Media Engineering and Technology Faculty
German University in Cairo

Graph Algorithms For Distributed Graph Databases

Bachelor Thesis

Author: Ziad Maged Morsi Elsabbagh
Supervisor: Dr. Wael Abulsadat
Submission Date: 19 May, 2024

Media Engineering and Technology Faculty
German University in Cairo

Graph Algorithms For Distributed Graph Databases

Bachelor Thesis

Author: Ziad Maged Morsi Elsabbagh
Supervisor: Dr. Wael Abulsadat
Submission Date: 19 May, 2024

This is to certify that:

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

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

---

Ziad Maged Morsi Elsabbagh
19 May, 2024

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. They have supported me throughout this entire project. I also want to thank my friends for giving me suggestions and help throughout my thesis work. I would also like to thank this project's predecessor, Elshymaa Bateh, for her help as well. Finally, I would like to thank Dr. Wael Abulsadat for giving me the chance to work on this project and for his continued feedback and support.

Abstract

Nowadays, distributed graph databases are becoming more and more popular because of their ability to handle large-scale graphs, and algorithms that can handle such massive data are becoming important. This thesis aims to enhance **GraphoPlex**, an already existing distributed graph database, by adding new commands to the query language and developing a Graph, that also supports graphs from Graph Theory.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, distributed graph databases are becoming more and more popular because of their ability to handle large-scale graphs, such as social network graphs and community network graphs. However, as the volume of the data grows larger, the need for proper graph algorithms to handle the large graphs becomes paramount. In particular, graph sharding, or partitioning, algorithms that can help reduce the communication overhead between servers in a database cluster. However, the functionalities that come with a graph database, like computing the shortest path for example, should not be overlooked.

1.2 Aim

This thesis aims at enhancing and building on a previously implemented graph database that is distributed across multiple nodes (servers). More accurately, to add more commands to the preexisting query language so that the client can have more functionalities available at their disposal. Furthermore, this thesis also aims at developing a Graph API for said database to allow programmers to use the database. In addition, the addition of numerous types of graphs from Graph theory, like Hamiltonian graphs, is to be introduced.

1.3 Organization

The thesis work is organized as follows. Firstly, chapter 2 is dedicated to discussing the background of graph databases. It also includes a brief description and comparison between different implementations of existing distributed, as well as the outlining of two graph database benchmarks to indicate the fundamental features that should be present

in a graphs database, distributed or not. Secondly chapter 3 discusses the Design goals as well as the implementation of the newly added features to the previously developed distributed graph database. It also includes implementation details for the Graph API. Finally, chapter 4 concludes the work of the thesis by mentioning achievements, limitations, and possible future improvements.

Chapter 2

Literature Review

2.1 Background

In the modern digital world, graph-like data are becoming more and more common, with data represented as vertices and relations between them represented as edges. Common examples of this include social networks, communication networks, web data, and biological data. However, in such data, the properties and relations are exploited by graph algorithms more than the data itself. Furthermore, evaluating subgraph queries, i.e., finding instances of a given subgraph in a larger graph, is a fundamental computation performed by many applications that process graphs. (Abul-Basher et al. (2016) [1] and Kankanamge et al. (2017) [8])

According to Abul-Basher et al. (2016) [1] and Ho, Wu, and Liu (2012) [7], while it is possible that traditional relational databases can store graph-like data, the computational expense and the efficiency of the system start to degrade when using SQL expressions for graph data traversal. Moreover, to explore graph-like data relationships in a traditional SQL database, complex queries that are difficult to understand and, sometimes, involves the joining of multiple tables. Furthermore, according to Meng, Cai, and Cui (2021) [11], traditional relational database usually cost higher when dealing with reverse queries, which is type of query where instead of searching for documents or data based on specific criteria, the system retrieves documents or data that reference a given document or data point, and multi-level relational queries. In contrast, graph database can play a greater advantage in dealing with such problems. The main reasons are as follows:

- Treat vertices and edges equally; giving them the same status.
- Using two-way pointers and native graph storage, the relationship between vertices can usually be constant.

One such example of the query being needlessly complex in SQL is as depicted in figure 2.1 below. The queries in the figure try to explore the structure among employees

in a company by running a Depth First Search (DFS) on the vertices. While, the SQL expression on the left is practically unreadable, the expression on the right, which is written using Neo4j's Cypher Query Language, is easy to read and understand. (Ho, Wu, and Liu (2012) [7])

SQL	Neo4j

SELECT LEVEL seq2, seq1,
element id, REPORT TO,
PARENT ID,MFLAG,
element_name, element _type
FROM (select LEVEL seq1,
element id, REPORT TO,
PARENT ID, MFLAG,
element name, element type,
FROM M WAY TREE
WHERE hierarchy id = 1
START WITH PARENT ID is NULL
CONNECT BY PRIOR
element id = PARENT ID)
START WITH REPORT TO IS NULL
CONNECT BY PRIOR
element id = REPORT TO
	Set nodes =
	trav -o depth -r
	PARENT_OF:outgoing
	for(Node v: nodes){
	print(v.getProperty(id))
	print(v.getProperty(name))
	print(v.getProperty(type))
	}

Figure 2.1: Graph traversal query of SQL and Neo4j. [7]

However, despite their advantage over relational databases, graph databases suffer from multiple problems. Firstly, There is a noticeable discrepancy in graph database performance because of the impact of hardware elements. As of right now, there is no mechanism in graph database technology for creating or exploiting SSD. For instance, the Neo4j graph database uses a system that requires a large amount of memory since it stores data on a mechanical hard drive, loads it into memory during querying, and then performs computations on the data once it is fully loaded. Then there is the fact that graph databases caching technologies are derived from relational databases caching technologies. One example is Neo4j, whose caching technology is derived from MySQL caching technology, which does nothing to improve the performance of the queries. Lastly, storing a huge graph, like social network graphs or community network graphs, on a single machine is not feasible since it would start to cause storage problems. As a result, graph distribution strategies are needed to store the graph on multiple different machines. (Meng, Cai, Cui (2021) [11])

For the above reasons, the development of distributed graph databases is necessary to ensure that huge graphs can be stored with ease. However, since the graphs will be partitioned on multiple different machines, the need for proper partitioning methods is a must to ensure that during query execution there is as little communication overhead between the different machines as possible and ensure the scalability of the database. (Dayarathna and Suzumura (2014) [6])

2.2 Related Works

2.2.1 Benchmarks

The *Toronto Graph Database Benchmark (TGDB)* and *BlueBench* are two graph database benchmarks that will be discussed in this section. This is important because, benchmarks will help in identifying the key features that should be present any graph database, regardless of whether or not the database itself is a distributed database.

TGDB is a graph database benchmark tool that was introduced by Abul-Basher et al. (2016) [1] because it was noted that while several benchmarks exist for relational and XML databases, there is currently no widely accepted standard benchmark for graph databases. This graph database benchmark was designed with data analysts and scientists specifically in mind to help them choose the best database system to suit there needs. The bench mark was also tested against three distinct, and widely known, graph databases: Neo4j, OrientDB, and Titan-Cassandra; this was done to illustrate various approaches to graph data storage. The benchmark provides 3 basic features the should be present in any graph database, which are Graph Description, Operations over Graphs, and Queries over Graphs.

1. ***Graph Description:***
   (a) **Attributes Of Vertices/Edges:** Certain graph databases hold information about the vertices and edges. The attributes, which represent the characteristics of vertices or edges, might be strings or numbers. For example, the edges' numerical values could represent the edges' weight, cost, or value. Every graph element is given a distinct identity by graph databases as well.
   (b) **Directed Graphs:** The symmetry or asymmetry of a graph's relations depends on the type of graph data. In symmetric relations, one could cross the same edge to get from one nearby vertex to the other. On the other hand, the head and tail of an edge distinguishes it from another in an asymmetric manner. The vertex where the edge starts is indicated by its tail, while the vertex where it finishes is indicated by its head.
   (c) **Labels for Vertices/Edges:** Several applications call for various kinds of edges and vertices. Therefore, graph databases assign distinct labels to different vertices and edges in order to distinguish between them. In a social network, for example, there may be two types of relationships: "friend" relationships and "relative" relationships. As a result, the edges in this case are labeled as "friend" and "relative" respectively.
2. ***Operations over Graphs:***
   (a) **Graph Traversal:** When doing a traversal operation, we start at a particular vertex and work our way recursively through the neighboring vertices until we

meet our objectives. These objectives may include determining the depth of the traverse using Breadth First Search (BFS) or DFS or locating a target vertex. Finding the shortest path is a common example of a traversal operation, where the goal is to find the least number of edges that connect two vertices. Another example of retrieving all the vertices that are k edges distant from a given vertex is K-Hops.

(b) **Graph Analysis:** Graph analysis techniques are used to investigate the topology and structure of the graph and to determine the properties of its various objects. They are employed in the computation of various graph object statistics, in matching specific patterns, and in determining the impact of individual graph edges and vertices. The graph analysis examples include diameter, cluster coefficient, and hop-plot (i.e., the increasing rate of the neighborhood size given the distance from a source vertex).

(c) **Components:** A connected component is a set of graph vertices in which there is a path connecting any two of the component's vertices. A vertex can therefore only be a part of one specific component. For several procedures, including bridges (sets of edges whose removal causes a connected component to separate) or cohesion (sets of vertices whose removal would cause the group to split), finding related components is crucial.

(d) **Community Detection:** A group of vertices is called a community if its members are closer to one another than they are to those outside of it. Among the procedures involved in finding a community are minimal-cut edges, dendrograms (a hierarchical clustering of vertices), and other clustering techniques.

(e) **Centrality Measures:** Based on how a vertex links to other vertices in the graph, a centrality metric can be used to determine how important a vertex is overall. The most widely used measures of centrality are closeness (which calculates the average length of a shortest path connecting a vertex to all other vertices), degree (which counts the number of edges a vertex has), and betweenness centrality (which counts the number of shortest paths that pass through a vertex).

3. ***Queries over Graphs:***

(a) **Transformation/Analysis:** Transformation queries change the graph's structure. Bulk loading of a network, adding or deleting new vertices and edges, generating new types of vertices, edges, and attributes, and changing an attribute's value are some common instances of transformation queries. However, in analytical queries, the graph's structure stays the same. If there is not enough memory available for the computation, analytical queries may, nevertheless, generate temporary vertices, edges, and characteristics to make the analytical process simpler. They may even store intermediate findings as a graph in a secondary storage.

(b) **Scale:** While some graph queries reach every vertex and every edge in the graph, others do not. Depending on the accessibility level, we can divide

questions into global and local categories. All of the graph's edges and vertices are accessible via global queries. Local searches only retrieve data from a subset of the graph.

(c) **Attributes:** A graph has other properties connected to its vertices and edges in addition to its structural elements, or edges and vertices. Therefore, we can categorize query access into attributes of vertices only, attributes of edges alone, attributes of both vertices and edges, or none at all, based on the kind of attributes that the queries require to retrieve.

(d) **Results:** A graph query may return the result as a graph, an aggregated result, or a set of elements. A graph, which is the outcome of the original graph's selection, projection, or transformation, is the most typical output of graph searches. Finding the graph's smallest spanning tree or the shortest path between two vertices are two examples of this kind of conclusion. The graph's statistics are included in the second type of result; for example, obtaining the graph's histogram of the degree of vertex distribution or community size yields an aggregated result. A collection of the graph's elements can be found in the third kind of result; an example of this would be a query that yields the graph's edges with the highest weight.

BlueBench is another graph database benchamrking tool that was developed by Kolomienko, Svoboda, and Mlnkov (2013) [9] as part of an experiment to compare several graph databases against each other. *BlueBench* employs a set of 13 graph operations that range from very complicated ones to very trivial ones. Those operations are as follows:

1. ***DeleteGraph:*** This operation sends the database back to its initial state by completely dropping all of the data along with the indices from disk.

2. ***CreateIndexes:*** This operation creates an index on the vertices of the graph using a selected property key before any elements are loaded into the database.

3. ***LoadGraphML:*** This operation inserts elemtnes into the database according to the GraphML input file, which is an XML-based file format used for describing graph structures.

4. ***Dijkstra:*** This operation computes the shortest path from a source vertex to all of the other reachable vertices in the graph using *Dijkstra*'s algorithm.

5. ***Traversal:*** This operation conducts a simple BFS search operation with a depth of 5 from a given source vertex.

6. ***TraversalNative:*** Contrary to the *Traversal* operation, the native API of the graph database engine is used instead of the standard *Blueprints* API, which *BlueBench* is implemented with, whenever possible.

7. ***ShortestPath:*** This operation, contrary to the *Dijkstra* operation, computes the shortest path between a source vertex and a target vertex.

8. ***ShortestPathLabeled:*** The same as the *ShortestPath* operation with the difference being that only edges with certain labels are considered when computing the shortest path.

9. ***FindNeighbors:*** This operation finds the closest neighbors of a randomly selected vertex. This operation is considered the most primitive traversal operation.

10. ***FindEdgesByProperty:*** The first non-traversing operation searches the graph database for all edges that have a certain property equal to a given string value. The value is selected from a dictionary containing every possible value that an edge could have; it is not determined at random.

11. ***FindVerticesByProperty:*** This operation does the exact same thing as the pervious operation with the exception that it is done for vertices instead of edges.

12. ***UpdateProperty:*** This operation tests how efficiently the database engine updates properties of elements. A predefined set of vertices is firstly selected, then this set is divided into pairs, and finally, every two vertices in each pair swap their properties.

13. ***RemoveVertices:*** This operation, as the name suggests, tests the performance of removing vertices from the graph.

2.2.2 Distributed Graph Databases

A1 is a distributed in-memory graph database that was introduced by Buragohain et al. (2020) [5]. It is used by the Bing search engine to support complex queries over structured data. The reason why A1 is able to function is because of the low cost and availability of Dynamic Random Access Memory (DRAM) and the high speed of Remote Direct Memory Access (RDMA). Moreover, A1 uses Fast Remote Memory (FaRM) as the underlying storage layer, while having a graph abstraction and query engine layers on top of it.

A1 also follows a standard layered architecture (Figure 2.2), 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.

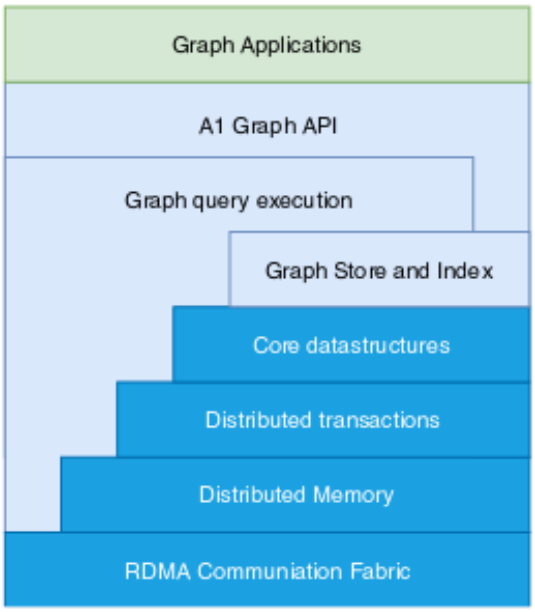

Figure 2.2: A1 Architecture Layers [5]

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, 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.

Currently, A1 uses a random partitioning strategy, where the vertices are placed randomly across the cluster, and locality is used to push query execution to where data reside. Moreover, it is observed that A1 primarily supports the K-Hop queries. However, due to the complexity of the *A1QL* Query Language (Example in Figure 2.3), A1 does not support other graph operations, like shortest path or community detection operations.

```
{ "id" : "steven.spielberg",
  "_out_edge" : { "_type" : "film.director",
      "_vertex" : {
          "_out_edge" : { "_type" : "film.actor",
              "_vertex" : {
                  "_select" : ["*"]
              }}}}}
```

Figure 2.3: A1QL Query to get all actors that have worked with Steven Spielberg. [5]

PrallelGDB is another distributed graph database that was developed by Barguno et al. (2011) [3]. It is based on specializing the local caches of any node in the system, providing a better cache hit ratio. The graph model used by *ParallelGDB* is an attributed directed graph, i.e. supports edge types, vertex types and any kind of attributes in edges and vertices.

ParallelGDB also specializes the computation in each node, by loading different subsets of the graph in each computing node. As a consequence of this specialization most of the graph can be loaded into memory, providing good scalability to the system due to a better cache hit ratio. Although each node stores in its local disk, a portion of the graph, each node in the system has access to the entire graph database through a distributed graph storage. [3]

The architecture of *ParallelGDB* is as follows (example in Figure 2.4). There are $\boldsymbol{n}$ nodes or computers in a cluster. Each node maintains access to the entire graph database through the Distributed Graph Storage (DGS), a system that spreads the physical data at the database page level among the nodes in the cluster. The DGS also allows any node to access the entire graph information (local and remote data). As a consequence, nodes' caches can be used to specialize them towards a specific data subset of the graph. All nodes cache the data they use to resolve the queries they receive. Specializing nodes causes the system faster, as most of the information used to answer a query is in the caches of the nodes. [3]

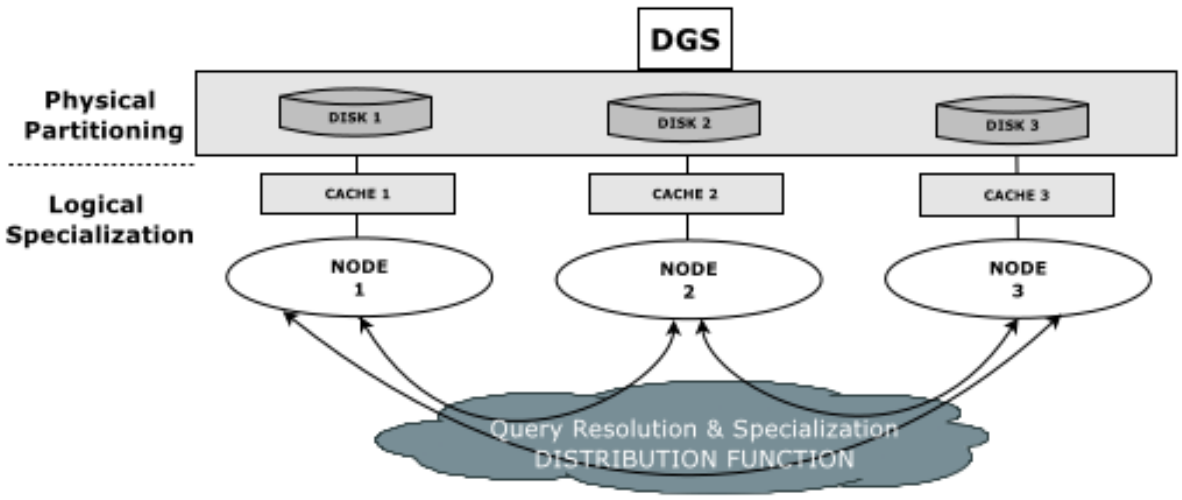

Figure 2.4: ParallelGDB architecture with 3 nodes. [3]

Like **A1**, *ParallelGDB* also employs a random partitioning strategy, where the vertices are assigned to a random node in the cluster. The reasoning for this is because graph partitioning techniques may produce unbalanced load among the nodes in the cluster if an area of the graph is continuously explored while other areas in the graph are never surveyed, which renders the graph database inflexible by real scenario standards where

query workloads change over time. Moreover, it is observed that *ParallelGDB* supports K-Hops and BFS queries only and lacks a supported query language, unlike *A1*.

Trinity is a general purpose graph engine over a distributed memory cloud that was proposed by Shao, Wang and Li (2013) [12]. Trinity manages to support fast graph exploration and efficient parallel computing by employing optimized memory management and network communication. The engine supports both online graph query processing and offline graph analytics, and it has been used for real-life applications, including knowledge-bases, knowledge graphs and social networks; it does so by leveraging graph access patterns in both online and offline computation to optimize memory and communication for best performance.

A Trinity system (Example in Figure 2.5) consists of several network-connected components. Slaves, proxies, and clients are the three categories into which these parts are separated according to their functions. Each slave is in charge of processing messages from clients, proxies, and other slaves in addition to storing a portion of the data. A Trinity proxy, on the other hand, manages messages without storing any information. It serves as a mediator between customers and slaves. Users can communicate with the Trinity cluster by using a Trinity client. These clients are programs that interface with Trinity slaves and proxies via the APIs provided by the Trinity Library.

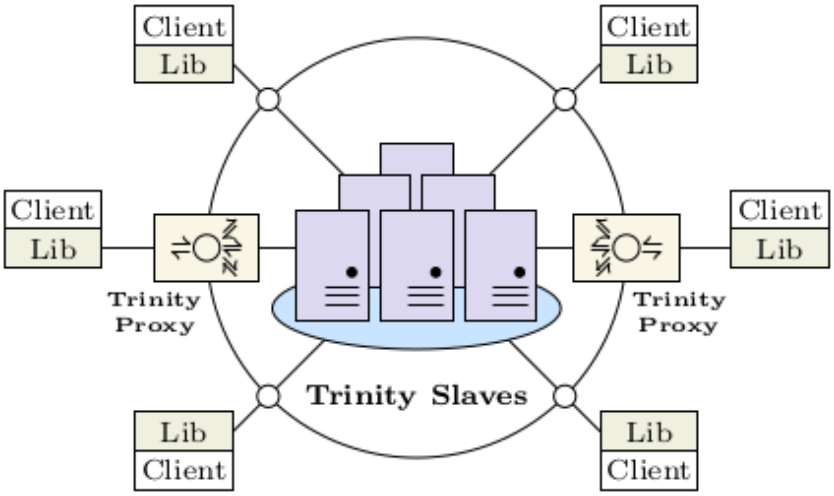

Figure 2.5: Trinity Cluster Structure [12]

In a Trinity system, as shown in Figure 2.6, the memory cloud is essentially a distributed key-value store, and it is supported by a memory storage module and a message passing framework. The memory storage module manages memory and provides mechanisms for concurrency control. The network communication module provides an efficient, one-sided, machine-to-machine message passing infrastructure. Furthermore, Trinity Provides a specification language called Trinity Specification Language (TSL) that bridges the graph model and the data storage. Due to the diversity of graphs and the diversity of graph applications, it is hard, if not entirely impossible, to support efficient general

purpose graph computation using fixed graph schema. Instead, Trinity let users define graph schema, communication protocols, and computation paradigms through TSL.

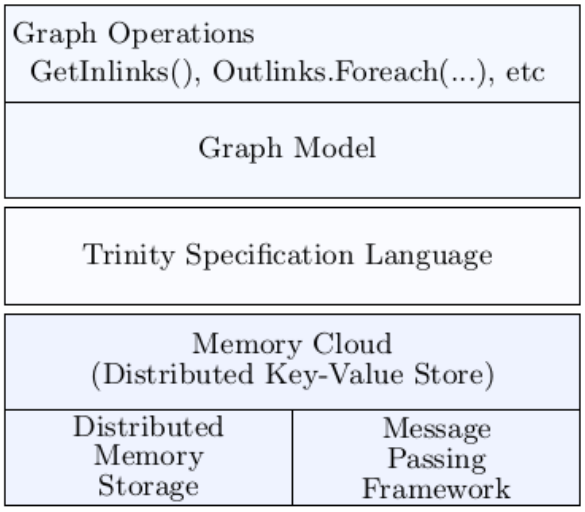

Figure 2.6: Trinity System Layer [12]

It is not clearly known what kind of partitioning strategy Trinity employs to partition large-scale graphs. However, Trinity is not constrained by any computation model. For example, Trinity can partition billion-node graphs within a few hours using a multi-level partitioning algorithm with the quality of the partitioning is comparable to that of the best partitioning algorithm (e.g., METIS). As for the supported queries, Trinity supports K-Hops queries, Shortest Path queries, BFS and DFS queries, and Page Rank queries. However, it is unclear what algorithm is used for the shortest path queries.

Acacia is a distributed graph database engine for scalable handling of large graph data that was introduced by Dayarathna and Suzumura (2014) [6].

Acacia has a distributed architecture that comprises a set of distributed processes and their associated local data stores. The system operates on a partitioned distributed graph using the Master-Worker pattern which provides more control over the system's execution compared to other approaches such as Peer-to-Peer (P2P) in medium-scale deployments. The Master node hosts a data loader that converts a plain edge list file to the METIS-compatible format to allow for proper partitioning.

The Workers are standalone, non-distributed graph stores that operate independently and concurrently. Each worker handles a part of the graph, and computational load balancing is achieved by running a graph processing algorithm, or MapReduce job, concurrently on multiple workers. Acacia assumes that each compute node imposes a limitation on its local data storage. When a new graph is added, Acacia first partitions the graph into a number of subgraphs equal to the number of workers that it runs. Initially, the partitioned graphs are distributed among Workers based on their available capacity.

The Central Store stores the edges that lie on two partitions as well as unconnected vertices. The metadata store of all operational information is kept on a relational

database, and HSQLDB is used for this purpose. Acacia keeps on monitoring the Local store capacities as well as the CPU instances that are run on each node on private/public cluster and moves data between public and private clusters dynamically to maintain the Service Level Agreement (SLA) values. The Remote Instances are represented by Java Virtual Machine (JVM) processes on the public cloud, and Acacia deploys remote JVM instances when required. The overall architecture of the Acacia system can be seen in Figure 2.7.

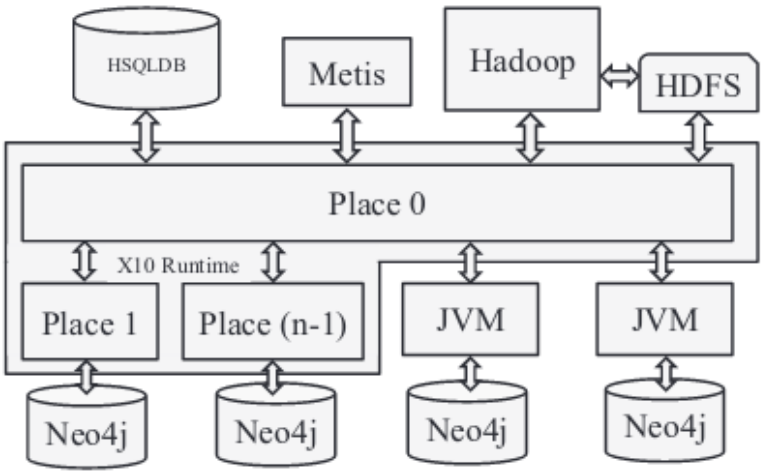

Figure 2.7: Acacia System Architecture [6]

To partition the graph, Acacia uses the METIS Algorithm. As mentioned above, this is done by the Master node, which hosts a data loader that converts a plain edge list file to the METIS-compatible format, partitions the graph using METIS, and distributes the partitions among the Workers and the Central Store. This produces high-quality graph partitions with less edge cut values between subgraphs which allows reducing the requirement of excessive communication between compute nodes. As for the supported graph operations in Acacia, it is only stated that the graph database engine supports the Top K PageRank operations, which calculate the average out-degree of the vertices of the graphs; which means that Acacia can support PageRank operations on large datasets.

System G is another distributed graph database developed by Tanase et al. (2018) [13] for efficient graph data storage and processing on modern computing architectures. In particular, it comprises a single node graph database and a runtime and communication layer that allows the composition a distributed graph database from multiple single node instances. System G's architecture consists of three primary components: the front-end, the back-end, and the data store.

A query manager and query clients make up System G's front end. Over the distributed database, query clients are in charge of creating graph queries, and query managers are in charge of arranging communication between query clients and the back-end nodes.

System G's back-end consists of several nodes, each of which is in charge of answering queries from clients. A copy of the graph data, or vertices and edges, is stored on each back-end node, which is connected to the query manager by Remote Procedure Calls (RPCs).

The data store of System G is used to store the graph data and associated metadata. System G employs a key-value store to store graph data and a file system to store metadata. In the current implementation, System G uses an LMDB key-value store, which allows for efficient querying of graph data via graph indices.

As for the partitioning strategy used by System G, a hash-based partitioning strategy is used to partition the graph data. The vertices are partitioned using a hash function applied to the external vertex identifier and edges are located with their source vertex by default. This involves the computation of the hash function to determine the shard where a particular vertex or edge is or will be located. This hash-based partitioning strategy enables efficient distribution of graph data across multiple nodes in a distributed graph database system. As for the supported graph operations in System G, the BFS and DFS traversal operations are supported, as well as PageRank, community detection operations, and operations to identify the connected components of a graph.

ByteGraph was described by Li et al. (2022) [10] as high-performance distributed graph database developed by ByteDance. With ByteGraph, ByteDance are able to efficiently store, query, and update massive amounts of graph data generated by their products. ByteGraph is designed to handle three types of graph workloads: online analytic processing (OLAP), online serving processing (OLSP), and online transaction processing (OLTP), each with its own set of characteristics and challenges. The database is optimized for high throughput, low latency, and scalability. ByteGraph consists of three layers: an execution layer (BGE), a cache layer in memory (BGS), and a durable storage layer based on a persistent KV store (See Figure 2.8).

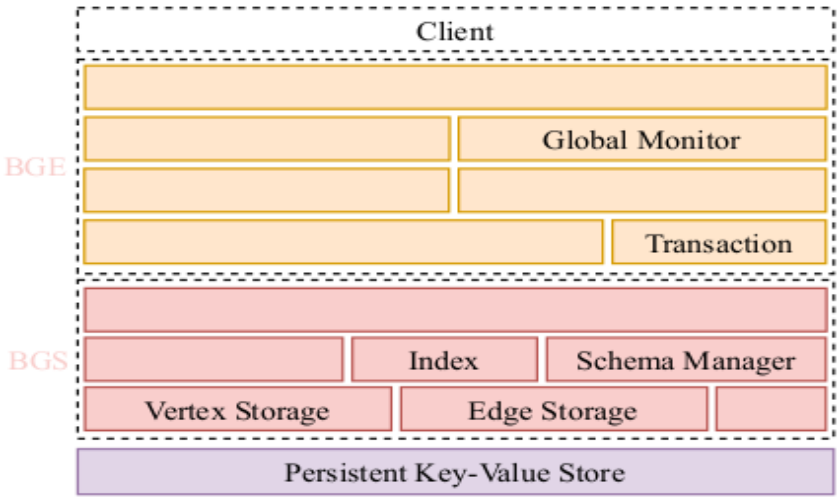

Figure 2.8: ByteGraph System Architecture [10]

The BGE handles computation-intensive operations such as sorting, aggregation, and other query processing tasks. The BGS focuses on graph-native cache data management and log management. Both layers can be independently scaled out according to the workloads and available resources.

The durable storage layer is responsible for persisting all KV pairs generated by BGS, such as graph data, logs, and metadata. Existing KV stores, such as RocksDB and TerarkDB, can be integrated as a black-box in ByteGraph.

To provide efficient data management, BGS receives requests from BGE and stores the accessed data in memory to facilitate quick access. In addition to the graph data, BGS also maintains system data that are used to accelerate processing or to ensure consistency and atomicity, such as indexes and logs.

Regarding the partitioning strategy used by ByteGraph, like **System G**, ByteGraph uses a hash-based partitioning strategy to ensures an even distribution of data across different partitions while keeping locality for each vertex's adjacency list. Specifically, vertices are partitioned based on their IDs, while the edges pointing to or originating from vertices in different partitions are hashed to the corresponding partitions. As for ByteGraph's supported graph operations, it supports both PageRank and Single Source Shortest Path (SSSP) operations, as well as vertex/edge filtering, range queries, and sorting.

Ho, Wu, and Liu (2012) [7] present an efficient distributed graph database architecture for large scale social computing. The structure of the system consists of two main subsystems: a distributed graph data processing system and a distributed graph data store.

After processing the data and writing the computation results back to the distributed data store, the distributed data processing system reads data from the distributed data store. Users develop and run their applications on the distributed graph processing system. The data processing system employs GoldenOrb, an open-source implementation of the Pregel model from Google, as its main processing engine. However, the default backend store of GoldenOrb is the Hadoop File System (HDFS), which is inefficient for graph processing. Therefore, the authors implemented an InputFormat interface and connected GoldenOrb to a Distributed Graph Data Storage System they developed.

The distributed graph data storage system consists of 3 layers (See Figure 2.9): Front-End Layer, Middle-ware Layer, and Back-end Storage Layer.

The front-end layer receives user queries, returns the results to users, and stores the metadata of the partitioned graph.

The middle-ware layer includes a distributed edge server and a graph server that provides an interface for applications to manipulate data in the graph system. The edge server provides the ability to retrieve certain edges based on the properties defined in the graph schema while the graph server provides functionality to manipulate the vertices and edges.

The back-end storage layer consists of several standalone, share-nothing Neo4j graph databases. Each back-end store has its queue so that the operations to the same back-end store are serialized.

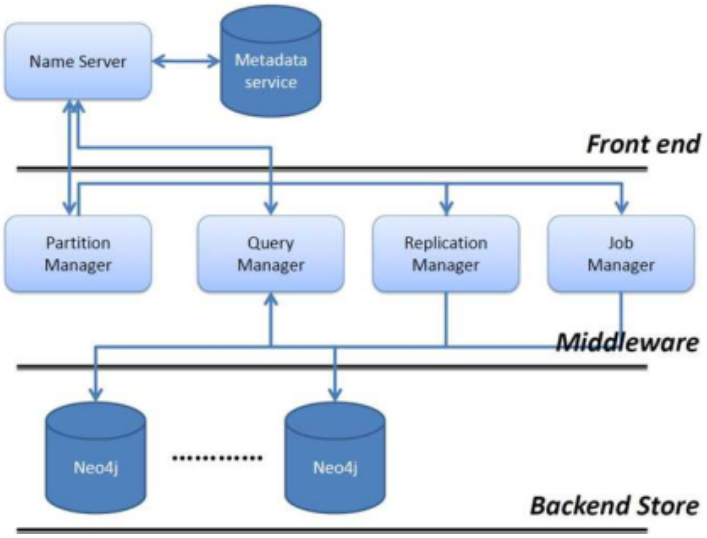

Figure 2.9: Distributed Graph Database Architecture [7]

Rather than using hashing as the partitioning strategy, just like **Acacia**, the METIS partitioning algorithm is employed as it ensures a balanced workload among partitions and minimizes the number of cut edges. As for the supported operations, they include PageRank, Maximum Value Propagation, Bipartite Matching, Influence Spreading, and Random Walk.

2.3 Summary

In this chapter a literature review of distributed graph databases was conducted. First, in Section 2.1, the background of graph-like data and graph databases was discussed, as well as the advantages graph databases have over traditional relational databases and the problems they face due to the large-scale nature of real world graph-like data.

Following the discussion is Section 2.2, which discusses related works, is divided into two sub-sections.

The first sub-section is Section 2.2.1, which discusses benchmarks for graph databases, irrespective of whether or not they are distributed graph databases. **TGDB** and **BlueBench** where two benchmarks that were introduced, as well as the operations that are to be featured in any graph database.

The second and final sub-section is Section 2.2.2, which discusses various distributed graph databases. For each of the seven graph databases mentioned in the section, the reasons they where developed or introduced where briefly discussed, followed by a detailed explanation of the database's architecture, as well as the partitioning strategy used by said database to distribute the graph properly across the servers in the cluster.

Chapter 3

System Design and Implementation

This chapter discusses the Design goals as well as the implementation of the newly added features to GraphoPlex, a previously developed distributed graph database.

3.1 Design Goals

The primary objective is to enhance the previously implemented distributed graph database, **GraphoPlex**, by adding new features and enhancing, or mending, already existing ones.

The goal is to create a Graph API that can allow programmers to load a graph from a text file and then store it in the distributed graph database, that is GraphoPlex. Additionally, as something that is not done by any preceding Graph Database, support for the various types of graphs from Graph theory, such as Hamiltonian Graphs, is to be added.

In addition, numerous new algorithms, such as the Bellman-Ford algorithm are to be added, along side new commands for the preexisting query language as well as fixing the problems that may arise due to GraphoPlex's reliance on hash-based partitioning, by adding new partitioning strategies.

3.2 Design Strategy

The design strategy for the new features is as follows:

1. For the Graph API, basic Object-Oriented Programming (OOP) features will be utilized along side the core data structures that GraphoPlex uses, like pre-implemented vertices and edges.

2. For the implemented types of graphs, each sub-type will have its own TS that is implemented as a class of its own. For the TSs, the strategy design pattern will be utilized.

3. For some of the newly implemented commands that are added to the query language and the algorithms, the Graph API will be utilized, since the majority of the commands and algorithms will require processing the whole graph, like getting the radius of the graph.

4. For the SS, or partitioning strategies, each strategy will also have a class of its own to properly partition the graph. The strategy design pattern will also be utilized for the SSs as well.

3.3 System Overview

There are three major components that are used to add the new features to GraphoPlex.

The first components of the system are the vertex and edge classes that are part of the core data structure of the system. They are used as the building blocks for the Graph API. The reason for that is because the classes themselves contain the data that is to be stored in the database. So, instead of having to construct said classes all over again, then having to create a converter that would convert from the Graph API's implementation of the classes to the core data structure implementation, It would be more convenient to use the core data structure implementation itself.

The second component is the Graph API itself, which is to be utilized when attempting to run the majority of the newly implemented commands and algorithms. The reason for that is because some of the new algorithms, like the Tarjan algorithm, require processing the whole graph. Thus, it would be easier to load the graph into memory and process it as part of the API's implementation (i.e. as instance methods inside of the classes). This is also going to prove useful in graph sharding and resharding, where the entire graph has to be loaded into memory, then resharded.

The last component is the rule-based engine, known as Jeasy. This engine is added to allow for graph partitioning based on rules that are predefined. Furthermore, it would allow the API users to shard or reshard the graph according to predefined or newly-defined rules, created by the engine.

3.4 System Design

3.4.1 Overview of GraphoPlex's Architecture

GraphoPlex consists of a client and a cluster of servers that runs on a P2P architecture (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. [4]

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. [4]

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. [4]

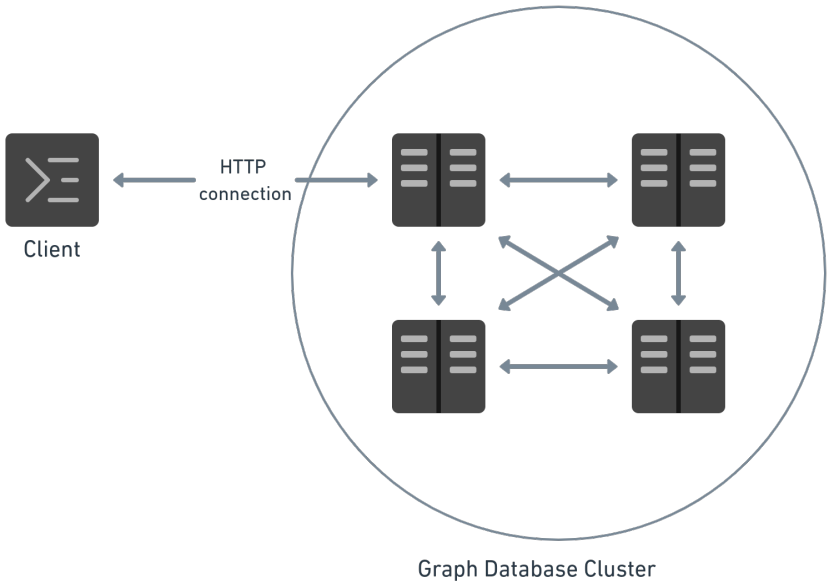

Figure 3.1: GraphoPlex Server Architecture [4]

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

3.4.2 Newly Added Commands

Previously in *GraphoPlex*, there were three types of implemented commands: Database Commands, CRUD Commands, and Path (or Match) Commands.

For Database Commands, there was creating, dropping, and deleting a database, as well as switching to a different database and getting the current active database.

For CRUD Commands, there was creating, updating, and deleting vertices and edges.

For Match commands, there was getting the Shortest Path and their getting the vertices that match a certain path, like K-Hops queries.

There were two different commands that were added to the Database Commands, as well as thirteen new Match commands and an improvement to the already existing Shortest Path command.

Database Commands:

There are two new commands that were added to the database-level commands in GraphoPlex: the assert command, and the reshard command.

1. **Assert Command:** The Assert command is used to assert the type of the graph that is stored in the database. as mentioned before in Section 3.1, support for Graph Theory graphs is to be added to the system. The command helps assert if the data stored is of the given type or not. (See Figure 3.2 For Example)

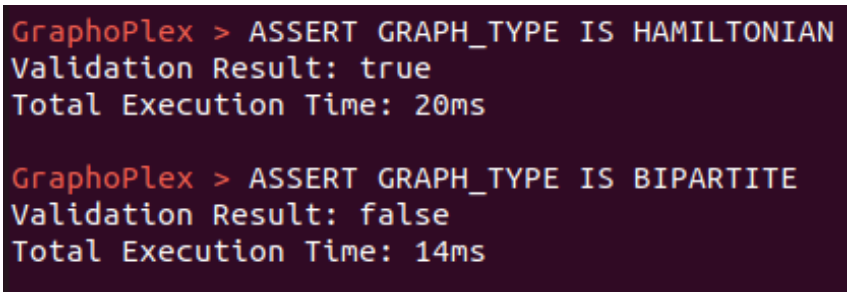

Figure 3.2: Assert Command for Hamiltonian and Bipartite graphs

2. **Reshard Command:** The Reshard command is used to reshard (or re-partition) the graph according to one of seven newly implemented SSs. (See Figure 3.3 For Example)

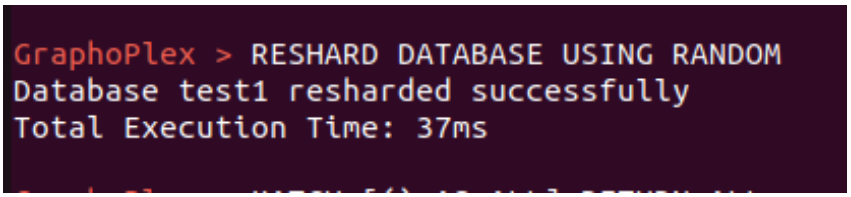

Figure 3.3: Reshard Command using a Random SS

Match Commands:

1. **Shortest Path:** 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. This command was previously implemented in GraphoPlex. However, it was implemented using Dijkstra's algorithm, which fails for negative edge weights. However, now there is an option to tell the command whether or not to use the Bellman-Ford Algorithm, by stating whether or not the cost property has negative edge weights. (See Figure 3.4 For Example)

```
GraphoPlex > MATCH SHORTEST PATH FROM 1 TO 4 WITH COST = cost
Shortest path: 1 ----> 2 {cost=5},2 ----> 4 {cost=1}

Shortest path cost: 6
Total Execution Time: 2ms

GraphoPlex > MATCH SHORTEST PATH FROM 1 TO 4 WITH COST = cost HAS NEGATIVE
Shortest path: 1 ----> 3 {cost=7},3 ----> 4 {cost=-2}
```

Figure 3.4: Shortest Path Query using Dijkstra and Bellman-Ford

2. **All Shortest Paths:** This is one of the thirteen new commands that were added to the system. unlike the previous command, where you get the shortest path between a source vertex and a destination vertex, here you get the cost of the shortest path between all pairs of vertices. This includes the path from the source vertex to itself. (See Figure 3.5 For Example)

```
GraphoPlex > MATCH ALL SHORTEST PATHS WITH COST = cost
From vertex: 1
To vertex: 4 cost: 5
To vertex: 2 cost: 5
To vertex: 3 cost: 7
From vertex: 4
From vertex: 2
To vertex: 4 cost: 1
From vertex: 0
To vertex: 1 cost: 2
To vertex: 4 cost: -2
To vertex: 2 cost: -3
To vertex: 3 cost: 9
From vertex: 3
To vertex: 4 cost: -2


All shortest paths computed successfully
Total Execution Time: 7ms
```

Figure 3.5: All Shortest Paths Query

3. **Topological Sort:** This command basically runs the topological sort algorithm on the vertices in the database and prints the sorted vertices to the terminal. (See Figure 3.6 For Example)

```
GraphoPlex > MATCH TOPOLOGICAL SORT
Sorted Vertices: 0,1,3,2,4

Topological Sort Result
Total Execution Time: 65ms
```

Figure 3.6: Topological Sort Query

4. **Minimum Spanning Tree:** This command simply allows the user to compute the minimum spanning tree of the whole graph that is stored in the database, by specifying the edge property that holds the cost. (See Figure 3.7 For Example)

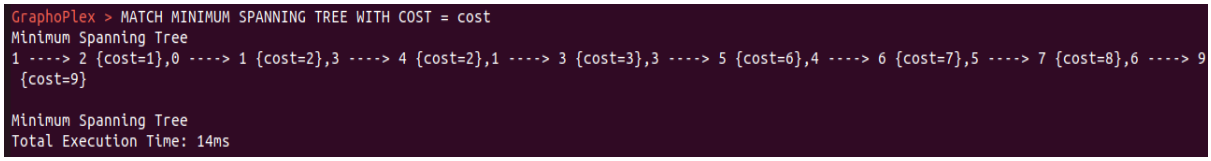

Figure 3.7: Minimum Spanning Tree Query

5. **Maximum Flow:** This command allows the user to compute the maximum flow from a source vertex to a sink vertex in a flow network graph, by having the user specify the capacity property of the graph's edges. (See Figure 3.8 For Example)

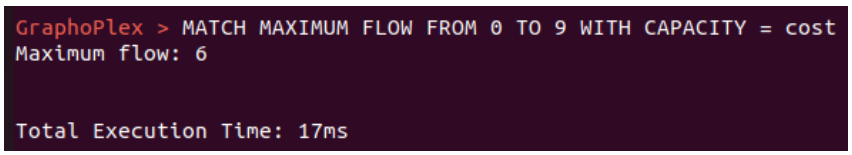

Figure 3.8: Maximum Flow Query

6. **Strongly Connected Components (SCCs):** this command is used to identify any SCCs in the graph. For example it could identify communities in a social network graph. (See Figure 3.9 For Example)

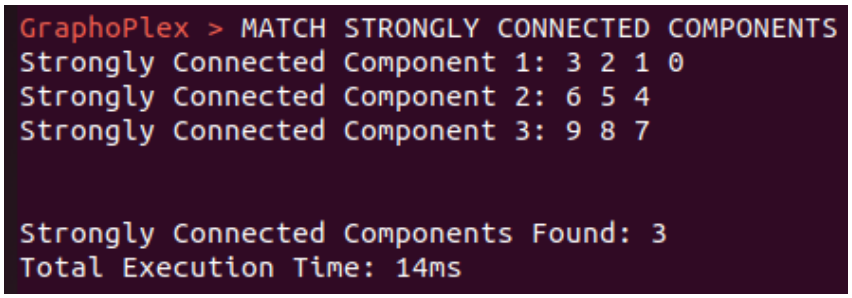

Figure 3.9: Strongly Connected Components (SCCs) query

7. **Bridge Edges:** This command simple tells the user what are the bridge edges that are present in the graph, which are edges whose removal would disconnect the graph or increase its number of connected components. (See Figure 3.10 For Example)

```
GraphoPlex > MATCH BRIDGE EDGES
Bridge Edges: [3 ----> 4 {}, 2 ----> 3 {}, 0 ----> 2 {}, 0 ----> 1 {}]

Bridge edges found: 4
Total Execution Time: 6ms
```

Figure 3.10: Bridge Edges Query

8. **Eccentricity:** This command allows the user to find the eccentricity of a given vertex, which is the greatest distance between that vertex and any other vertex in the graph. (See Figure 3.11 For Example)

```
GraphoPlex > MATCH ECCENTRICITY OF 0
Eccentricity: 2
Total Execution Time: 4ms
```

Figure 3.11: Eccentricity Query

9. **Radius:** This command enables the user to get the radius of the whole graph that is stored in the database. The radius is the minimum eccentricity of any vertex in the graph. It represents the minimum "worst-case" distance from any vertex to the furthest vertex from it. (See Figure 3.12 For Example)

```
GraphoPlex > MATCH RADIUS
Radius: 1
Total Execution Time: 6ms
```

Figure 3.12: Radius Query

10. **Articulation Points:** This command enables the user to get all of the articulation points in the database, which are vertices whose removal increases the number of connected components. (See Figure 3.13 For Example)

```
GraphoPlex > MATCH ARTICULATION POINTS
Articulation Points: 3, 2, 1

Articulation points found: 3
Total Execution Time: 68ms
```

Figure 3.13: Articulation Points Query

11. **Girth:** This command allows the user to calculate the girth of the database, which is the length of the shortest cycle contained in the graph. (See Figure 3.14 For Example)

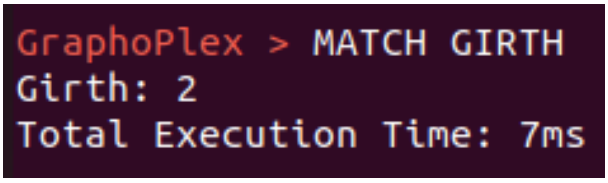

Figure 3.14: Girth Query

12. **Diameter:** Just, like the radius, this command gives the user the diameter of the graph that is stored in the database. The diameter of a graph is the longest shortest path between any two vertices in the graph. It measures the maximum distance between any pair of nodes. (See Figure 3.15 For Example)

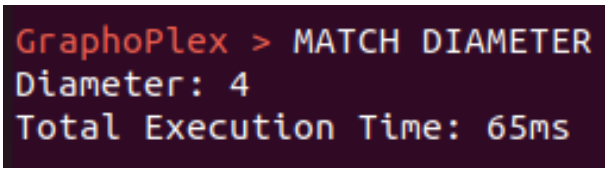

Figure 3.15: Diameter Query

13. **Vertex and Edge Connectivity:** These two commands allow the user to get the vertex connectivity and the edge connectivity of the graph, where edge connectivity is the minimum number of edges that need to be removed to make the graph disconnected and vertex connectivity is the minimum number of vertices that need to be removed to make the graph disconnected. (See Figure 3.16 For Example)

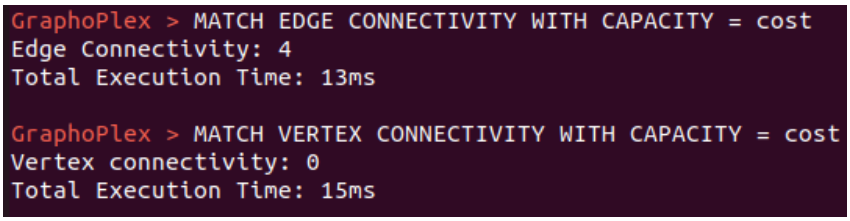

Figure 3.16: Vertex and Edge Connectivity Queries

3.4.3 Newly Added Graph API

There are essentially two main ways for the user to interact with the system.

The first is as a database administrator, through the interactive shell and issuing some of the commands mentioned during this section. The second way is as a programmer through the Graph API and the system connection that was designed to interact with the system through an IDE.

Establishing A Connection:

GraphoPlex can run in two different ways. The first is as a single-node graph database, like Neo4j [11], or as a distributed graph database with a cluster of servers that runs in a P2P architecture [4].

In either way, a connection to the system has to be established. For this purpose, the programmer can use the *GraphoPlexConnection* object, which establishes a connection with the server. If *GraphoPlex* is running as a single-node database on the local machine, then no configuration is needed, since by default the connection connects to the server.

However, if a cluster is running and the programmer wishes to connect, then the programmer must configure the host IP address as well as the port number of one of the servers in the cluster to be able to connect. As seen in Figure 3.17 for example. If a single-node database is used, the `host` and `port` methods can be removed.

```
GraphoPlexConnection conn = new GraphoPlexConnection()
        .host(host)
        .port(serverPort)
        .connect();
```

Figure 3.17: Establishing a connection with a cluster

Creating a Database:

After establishing a connection with the server, the programmer can either switch to pre-existing database, if any are available, or create a new database and then switch to it. See the example in Figure 3.18, where the programmer creates a database called ***testDB***, then switches to it.

```
conn.createDatabase( databaseName: "testDB");
conn.switchDatabase( databaseName: "testDB");
```

Figure 3.18: Creating and Switching to Database ***testDB***

Dropping Or Deleting Databases:

Similarly, the programmer can either drop the data that was stored in a database, if any, or delete the database itself along with any data that was stored in it. In Figure 3.19, the programmer first drops the data of database ***testDB***, then proceeds to delete the database itself.

```
conn.dropDatabase( databaseName: "testDB");
conn.deleteDatabase( databaseName: "testDB");
```

Figure 3.19: Dropping and Deleting Database ***testDB***

Creating A Vertex:

After creating a database and switching to it, the programmer has to start inserting data into the database, and the first thing to be done is to insert a vertex into the database.

Now, the graph model that *GraphoPlex* uses is a Directed Property Graph. This means that each vertex and edge can have a label, as well as properties to represent the data stored. There are two ways for the programmer to create vertices using the connection and the API. (See Figure 3.20 For Example)

The first method is to use the Vertex class from the core data structures to create a vertex object and manually manipulate its data. Then call the `createVertex(Vertex vertex)` method that is present in the connection to create the vertex.

The second method is to write the data in the method immediately using the createVertex(String id, String label, String... properties) method that is also in the connection.

```
Vertex v = new Vertex( id: "v1");
v.setLabel("Person");
v.setProperty("name", "Alice");
v.setProperty("age", "25");
conn.createVertex(v);

conn.createVertex( id: "v2", label: "Person", ...properties: "name:Melissa", "age:26");
```

Figure 3.20: Methods To Create Vertices

Creating An Edge:

Creating an edge is almost exactly similar to creating a vertex. The only exception is that for creating edges, the programmer has to specify the source and destination vertices, a label for the edge if available, and properties, also if available. (See Figure 3.21 For Example)

```
Edge edge = new Edge( sourceVertexId: "v1",  destinationVertexId: "v2");
edge.setLabel("KNOWS");
edge.addProperty("since", "2020");
conn.createEdge(edge);

conn.createEdge( source: "v2",  destination: "v1",  label: "KNOWS",  ...properties: "since:2020");
```

Figure 3.21: Methods of creating an edge

Creating A Full Graph:

The Graph API also comes with an implementation of the Graph data structure, allowing the programmer to create full graphs in memory and manipulate them according to convenience before inserting the data into the database.

The programmer first has to define the type of graph he/she is going to use. There are thirteen different types of graphs from Graph Theory that are supported in API. These graphs are Hamiltonian, Eulerian, Cubic, Interval, Line, Star, Split, Wheel, Tournament, Regular, Grid, Complete Bipartite, and Regular Bipartite Graphs. And, if the programmer does not want to use any of those graphs, he/she can simply use a Normal Graph. Once the programmer creates the graph he/she can use the createGraph(Graph graph) method that is also present in the connection to insert the whole graph. (See Figure 3.22 For Example)

```
Graph graph = new HamiltonianGraph( nodes: 3,  vertices: 3,  edges: 3, GraphType.DIRECTED);
graph.addVertex( id: "v1");
graph.addVertex( id: "v2");
graph.addVertex( id: "v3");
graph.addEdge( source: "v1",  destination: "v2");
graph.addEdge( source: "v2",  destination: "v3");
graph.addEdge( source: "v3",  destination: "v1");

conn.createGraph(graph);
```

Figure 3.22: Creating a full graph using the API

Fetching Basic Graph Utilities:

as it stands right now, *GraphoPlex* has eight graph utilities supported. These utilities are radius, diameter, eccentricity of a vertex, bridge edges, articulation points, girth, edge connectivity, and vertex connectivity. The API gives the programmer the ability, using methods, to fetch the data and print them to the console.

The utilities can be called using the established connection to the database as seen in Figure 3.23 below.

```
conn.radius();
conn.diameter();
conn.bridgeEdges();
conn.articulationPoints();
conn.eccentricity(vertexId);
conn.girth();
conn.vertexConnectivity(capacityProperty);
conn.edgeConnectivity(capacityProperty);
```

Figure 3.23: Example Of Calling the Basic Utility methods

Other Graph Features:

Aside from the above utilities, there are eight other features that can also be fetched and printed to the console. These features are asserting the graph type, resharding the graph, finding the shortest path between two vertices, finding the maximum flow from a source vertex to a sink vertex, finding the minimum spanning tree, finding SCCs, finding the cost of all of the shortest paths in a graph, and finally finding the topological sort of all of the vertices in a graph. Figure 3.24 shows how to call on these functionalities using he established connection.

```
conn.assertGraphType(graphType);
conn.reshardDatabase(shardingStrategy);
conn.shortestPath(sourceId, destinationId, costProperty, hasNegative);
conn.maximumFlow(sourceId, sink, capacityProperty);
conn.minimumSpanningTree(costProperty);
conn.stronglyConnectedComponents();
conn.allShortestPaths(costProperty);
conn.topologicalSort();
```

Figure 3.24: Methods calling other graph features

3.5 Implementation

In this section, the implementation details of the newly added components are to be discussed in detail.

3.5.1 System Modules Thus Far

Previously, GraphoPlex consisted of five main modules: *The Storage Layer*, *The network Layer*, *The Core Data Structures Layer*, *The Graph Algorithms Layer*, and *The Query Manager Layer*.

The client uses the command line interface to send queries to the *Query Manager*. Depending on the type of query, it either forwards it to the *Graph Algorithms Layer* or directly to the *core DS*. The *Core DS Layer* is responsible for data retrieval services, so if the data is in a different server in the cluster, it sends a request to the *Network Layer* to get the data from the other servers. And, if the data is present in the server receiving the query, it directly gets it from the *Data Storage Layer*.

Now, however, we have two new modules added: *The Graph API* and *The Connection Layer*. See Figure 3.25.

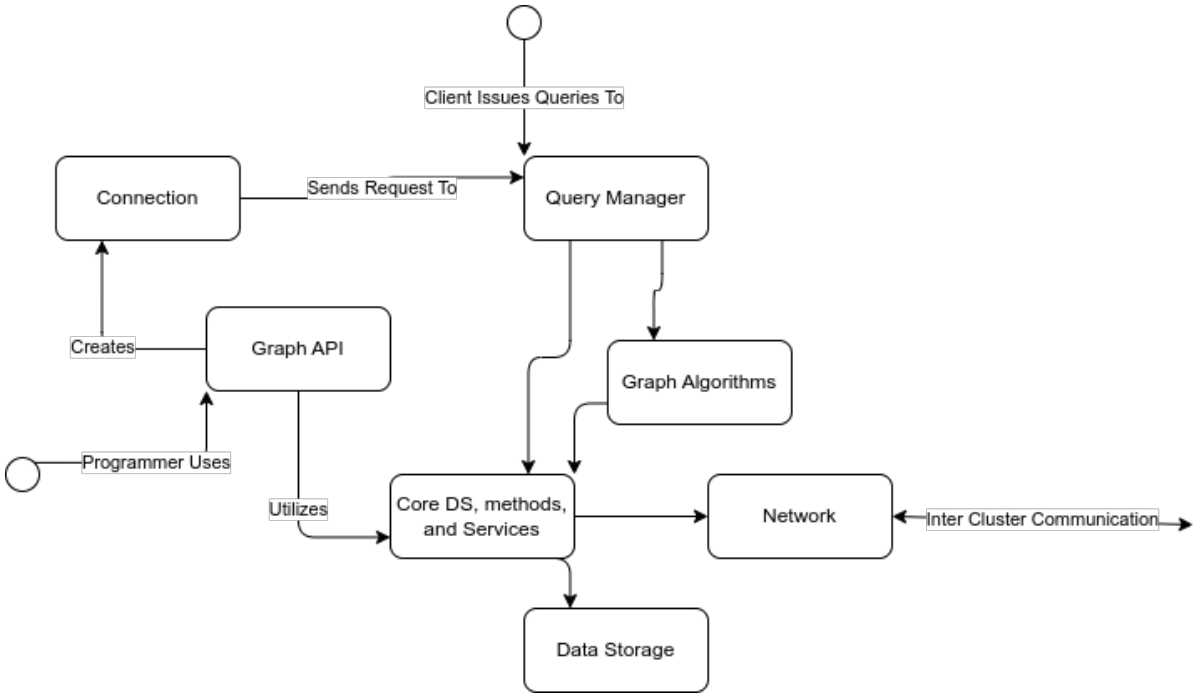

Figure 3.25: System Modules

The client interacts directly with the *Query Manager* through the interactive command line interface. However, the programmer interacts with the System through the

API. The programmer uses the API to create the nessecary graph data that is to be inserted, then the programmer establishes a *Connection* with the System, which connects to one of the servers in the cluster (See Section 3.4.3). Then the connection sends a query to the *Query Manager* to ask for the data that the programmer wants. For example, if the programmer wants the Topological Sort of the vertices in the database, then the connection will send a Topological Sort query to the *Query Manager* (See Figure 3.6).

3.5.2 Inside the Graph API

Firstly, the Graph class is implemented as an abstract class with 11 attributes. There is the node attribute of type integer, which indicates the number of shards or servers available in the cluster. Then there are the vertices and edges attributes of type integer, which tell how many vertices and edges are to be placed in the graph; the programmer sets them manually, like the node attribute. Then there is the *currentNumberOfEdges* attribute, which tells the programmer how many edges have been inserted. After that there is the type attribute, which indicates whether the graph is directed or undirected. Then there are the vertexMap and edgeMap attributes of type HashMap, which store the vertices and edges of the graph. Following that is the colorMap attribute of type HashMap, which is used to check if the graph is bipartite or not and is also used in the sharding strategies. Then there is the properties HashMap, which is used to store the important graph properties and utilities: like the graph type, the articulation points, the bridges, etc. Lastly, we have the shardingStrategy and the testingStrategy attributes, which are used to shard the graph and assert the validity of the graph, respectively. Proper getters and setters for all of the attributes in the graph class are implemented as well.

Next are the 13 different types of graphs, which stem from Graph theory. All of the 13 graph classes, listed in Section 3.4.3, are implemented as inheritors of the abstract graph Class, except for the GridGraph class, which has a row and a column attributes to indicate the number of rows and columns in the graph. Each graph class has its own testing strategy used to validate that the current graph corresponds to the given graph type. (See Figure 3.26 For Example)

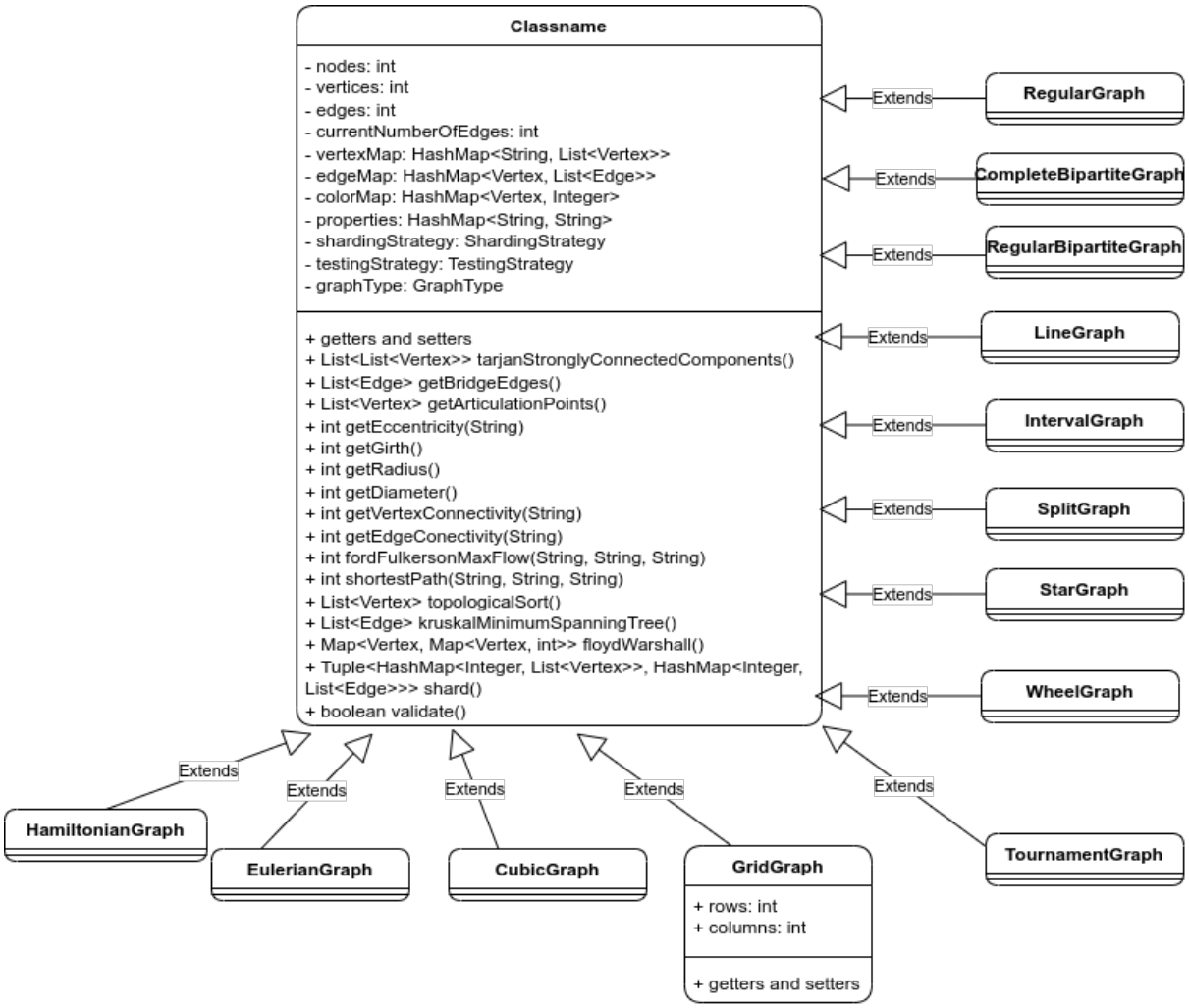

Figure 3.26: Graphs Class Diagram

3.5.3 Testing Strategies (TSs)

The assert query (example in Figure 3.2) work by validating whether or not the graph in the database is of the provided type or not. This is done using the help of the TSs that are used to validate the graphs. These strategies are both part of the API and the *Graph Algorithms Layer*. And, as indicated by the name, the Strategy design pattern is utilized to perform the validation task.

Note in Figure 3.26 that each class has a *testingStrategy* attribute and a (`boolean validate()`) method. In addition, since there are 13 different graphs, there are 13 different TSs (See Figure 3.27 For Example). It is important to note that there is a 14^{th} type of graph, called a NormalGraph; this is basically an ordinary graph. It could be a social network graph, or a Hamiltonian graph, or anything the programmer wants, in the event that he/she do not desire any of the other 13 graphs. It is also important to note, that the validate method of the abstract Graph class calls the validate method of the TS

that is stored in the *testingStrategy* attribute irrespective of the type of graph, and the programmer has the ability to manually change the TS.

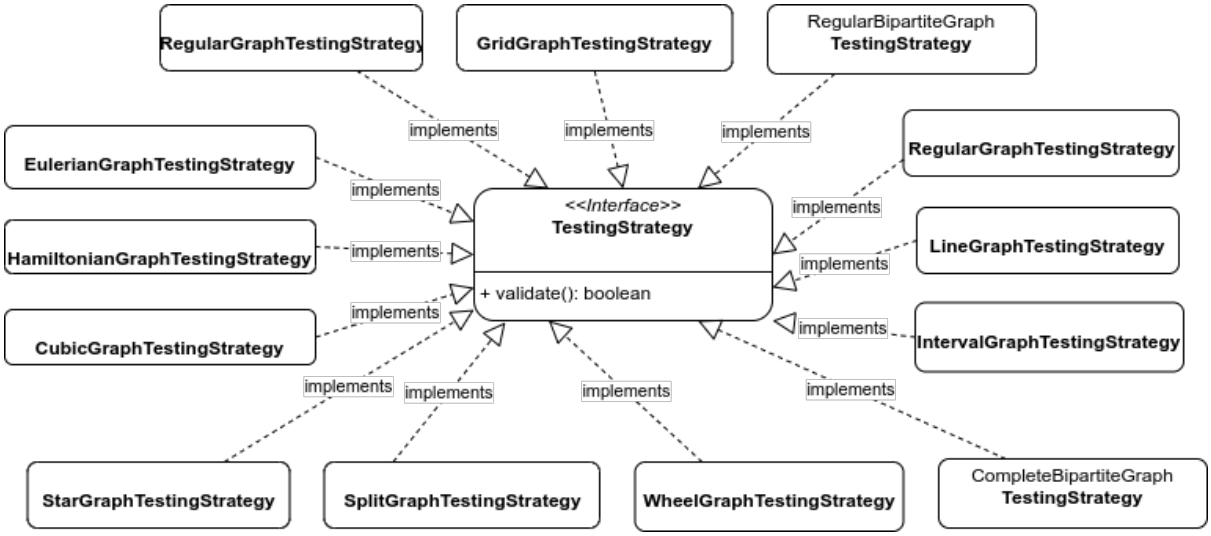

Figure 3.27: Testing Strategies Class Diagram

3.5.4 Sharding Strategies (SSs)

Sharding strategies, also known as partitioning strategies, are basically used to partition the graph and distribute it on the servers in the cluster. Just like the reshard command (refer back to Figure 3.3) gives the client the ability to re-partition the graph, the SSs give the programmer the ability to either partition the graph with a predefined SS or to also reshard the graph after its creation. Like the testing strategies, sharding strategies are also part of the *Graph Algorithm Layer* of the system. And, like the testing strategies, the Strategy Design pattern has been utilized to perform the sharding process.

Referring back to Figure 3.26, it is noted that the abstract Graph class has a (`Tuple shard()`) method, which is used to return a tuple of HashMaps. Each HashMap consists of an integer as a key, representing the server ID, and a list of vertices or edges as value. Since there are ***n*** servers in the cluster, the server IDs are numbered from 0 to ***n - 1*** in both HashMaps. There are currently seven different SSs supported by *GraphoPlex* (See Class Diagram in Figure 3.28) and both the programmer and the client have the ability to shard or reshard using any one of them, with the exception of the Rule-Based SS (More of that later).

The Sharding Strategies:

1. **HashBasedShardingStrategy:** This is the default sharding strategy used by *GraphoPlex*, and it was the first one to be implemented. What happens is that

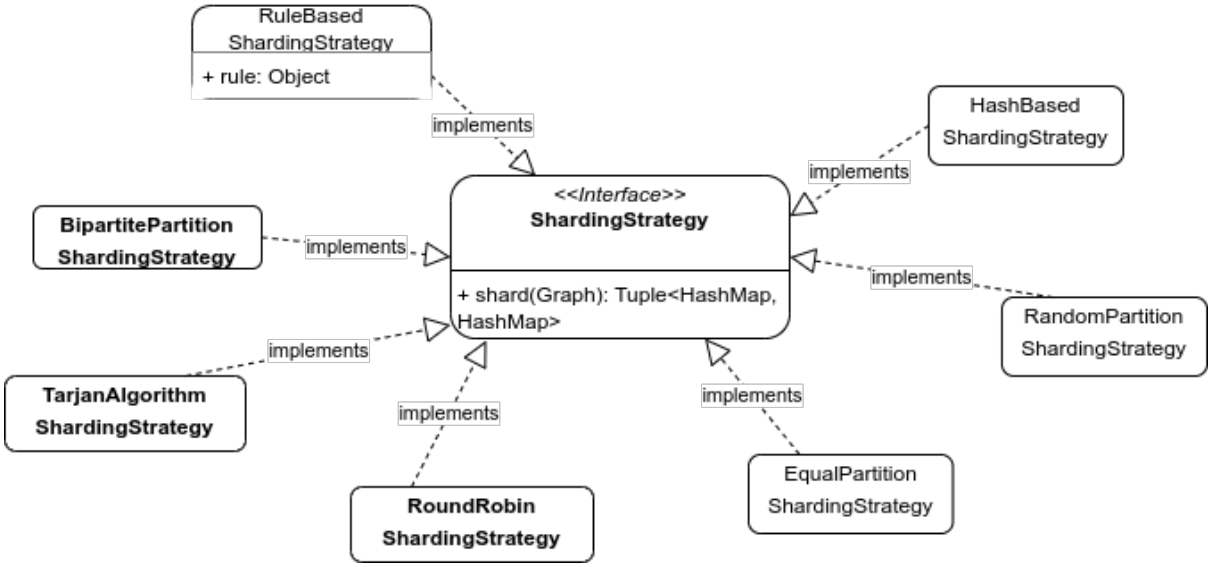

Figure 3.28: Sharding Strategy Class Diagram

the vertices' IDs are hashed by the number of servers (refer back to the (nodes) attribute in Figure 3.26) using a built-in hash function. As for the edges, they are placed in the same shard as the source vertex. Referring to the reshard query (Figure 3.3), the client can reshard the graph with this strategy by replacing the word RANDOM with HASH.

2. **RoundRobinShardingStrategy:** This is one of six newly added sharding strategies. What happens is that, like the name suggests, the vertices are distributed on the shards in a Round Robin fashion. And, if there are still more vertices to be sharded when the final server is reached, then the process is restarted from the first server again until the whole graph is properly partitioned. As for the edges, like the preceding strategy, they are sent to the same partition as the source vertex. Referring to the reshard query (Figure 3.3), the client can reshard the database with this strategy if he/she replaces the word RANDOM with ROUND_ROBIN

3. **RandomPartitionShardingStrategy:** This is another newly added sharding strategy. In this SS, the vertices are randomly assigned to any of the servers in the cluster, while the edges also go to the server that holds their source vertex. The client can reshard the database using this strategy by using the same command in Figure 3.3.

4. **EqualPartitionShardingStrategy:** The third newly added sharding strategy. In this strategy, the vertices are divided equally into $\boldsymbol{n}$ (the number of servers) sets, with each set going to one of the servers. Like all the other strategies, the edges are partitioned with their source vertex. The client can reshard the database using this strategy by replacing the word RANDOM with EQUAL in the command in Figure 3.3.

5. **TarjanAlgorithmShardingStrategy:** This is another new strategy that involves running the tarjan algorithm on the graph to find all of its Strongly Connected Components (SCCs). After this, each SCC is partitioned on the servers in a Round Robin manner, just like in the *RoundRobinShardingStrategy* but this time it is sets of vertices instead of individual vertices. Just like before, the edges are partitioned with the source vertex. The client can reshard the database with this strategy by replacing the word RANDOM with TARJAN in Figure 3.3.

6. **BipartitePartitionShardingStrategy:** This is another sharding strategy that was newly developed. However, this algorithm only works for ***bipartite*** graphs, which is a graph where the vertices can be divided into two disjoint sets such that all edges connect a vertex in one set to a vertex in another set. In *GraphoPlex*, there are currently two types of bipartite graphs: Complete Bipartite and Regular Bipartite. However, if any graph excluding those two is proven to be bipartite, it will get sharded with this strategy. And, the abstract graph class does have a method to check if a graph is bipartite or not. What happens in this strategy is that the vertices are divided into two disjoint sets. Then, the servers in the cluster are divided into 2 ($\frac{\boldsymbol{n}}{2}$) sets of servers. The first set of vertices are hashed to any of the servers in the 1^{st} set of servers, while the second set of vertices is hashed to any of the servers in the 2^{nd} set of servers. Just like all of the others, the edges are partitioned with the source vertex. And, the client can reshard a database with this strategy by replacing the word RANDOM with BIPARTITE in the command in Figure 3.3.

7. **RuleBasedShardingStrategy:** This is the last of the newly implemented sharding strategies. It primarily depends on the Jeasy rule engine. This shardnig strategy takes a rule as input and partitions the graph according to the rule. The rules can be custom made by the programmer. Currently, there is only one rule supported that partitions the vertices of a graph depending on certain attributes in the vertex itself. However, the client using the command line interface to interact with the system will not be able to use this SS to reshard the graph. Only the programmer can shard or reshard a graph with this strategy because it is designed primarily for the API. Unlike the other strategies, the programmer can set the rules so that the edges and vertices are sharded differently, instead of the edges solely relying on their source vertex.

3.5.5 Graph Algorithms

previously, *GraphoPlex* supported the basic algorithms to provide the basic features. Those algorithms were the BFS, Grouped BFS, DFS, Dijkstra's Algorithm, and the A* Algorithm.

Both the BFS algorithms and the DFS algorithm were used for the path queries, while Dijkstra and A* were used to compute the shortest path. While both Dijkstra and A* are

SSSP algorithms, Dijkstra is more of a greedy algorithm that does not work for graphs with negative edge weights and A* is an algorithm that works provided that the vertices have certain attributes to be able to run certain Heuristics on them (i.e. Manhattan or Euclidean).

As the Graph API was implemented, other new algorithms have been implemented to enhance the functionality of the API as well as the System itself. Of the newly implemented Algorithms, only two are implemented independent of the API, while the others require the API to gather the whole graph into memory to process the graph. Here is the list of the new algorithms:

1. ***Bellman-Ford's Algorithm:***

   This is one of the two algorithms implemented, independent of the API. Like Dijkstra and A*, this is another SSSP algorithm used to compute the shortest path between two vertices. However, unlike Dijkstra, it works for graphs with negative edge weights. However, it fails if there is a negative edge cycle. This algorithm is used in the Shortest Path Command (See Figure 3.4) if the user specifies if the attribute to be treated as the cost has negative values or not.

2. ***Topological Sort Algorithm:***

   The second of the algorithms implemented independently of the API. This algorithm is used by the Topological Sort Command (See Figure 3.6) to sort the vertices in the graph and return the sorted result.

3. ***Kruskal's Algorithm:***

   This algorithm is used to compute the Minimum Spanning Tree of a graph if given the edge property that would act as the cost. The algorithm is used in the minimum spanning tree command (See Figure 3.7). The implementation of this algorithm is in the Graph class. (See Figure 3.26 the (`kruskalMinimumSpanningTree`) method).

4. ***Ford-Fulkerson's Algorithm:***

   This algorithm is used to compute the maximum flow from a source vertex to a sink vertex if given the edge property that will act a the capacity. This algorithm is used in the Maximum Flow Command (See Figure 3.8). The implementation of this algorithm is also in the Graph class (See Figure 3.26 the (`fordFulkersonMaxFlow`) method).

5. ***Tarjan's Algorithm:***

   This algorithm is used to compute the SCCs of a graph. It is used in the Strongly Connected Components Command (See Figure 3.9). The implementation is also in the Graph class (See Figure 3.26 the (`tarjanStronglyConnectedComponents`) method).

6. ***Floyd-Warshall's Algorithm:***

   This algorithm is used to compute the cost of the shortest path between all pairs of vertices, including any vertex to itself, if given the edge property that will act as the cost. The algorithm was used in the All Shortest Paths Command (See Figure 3.5). The implementation is also in the Graph class (See Figure 3.26 the (`floydWarshall`) method).

Algorithm 1 Floyd-Warshall Algorithm

Let INF be the maximum integer value divided by 2
Initialize *distanceMap* as a new HashMap
for all $vertex$ in $vertexMap.values()$ **do**
  Initialize *vertexDistance* as a new HashMap
  for all *edge* in $edgeMap.getOrDefault(vertex, Collections.emptyList())$ **do**
    $vertexDistance.put(vertexMap.get(edge.getDestinationVertexId()), Integer.parseInt(edge.getProperty(costProperty)))$
  end for
  $distanceMap.put(vertex, vertexDistance)$
end for
for all k in $vertexMap.values()$ **do**
  for all i in $vertexMap.values()$ **do**
    for all j in $vertexMap.values()$ **do**
      if $distanceMap.get(i).containsKey(k)$ and $distanceMap.get(k).containsKey(j)$ **then**
        $throughK \leftarrow distanceMap.get(i).get(k) + distanceMap.get(k).get(j)$
        $direct \leftarrow distanceMap.get(i).getOrDefault(j, INF)$
        $distanceMap.get(i).put(j, \min(direct, throughK))$
      end if
    end for
  end for
end for
return distanceMap

3.6 Comparison with Related Works

3.6.1 Benchmarks

GraphoPlex follows all of the operations stated in the ***BlueBench*** benchmark, except for the *LoadGraphML* and the *FindEdgesByProperty* operations. And, regarding the ***TGDB*** benchmark, GraphoPlex also satisfies all of its specifications. However, when it come to *Graph Analysis*, *GraphoPlex* does not satisfy everything. For example, there are no cluster coefficient calculations in *GraphoPlex*.

3.6.2 Distributed Graph Databases

	Random	Hashing	METIS	Others
GraphoPlex	Yes	Yes	No	Yes
A1	Yes	No	No	No
ParallelGDB	Yes	No	No	No
Trinity	No	No	Yes	No
Acacia	No	No	Yes	No
System G	No	Yes	No	No
ByteGraph	No	Yes	No	No
Ho et al. [7]	No	No	Yes	No
HGraph [2]	No	No	No	Yes

Table 3.1: Supported Partitioning Strategies

As obvious from Table 3.1 above, GraphoPlex supports a wide range of partitioning strategies. Just like **A1** and **ParallelGDB**, it supports Random partitioning. And, just like **System G** and **ByteGraph** it supports Hashing as a partitioning strategy. However, unlike **Trinity** and **Acacia**, *graphoPlex*, does not support the METIS algorithm as a partitioning algorithm. In addition, unlike **HGraph** (Adoni et al. (2021) [2]), it also does not use the DPHV algorithms as a partitioning strategy.

Looking at Table 3.2, it is clear that, unlike **Acacia** and **ByteGraph**, all of the distributed graph databases, including *GraphoPlex*, support the K-Hops query. And, like **ParallelGDB** and **A1**, *GraphoPlex* does not support the PageRank query. In addition, it supports Single Source Shortest Path (SSSP) queries, like **Trinity** and **ByteGraph**. As for the BFS or DFS queries, *Graphoplex* supports both, while **Trinity**, **ParallelGDB**, and **System G** only support BFS queries. As for the Strongly Connected Component (SCC) queries, only *graphoPlex* and **System G** support them.

	K-Hops	PageRank	SSSP	BFS or DFS	SCC	Others
GraphoPlex	Yes	No	Yes	Yes	Yes	Yes
A1	Yes	No	No	No	No	No
ParallelGDB	Yes	No	No	Yes	No	No
Trinity	Yes	Yes	Yes	Yes	No	No
Acacia	No	Yes	No	No	No	No
System G	Yes	Yes	No	Yes	Yes	No
ByteGraph	No	Yes	Yes	No	No	Yes
Ho et al. [7]	Yes	Yes	No	No	No	Yes

Table 3.2: Supported Queries

3.7 Summary

In this the Design Goals of enhancing GraphoPlex, a previously implemented distributed graph database, were discussed. In addition, the Design Strategy and the overview of the major components of the system where mentioned. Furthermore, the architecture of GraphoPlex was mentioned before the mention of the newly added commands and the newly added Graph API. After that the implementation of the Graph API was discussed. And in the end, a brief comparison with the previous works mentioned in Chapter 2 Section 2.2.

Chapter 4

Conclusion

4.1 Achievements

The aim of this thesis was to enhance the already existing distributed graph database, **GraphoPlex**, by adding new commands to the implemented query language and developing a Graph API. Numerous new commands have been added to the system and they can all be seen in Section 3.4.2. In addition, a Graph API has been implemented, which allows programmers to load their own graphs and manipulate them before having to load them into the database (detail can be seen in Section 3.4.3). Furthermore, the API supports thirteen different graphs from Graph Theory, which is something that has not been introduced in any graph database engine before, distributed or not.

4.2 Limitations

There are many limitations that face **GraphoPlex** as of right now. Here are some of them:

1. ***Testing:***

   Due to the fact that there were not many machines available, the performance of GraphoPlex in a truly distributed environment is still unknown.

2. ***No Parallel Algorithms:***

   There are many important graph algorithms that do not have distributed implementation, like Dijkstra's Algorithm and Ford-Fulkerson's Algorithm. So the only way for GraphoPlex to run these algorithms is sequentially.

3. ***Not All Graph Theory Graphs are supported:***

   so far only thirteen of the 26 graphs that are found in Graph Theory are supported.

4. ***Graph Connection:***

   For the API to be able to send data to a database a connection has to be established. the problem is when the programmer tries to query this data back, the connection only prints to the console, and does not return tangible data.

4.3 Future Works

Here are some imporvements and evaluations that can be done in the future:

1. ***Testing GraphoPlex:***

   If more machines can be found to test GraphoPlex in a truly distributed environment, that will help settle how good GraphoPlex actually is against other distributed database. In addition, not only the database that should be tested, but also the API as well.

2. ***Adding More Graphs:***

   As previously mentioned, the Graph API in GraphoPlex only supports 13 of 26 graphs from Graph Theory. If more can be added that would be benficial in the future.

3. ***Parallel Algorithms:***

   Right now, there is a severe lack in graph algorithms that can be run in a distributed manor. Finding parallel implementations of said algorithms would be benefical for the effeciancy of the database engine.

4. ***Adding More Commands:***

   Right now, there many algorithms that are not implemented, such are Prim's algorithm. For example, some algorithms take graph labels into account, while Tarjan's Algorithm does not.

5. ***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.

4.4 Summary

To summarize, this thesis work was divided into 3 phases. The first phase was to review and compare existing distributed graph database systems and their system design choices, as well as discuss certain benchmarks. Secondly, the Design goals as well as the implementation of the newly added features to GraphoPlex, a previously developed distributed graph database, were discussed. Finally, the limitations of the current system as well as future work to improve it were mentioned.

Appendix

Appendix A

Lists

BFS	Breadth First Search
DFS	Depth First Search
SSSP	Single Source Shortest Path
SS	Sharding Strategy
TS	Testing Strategy
TGDB	Toronto Graph Database Benchmark
DRAM	Dynamic Random Access Memory
RDMA	Remote Direct Memory Access
FaRM	Fast Remote Memory
DGS	Distributed Graph Storage
TSL	Trinity Specification Language
SLA	Service Level Agreement
JVM	Java Virtual Machine
P2P	Peer-to-Peer
RPC	Remote Procedure Call
API	Application Programming Interface
OOP	Object-Oriented Programming
SCC	Strongly Connected Component

List of Figures

List of Tables

Bibliography

[1] Z. Abul-Basher, M. H. Chignell, P. Godfrey, and N. Yakovets. Tgdb: towards a benchmark for graph databases. *IBM Corp*, 2016.

[2] W. Y. H. Adoni, N. Tarik, M. Krichen, and A. El Byed. Hgraph: Parallel and distributed tool for large-scale graph processing. In *2021 1st International Conference on Artificial Intelligence and Data Analytics (CAIDA)*, pages 1–7. IEEE, 2021.

[3] L. Bargu, V. Munts-Mulero, D. Dominguez-Sal, and P. Valduriez. Parallelgdb: a parallel graph database based on cache specialization. *Association for Computing Machinery*, 2011.

[4] Elshymaa Bateh. Distributed graph databases, 2023.

[5] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 329–344, New York, NY, USA, 2020. Association for Computing Machinery.

[6] M. Dayarathna and T. Suzumura. Towards scalable distributed graph database engine for hybrid clouds. *IEEE Press*, 2014.

[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*, pages 455–462, 2012.

[8] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, page 1695–1698. Association for Computing Machinery, 2017.

[9] V. Kolomienko, M. Svoboda, and I. H. Mlnkov. Experimental comparison of graph databases. *Association for Computing Machinery*, 2013.

[10] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. Bytegraph: a high-performance distributed graph database in bytedance. *Proc. VLDB Endow.*, 15(12):3306–3318, 2022.

[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)*, page 5, Manchester, United Kingdom, October 23–25 2021. ACM.

[12] Bin Shao, Haixun Wang, and Yang Li. Trinity: a distributed graph engine on a memory cloud. *Association for Computing Machinery*, 2013.

[13] G. Tanase, T. Suzumura, J. Lee, C. F. Chen, and J. Crawford. System g distributed graph database. *arXiv preprint arXiv:1802.03057*, 2018.