Media Engineering and Technology Faculty
German University in Cairo

Distributed Graph Database

Bachelor Thesis

Author: Elshimaa Ahmed Betah
Supervisors: Dr. Wael Mohamed AbulSadat

Submission Date: 1 June, 2023

Media Engineering and Technology Faculty
German University in Cairo

Distributed Graph Database

Bachelor Thesis

Author: Elshimaa Ahmed Betah
Supervisors: Dr. Wael Mohamed AbulSadat

Submission Date: 1 June, 2023

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

---

Elshimaa Ahmed Betah
1 June, 2023

Acknowledgments

In the name of Allah, the Most Gracious, the Most Merciful (Qur'an ,1:1)

After thanking Allah, I want to express my gratitude to my family. Nothing could have been done without their support. I want also to thank my friends for their suggestions and help throughout the thesis work. Finally, I want to thank Dr. Wael for giving me the chance to work on such a project and for his continuous help and feedback.

Abstract

Graph Databases are evolving in recent years motivated by the need to process and analyze interconnected data that could be modeled as graphs. In addition, the increasing volume of data that demands processing has become a powerful driving force behind the ongoing research and development of distributed systems for graph databases. This paper introduces a distributed graph database called **GraphoPlex**. The system runs in a distributed manner to store and process users' graph data. Users could interact with the system through a supported query language. Finally, performance tests show the efficiency of the implemented system compared to non-distributed graph databases and relational databases.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, in our data-driven world, various industries and domains are recognizing the immense value of graph databases. Social Networks, Recommendation Systems, and Bio-informatics are examples of such domains that could benefit from using a graph database to store and process their intricate and interconnected data. Unlike traditional SQL(tabular) databases, graph databases provide flexible semantics that aligns with business models in addition to eliminating the unnecessary overhead associated with SQL databases when querying complex and interconnected data. In addition, the increasing amount of data needed to be stored requires graph database systems to be distributed across multiple servers

1.2 Aim

This thesis aims to implement a graph database engine that is distributed across multiple nodes (servers). Users should be able to interact with the system through a supported query language. The system should allow users to upload, update and delete their data as well as to execute queries with the semantics of describing graph-like data. The system should be able to store large amounts of graph-like data and ensure timely responses to user queries, providing prompt and efficient retrieval of information.

1.3 Thesis Overview

The thesis work is organized as follows. Firstly, chapter 2 is dedicated to discussing the background of NoSQL and graph databases. It also includes a brief description and comparison between different implementations of existing distributed and non-distributed

graph database engines. Secondly chapter 3 shows the architecture of the implemented system alongside the set of supported commands. It also includes implementation details for the different modules of the system. Consequently, chapter 4 discusses the results of testing the performance of the implemented system. Finally, chapter 5 concludes the work of the thesis by mentioning achievements, limitations, and possible future improvements.

Chapter 2

Literature Review

2.1 Background

The shortcomings of conventional databases, specifically the relational model, in meeting the demands of modern application areas have prompted the creation of novel technologies known as No SQL databases. These databases are classified based on their data model and include Wide-column stores that follow Google's Big-Table model (such as Cassandra), Document stores that are designed to store semi-structured data (such as MongoDB), Key-value stores that use a persistent map for data indexing and retrieval (such as Berkeley DB), and Graph Databases that specialize in storing graph-like data (Angeles, 2012)[1].

A graph is a collection of nodes (vertices) that could hold some properties connected with edges that represent relationships between two nodes (Figure 2.1). As mentioned before graph database is categorized as No SQL databases that is well known for handling complex and interconnected data.

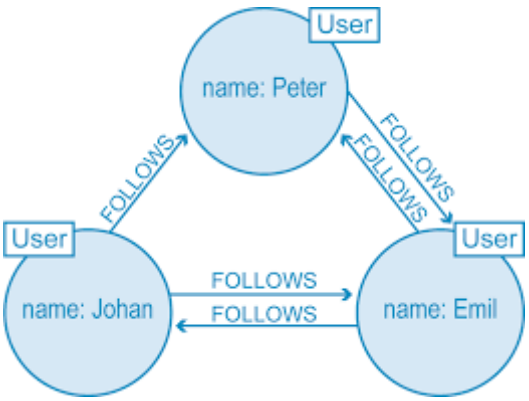


Figure 2.1: Basic Attributed Graph Example

Meng et al (2021) [11] suggested that the purpose of a graph database is not to replace the Relational database, but rather to offer an alternative solution for querying certain intricate data structures. Graph databases have their own set of unique use cases, which often involve significant performance overhead if a MySQL database is utilized instead. For example, developers of social network applications frequently encounter the need to explore data relationships using complicated SQL queries. These queries typically require joining multiple tables, which can significantly diminish the system's performance (Ho et al., 2012) [7].

Using a graph database has many advantages. The graph model has an inherent index data structure, which means it doesn't have to load or access unnecessary data when executing queries based on certain conditions. This is accomplished by treating nodes and edges equally and using two-way pointers and native graph storage to create a table. As a result, the query will only interact with the data that is relevant to the query[11]. Moreover, in a graph database, you have the freedom to define any type of vertex to represent objects and define edge types to represent specific relationships. The advantage of the graph model is that it allows for flexible semantics that align precisely with your needs, without the need for standardization or wasted effort [11].

Despite graph databases' advantages in managing complex and interconnected data, graph databases suffer from some potential disadvantages and challenges worth considering, such as:

1. Lack of optimization of storage policy: A significant disparity in the performance of graph databases can be observed due to the influence of hardware components. Currently, graph database technology lacks a mechanism for either developing or utilizing SSD. For Example, Neo4j graph database employs a mechanism of storing data in a mechanical hard disk, loading it into memory during querying, and performing calculations on the data once it is completely loaded, which necessitates a high memory capacity [11].

2. Cache Technology is developing slowly: Several graph database caching technologies currently in use are derived from relational database caching methods, such as Neo4j file caching system which utilizes MySQL caching technology and lacks any graph-specific features, resulting in ineffective improvements to query speed[11].

3. The need for mature distribution strategy: Due to its size, it is not feasible to store such a massive graph on a single machine, necessitating distributed solutions such as Titan, which adopts an embedded strategy to relinquish storage of graph data and rely on HBase for data storage [11].

Optimizing cache technology, adopting new distributed strategies, and utilizing new storage materials are crucial not only for the success of graph databases currently in development but also for determining whether graph databases can effectively compete with other databases in the future[11].

2.2 Related Work

2.2.1 Neo4j

Neo4j is a robust (fully ACID) transaction-based property graph database that is open-source and written in Java. A graph in Neo4j is made up of many nodes and relationships. Both nodes and relationships can have properties expressed as key-value pairs [8]. Neo4l is a pointer-chasing graph storage model that is disk-based. It stores graph vertices and edges in a denormalized, fixed-length structure and visits them using pointer-chasing rather than index-based methods[15].

The Neo4j system consists of several components (Figure 2.2).The lowest layer is disks. As mentioned before, Neo4j stores its data on disk in the form of fixed record files. To improve the speed of accessing data, Neo4j system utilizes 2 layers of caching(file system cache and object cache). There also exists a transaction module that is responsible for logging and ensuring Atomicity, Consistency , Isolation and Durability (ACID) of the transaction. For Availability, HA(High Availability) module is used to replicate data between multiple machines of a cluster with a Load Balancer deployed in front of a cluster of machines to provide high availability. On the top, there is the API layer that provides the interface for clients[8].

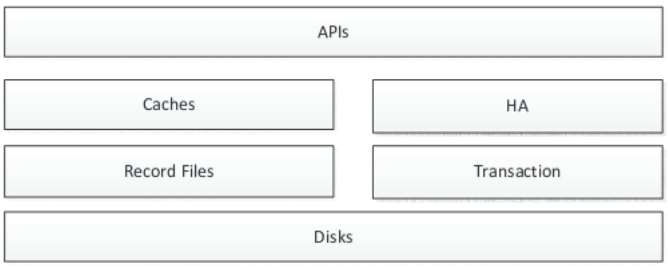


Figure 2.2: Neo4j System Architecture [12]

Neo4j is a native graph store that uses Index-Free Adjacency to store graph data(Figure 2.3)

1. *Nodes:* The nodes in Neo4j are stored in a file named "neostore.nodestore.db" on disk. The node store file consists of fixed-size records of size 9 bytes.Neo4j uses fixed-size records to enable a fast search of a specific node. For Example, if we want to get a node with id 100 we simply access the record starting from byte number 900. Nodes in Neo4j only store a pointer for the first edge and first property; because of the limited size each node has and also to enable fast traversal[8].

2. *Relationships:* In the Neo4j database, relationship data is saved in a file called "neostore.relationshipstore.db" on the disk. The relationship store file is similar

to the node store file in that it consists of fixed-size records that are 33 bytes each. Each record contains the IDs of the nodes at the beginning and end of each relationship. Each record in the relationship store file of the Neo4j database also includes pointers to the next and previous relationships for both the start and end nodes of the relationship. This enables efficient navigation of relationships in both directions for each node[8].

3. *Properties:* The data of properties in Neo4j database is stored in the file named "neostore.propertystore.db on disk".Each record has a pointer to a property index store file(neostore.propertystore.db.index) which contain the name(key) of the property. the value of the property is stored within the record itself or referencing a dynamic store record(being either in the Dynamic String Store in "neostore.propertystore.db.strings" or the Dynamic Array Store in (neostore.propertystore.db.arrays) depending on the property type)[8].

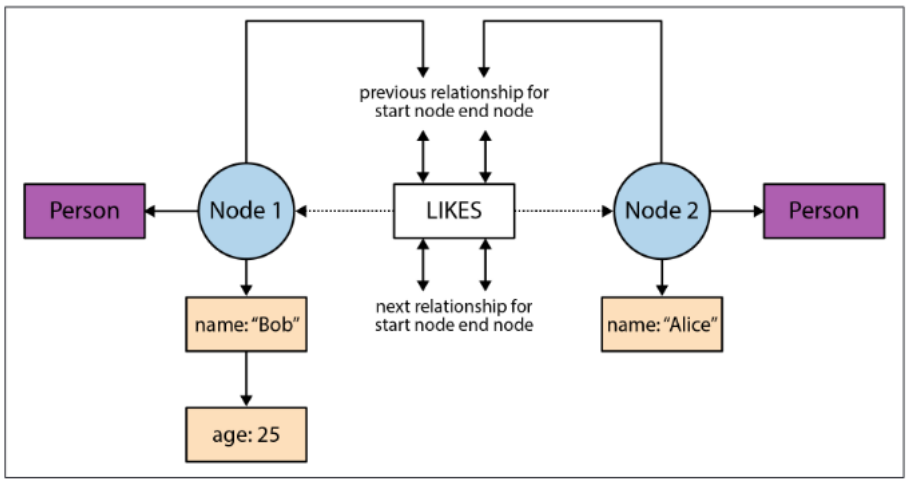


Figure 2.3: How a graph is physically stored in Neo4j [**neo4j**]

2.2.2 Distributed Graph Databases

Neo4j is a robust and native graph database system but it is not distributed. It is not suitable for too large graphs that cannot be stored in one machine. That is the purpose of the upcoming systems reviewed in this section.

Trinity (Shao et al., 2013) [14] is a graph engine designed for general purposes, capable of running on a distributed memory cloud. The performance of Trinity is optimized through efficient network communication and memory management that enables fast graph traversal with efficient parallel computing. Trinity utilizes a distributed memory storage that is globally addressable which provides abstract random access to process big graphs [14].

In a Trinity system (Figure 2.4), there are multiple components that communicate over a network. These components are divided into three types based on their roles: slaves, proxies, and clients. Each slave is responsible for storing a part of the data alongside processing messages from other slaves, clients, or proxies. On the other hand, a Trinity proxy is responsible for handling messages without storing any data. It acts as a middle layer between slaves and clients. A Trinity client enables users to interact with the Trinity cluster. These clients are applications that use the APIs offered by the Trinity library to communicate with Trinity slaves and proxies. [14]

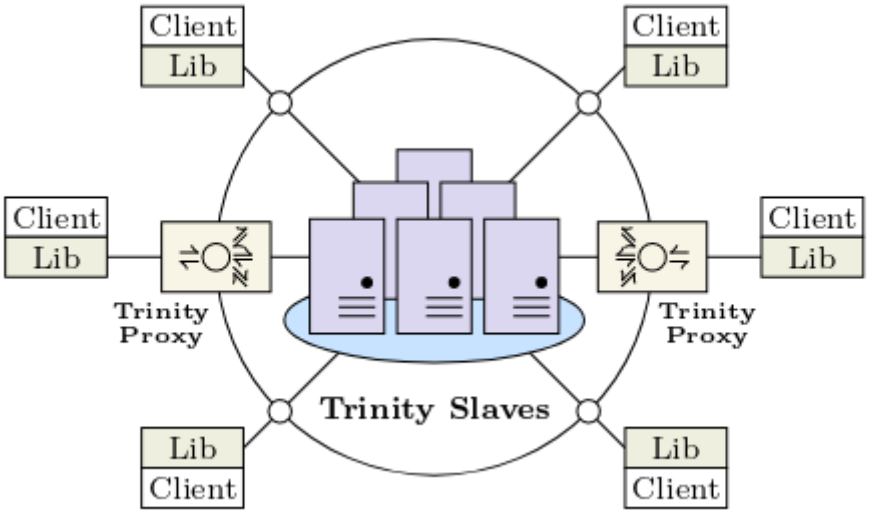


Figure 2.4: Trinity System [14]

Trinity uses a distributed memory cloud as its storage infrastructure. The memory cloud consists of 2^p memory trunks. Trinity system consists of a key-value store. This key-value store is implemented on top of the distributed memory cloud. Keys consist of 64-bit globally unique identifiers. Those keys can be addressable using a hashing mechanism, Keys are hashed to $p-$ bit value i in $i \in [0, 2^p - 1]$. This means the key is stored in trunk number i. An addressing table is used to locate trunk i in one of the machines [14].

Trinity stores graph data on top of the key-value store described before. Keys are 64-bit identifiers while values are blobs of arbitrary length. The (key, value) pair becomes (cellId, cell) pair. To represent graphs using a key-value store, a node in the graph can be implemented using a cell. A cell can hold a significant amount of information. In undirected graphs, each cell that represents a graph node has a set of cell Ids representing its neighboring nodes. In directed graphs, each cell has instead two sets of cell IDs to store both incoming links and outgoing links [14].

Ho et al.(2012) [7] developed another distributed graph database targeting large-scale graph data processing for social computing. The structure of the system consists of two

main sub-systems: a distributed graph data processing system and a distributed graph data store.

GoldenOrb, which is an open-source implementation of *Pregel*[10], is utilized as the distributed graph data processing system. In contrast, the back-end storage layer of *GoldenOrb* is not suitable for graph processing since it uses the Hadoop file system. As a solution, an InputFormat interface is implemented, and the system is instead connected to a distributed graph data storage system that was specifically implemented for this purpose [7].

There are 3 layers in the distributed graph data storage system (Figure 2.5): *Front End Layer*, *Middleware Layer* and *Backend Storage Layer*. The *Front End Layer* includes a *name server* and a *metadata server*. The *name server* is responsible for looking up necessary data from the *metadata server* upon receiving queries then sending it to *query manager* for processing. The responsibility of *metadata server* is to store the topology of the entire graph alongside the locations of all nodes and edges only without their actual data. The middleware layer comprises four components: a *partition manager* for graph partitioning, a *replication manager* for the replication of graph data, a *query manager* for processing graph queries, and a *job manager* for queuing all I/O operations to the backend store to execute these operations from the queue. In the end, the *backend storage layer* consists of a number of stand-alone Neo4j graph databases. [7].

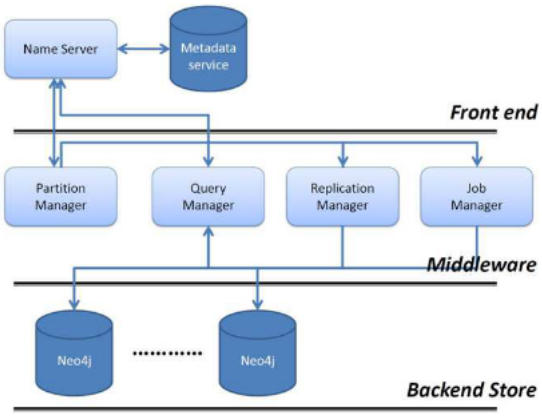


Figure 2.5: Distributed Graph Storage Architecture [7]

Rather than using hashing for graph partitioning, METIS [9] is employed to partition the graph, as it ensures a balanced workload among partitions and minimizes the number of cut edges[7].

A1 (Buragohain et al., 2020) [3] is a type of distributed database that the Bing search engine uses in memory to handle complicated queries on structured data. A1 is made possible thanks to the low cost and availability of Dynamic Random Access Memory (DRAM)

and high-speed Remote Direct Memory Access (RDMA) in commodity hardware. A1 utilizes Fast Remote Memory (FaRM) (Dragojević et al., 2014) [5] as its storage layer and builds its query engine on top of it.

A1 is able to function because of two main hardware trends which are the availability of low-cost DRAM and high-speed networks that support RDMA. To take advantage of these trends, A1 uses machines with hundreds of gigabytes of DRAM, which means that a set of racks can hold more than 200 terabytes of memory[3].

A1 follows a standard layered architecture (Figure 2.6), starting with the networking layer at the bottom and ending with the query processing at the top. The bottom four layers of the architecture work together to create a distributed storage platform named FaRM, which provides transactional storage with generic indices structures. The above layers of A1 provide the core graph data structures and a graph query engine[3].

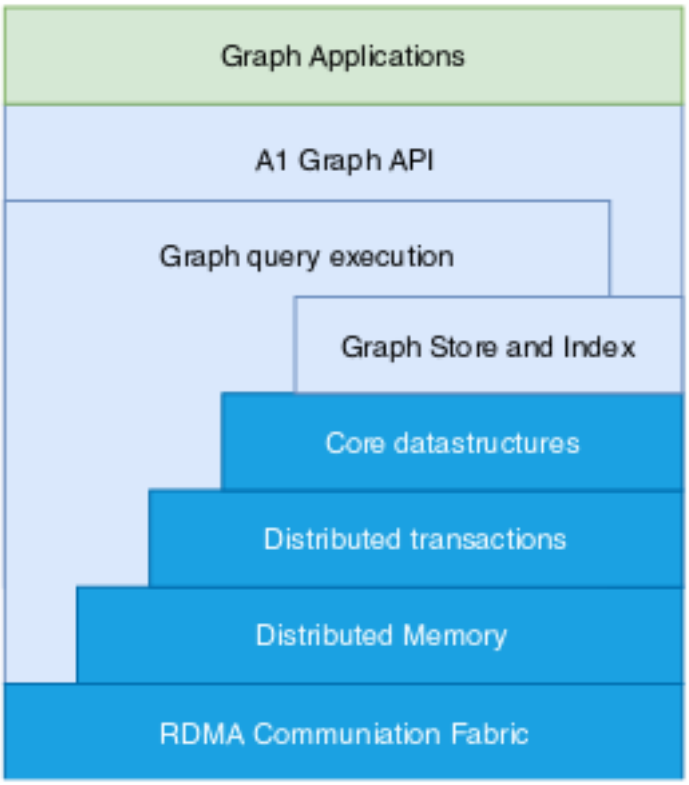


Figure 2.6: A1 System Architecture [3]

The A1 storage model uses a graph that consists of vertices and directed edges which can be typed and have attributes. These attributes are properties associated with them, and the vertex/edge type defines the schema of these attributes. Unlike the Neo4J property graph model, the schema is enforced. The A1 storage model optimizes traversal queries over index lookups since they are more frequent. Each vertex is stored as two

FaRM objects (Figure 2.7), namely the header and the data object, with the header containing the vertex type, pointers to some data structures that hold outgoing and incoming edges associated with the vertex, and a pointer to the data of the vertex [3].

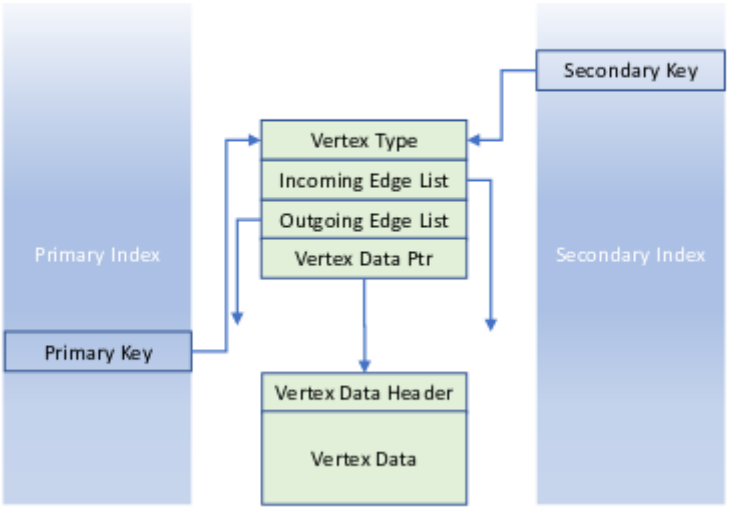


Figure 2.7: A1 vertex storage [3]

System G (Tanase et al., 2018) [15] comprises a single node graph database and a runtime and communication layer that facilitates the creation of a distributed graph database by combining multiple single node instances.

The storage model consists of each graph named as specified by the user, which includes vertices, edges, and their respective properties. User-specified external vertex identifiers are used to uniquely identify each vertex, along with automatically generated internal vertex identifiers. Each edge is identified by the vertex IDs of its source and target vertices, along with a unique edge ID automatically generated for it. To store these representations in both memory and on disk, a high-performance key-value store called Lightning Memory-Mapped Database (LMDB) is utilized [15].

To keep track of individual vertices, two key-value store databases are used: $ex2i$ (external id-to-internal id) and $i2ex$ (internal id to external id). On the other hand, the $vi2e$ key-value database is responsible for managing edges. In this storage schema, the internal identifier of the source vertex is used as the key to locate edges that go out from that vertex. For each edge, the value stored includes the internal identifier of the target vertex, label identifier, and edge identifier[15].

In the distributed version of the **System G** graph database, a fixed set of single node graph databases, known as shards, are combined to create the distributed graph. This distributed graph keeps track of a list of computation nodes, as well as the mapping of shards to nodes. An API is also implemented in such a way that users only interact with

a single database instance, despite it being a distributed system. By default, the graph distributes its vertices based on a hash function applied to its external vertex identifiers. Similarly, an edge is located based on its source vertex. As a result, a typical distributed graph method computation will first determine the shard where a specific vertex or edge is located or will be located. Once the shard is determined, the method invocation is forwarded to that shard using Remote Procedure Call (RPC) to complete the method execution [15].

Acacia (Dayarathna & Suzumura, 2014) [4] is another distributed Graph Database with an aim to work between the boundaries of private and public clouds.

Acacia divides the graph into sub-graphs ,using METIS (Karypis & Kumar, 1998) [9], that match the number of workers it uses. These sub-graphs are then distributed to individual workers, where each worker is an independent Neo4j graph store that is not distributed.

Acacia System (Figure 2.8) is built using the Master-Worker pattern, which provides greater control over how the system is executed, especially when compared to other approaches like P2P for medium-scale deployments.

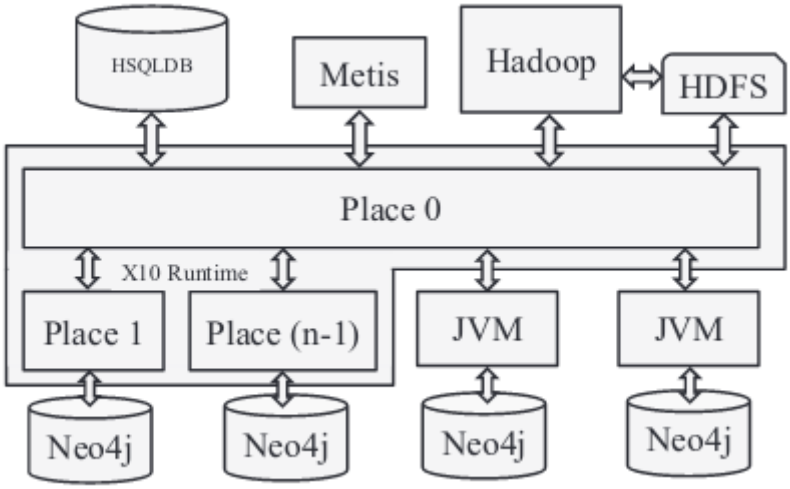


Figure 2.8: Acacia System Architecture [4]

Graph partitioning benefits from the load balancing of computation that is achieved by running a graph processing algorithm concurrently on multiple workers. *Acacia* implements graph partitioning through a sequence of Map Reduce jobs, which are executed on Hadoop with the support of METIS. As the current version of Acacia stores graphs as edge lists, the partitioned graph edges are then stored on local Neo4j while cut-off edges, edges lying on two different partitions, are stored on the central node [4].

Acacia system comprises two main components (Figure 2.9), namely the Master and the Worker. The front end of *Acacia* is a command-line interface that allows users to execute predetermined commands such as graph upload/deletion, statistics retrieval, and graph algorithm execution. Within the Acacia master, a data loader component resides, which carries out the conversion of plain edge list files to METIS-compatible files, partitions them using METIS, and distributes the resulting sub-graphs between the central store and the workers. The partitioning process involves the Vertex Processor, CSR Converter, and Partitioner components of Acacia's Data Loader, with the support of Hadoop and HDFS, solely for data loading purposes. The MetaDB component is responsible for managing the stand-alone HSQLDB-based metadata store, while the Central Store component consists of a number of HSQLDB-embedded databases that serve as Acacia's central storage. The communication between the masters and workers is managed by two protocols: the Worker Protocol for communication from master to worker, and the Back-end protocol for communication initiated by workers to the master. When running a graph algorithm, the master tells the workers to execute the algorithm in parallel on the given data set. In the current version of Acacia, graph algorithms are implemented with a hybrid mix of Java and X10 on the worker, separately from Neo4j's APIs, although the local graph storage is a Neo4j embedded server. This setup enables the use of heterogeneous hardware, such as GPUs, for graph algorithm optimization.

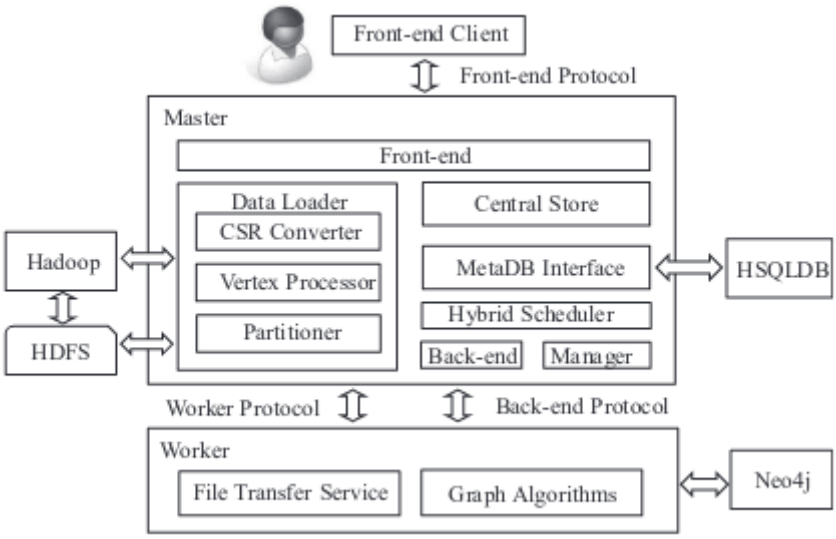


Figure 2.9: Acacia System Components [4]

2.3 Summary

In this chapter, we have discussed NO SQL database types. Graph Databases is one of those NO SQL databases. We have also shown the advantages and use cases of using Graph databases instead of Relational Databases when querying interconnected data.

Challenges that prevent current graph database systems from competing with other mature database systems were mentioned.

Neo4j was mentioned as a transactional and native graph database system. Neo4j has a native storage graph-like data that persists on the desk. On the other hand, It's a single server system so It faces some challenges to scale horizontally to store big graphs.

In all distributed graph databases that have been discussed, They delegate storage of data to some other system. Ho et al.(2012) [7] and Acacia (Dayarathna & Suzumura, 2014) [4] both are using multiple stand-alone Neo4j instances as a storage layer to store graph data while they are both implementing Pregel run time as a graph processing layer with a focus from Acacia [4] on scaling the system to work on hybrid cloud.

Trinity(Shao et al., 2013) [14] and A1 (Buragohain et al., 2020) [3] are both in-memory systems that make use of recent research and development of DRAM. While A1 utilized FaRM as its low latent distributed storage layer, Trinity uses distributed cloud memory that is globally addressable to build their query engine on top of it.

System G (Tanase et al., 2018) [15] utilized a key-value store called LMDB as a storage layer.Tanase et al.(2018) [15] extended their single-node system to be distributed by combining multiple similar nodes together with a communication layer through RPC between the nodes of the system.

Chapter 3

System Design and Implementation

This chapter discusses the system's design goals, architecture design, and implementation details of its base modules.

3.1 Design Goals

The final system is a distributed graph database engine called **GraphoPlex** that utilizes a directed property graph model to store graph-like data. This model comprises a group of vertices capable of retaining specific properties and edges that are directed and capable of holding certain properties as well. The system comprises numerous nodes or servers where the data is partitioned, and communication occurs among them to facilitate horizontal scalability and parallel processing.

The system's main objective is to offer users a database management system for graph-like data. This will allow them to load their data and execute match and shortest path queries through a supported query language.

3.2 System Overview

3.2.1 GraphoPlex Architecture

GraphoPlex consists of a client and a cluster of servers (Figure 3.1). The system's client is an interactive shell created using Python where the user could type his/her commands from a collection of supported commands (subsection 3.2.2).

The system's cluster is a collection of multiple similar servers(nodes). Each server is a spring boot application running the same code with different environment variables (Ex. server_id) that distinguish one server from another. There is no master in the cluster. Each server could receive and process users' queries. Each server holds a sub-graph of the

user's stored graphs. Therefore, servers should communicate with each other to process user queries.

The client sends the user's query through an HTTP request to a server from the cluster where this query gets parsed and executed. After processing the query which could involve inter-cluster communication, The result is returned by the server to the client where the client is responsible for visualizing results to the user.

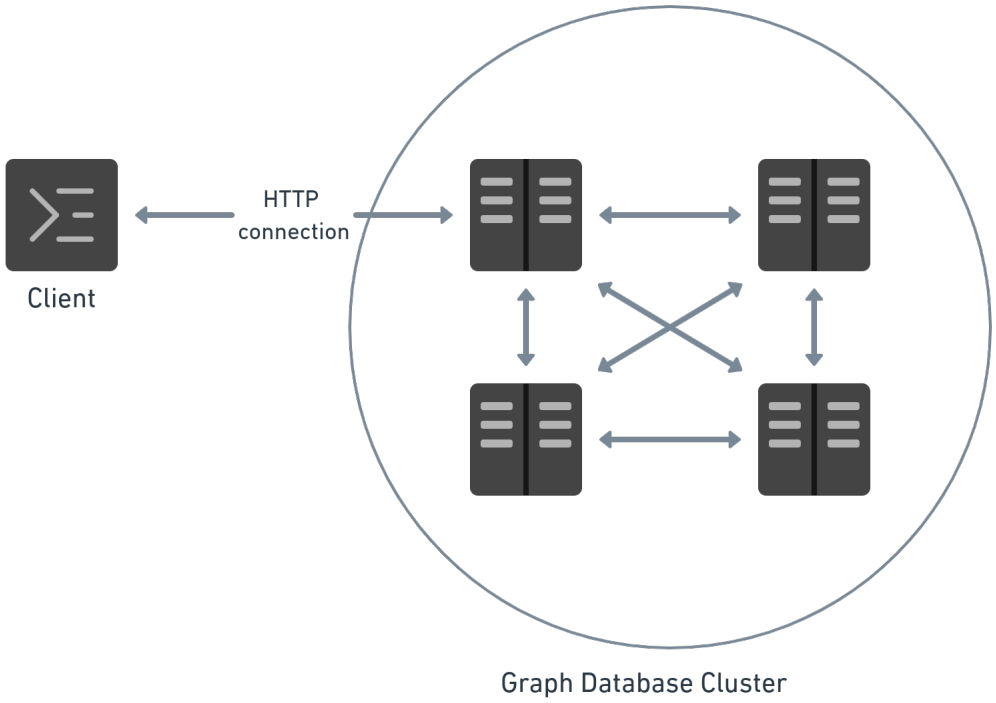


Figure 3.1: GraphoPlex Architecture

Since there is no master server in the cluster. The client could connect to any server in the cluster to send user queries.

3.2.2 Supported Commands

Database Configuration Commands

GraphoPlex supports users with the ability to create multiple databases(graphs). Therefore, there should be supported commands for creating, switching, and deleting existing databases

1. **Create Database:** The user can create a new database by specifying its name. (Figure 3.2). If the user tries to create a database that already exists, an error message will be received.

```
distributed_graphdb > CREATE DATABASE graph1
Database graph1 created
Total Execution Time: 182ms
```

Figure 3.2: Create Database Command

2. **Switch Database:** The user can switch the current working database to another one by specifying its name. The user can't switch to a non-existing database that was not created before.

```
distributed_graphdb > SWITCH DATABASE graph1
Database switched to graph1
Total Execution Time: 58ms
```

Figure 3.3: Switch Database Command

3. **Get Current Database:** This command is used to get the name of the current working database (Figure 3.4).

```
distributed_graphdb > GET CURRENT DATABASE
Current database is graph1
Total Execution Time: 1ms
```

Figure 3.4: Get Current Database Command

4. **Drop Database:** This command is used to delete all vertices, edges, and indices associated with a given database name (Figure 3.5).

```
distributed_graphdb > DROP DATABASE graph1
Database graph1 dropped
Total Execution Time: 38ms
```

Figure 3.5: Drop Database Command

5. **Delete Database:** This command is used to delete all vertices, edges, and indices associated with a given database in addition to deleting the database itself from the list of existing databases in the system.

```
distributed_graphdb > DELETE DATABASE graph1
Database graph1 deleted
Total Execution Time: 62ms
```

Figure 3.6: Delete Database Command

Vertex Commands

Creating, updating, and deleting vertices are the fundamental commands that should exist in any graph database.

1. **Create Vertex :** The user can create a vertex by specifying an id. It's optional for the user to add a label and key-value properties when creating a vertex (Figure 3.7). An error message will be received when trying to create an existing vertex.

```
distributed_graphdb > CREATE VERTEX (v1:PERSON {name:test1, age:20})
Vertex with id v1 created
Total Execution Time: 189ms
```

Figure 3.7: Create Vertex Command

2. **Update Vertex :** After creating a vertex. It's possible to update an existing property or add a new one. (Figure 3.8)

```
distributed_graphdb > UPDATE VERTEX v1 SET age=21, hobby=football
Vertex with id v1 updated
Total Execution Time: 18ms
```

Figure 3.8: Update Vertex Command

3. **Delete Vertex :** A vertex could be deleted by only specifying its id (Figure 3.9)

```
distributed_graphdb > DELETE VERTEX v1
Vertex with id v1 deleted
Total Execution Time: 12ms
```

Figure 3.9: Delete Vertex Command

Edge Commands

1. **Create Edge:** The user can add an edge between two existing vertices by specifying source id, destination id, and label. Those 3 elements act as an id for the edge in the system (Figure 3.10). There are no two edges in the system that share those 3 variables. If the user tries to create an edge with a source or destination id that does not exist, an error message will be received.

```
distributed_graphdb > CREATE EDGE follow FROM v1 TO v2 WITH {duration:1}
Edge between v1 and v2 with follow label created
Total Execution Time: 48ms
```

Figure 3.10: Create Edge Command

2. **Update Edge:** The user could update an existing edge by specifying source, destination, and label followed by a set of key-value properties to be added or updated (Figure 3.11).

```
distributed_graphdb > UPDATE EDGE follow FROM v1 TO v2 SET duration = 2
Edge between v1 and v2 with follow label updated
Total Execution Time: 22ms
```

Figure 3.11: Update Edge Command

3. **Delete Edge:** An edge could be deleted by specifying its id which is source, destination, and label (Figure 3.12).

```
distributed_graphdb > DELETE EDGE follow FROM v1 TO v2
Edge between v1 and v2with follow deleted
Total Execution Time: 14ms
```

Figure 3.12: Delete Edge Command

Secondary Index Commands

The system supports creating secondary indices on fields (property key) of vertices. There is also a default secondary index created on the label of vertices.

1. **Create Index:** The user could create an index by specifying which field to create an index on (Figure 3.13).

```
distributed_graphdb > CREATE INDEX ON age
Index Created on age
Total Execution Time: 52ms
```

Figure 3.13: Create Index Command

2. **Delete Index:** An index created before could be deleted by providing the field name (Figure 3.14).

```
distributed_graphdb > DELETE INDEX ON age
Index Deleted on age
Total Execution Time: 21ms
```

Figure 3.14: Delete Index Command

Match Commands:

Match Commands are used to match a subgraph (a chosen set of vertices and edges) of the entire graph based on the user's query. Graph algorithms are used to answer those queries.

1. **Path Command:** The user could query for some sub-graph by describing a path along the graph. A path of length n will consist of n vertex descriptions and $n-1$ edge descriptions that connect between vertices. And the query result will be all subgraphs that match the path description. An empty description could match with any vertex or edge.

   Figure 3.15 shows the query that matches all the vertices in the graph. This query describes a path with length 1 with no filters to select vertices based on.

```
distributed_graphdb > MATCH [() AS all_vertices] RETURN all_vertices
all_vertices                                                        |
--------------------------------------------------------------------+-
Vertex [id=v2, label=PERSON, properties={name=test2, age=22}] |
Vertex [id=v1, label=PERSON, properties={name=test1, age=21}] |

2 match(s) found
Total Execution Time: 16ms
```

Figure 3.15: Path Query that matches all vertices

   Filters can be added to the previous query to select vertices based on some filters on properties (Figure 3.16).

```
distributed_graphdb > MATCH [({age=22}) AS filtered_vertices] RETURN filtered_vertices
filtered_vertices                                                   |
--------------------------------------------------------------------+-
Vertex [id=v2, label=PERSON, properties={name=test2, age=22}] |

1 match(s) found
Total Execution Time: 17ms
```

Figure 3.16: Path Query of length 1 with filters

   The user can describe a path of length more than 1. This can be done by adding an edge description between every two adjacent vertex descriptions. Edge description can consist of a label and some select operators. Figure 3.17 shows a query that returns all the following list of vertex v1.

```
distributed_graphdb > MATCH [(v1)] -follow-> [() AS following] RETURN following
following                                                                |
-------------------------------------------------------------------------+-
Vertex [id=v2, label=PERSON, properties={name=test2, age=22}] |

1 match(s) found
Total Execution Time: 28ms
```

Figure 3.17: Path Query of length 2 Example

The direction of edge descriptions could be reversed. It allows for more flexible queries to be formed. For example, Figure 3.18 shows a query where all followers of vertex v2 are returned. This query makes use of the ability to reverse the edge direction to describe an incoming edge within the path.

```
distributed_graphdb > MATCH [(v2)] <-follow- [() AS followers] RETURN followers
followers                                                                |
-------------------------------------------------------------------------+-
Vertex [id=v1, label=PERSON, properties={name=test1, age=21}] |

1 match(s) found
Total Execution Time: 32ms
```

Figure 3.18: Path Query of length 2 with incoming edge Example

2. **Shortest Path Command:** The user could query for the shortest path between any two vertices by specifying their ids. In addition, the user should tell the engine which property on the edges represents the cost (weight) that should be minimized. The returned result is the edges that form the shortest path followed by the cost of that path (Figure 3.19).

```
distributed_graphdb > MATCH SHORTEST PATH FROM cairo TO alex WITH COST = cost
Shortest path: cairo ----> benha {cost=1},benha ----> alex {cost=2}

Shortest path cost: 3
Total Execution Time: 88ms
```

Figure 3.19: Shortest Path Query

3.3 Server Implementation

Each server (node) of the cluster mainly consists of 4 main modules besides Core data structures and methods. Those modules are *Storage Layer*, *Network Layer*, *Graph Algorithms Layer*, and *Query Manager Layer* (Figure 3.20).

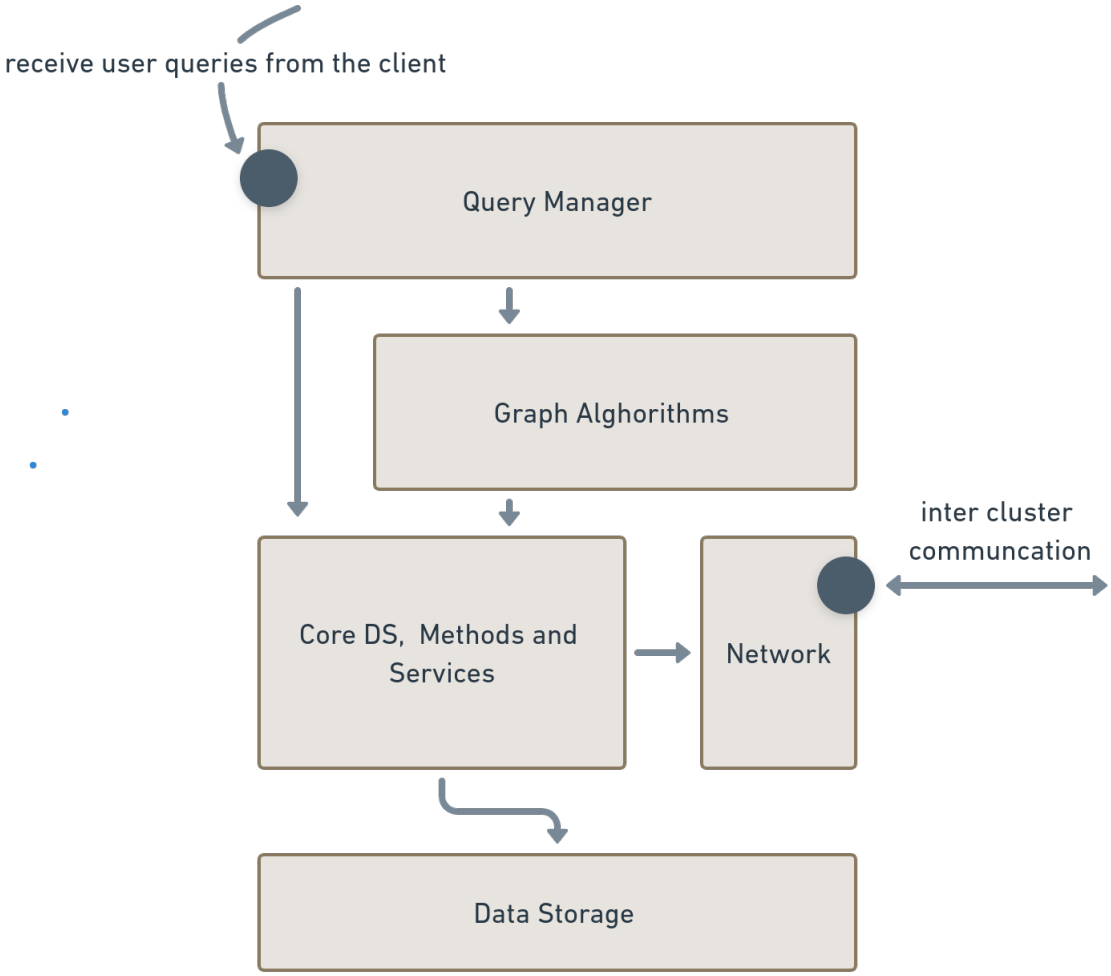


Figure 3.20: Single Server Architecture

The user interacts with the server solely via the Query Manager Layer. This layer builds the logical query object and then executes the query using the Core Methods and Services module alongside the Graph Algorithms module.

The Graph Algorithms module implements well-known graph algorithms to execute the user's queries. Currently, this module employs Breadth First Search (BFS), Depth First Search (DFS), Dijkstra, and A*. To carry out algorithm execution, the Graph Algorithms module uses Core DS and Methods to obtain and filter related data.

The Core DS and Methods are the only modules that engage with the Data Storage Layer through an interface. Core Services generally fetch data from the Data Storage Layer locally or transmit a request through the Network module to the servers that store the required data fragment.

3.3.1 Data Storage Layer

The Data Storage Layer plays a crucial role in the server, as it stores the graph's actual data, including vertices and edges, index data, and database configurations. The decision was made to use Redis to implement this layer. In this section, we will delve into the storage model and architecture and the rationale behind selecting Redis for this layer.

Redis

Redis [13] is a freely available in-memory data structure store that serves as a database, cache, streaming engine, and message broker. It is distributed under the BSD license. Redis has various data structures, including strings, lists, hashes sets, and sorted sets. In addition to its in-memory capabilities, Redis also supports persistence, allowing data to be saved on disk for durability. This feature makes it ideal for use cases where data must be retained even in the event of a system restart or crash. Furthermore, Redis is a key-value store, meaning that data is accessed using keys, making it simple to use and integrate with a wide range of applications.

Architecture

Different Redis data structures (Table 3.1) are used to store the actual graph data entered by the user in addition to data related to database configurations and secondary indices data

	Key	Redis Data Structure
Vertex	$databaseName:Vertex:$vertexId	Redis Hash
Outgoing Edges	$databaseName:$vertexId:outgoingEdges	Redis Set
Incoming Edges	$databaseName:$vertexId:incomingEdges	Redis Set
databases names	databases	Redis Set
Current Database	current_database	Redis String
Existing Indices	$databaseName:secondaryIndex	Redis List
Secondary Index	$databaseName:$indexField:$indexValue	Redis Set

Table 3.1: Data Store Architecture

1. **Vertex**: a vertex is stored as a Redis hash in the system's storage layer. Its key is composed of a prefix of database name followed by "Vertex" followed by the vertex id. As the system enables users to create multiple databases, a prefix of the database name is added to differentiate between vertices of different user's created databases.

2. **Outgoing Edges**: For each vertex, a collection of its outgoing edges is stored as a Redis Set with a key composed of the database name followed by the source

vertexId followed by "OutgoingEdges". Redis Set is used to enable fast retrieval of specific edges that enhances updating and deleting specific edges as it utilizes O(log(n)) complexity for getting specific elements.

3. **Incoming Edges**: For each vertex, a collection of its incoming edge is stored. The key used is the same for Outgoing Edges with a change that vertex id here is the destination vertex id for this edge and the key is suffixed by "IncomingEdges". Storing Edge twice in the source vertex set and destination vertex set may seem redundant. However, the power of that will appear when traversing the graph to execute match queries.

4. **Current Database**: The current database name is stored and synced in each server as a simple Redis String with a key "current_database". .

5. **databases names**: All different database names that the user creates are stored and synced in each server as a Redis Set with a key "databases".

6. **Existing Indices**: All vertex property field names that have secondary indices created by the user are stored in a Redis List with a key prefixed by the database name followed by "secondaryIndex."

7. **Secondary Index**: For each secondary index field created by the user, there is a number of sets equal to the number of unique values for this field. For each unique value, there is a set that has a key prefixed with the database name followed by the field name and the value. Each set contains vertices Ids that match the set's field value (Figure 3.22). To allow for fast deletion and update for the synchronization process between index and graph data, Redis sets are used over lists.

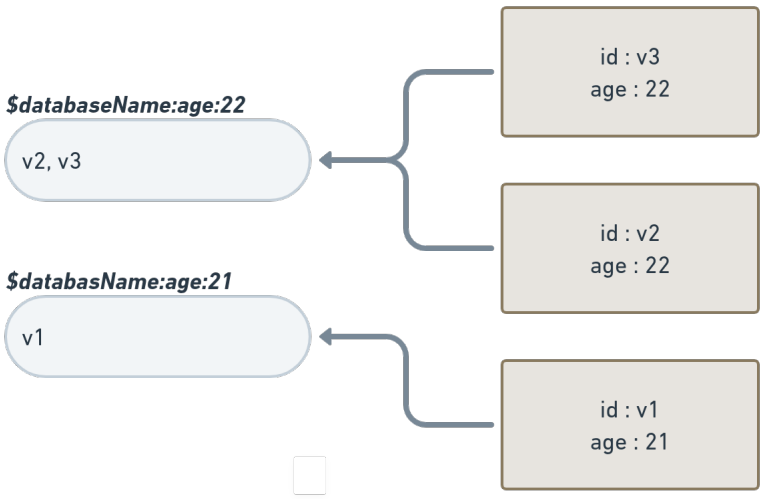


Figure 3.21: Secondary Index Example

Interface

The Data Storage module consists of 3 main classes *RedisDataAccess* which is responsible for all operations that deal with vertices and edges, *RedisDatabaseConfig* which deals with databases configuration operations and *RedisIndexDataManager* for secondary index-specific functions. All methods of the 3 classes are abstracted into 3 different interfaces to hide implementation details from the Core module and provide flexibility with the implementation and of the data store itself if needed to be altered in the future.

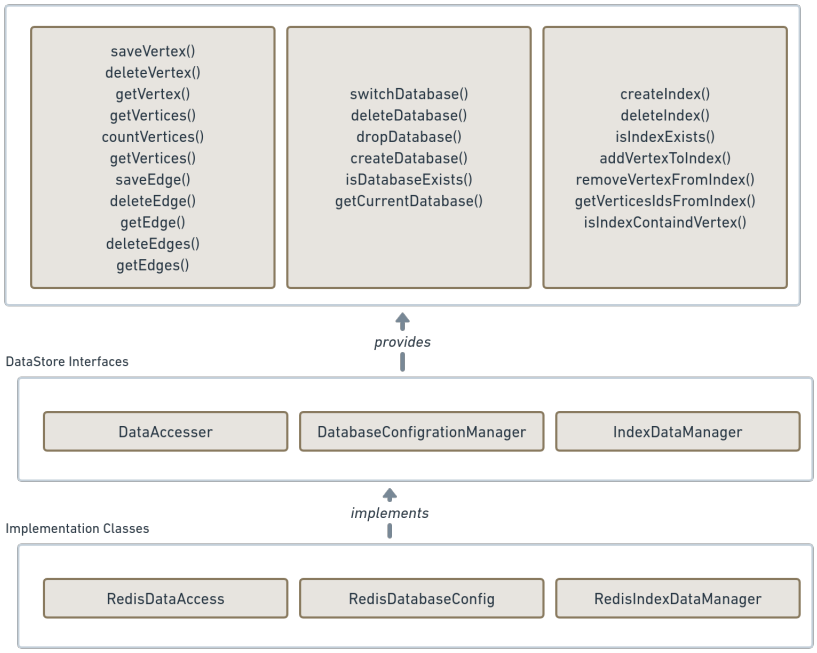


Figure 3.22: Data Storage Interface

Here is a brief description of how interface methods are implemented in Redis Implementation Classes

1. **DataAccesser Interface**

   (a) `void saveVertex (Vertex vertex);`

   This method is responsible for saving core Vertex object data (more info on how this class is structured in the section about Core module) in the data store. Redis implementation class implemented this method to save vertex objects in a Redis Hash.

   (b) `void deleteVertex (String vertexId);`

   This method typically takes the id of the vertex and deletes its corresponding hash from Redis

(c) `Vertex getVertex (String vertexId)`

This method takes vertex Id, retrieve it from Redis, and returns it in the form of a Core Vertex Object.

(d) `Iterable<Vertex> getVerticesById (Iterable<String> verticesIds)`

With a list of vertices Ids, this method is responsible for returning a list of all corresponding vertices.

(e) `int countVertices();`

Returns the count of vertices in this Local Data Storage

(f) `void saveEdge(Edge edge, boolean isOutGoing);`

Takes an object of type Edge that exists in the Core Module and saves that edge in the corresponding outgoing/incoming Redis set of the edge's source/destination vertex.

(g) `void deleteEdge (String sourceId, String destinationId, String label, boolean isOutGoing)`

This method uses source Id, destination Id, and label all together to act as an identification of the edge that needs to be deleted then delete it from its corresponding set.

(h) `Iterable<Edge> getEdges (String vertexId , boolean isOutgoing);`

This method takes vertex Id and a Boolean indicates whether the needed edges is outgoing or incoming then return all edges in the corresponding Redis set.

2. **IndexDataManager Interface**

(a) `void createIndex(String indexName);`

This method takes the name of the field on which the user wants to create a secondary index and pushes it to the Redis list of existing indices.

(b) `void deleteIndex(String indexName);`

This method takes the name of the field which the user wants to delete its index and remove it from the Redis list of existing indices, In addition to the removal of all Redis sets related to this field

(c) `boolean isIndexExists(String indexName);`

This method takes the name of the field and returns whether there is an index for that field or not.

(d) `void addVertexToIndex(String fieldName, fieldValue, String vertexId);`

This method uses the field name and value provided to identify the corresponding set and then add vertex id to that set.

(e) `void removeVertexFromIndex(String fieldName, fieldValue, String vertexId);`

This method uses the field name and value provided to identify the corresponding set and then remove the vertex id from this set.

(f) `boolean isIndexContainsVertex(String fieldName, fieldValue, String vertexId);`

This method uses the field name and value provided to identify the corresponding set and then returns whether vertex id exists in this set or not.

3. **DatabaseConfigrationManager Interface**

(a) `void createDatabase(String databaseName);`

This method adds the given database name to the database's Redis Set.

(a) `void dropDatabase(String databaseName);`

This method removes all Redis keys that are prefixed with the given database name. This includes all vertices, edges, and indices that were associated with the given database.

(b) `void switchDatabase(String databaseName);`

This method change value stored in the "current_database" key to the given name.

(c) `void deleteDatabase(String databaseName);`

This method does drop the database given as well as removes its name from the Redis set of databases.

(d) `String getCurrentDatabase();`

This method returns the name of the current database. This method typically gets called by previous data store classes to prefix all keys with the name of the current database.

(e) `boolean isDatabaseExists(String databaseName);`

This method returns whether or not a given database exists in the Redis set of existing databases.

3.3.2 Network Layer

Network Layer plays an important role. As *GraphoPlex* operates in a distributed manner, it becomes crucial for all servers within the cluster to possess the ability to establish communication with each other. This is facilitated by the Network module which acts as the interface for this inter-cluster communication. This section is dedicated to discussing how Network Layer is implemented using gRPC.

gRPC

gRPC [6] is RPC framework that allows a gRPC stub (client) to call a method on a remote server. That it is used to create distributed services easily. gRPC is implemented on top of the concept of defining a service that involves listing the methods that can be

called remotely alongside their respective parameters and return types. This interface is implemented by the gRPC to manage incoming client requests. Conversely, on the client side, a stub (client) is used to access the same set of methods as those provided by the server (Figure 3.23).

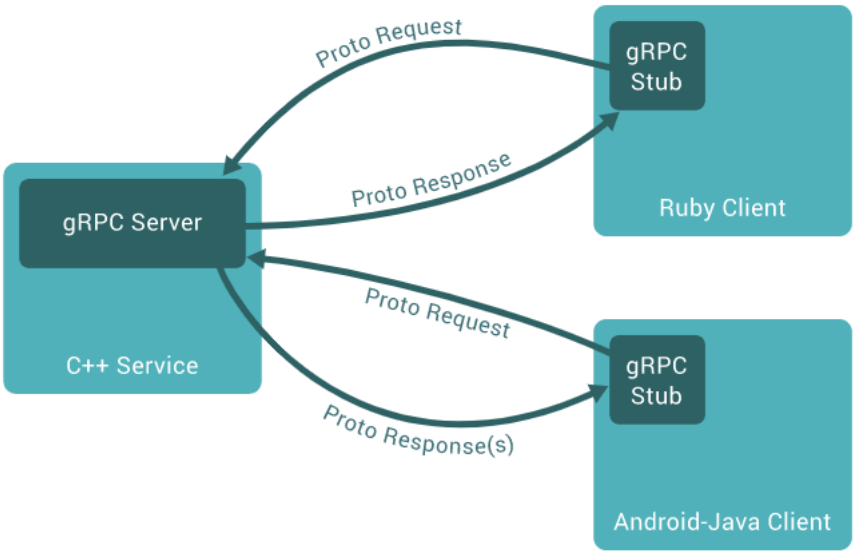


Figure 3.23: gRPC example [6]

gRPC uses Protocol Buffers as a messaging layer between stubs and services to serialize structured data. To begin working with protocol buffers, the first step involves defining the structure for the data that needs serialization in a .proto file. This file is a simple text file with the .proto extension. In protocol buffers, the data is structured as messages as each message represents a small logical piece of information, comprising several name-value pairs, known as fields. The protocol buffer compiler protoc is used to compile these .proto files to generate data access classes in any preferred language(s) based on the proto definition. The RPC method parameters and return types for gRPC services should be defined as protocol buffer messages in regular .proto files.

Architecture

As mentioned before, Network Layer in our system is implemented using the gRPC framework. This module is typically composed of several sub-modules Figure 3.24 listed and described as below:

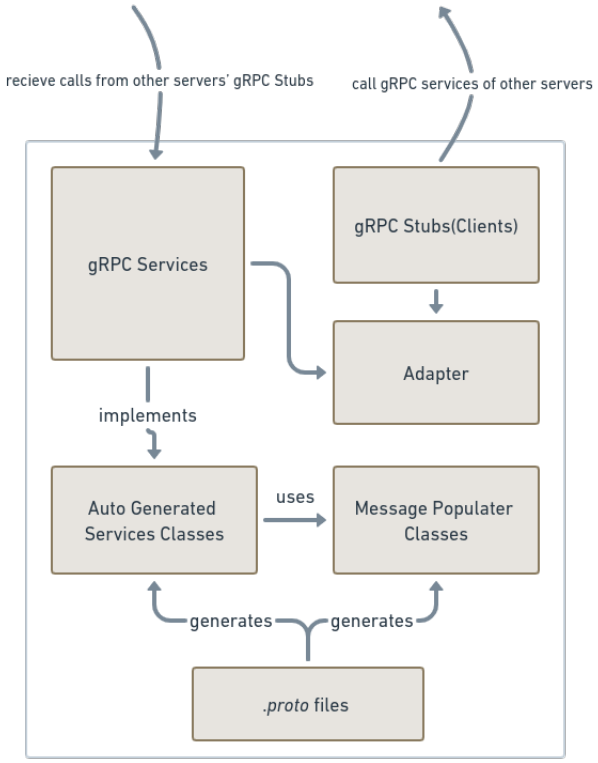


Figure 3.24: Network Module Architecture

1. **.proto Files**: All messages that need to be exchanged between servers of the system in addition to the definition of RPC services are defined using *proto*3 in *.proto files* and compiled using *protoc* to generate all Messages Builder Classes and Service interfaces.

2. **Message Populater Classes** Those Java classes are auto-generated by the compilation of *.proto files*. Message Builder Classes are Java classes that are used to build equivalent objects to construct messages defined in *.proto files*. Builder Classes also wrap the operation of converting and serializing messages to their equivalent proto buffers representation.

3. **Auto Generated Services Classes** These classes are final Java classes auto-generated by the compilation of *.proto files*. Auto-Generated Service Classes abstract methods defined by RPC services in *.proto files* to be overridden by actual implementation services classes. Each Service class also provides a stub (client class) to be used by actual client classes to send messages to this specific service.

4. **Adapter**: Adapter is a class that supports different methods that are used to convert objects from auto-generated gRPC messages representation to Core Objects of the system described later and vice versa. For Example:

```
Vertex vertexResponseToVertex(GrpcVertex vertexResponse)
```

This method is used to convert $GrpcVertex$ object which is created by an auto-generated Message Builder class to $Vertex$ core object that is used for all Core Logic of the system.

5. **gRPC Services** gRPC Services classes implement the Auto-Generated Services Classes and override their RPC methods. Each method accepts some defined request message sent by a gRPC client, converts it to an appropriate core object(s), and calls some core method. If the response of the gRPC method implemented is not empty. The adapter is called again to convert from the core object returned by the core method to the response-defined message to be sent to the client.

6. **gRPC Clients** gRPC Clients are classes that have some methods used by core system methods when there is a need to send a request to another server of the cluster. Client methods use the adapter to convert from core objects to defined requests, Then forward this request to the targeted server using its dedicated IP address and port. On a receive of the response from the remote server, the Adapter class is used to convert this response back to the core object to be returned.

Example

As previously stated, the Network Module is responsible for facilitating communication between servers within our distributed system. The retrieval of a particular vertex stands as a fundamental and pivotal function necessary for the engine's operation. Given the distributed nature of the system, it is possible for the requested vertex to reside on a remote server other than the one handling the user's query. Consequently, the querying server aims to dispatch a gRPC request in order to retrieve the desired vertex.

This process consists of multiple steps (Figure 3.25) that are explained as below:

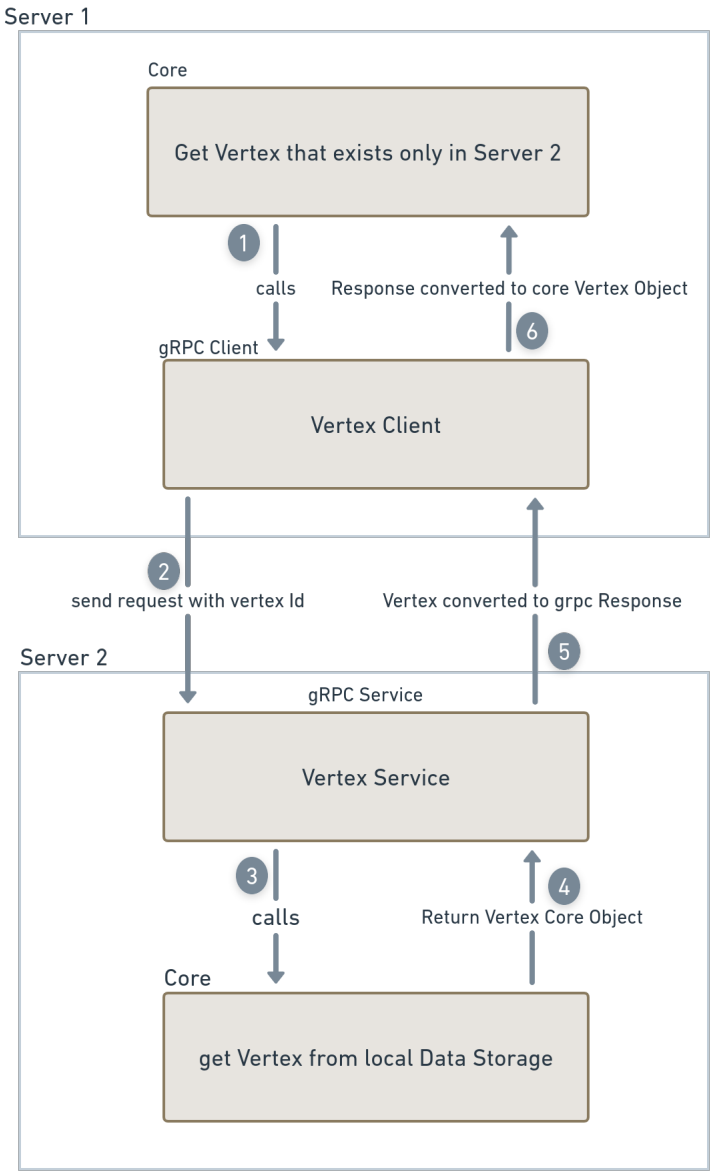


Figure 3.25: How to get Vertex from Remote Server

1. The core method that is responsible for obtaining the vertex giving its id from server 1 (the querying server) calls a gRPC Vertex client method to get the desired vertex with inputs of vertex id and server id that contains the vertex.

2. Vertex client sends a gRPC request message using the stub class generated to the IP address and gRPC port number of the remote server.

3. On the remote server, the sent request will be received by a specific gRPC service, which is Vertex Service in our case, and call specific RPC overridden method implementation.

This method calls the local core method that is responsible for getting the vertex from local Data Storage.

4. Core method returns the needed vertex into a core Vertex object representation to the gRPC method.

5. gRPC uses the Adapter class methods to convert from core Vertex object to gRPC auto-generated response object then send it back to the client of server 1.

6. On receiving a response at server 1 client, an Adapter method is called to convert back again from gRPC auto-generated response object to a core Vertex object to finally return it back to the core method.

3.3.3 Core Data Structures and Services Layer

This layer serves as the brain of the server that consists of multiple components (Figure 3.26). It comprises a set of Data Structures and classes that effectively represent the majority of logical entities within our system, namely Vertices, edges, indices, bindings, and operators.

Additionally, this layer encompasses a range of services dedicated to managing all CRUD operations associated with graph data. These services also handle filtering, partitioning, index manipulation, and database configuration tasks.

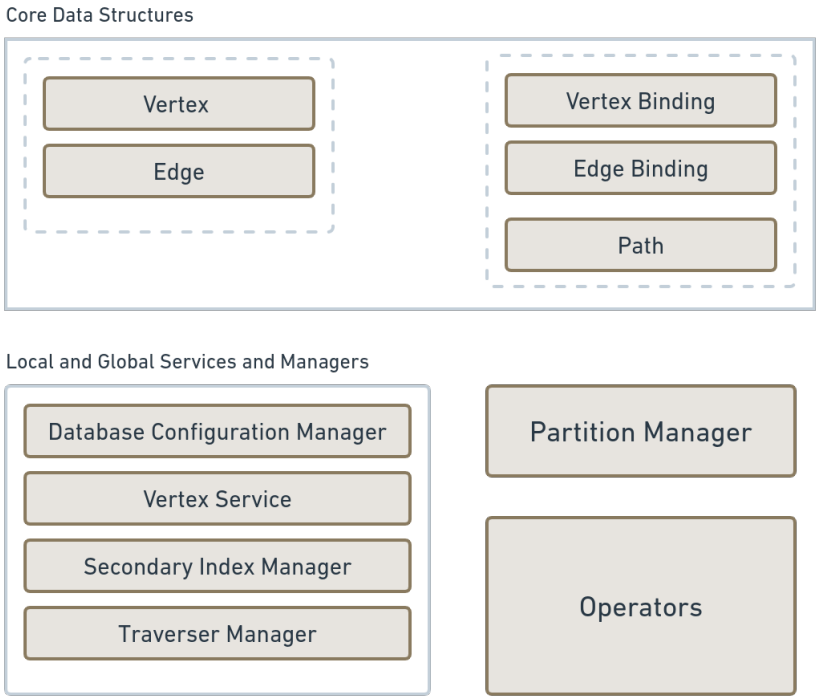


Figure 3.26: Core Layer Components

1) Partition Manager

Since *GraphoPlex* operates across numerous servers, it is essential to partition the graph data into multiple sub-graphs. The Partition Manager is a sub-module within the core layer and is responsible for determining the appropriate partition (server) within our system that should store a specific vertex along with all its incoming and outgoing edges.

Numerous algorithms are available to optimize graph partitioning and reduce the number of cut-off edges. However, using such algorithms requires loading the entire graph before partitioning. This approach works well for large static graphs that do not change or expand. However, since our system functions as a database engine, supporting operations like creating, deleting, and updating vertices and edges is crucial. Therefore, hashing was chosen as a method for partitioning graph data.

A hash function is applied to the vertex id for the partition manager to determine which server should hold some vertex. After that, a modulo operation of this hash value over the total number of servers should be executed to return the server id for that vertex (Figure 3.27). Server id is an environment variable configured for each server of the cluster and should be a number between 0 and $n - 1$ where n is the total number of servers configurable before running the cluster.

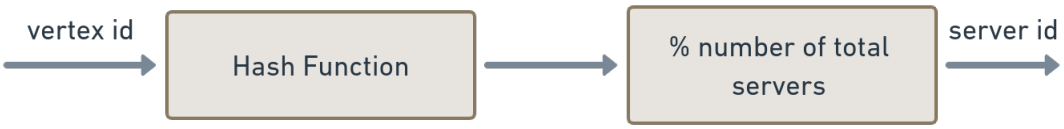


Figure 3.27: Getting the server id for a given vertex

2) Core Data Structure

This sub-module contains classes and interface that represents core data structures and object types that are needed by other components of the system. The most five important classes are chosen for further discussion about their implementation (Figure 3.28).

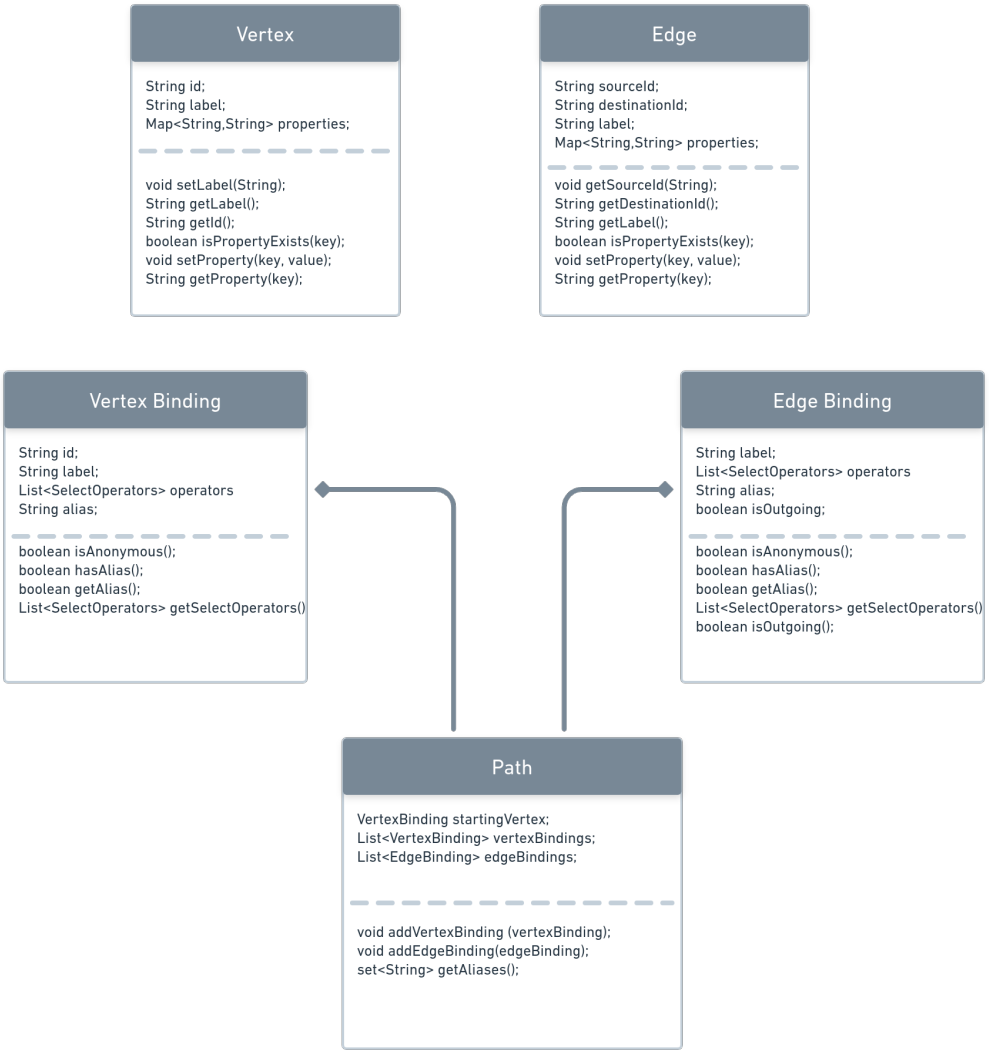


Figure 3.28: Core Data Structures

1. **Vertex Class**: Vertex class is used to represent vertices in the engine logic. It consists of an id, label, and map of properties that represent properties of the vertex in a key-value representation. The class also implements getters/setters for label and properties.

2. **Edge Class**: Edge class is used to represent edges in the engine logic. It consists of source id, destination id, label, and map of properties. source id is combined with the destination id and label to form the id of the edge which means there can not be 2 edges between the 2 same vertices with the same label stored.

3. **VertexBinding Class**: VertexBinding class is used to represent all the filters that are needed to retrieve some vertices. This class is used to build a path object to answer user path queries. A Vertex Binding object may consist of an id when the user is searching for a specific vertex by its id. Otherwise, It may contain some

label that the user is searching for all vertices with this label. Moreover, it may contain a list of select operators to further filter the vertices. Select Operators are discussed in the upcoming section.

4. **EdgeBinding Class**: EdgeBinding class is similar to VertexBinding class except it may not have an id as an instance variable.

5. **Path Class** : Path class is used to represent a path queried by the user in match sub-graphs queries. It consists of a starting vertex binding alongside two lists of vertex binding and edge bindings to represent the rest of the path. Both two lists should have the same length.

3) Operators

The Operators sub-module consists of a set of classes that represent operators used to answer user queries. Currently, this module only consists of select operators. Select operators are similar to relational algebra select operators. A select operator filters vertices on a specific value for a specific property. Operators supported are $=$, $<$, $>$, $\leq$, $\geq$.

There exists a class for each select operator based on operator type. All these classes extend the Select Operator abstract class (Figure 3.29) and implement all filtering methods based on the type of each operator.

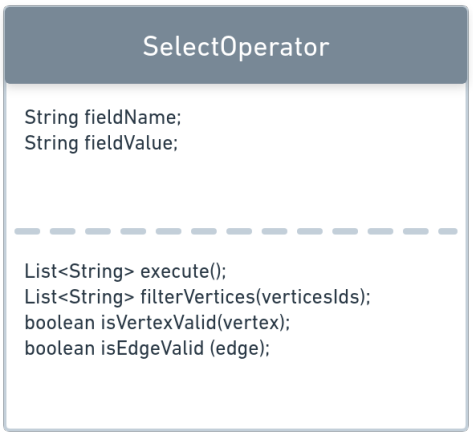


Figure 3.29: Select Operator Abstract Class

Here is an explanation of the functions of Select Operator methods:

1. `List<String> execute()`

   This method filters all local vertices and returns a list of vertices ids that are selected by the operator.

2. `List<String> filterVertices(List<String> verticesIds)`

   This method takes a list of some vertices ids and outputs a list of ids that should be selected by the operator.

3. `boolean isVertexValid(Vertex vertex)`

   This method returns whether a given vertex should be selected by the operator.

4. `boolean isEdgeValid(Edge edge)`

   This method returns whether a given edge should be selected by the operator.

4) Local and Global Services and Managers

As shown in (Figure 3.26), 4 different managers provide different functionalities to the system. When executing user commands through the query manager layer and algorithms layer, global managers are used. Global managers hide the complexity of dealing with multiple servers of the cluster. Each method is called within global services and managers are responsible to do its function either by calling its peer local function or calling the local method on other remote servers in the cluster through The Network Module. On the other hand, the local managers' methods only deal with data that is stored on the local server.

For Example, the $getVertices()$ method is provided by Global Vertex Service. It takes a list of ids and should return a list of vertices for input ids. The flow execution consists of 4 main steps (Figure 3.30):

1. Query Manager of the server that received the query (server 0) calls the $getVertices()$ method of Global Vertex Service.

2. First step that is done by the method is to group the provided ids by server ids by calling some method in the Partition Manager.

3. For each group of ids, Global Vertex Service either calls directly the Local Vertex Service method for the group of ids that exist on the same local server, Or forward The request to the gRPC Client to retrieve vertices that exists on the other servers of the cluster.

4. The Local Vertex Service of server 0 gets the vertices from Redis Data Manager (Data Storage Layer), while the Vertex gRPC client forward requests to all other servers' gRPC service. Each gRPC service calls directly the Local Vertex Service to retrieve needed vertices.

   To enhance performance, There is a separate thread running for each group of ids coming from Step 2. After each thread finishes execution, all results are combined into one list to be returned by the Global Vertex Service.

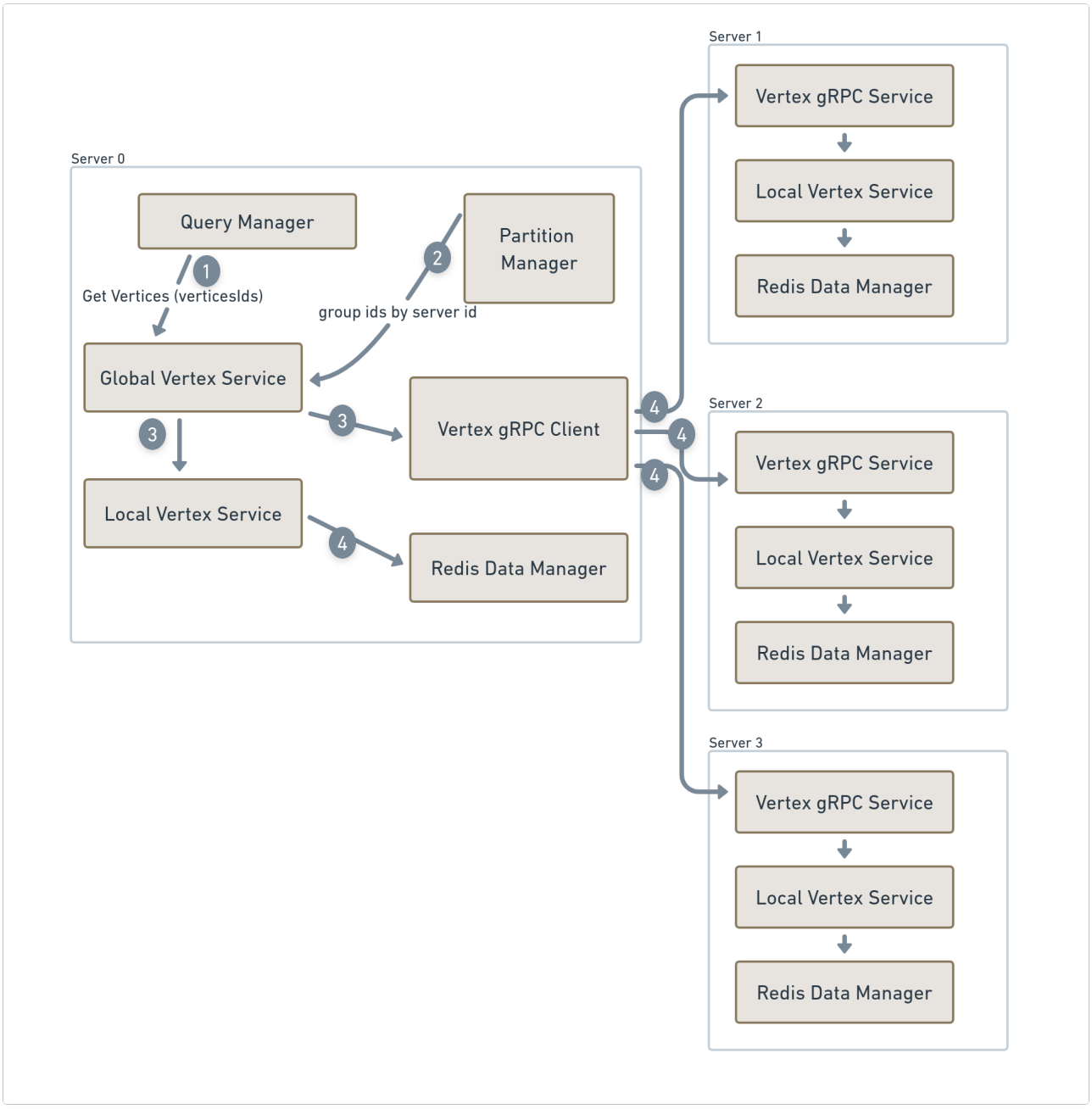


Figure 3.30: Flow of executing getVertices() method

3.3.4 Graph Algorithms Layer

This layer includes well-known graph algorithms that are used to execute user queries. Currently, There are four implemented graph algorithms: BFS, DFS, Dijkstra, and A*.

1) Breadth First Search

Breadth First Search (BFS) is used to answer path queries. It takes a start vertex and traverses the graph level by level. BFS is implemented in the system to execute in a multi-threaded way. Each level is partitioned into sub-sets that are equal to the number of servers in the cluster. Each sub-set is assigned to a separate thread. The algorithm can not continue execution for the next level until all threads finish execution.

There are two implementations of Breadth First Search in the Graph Algorithms Layer. The first one is the general one that takes a source node and traverses the graph level by level until all vertices are visited(Algorithm 1).

Algorithm 1 General Breadth First Search Algorithm

```
1: visited ← {}
2: currLevel.add(startVertexId)
3: while !currLevel.isEmpty() do
4:     nextLevel ← {}
5:     edges ← globalVertexService.getOutgoingEdges(currLevel)
6:     for all edge ∈ edges do
7:         if !visited.contains(edge.getDestinationId) then
8:             nextLevel.add(edge.getDestinationId)
9:             visited.add(edge.getDestinationId)
10:        end if
11:    end for
12:    currLevel ← nextLevel
13: end while
```

The Second one is a limited depth BFS that takes a *path* variable (item 5) and traverses the graph level by level trying to match the path described (Algorithm 2). Getting to the next level can be split into two main steps. The first step is to get all matching edges from the current level vertices that match the current edge binding (lines 10 -14). The next step is to filter all potential next vertices that match the current vertex binding (lines 15-25).

Algorithm 2 Fixed Depth Breadth First Search Algorithm

```
1: function COMPUTE(path)
2:     currLevel ← globalTraverserManagel.getVertices(path.startVertexBinding())
3:     for i ∈ {1..path.length()} do
4:         currEdgeBinding ← path.getEdgeBinding(i)
5:         currVertexBinding ← path.getVertexBinding(i)
6:         currLevel ← PROCESSANDGETNEXTLEVEL(currLevel,currEdgeBinding
   ,currVertexBinding)
7:     end for
8: end function
9: function PROCESSANDGETNEXTLEVEL(currLevel, edgeBinding, vertexBinding)
10:    if edgeBinding.isOutgoing() then
11:        edges ← globalTraverserManager.filterOutgoingEdges(currLevel,
   edgeBinding)
12:    else
13:        edges ← globalTraverserManager.filterIncomingEdges(currLevel,
   edgeBinding)
14:    end if
15:    potentialNextVertices ← {}
16:    if edgeBinding.isOutgoing() then
17:        for all edge ∈ edges do
18:            potentialNextVertices.put(edge.getDestinationId())
19:        end for
20:    else
21:        for all edge ∈ edges do
22:            potentialNextVertices.put(edge.getSourceId())
23:        end for
24:    end if
25:    nextLevel ← globalTraverserManager.filterVertices(potentialNextVertices,
   vertexBinding)
26:    return nextLevel
27: end function
```

2) Depth First Search

DFS traverses the graph to the deepest point first then track back to traversing the rest of the graph. Generally, DFS is implemented in a recursive way. However, on large graphs that could be loaded, recursive implementation will most likely throw a stack overflow exception. Therefore, DFS in the Graph Algorithms layer is implemented iteratively (Algorithm 3). Unlike BFS, DFS is executed synchronously in a single thread.

Algorithm 3 Iterative Depth First Search Algorithm

```
1: visited ← {}
2: stack ← {}
3: stack.push(startVertexId)
4: visited.put(startVertexId())
5: while !stack.isEmpty() do
6:     currVertexId ← stack.pop()
7:     edges ← globalVertexService.getOutgoingEdges(currVertexId)
8:     for all edge ∈ edges do
9:         if !visited.contains(edge.getDestinationVertexId()) then
10:            stack.push(edge.getDestinationVertexId())
11:            visited.put(edge.getDestinationVertexId())
12:        end if
13:    end for
14: end while
```

3) Dijkstra

Dijkstra is a Single Source Shortest Path (SSSP) algorithm. It returns the shortest path between two vertices in a weighted graph. The user should provide the field property name that represents weight in the graph when querying for the shortest path. (Algorithm 4).

Algorithm 4 Dijkstra Algorithm

```
1: function COMPUTE(sourceId, destinationId, costField)
2:     visited ← {}
3:     piriorityQueue ← {}        ▷ sort triplets (vertexId, cost, precedingEdge) on cost
4:     priorityQueue.push(sourceId, 0, null)
5:     visited.add(sourceId)
6:     while !priorityQueue.isEmpty() do
7:         currVertex ← priorityQueue.poll()
8:         if currVertex.vertexId = destinationId then
9:             return currVertex.distance
10:        end if
11:        if !visited.contains(currVertex.vertexId) then
12:            visited.put(currVertex.vertexId, currVertex)
13:            edges = globalVertexService.getOutgoingEdges(currVertex.vertexId)
14:            for all edge ∈ edges do
15:                priorityQueue.push(edge.getDestinationId(), currVertex.getDistance+
    edge.getProperty(costField), edge)
16:            end for
17:        end if
18:    end while
19: end function
```

4) A*

A* is another SSSP algorithm that aims to find the shortest path between two nodes. Its implementation is almost the same as Dijkstra but with a small change in the strategy of picking the next vertex from the priority queue. Unlike Dijkstra, A* does not pick the next vertex based on the lowest destination from the source vertex, it rather picks that vertex based on the destination from the source added to an estimated destination between that vertex to the destination vertex. This estimated distance is usually calculated based on a heuristic function (Algorithm 5). The most popular heuristics are Manhattan and Euclidean distance.

Algorithm 5 A* Algorithm

```
1: function COMPUTE(sourceId, destinationId, costField)
2:     visited ← {}
3:     piriorityQueue ← {} ▷ sort quadruples (vertexId, cost,heuristic ,precedingEdge)
   on (cost+heuristic)
4:     destinationVertex ← globalVertexService.getVertex(destinationId)
5:     priorityQueue.push(sourceId, 0, null)
6:     visited.add(sourceId)
7:     while !priorityQueue.isEmpty() do
8:         currVertex ← priorityQueue.poll()
9:         if currVertex.vertexId = destinationId then
10:            return currVertex.distance
11:        end if
12:        if !visited.contains(currVertex.vertexId) then
13:            visited.put(currVertex.vertexId, currVertex)
14:            edges = globalVertexService.getOutgoingEdges(currVertex.vertexId)
15:            for all edge ∈ edges do
16:                nextVertex ← globalVertexService.getVertex(edge.getDestinationId())
17:                distanceToDestination = heuristic(nextVertex, destinationVertex)
18:                priorityQueue.push(edge.getDestinationId(), currVertex.getDistance+
   edge.getProperty(costField), distanceToDestination, edge)
19:            end for
20:        end if
21:    end while
22: end function
```

3.3.5 Query Manager Layer

The Query Manager Layer is responsible for receiving user queries through the client, parsing it, executing it, and returning the results to the client. Many components are working together to provide these functionalities (Figure 3.31).

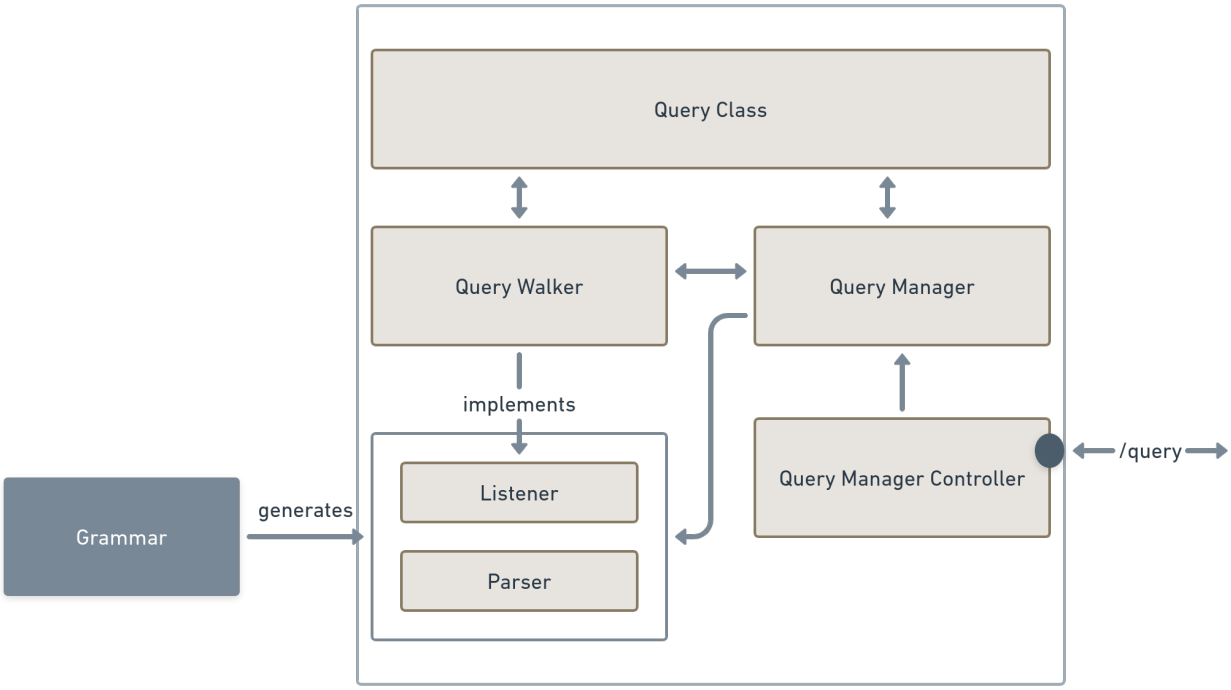


Figure 3.31: Query Manager Layer Architecture

1. **Grammar** The grammar consists of a set of rules that defines the supported query language. Grammar rules are written as ANTLR [2] ".g4" file.

   > ANTLR is a powerful tool used to generate parsers for structured text or binary files. It is widely employed for creating languages, tools, and frameworks. By utilizing grammar, ANTLR can generate a parser capable of constructing and traversing parse trees.

2. **Parser and Listener**: Parser and Listener classes are generated by compiling the grammar ".g4" file. The Parser class is responsible for building a parse tree for text input following grammar rules while the Listener is a base class that has some functions being called when visiting nodes of the parse tree.

3. **Query Walker**: Query Walker is a class that extends the Listener class. It implements methods that are executed when parse tree nodes are visited. Executing these methods is aiming to construct an instance of the Query class.

4. **Query Class**: Query class is a class that wraps logical representation of user query. It delegates its execution to a command object. There exist different command classes for each type of command supported by the system. Each Command class overrides the *execute*() method to perform its dedicated task. At the end of execution, an object of type Result should be built to be returned as the result of query execution. The basic Result class consists of instance variables representing a message and total execution time. Some commands like Path and Shortest Path

commands returned more complex Result objects that are children of the basic Result class. (Figure 3.32) shows the class diagram of the described structure

5. **Query Manager**: Query Manager class receives a string that represents a user query. It is responsible for initializing Parser and Walker instances, call walk() method of the Walker class. After the Walker finishes its execution and returns the query object. Query Manager calls execute() of the query object and returns its results or throws an exception.

6. **Query Manager Controller**: Query Manager Controller exposes an endpoint to receive user queries. Then Query Manger is called in a try/catch block. On successful execution, a response that contains the result with a status code of 200 is returned. On failure, a response that contains the exception message is returned with a status code of 400.

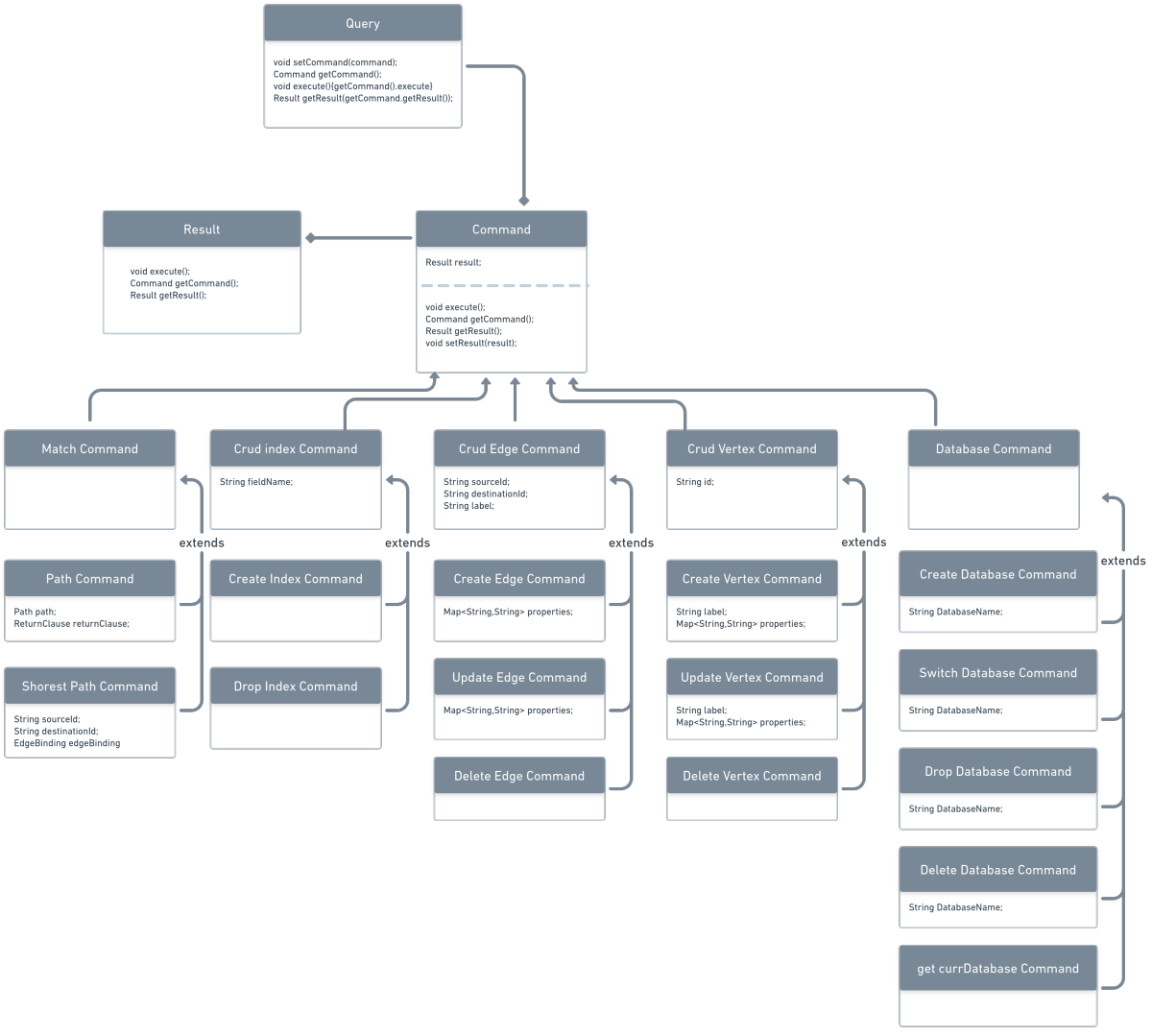


Figure 3.32: Query Class Diagram

3.4 Summary

In this section, we have discussed the design goals of *GraphoPlex* alongside its architecture as a client sending user queries and commands to a server in a cluster. All commands that are supported by *GraphoPlex* were listed alongside an example for each one. The last section was reserved to discuss server implementation that consists of 5 layers. Firstly, There is the Data Storage layer that serves as a store for users' graph data and was implemented using Redis. Next, The Network Layer facilitates inter-cluster communication and was implemented using gRPC. The system also includes Core Data Structure and Services Layer. Moreover, the Graph Algorithms Layer includes well-known graph algorithms which are used to execute user queries. Lastly, Query Manager Layer which serves as an interface between client and server logic.

Chapter 4

System Testing

This chapter is dedicated to showing and discussing performance test results for the implemented system.

4.1 Testing Enviroment

All tests are conducted on one machine with specs shown in (Table 4.1). A Docker image is built from the *jar* file of the server spring boot application. To simulate a multi-server cluster, this docker image is used to run a container for each server in the cluster.

Operating System	Pop!_OS 22.04 LTS
CPU	Intel Core i7-1065G7 @ 1.30GHz x 8
RAM	16.0 GiB

Table 4.1: System Specs

Each docker container should be configured with some environment variables that identify one docker container (server) from another (Table 4.2).

Variable	**Description**
SERVER_ID	a number between 0 and $n-1$ where n is the number of servers in the cluster
GRPC_SERVER_PORT	indicates the port where the server listens to gRPC calls on
SERVER_PORT	indicates the port where the server listens to HTTP calls on

Table 4.2: Server Environment Variables

The image that is used to build servers' containers should be configured with the number of servers alongside the IP address and gRPC port for each server in the cluster. That is necessary for servers to be able to communicate through gRPC.

4.2 Graph Algorithms Performance Testing

This section is dedicated to discussing the results of testing the performance of the implemented graph algorithms when increasing the number of instances (servers) in the cluster. Two data sets that differ in size have been used to test the performance of graph algorithms:

1. **Data Set 1**: data set 1 is quite small. it consists of **10,000** vertices and **100,000** edges.

2. **Data Set 2**: data set 2 is much bigger .It consists of **100,000** vertices and around **1,000,000** edges

(Table 4.3) and (Table 4.4) show the results that have been obtained when testing the main three graph algorithms that are implemented in our system: Breadth First Search (BFS), Depth First Search (DFS), and Dijkstra. Each test was run five times to take the average between all five observations.

	1 server	**2 servers**	**4 servers**	**8 servers**
BFS	3441.4 (ms)	2483.8 (ms)	1995.2 (ms)	1345 (ms)
DFS	4905.6 (ms)	8273.8 (ms)	10034.6 (ms)	11371.4 (ms)
Dijkstra	3375.8 (ms)	5194.2 (ms)	6623.2 (ms)	6870.4 (ms)

Table 4.3: Data Set 1 Average Execution Time

	1 server	**2 servers**	**4 servers**	**8 servers**
BFS	27664.8 (ms)	18706.8 (ms)	12752 (ms)	9126.6 (ms)
DFS	26807.8 (ms)	48042 (ms)	64148.4 (ms)	78371 (ms)
Dijkstra	29254 (ms)	48687.4 (ms)	66503.6 (ms)	78436 (ms)

Table 4.4: Data Set 2 Average Execution Time

As shown in (Figure 4.1), There is a decrease in the average execution time of the Breadth First Search algorithm when increasing the number of servers in the cluster for both datasets. This improvement in the performance of the algorithm when increasing the number of servers is due to the way Breadth First Search is implemented in the system. As mentioned before, Breadth First Search is executed in a multi-threaded way where the number of threads to execute each level is equal to the number of servers in the cluster.

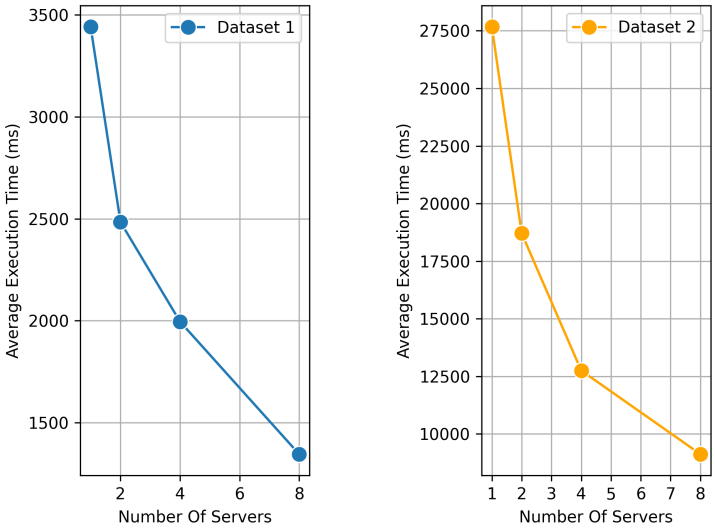
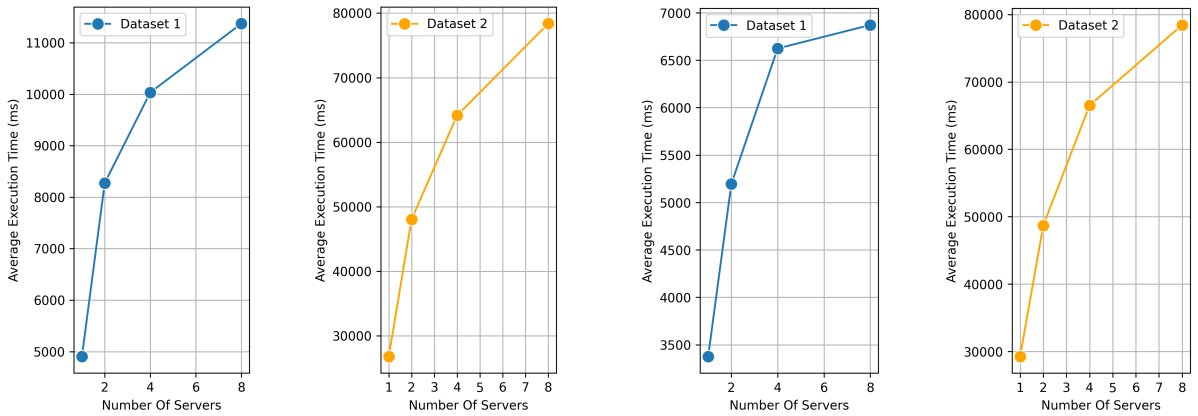


Figure 4.1: Breadth First Search Execution Time

On the other hand, an increase in the average execution time is observed on increasing the number of servers when conducting tests for Depth First Search and Dijkstra (Figure 4.2). This is due to the implementation of both algorithms in the system. Contrary to Breadth First Search, Depth First Search and Dijkstra are hard to be parallelized. The two algorithms are implemented in the system to be executed in a single thread. That explains why there is no improvement in the execution time when increasing the number of servers. Moreover, Network Latency when doubling the number of servers leads to the execution time of those two algorithms increasing.



(a) DFS (b) Dijkstra

Figure 4.2: Dijkstra and Depth First Search Execution Time

4.3 Path Query Performance Testing

This section is dedicated to discussing the results of evaluating *GraphoPlex* against other existing systems. Tests were conducted to evaluate **GraphoPlex** (8 servers) against **Neo4j** as a non-distributed graph database and **MySQL** as a non-graph database.

Data Set

The Data set that is used consists of **100,000** vertices that are all labeled by (PERSON) and have (name, age) properties and around **1,000,000** edges between vertices that are labeled by (FOLLOW). Since **MySQL** is not a graph database, There is no semantics to describe edges directly like graph databases. To insert the same data in MySQL, two tables are created. The first table is named "Person" with fields (id, name, age). The second table is called "Follow" with columns (user1,user2). The "Follow" table is used to store all edges as the "user1" column references the source vertex and the "user2" references the destination vertex.

Test Queries

All test queries are path queries but with increasing length. The query *(a follows b return b)* is considered to be a path query of length 1 while *(a follows b follows c follows d follows e follows f return f)* is a path query of length 5. Tests are conducted for path queries for lengths ranging between 1 and 5.

To write an equivalent path query in **MySQL**, *INNER JOIN* is utilized on the table "Follow" to represent the path query where the number of joins needed is equal to the length of the path.

Results

(Table 4.5) shows the result of comparing the performance (average execution time) of path queries of length ranging between 1 and 5 between **GraphoPlex** (8 servers), **Neo4j**, and **MySQL**.

	length 1	**length 2**	**length 3**	**length 4**	**length 5**
GraphoPlex	61.8 (ms)	74.6 (ms)	189.4 (ms)	707 (ms)	5087 (ms)
Neo4j	106.2 (ms)	120.6 (ms)	200.8 (ms)	480.4 (ms)	7199.6 (ms)
MySQL	1163.8 (ms)	2513 (ms)	3653 (ms)	5047 (ms)	10100 (ms)

Table 4.5: Path Query Average Execution Time

As shown in (Figure 4.3), The performance of **GraphoPlex** is quite better than **Neo4j**, especially in the path query of length 5. That could be due to the multi-threading

way of executing Breadth First Search in **GraphoPlex** which benefits from distributing data across multiple servers and running multiple threads to execute each level of path query.

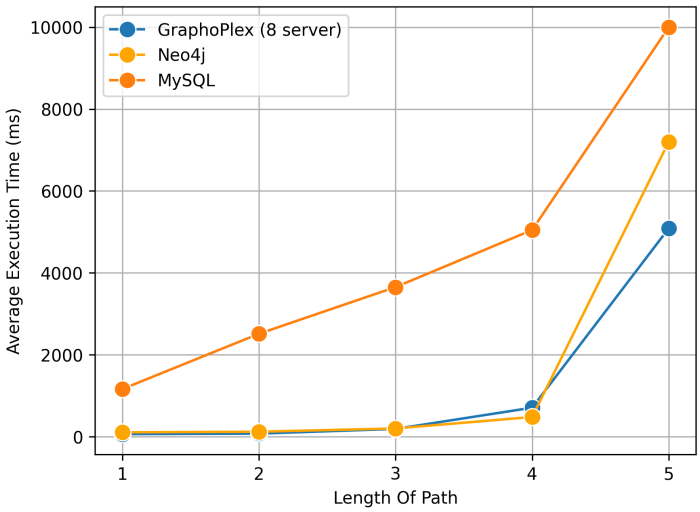


Figure 4.3: Path Query Execution Time

On the other hand, **MySQL** is quite slow compared with graph databases **Neo4j** and **GraphoPlex**, especially when comparing the increase of the execution time in the queries of length 1 to 4 when the result set is small (around only 10% of the vertices are returned) (Figure 4.3).

That is because graph databases execution time is only proportional to the amount of the graph being traversed to answer the query not the size of the whole graph. On the other hand, SQL databases use joins to answer such path queries which makes performance time proportional to the size of the whole table (graph) and the number of joins regardless of the amount of graph needed to be traversed to answer user queries.

4.4 Summary

In this chapter, we have described the testing environment that was used to evaluate the performance. of **GraphoPlex**.This was followed by the next section which was dedicated to discussing Graph Algorithms performance testing when increasing the number of servers in the cluster. It has been concluded the BFS performs better when increasing the number of servers because of multi-threading execution while DFS and Dijkstra suffer from slowing down because of network latency and them being hard to parallelize. The last section was reserved to discuss the results of comparing path queries of different lengths between **GraphoPlex** (8 servers),**Neo4j**, and **MySQL**. It has been found that GraphoPlex is quite better than Neo4j because of the multi-threading execution of path queries while both databases outperform MySQL due to the expensive joins that MySQL should execute to answer path queries.

Chapter 5

Conclusion

5.1 Achievements

The aim of the thesis was to implement a graph database engine that could be distributed among several(nodes) with a supported query language. A distributed graph database engine, **GraphoPlex**, has been developed. GraphoPlex stores user graph data across a cluster of multiple servers. Users could interact with the cluster through an interactive shell client application using a list of supported commands (subsection 3.2.2). After conducting performance tests (chapter 4), It has been found that the distribution of data and multi-threading execution makes GraphoPlex outperform Neo4j [12] in executing path queries.

5.2 Limitations

The main limitation of the thesis work was in the testing phase. Since available machines were limited, only one personal laptop was used to conduct performance tests. This made it hard to see the performance of the system in a truly distributed environment of multiple machines. This also could lead to inaccurate measurement of the effect of network latency on the system performance.

The limitation of available machines also made it hard to test the performance of the system against real-world big graph data that could be measured in Terabytes.

5.3 Future Work

The current version of *GraphoPlex* is enough as a proof of concept of what distributed graph databases could achieve but there are many improvements and features that could be added in the future:

1. **Parallel Implementation of DFS and Dijkstra :** The results of system performance testing (chapter 4) suggest improvements that should have been added to the implementation of both algorithms. Like BFS, parallel implementation for DFS and Dijkstra will improve the performance of both algorithms when increasing the number of servers in the cluster.

2. **Partitioning Strategy Improvement**: The current version of *GraphoPlex* employs hashing as a partitioning strategy to distribute data among servers. However, hashing is not the best way to distribute graph data. A machine-learning model could be developed to replace the current hashing partitioning strategy.

3. **New Commands and Algorithms**: There is plenty of graph algorithms such as (D* , Kruskal , and Floyd-Warshall) that could be added to the engine alongside many commands and operators such as (where clause, count, and, or, union) to answer more complex queries.

4. **Result Set Visualization**: The client application could be developed to visualize graph data in many forms rather than printing a table on the terminal. The existing client application could also be replaced with a web app client for better usability and interactivity.

5. **Traversal API :** support of interface for developers to use *GraphoPlex* traversal operators in their code through framework or library apart from the query language will allow them to extend the engine logic with their custom logic to handle more complex queries.

6. **Authentication and Support for Multiple Users :** Since any database engine stores crucial pieces of data, Authentication should be added for users to be able to connect to the engine. support for multiple user profiles with different grants and roles could be added.

5.4 Summary

To summarize, this thesis work was divided into 3 phases. The first phase was to review and compare existing graph database systems and their system design choices. Secondly, an overview of thesis project *GraphoPlex* was mentioned followed by a list of supported commands and implementation details of the system base modules. Finally, performance tests were conducted for the basic graph algorithms that exist in the system in addition to comparing system performance in executing path queries with a non distributed graph database Neo4j and a relational database MySQL.

Appendix

Appendix A

Lists

FaRM	Fast Remote Memory
DRAM	Dynamic Random Access Memory
RDMA	Remote Direct Memory Access
LMDB	Lightning Memory-Mapped Database
RPC	Remote Procedure Call
BFS	Breadth First Search
DFS	Depth First Search
SSSP	Single Source Shortest Path
ACID	Atomicity, Consistency , Isolation and Durability

List of Tables

List of Algorithms

List of Figures

Bibliography

[1] Renzo Angles. "A Comparison of Current Graph Database Models". In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. 2012 IEEE International Conference on Data Engineering Workshops (ICDEW). Arlington, VA, USA: IEEE, Apr. 2012, pp. 171–177. ISBN: 978-0-7695-4748-0 978-1-4673-1640-8. DOI: 10.1109/ICDEW.2012.31. URL: http://ieeexplore.ieee.org/document/6313676/ (visited on 02/21/2023).

[2] antlr. *What is antlr*. https://www.antlr.org/.

[3] Chiranjeeb Buragohain et al. "A1: A Distributed In-Memory Graph Database". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS '20: International Conference on Management of Data. Portland OR USA: ACM, June 11, 2020, pp. 329–344. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3386135. URL: https://dl.acm.org/doi/10.1145/3318464.3386135 (visited on 02/21/2023).

[4] Miyuru Dayarathna and Toyotaro Suzumura. "Towards Scalable Distributed Graph Database Engine for Hybrid Clouds". In: *2014 5th International Workshop on Data-Intensive Computing in the Clouds*. 2014 5th International Workshop on Data-Intensive Computing in the Clouds (DataCloud). New Orleans, LA, USA: IEEE, Nov. 2014, pp. 1–8. ISBN: 978-1-4799-7034-6. DOI: 10.1109/DataCloud.2014.9. URL: http://ieeexplore.ieee.org/document/7017947/ (visited on 02/21/2023).

[5] Aleksandar Dragojević et al. "FaRM: Fast Remote Memory". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 401–414. ISBN: 9781931971096.

[6] gRPC. *What is gRPC*. https://grpc.io/docs/what-is-grpc/introduction/.

[7] Li-Yung Ho, Jan-Jan Wu, and Pangfeng Liu. "Distributed Graph Database for Large-Scale Social Computing". In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012 IEEE 5th International Conference on Cloud Computing (CLOUD). Honolulu, HI, USA: IEEE, June 2012, pp. 455–462. ISBN: 978-1-4673-2892-0 978-0-7695-4755-8. DOI: 10.1109/CLOUD.2012.33. URL: http://ieeexplore.ieee.org/document/6253538/ (visited on 02/21/2023).

[8] Hongcheng Huang and Ziyu Dong. "Research on Architecture and Query Performance Based on Distributed Graph Database Neo4j". In: *2013 3rd International Conference on Consumer Electronics, Communications and Networks*. 2013 3rd International Conference on Consumer Electronics, Communications and Networks (CECNet). Xianning, China: IEEE, Nov. 2013, pp. 533–536. ISBN: 978-1-4799-2860-6 978-1-4799-2859-0. DOI: 10.1109/CECNet.2013.6703387. URL: http://ieeexplore.ieee.org/document/6703387/ (visited on 02/21/2023).

[9] G. Karypis and V. Kumar. "Multilevel Algorithms for Multi-Constraint Graph Partitioning". In: *Proceedings of the IEEE/ACM SC98 Conference*. SC98 - High Performance Networking and Computing Conference. Orlando, FL, USA: IEEE, 1998, pp. 28–28. ISBN: 978-0-8186-8707-5. DOI: 10.1109/SC.1998.10018. URL: http://ieeexplore.ieee.org/document/1437315/ (visited on 02/23/2023).

[10] Grzegorz Malewicz et al. "Pregel: A System for Large-Scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS '10: International Conference on Management of Data. Indianapolis Indiana USA: ACM, June 6, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807184. URL: https://dl.acm.org/doi/10.1145/1807167.1807184 (visited on 02/23/2023).

[11] Xianyu Meng, Xiaoyan Cai, and Yanping Cui. "Analysis and Introduction of Graph Database". In: *2021 3rd International Conference on Artificial Intelligence and Advanced Manufacture*. AIAM2021: 2021 3rd International Conference on Artificial Intelligence and Advanced Manufacture. Manchester United Kingdom: ACM, Oct. 23, 2021, pp. 392–396. ISBN: 978-1-4503-8504-6. DOI: 10.1145/3495018.3495086. URL: https://dl.acm.org/doi/10.1145/3495018.3495086 (visited on 02/21/2023).

[12] neo4j. *Neo4j Graph Database*. https://neo4j.com/.

[13] Redis. *About - Redis*. https://redis.io/docs/about/.

[14] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS'13: International Conference on Management of Data. New York New York USA: ACM, June 22, 2013, pp. 505–516. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2467799. URL: https://dl.acm.org/doi/10.1145/2463676.2467799 (visited on 02/25/2023).

[15] Gabriel Tanase et al. *System G Distributed Graph Database*. Feb. 8, 2018. arXiv: 1802.03057 [cs]. URL: http://arxiv.org/abs/1802.03057 (visited on 02/23/2023).