ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

EMBEDDED GRAPH DATABASE MANAGEMENT SYSTEM

Graduation Project Final Report

Kağan Hüseyin Erol
150190738

Department: Computer Engineering
Division: Computer Engineering

Advisor: Hayri Turgut Uyar

May 2023

ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

EMBEDDED GRAPH DATABASE MANAGEMENT SYSTEM

Graduation Project Final Report

Kağan Hüseyin Erol
150190738

Department: Computer Engineering
Division: Computer Engineering

Advisor: Hayri Turgut Uyar

May 2023

Statement of Authenticity

I hereby declare that in this study

1. all the content influenced by external references is cited clearly and in detail,
2. and all the remaining sections, especially the theoretical studies and implemented software/hardware that constitute the fundamental essence of this study is originated from my own authenticity.

İstanbul, May 2023

Kağan Hüseyin Erol

EMBEDDED GRAPH DATABASE MANAGEMENT SYSTEM

(SUMMARY)

A graph database is any storage system that uses graph structures with vertices called as nodes and their respective edges, to represent and store data. Graph databases provide insights and advantages on complex and dynamic relationships in highly interconnected data, one of the important business trends of today.

Nowadays, they have helped solve important problems in the areas of social networking, master data management, geospatial, recommendations, and more. The essential factor for increased focus on graph databases is the commercial success of companies such as Facebook, Google, and Twitter, all of whom have centered their business models around graph technologies. Graph databases and graph technology contain features inherent to traditional relational databases such as ACID compliancy and availability.

To better understand relationships between customers, elements in a social network, or genes and proteins, the ability to comprehend and analyze vast graphs of highly interconnected data is crucial. For that purpose, graph database management systems enable storing, processing, and analyzing such large, evolving, and interconnected datasets.

The aim of this project is to develop an embedded graph database management system to provide an embeddable, serverless, and lightweight solution for projects which want to store their data as a graph. This includes implementing a query language for querying graphs and providing the project as a library for achieving embeddability in various applications and languages. The application area is databases, specifically graph databases and embedded databases. This novelty of this project is to implement an embedded database management system similar to the operation of SQLite in relational databases but in the area of graph databases.

Therefore, it is going to utilize the openCypher standards for the graph query language and labeled property graph model for the data model of graphs. The main characteristic and final outcome of the project is a library code used as a graph database management system that can be embedded in other software applications. It can accomplish the essential graph querying operations such as creating and deleting nodes with their respective labels and properties, creating relationships between nodes also with their respective labels and properties, pattern matching, filtering the matched results, and returning the matched results. The library is object-oriented, modular, and designed with considering potential objectives. Such potential objectives are to integrate a robust on-disk data storage strategy, implementation of wrappers to support different languages, an implementation of a more extensive command-line interface, an implementation of an application that can visualize the graph data, and progress towards ACID compliance.

GÖMÜLÜ GRAF VERİTABANI YÖNETİM SİSTEMİ

(ÖZET)

Graf veri tabanı verileri temsil etmek ve depolamak için düğüm olarak adlandırılan noktalar ve bu düğümlere karşılık gelen kenarlar içeren graf yapılarını kullanan herhangi bir depolama sistemidir. Graf veri tabanları, günümüzün önemli iş trendlerinden biri olan yüksek düzeyde birbirine bağlı verilerdeki karmaşık ve dinamik ilişkiler hakkında öngörüler ve avantajlar sağlar.

Günümüzde, sosyal ağ oluşturma, ana veri yönetimi, coğrafi konum, öneriler ve daha pek çok alanda önemli sorunların çözülmesine yardımcı olmaktadırlar. Graf veri tabanlarına ilgiyi artıran temel faktör, iş modellerini graf teknolojileri etrafında toplamış olan Facebook, Google ve Twitter gibi şirketlerin ticari başarısıdır. Graf veri tabanları ve graf teknolojisi, ACID uyumluluğu ve ulaşılabilirliği gibi geleneksel ilişkisel veri tabanlarına özgü özellikler içerir.

Müşteriler arasındaki, bir sosyal ağdaki öğeler arasındaki veya genler ve proteinler arasındaki ilişkileri daha iyi anlamlandırabilmek için, yüksek düzeyde birbirine bağlı verilerin büyük graflarını anlama ve analiz etme yeteneği çok önemlidir. Bu amaçla, graf veri tabanı yönetim sistemleri, bu tür büyük, gelişen ve birbirine bağlı veri kümelerinin depolanmasını, işlenmesini ve analiz edilmesini sağlar.

Bu projenin amacı, projelerinde veya uygulamalarında verileri graf biçiminde depolamak isteyen kullanıcılar için gömülü bir graf veri tabanı yönetim sistemi geliştirmektir. Aynı zamanda grafları sorgulayabilmek için bir sorgu dilinin uygulanmasını ve çeşitli uygulamalarda gömülebilirliğini sağlamak için bir kütüphane sunmayı içerir. Uygulama alanı graf veri tabanları ve gömülü veritabanlar özelinde veri tabanlarıdır. Bu projenin yenilik maksadı, graf veri tabanları alanında SQLite'ın ilişkisel veri tabanlardaki çalışma sistemine benzeyen, gömülü bir veri tabanı yönetim sistemi gerçekleştirmektir.

Bu nedenle, graf sorgulama dili için openCypher standartları ve grafların veri modeli için etiketlenmiş özellikli graf modeli kullanılacaktır. Projenin temel niteliği ve nihai sonucu, diğer yazılım uygulamalarına gömülebilen bir graf veri tabanı yönetim sistemi olarak kullanılan bir kütüphane kodudur. Bu kütüphane, temel graf sorgulama işlemlerini başarıyla gerçekleştirebilir, ilgili etiketleri ve özellikleriyle düğümleri oluşturma ve silme, düğümler arasında ilişkileri ilgili etiketleri ve özellikleriyle oluşturma, örüntü eşleme, eşleşen sonuçları filtreleme ve eşleşen nihai sonucu kullanıcıya döndürme bu özellikler arasındadır. Kütüphane nesne yönelimlidir, modüler bir şekildedir ve potansiyel hedefleri göz önünde bulundurarak tasarlanmıştır. Bu potansiyel hedefler, sağlam bir disk tabanlı veri depolama stratejisi entegrasyonu, farklı dilleri desteklemek için aracı modüllerin uygulanması, daha kapsamlı bir komut satırı arabiriminin uygulanması, graf verilerini görselleştirebilen bir uygulama tasarlanması ve ACID uyumluluğuna doğru ilerlemeyi içerir.

Contents

1 Introduction and Project Summary

A graph database is a specialized database designed to efficiently manage graph data models based on graph theory principles. It consists of nodes representing entities and edges representing relationships. Relationships in graph databases can be labeled, directed, and assigned properties, and can be either directed or undirected. This allows for flexible and expressive data modeling and querying capabilities. Graph databases leverage the inherent structure of graphs to provide advantages such as efficient traversal and navigation of relationships. By prioritizing relationships and offering labeling and property features, graph databases enable powerful data management and analysis.

Two widely recognized graph data models are the Labeled Property Graph and the Resource Description Framework (RDF). In the Labeled Property Graph model, nodes, relationships, properties, and labels play key roles. Nodes and relationships can be labeled and store properties as key-value pairs, allowing for efficient grouping and detailed information representation. Relationships in this model are always directed, connecting a start node to an end node, enabling rapid traversal through direct storage. In contrast, RDF represents each piece of information as a separate node, offering a different approach to data representation within the graph database context.

Graph databases provide significant benefits for associative datasets and seamlessly integrate with the structure of object-oriented applications. They exhibit natural scalability, effortlessly accommodating large datasets without the need for intricate join operations. The flexibility and schema-less nature of graph databases make them particularly adept at handling ad hoc and evolving data with dynamic schemas. In contrast, relational database management systems tend to excel in operations involving a substantial volume of data elements.

Graph databases offer specialized graph query languages designed specifically for querying graph data. Cypher, which is based on the Labeled Property Graph model, stands out as an expressive, compact, and user-friendly query language, facilitating efficient and intuitive graph querying. SPARQL is tailored for RDF and enables querying of semantic data. Another notable graph query language is Gremlin, which follows an imperative, path-based approach, allowing for powerful graph traversals. Moreover, graph databases and graph technology encompass essential features like ACID compliance for transactional integrity and inherent availability, ensuring robustness and reliability comparable to traditional databases.

Graph databases find widespread applicability across diverse domains due to the inherent interconnectedness of the real world. They offer valuable insights and effective data management solutions in areas such as social networks, IoT and smart homes, and genetic research involving genes and proteins. The commercial triumph of tech giants like Facebook, Google, and Twitter, who have built their business models around graph database technologies, has played a significant role in amplifying the attention and interest in graph databases within the computer world. The ability of graph databases to capture and analyze complex relationships and interconnected data sets makes them indispensable tools for understanding and navigating the intricacies of real-world networks.

This project undertakes the development of an embedded graph database management system, offering a lightweight and serverless solution for projects seeking efficient storage of graph-based data. The project aims to create a programming library that serves as an embedded database system, akin to SQLite but optimized for graph databases, encompassing functionalities comparable to SQLite. A key aspect involves the implementation of a graph query language specifically designed for querying graph data, enabling seamless integration of the embedded database in various applications. The project's scope revolves around databases, with a particular focus on graph databases and embedded databases, aiming to deliver a versatile solution tailored to the specific requirements of graph-based data management.

The project utilizes the openCypher standards for the graph query language and the labeled property graph data model for graphs. The main outcome is a library code serving as a graph database management system that can be seamlessly embedded into other software applications. It accomplishes essential graph querying operations such as creating and deleting nodes with their respective labels and properties, creating relationships between nodes also with their respective labels and properties, pattern matching, filtering the matched results, and returning the matched results, that will serve as the foundation for other objectives. Additional potential objectives include enhancing the functionality of the graph query language, integrating a robust on-disk data storage strategy, implementing language-specific wrappers, creating a more elaborative command-line API, implementing an application that can visualize graph data, and ensuring transactionality through ACID compliance. These objectives can be built upon the developed embedded graph database management system.

The project leverages the openCypher standards for the graph query language and adopts the labeled property graph data model as the foundation for graph representation. The primary deliverable is a library code that acts as a fully embedded graph database management system, seamlessly integrable into diverse software applications. It encompasses essential graph querying operations, including node creation and deletion with corresponding labels and properties, establishment of relationships between nodes along with their labels and properties, pattern matching, filtering the matched results furthermore, and retrieval of the matched results. These core functionalities establish the groundwork for achieving other objectives. Potential additional goals involve enhancing the graph query language's capabilities, incorporating a robust on-disk data storage strategy, developing language-specific wrappers for smoother integration, creating a more comprehensive command-line API, implementing a visualization application for graph data, and ensuring transactionality through adherence to ACID compliance. These objectives can be pursued in a progressive manner, building upon the foundational embedded graph database management system that forms the core of the project.

1.1 Engineering Standards and Multiple Constraints

In the process of designing and implementing the project, adherence to engineering standards and consideration of multiple constraints played a crucial role. By incorporating established engineering practices and guidelines, the project aimed to ensure the reliability, compatibility, and quality of the developed solution.

The project adhered to relevant engineering standards, such as those pertaining to data modeling and query languages for graph databases. Standards, such as openCypher, were utilized to provide a common and interoperable foundation for the graph query language implementation. By aligning with these standards, the project ensured compatibility with existing graph database ecosystems and facilitated seamless integration with other systems and tools.

Additionally, various constraints were taken into account throughout the project lifecycle. For example, the project considered constraints related to cost-effectiveness and efficiency. By developing an embedded graph database management system, the project aimed to provide a lightweight and serverless solution, minimizing resource requirements and operational costs. The constraints of constructability and maintainability were also addressed in the report, facilitating ease of development, maintenance, and future enhancements.

Usability constraints were given importance to ensure that the developed system is user-friendly and intuitive. By incorporating industry-standard query languages, the project aimed to enable developers to seamlessly integrate and utilize the embedded graph database system within their applications.

Moreover, the project acknowledged constraints related to extensibility and interoperability. By implementing the graph query language based on open standards like openCypher, the system ensured comparability with other graph database implementations and planned future enhancements and extensions. This design decision allowed for the seamless integration of the embedded graph database system with a diverse range of software applications and ecosystems.

Throughout the project, various constraints were carefully evaluated, and trade-offs were made to achieve a high-quality solution that meets the desired needs and specifications. The iterative nature of the engineering design process enabled the consideration of different constraints at each stage, trying to ensure that the final solution strikes a balance between functionality, performance, maintainability, and compatibility.

In summary, the project embraced engineering standards and navigated multiple constraints to develop a lightweight embedded graph database management system. By aligning with established standards, considering various constraints, and making informed trade-offs, the project aimed to deliver a solution that not only meets the requirements but also ensures compatibility, usability, and extensibility.

2 Comparative Literature Survey

In the rapidly evolving landscape of Database Systems, Graph Databases have emerged as a compelling solution, specifically tailored for managing data with intricate graph structures comprising nodes and edges. The significance of Graph Databases is underscored by their ability to efficiently store, process, and query graph-based data, leading to their increasing adoption across various domains. This comparative literature survey aims to explore, to analyze, and to compare with the final design the key aspects of Graph Databases, encompassing a comprehensive examination of their characteristics, a comparative evaluation of different Graph Database systems, an investigation into Graph Query Languages for effective data retrieval, an in-depth exploration of the Labeled Property Graph Model, and a thorough exploration of storage strategies for Graph Data. By delving into these interconnected subjects, this survey seeks to shed light on the potential of Graph Databases as a sophisticated and robust solution, addressing the growing demands of modern data management and unlocking new opportunities for data analysis and decision-making processes.

In his study, Pokorny [1] critically examines the advancements, limitations, and potential future approaches within the realm of graph databases. The research highlights three key focal points of graph databases: the efficient processing of highly connected data, the flexibility offered by graph-based data models, and the exceptional performance achieved through local reads via graph traversal. However, when compared to traditional relational database management systems (RDBMS), potential users may encounter challenges in identifying the most suitable use cases for each product.

One current limitation and an area with potential for advancement lies in the utilization of graph databases for large-scale, unstructured datasets. Addressing this challenge will require a reevaluation of existing designs to ensure efficient handling of such datasets. Moreover, as emerging technologies like the Internet of Things (IoT) and Smart Homes gain traction, new application areas for graph databases are expected to emerge. This calls for future considerations and revisions of previous designs to accommodate the evolving landscape and cater to these emerging domains.

By critically assessing these aspects, Pokorny's research provides valuable insights into the current state of graph databases, their limitations, and potential areas for improvement. This analysis contributes to the broader understanding of graph database technologies and paves the way for future advancements and innovations in this rapidly evolving field.

Recognizing these powers and restrictions is of capital importance as the project is based on Graph Database Model. Before going into further detail, a comparison of current Graph Database systems should be surveyed.

2.1 Comparison of Graph Databases

In line with prior research, Besta et al. [2] have undertaken a comprehensive survey to establish a taxonomy of contemporary graph databases, shedding light on their design-related challenges. This pioneering study stands as the first dedicated exploration of the system aspects of graph databases, encompassing crucial topics such as data organization, distribution strategies, language support, and transactional capabilities. By delving into these key areas, the research not only provides valuable insights into the current landscape of graph databases but also contributes significantly to the understanding and advancement of this field. The taxonomy developed in the study serves as a valuable resource for researchers and practitioners, offering a structured framework for classifying and analyzing various graph database systems based on their fundamental characteristics and features.

This paper offers a wealth of information and in-depth analysis concerning the intricate design aspects of graph databases, thereby providing valuable insights that inform the design choices made for the current project. A key aspect highlighted is the influence of the supported conceptual graph model on query language support. Systems aligned with the RDF model typically offer support for SPARQL, while those centered around the Labeled Property Graph model tend to prioritize query languages such as Cypher or Gremlin. Notably, a significant observation is that there is currently a dearth of graph databases that prioritize embeddability, which distinguishes this project from existing solutions. By focusing on the development of a lightweight and embedded graph database management system, this project provides this gap within the graph database ecosystem.

Several other studies [3], [4], [5], [6] contribute to the comparison and understanding of graph databases. However, it becomes apparent that the term "graph database" can be ambiguous, as its interpretation varies across different studies. For instance, some studies, like [3], view it as a database model, while others, like [4], treat it as a database management system. The historical perspective offered in [3] may be seen as outdated, as it primarily surveys older studies and fails to mention popular data models or query languages. Conversely, the same researcher presents a more contemporary examination of graph database systems in [4]. The research discussed in [5] focuses on comparing graph databases, considering factors such as flexibility, scalability, and the capabilities of the query language. Meanwhile, McColl et al. [6] concentrate on comparing open-source graph databases, taking into account performance metrics and emerging trends in the realm of big data. These diverse studies contribute to a comprehensive understanding of graph databases and provide valuable insights for the current research.

While a canonical graph representation has yet to emerge, as highlighted in [2], it becomes evident that graph databases utilizing the Labeled Property Graph data model have gained prominence in terms of prevalence, timeliness, and, to some extent, performance. Considering this outcome, the project's selection of the query language was primarily narrowed down to Cypher and Gremlin before delving into further research on Graph Query Languages. Eventually, the openCypher standards were chosen as the foundation for implementing the graph query language, considering its widespread adoption and compatibility with the LPG data model. By leveraging openCypher, the project aims to provide a standardized and expressive query language for seamless integration with its graph database system.

2.2 Graph Query Languages

The paper [7] provides a comprehensive analysis of the fundamental features that form the basis for querying graph data, including graph database models, pattern matching, and data navigation. By examining recent developments in the field, the paper offers an insightful overview and categorization of these features. This research serves as a valuable resource for understanding the advancements in graph data querying and provides a framework for the project's exploration of relevant concepts and techniques. By leveraging the insights from

[7], the project can align with the latest trends and advancements in the field of graph database querying, ensuring that it incorporates state-of-the-art methodologies in its design and implementation.

In the study conducted by Holzschuher et al. [8], a thorough comparison of Cypher, Gremlin, SQL, and Native Access in Neo4j was carried out, focusing on their performance in highly interconnected data scenarios, resembling use cases found in social networks. The findings indicate that graph query languages, such as Cypher and Gremlin, outperform SQL in these contexts. Specifically, Cypher was found to be more intuitive for developers with prior SQL experience and demonstrated strong performance across various cases. However, Native Access in Neo4j was deemed irrelevant to the project and thus not included in the comparison. These insights provide valuable guidance for the project's selection of a suitable graph query language, highlighting the strengths of Cypher and its compatibility with developers familiar with SQL.

In the studies presented by [9] and [10], novel and standardized graph query languages based on the property graph model are proposed. [9] introduces a new query language called PGQL, which critiques the imperative nature of Gremlin, deeming it more suitable for expressing graph analysis algorithms rather than general querying in large-sized graphs. However, it is noted that newer versions of Cypher have addressed some of these concerns. On the other hand, [10] proposes a query language named G-CORE, aiming to serve as a foundation for future advancements in graph querying. G-CORE emphasizes the principles of composability, allowing graphs and their mental models to act as input and output components and highlighting the significance of paths as first-class citizens, which is a highly valued feature in graph databases. These studies contribute to the evolving landscape of graph query languages, offering alternative options and highlighting key considerations for the project's choice of query language.

In the study [11], the focus is on Cypher, which is recognized as one of the most widely used graph query languages in the field and is designed specifically for the property graph data model. The paper highlights that Cypher benefits from openCypher, which provides comprehensive documentation and standards for the language. This openness enables independent implementations and fosters a collaborative environment for the advancement of Cypher. The study suggests that Cypher's compatibility with newer standardized versions and its openness to independent strategies make it a favorable choice for long-term support in the project. The findings of [11] contribute to the decision-making process by emphasizing the suitability of Cypher as a reliable and evolving graph query language for the project.

2.3 The Labeled Property Graph Model and Storage

In Angles' study [12], a mathematical definition of the property graph data structure is provided, along with the introduction of a graph query language that bears resemblance to SPARQL and Cypher. The paper offers valuable insights by establishing a formal connection between the data model and the query language, which enhances the understanding of query semantics. This formal perspective aids in comprehending the underlying principles of the graph query language and its relationship to the property graph data structure. However, it is important to note that although Angles' work presents a more comprehensive version of property graphs, the focus of this research lies outside the scope of labeled property graphs. Nevertheless, the study serves as a relevant reference for understanding the fundamental aspects of query languages and their association with graph data structures.

Donkers' study [13] provides a comparative analysis between RDF and labeled property graphs using a dataset specifically related to smart homes. The results indicate that RDF performs better than labeled property graphs for small queries and proves to be more suitable in the context of smart homes. However, the research also highlights that labeled property graphs exhibit greater intuitiveness when dealing with complex queries on highly interconnected data. This investigation offers an intriguing perspective on the application of graph databases within the Internet of Things (IoT) domain, which was previously identified as a potential area of future application in [1]. By exploring the performance and usability aspects of graph databases in the context of IoT data, Donkers' research contributes to a deeper understanding of the strengths and limitations of RDF and labeled property graphs, thereby shedding light on their applicability in real-world scenarios such as smart homes.

Steinhaus et al. [14] present an innovative approach to storing graph data on disk with the introduction of G-Store, a lightweight disk-based storage manager. The paper outlines a placement strategy for efficiently organizing vertices and edges, along with a storage algorithm designed to optimize data placement on disk. Additionally, the authors provide a programming interface for integration with the storage manager. Although the current project does not incorporate on-disk data storage, the insights and techniques presented in this study offer valuable considerations for potential future integration of on-disk storage capabilities. By leveraging the strategies and algorithms proposed by Steinhaus et al., the project could benefit from enhanced data storage efficiency, improved performance, and the ability to handle larger datasets while ensuring optimal disk utilization.

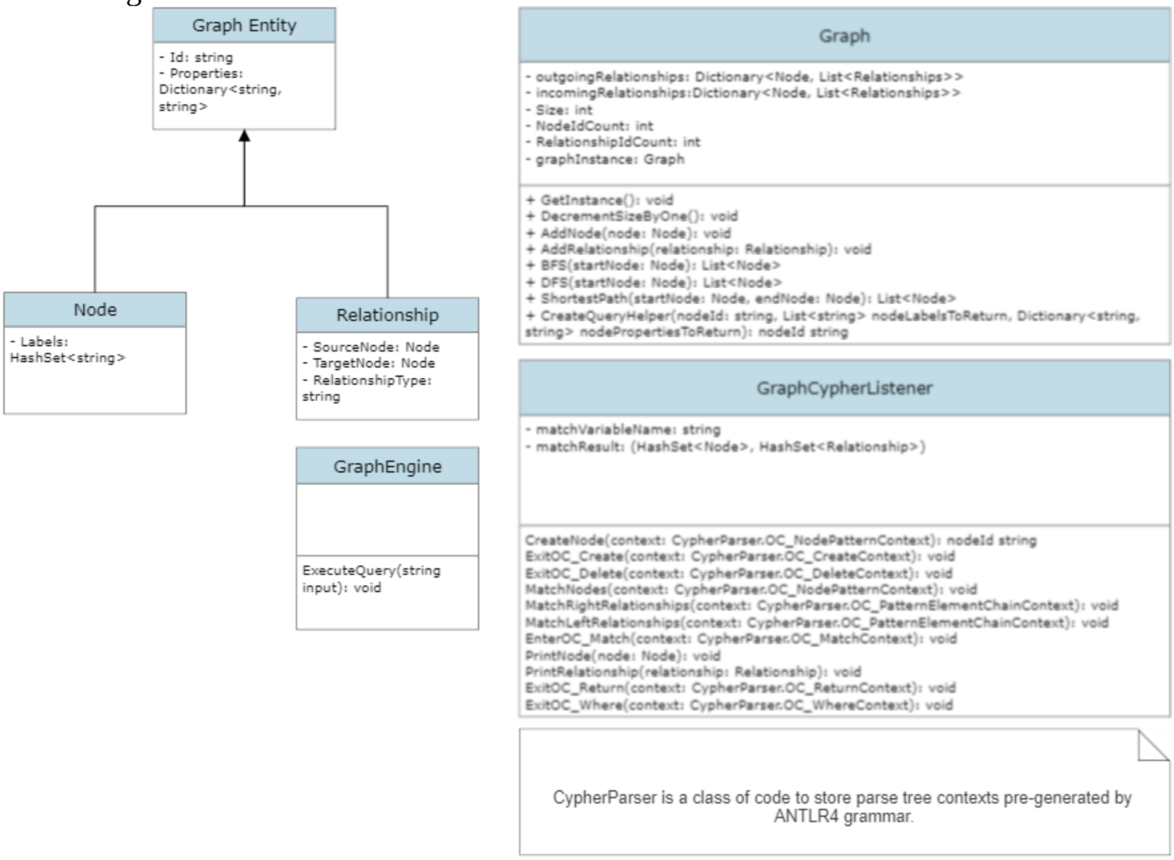
3 Developed Approach and System Model

3.1 Data Model

Data Structures:

1. *Graph Entity*: Represents a graph entity in the graph. It contains an *Id* to uniquely identify the entity. It contains a *Properties* dictionary to store its set of properties that can be uniquely identifiable with their respective property keys.
2. *Node*: Represents a node in the graph. It is a subclass of graph entity. Along with structures in the graph entity, it contains a Labels hash set to store its set of labels. It may be assigned a set of unique labels. It may have zero or more outgoing relationships represented in the *Graph* structure. It may have zero or more incoming relationships also represented in the *Graph* structure.
3. *Relationship*: Represents a directed relationship between two nodes in the graph. It is a subclass of graph entity. Along with structures in the graph entity, it contains a *SourceNode* node structure pointing to the source node, a *TargetNode* node structure pointing to the target node, and a *RelationshipType* to store the relationship type. It is assigned exactly one relationship type.
4. *Graph*: It is the main data structure for the graph database. Represents a directed graph with nodes and relationships. It maintains two dictionaries *outgoingRelationships* and *incomingRelationships*, where each node maps to a list of outgoing and incoming relationships, respectively. It contains *Size* to store the size of the graph in terms of the number of nodes, *NodeIdCount* to assign a unique id to nodes inside, and a *RelationshipIdCount* to assign a unique id to relationships maintained inside.

Class Diagrams:



3.2 Structural Model

Algorithm Definitions:

1. *BFS (Breadth-First Search)*: traverses the graph in a breadth-wise motion, exploring all the neighbors of a node before moving to the next level. It visits nodes at each level before moving to the next level. It is written for future functionality of graph database such as pattern matching relationships between nodes faster.
2. *DFS (Depth-First Search)*: traverses the graph in a depth-wise motion, exploring as far as possible along each branch before backtracking. It visits nodes in depth-first order. It is written for future functionality of graph database such as pattern matching relationships between nodes faster.
3. *Shortest Path*: finds the shortest path between two nodes in terms of the minimum number of edges. It uses Dijkstra's algorithm to compute the shortest path. It is written for future functionality of graph database such as pattern matching relationships between nodes faster.

Pseudocode:

```
BFS(startNode):
    visited = []
    queue = []

    visited.Add(startNode)
    queue.Enqueue(startNode)

    while queue is not empty:
        currentNode = queue.Dequeue()

        for each outgoing relationship from currentNode:
```

```
            neighbor = relationship.EndNode
            if neighbor is not in visited:
                visited.Add(neighbor)
                queue.Enqueue(neighbor)

    return visited

DFS(startNode):
    visited = []
    stack = []

    stack.Push(startNode)

    while stack is not empty:
        currentNode = stack.Pop()

        if currentNode is not in visited:
            visited.Add(currentNode)

            for each outgoing relationship from currentNode:
                neighbor = relationship.EndNode
                stack.Push(neighbor)

    return visited

ShortestPath(startNode, endNode):
    distances = dictionary of node distances (initialized with infinity)
    previous = dictionary of previous nodes (initialized with null)
    unvisited = list of all nodes

    distances[startNode] = 0

    while unvisited is not empty:
        currentNode = node with minimum distance in distances from unvisited

        if currentNode is null:
            break

        unvisited.Remove(currentNode)

        for each outgoing relationship from currentNode:
            neighbor = relationship.EndNode
            distance = distances[currentNode] + 1 // Assuming each edge has weight 1

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous[neighbor] = currentNode

    shortestPath = []
    current = endNode

    while current is not null:
        shortestPath.Insert(0, current)
        current = previous[current]

    return shortestPath
```

4 Experimental Environment and Design

The project incorporates experiments to evaluate the effectiveness and performance of the developed embedded graph database management system. To ensure reliable and reproducible results, a well-defined experimental environment was established.

The development environment consisted of Microsoft Visual Studio. The programming language of the library is C#. Antlr is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. From a grammar, ANTLR generates a parser that can build and walk parse trees. It is utilized to create pre-generated parser code from a grammar provided by openCypher to parse inputs from user. Functionality of the code to perform necessary operations is coded by me authentically.

Additionally, the experimental setup included representative datasets from openCypher standards that simulated real-world scenarios on queries. These datasets were carefully selected to cover a wide range of graph sizes and complexities, allowing for comprehensive testing and evaluation of the system's capabilities.

In terms of resource requirements, the project demanded a suitable development machine with adequate processing power, memory, and storage to support the development and testing phases. The machine should meet the minimum system requirements of the chosen development tools, programming languages, and libraries.

Furthermore, the project SpecFlow and its Gherkin structure to utilize various feature-scenario based testing methodologies on graph query language clauses, including unit tests, functional tests, and performance tests. These tests were designed to assess the system's functionality, validate its behavior against predefined requirements, and evaluate its performance under different workloads and data sizes.

By establishing a well-defined experimental environment and considering resource requirements, the project ensures that the evaluation process is conducted in a controlled and reliable manner. This allows for accurate assessment and comparison of the system's performance and functionality.

5 Comparative Evaluation and Discussion

In this section, I evaluate the system design and compare it with approaches found in the literature, focusing on quantitative measurements. The evaluation criteria are based on the goals and evaluation criteria presented in the interim report.

One of the evaluation criteria stated in the interim report was the successful execution of at least five openCypher clauses. I am pleased to report that our system meets this criterion, as it is capable of executing CREATE, DELETE, MATCH, WHERE, RETURN openCypher clauses effectively. Through testing and verification, it is confirmed that the system can process a wide range of openCypher queries accurately and efficiently.

Another evaluation criterion was the implementation of at least one wrapper for a programming language. In the project, we have successfully implemented a wrapper in C#, providing a convenient REPL (Read-Eval-Print Loop) interface. This wrapper utilizes the library DLL, which contains the essential functionality of our embedded graph database management system. The C# wrapper allows developers to interact with the graph database using familiar programming constructs and seamlessly integrate it into their C# applications.

By meeting these evaluation criteria, our system demonstrates its capability to execute openCypher queries and provides a user-friendly programming interface. We have achieved the intended goals set forth in the interim report and created a system that aligns with the design objectives.

In conclusion, our system design has met the evaluation criteria outlined in the interim report, successfully executing openCypher queries and implementing a programming language wrapper. This establishes a solid foundation for future enhancements and opens up possibilities for further quantitative evaluations, ensuring the continued improvement and refinement of our embedded graph database management system.

6 Conclusion and Future Work

The project at hand focuses on the development of an embedded graph database management system, offering a lightweight and serverless solution for projects that require efficient graph-based data storage. By adopting the openCypher standards for the graph query language and leveraging the labeled property graph data model, the system aims to provide a flexible and expressive platform for managing graph data effectively.

The primary objective is to create a library code that seamlessly integrates into other software applications, enabling developers to harness the power of graph databases within their projects. To achieve this, the project will initially concentrate on implementing fundamental graph querying operations, including node and relationship creation, deletion, and pattern matching. These core functionalities serve as the building blocks for subsequent objectives and pave the way for advanced data manipulation and analysis.

Looking towards future enhancements, one potential avenue is to enrich the functionality of the graph query language. By expanding its capabilities and expressive power, the system can accommodate more intricate queries and complex data transformations. This could involve incorporating features like aggregations, advanced filtering mechanisms, and support for graph algorithms, empowering users to perform sophisticated analysis and gain deeper insights from their graph data.

Furthermore, integrating a robust on-disk data storage strategy can significantly enhance the system's scalability and performance. Drawing insights from the lightweight disk-based storage manager proposed by Steinhaus et al. [14], the project can devise efficient placement strategies for vertices and edges, optimize storage algorithms, and develop a programming interface for seamless interaction with the storage manager. By efficiently managing the on-disk data, the system can handle larger datasets, enable faster data retrieval, and provide improved resilience and durability.

To enhance usability and streamline integration with different programming languages, language-specific wrappers can be developed. These wrappers would provide developers with idiomatic interfaces to interact with the embedded graph database system, abstracting away low-level implementation details and offering a more intuitive and seamless experience. This expansion would ensure wider adoption and make the system accessible to developers from diverse backgrounds and language preferences.

Improving the user experience is also essential, and one way to achieve this is by creating a more elaborate command-line API. By enriching the command-line interface with comprehensive commands, options, and intuitive syntax, users can easily perform various graph-related tasks, such as data management, querying, and visualization, in a more efficient and user-friendly manner.

Additionally, the project can explore the implementation of visualization capabilities, enabling users to gain visual insights into their graph data. By integrating graph visualization libraries or building a dedicated visualization module, users can interactively explore the graph structure, analyze relationships, and identify patterns and clusters. Visualizations provide an intuitive means of understanding complex graph data, facilitating decision-making and discovery of meaningful information.

Lastly, ensuring transactionality through ACID compliance is crucial for maintaining data integrity and consistency. Future work can involve implementing transaction support within the embedded graph database system, allowing users to execute multiple operations atomically and providing mechanisms for rollback and recovery in case of failures. This would guarantee the reliability and consistency of the data, especially in scenarios where data modifications need to be performed as a cohesive unit.

By considering these future possibilities and extensions, the project can evolve into a comprehensive and versatile embedded graph database solution. It can cater to diverse application domains, offering extended functionality, scalability, and ease of use. This would empower developers and users to leverage the full potential of graph databases, enabling advanced data analysis, insights generation, and decision-making based on rich and interconnected graph data.

7 References

[1] J. Pokorny, “Graph databases: Their power and limitations,” in *Computer Information Systems and Industrial Management*, K. Saeed and W. Homenda, Eds. Cham: Springer International Publishing, 2015, pp. 58–69.
[2] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, “Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries,” 2019. [Online]. Available: https://arxiv.org/abs/1910.09017
[3] R. Angles and C. Gutierrez, “Survey of graph database models,” *ACM Comput. Surv.*, vol. 40, no. 1, feb 2008. [Online]. Available: https://doi.org/10.1145/1322432.1322433
[4] R. Angles, “A comparison of current graph database models,” in *2012 IEEE 28th International Conference on Data Engineering Workshops*, 2012, pp. 171–177.
[5] D. Fernandes and J. Bernardino, “Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb,” in *Proceedings of the 7th International Conference on Data Science, Technology and Applications*, ser. DATA 2018. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2018, p. 373–380. [Online]. Available: https://doi.org/10. 5220/0006910203730380
[6] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, “A performance evaluation of open source graph databases,” in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. PPAA ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 11–18. [Online]. Available: https://doi.org/10.1145/2567634.2567638
[7] R. Angles, M. Arenas, P. Barcelo, A. Hogan, J. Reutter, and D. Vrgoc, “Foundations of modern query languages for graph databases,” *ACM Comput. Surv.*, vol. 50, no. 5, sep 2017. [Online]. Available: https://doi.org/10.1145/3104031
[8] F. Holzschuher and R. Peinl, “Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j,” in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, ser. EDBT ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 195–204. [Online]. Available: https://doi.org/10.1145/2457317.2457351
[9] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, “Pgql: A property graph query language,” in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2960414.2960421
[10] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt, “G-core: A core for future graph query languages,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1421–1432. [Online]. Available: https://doi.org/10.1145/3183713.3190654
[11] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, “Cypher: An evolving query language for property graphs,” in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1433–1445. [Online]. Available: https://doi.org/10.1145/3183713.3190657
[12] R. Angles, “The property graph database model,” in *AMW*, 2018.
13] A. Donkers, D. Yang, and N. Baken, “Linked data for smart homes: Comparing rdf and labeled property graphs,” in *LDAC 2020 Linked Data in Architecture and Construction*, ser. CEUR Workshop Proceedings, M. Poveda-Villalon, A. Roxin, K. McGlinn, and P. Pauwels,

Eds. CEUR-WS.org, 2020, pp. 23–36, null ; Conference date: 17-06-2020 Through 19-06-2020. [Online]. Available: http://linkedbuildingdata.net/ldac2020/
[14] R. Steinhaus, D. Olteanu, and T. Furche, “G-store: a storage manager for graph data,” Ph.D. dissertation, Citeseer, 2010.