INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Usando GraphQL com base de dados em grafo

TIAGO FILIPE NASCIMENTO BARBOSA
Junho de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Using GraphQL with Graph databases

Tiago Filipe Nascimento Barbosa

Dissertation to obtain a Master's degree in Informatics Engineering,
Area of Specialization in Software Engineering

Advisor: Isabel Azevedo

Porto, June 2024

Statement of Integrity

I declare that I have conducted this academic work with integrity.

I have not plagiarized or engaged in any form of improper use of information or falsification of results throughout the process that led to its preparation.

Therefore, the work presented in this document is original and my own, and has not previously been used for any purpose not related to this project in its different phases.

I also declare that I have full knowledge of the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, 29 June 2024

Dedicatory

To my friends and family who supported me from the beginning and believed in me.

To my supervisor, who guided me during this work.

Resumo

Bases de dados em grafos e a linguagem de consulta GraphQL têm testemunhado uma crescente adoção em várias indústrias nos últimos anos. Existe uma potencial sinergia resultante da integração de GraphQL com bases de dados em grafo, o que levanta a necessidade de avaliar e comparar esta colaboração em contraste com a integração de GraphQL com bases de dados relacionais, especialmente em termos de desempenho e manutenibilidade.

Este documento explora as sinergias entre GraphQL e as bases de dados em grafo, aprofundando-se na sua integração, desafios e perspetivas. Os capítulos iniciais estabelecem o contexto, abordando o problema, objetivos, metodologia de pesquisa, considerações éticas e organização estrutural do documento.

De seguida, os conceitos de GraphQL e bases de dados em grafo são explorados, analisando as suas estruturas fundamentais, mecanismos de consulta, organização de dados e cenários ideais de uso. Este entendimento fundamental precede uma investigação detalhada da sua integração, conforme delineado no capítulo do estado da arte. Dentro deste capítulo, é realizada uma revisão sistemática da literatura, onde são delineadas as questões de pesquisa que guiam a exploração das estratégias e considerações de integração. A discussão abrange os principais casos de uso que impulsionam a adoção de GraphQL em conjunto com as bases de dados em grafo, a seleção de tecnologias adequadas para a sua integração mais robusta e a elucidação dos principais desafios que impedem a sua adoção generalizada e implementação bem-sucedida.

Na fase de análise e design, um projeto para migração é selecionado, utilizando o *Analytic Hierarchy Process* (AHP), caracterizado e explicado arquitetonicamente. O processo de migração é então detalhado, incluindo a transformação da modelação de dados e as tecnologias e abordagem utilizadas. O capítulo de implementação descreve as alterações na migração, com especial atenção aos *custom resolvers*, classes do domínio, autenticação e autorização, e ajustes ao projeto inicial.

De seguida, a experimentação e avaliação das soluções é realizada seguindo a abordagem *Goal, Questions, Metrics* (GQM), focando-se na avaliação do desempenho e da manutenibilidade de cada solução. Os resultados são analisados, concluindo que a solução utilizando uma base de dados em grafo foi superior em termos de manutenibilidade e, em alguns casos, em termos de desempenho também. Com um maior número de utilizadores, a solução com base de dados em grafo apresentou melhores resultados gerais. A solução com base de dados relacional foi superior nos casos com um pequeno a médio número de utilizadores. Se as *queries* e domínio não necessitarem de dar uso a relações hierárquicas profundas, então uma base de dados relacional é preferível. Por fim, a dissertação conclui com um resumo das realizações, uma descrição das dificuldades encontradas, uma avaliação das ameaças à validade e sugestões para trabalhos futuros.

Palavras-chave: GraphQL, Base de dados em grafo, Neo4j, Base de dados relacional

Abstract

Graph databases and GraphQL query language have seen increasing adoption across various industries in recent years. There is a potential synergy arising from integrating GraphQL with graph databases, raising the need to evaluate and compare this collaboration in contrast to integrating GraphQL with relational databases, especially in terms of performance and maintainability.

This document explores the synergies between GraphQL and graph databases, delving into their integration, challenges, and prospects. The initial chapters establish the contextual landscape, addressing the problem, objectives, research methodology, ethical considerations, and structural organization of the study.

Next, there's an introduction to the concepts of GraphQL and graph databases, dissecting their fundamental structures, querying mechanisms, data organization, and ideal usage scenarios. This foundational understanding sets the stage for a detailed investigation into their integration, as outlined in the state-of-the-art chapter. Within this chapter, a systematic literature review is conducted where the research questions guiding the exploration of integration strategies and considerations are elucidated. The discussion encompasses primary use cases driving the adoption of GraphQL in conjunction with graph databases, the selection of suitable technologies for seamless integration, and the elucidation of key challenges impeding their widespread adoption and successful implementation.

In the analysis and design phase, a project for migration is selected, using the Analytic Hierarchy Process (AHP), characterized, and architecturally explained. The migration process is then elaborated, including data modeling transformation and the technologies and approaches used. The implementation chapter describes the migration changes, with specific attention to domain classes and custom resolvers, authentication and authorization, and adjustments to the initial project.

Next, the experimentation and evaluation of the solutions is achieved following the Goal, Questions, Metrics (GQM) approach, focusing on evaluating the performance and maintainability of each solution. The results are analyzed, concluding that the solution using a graph database is superior in terms of maintainability and, in some cases, in terms of performance too. With a higher number of concurrent users, the graph database solution presented better results overall. The relational database solution outperformed when using a small to medium number of users. If the queries and domain do not require deep nesting relationships, then the relational database seems more desirable, in terms of performance.

Finally, the dissertation concludes with a summary of achievements, an outline of difficulties encountered, an assessment of threats to validity, and suggestions for future work.

Keywords: GraphQL, Graph database, Neo4j, Relational database

Acknowledgment

First and foremost, I would like to thank everyone who supported and encouraged me during this dissertation project.

I would also like to express my gratitude to my advisor, Professor Isabel Azevedo, who supported me and provided me with insightful feedback. Her expertise and support have guided me through every step of this project.

Finally, I want to thank my family and friends for their unwavering support that made this journey more enjoyable.

Index

List of Figures

List of Tables

List of Code snippets

Acronyms and Symbols

List of Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
ACM	Association of Computing Machinery
AHP	Analytic Hierarchy Process
API	Application Programming Interface
CI	Consistency Index
CR	Consistency Ratio
CRUD	Create, Read, Update and Delete
ERD	Entity Relationship Diagram
GB	Graph Database
GDBMS	Graph Database Management System(s)
GQM	Goal, Questions, Metrics
IOT	Internet of Things
KG	Knowledge Graph
LOC	Lines of Code
REST	Representational State Transfer
RI	Random Index

List of Symbols

λ	Lambda

1 Introduction

This chapter presents the dissertation project developed during the Master's Degree in Software Engineering at Instituto Superior de Engenharia do Porto (ISEP). It starts by contextualizing the studied problem and delineating the objectives necessary to address it. Then, the applied methodologies and ethical considerations are described. Lastly, the document structure is presented.

1.1 Context

The landscape of Graph Database Management Systems (GDBMS) has been evolving rapidly in recent years, gaining significant traction in various industries, including social media, recommendation engines, fraud detection, network analysis, and knowledge graph applications [1]. Popular GDBMS include Neo4j [2] , JanusGraph [3] and Memgraph [4].

On the other hand, GraphQL query language has also gained significant popularity and adoption among developers, particularly in the web and mobile application development communities. Major companies and organizations, including Facebook, GitHub, Shopify, and others, have adopted GraphQL for their APIs [5].

Although they are distinct technologies, both can work together, GraphQL can be used as an API layer to interact with graph databases. Having application data represented as a graph with no additional translations can bring benefits, for instance, the description of Neo4j GraphQL Library mentions "high performance" and increased developer productivity [6] to describe possible benefits, although not providing an actual comparison with other types of database technologies, such as relational databases.

1.2 Problem

There is a congruence between GraphQL's query flexibility and the inherent nature of graph databases. This synergy enables precise and efficient retrieval of interconnected data, effectively resolving issues such as over-fetching or under-fetching data, common in traditional RESTful APIs or standard database systems [7], [8].

Selecting the right technologies, especially when integrating GraphQL with graph databases, is pivotal in determining the success and efficiency of a project. In the realm of GraphQL and graph database integration, the choice of technologies may greatly influence performance, maintainability, and overall development experience. Furthermore, when choosing these technologies, evaluating their compatibility and ease of integration becomes imperative. The selected stack should seamlessly blend GraphQL's querying flexibility with the graph database's capacity to handle interconnected data. The aim is to optimize data retrieval, minimize complexities in schema mapping, foster a smooth development process, and improve project maintainability.

On the other hand, GraphQL does not impose any technology for the data source. A recent study [9] highlighted the need to consider the effects of different database technologies (relational and non-relational) and GraphQL operation in future works.

The integration of GraphQL with graph databases provides a powerful toolkit for developers. It requires a careful evaluation of its advantages and drawbacks, especially compared with GraphQL integrations with other database technologies, regarding performance and maintainability. Understanding its capabilities, performance benefits, complexities, and the need for selecting the right technologies ensures an informed decision-making process, enabling teams to leverage the strengths of this integration while mitigating its potential challenges.

1.3 Objective

The objective is to compare the performance and maintainability of graph and relational databases when GraphQL technology is used and identify its benefits and drawbacks. GraphQL and graph databases have been commonly used in an integrated way for exposing knowledge graphs and aggregating heterogeneous data sources. There is a lack of research on how the integration of these two technologies compares with an integration of GraphQL with a relational database, specifically in use cases where the former is not commonly used.

To explore the use of Graph Databases (GB) with GraphQL, benefits and drawbacks regarding performance and maintainability, a solution using a graph database based on an initial project using GraphQL and relational databases is to be developed and assessed. Preliminary to that, a synthetization of relevant information and research on graph databases, GraphQL, and their integration is to be provided.

Controlled experiment is the main research strategy to apply and measurements to be applied based on the Goal Question Metric (GQM) paradigm [10]. The goal is the objective of this dissertation project mentioned at the beginning of this section. The GQM questions are the following research questions:

RQ_A: What are the performance characteristics of graph databases (GB) compared to relational when GraphQL is used?

RQ_B: What are the maintainability characteristics of graph databases (GB) compared to relational when GraphQL is used?

1.4 Research methodology

To achieve the objectives defined, a methodology was proposed. The first step is to identify the problem, motivation, and explore it through the following tasks:

- **Problem interpretation:** for a better understanding of the problem context and motivations for this work (present in Chapter 1).
- **State of the art:** conduct a systematic literature review on the integration of GraphQL with graph databases, guided by the following research questions (present in Chapter 3):
    - What are the primary use cases and practical applications that drive the adoption of GraphQL in conjunction with graph databases? (present in section 3.2.1)
    - What are some of the most appropriate technologies when integrating GraphQL with graph databases? (present in section 3.2.2)
    - What are the principal challenges and considerations encountered influencing their adoption and successful implementation? (present in section 3.2.3)

- **Analysis of open-source projects and application of multicriteria method:** to select a project that serves as a base for the prototype. The multicriteria method applied was the Analytic hierarchy process (AHP) (present in section 4.1).

Next, the design and development phase, through the following actions:

- **Design of prototype with graph databases based on the selected open-source project:** following robust software engineering practices and considerations found in the literature review (present in sections 4.2).
- **Develop the designed prototype:** with the chosen technologies and proper documentation (present in Chapter 5).

Followed by the experimentation and evaluation phase, performing these tasks:

- **Application of GQM Approach:** identifying the goal, questions and outlining the metrics to be applied and the tools to use to obtain those metrics (present in sections 1.3 and 6.1).
- **Controlled experiment and tests:** perform tests on each solution in a controlled environment, with previously defined tools, obtaining the necessary metrics for comparison and evaluation (present in sections 6.2 and 6.3).
- **Result analysis:** analysis of the results obtained, comparing the developed solution to the initial project, identifying limitations and workarounds/mitigation steps to overcome them (present in sections 6.2, 6.3 and 6.4).

Finally, the conclusion and final thoughts phase where a summary of the research is presented, mentioning difficulties found, future work, and contribution to the fields of study (present in Chapter 7).

1.5 Ethical Considerations

The Code of Ethical Conduct of P.PORTO [11] was followed, since the author of this work is a student of this institution. Most importantly the following point was followed:

- Article 6, point 2.8 of point n). Using ideas, phrases, paragraphs, or complete texts from third parties, colleagues, or authors without citing and referencing the respective sources.

The Software Engineering Code of Ethics and Professional Practice [12] was followed, with more specific attention to points:

- 3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.
    - The goals and objectives set for this work are realistic and feasible within the given constraints, including resources, time, and technological capabilities.
- 3.03. Identify, define and address ethical, economic, cultural, legal and environmental issues related to work projects.
    - Only open-source or free technologies will be considered for possible use, ensuring all legal compliances related to licensing.
    - Only open-source projects will be considered for possible use as a base project, always ensuring all legal compliances related to intellectual property rights and licensing.

The data collected through experiment(s) will be made available in a public repository.

1.6 Organization

This document is composed of the following chapters:

- Introduction: It sets the context and sheds light on the problem, followed by the study's objectives and outlining the chosen research methodology. Finally, ethical considerations are exposed, and the document's structure is emphasized.
- Background: Provides a comprehensive overview of the main concepts under investigation, GraphQL and graph databases.
- State of the art: A systematic literature review is conducted to answer three research questions. This chapter gathers knowledge on the topic of the integration of GraphQL with graph databases, mainly existing practical applications, technologies used and best practices and considerations.
- Analysis and Design: This chapter initially describes the project selection process, with the application of the AHP multicriteria method to select the most suitable base project. Then, explores the base project characteristics and important concepts of its domain. Details the transformation of the base project's relational database to a graph database, its process, and implications.
- Implementation: This chapter showcases the implemented solution. It explains how the transition of the base project to a graph database and functionalities were implemented.
- Experimentation and Evaluation: Using GQM approach, the implemented solution is evaluated, through tests for maintainability and performance. The base project is evaluated too, to then be able to make a comparison between both.
- Conclusion: This is the final chapter of the document, it addresses the achievements, limitations, threats to validity, and future work.

2 Background

This section aims to provide background knowledge about the main concepts approached in this document, GraphQL and graph databases.

2.1 GraphQL

GraphQL is a query language for APIs, and a server-side runtime for executing queries with existing data [13]. It was designed to be more efficient, performant, and provide a better developer experience for working with data in a client-server architecture [14] . GraphQL is data source agnostic, wherein the source of data is not important. Whether sourced from diverse origins such as databases, micro-services, or underlying RESTful APIs, GraphQL maintains a neutral and indifferent approach towards data provenance. This inherent agnosticism towards data sources defines one of GraphQL's foundational principles within its operational framework [14].

2.1.1 Schema and resolve functions

Contrary to the conventional REST architecture that organizes its APIs around endpoints linked to resources, GraphQL takes a different approach. GraphQL APIs are primarily built upon type definitions, outlining data types, their respective fields, and their interconnections within the API. These definitions collectively shape the API's schema, such as the example that can be observed in Figure 1 , which, in GraphQL, is accessed through a unified endpoint [7].

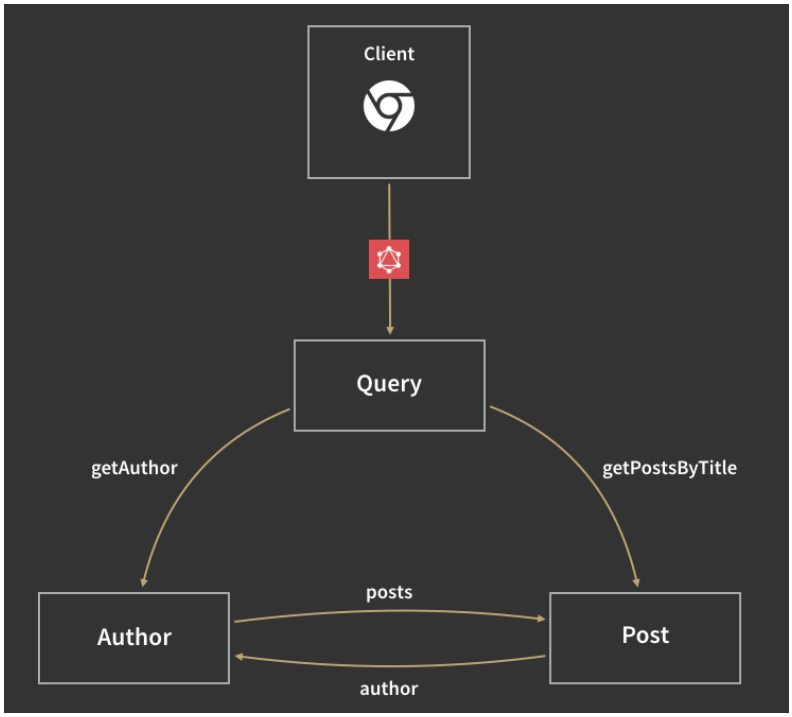

Figure 1 – Example of GraphQL schema
Image from [15]

In Code snippet 1, the same schema example can be seen, but in GraphQL schema notation. The schema defines three types: *Author*, *Post* and *Query*. The latter marks the entry point into the schema, which in this case means that every query must start with either *getAuthor* or *getPostsByTitle* [15]. GraphQL schemas may also contain a *Mutation* type, which defines the entry points for write operations into the API [7].

```
type Author {
  id: Int
  name: String
  posts: [Post]
}type Post {
  id: Int
  title: String
  text: String
  author: Author
}type Query {
  getAuthor(id: Int): Author
  getPostsByTitle(titleContains: String): [Post]
}schema {
  query: Query
}
```

Code snippet 1 – Example in GraphQL schema notation
retrieved from [15]

To conclude, the schema defines what queries and operations clients are allowed to make to the server and the relations between the different types. Additionally, the GraphQL server needs to know how to fetch data for each type, this is where resolve functions step in.

Resolve functions contain the logic for resolving the data for an arbitrary GraphQL request from the data layer, which can be represented in the form of arbitrary code. This allows GraphQL servers to interface with different backend systems, including other GraphQL servers. For instance, while Author data might reside in a SQL database, Posts could be stored in MongoDB or managed through a separate microservice. This adaptability enables GraphQL servers to effectively communicate and retrieve data from diverse backend sources, while hiding all the backend complexity from clients [15]. In Code snippet 2, two examples of resolve functions can be seen.

```
getAuthor(_, args){
  return sql.raw('SELECT * FROM authors WHERE id = %s', args.id);
}posts(author){
  return request(`https://api.blog.io/by_author/${author.id}`);
}
```

Code snippet 2 – Example of resolve functions
retrieved from [15]

2.1.2 Querying

In this subsection, the GraphQL server query execution steps are be identified. In Code snippet 3 an example of a query compatible with the GraphQL Schema showed in the previous subsection is displayed.

```
{
  getAuthor(id: 5){
    name
    posts {
      title
      author {
        name # this will be the same as the name above
      }
    }
  }
}
```

Code snippet 3 – Example of query
retrieved from [15]

The server performs the following three high-level steps to respond to a query:

1. **Parse** - The server parses the query string and transforms it into an abstract syntax tree, while checking for syntax errors [15].

2. **Validate** - In this step, the query is validated against the defined schema. Compared to most REST APIs, where the developer is responsible for making sure parameters are valid, the GraphQL Server performs it automatically [15].
3. **Execute -** Every GraphQL query has the shape of a tree, meaning execution begins at the root of the query. The executor starts by calling the resolve function of the top level fields, waiting for the result before proceeding down the tree [15]. The execution flow of the query presented is displayed in Figure 2.

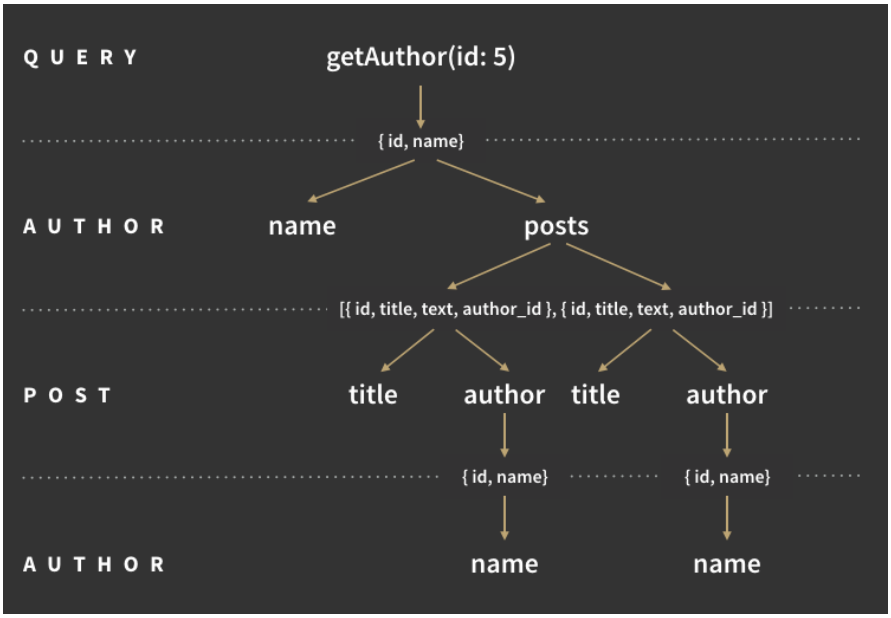

Figure 2 – Diagram of the execution flow
Image from [15]

2.1.3 Mutations

While queries are used to fetch data, mutations are used to modify server-side data. Queries are the GraphQL equivalent to GET calls in REST, while mutations represent the state-changing methods in REST [16].

```
mutation CreateNewPost($title: String!, $text: String!, $authorId: Int!) {
  createPost(title: $title, text: $text, authorId: $authorId) {
    id
    title
    text
    author {
      id
      name
    }
  }
}
```

Code snippet 4 – Example of mutation
based on examples from [16]

The *CreateNewPost* mutation, represented in Code snippet 4, expects values for the *title*, *text* and *authorId* variables. An example of request data for this mutation can be observed in Code snippet 5.

```
{
  "title": "Example Post",
  "text": "This is an example post content.",
  "authorId": 456
}
```

Code snippet 5 – Input example for mutation
based on examples from [16]

And the respective output in Code snippet 6.

```
{
  "data": {
    "createPost": {
      "id": 789,
      "title": "Example Post",
      "text": "This is an example post content.",
      "author": {
        "id": 456,
        "name": "Jane Doe"
      }
    }
  }
}
```

Code snippet 6 – Output example for mutation
based on examples from [16]

Since a mutation must have a resolver on the server side, an example for that is provided in Code snippet 7.

```
const resolvers = {
  ...
  Mutation: {
    createPost: async (root, args, context) => {
      const { title, text, authorId } = args;
      const author = await context.dataSources.postsAPI.findAuthor({
authorId});
      if (!author) {throw new Error('Author not found');}
      const newPost = await context.dataSources.postsAPI.addPost({ title,
text, authorId });
      return {
        id: newPost.id,
        title,
        text,
        author: {
              id: author.id
              name: author.name
      }
    }
  }
};
```

Code snippet 7 – Example of resolver for mutation
based on examples from [16]

2.2 Graph databases

A graph database is a type of database system designed to store and manage data using graph structures. Unlike traditional relational databases that organize data in tables with rows and columns, graph databases employ graph theory principles, where data is represented as nodes, edges, and properties [17]. Because data is stored without restricting it to a pre-defined model, it can often provide better performance and flexibility, since they are more suited for modeling real-world scenarios [17], [18].

2.2.1 Data organization

The components of graph database model are the following:

- **Nodes** - Nodes represent entities or objects, such as people, places, or things. Each node can have various attributes or properties associated with it [17], [18].
- **Edges** - Edges denote the relationships between nodes. They illustrate connections or interactions between entities and can also hold attributes or properties. These

relationships can encompass both one-to-many and many-to-many connections. An edge, inherently, possesses a start node, an end node, a specified type, and a defined direction [17], [18].

- **Properties** - Properties provide additional information or attributes associated with nodes or edges, offering details about the data they represent. Graphs with properties are also called property graphs [17], [18].

An example of a property graph model with its components can be observed in Figure 3.

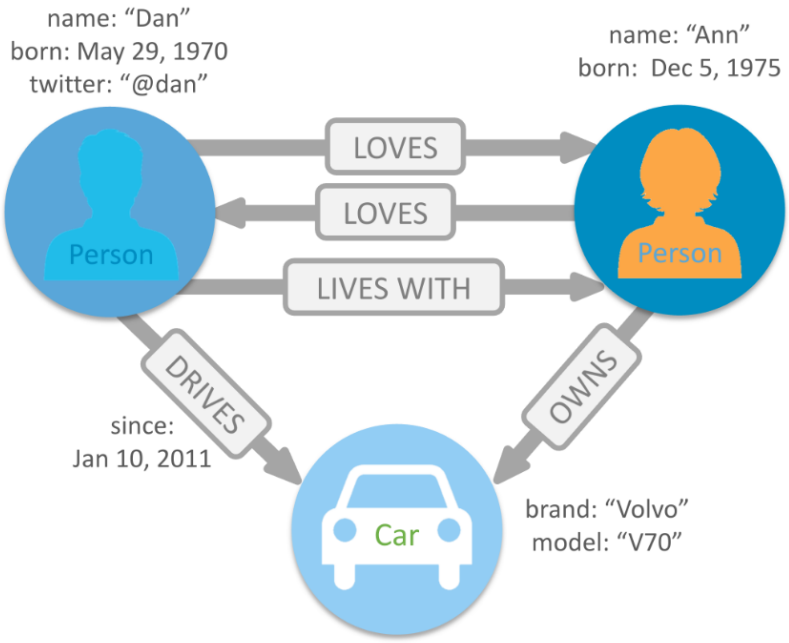

Figure 3 – Example of a property graph model
Image from [18]

The example (Figure 3) has two types of entities (nodes), *Person* and *Car*, each one holding its specific properties (for example, *name*), along with the relationships (edges) between them (e.g . *Person DRIVES Car)* [18].

The structure of graph databases allows for efficient querying of complex relationships, making them particularly useful for scenarios where relationships between data points are crucial [18].

2.2.2 Query Languages

The two main query languages used to access data in graph databases are Cypher, which is declarative and similar to SQL, and Gremlin, a low-level graph traversal language.

Cypher

Cypher is designed to be easily read and understood by developers, database professionals and business stakeholders alike. It's easy to use because it matches the way people intuitively describe graphs using whiteboard-like diagrams [19].

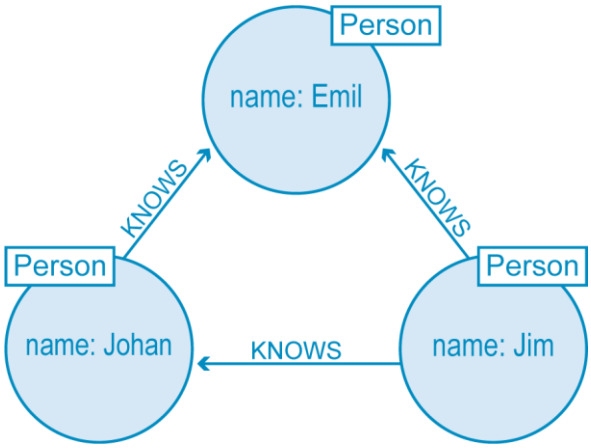

Figure 4- Graph diagram that describes three mutual friends.
Image from [19]

In Cypher, the pattern of the graph represented in Figure 4, would be represented like in Code snippet 8.

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(johan)-[:KNOWS]->(emil)
```

Code snippet 8 – Cypher statement representing the graph structure
retrieved from [19]

To bind the pattern to specific nodes and relationships in an existing dataset, the specification of some property values and node labels are necessary, such as in Code snippet 9, in order to help locate the relevant elements in the dataset [19].

```
(emil:Person {name:'Emil'})
     <-[:KNOWS]-(jim:Person {name:'Jim'})
     -[:KNOWS]->(johan:Person {name:'Johan'})
     -[:KNOWS]->(emil)
```

Code snippet 9 – Detailed Cypher statement representing the graph
retrieved from [19]

Each node is bound to its identifier using its *name* property and *Person* label. The *emil* identifier, for example, is bound to a node in the dataset with a label *Person* and a *name* property whose value is *Emil* [19].

The simplest queries in Cypher consist of a MATCH clause followed by a RETURN clause. An example can be observed in Code snippet 10.

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c),
      (a)-[:KNOWS]->(c)
RETURN b, c
```

Code snippet 10 – Example of Cypher query
retrieved from [19]

Nodes are represented between parentheses, while relationships are represented using pairs of dashes with greater-than or less-than signs, where the latter represents the relationship direction. Between the dashes, relationship names are enclosed by square brackets and prefixed by a colon. Node and relationship property key-value pairs are specified within curly braces [19].

In Code snippet 10, for the query to be anchored in one or more places in the graph, it is specified a lookup node labeled *Person,* with *name* property value “Jim” and bound the result to the identifier *a.* Cypher proceeds by matching the rest of the pattern with the nearby nodes connected to this anchor point, utilizing the given data on connections and adjacent nodes. During this process, it identifies nodes to associate with other identifiers. While *a* is always anchored to “Jim”, *b* and *c* are bound to a sequence of nodes as the query executes [19].

Gremlin

Gremlin is a path-oriented language which succinctly expresses complex graph traversals and mutation operations. It operates as a functional language, with traversal operators linked together to construct expressions resembling paths [20].

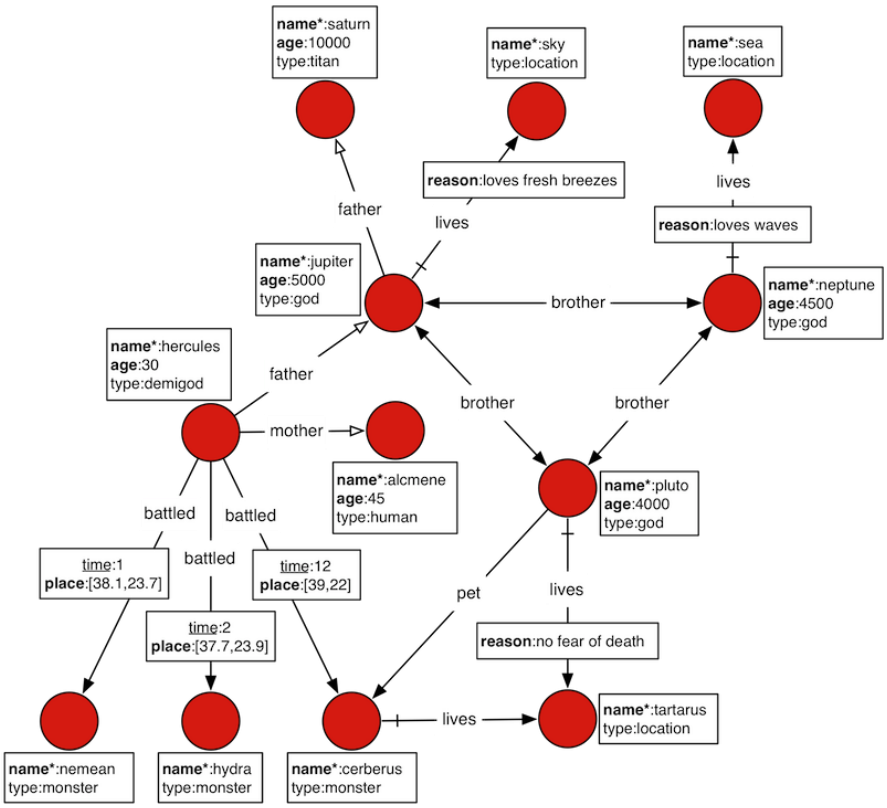

Figure 5 - Example of graph dataset
Image from [21]

A Gremlin query is a chain of operations/functions that are evaluated from left to right. Using the dataset represented in Figure 5, a simple query can be performed, which can be viewed in

```
gremlin> g.V().has('name',
'hercules').out('father').out('father').values('name')

==>saturn
```

Code snippet 11 – Example of simple gremlin query
retrieved from [20]

The query, presented in Code snippet 11, can be read (steps retrieved from [20]):

1. g: for the current graph traversal.
2. V: for all vertices in the graph
3. has('name', 'hercules'): filters the vertices down to those with name property "hercules" (there is only one).
4. out('father'): traverse outgoing father edge's from Hercules.
5. 'out('father')': traverse outgoing father edge's from Hercules' father's vertex (i.e. Jupiter).
6. name: get the name property of the "hercules" vertex's grandfather.

Within Gremlin, each operation, separated by a period, functions as a distinct operation acting upon the output generated by the preceding operation. The Gremlin language encompasses a diverse range of such operations. By altering a single operation or rearranging their sequence, one can evoke varied traversal semantics, thereby facilitating nuanced exploration of the data structure [20].

2.2.3 Common use cases

As pointed out in the previous section, graph databases excel when the number, complexity and depth of relationships increases compared to relational databases. The latter type handles relationships poorly, using expensive JOIN operations for navigating them, while in a graph database there are no JOINs or lookups, with relationships being stored in a much more flexible format. Graph databases systems are optimized for traversing through data quickly [18].

This type of database technology is optimal for use cases that involve many-to-many relationships with heterogeneous data that sets up needs to: navigate deep hierarchies, find hidden connections between distant items and discover inter-relationships between items [18]. The most common use cases are as follows [17]:

- **Fraud detection** - Graph databases excel in fraud detection due to their ability to swiftly traverse interconnected relationships. Analyzing complex networks of transactions, accounts, and user interactions becomes more efficient with graph structures. This allows for real-time pattern identification and anomaly detection, critical in fraud prevention.
- **Recommendation engines** - Due to the way graph databases handle relationships, they are ideal for recommendation engines by effortlessly navigating connections between users, items, and their interactions. Their capability to swiftly identify patterns and similarities among diverse nodes enables personalized and accurate recommendations, enhancing user engagement.
- **Route optimization** - Analyzing relationships between various points—factoring in variables like distances, traffic, and preferences—allows for quick and precise route optimization, benefiting logistics and navigation systems. Graph databases are ideal for route optimization because of their ability to model complex networks efficiently.
- **Pattern discovery** - Graph databases are well-suited for pattern discovery due to their adeptness at uncovering intricate relationships among interconnected data. The ability to traverse and analyze complex networks helps in unveiling hidden patterns or anomalies, aiding in research and identifying non-obvious connections.
- **Knowledge management** - In this context, information often exists in intricate networks with multiple interdependencies. Graph databases efficiently capture and store these relationships, enabling effective organization and retrieval of interconnected information. This information often represents complex metadata or domain concepts in a standardized format and provides rich semantics for natural language processing.

3 State of the art

This chapter describes the current state of GraphQL, graph databases and their integration. To achieve that a systematic literature review was performed, to answer the research questions specified. Technologies and implementation considerations resulting from the research were further explored, to simplify important decisions when designing the solution in the future.

3.1 Integration of GraphQL with graph databases

This literature review focuses on the analysis of existing papers that address applications, concerns, and technologies to consider when integrating graph databases with GraphQL. This analysis allows a clearer understanding of both technologies. This review's method starts by presenting the relevant research questions. Then the data sources used are presented, followed by presenting and explaining the search query utilized to obtain relevant articles. This is followed by the eligibility criteria that each article must adhere to. Finally, the data collection process is explained in detail.

3.1.1 Research Questions

In order to achieve the objectives stated in section 1.3, such as the prototype design and consequent implementation and experimentation so that RQ_A and RQ_B questions can be answered, three Research Questions (RQ)s were defined, which can be observed in Table 1. The purpose of these is to gather relevant information and to understand the best practices, use cases and technologies related to graph databases, GraphQL and alternatives for their integration.

Table 1- Research Questions

Id	Question	Justification
RQ1	What are the primary use cases and practical applications that drive the adoption of GraphQL in conjunction with graph databases?	To explore the practical use cases and analyze the key scenarios and applications where the combined use of GraphQL and graph databases may be most beneficial and prevalent.
RQ2	What are some of the most appropriate technologies when integrating GraphQL with graph databases?	To understand the technologies that allow a robust construction of a solution that integrates both concepts.
RQ3	What are the principal challenges and considerations encountered influencing their adoption and successful implementation?	To explore challenges and success factors in adopting GraphQL in tandem with graph databases, including insights into common practices and adoption patterns.

3.1.2 Data sources

Table 2 identifies the data sources to use in the systematic review. When retrieving documents in section 3.1.5, one of the documents retrieved from Google Scholar is grey literature. It was considered relevant and was used in the discussion of the results in section 3.2.

Table 2 - Selected databases

Database	URL
ACM Digital Library	https://dl.acm.org/
B-on	https://www.b-on.pt/
Google Scholar	https://scholar.google.com/

Additionally, two technical blogs, presented in

Table 3, were also used to retrieve relevant documents for this review, due to their relative significance on the topic of the literature review.

Table 3 - Technical blogs selected

Blog	URL
Neo4J	https://neo4j.com/
O'Reilly	https://www.oreilly.com/

3.1.3 Search terms

The keywords for the search query are: GraphQL, Graph Database, Graph Database Management System, GDBMS, Neo4j, FlockDB, JanusGraph.

Using these keywords, a search query was created:

- *"GraphQL" AND ("Graph Database*" OR "Graph Database Management System" OR "GDBMS" OR "neo4j" OR "FlockDB" OR "JanusGraph")*

By combining these terms using the OR operator with the AND operator, the search query only returns articles that refer GraphQL and at least one of the specified graph databases' terms.

3.1.4 Eligibility criteria

The eligibility criteria for a document to be selected are presented in Table 4 .

Table 4 - Eligibility criteria

Criteria	**Justification**
Inclusion	The source explores practical use cases and applications where GraphQL and graph databases were used together.
	The source mentions technologies, patterns, tools, common practices, and challenges when implementing a solution that integrates both GraphQL and graph databases.
	The source is a research article, technical report, or book.
Exclusion	The source is not written in English
	The source is unavailable to the author of this document.
	The source is not within the software engineering scope.

3.1.5 Data collection process

To obtain the relevant articles the PRISMA systematic review process [22] was followed this process consists of three steps and is represented in Figure 6:

- **Identification** consisted of obtaining all the articles from the data sources that were included in some of the eligibility criteria.
- **Screening** consisted of two parts, the first was a quick screening of all the identified articles, where 142 were excluded due to not complying with the eligibility criteria defined in Table 4. In the second part the relevant articles were thoroughly analyzed with the goal of finding information that fit the research questions. 14 articles were deemed to answer at least one research question.

- **Inclusion** consisted of gathering each of the remaining article's information that was relevant to the research.

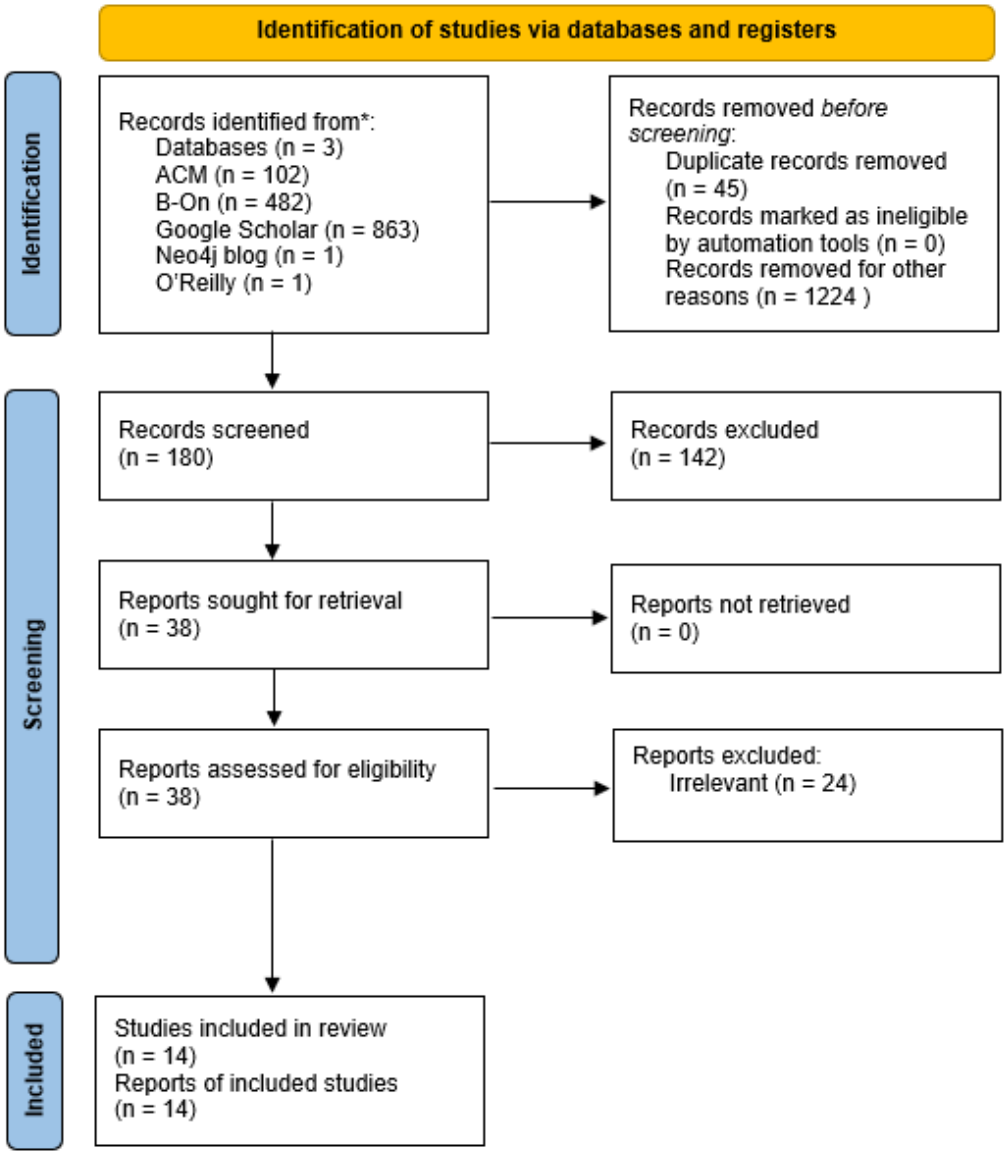

Figure 6 – PRISMA Flow diagram for the data collection process of the review

After the collection process, the documents to be included in the review are presented in Table 5.

Table 5 – Documents included in this review

Id	Title	Reference
1	BioDWH2: an automated graph-based data warehouse and mapping tool	[23]
2	Building Knowledge Graphs A Practitioner's Guide	[24]
3	Crossing the chasm between ontology engineering and application development: A survey	[25]
4	Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries	[26]
5	Establishment of a mindmap for medical e-Diagnosis as a service for graph-based learning and analytics	[27]
6	FullStack GraphQL Applications with React, Node.js, and Neo4j	[7]
7	GraphQL for archival metadata: An overview of the EHRI GraphQL API	[28]
8	Human AGEs: an interactive spatio-temporal visualization and database of human archeogenomics.	[29]
9	Literature review and example implementation on knowledge graphs	[30]
10	Managing and publishing standardized data catalogues to support BIM processes	[31]
11	SEDIA: A Platform for Semantically Enriched IoT Data Integration and Development of Smart City Applications	[32]
12	Trellis for efficient data and task management in the VA Million Veteran Program.	[33]
13	VARDA (VARved sediments DAtabase) – providing and connecting proxy data from annually laminated lake sediments.	[34]
14	Visual Design of GraphQL Data	[8]

3.2 Discussion

The section aims to provide answers to the research questions defined in section 3.1.1, with the data from the documents retrieved.

3.2.1 RQ1 - What are the primary use cases and practical applications that drive the adoption of GraphQL in conjunction with graph databases?

Friedrichs presents BioDWH2, a graph-based data warehouse and mapping tool that helps researchers with data integration and mapping tasks [23]. This tool is open-source and has a modular architecture. Thus, it is easily extensible and maintainable. Users can create multiple separate data warehouse projects. In each workspace, the data sources can be configured and using the tool transforms the raw source data into an internal graph data structure of nodes and edges. After this task is completed for every data source, each generated graph is merged into a large one, where further mapping is done to connect the various data sources [23].

Then to provide analysis capabilities, two additional tools are available:

- BioDWH2-Neo4j-Server – where the user can create a Neo4j graph database from the workspace database.
- BioDWH2-GraphQL-Server - provides a GraphQL endpoint for analysis queries, that operate directly on the workspace database.

Mohammed and Fiaidhi propose a mindmap that uses GraphQL for designing a medical e-diagnosis service that integrates healthcare data based on knowledge graphs [27]. In this solution, instead of having a complete data graph on a single codebase, the responsibilities of different parts of the graph are split across multiple microservices. All this while adhering to healthcare data exchange standards and harmonization between them. Moreover, due to the proposed ecosystem modular nature, microservices that bring graph-based machine learning capabilities and analytics can be simply added to the system.[27]. The GraphQL API is designed to cope with the need for more flexibility and efficiency in coordinating between microservices compared to the legacy REST API. It is highlighted that "GraphQL is an API to query knowledge and not data […] This unique criterion boosts the popularity of GraphQL and the adoption of graph databases (NoSQL) to make up for the limitations in relational databases" [27].

Bryant presents a GraphQL API that mediates access to the European Holocaust Research Infrastructure graph-based data store [28]. This API offers access to information about Holocaust-related archival material held in almost 500 institutions worldwide to researchers with a diverse range of needs. All this information is stored on a Neo4j graph database. GraphQL was chosen because it makes this type of data more accessible than a REST-style API, it would be more difficult to expose or fetch via the latter. The main criterion would be the problem of under-fetching or over-fetching that GraphQL solves efficiently, making it highly relevant for archival data [28].

Human AGEs [29] is a web server dedicated to the interactive visualization of archeogenomic data, and it consists of an interactive spatio-temporal map and a graph database. Users can load their data for visualization or select one of the data sources available on the graph database, which consists of published genomic data sources that were incorporated into it. The database is powered by Neo4j and the graph data exchange between the client and the Human AGEs server is managed by a GraphQL API [29].

Clemen, Thurm and Schilling present "datacat", an open source software stack for managing and publishing standardized data catalogues to support Building Information Modelling (BIM) processes [31]. At persistence level, uses a graph database powered by Neo4, in which the "datacat" GraphQL API, built with Spring Framework, can perform complex queries. A browser-based user tool for navigating and editing the database is also provided. A graph database was chosen compared to a relational database because since the used data was in a graph structure, to represent it in tables would implicate expensive operations when querying relationships. This is also one of the reasons GraphQL was opted instead of REST, due to the fact that in some use cases, like sub-views of the data catalogue, many relationships originating from a record need

to be queried at once. With REST this could be achieved, but more requests would be needed to be sent to the server's API to retrieve missing information [31].

Lymperis and Goumopoulos introduce SEDIA, a platform for developing smart city applications, including geographical information, to support a semantically enriched data model for effective data analysis and integration. The solution integrates data from heterogeneous sources, and by providing a unified view of data, enables a more comprehensive understanding of urban environments [32].

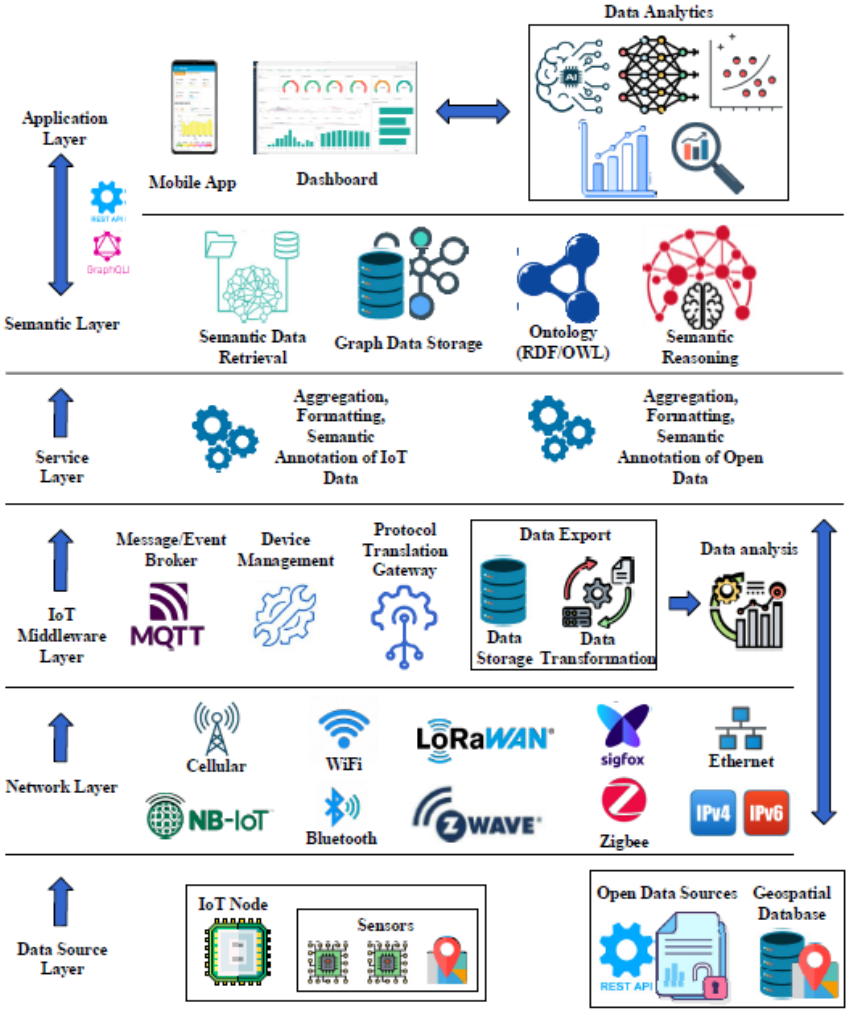

Figure 7 – SEDIA Architecture
Image from [32]

The architecture of this platform can be seen in Figure 7, and, in the current research, the focus is on the Data Source, Semantic and Application layers. In the Data Source layer, data from multiple sources is collected and sent to upper levels. Lymperis and Goumopoulos exemplify by stating that in an environmental application scenario, this data could be temperature, humidity, and air quality sensors, among others [32]. In the Semantic layer, this data is converted into

semantic form and stored in a graph database, in this case using Neo4j, because is open-source, highly scalable, and uses a schema-less property graph model that provides crucial flexibility for semantic annotation of raw data in smart city applications and robust developer community with a rich ecosystem of tools and libraries [32]. In the Application layer, resides a full-stack application employing GraphQL, using the Neo4j GraphQL library and Apollo.

Trellis [33] represents a cloud-based framework for data and task management, streamlining the entire workflow from data ingestion to presenting results. It operates with a graph database to orchestrate data processing workflows, ensuring data lineage tracking, simplified information querying, and bolstered fault-tolerance and scalability. By leveraging a scalable microservice architecture for executing bioinformatics tasks, Trellis has successfully facilitated efficient variant calling across a vast dataset of 100,000 human genomes gathered within the VA Million Veteran Program. Interaction with the database is mediated by a GraphQL API, which is enabled by the Neo4j GraphQL library. Graph-based database was chosen because it allows to model metadata natively according to the structure of a bioinformatics workflow [33].

Ramisch *et al.* presented "VARved sediments DAtabase" (VARDA) which is a data compilation for varve chronologies and associated paleoclimatic proxy records. Users can utilize VARDA to evaluate outcomes derived from climate models alongside high-resolution terrestrial paleoclimatic proxies. Furthermore, VARDA offers a technical platform to explore varved lake sediment databases through an interconnected data model. It has the capability to produce cutting-edge graphical representations for multisite comparisons [34]. To integrate the various paleoenvironmental datasets into one, the authors used a Neo4j graph database, which provides a flexible data structure that enables developers to quickly adapt the data model to changes in scientific or technical requirements, comparing to relational databases. To mediate the client communication with the database, a GraphQL API was implemented with the use of the Neo4j GraphQL library [34].

GraphQL is a very useful to expose knowledge graphs to consumers, providing a more flexible strategy for data consumption over a single endpoint [24], [25].

Summarizing, GraphQL and graph database are used together mainly for exposing knowledge graphs and heterogeneous data sources integration.

3.2.2 RQ2 - What are the most appropriate technologies when integrating GraphQL with graph databases?

Answering this question, it is expected to find the most popular technologies when integrating GraphQL with graph databases. In Table 6, there is represented a summary of the technologies identified and which authors mentioned them.

Table 6 – Technologies identified.

Technology/Method	References
Direct access/Manual Implementation	[23], [31]
Neo4j GraphQL Library	[7], [8], [24], [27]–[29], [32]–[34]
Dgraph	[26], [30]

It is evident that the most popular technology for integrating GraphQL with graph databases is the Neo4j GraphQL library.

3.2.2.1 Direct access

Friedrichs presented the BioDWH2-GraphQL-Server tool, that is implemented in the Java programming language, where GraphQL is used to perform queries directly on the workspace graph database [23].

This type of integration requires more implementation overhead and maintenance since many tasks related to data-fetching logic, type definitions and others need to be implemented manually.

3.2.2.2 Neo4j GraphQL Library

The Neo4j GraphQL library is a Node.js library designed to work with JavaScript GraphQL implementations, which in Figure 8 can be seen how it fits into a larger architecture. It allows developers to generate a fully functional GraphQL API from GraphQL type definitions, driving the database data model from GraphQL and autogenerating resolvers for data fetching and mutations, including complex filtering, ordering, and pagination. The Neo4j GraphQL library also enables adding custom logic beyond the generated create, read, update, delete (CRUD) operations [7].

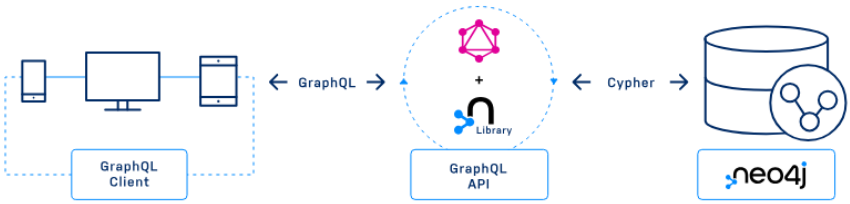

Figure 8 – API layer using Neo4j GraphQL library
Image from [7]

The API layer between the client and the database allows developers to implement features such as authorization and custom logic. The two main functions of the Neo4j GraphQL library are *GraphQL schema generation* and *GraphQL to Cypher translation*. The latter allows the generation of single database query at runtime from GraphQL requests and handling of custom logic defined in the GraphQL schema as subqueries in the generated queries [7].

The GraphQL schema generation uses the type definitions to generate an API with CRUD operations for them. It automatically generates queries and mutations, which are the semantics used in GraphQL for the CRUD operations, while also generating the necessary resolvers for them. The end result is an API with support for filtering ordering, pagination, and native database types, such as spatial and temporal types, without having to define these manually in the type definitions [7].

The resulting GraphQL schema object can subsequently be supplied to a GraphQL server implementation like Apollo Server. The server serves the API, managing networking and the execution of GraphQL processes [7].

This library also mitigates some common GraphQL problems, these are addressed in section 3.2.3.

3.2.2.3 Dgraph

Dgraph serves as a high-speed, transactional, horizontally scalable, and distributed GraphQL database utilizing a graph backend written in Go language. It excels in offering ACID transactions and operates as a native GraphQL database, strategically arranging data on disk to optimize query performance, production efficiency, and minimize disk seeks and network calls within clustered environments [30].

Compared to Neo4j, Dgraph performs better and consumes less memory. It is a newer technology and another difference is that while Neo4j uses Cypher has its query language, Dgraph uses GraphQL directly [30].

To conclude, Neo4j GraphQL library seems to be the most appropriate technology to integrate GraphQL with graph databases, although Dgraph, a newer technology, seems to have some advantages compared to it.

3.2.3 RQ3 - What are the principal challenges and considerations encountered influencing their adoption and successful implementation?

In this subsection, the main challenges and considerations when integrating both technologies should be identified, along with possible countermeasures to mitigate potential problems.

3.2.3.1 Boilerplate and developer productivity

Espinoza-Arias, Garijo and Corcho point out that when developing an API for KG consumption with GraphQL, the developers are often required to know the data structure (ontology) before defining the schema [25].

Building a typical GraphQL service requires several components: a schema delineating the GraphQL service, another schema for the underlying database, resolver functions responsible for data retrieval, as well as mutations to facilitate data creation and updates. This repetitive task can significantly hinder developer productivity, diverting their focus away from core application components. Instead of concentrating on critical aspects, developers spend valuable time crafting straightforward data-fetching logic for each type and field, slowing down the overall development process [7], [8].

The Neo4j GraphQL library, mentioned in [7], [8], mitigates this issue by automatically generating CRUD operations and their consequent resolvers from the GraphQL schema.

3.2.3.2 Poor performance and $n + 1$ query problem

When Lyon asks "If the client is free to compose queries as they wish, how can we ensure these queries don't become so complex as to impact performance significantly or overwhelm the computing resources of our backend infrastructure?", it also mentions ways to mitigate this issue. GraphQL allows developers to restrict the depth of the queries and restrict the queries that can be run to whitelisted selection [7].

The $n + 1$ query problem is common in GraphQL data fetching implementations, it occurs when an application needs to return a set of data that includes related nested data. To address this problem, developers can use tools like Dataloader that allow to batch and cache requests to the database. Another possible solution is by using GraphQL database integrations like Neo4j GraphQL library, because it generates a single Cypher query for any arbitrary GraphQL operation, assuring only one round trip to the database per request [7], [8].

3.2.3.3 Other considerations

Also, since GraphQL is an API query language (not a database query language), it lacks many semantics (e.g., projections) that we would expect in a database query language. Barrasa and Webber note that GraphQL doesn't come with features for using the topologies in knowledge graphs. But, using the Neo4j's GraphQL library, developers can use the `@relationship` directive to describe relationships between nodes. Using this directive allows to enrich the schema and have a better understanding of the model [24].

Enhancements to the functionality of a GraphQL API built with Neo4j's GraphQL library can be introduced through two primary methods. First, utilizing the `@cypher` schema directive enables the precise definition of custom logic for individual fields within the schema. Alternatively, custom resolvers can be developed and seamlessly integrated into the GraphQL schema to address specific requirements [7].

Furthermore, in scenarios where an established Neo4j database exists, the "@neo4j/introspector" package serves as a valuable tool. This package facilitates the automatic generation of GraphQL type definitions and the establishment of a GraphQL API that seamlessly interacts with the underlying Neo4j database structure. This approach streamlines the process of bridging the Neo4j data model with the GraphQL schema, allowing for efficient data querying and manipulation within the GraphQL ecosystem [7].

Instead of having two schemas, one for GraphQL and another for database, the Neo4j GraphQL library should be used to infer what the Neo4j data model should be, using the GraphQL schema [8].

Integrating GraphQL with graph databases, specifically using the Neo4j GraphQL library, automatically prevents the problem of Server/Client data mismatch that may occur when the database for the GraphQL service is not a graph database[8].

To conclude, most of the major challenges in GraphQL and graph databases integrations are already addressed by the Neo4j GraphQL library, which facilitates overall development.

3.3 Summary

Through research question RQ1, practical cases and common use cases of integrations of GraphQL with graph databases were uncovered. It was found that the most common applications are:

- Integration of various heterogeneous data sources in a single graph database, that is then made available through a GraphQL API.
- Exposing knowledge graphs to consumers, may that be a complete data graph on a single codebase or split across many components.
- A GraphQL API to expose already established graph-based data stores.

RQ2 delved into exploring technologies and how these integrations are being implemented, such as Neo4j GraphQL Library, DGraph and manual implementation. With RQ3 common problems and challenges to be aware of when integrating GraphQL with graph databases were identified, like boilerplate code, developer productivity and poor performance, while also providing possible solutions.

It is concluded that this chapter provided insights into conceiving and implementing solutions that integrate both technologies with suitable software practices and technologies for various needs.

4 Analysis and Design

This chapter delineates the analysis and design of the developed implementation. As outlined in section 1.3, a base open-source project that uses a relational database and GraphQL is necessary. Initially, the project selection process is explained, starting with describing the project search and selection of projects to apply decision-making methods. With the projects selected, the Analytic Hierarchy Process (AHP) is then applied to select an adequate project to serve as a base for the rest of the work. After, a design is delineated for the project's database migration to a graph database. Therefore, after the selection process, this chapter explains the characteristics of the initial project, mentioning the available functionalities, domain, and the defined architecture. Then, it describes the process that was used for the database technology transition.

4.1 Project to migrate

As mentioned in section 1.3, a project would be migrated to analyze the effects of integrating graph databases with GraphQL. When selecting the project, a few criteria were considered. As such, the project must have:

- The source code is available to everyone (Open-source), under a License that that allow content to be used, modified, and shared, such as the Apache License [35], the MIT License [36], and the GNU General Public License (GPL) [37], among others.
- Must use relational database technology, such as PostgreSQL and MySQL.
- The programming language should preferably be Java or Javascript/Typescript.
- The project must not fit into the frequent use cases for graph databases (see section 3.2.1).
- The project must use GraphQL queries and mutations.
- The project must have a small number of entities (tables), maximum 10.

A search was conducted on GitHub, using filters for the preferred programming languages, database technologies and to ensure GraphQL is used. After manually analyzing the results, three projects were selected to then apply a multicriteria method:

- **Project 1 – apollobank** [38] - full stack GraphQL banking application built using React, Node.js and TypeScript where, for example, a user can consult and create transactions, accounts, cards.
- **Project 2 – books-api** [39] - GraphQL API built using Spring Boot (Java) for book tracking. Readers can track books they would like to read, are currently reading, have read, or have not finished.
- **Project 3 – nutrition-tracker-BE** [40] - a Javascript GraphQL API for a food tracking application, it has full CRUD functionality for foods, exercise entries, weight entries, among other use cases.

4.1.1 Selection

The utilization of the Analytic Hierarchy Process (AHP) facilitated the selection of one of the options outlined in the previous section. This method enables the incorporation of qualitative or quantitative factors in the assessment process by breaking down the problem into hierarchical levels for easier evaluation.

Initiating the AHP process involves constructing a Hierarchical Decision Tree, outlining the objective, decision criteria, and alternatives. The alternatives specified in section 4.1 are to be utilized. The chosen criteria include:

- **Time**: Evaluating the time constraints of this dissertation to determine the most suitable alternative for implementation.
- **Documentation**: Quality and quantity of support documentation of the project.
- **Adequacy**: The adequacy criteria ascertain which option is more suitable for addressing the research questions established earlier and more adequate for this work.

Figure 9 contains the representation of the Hierarchical Decision Tree, which contains the decision criteria and alternatives to consider.

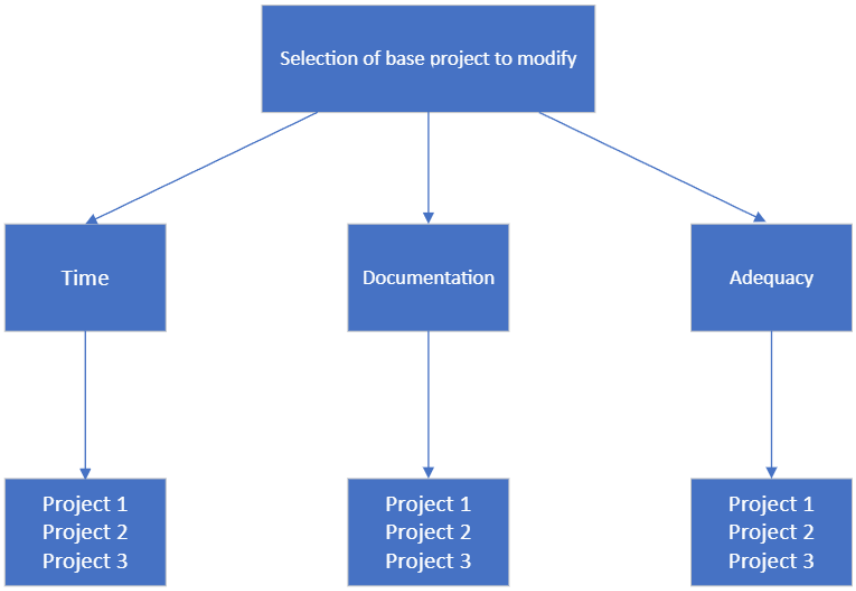

Figure 9 - Hierarchical Decision Tree

Once the decision tree is constructed, the subsequent step in the AHP involves crafting a comparison matrix to evaluate the significance of the criteria. Each criterion is then allocated a priority level in accordance with the Saaty fundamental scale (Table 7).

Table 7 - Saaty fundamental scale [41]

Level of Importance	**Definition**	**Explanation**
1	Equal importance	Two activities contribute equally to the objective
3	Weak importance of one over another	Experience and judgment slightly favor one activity over another
5	Essential or strong importance	Experience and judgment strongly favor one activity over another
7	Demonstrated importance	An activity is strong favored and its dominance is demonstrated in practice
9	Absolute importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgments	When compromise is needed

With the previously defined criteria, the following comparison matrix (Table 8) was created.

Table 8 – Criteria comparison matrix

	Time	**Documentation**	**Adequacy**
Time	1	1/2	1/5
Documentation	2	1	1/3
Adequacy	5	3	1

The next step consists of normalizing all the values of the previous matrix and calculating the priority vector. Every value is divided by the sum of the column it belongs to for normalization. The arithmetic mean of the normalized values in each row is then employed to derive the priority vector (Table 9).

Table 9 – Normalized comparison matrix

	Time	**Documentation**	**Adequacy**	**Priority Vector**
Time	0,1250	0,1111	0,1304	0,1222
Documentation	0,2500	0,2222	0,2174	0,2299
Adequacy	0,6250	0,6667	0,6522	0,6479

The next step is to calculate the Consistency Ratio (CR) to measure how consistent the values of the priority vector are. In order to determine CR, the Consistency Index (CI) is divided by the Random Index (RI), as displayed in the following formula:

$$CR = \frac{CI}{RI} \tag{1}$$

To obtain the value of the CI, the following formula is used:

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{2}$$

To obtain λ_{max} value the following formula is used:

$$Ax = \lambda_{max} x \tag{3}$$

Where A corresponds to the normalized comparison matrix and x corresponds to the priority vector previously obtained. The first step to compute λ_{max} is to multiply the criteria comparison matrix with the priority vector, which results are presented in Table 10. After that, the resulting values are averaged across the priority vector (4).

Table 10 – Consistency comparison matrix

Time	0,3667
Documentation	0,6902
Adequacy	1,9485

$$\lambda_{max} = \text{Mean}(\frac{0{,}3667}{0{,}1222}, \frac{0{,}6902}{0{,}2299}, \frac{1{,}9485}{0{,}6479}) \approx 3{,}0037 \quad (4)$$

Applying the CI formula, the following value is obtained:

$$CI = \frac{3{,}0037 - 3}{3 - 1} \approx 0{,}0018 \quad (5)$$

The last step is to infer the value of RI, which since the matrix dimension is 3, corresponds to a value of 0,58. Finally, CR can be calculated, applying the formula:

$$CR = \frac{CI}{RI} = \frac{0{,}0018}{0{,}58} \approx 0{,}0032 \quad (6)$$

Since the value obtained is less than 0,1, it means that the priority values are deemed as consistent and reliable.

In the next phase, comparison matrixes are created for each criterion, which can be seen from Table 11 to Table 16.

Table 11 – Comparison matrix for time criterion

	Project 1	**Project 2**	**Project 3**
Project 1	1	5	7
Project 2	1/5	1	4
Project 3	1/7	1/4	1

Table 12 – Normalized comparison matrix for time criterion

	Project 1	**Project 2**	**Project 3**	**Local priority**
Project 1	0,7447	0,8000	0,5833	0,7093
Project 2	0,1489	0,1600	0,3333	0,2141
Project 3	0,1064	0,0400	0,0833	0,0766

Table 13 – Comparison matrix for documentation criterion

	Project 1	**Project 2**	**Project 3**
Project 1	1	1/3	1/7
Project 2	3	1	1/5
Project 3	7	5	1

Table 14 – Normalized comparison matrix for documentation criterion

	Project 1	**Project 2**	**Project 3**	**Local priority**
Project 1	0,0909	0,0526	0,1064	0,0833
Project 2	0,2727	0,1579	0,1489	0,1932
Project 3	0,6364	0,7895	0,7447	0,7235

Table 15 – Comparison matrix for adequacy criterion

	Project 1	**Project 2**	**Project 3**
Project 1	1	3	5
Project 2	1/3	1	2
Project 3	1/5	1/2	1

Table 16 – Normalized comparison matrix for adequacy criterion

	Project 1	**Project 2**	**Project 3**	**Local priority**
Project 1	0,6522	0,6667	0,6250	0,6479
Project 2	0,2174	0,2222	0,2500	0,2299
Project 3	0,1304	0,1111	0,1250	0,1222

After creating all the matrices and obtaining the local priority vectors, the composite priority value of each alternative can be calculated. This value is achieved by multiplying a matrix obtained by merging the local priority vectors (Table 12,Table 14,Table 16) with the relative priority vector (Table 9):

$$\begin{bmatrix} 0{,}7093 & 0{,}0833 & 0{,}6479 \\ 0{,}2141 & 0{,}1932 & 0{,}2299 \\ 0{,}0766 & 0{,}7235 & 0{,}1222 \end{bmatrix} \times \begin{bmatrix} 0{,}1222 \\ 0{,}2299 \\ 0{,}6479 \end{bmatrix} = \begin{bmatrix} 0{,}5257 \\ 0{,}2195 \\ 0{,}2548 \end{bmatrix} \quad (7)$$

Since P1 (apollobank) obtained the highest value (0,5257), it is considered the best project to modify by a considerable margin. The other two alternatives have approximate values from each other. It is concluded that the application of the AHP method aided in the selection of a relevant base project, a core component of this dissertation project.

4.1.2 Characterisitics

As a result of the project selection process stated in section 4.1.1, the selected base project is apollobank [38], a full-stack GraphQL banking application project on GitHub. The characteristics of the project are:

- Created on October 17th 2021.
- Backend developed in Node.js, using PostgreSQL relational database.
- GraphQL API exposed using Apollo Server on the backend.
- Frontend using React.
- GraphQL mutations and queries are used.
- Features available include the creation of accounts, cards, and transactions, along with analytics and other functionalities.
- The project is available under the MIT License[38].

It is a banking application to help users keep track of their finances with diverse functionalities as follows:

- User registration and authentication.
- User password management.
- Users can create, delete and destroy currency accounts.
- Users can add money and exchange money between currency accounts.
- Users can create a card.
- Users can create a transaction associated with an account.
- Users can list their accounts.
- Users can list their cards.
- Users can list their transactions.
- Users can view account information.

Figure 10 details the main concepts identified.

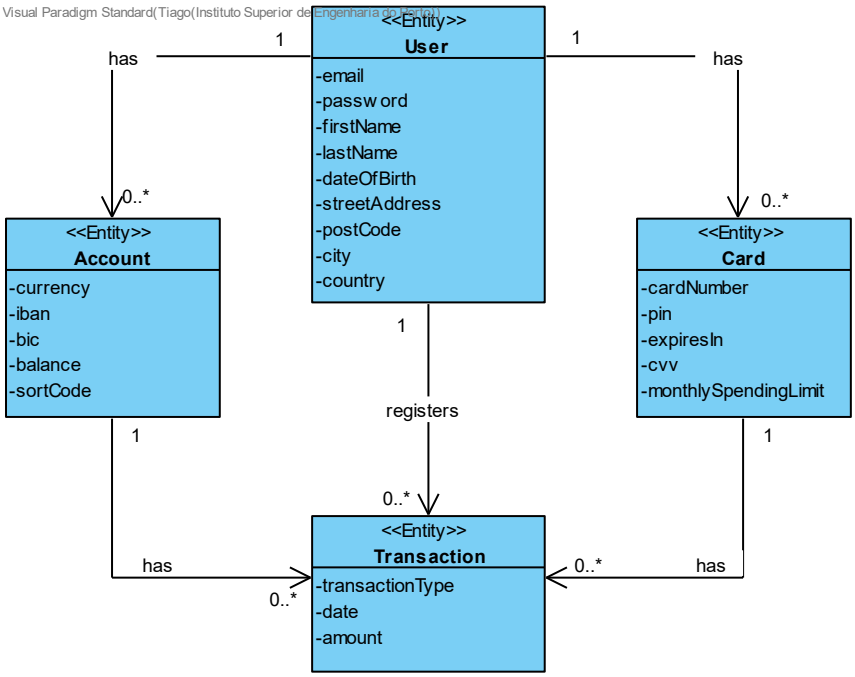

Figure 10 - Apollobank domain model diagram

Table 17 defines the main concepts and some of their attributes.

Table 17 – Glossary of the project

Concept	**Description**
User	An individual who interacts with the banking application by creating accounts, cards, and conducting transactions
Account	A financial account held by a user within the banking application.
Card	A payment card associated with a user's account for making transactions.
Transaction	A financial activity involving the movement of funds, associated with an account and/or card (optional and not implemented in the base project).
Transaction type	Type of transaction. A transaction must have one of the following types: withdrawal, deposit, invoice and payment.
Address	A physical location. Characterized by street address (name and number), postal code, city and country.
Sort Code	Six-digit number that identifies a bank. It is used by financial institutions in the United Kingdom, and in the Republic of Ireland.
Currency	System of money in general use in a particular country or economic context. Examples: EUR, USD, GBP.

4.1.3 Architecture

The architecture is simple, represented in Figure 11, with only 3 components. This work focuses only on the backend and database side of the base project.

Figure 11 – Architecture of apollobank

A description of the components is available in Table 18.

Table 18 – Components of apollobank application

Component	Description
apollobankClient	The apollobankClient is a React application that allows users to register themselves, create accounts and cards, view and register their transactions among other features.
apollobankServer	The apollobankServer is a Node.js application that provides a GraphQL API, using Apollo Server and Express, for the apollobankClient, handling all functionalities and authentication. Connection with the PostgreSQL database is handled with TypeORM.
PostgreSQL	PostgreSQL is a relational database used to store data handled by the apollobankServer.

Figure 12 presents a sequence diagram of the Account creation process. Noteworthy to mention the application uses TypeORM entities classes to manage operations with the database.

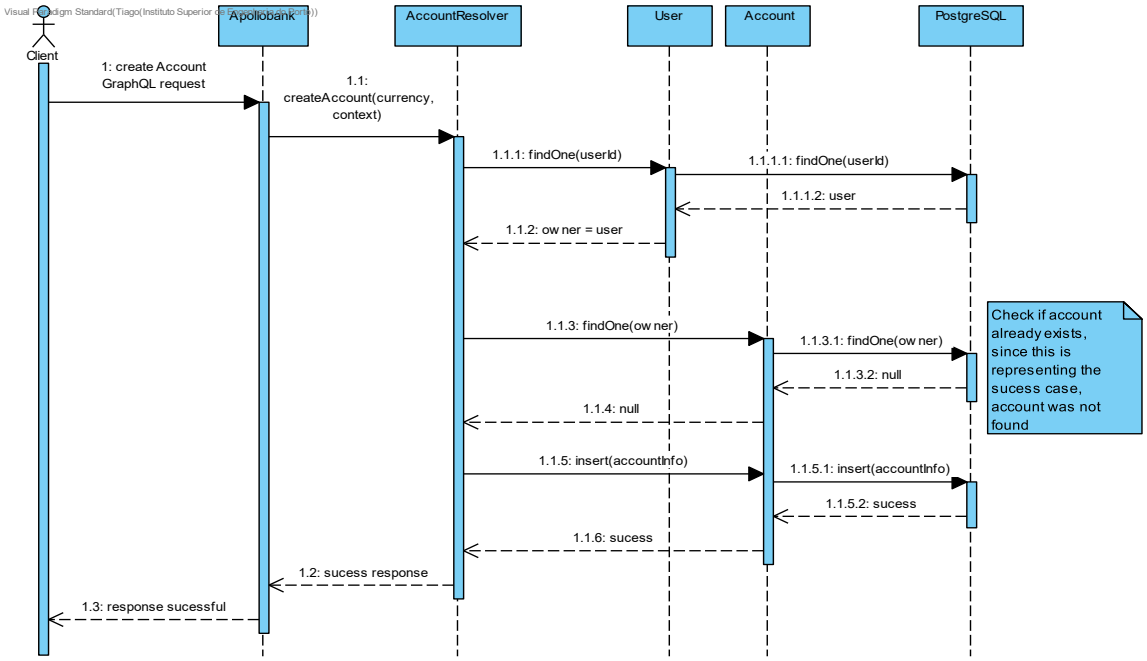

Figure 12 – High-level diagram of Account creation in Apollobank with relational DB

4.2 Migration Process

The transition from a relational database to a graph structure requires knowing the conceptual differences between these two structures and data models. Technologies and approaches to do this transition must be considered and delineated, with knowledge obtained from the previous chapters. After transitioning to a graph structure, it is expected that all functionalities available in the base project are present in the new one.

4.2.1 Data modeling transformation

The first step in the transition is to translate the existing relational data model into a graph structure. It involves identifying the entities, relationships, and attributes, to then translate them to their graph equivalent. To aid in this task, an Entity Relationship Diagram (ERD) of the base project was extracted, displayed in Figure 13, using pgAdmin [42], a PostgreSQL database administration tool.

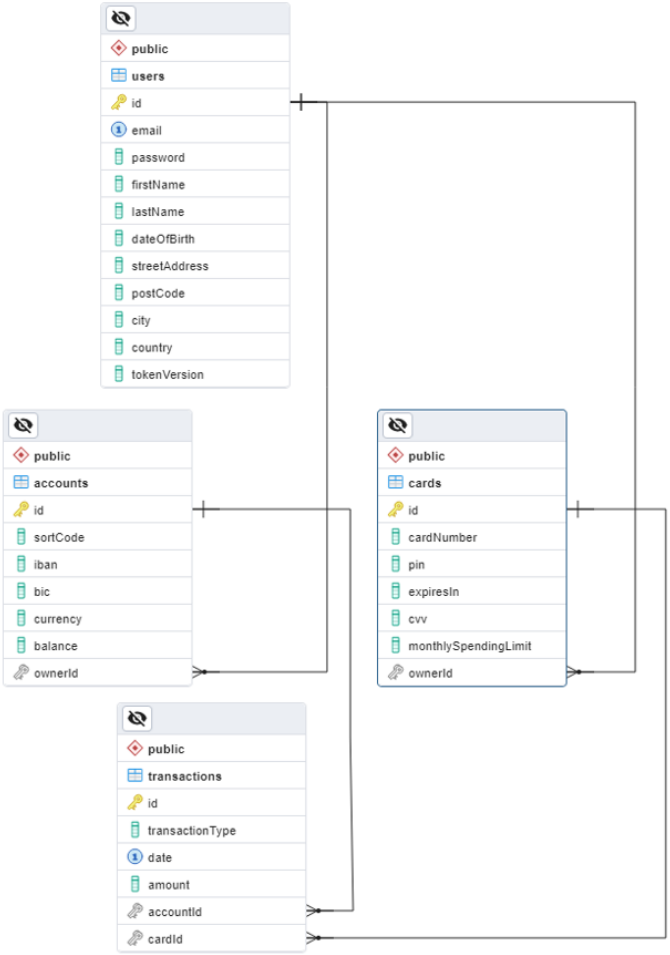

Figure 13 - ERD for the base project relational database

Analyzing the diagram, and with knowledge from previous sections, the following entities and their relationships were identified:

Entities:

- Users
- Accounts
- Cards
- Transactions

Relationships:

- Users can have Accounts (one-to-many)
- Accounts can have Transactions (one-to-many)
- Users can have Cards (one-to-many)
- Cards can have Transactions (one-to-many)

Constraints:

- User *email* should be unique.
- Transaction *date* should be unique.

Additional constraints and default values are implemented in the project's code which are:

- Account *sortCode* has a default value of "00-00-00".
- Account *balance* has a default value of 1000.
- User *email* and *password* are required.
- User *password* must have only alphanumeric characters, a minimum length of 6 and a maximum length of 30 characters.

When transitioning from a relational database to a graph database model, some guidelines were considered (obtained from [43]):

- ***Table to Node Label*** - Each entity represented by a table in the relational model becomes a node label in the graph model. These labels categorize and provide meaning to the nodes within the graph.
- ***Column to Node Property*** - The columns (fields) within the relational tables translate to node properties in the graph. These properties store the specific attributes associated with each node.
- ***Add Constraints/Indexes*** - add unique constraints for business primary keys, ensuring data uniqueness.
- ***Foreign keys to Relationships*** - Foreign keys in the relational model, which link tables together, become relationships in the graph model.

Conversion steps

Firstly, the node labels were identified. In this case, User, Account, Card and Transaction become labels in the graph model. As stated in the guidelines, the columns become the properties on those nodes, which were observed in Figure 13. Technical primary keys were kept, staying consistent with the base project. Unique constraints for business primary keys, User's *email*, and Transaction's *date*, were added to ensure the database does not allow duplicates. Foreign keys were transformed into relationships, as they show the links between the nodes.

The following relationships were identified:

- User -> OWNS -> Account
- User -> OWNS -> Card
- Account -> HAS_TRANSACTION -> Transaction
- Card -> HAS_TRANSACTION -> Transaction

Constraints are explored in further sections, because it's not possible to represent them in a graph structure. Default values were kept, remaining consistent with the base project definitions.

After all these steps, an example of the project's graph structure can be observed in Figure 14, where a sample user that owns two accounts and a card is represented, along with transactions.

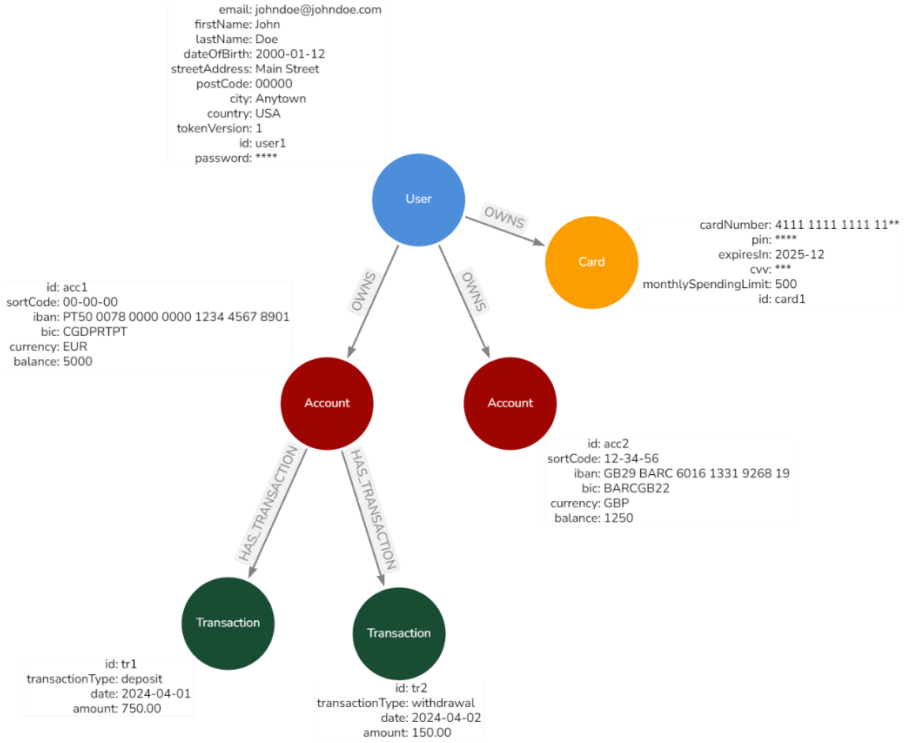

Figure 14- Example of graph structure with sample data

4.2.2 Technologies and approach

To perform the database transition, there are some alternatives, as found in section 3.2.2 of the literature review. One alternative would be to manual implement the transition, but that option is not optimal in terms of maintainability and developer productivity, since it takes longer, and unnecessary boilerplate code must be implemented. So, a better approach that explores the synergies between GraphQL and graph databases is to use the Neo4j GraphQL Library [6], previously explained in section 3.2.2.2.

Neo4j GraphQL Library reduces the amount of boilerplate code and time of development by automatically generating resolvers for queries and mutations regarding the entities represented in type definitions. It also optimizes the Cypher query sent to the database, something that could be done manually, but someone with less experience in this technology maybe could not achieve. Figure 15 presents a high-level diagram of how the library works.

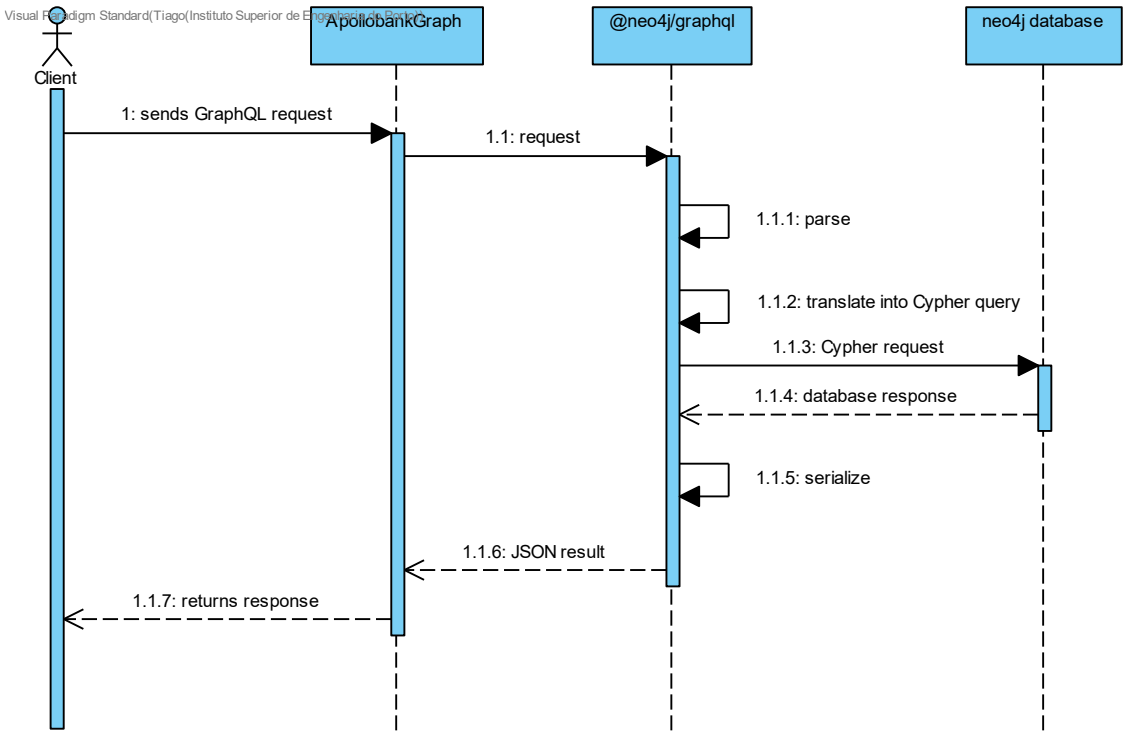

Figure 15 - High-level diagram of how Neo4j GraphQL library handling a request

It is important to mention that this execution process only applies for the auto-generated operations, others follow a normal execution process, similar to the diagram presented in Figure 16.

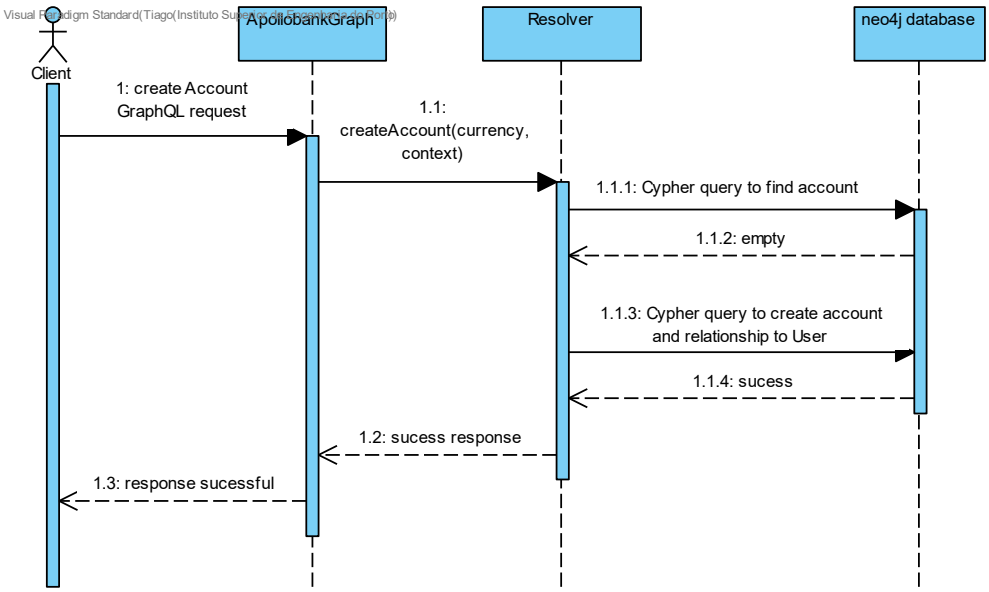

Figure 16 - High-level sequence diagram of account creation request

Comparing to the diagram presented for the same request in the base project (Figure 12), it is noteworthy to point out the elimination of TypeORM classes and the direct connection from the resolver to the database. Also, one of the queries was eliminated, because using Cypher, to create the relationship between the Account node and the User node, it first needs to find the latter so if it fails to do so, it stops the query execution.

Using this library, available for the base project's backend technology, Node.js, a GraphQL schema with the type definitions which map to the nodes and relationships in the Neo4j database was designed, which can be seen in Code snippet 12.

```
type User {
  id: ID! @id
  email: String! @unique
  firstName: String!
  lastName: String!
  dateOfBirth: String!
  streetAddress: String!
  postCode: String!
  city: String!
  country: String!
  accounts: [Account!]! @relationship(type: "OWNS", direction: OUT)
  cards: [Card!]! @relationship(type: "OWNS", direction: OUT)
   tokenVersion: Int! @default(value: 0)
  password: String! @selectable(onRead: false, onAggregate: false)
}

type Account {
  id: ID! @id
  sortCode: String  @default(value: "00-00-00")
  iban: String
  bic: String
  currency: String!
  balance: Float! @default(value: 1000.0)
  owner: User! @relationship(type: "OWNS", direction: IN)
  transactions: [Transaction!]! @relationship(type: "HAS_TRANSACTION",
direction: OUT)
}

type Transaction {
  id: ID! @id
  transactionType: String!
  date: DateTime! @timestamp
  amount: String!
  account: Account @relationship(type: "HAS_TRANSACTION", direction: IN)
  card: Card @relationship(type: "HAS_TRANSACTION", direction: IN)
}

type Card {
  id: ID! @id
  cardNumber: String!
  pin: Int!
  expiresIn: DateTime! @timestamp
  cvv: Int!
  monthlySpendingLimit: Float!
  owner: User! @relationship(type: "OWNS", direction: IN)
  transactions: [Transaction!]! @relationship(type: "HAS_TRANSACTION",
direction: OUT)
}
extend schema @mutation(operations: [])
```

Code snippet 12 – Initial GraphQL type definitions

With the type definitions (Code snippet 12), the Neo4j GraphQL Library generates queries and mutations for CRUD interactions, although in this specific case the auto-generation of mutations was disabled, using schema extensions with `@mutation`. The reason is that the mutations needed to have custom business logic, something that can't be done with the auto-

generated mutations. To follow the structure defined in section 4.2.1 and apply some of the business rules identified, some of the library's directives were used to extend the GraphQL schema functionality.

These directives were:

- `@relationship –` to configure relationships between object types.
- `@timestamp -` flags fields to be used to store timestamps on create/update events.
- `@id –` marks a field as the unique ID for an object type and allows for auto generation of IDs.
- `@unique –` indicates that there should be a uniqueness constraint in the database for the fields that it is applied to.
- `@default – to` set a default value for a field on object creation.

Although the Transaction's date field has a unique constraint, this can only be applied by using the library's functionality extending options, because it is not possible to use both `@unique` and `@timestamp` directives simultaneously.

In the following chapter, the implementation of the designed schema together with the Neo4j GraphQL Library is further explored.

5 Implementation

This chapter details the implementation process of the solution using a graph Database. It describes the changes made to the initial project to migrate to a graph database, following the process described in the Analysis and Design (Chapter 4).

5.1 Migration changes

This section describes all the major modifications made to the initial project with a relational database. Firstly, explains the initial graph database setup. Then, it details the implementation using the Neo4j GraphQL library, including how custom business logic and authentication were handled. Finally, provides the completed GraphQL schema that was initially designed in section 4.2.2.

The first step was to create the graph database, using the Neo4j Desktop application, the local development environment for projects that use Neo4j [44]. In this application, a project was created, and then a local graph database was configured, as seen in Figure 17.

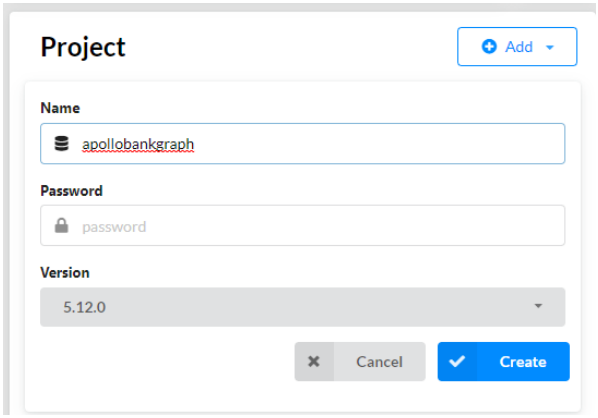

Figure 17 – Creation of local graph database instance in Neo4j Desktop

Having created the database instance, the next step was to install the Neo4j GraphQL library in the base project. There were some problems encountered due to some dependencies' versions, but they were solved.

5.1.1 Domain and custom resolvers

The first modifications to the project's code were related to the domain. The base project used TypeORM and *type-graphql* to implement the domain entities and properties, but since using the Neo4j GraphQL library the schema is the definition of those domain entities and properties, there was no longer need for those files to exist, so they were deleted. The GraphQL schema, which the domain entities configurations were already designed in section 4.2.2, was inserted in the *typedefs.ts* file. As mentioned in 4.2.2, all mutations couldn't be automatically generated, as well as one query, so they were needed to be added to the schema (Code snippet 13) and custom resolvers implemented for each one.

```
type Query {
    me: User @auth
  }

type Mutation {
    logout: Boolean! @auth
    revokeRefreshTokensForUser(userId: ID!): Boolean! @auth
    login(password: String!, email: String!): LoginResponse!
    register(
      country: String!, city: String!, postCode: String!
      streetAddress: String!, dateOfBirth: String!
      lastName: String!, firstName: String!
      password: String!, email: String!
    ): Boolean!
    updatePassword(newPassword: String!, oldPassword: String!): Boolean! @auth
    destroyAccount: Boolean! @auth
    addMoney(currency: String!, amount: Float!): AccountResponse! @auth
    exchange(
      amount: Float!
      toAccountCurrency: String!
      selectedAccountCurrency: String!
    ): AccountResponse! @auth
    createAccount(currency: String!): Boolean! @auth
    deleteAccount(currency: String!): Boolean! @auth
    createTransaction(currency: String!): Float @auth
    createCard: Boolean! @auth
  }
```

Code snippet 13 – Custom mutations and queries in the GraphQL schema

Since each mutation already had a resolver implemented in the initial project, the new resolvers code was an adaptation of that, following the same behavior of the original. For each entity, there was a file dedicated to having their operations' resolvers, like in the initial project, but structurally different internally. In the initial project, they were set as a class and the resolvers were functions of said class. In the new one, it was a JSON object that was exported in each file.

All resolvers could be in a single object in a single file, but they were separated into different files for each entity, for better project structure and code readability.

```
export const resolverAccounts = {
    Mutation: {
      createAccount: async (_parent,{currency}, context, _resolveInfo) => {
        const session = context.executionContext.session();
        try {
          if (!context.payload) {
            return false;
          }
          const userId = context.payload.userId;
          return await session.writeTransaction(async (transaction) => {
            try {
               // Find the Account node with the provided currency
              const findAccountResult = await transaction.run(
               'MATCH (a:Account {owner:$userId , currency: $currency}) RETURN a',
                 { userId, currency }
              );

              if (findAccountResult.records.length > 0) {
                throw new Error(`You already have a ${currency} account`);
              }
              const id= crypto.randomUUID();
              // Create the Account
              await transaction.run(
                `MATCH (a:User {id: $userId})
                 CREATE (u:Account {owner: $owner, currency: $currency,
                  sortCode: $sortCode, iban: $iban, bic: $bic,
                  balance: $balance, id: $id
                 })
                 MERGE (a)-[:OWNS]->(u)
                 RETURN u`,{
                  userId, owner:userId, currency: currency,
                  sortCode: currency === "GBP" ? createRandomSortCode() : "00-
00-00",
                  iban: createRandomIbanCode(), bic: createRandomBicCode(),
                  balance: 1000, id:id
                }
              );
            } catch (err) {
              console.log(err);
              return false;
            }
            // Return
            return true;
        });
        } finally {
          session.close();
        }
      },
…
```

Code snippet 14 – Excerpt of resolvers implementation for Account operations

Code snippet 14 presents an excerpt of *AccountResolver.ts*, including the *createAccount* mutation resolver implementation. In that resolver, all interactions with the database are

wrapped under a transaction, using the *session.writeTransaction* function. Firstly, it queries the database to check if the logged user already has an account for the specified currency. If not, it creates the Account node and creates a relationship between that node and the User node, using the MERGE cypher clause. After implementing the custom mutation resolvers, they were added to the Neo4j GraphQL instance properties in the *index.ts* file.

The rest of the custom mutation resolvers' implementations can be consulted in the projects' GitHub repository [45].

5.1.2 Authentication and Authorization

In the initial project, authentication and authorization were implemented using middleware that was not possible to reuse, so a new approach was needed. Neo4j GraphQL library provides directives to be able to implement that, but it doesn't work for custom mutations and queries, so a custom directive was be implemented for those. Since the solution used auto-generated queries too, then the two systems needed to be set, meaning that the directives provided by the library were used also.

The Neo4j GraphQL provided directives, `@authentication`, and `@authorization`, were applied to each entity-related type in the GraphQL schema. Code snippet 15 presents the application of those directives for the Account entity type definition. For authorization, filtering rules were applied so that only Accounts that belonged to the logged user were accessed.

```
 type Account @authentication @authorization(filter: [
    { where: { node: { owner: { id: "$jwt.userId" } } } }
]) {
    id: ID! @id
    sortCode: String  @default(value: "00-00-00")
    iban: String
    bic: String
    currency: String!
    balance: Float! @default(value: 1000.0)
    owner: User! @relationship(type: "OWNS", direction: IN)
    transactions: [Transaction!]! @relationship(type: "HAS_TRANSACTION",
direction: OUT)
  }
```

Code snippet 15 – Authentication and authorization directives in Account type definition

For the custom mutations, a custom `@auth` directive was implemented to handle authentication only, as authorization was handled at the resolver level, as in the initial project. Code snippet 16 demonstrates the directive code. The directive was then applied to the schema in the *index.ts* file, and the `@auth` tag was added at the end of each mutation declaration (see Code snippet 13)

```
export function authDirective(
    directiveName: string,
  ) {
    const typeDirectiveArgumentMaps: Record<string, any> = {}
    return {
      authDirectiveTypeDefs: `directive @${directiveName} on FIELD_DEFINITION`,
      authDirectiveTransformer: (schema: GraphQLSchema) =>
        mapSchema(schema, {
            [MapperKind.TYPE]: type => {
                const authDirective = getDirective(schema, type,
directiveName)?.[0]
                if (authDirective) {
                  typeDirectiveArgumentMaps[type.name] = authDirective
                }
                return undefined
              },
              [MapperKind.OBJECT_FIELD]: (fieldConfig, _fieldName, typeName) =>
{
                const authDirective =
                  getDirective(schema, fieldConfig, directiveName)?.[0] ??
                  typeDirectiveArgumentMaps[typeName]
                if (authDirective) {
                    const { resolve = defaultFieldResolver } = fieldConfig
                    fieldConfig.resolve = function (source, args, context, info)
{

                    const authorization: string | undefined =
context.req.headers["authorization"];

                    if (!authorization) {
                        throw new Error("Not authenticated");
                    }

                    try {
                        const token: string = authorization.split(" ")[1];
                        const payload: string | object = verify(token,
process.env.ACCESS_TOKEN_SECRET!);
                        context.payload = payload as any;
                    } catch (err) {
                        console.log(err);
                        throw new Error("Not authenticated");
                    }

                      return resolve(source, args, context, info)
                    }

                }
                return fieldConfig
              }
            })
        }
      }
```

Code snippet 16 – Custom directive for authentication

5.1.3 Server and Neo4j GraphQL instance configuration

Having implemented the custom mutation and query resolvers, as well as the custom authentication directive, they needed to be added to the Neo4j GraphQL instance configuration. Code snippet 17 provides an excerpt of that configuration, mainly the Neo4j GraphQL instance configuration and the application of the custom directive on the schema.

```
(async () => {
    try{
    const app = express();
    const httpServer = http.createServer(app);

    const driver = neo4j.driver(process.env.NEO4J_DATABASE_URL!,
neo4j.auth.basic(process.env.NEO4J_USERNAME!, process.env.NEO4J_PASSWORD!));

    const { authDirectiveTypeDefs, authDirectiveTransformer } =
authDirective('auth');
    const neoSchema = new Neo4jGraphQL({
    typeDefs: [
        authDirectiveTypeDefs,
        typeDefs,
    ],
    resolvers:[resolversUser,resolverCard,resolverAccounts,resolverTransaction],
    driver,
    features: {
        authorization: {
            key: process.env.ACCESS_TOKEN_SECRET!,
        },
    },
    //debug: true,
    });
    const schema = authDirectiveTransformer(await neoSchema.getSchema());
    const server = new ApolloServer<MyContext>({
        schema: schema,
        introspection: true,
        plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],
    });
 ...
```

Code snippet 17 – Excerpt of the server and Neo4j GraphQL instance configuration

The rest of the configuration can be consulted on the project's GitHub repository [45].

5.1.4 Changes to the initial project

When analyzing the initial project, it was noted that the *accounts* query was not able to return any Transactions. This meant all queries were relatively simple, and there wasn't a more complex one to use in the experiments to compare with the solution with the graph database. Since the queries in the new solution were auto-generated, they already returned the Transactions if needed.

So, the author decided to alter the initial project, to then be able to provide a better performance evaluation in Chapter 6. The alteration made it possible to return the Transactions related to an Account in the *accounts* query. The changes were made in the *Account.ts* and *AccountResolver.ts* files.

```
@Field((type) => [Transaction],{nullable:true})
@OneToMany(() => Transaction, (transaction) => transaction.account, { onDelete:
"CASCADE" })
transactions: Transaction[];
```

Code snippet 18 – Changes to transactions field definition in the initial project

```
@Query(() => [Account])
    @UseMiddleware(isAuth)
    async accounts(@Ctx() { payload }: MyContext) {
        if (!payload) {
            return null;
        }

        const owner: User | undefined = await User.findOne({ where: { id:
payload.userId } });

        if (owner) {
            return Account.find({ where: { owner: owner },
relations:["transactions"] });
        }

        return null;
    }
```

Code snippet 19 - Changes to *accounts* query resolver in the initial project

6 Experimentation and Evaluation

This chapter describes the experimentation and evaluation approach applied to the different solutions. As stated in section 1.3, Goals, Questions, Metrics (GQM) was the selected approach to evaluate the quality attributes, which are performance and maintainability. Metrics from both solutions were obtained and to validate the results differences, hypothesis tests were conducted, when possible.

6.1 Goals, Questions, Metrics

GQM is a systematic approach for defining and interpreting software measurement. It involves three levels: defining clear goals, developing questions to achieve those goals, and identifying metrics to answer the questions [10]. After measurement, the plan can be used from the bottom up to interpret the results and answer the questions, as can be seen in Figure 18. This approach is widely used in Software Engineering to enhance the understanding, control, and improvement of various processes and products through structured measurement.

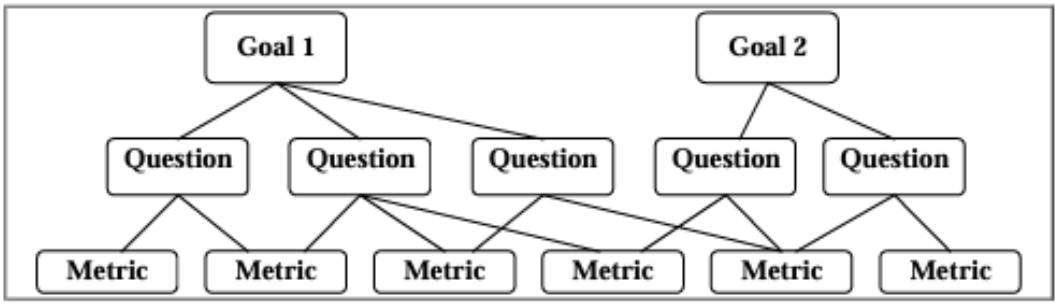

Figure 18 – GQM model structure
Image from [10]

Table 19 illustrates the application of the GQM approach, showcasing the goal and questions already stated in section 1.3, and the definition of the metrics to be obtained from each solution.

Table 19 – GQM Approach application

Goal	Questions	Metrics
Explore the performance and maintainability characteristics of graph databases (GB) compared to relational when GraphQL is used.	What are the maintainability characteristics of graph databases (GB) compared to relational when GraphQL is used?	Lines of Code (LOC)
		Cyclomatic complexity
	What are the performance characteristics of graph databases (GB) compared to relational when GraphQL is used?	Response time
		Throughput

The following sections describe the chosen metrics, which tools were used for obtaining their values and how they were configured.

6.1.1 Maintainability

Lines of Code (LOC) refers to the number of physical lines in a file that contain at least one non-whitespace character, excluding tabs and comments [46]. Cyclomatic complexity is a quantitative metric that measures the number of unique paths through the code. The complexity increases by one each time the control flow splits within a function. Every function has a baseline complexity of 1 [46].

SonarQube is a code quality, security and static analysis tool used to perform code inspections on projects, providing feedback and in-depth guidance on detected issues, contributing to a healthier codebase [47]. This tool was used in conjunction with the *sonarqube-scanner* npm module, which allows the execution of SonarQube/SonarCloud analyses on a JavaScript/TypeScript codebase [48], assuming SonarQube is installed locally.

6.1.2 Performance

The performance comparison between the solutions aims to evaluate and compare the speed, responsiveness, and stability of both solutions, focusing on the Response Time and Throughput metrics. Response Time is the duration from immediately before the request is sent until just after the final response is received [49]. Throughput is the number of requests per unit of time, the time is calculated from the start of the first sample to the end of the last sample [49].

The tool used to obtain the values was JMeter, which is an open-source Java software designed to load test applications and measure performance [50]. A test plan was designed to evaluate both GraphQL query and mutations (Figure 19), simulating common use cases, such as adding currency accounts, transactions and viewing the accounts transactions. Before a volume test was performed, the database was cleared and a certain number of users were registered and logged on, depending on the configuration of the test. These tasks were performed in *setUp* Thread Groups.

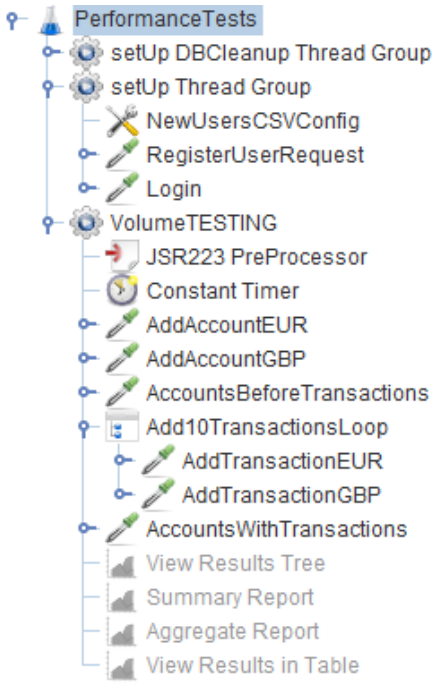

Figure 19 – JMeter test plan

Three different configurations of the test plan were executed on each of the solutions, each one differing on the number of users executing the test plan:

- 10 concurrent users.
- 100 concurrent users.
- 500 concurrent users.

Each request performed in the test plan was configured as a GraphQL HTTP Request, where a query or mutation was defined, and, if necessary, variables were set, which was the case for some mutations. An exemplification of a query used in the tests is presented in Code snippet 20.

```
query Accounts {
    accounts {
      balance
      currency
      iban
      transactions {
        date
        transactionType
        amount
      }
    }
  }
```

Code snippet 20 – GraphQL query used in JMeter test case

The query obtains all currency accounts related to the logged user and their respective transactions. The query itself tends to favor relational databases, due to its relatively flat structure. A query with deeper nested relationships would favor graph databases.

6.2 Experiments

After delineating the evaluation approach in section 6.1 and identifying the metrics to be evaluated, including an explanation on how they were obtained, it is fundamental to present the results and analyze them. This section presents and compares the obtained metrics' values, providing an in-depth analysis of them.

6.2.1 Maintainability

Following the execution of SonarQube analyses, the values of the relevant metrics were collected from the SonarQube project dashboard and presented in Table 20.

Table 20 - Maintainability metrics obtained for both solutions

Solution	LOC	Cyclomatic complexity (cumulative)
Apollobank with graph database	814	102
Apollobank with relational database	869	147

From the analysis of Table 20, the solution after migration to a graph database showed 6.9% less lines of code and a 30.6% decrease in cyclomatic complexity compared to the initial solution with a relational database. It is important to note that the values obtained for the cyclomatic complexity were in the project scope, it was not the maximum value achieved by a single function, meaning that it was the sum of the cyclomatic complexity of all functions in all source code files. One important contributor for this decrease in lines and complexity was the use of

the Neo4j GraphQL library, that automatically generated some GraphQL queries resolvers code, so the related code in the original solution was removed during the migration to a graph database. The difference would be even more noticeable if custom resolvers didn't need to be implemented.

6.2.2 Performance

A controlled test environment was assured by using the same computer to perform the tests, solely running the necessary programs. Following the execution of the test scenarios presented in section **Error! Reference source not found.** in JMeter for both solutions, the resulting values for the performance metrics were collected and are detailed in Table 21.

Table 21 – Performance metrics obtained for both solutions

	Scenario					
	10 users		**100 users**		**500 users**	
	Graph	**Relational**	**Graph**	**Relational**	**Graph**	**Relational**
Average (ms)	66	61	145	102	171	283
Min (ms)	9	5	9	4	7	4
Max (ms)	406	949	829	808	1228	2249
Throughput (requests/sec)	344.8	147.5	1688.8	1732.7	5700.3	3112.5

Analyzing the results, in small to medium number of users the solution with relational database had a slight advantage in terms of response times, but with more than double maximum response time compared to the other solution in the 10 concurrent users scenario. With a higher number of concurrent users, the solution with graph database notably showed better results, with an average response time 112 milliseconds quicker than the one from the initial solution (see Figure 20).

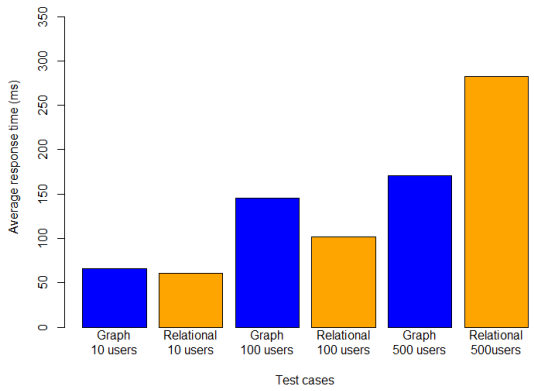

Figure 20 – Average response time by solution/scenario

In terms of throughput, the graph database solution achieved better results on the 10 and 500 users scenarios, with similar results in the 100 concurrent users scenario (see Figure 21).

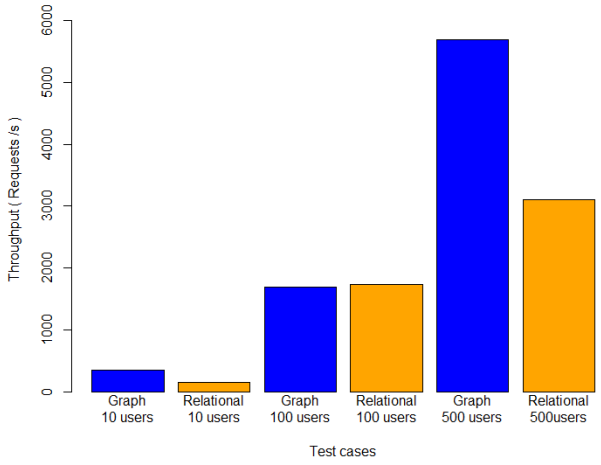

Figure 21 - Throughput by solution/scenario (higher is better)

When evaluating the performance of two solutions, particularly in the context of databases, it is also essential to distinguish between mutation and query operations. Mutations, which involve data modification operations such as inserts, updates, and deletes, can have different performance characteristics compared to queries, which primarily involve data retrieval. To better understand and compare both solutions results, mutations, and queries results were analyzed separately in the following sections.

6.2.2.1 Mutations

The mutations' average response times were aggregated and displayed in Table 22. Interpreting the results, in a small to medium number of users, the relational database solution outperformed the graph database solution five times, with the latter outperforming three times, mainly in the first request of the test plan (AddAccountEUR). The graph database solution in general outperformed the initial solution in a scenario with a higher number of users (500).

Table 22 – Mutations' average response times per solution/scenario

Mutation / Avg. Response time (ms)	**Scenarios**					
	10 users		**100 users**		**500 users**	
	Graph	**Relational**	**Graph**	**Relational**	**Graph**	**Relational**
AddAccountEUR	103	673	346	738	867	2160
AddAccountGBP	85	9	115	204	184	540
AddTransactionEUR	56	14	65	41	68	110
AddTransactionGBP	36	15	44	28	118	79

In terms of throughput, the graph database solution achieved better results overall in all scenarios, with some outliers (see Table 23).

Table 23 – Mutations' throughput values per solution/scenario

Mutation / Throughput (Requests/sec)	Scenarios					
	10 users		100 users		500 users	
	Graph	Relational	Graph	Relational	Graph	Relational
AddAccountEUR	94,3	10,5	215,5	123,8	407,2	222,3
AddAccountGBP	24,6	909,1	568,2	383,1	1597,4	713,3
AddTransactionEUR	458,7	270,3	2347,4	2538,1	9636,3	5694,8
AddTransactionGBP	757,6	187,3	5208,3	2747,3	2492,5	6983,2

Overall, in a small to medium number of users, the relational database solution outperformed in terms of response time but underperformed in throughput values. With a higher number of users, the graph database solution showed better results overall in terms of throughput and response times.

6.2.2.2 Queries

The average response times of the test plan's queries are presented in Table 24. The relational database solution obtained a faster average response time in all scenarios compared to the graph database one.

Table 24 – Queries' average response times per solution/scenario

Query / Avg. Response time (ms)	Scenarios					
	10 users		100 users		500 users	
	Graph	Relational	Graph	Relational	Graph	Relational
Accounts (before adding transactions)	216	9	488	116	297	274
Accounts (after adding transactions)	46	9	528	14	109	31

The relational database achieved superior throughput (see Table 25).

Table 25 – Queries' throughput values per solution/scenario

Query / Throughput (Requests/sec)	Scenarios					
	10 users		100 users		500 users	
	Graph	Relational	Graph	Relational	Graph	Relational
Accounts (before adding transactions)	40,8	588,2	166,7	543,5	1041,7	1039,5
Accounts (after adding transactions)	104,2	625	120,6	2857,1	2032,5	4424,8

The relational database solution outperformed the graph database one in query performance, before and after adding the transactions to the database. It confirmed that the query itself favored relational databases, due to its flatter structure.

6.3 Hypothesis tests

Hypothesis tests were made to determine if the differences in response times were indeed noteworthy. R Software is a popular free software utilized for statistical computing and graphics [51]. It was chosen to perform these tests because the author has experience working with it in his academic course. All files with test results and the R file with the hypothesis tests are available in the GitHub repository [45].

For each test scenario results obtained in section 6.2.2, two samples were obtained, one for the initial solution with a relational database, and another for the solution after migration to a graph database. This resulted in 3 pairs of samples, where a hypothesis test was formulated for each pair. When executing these tests, the distribution normality must be verified beforehand. That can be achieved with visual inspection but is often unreliable, so the application of a significance test is necessary, usually using the Shapiro-Wilk's method [52]. If the sample size exceeds the limit of the Shapiro-Wilk's method in R (5000 data points), other normality tests should be used. Razali and Wah concluded in their work comparing different normality tests that the Anderson-Darling test was the most comparable to Shapiro-Wilk's, so that would be the preferable alternative [53].

If normality is assumed the most adequate test is a parametric test, namely the paired samples t-test [54], if not then the test must be a non-parametric test, like the paired samples Wilcoxon test (also known as Wilcoxon signed-rank test)[55].

6.3.1 Hypothesis test for 10 concurrent users scenario samples

The first step was to verify the distribution normality of the samples, which could be visually inspected, but is often unreliable, so a significance test was applied to ascertain the normality of the distribution, using Shapiro-Wilk's method. Two hypotheses were defined:

- H0: data normally distributed.
- H1: data is not normally distributed.

In R software the formula for the significance test was applied (8), obtaining a *p-value* less than $2.2*10^{-16}$, rejecting the null hypothesis (*p-value* < 0,05) and consequently not assuming distribution normality.

$$shapiro.test(sample10Graph - sample10Relational) \quad (8)$$

Thus, the non-parametric Wilcoxon signed-rank test was applied, defining the hypotheses:

- H0: there is no variation in response times.
- H1: there is variation in response times.

The formula used to apply the Wilcoxon signed-rank test is presented in (9), obtaining a value of *p-value* equal to $6.158*10^{-12}$. The null hypothesis is rejected (*p-value* < 0,05), concluding that the response times before intervention were significantly different than the response times after the intervention in this test scenario.

$$wilcox.test(sampleGraph10, sampleRel10, paired = TRUE) \quad (9)$$

6.3.2 Hypothesis test for 100 concurrent users scenario

The distribution normality of the samples was verified using Shapiro-Wilk's method. Two hypotheses were defined:

- H0: data normally distributed.
- H1: data is not normally distributed.

In R software the formula for the significance test was applied (10), obtaining a *p-value* less than $2.2*10^{-16}$, rejecting the null hypothesis (*p-value* < 0,05) and consequently not assuming distribution normality.

$$shapiro.test(sample100Graph - sample100Relational) \quad (10)$$

The non-parametric Wilcoxon signed-rank test was applied, defining the hypotheses:

- H0: there is no variation in response times.
- H1: there is variation in response times.

The formula used to apply the Wilcoxon signed-rank test is presented in (11), where the value of *p-value* was less than $2.2*10^{-16}$. The null hypothesis is rejected (*p-value* < 0,05), concluding that the response times before intervention were significantly different than the response times after the intervention in this test scenario.

$$wilcox.test(sampleGraph100, sampleRel100, paired = TRUE) \quad (11)$$

6.3.3 Hypothesis test for 500 concurrent users scenario

The sample size exceeds the maximum limit to use Shapiro-Wilk's method in R, so distribution normality was verified using the Anderson-Darling test. Two hypotheses were defined:

- H0: data normally distributed.
- H1: data is not normally distributed.

In R software the formula for the significance test was applied (10), obtaining a *p-value* less than $2.2*10^{-16}$, rejecting the null hypothesis (*p-value* < 0,05) and consequently not assuming distribution normality.

$$ad.test(sample500Graph - sample500Relational) \quad (12)$$

The non-parametric Wilcoxon signed-rank test was applied, defining the hypotheses:

- H0: there is no variation in response times.
- H1: there is variation in response times.

The formula used to apply the Wilcoxon signed-rank test is presented in (11), where the value of *p-value* was less than $2.2*10^{-16}$. The null hypothesis is rejected (*p-value* < 0,05), concluding that the response times before intervention were significantly different than the response times after the intervention in this test scenario.

$$wilcox.test(sampleGraph500, sampleRel500, paired = TRUE) \quad (13)$$

6.4 Summary

The evaluation insights presented in this chapter helped answer the GQM questions identified in section 1.3 and detailed in section 6.1.

Maintainability-wise, the solution with a graph database exhibited better results, showing fewer lines of code and lower complexity overall compared to the initial solution with a relational database. These differences in results may vary depending on the technologies used during implementation because as described in the State of the Art (Chapter 3) and mentioned in section 4.2.2, the Neo4j GraphQL library automatically generates CRUD operations for the types defined in the schema. Consequently, this reduces the amount of code, improving maintainability. Another aspect to consider is the use case itself and its business rules, which in this specific case, some CRUD operations could not be automatically generated, since custom business logic had to be implemented. As such, the findings presented in this chapter related to maintainability suggest that a solution that uses the Neo4j GraphQL library to integrate

GraphQL with a graph database might offer better maintainability compared to one with a relational database.

In terms of performance, the results showed that with a higher number of users (500), the graph database solution produced better results, with an almost 40% decrease in average response time and an 83% increase in throughput after migrating the relational database solution. However, for smaller numbers of users (10 and 100), the relational database solution obtained better average response times, although in throughput it still tended slightly to favor the graph database solution.

Further analyzing the results obtained for each type of operation, it was concluded that the relational database showed significantly better results for queries in all scenarios. In terms of mutations, in a small to medium number of users, the relational database solution had better average response times but lower in throughput values overall. With a higher number of users, the graph database solution was the better performant in mutations, in both metrics. The discrepancy in the performance results related to queries might be due to the structure of the domain and query itself (Code snippet 20). The query, although featuring a nested relationship, still had a relatively flat structure, so it could be a factor why it favored a relational database. The hypothesis tests confirmed that the difference in response times were considered significant in all test case scenarios.

In summary, it was concluded that when handling a higher number of users, the graph database solution is more advisable while offering better maintainability as well. In small to medium numbers of users, relational seems to be preferable, in terms of average response times. For software that deals with more data modification operations, and a higher number of users, the graph database solution is preferable. In terms of data retrieval operations, if the queries and domain do not require deep nesting relationships, an advantage of graph databases, then the relational database seems more desirable, in terms of performance.

7 Conclusion

This chapter describes the results achieved with this work, according to the objectives delineated. After, difficulties found are exposed and threats to this work's validity are identified. Finally, future work to minimize the threats is described.

7.1 Achievements

This document provides a literature review and synthesizes relevant information about practical applications, appropriate technologies and considerations on the integration of graph databases and GraphQL technology. Open-source projects that use relational databases with GraphQL were analyzed and, after a selection process, one was used as a base project to design and implement into a new one using a graph database, which is available on GitHub [45].

The developed solution and the initial project were evaluated and compared, in terms of performance and maintainability, following a GQM approach. This work achieved the GQM goal: "Explore the performance and maintainability characteristics of graph databases (GB) compared to relational when GraphQL is used".

7.2 Difficulties

The implementation process encountered several challenges, primarily stemming from dependency version conflicts that impeded the initial development of the graph database solution. Additionally, issues arose during the implementation of authentication, largely due to insufficient documentation on incorporating it within custom mutations. It was also difficult to select a more complex query initially, that led to modifying the base project to then be able use one in the experiments.

7.3 Threats to Validity

Due to its focus on a single project, this study's findings may not be applicable to other projects in general. To do so, it would be necessary to conduct other studies with different projects, with a different number of entities, with more complex relationships that allow deeper nested querying in GraphQL.

Also, the implemented project utilized specific technologies, NodeJS and Neo4j GraphQL Library, so the results cannot be generalized to other projects with graph databases. Other studies that use different technologies to integrate graph databases with GraphQL should be conducted.

The experiments were performed in a local environment with limited resources, meaning that there was no possibility to perform more intensive tests to access the solutions' performance.

7.4 Future Work

To minimize the threats found and add value to the developed work, additional points would be interesting to address:

- Perform other types of performance tests on the projects, such as stress tests.
- Implement more complex queries in both projects, that try to take more advantage of graph database qualities, to then measure and compare performance.
- Develop other projects, using other technologies to integrate GraphQL with graph database.

Bibliography

[1] Linkurious, “Introduction to the graph technology landscape.” https://linkurious.com/blog/introduction-graph-technology-landscape/ (accessed Jan. 03, 2024).

[2] “Neo4j Graph Database & Analytics | Graph Database Management System.” https://neo4j.com/ (accessed Nov. 09, 2023).

[3] “JanusGraph.” https://janusgraph.org/ (accessed Nov. 09, 2023).

[4] “Memgraph.” https://memgraph.com/ (accessed Nov. 09, 2023).

[5] GraphQL, “Who’s Using | GraphQL.” https://graphql.org/users/ (accessed Jan. 03, 2024).

[6] “Open Source GraphQL Library | Open GraphQL API Library | Neo4j.” https://neo4j.com/product/graphql-library/ (accessed Nov. 09, 2023).

[7] W. Lyon, *FullStack GraphQL Applications with React, Node.js, and Neo4j*. 2022.

[8] T. Frisendal, *Visual Design of GraphQL Data*. Berkeley, CA: Apress, 2018.

[9] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, “GraphQL: A Systematic Mapping Study,” *ACM Comput. Surv.*, vol. 55, no. 10, 2023, doi: 10.1145/3561818.

[10] V. R. Basili, G. Caldiera, and H. D. Rombach, “THE GOAL QUESTION METRIC APPROACH,” *Encycl. Softw. Eng.*, pp. 528–532, 1994.

[11] Instituto Politécnico do Porto, “Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto,” [Online]. Available: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf.

[12] “Software engineering code of ethics and professional practice,” *Sci. Eng. Ethics*, vol. 7, no. 2, pp. 231–238, Jun. 2001, doi: 10.1007/s11948-001-0044-4.

[13] GraphQL, “Introduction to GraphQL | GraphQL.” https://graphql.org/learn/ (accessed Jan. 05, 2024).

[14] K. Stemmler, “What is GraphQL? GraphQL introduction | Apollo GraphQL Blog.” https://www.apollographql.com/blog/what-is-graphql-introduction (accessed Jan. 05, 2024).

[15] J. Helfer, “GraphQL explained | Apollo GraphQL Blog.” https://www.apollographql.com/blog/graphql-explained (accessed Jan. 05, 2024).

[16] K. Stemmler, “GraphQL Mutation vs Query – When to use a GraphQL Mutation | Apollo GraphQL Blog.” https://www.apollographql.com/blog/mutation-vs-query-when-to-use-graphql-mutation (accessed Feb. 26, 2024).

[17] Amazon Web Services, “What Is a Graph Database?” https://aws.amazon.com/nosql/graph/ (accessed Jan. 06, 2024).

[18] Neo4j, “What is a Graph Database? - Developer Guides.” https://neo4j.com/developer/graph-database/ (accessed Jan. 06, 2024).

[19] B. M. Sasaki, “Why Graph Database Query Language Matters More Than You Think.” https://neo4j.com/blog/why-database-query-language-matters/ (accessed Feb. 22, 2024).

[20] JanusGraph, “Gremlin Query Language - JanusGraph.” https://docs.janusgraph.org/getting-started/gremlin/ (accessed Feb. 26, 2024).

[21] JanusGraph, “Chapter 3. Getting Started.” https://old-docs.janusgraph.org/0.1.1/getting-started.html (accessed Feb. 26, 2024).

[22] M. J. Page *et al.*, “The PRISMA 2020 statement: an updated guideline for reporting systematic reviews,” *Syst. Rev.*, vol. 10, no. 1, p. 89, 2021, doi: 10.1186/s13643-021-01626-4.

[23] M. Friedrichs, “BioDWH2: an automated graph-based data warehouse and mapping tool,” *J. Integr. Bioinform.*, vol. 18, no. 2, pp. 167–176, Jun. 2021, doi: 10.1515/jib-2020-0033.

[24] J. Barrasa and J. Webber, *Building Knowledge Graphs*. O’Reilly Media, Inc., 2023.

[25] P. Espinoza-Arias, D. Garijo, and O. Corcho, “Crossing the chasm between ontology engineering and application development: A survey,” *J. Web Semant.*, vol. 70, no. C, p. 100655, Jul. 2021, doi: 10.1016/j.websem.2021.100655.

[26] M. Besta *et al.*, “Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries,” *ACM Comput. Surv.*, vol. 56, no. 2, pp. 1–40, Feb. 2024, doi: 10.1145/3604932.

[27] S. Mohammed and J. Fiaidhi, “Establishment of a mindmap for medical e-Diagnosis as a service for graph-based learning and analytics.,” *Neural Comput. Appl.*, vol. 35, no. 22, pp. 16089–16100, Aug. 2023, [Online]. Available: http://10.0.3.239/s00521-021-06200-6.

[28] M. Bryant, “GraphQL for archival metadata: An overview of the EHRI GraphQL API,” in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017, pp. 2225–2230, doi: 10.1109/BigData.2017.8258173.

[29] L. Ciecierski, I. Stolarek, and M. Figlerowicz, “Human AGEs: an interactive spatio-temporal visualization and database of human archeogenomics.,” *Nucleic Acids Res.*, vol. 51, no. W1, pp. W269–W273, Jul. 2023, [Online]. Available: http://10.0.4.69/nar/gkad428.

[30] C. Zafeiriou, “Literature review and example implementation on knowledge graphs,” 2021, [Online]. Available: https://repository.ihu.edu.gr//xmlui/handle/11544/29887.

[31] C. Clemen, B. Thurm, and S. Schilling, “Managing and publishing standardized data catalogues to support BIM processes,” in *Proc. of the Conference CIB W78*, 2021, vol.

2021, pp. 11–15.

[32] D. Lymperis and C. Goumopoulos, “SEDIA: A Platform for Semantically Enriched IoT Data Integration and Development of Smart City Applications,” *Futur. Internet*, vol. 15, no. 8, p. 276, Aug. 2023, doi: 10.3390/fi15080276.

[33] P. B. Ross, J. Song, P. S. Tsao, and C. Pan, “Trellis for efficient data and task management in the VA Million Veteran Program.,” *Sci. Rep.*, vol. 11, no. 1, pp. 1–12, Dec. 2021, [Online]. Available: http://10.0.4.14/s41598-021-02569-5.

[34] A. Ramisch *et al.*, “VARDA (VARved sediments DAtabase) – providing and connecting proxy data from annually laminated lake sediments.,” *Earth Syst. Sci. Data*, vol. 12, no. 3, pp. 2311–2332, Jul. 2020, [Online]. Available: http://10.0.20.74/essd-12-2311-2020.

[35] A. S. Foundation, “Apache License, Version 2.0.” https://www.apache.org/licenses/LICENSE-2.0 (accessed Jun. 01, 2024).

[36] MIT, “MIT License.” https://mit-license.org/ (accessed Jun. 01, 2024).

[37] Free Software Foundation, “The GNU General Public License v3.0 - GNU Project - Free Software Foundation.” https://www.gnu.org/licenses/gpl-3.0.en.html (accessed Jun. 01, 2024).

[38] E. Cui, “apollobank.” https://github.com/edwardcdev/apollobank (accessed Mar. 01, 2024).

[39] Project-Books, “books-api.” https://github.com/Project-Books/books-api (accessed Mar. 01, 2024).

[40] labspt3-nutrition-tracker, “nutrition-tracker-BE.” https://github.com/labspt3-nutrition-tracker/nutrition-tracker-BE (accessed Mar. 01, 2024).

[41] T. L. Saaty, “How to make a decision: The analytic hierarchy process,” *Eur. J. Oper. Res.*, vol. 48, no. 1, pp. 9–26, 1990, doi: https://doi.org/10.1016/0377-2217(90)90057-I.

[42] “pgAdmin - PostgreSQL Tools.” https://www.pgadmin.org/ (accessed Apr. 17, 2024).

[43] M. Hunger, R. Boyd, and W. Lyon, “The Definitive Guide to Graph Databases for the RDBMS Developer,” p. 34, 2021.

[44] Neo4j, “About Neo4j Desktop - Neo4j Desktop.” https://neo4j.com/docs/desktop-manual/current/about-desktop/ (accessed Jun. 01, 2024).

[45] T. Barbosa, “GitHub - TiagoFNB/apollobank: A full stack GraphQL banking application using React, Node & TypeScript.” https://github.com/TiagoFNB/apollobank (accessed Jun. 28, 2024).

[46] SonarSource, “Metric definitions.” https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/ (accessed May 31, 2024).

[47] SonarSource, “SonarQube 10.5.” https://docs.sonarsource.com/sonarqube/latest/ (accessed May 31, 2024).

[48] SonarSource, “GitHub - SonarSource/sonar-scanner-npm: SonarQube Scanner for the JavaScript world.” https://github.com/SonarSource/sonar-scanner-npm (accessed May 31, 2024).

[49] Apache JMeter, “Apache JMeter - User’s Manual: Glossary.” https://jmeter.apache.org/usermanual/glossary.html (accessed May 27, 2024).

[50] Apache JMeter, “Apache JMeter - Apache JMeter™.” https://jmeter.apache.org/index.html (accessed May 27, 2024).

[51] R Foundation, “R: The R Project for Statistical Computing.” https://www.r-project.org/ (accessed May 28, 2024).

[52] STHDA, “Normality Test in R - Easy Guides - Wiki - STHDA.” http://www.sthda.com/english/wiki/normality-test-in-r (accessed May 28, 2024).

[53] N. M. Razali and Y. B. Wah, “Power comparisons of Shapiro-Wilk , Kolmogorov-Smirnov , Lilliefors and Anderson-Darling tests,” 2011, [Online]. Available: https://api.semanticscholar.org/CorpusID:18639594.

[54] STHDA, “Paired Samples T-test in R - Easy Guides - Wiki - STHDA.” http://www.sthda.com/english/wiki/paired-samples-t-test-in-r (accessed May 31, 2024).

[55] STHDA, “Paired Samples Wilcoxon Test in R - Easy Guides - Wiki - STHDA.” http://www.sthda.com/english/wiki/paired-samples-wilcoxon-test-in-r (accessed May 31, 2024).