done using a variety of decision-making algorithms [22]. It is important to note that if one of the requirements of the database relies on something that only one type of model can supply, then it is the one that must be chosen. In the following section, we will analyze one example of conflicting aspects to determine the importance of each criterion in a scenario.

6 Experiments

6.1 Dataset used

We aimed to find a dataset that lends itself well to representation in a relational model, while also featuring relationships that can be iterated, making it suitable for a graph model as well. In this experimental phase, we chose a dataset focused on articles, encompassing information about authors, titles, and publication sources [8]. This common dataset served as the foundation for our experiments in both the relational and graph database scenarios.

6.2 Data preprocessing

The dataset posed challenges that required a step of data cleanup and preprocessing. To address entries without a venue name, we introduced an 'unknown' category to prevent null values. Additionally, we removed textual lines masquerading as articles. All cleaning procedures were executed using Python.

Given the dataset's horizontal format, we opted to partition it. We generated separate files for Article, Author, Venue, Published, Reference, and Wrote. Each file comprises the following number of records:

- **Article:** 200000 records
- **Author:** 308375 records
- **Venue:** 3019 records
- **Published:** 200000 records
- **Reference:** 478929 records
- **Wrote:** 617411 records

6.3 Relational database implementation

For the implementation of this dataset in a relational database, our initial step involved creating the corresponding Entity-Relationship (ER) diagram, as depicted in Figure1. In this scenario, the entities do not encompass numerous attributes because the initial dataset contained limited information regarding the author and the venue. However, we opted to create distinct entities in both cases rather than including the values as attributes of the article. We made this decision for two reasons: firstly, there exist many-to-many relationships between articles and authors, so it is preferrable to have separate entities; secondly, this approach ensures that the structure is adaptable and easily modifiable in the presence of datasets containing additional information. In fact, it would suffice to add columns related to new attributes rather than recreating the entire structure.

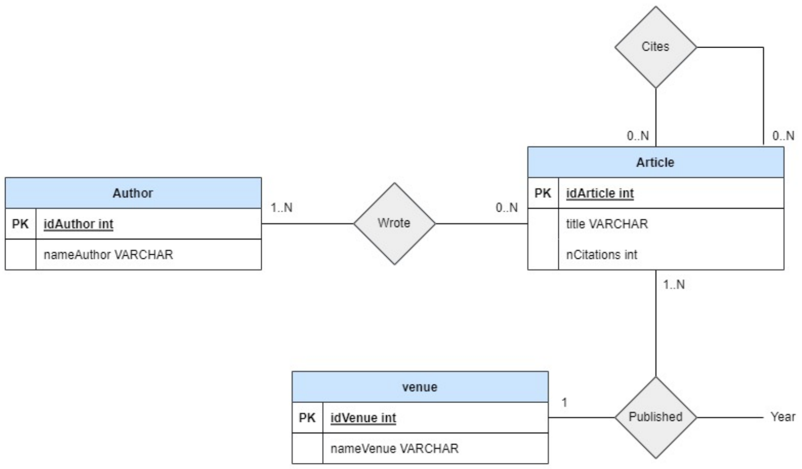

Fig. 1. Entity-Relation Diagram for the article dataset

From this diagram, we formulated the tables needed for the correct implementation following the appropriate conversion rules. According to the multiplicity of the relationships in Figure1, there was a need to introduce one extra table per relationship as represented in Figure 2.

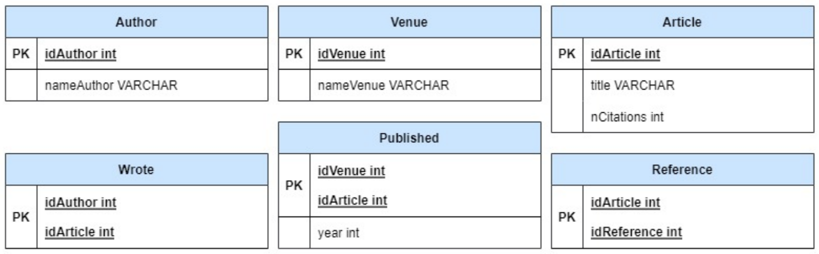

Fig. 2. Necessary tables for the implementation of the dataset as a relational database

6.4 Graph database implementation

To transpose the dataset into a graph database format, our initial step involved designing its graphical representation to identify the appropriate nodes and relationships. The diagram illustrating this representation is presented in Figure 3. This visual depiction not only served as a guide for structuring the graph but also assisted in identifying key entities and connections within the dataset.

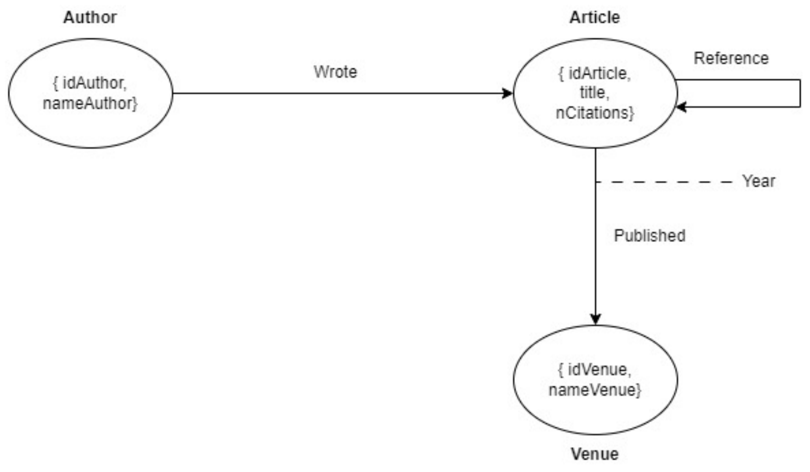

Fig. 3. Graph database representation of the article dataset

6.5 Setup

For this project, we chose two distinct software tools, one for each type of database. To work with relational databases we opted for PostGreSQL 16 version 7.8 [6] due its better performance and more complete analysis tools when compared to other available software. On the other hand, for the graph database we used Neo4j enterprise version 5.14.0 [7] due to our familiarity with it. The machine we used to run the tests has a Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00 GHz processor with 8,00 GB (7,83 GB usable) of RAM.

6.6 Test queries

In this section, we present a compilation of queries that intend to substantiate or dispute the claims made in previous sections. By presenting queries that theoretically outperform one another for each database type when applied to identical datasets, we are able to compare the scenarios and determine if the predetermined selection criteria for each database type hold true. To get a proper cover of all scenarios we provide:

– queries that should perform better on relational databases,
– queries that should perform better on graph databases,
– queries that should perform similarly on both database types,
– queries that can only be done in one type of database

The queries for the relational database will be written in the SQL language while the graph database queries will be written in cypher.

In our research, we verified that for SQL queries the number of joins has a big influence on performance. This means that a query with many joins in SQL should perform worse than a similar query in Cypher. While Cypher has the upper hand regarding joins, it tends to perform worse when using aggregations.

With this in mind, we came up with queries that fulfilled different test conditions as seen in Table 2. Another aspect to consider are queries that rely on graph operations, like traversing a path. Since SQL does not have the expressiveness required for recursion, it is impossible to determine reachability in PostgreSQL. If we limit the path traversing operations to a low threshold, although costly, it is possible to do in SQL. For this experiment we opted for a threshold of 2 to be able to compare the performance between database systems. For each developed query we defined a series of expected results which we intend to confirm or refute through the experiments.

- **Query 1:** test if a simple query with no joins and no aggregations performs similarly on both systems.
- **Query 2:** we expect that the addition of an agreggation to a simple query with a small number of joins will lead to a better performance of PostgreSQL.
- **Query 3:** we expect that this query will perform better oon Neo4j due to the high cost of SQL joins.
- **Query 4:** in this case we are testing which factor (high number of joins or aggregation use) has a bigger impact on performance.
- **Query 5:** we expect Neo4j to perform better in this scenario as it is a graph operation
- **Query 6:** this query tests if a graph database in fact, performs better for path traversing queries.

Description	Query
Small number of joins and no aggregation	1
Small number of joins and aggregation	2
Large number of joins and no aggregation	3
Large number of joins and aggregation	4
Traversing a path	5,6

Table 2. Comparison parameters and corresponding queries

1 - Find all articles published in 1994

SQL

```
SELECT *
FROM "Article"
WHERE "idArticle" IN (
SELECT "idArticle"
FROM "Published"
WHERE year = 1994
);
```

CYPHER

```
profile MATCH (a:Article)-[p:Published]->(v:Venue)
where p.year = 1994
RETURN a;
```

2 - For each venue, find the average number of articles written by each author

SQL

```
SELECT
    subquery."idVenue",
    subquery."idAuthor",
    AVG(subquery.article_count) AS avg_articles_per_author_per_venue
FROM (
    SELECT
        p."idVenue",
        w."idAuthor",
        COUNT(p."idArticle") AS article_count
    FROM
       "Wrote" w
    JOIN
        "Published" p ON w."idArticle" = p."idArticle"
    GROUP BY
        p."idVenue", w."idAuthor"
) AS subquery
GROUP BY
    subquery."idVenue", subquery."idAuthor";
```

CYPHER

```
MATCH (author:Author)-[:Wrote]->(article:Article)-[:Published]->(venue:Venue)
WITH venue, author, COUNT(article) AS articlesCount
RETURN venue.idVenue AS idVenue, author.idAuthor AS idAuthor,
AVG(articlesCount) AS avgArticlesPerAuthor;
```

3 - Find all articles written by all the authors that have also written an article that references 'Scheduling slack time in fixed priority preemptive systems'

SQL

```
SELECT
    au."idAuthor",au."nameAuthor", array_agg(a."idArticle")
    AS groupedArticles
FROM
    "Article" a
JOIN
    "Wrote" w on a."idArticle" = w."idArticle"
JOIN
    "Author" au on au."idAuthor" = w."idAuthor"
WHERE
    au."idAuthor" in (
        SELECT DISTINCT
            au."idAuthor"
        FROM
            "Author" au JOIN "Wrote" w
            ON au."idAuthor" = w."idAuthor"
        JOIN
            "Article" a ON w."idArticle" = a."idArticle"
        JOIN
            "Reference" r ON r."idArticle" = a."idArticle"
JOIN "Article" art ON r."idReference" = art."idArticle"
        WHERE
            art."title" = 'Scheduling slack time in fixed priority pre
            -emptive systems')
GROUP BY au."idAuthor"
```

CYPHER

```
MATCH (au:Author)-[:Wrote]-(a:Article)-[:Reference]-(r:Article )
WHERE r.title ='Scheduling slack time in fixed priority pre
                -emptive systems'
MATCH (au)-[:Wrote]-(ar:Article)
RETURN au.idAuthor, collect(ar.idArticle);
```

4 - Return one row per author, including the author's ID and name, the total number of distinct articles written by that author, the average number of citations for those articles, and the year of the author's latest article published

SQL

```
SELECT
    a."idAuthor",a."nameAuthor",total_articles.totalArticles,
    avg_references.avgReferences,
    latest_publication.latestPublicationYear
FROM
    "Author" a
JOIN (
    SELECT w."idAuthor", COUNT(DISTINCT w."idArticle") AS totalArticles
    FROM "Wrote" w
    GROUP BY w."idAuthor"
) total_articles ON a."idAuthor" = total_articles."idAuthor"
JOIN (
    SELECT
        w."idAuthor", AVG(reference_count.referenceCount) AS avgReferences
    FROM
        "Wrote" w
    JOIN
        "Article" ar ON w."idArticle" = ar."idArticle"
    LEFT JOIN (
        SELECT
            "idArticle", COUNT(DISTINCT "idReference") AS referenceCount
        FROM
            "Reference"
        GROUP BY
            "idArticle"
    ) reference_count ON ar."idArticle" = reference_count."idArticle"
    GROUP BY w."idAuthor"
) avg_references ON a."idAuthor" = avg_references."idAuthor"
JOIN (
    SELECT
        w."idAuthor", MAX(p."year") AS latestPublicationYear
    FROM
        "Wrote" w
    JOIN
        "Article" ar ON w."idArticle" = ar."idArticle"
    LEFT JOIN
        "Published" p ON ar."idArticle" = p."idArticle"
    GROUP BY
        w."idAuthor"
) latest_publication ON a."idAuthor" = latest_publication."idAuthor";
```

CYPHER

```
MATCH (a:Author)-[:Wrote]->(article:Article)-[p:Published]->(venue:Venue)
OPTIONAL MATCH (article)-[:Reference]->(refArticle:Article)
WITH a,
     COUNT(DISTINCT article) AS totalArticles,
     COUNT(DISTINCT refArticle) AS referenceArticles,
     MAX(p.year) AS latestPublicationYear
WITH a, totalArticles, referenceArticles, latestPublicationYear
WITH a, totalArticles, AVG(referenceArticles) as avgReferences, latestPublicationYear
RETURN a.idAuthor AS idAuthor, a.nameAuthor AS nameAuthor,
       totalArticles, avgReferences, latestPublicationYear;
```

5 - Articles that reference other article with maximum path of two

SQL

```
SELECT
    A."title" AS "Article", B."title" AS "Reference"
FROM
    "Reference" R1
JOIN
    "Reference" R2 ON R1."idReference" = R2."idArticle"
JOIN
    "Article" A ON R1."idArticle" = A."idArticle"
JOIN
    "Article" B ON R2."idReference" = B."idArticle";
```

CYPHER

```
match(a:Article)-[r:Reference *2]->(a2:Article)
return a.idArticle, a2.idArticle
```

6- Articles that reference other article with any length path

SQL For this type of query, SQl does not have enough expressive power to process it.

CYPHER

```
match(a:Article)-[r:Reference*..]->(a2:Article)
return a.idArticle, a2.idArticle
```

7 Results

To correctly compare the results of running the queries, we took into consideration the cold start as well as the average of the six following runs. The results from this experiment can be found in table 3.

The execution times from Neo4j were obtained from the logs of each query. All execution times stated in this section are presented in seconds.

	PostgreSQL			Neo4j	
Query	Number of joins	Cold start	Avg execution time	Cold start	Avg execution time
1	0	0,313	0,1622	0,057	0,0144
2	1	2,267	2,2094	9,371	8,5928
3	5	0,439	0,2948	0,0138	0,0084
4	7	3,449	3,1862	5,014	4,3182
5	3	4,741	4,4844	0,524	0,349
6	-	-	-	1,367	0,4158

Table 3. Table showing the results of running each query on both software systems

After an analysis of the results in table 3, we determined which model was favorable for each query based on both cold start and average execution time. Both queries in cold start and the average of subsequent queries lead to the same conclusions. A summary of the analysis can be found in table 4.

Query	PostgreSQL	Neo4j
1		✓
2	✓	
3		✓
4	✓	
5		✓
6		✓

Table 4. Summary table of the comparison between query performance

8 Conclusions

In our study, we set out expectations regarding the outcomes of our experiment, and the results we obtained align with those initial expectations. Our anticipated outcomes were based on our study of the structure of each database and how this structure works to answer queries.

As we compare the results of a simple query with no joins or aggregations (query 1), we can see that the execution times are quite similar. Although Neo4j performed slightly better, both systems are adequate for these types of queries.

Taking into account both the queries with a low number of joins (query 1 and query 2), we can conclude that a simple aggregation is enough to make PostgreSql perform better than Neo4j. The same scenario is presented when considering a higher number of joins (query 3 and query 4). Query 3 highlights that when there is no aggregation required, neo4j outperforms PostgreSQL significantly, showing the power of graphs when considering relationships. On the other hand, query 4, which contains an aggregation, inverts this tendency. In this query, the discrepancy in performance is smaller than in query 2 as it is balanced by the higher cost of joins in PostgreSQL, however, it is still present. This demonstrates that in a system that requires many aggregations, neo4j might not be the ideal choice.

Our results show that when deciding which model to use, a higher priority should be given to the number of aggregations and their complexity when compared to the number of joins when considering performance. However, it is important to note that this experiment was done with a maximum number of 7 joins. So when considering a significantly higher number of joins, this conclusion might not remain true.

Regarding queries 5 and 6, which rely on path traversal operations, a graph-based system is clearly more appropriate. As seen in query 6, SQL does not have the ability to express recursion, and therefore, if a system requires this operation then graph-based models are the best choice.

In conclusion, our thorough analysis of relational and graph models reveals that neither type of model universally outshines the other. Each scenario demands a meticulous evaluation, akin to our experiment, to select the most suitable model based on the specific requirements and values of the given context and problem at hand. This nuanced approach, considering the strengths and weaknesses of both relational and graph models, underscores the importance of tailored analysis when making informed decisions in database model selection.

References

1. Codd, Edgar F. "A relational model of data for large shared data banks." Communications of the ACM 13.6 (1970): 377-387.
2. Revesz, Peter. "Introduction to databases." Texts in Computer Science (2010).
3. Libkin, Leonid. "Expressive power of SQL." Theoretical Computer Science 296.3 (2003): 379-404.

4. Khan, Wisal, et al. "SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review." Big Data and Cognitive Computing 7.2 (2023): 97.
5. Angles, Renzo, and Claudio Gutierrez. "Survey of graph database models." ACM Computing Surveys (CSUR) 40.1 (2008): 1-39.
6. "Windows Installers." PostgreSQL, www.postgresql.org/download/windows/. Accessed 8 Dec. 2023.
7. "Download Neo4j Desktop." Graph Database & amp; Analytics, 31 Oct. 2023, neo4j.com/download/.
8. Mohammed, Nechba. "Research Papers Dataset." Kaggle, 8 May 2023, www.kaggle.com/datasets/nechbamohammed/research-papers-dataset.
9. Vicknair, Chad, et al. "A comparison of a graph database and a relational database: a data provenance perspective." Proceedings of the 48th annual Southeast regional conference. 2010.
10. Jaroslav Pokorný. Graph Databases: Their Power and Limitations. 14th Computer Information Systems and Industrial Management (CISIM), Sep 2015, Warsaw, Poland. pp.58-69, ff10.1007/978-3-319-24369-6_5ff. ffhal-01444505f
11. R. kumar Kaliyar, "Graph databases: A survey," International Conference on computing, Communication & Automation, Greater Noida, India, 2015, pp. 785-790, doi: 10.1109/CCAA.2015.7148480.
12. Barceló Baeza, Pablo. "Querying graph databases." Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems. 2013.
13. Angles, Renzo, and Claudio Gutierrez. "Survey of graph database models." ACM Computing Surveys (CSUR) 40.1 (2008): 1-39.
14. Guia, José, Valéria Gonçalves Soares, and Jorge Bernardino. "Graph Databases: Neo4j Analysis." ICEIS (1). 2017.
15. Sumathi, Sai, and S. Esakkirajan. Fundamentals of relational database management systems. Vol. 47. Springer, 2007.
16. Francis, Nadime, et al. "Cypher: An evolving query language for property graphs." Proceedings of the 2018 international conference on management of data. 2018.
17. Codd, Edgar F. The relational model for database management: version 2. Addison-Wesley Longman Publishing Co., Inc., 1990.
18. Kotiranta, Petri, Marko Junkkari, and Jyrki Nummenmaa. "Performance of graph and relational databases in complex queries." Applied Sciences 12.13 (2022): 6490.
19. Khan, Wisal, et al. "SQL database with physical database tuning technique and NoSQL graph database comparisons." 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). IEEE, 2019.
20. Kumawat, Divya, and Aruna Pavate. "Correlation of NOSQL & SQL Database." IOSR J. Comput. Eng.(IOSR-JCE) 18 (2016): 70-74.
21. Győrödi, Cornelia A., et al. "Performance analysis of NoSQL and relational databases with CouchDB and MySQL for application's data storage." Applied Sciences 10.23 (2020): 8524.
22. Tzeng, Gwo-Hshiung, and Jih-Jeng Huang. Multiple attribute decision making: methods and applications. CRC press, 2011.