AN INCREMENTAL ENHANCEMENT OF AGENT-BASED GRAPH DATABASE SYSTEM

SHENYAN CAO

A whitepaper
submitted in partial fulfillment of the
requirements of the degree of

Master of Science in Computer Science and Software Engineering

University of Washington
June 3rd, 2024

Abstract

In the domain of big data analytics, a graph database (DB) system is vital for managing complex data structure. This project delves into the enhancement of an existing agent-based graph DB system leveraging the Multi-Agent Spatial Simulation (MASS) Java library. Motivated by the limitations of the existing agent-based graph DB system, this project aims to re-engineer the structure of the existing system to make it capable to handle complex dataset, aligning with the property graph model. Furthermore, it endeavors to integrate with Cypher, a graph query language, to enhance the querying capabilities of the agent-based graph DB system.

Building upon the previous work within the MASS framework, which incorporates a distributed graph data structure, this project seeks to enhance functionality and efficiency in managing large and complex graph datasets. Through a comparative analysis of popular industrial graph DB systems such as Neo4j, RedisGraph, JanusGraph, and ArangoDB, this project establishes design principles focusing on the adoption of the property graph model, Cypher query language, in-memory distributed graph structures, and agent utilization.

Detailed insights into the design and implementation processes are provided, which includes parsing Cypher queries into Abstract Syntax Trees (AST), planning execution strategies, and executing queries to traverse graph and manipulating graph data. Comprehensive testing ensures the functionality of the enhanced agent-based graph DB system. Results include successful execution and verification of building the property graph from csv files, along with the execution and verification of CREATE and MATCH cypher queries.

In summary, this project demonstrates the successful extension of the agent-based graph DB system to handle complex node and relationship property information, accurate execution of CREATE and MATCH cypher queries, and outlined plans for future development. This project report encapsulates the goals, design, implementation, and evaluation of the project, underscoring the significance of enhancing the agent-based graph DB system for handling complex and interconnected data structures.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Background and motivation

With the rise of big data and the need to analyze interconnected data, graph DB system has gained popularity as a valuable tool for managing and exploring large and complex datasets [1]. A graph DB is a type of DB that uses graph structures to represent and store data. In a graph DB, data is organized as nodes and edges. Nodes represent entities, and edges represent the relationships between those entities. For instance, in a film industry graph DB, nodes can represent various entities such as movies, actors, directors, genres, and production companies. Edges between nodes signify relationships, such as actors starring in movies or directors directing films. This graph-based representation allows for efficient exploring and analysis of complex relationships and connections within the data. A graph DB system refers to the entire software ecosystem built around the graph DB. It encompasses not only the storage mechanism but also the query language, data management tools, indexing mechanisms, and other features that enable users to interact with and manipulate the data stored in the graph DB [1,2].

Traditional methods in the field of big data analytics have heavily relied on data streaming tools like Hadoop MapReduce, and Apache Spark to handle and analyze vast amounts of data [3]. These tools have been instrumental in enabling real-time data processing and building scalable analytics pipelines. Their support for big-data computing and data sciences, with data formats primarily in text, has made them widely adopted tools. However, it is important to note that these data streaming tools may face limitations when it comes to analyzing complex data structures such as graphs. Data streaming tools often operate based on the principle of dividing data into smaller chunks and processing them individually. This approach may not be ideal for complex data structures like graphs, as they consist of interconnected nodes and edges. Decomposing and streaming such structures through memory can lead to challenges in maintaining the integrity of relationships and may require additional processing steps to reconstruct the complete structure.

When dealing with a dataset with graph structure, it is more logical to retain the structure in distributed memory and deploy agents, (i.e., autonomous execution entities) within it, as opposed to breaking down the structure and streaming with traditional data streaming tools [4]. Agents hold significant potential in supporting the analysis of graph DB due to their ability to be deployed repetitively within data structures mapped over distributed memory. With agent-based graph DB system, it can achieve efficient graph construction and analysis, facilitating the processing of highly interconnected data [5].

Throughout previous years, the Distributed Systems Laboratory (DSL) at the University of Washington has developed an agent-based graph DB system that leverages the MASS Java library. Their approach involves constructing graphs over a cluster system, deploying numerous reactive agents to datasets, and assigning these agents the task of computing data attributes or relationships. [6]. Nevertheless, the current graph DB system is structured to accommodate nodes and edges with specific properties. Each node in the graph is defined by its unique identifier (ID), and edges are characterized by the IDs of connected nodes along with a weighted relationship. This simplicity of the existing agent-based graph DB system may be suitable for scenarios where the relationships between entities can be adequately represented using only these three attributes. However, it is essential to recognize that a more complex data model with additional attributes would be required for applications where richer information about nodes and relationships is necessary. In the film industry example provided earlier, a complex data model might be used to represent people and movies with detailed relationships, each with its own properties. The property graph model, with its ability to represent nodes, relationships, and attributes, aligns seamlessly with the requirements of such a complex data model in the film industry. This capstone project seeks to re-engineer the existing agent-based graph DB system to enhance its capability to handle complex datasets with property information about nodes and relationships in accordance with the property graph model.

When utilizing a graph DB system, a query mechanism is typically necessary for creating, manipulating, and retrieving data. However, the existing agent-based graph DB system does not support any query language. Cypher, as an open query language designed for graph databases, is particularly optimized for querying property graph models. This capstone project seeks to integrate Cypher with the agent-based graph DB system by implementing CREATE and MATCH Cypher clauses.

Given the motivations outlined, our DSL team is presently dedicated to improving the existing agent-based graph DB system and evaluating its performance using benchmark applications. While Michelle Dea is tasked with building these benchmark applications, utilizing various large graph datasets to measure the execution performance of MASS, Neo4J, and ArangoDB, my responsibility involves enhancing the functionality of the agent-based graph DB system. The project goals for this capstone work are as follows: 1) re-engineer the existing agent-based graph DB system to enhance its capability to handle graph datasets with various property information about nodes and relationships in accordance with the property graph model, 2) integrate Cypher into the agent-based graph DB system by implementing CREATE and MATCH Cypher clauses, and 3) conduct comprehensive testing on the functionality of the enhanced agent-based graph DB system to ensure its functionality meet the desired standards.

2. Previous Work

This section aims to provide an in-depth overview of the MASS Java library and the existing agent-based graph DB system. It also discusses challenges posed by the existing agent-based graph DB system and outlines the tasks intended to mitigate these challenges within the scope of this project.

2.1 Existing MASS Java library

At its core, MASS leverages a distributed architecture to manage and process large-scale datasets efficiently. In the distributed environment, the system operates across multiple computing nodes, with each equipped with its own memory and processing capabilities. Data is stored in memory for efficient access and manipulation. The MASS Java library excels at managing simulations where entities, represented as Agents, dynamically interact with each other and their environment, depicted as Places [7].

Places serve as the spatial framework upon which simulations are constructed. They are the elements organized in a matrix and distributed across a cluster of computing nodes. Each Place instance is uniquely identified by a set of matrix indices and possesses the ability to exchange information stored within it with other places within the simulation [7]. Once initialized, a Place instance remains fixed on its respective computing node throughout program execution.

Agents represent the dynamic entities that inhabit and interact within the spatial framework defined by Places [7]. They are the execution instances capable of residing in a specific place and interact with the data stored within the place they inhabit. Agents can also spawn new agents, terminate themselves, migrate between places, and interact with other agents and places. Agents are organized into bags and allocated to processes within the distributed computing environment, with threads managing their execution and interaction, which facilitates parallel execution.

MASS leverages the parallel execution of Places and Agents through multithreaded communication processes, which are forked across cluster nodes and interconnected via Transmission Control Protocol (TCP) sockets [8]. Utilizing message passing, places and agents can exchange data, coordinate distributed computations, and synchronize updates to maintain data consistency across the distributed environment. TCP ensures reliable and ordered delivery of messages between places and agents.

Figure 2.1 illustrates the distribution of places across a cluster of three computing nodes, each with four execution threads for parallel manipulation of places and agents [8]. An Agents instance, containing a bag of agents, is associated with each Places instance. Parallel computation involving Places and Agents is facilitated using methods like Places.callAll(func) for invoking a specified function across all places, Agents.callAll(func) for executing a given function across all agents, Places.exchangeAll(func) for exchanging data among neighboring places via a function call, and Agents.manageAll() for handling agent creation, termination, and migration operations scheduled in the last Agents.callAll().

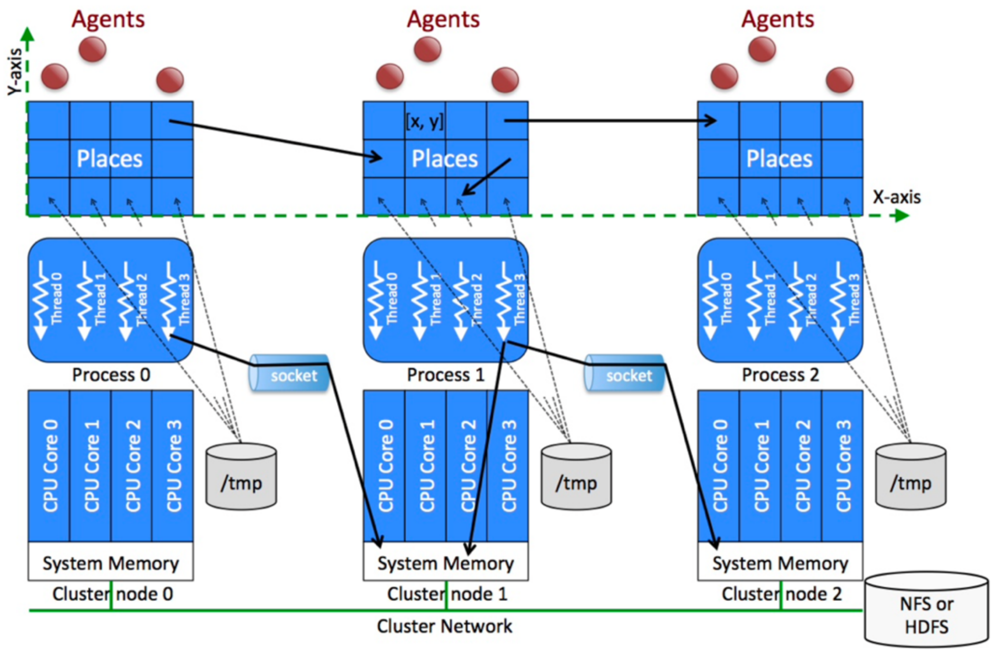


Figure 2.1: MASS library architecture with Places and Agents [8].

2.2 Existing agent-based graph DB system

In recent years, DSL has built with an agent-based graph DB system leveraging the MASS Java library [9]. The agent-based graph DB system comprises two essential components: GraphPlaces and GraphAgent. GraphPlaces, as explained in Table 2.1, extends from Places and includes a collection of VertexPlace, which further extends from Place. Meanwhile, GraphAgent extends from Agent.

Table 2.1: Graph DB system leveraging the MASS Java library.

MASS Basic Library	**MASS Graph Library**
`Places`	⟸ `GraphPlaces`
Contains ⇓	*Extends* *Contains* ⇓
`Place`	⟸ `VertexPlace`
`Agents`	
Contains ⇓	*Extends*
`Agent`	⟸ `GraphAgent`

2.2.1 GraphPlaces and VertexPlace

The GraphPlaces class is designed to store and manage graph data. GraphPlaces serve as fundamental components for a graph DB. It provides functionalities for creating and modifying graph structures. GraphPlaces instances embody a graph consisting of vertices and edges. Vertices are represented by instances of VertexPlace, while edges are implicitly defined by the edge attributes of the VertexPlace instances. Figure 2.2 describes the distribution of three GraphPlaces instances across a cluster of three computing nodes. It adheres to the in-memory distributed structure of Places but stores places in Vectors, which can dynamically expand to accommodate large datasets.

Figure 2.3 is partial code of the VertexPlace class, which defines edges using a list of neighbor IDs stored in Vector<Object> neighbors and neighbor edge weights stored in Vector<Object> weights. Figure 2.4 is an example of the VertexPlace instances distributed evenly across a cluster of three computing nodes. Each VertexPlace is characterized by three attributes: its unique identifier (ID), a list of neighbor IDs, and a list of corresponding edge weights. The edges within this representation are all outgoing edges and are implicitly defined by the associated lists of neighbor IDs and edge weights within each VertexPlace.

The simplicity of the existing graph DB system may be suitable for straightforward scenarios where the relationships between entities can be adequately represented using only these three attributes (vertex ID, neighbor ID, edge weight). However, it poses limitations when dealing with more complex datasets requiring richer information about nodes and their connections. Consider the earlier example within the film industry, the dataset must store detailed information about people and movies, including their relationships. Nodes could represent individuals (e.g., actors, directors) and movies, each with numerous attributes such as names, birthdates, genres, release dates, and ratings. Edges would represent relationships such as ACTED_IN and DIRECTED. Each of these relationships could have additional attributes. For instance, the ACTED_IN relationship might include the role played and the year of performance. The existing agent-based graph DB system fails to adequately store and manage such complex datasets with rich and detailed node and edge information.

This project aims to re-engineer the existing system by adopting the property graph model, which addresses the above limitations. The property graph model organizes data into nodes, edges and properties. Nodes represent entities in the graph, relationships define connections between nodes, and properties store key-value pairs associated with nodes and relationships. The property graph model allows for flexible and efficient representation of complex datasets with rich relationships between entities. Neo4j, RedisGraph and JanusGraph are all based on the property graph model [10]. Property graph model is particularly well-suited for scenarios where the relationships between data points are as important as the data points themselves, such as social networks, recommendation systems, and network infrastructure management.

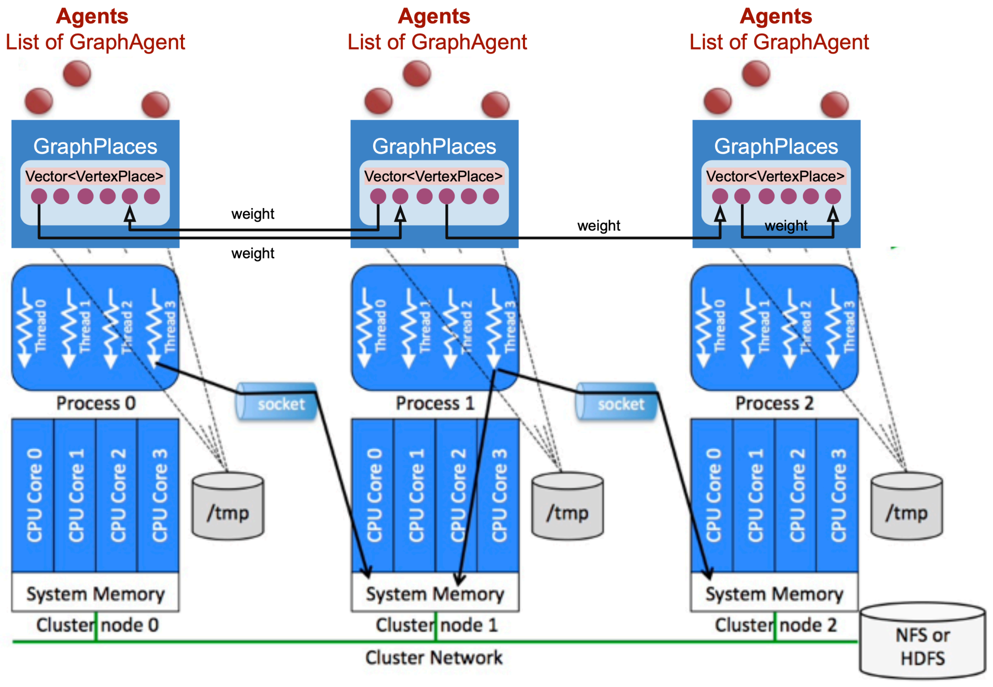


Figure 2.2: The existing agent-based graph DB system with GraphPlaces.

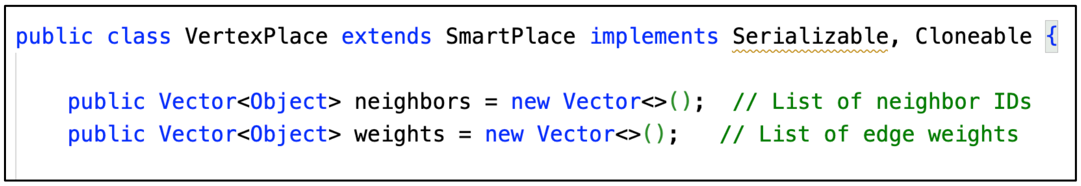

```
public class VertexPlace extends SmartPlace implements Serializable, Cloneable {

    public Vector<Object> neighbors = new Vector<>();  // List of neighbor IDs
    public Vector<Object> weights = new Vector<>();   // List of edge weights
```

Figure 2.3: Attributes in the VertexPlace class.

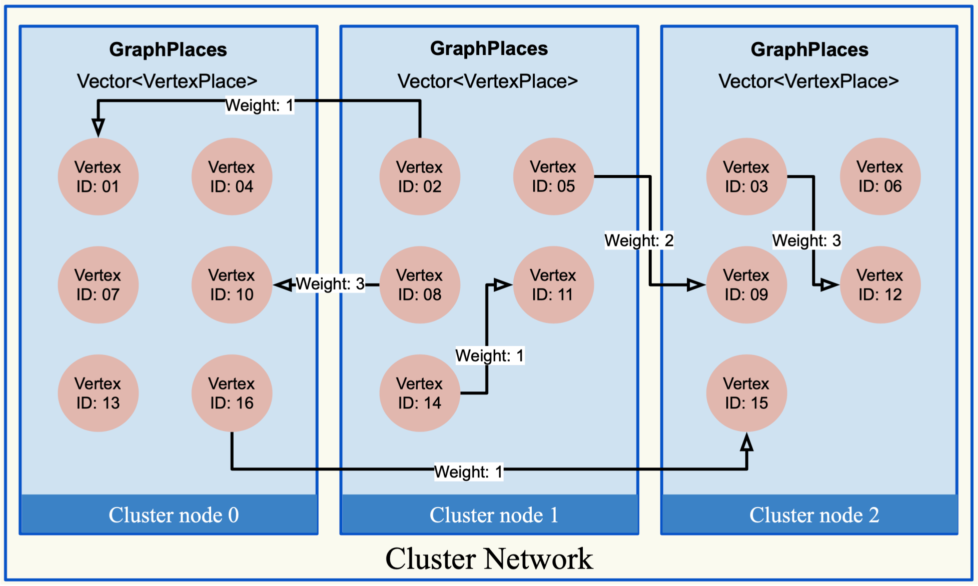


Figure 2.4: An illustration of GraphPlaces and VertexPlace in a cluster network.

2.2.2 GraphAgent

The GraphAgent class inherits from Agent and updates the map function with one that distributes the initial population of agents across the graph. GraphAgent also incorporates the necessary operations for agents to function properly within the graph data structure, including spawn(), migration(), and kill(). Following the graph's construction, agents could migrate from one vertex to another through the edges. The current GraphAgent lacks the capability to retain path results during graph DB traversal. This project extends GraphAgent class to a new class named PropertyGraphAgent to address the above limitation. This extension will empower agents to carry path results as they navigate and explore the graph DB, enhancing their functionality within the system.

3. Related Works

In this section, we describe the data storage and management in four industrial graph DB systems, Neo4j, RedisGraph, JanusGraph, and ArangoDB, outlining their key strengths and weaknesses. The selection of these graph DB systems is driven by their significant presence and popularity in the industry.

3.1 Neo4j

Neo4j is a leading graph DB system designed to efficiently store, manage, and query highly interconnected data. It excels in handling complex relationships and interconnectedness inherent in real-world datasets, making it ideal for applications like social networks, recommendation engines, and many more [2].

Neo4j offers two primary deployment options: single-instance and clustered. In a single-instance setup, Neo4j runs on a single computing node. This setup is suitable for smaller-scale deployments or development environments where the entire graph DB can fit on a single machine's resources. While this approach offers advantages such as low latency and high throughput, it may pose limitations in scenarios requiring scalability beyond the capacity of a single machine. Scaling resources vertically by upgrading hardware is the only option, which may not be cost-effective or sufficient for long-term growth.

Neo4j can also be deployed in a clustered environment, where multiple instances of Neo4j across different computing nodes collaborate to form a distributed system. In a clustered setup, each computing node stores a portion of the graph DB on its local disk. Neo4j also employs an extensive caching mechanism to improve query performance by caching frequently accessed data in memory. While Neo4j's clustered environment caters to scalability requirements by distributing graph data across multiple nodes, inter-node communication in the clustered setup may introduce higher latency compared to a single-instance environment, especially over networks with high latency or congestion. Besides, as data is stored on disks, it typically involves slower access times compared to in-memory storage due to the physical read or write operations required to access data from the disk. These can result in performance degradation, affecting overall system responsiveness and query execution times.

3.2 RedisGraph

RedisGraph is an in-memory graph DB system deployed with a single-instance setup [12]. Leveraging Redis's in-memory storage, RedisGraph facilitates rapid traversal and querying of graph data. However, storing graph data in the memory of a single computing node can lead to substantial memory requirements, particularly for sizable datasets, which constrains its scalability.

Redis announced the phase-out of RedisGraph in 2023, citing multiple reasons for the decision. A primary factor was the higher-than-expected costs of expanding support and sales capacities for RedisGraph, despite its technical competitiveness. Redis aims to concentrate on segments with more substantial growth potential, such as Search and Query, JSON, and Vector features within Redis Enterprise.

Despite the phasing out of RedisGraph, we continue to examine its advantages and challenges, as its in-memory deployment is comparable to our agent-based graph DB system. The main limitation of RedisGraph lies in its scalability within a single-node environment. In contrast, deployment in our agent-based graph DB system is configured within a distributed environment comprising multiple computing nodes. Our approach addresses the scalability issue present in RedisGraph while enabling low latency and high efficiency in graph traversal and manipulation with data store in memory.

3.3 JanusGraph

JanusGraph is an open-source graph DB released in 2017 [13]. It offers two primary ways to store data: in-memory storage and disk-based storage. In the in-memory storage option, graph data is stored exclusively in the memory of a single computing node. Similar to RedisGraph, JanusGraph with the in-memory storage option faces challenges related to scalability.

JanusGraph also supports disk-based storage, which can be distributed across a cluster network. JanusGraph supports various distributed storage backends such as Apache Cassandra, Apache HBase, Google Cloud Bigtable, and others. Similar to Neo4j's clustered setup, JanusGraph with disk-based storage is highly scalable but may encounter potential latency issues and performance degradation.

3.4 ArangoDB

ArangoDB stands out as a multi-model DB system, accommodating various data formats like key-value pairs, documents, and graphs [14]. ArangoDB can be deployed in various configurations to suit different needs [15]. Running a single server instance of ArangoDB is the simplest way to start, where the ArangoDB server binary is run stand-alone without replication, failover capabilities, or clustering with other nodes. However, single server instances may face scalability limitations similar to Neo4j's single-instance setup.

ArangoDB clusters, on the other hand, consist of multiple servers and facilitates synchronous data replication between servers [15]. Similar to Neo4j’s clustered setup, ArangoDB's clustering feature offers high scalability but may experience latency issues and performance degradation.

3.5 Summary of related works

RedisGraph and JanusGraph’s single-node setup leverages in-memory storage, enabling rapid traversal and querying of graph DBs with low latency. However, in a single computing node setup, they may face scalability constraints due to limited memory size. Similarly, Neo4j and ArangoDB encounter scalability issues in their single-instance setups due to limited disk space. While distributed setups are available for Neo4j, JanusGraph, and ArangoDB to distribute graph data across multiple computing nodes, they may encounter potential latency issues and performance degradation due to disk-based storage and the distributed environment.

As explained in the previous sections, the existing agent-based graph DB system exhibits two limitations. Firstly, it cannot store and manage complex datasets containing property information pertaining to nodes and edges. Secondly, it lacks support for any query languages. To tackle the constraints of the existing agent-based graph DB system and address the scalability, latency, and performance degradation issues encountered by industrial graph DB systems, our project adheres to four design principles outlined below and further elaborated in Section 4: 1) adopting property graph model to handle complex datasets; 2) supporting Cypher query language with a standardized query interface; 3) adhering to the existing in-memory distributed graph structure; and 4) using agents and PropertyGraphAgent for graph traversal and analysis.

4. Design and Implementation

This section will elaborate on the project's overall design, as well as the design and implementation of crucial components, aligning with the four design principles outlined in Section 3.

4.1 Overall Design

The entire design and implementation of this project is presented in Figure 4.1. The main design of the whole project follows a three-tier architecture pattern, with presentation layer, logic layer and data access layer. Presentation Layer is about the interface that users interact with. Users have no direct access to the underlying graph data or knowledge of the DB structure. Users make requests to the system through the GraphManager component. Logic Layer serves as an intermediary between the presentation layer and the data access layer. This layer contains the GraphManager class, which exposes create, read, or update methods to users, providing a high-level interface for interaction with graph DB. Data Access Layer is where the actual agent-based property graph DB resides. Users, through the GraphManager, interact with the DB using Cypher queries.

When a user inputs a Cypher query to query the graph DB, GraphDBHandler will call GraphManager's queryHandler() method, which is responsible for handling Cypher queries. In the queryHandler method, the Cypher query text undergoes parsing into AST representation using the CypherVisitor method. This AST serves as the foundation for constructing an execution plan tailored to the query's requirements. Through the invocation of ProeprtyGraphCypherQuery.execute() method, the execution plan is built, and each step is executed within the provided context, interacting with agent-based graph DB via the PropertyGraphCypherQueryContext interface. The underlying structures of the graph DB and query handler are abstracted from users, ensuring they only need to be concerned with the high-level create, read, or update operations, while the Graph DB and Cypher Handler manages the low-level details of graph DB and query parsing and execution.

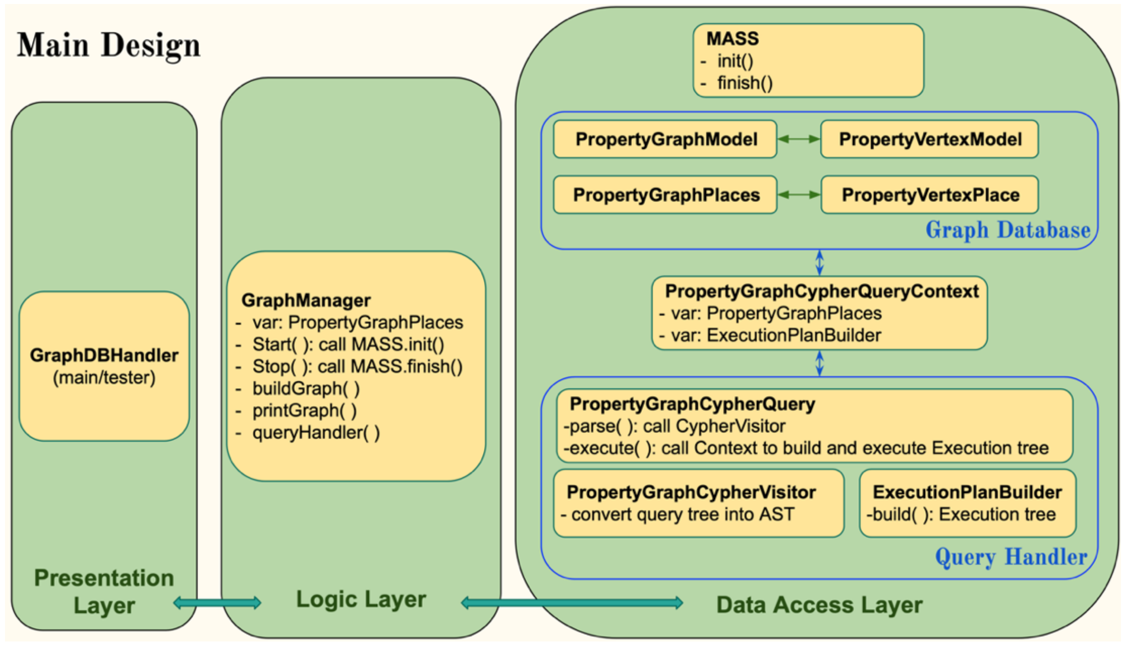

Main Design
MASS
- init()
- finish()
PropertyGraphModel
PropertyVertexModel
PropertyGraphPlaces
PropertyVertexPlace
Graph Database
PropertyGraphCypherQueryContext
- var: PropertyGraphPlaces
- var: ExecutionPlanBuilder
PropertyGraphCypherQuery
-parse( ): call CypherVisitor
-execute( ): call Context to build and execute Execution tree
PropertyGraphCypherVisitor
- convert query tree into AST
ExecutionPlanBuilder
-build( ): Execution tree
Query Handler
GraphDBHandler
(main/tester)
GraphManager
- var: PropertyGraphPlaces
- Start( ): call MASS.init()
- Stop( ): call MASS.finish()
- buildGraph( )
- printGraph( )
- queryHandler( )
Presentation Layer
Logic Layer
Data Access Layer

Figure 4.1: Overall design of the whole project.

4.2 Design Principle 1: Adopting Property Graph Model

The existing agent-based graph DB system poses limitations when dealing with complex datasets with property information about nodes and edges. This project aims to re-engineer the existing system by leveraging the property graph model, which is capable of representing data with nodes, edges and properties. This subsection provides a comprehensive overview of the property graph model, elaborating on its key concepts and principles. Subsequently, it delves into an explanation of the implementation process for enhancing the existing agent-based graph DB in accordance with the principles of the property graph model.

4.2.1 Property graph model

For domains with inherently connected and interdependent data, such as social networks, or supply chains, the property graph model provides a natural and efficient way to model and represent complex node information and relationships between nodes. The Property Graph

Model is a type of graph model that represents data as a graph, consisting of nodes, relationships, and properties [1,2,11].

Nodes are the entities (discrete objects) in the graph, which can be tagged with labels, representing their different roles in a domain [10]. Nodes can also hold any number of key-value pairs, or properties. The simplest possible graph is a single node with no relationships. Consider the following graph as shown in Figure 4.2, it consists of a single node and represents an entity in the film industry. The node labels are Person and Actor. The node properties are “name: Tom Hanks” and “born: 1956”.

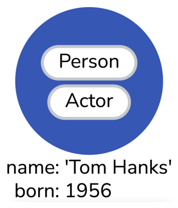


Figure 4.2: An illustration of a node in the property graph model [10].

A relationship defines the connection between a source node and a target node. Relationships provide directed and named connections between two node entities. Relationships always have a direction, a type, a start node, and an end node, and they can have properties, just like nodes [10]. Nodes can have any number or type of relationships. Although relationships are always directed, they can be navigated efficiently in any direction. Relationships play a pivotal role in structuring nodes, enabling a graph to take on various forms such as a list, a tree, a map, or a compound entity. These structures can be further combined to create intricate, interconnected systems. Figure 4.3 is an example of a relationship example in a film industry. The relationship type is ACTED_IN. The properties are “roles: ['Forrest']” and “performance: 5”.

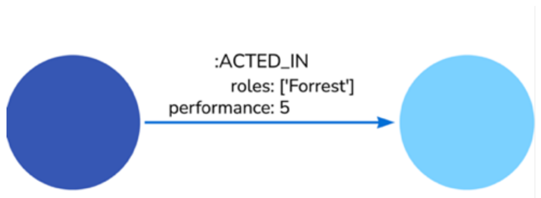


Figure 4.3: An illustration of relationship in the property graph model [10].

A path in the property graph model represents a sequence of nodes and relationships traversed from one node to another. Figure 4.4 is an example of a path example in the film industry. Path captures the route or journey through the graph from a starting point to an endpoint. It can include multiple nodes and relationships, providing valuable insights into the connections between entities in the graph. Traversals in property graph DB involve navigating through a path by following relationships from one node to another based on certain criteria.

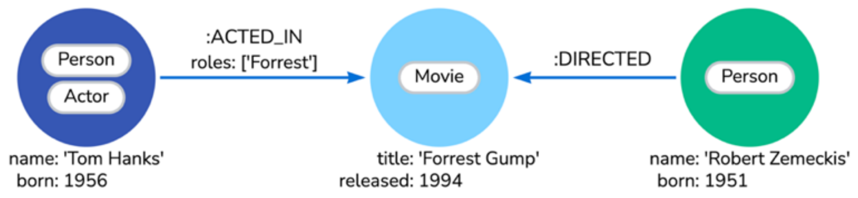


Figure 4.4: A path demonstration in the property graph model [10].

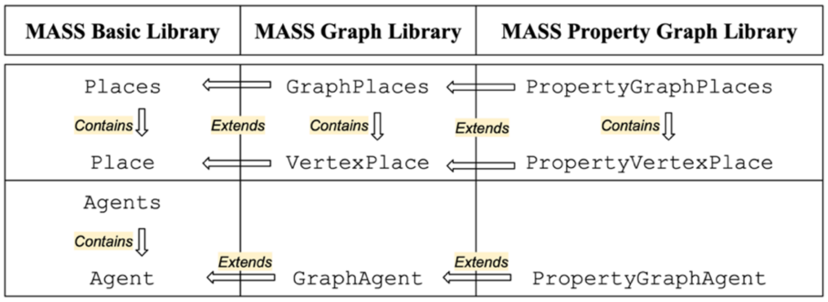
4.2.2 Implementation of property graph model

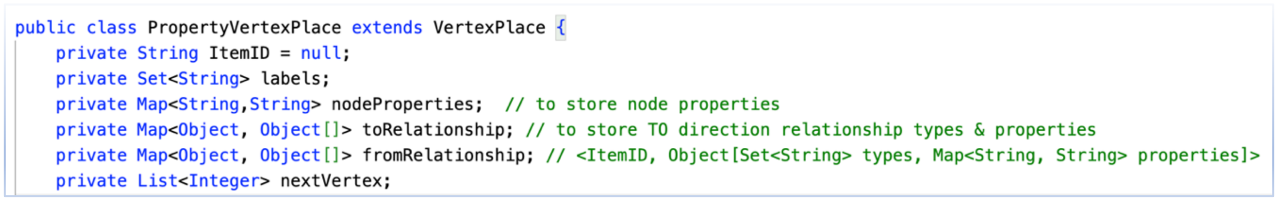
As demonstrated in Table 4.1, to enhance the functionality of the existing system in accordance with the property graph model, a new PropertyGraphPlaces class is designed and implemented to extend GraphPlaces, while a new PropertyVertexPlace class is designed and implemented to extend VertexPlace.

The PropertyVertexPlace class is designed to represent nodes or vertices in a graph structure in accordance with the property graph model. The Java code snippet, as shown in Figure 4.5, presents the implementation of the PropertyVertexPlace class. The key attributes of the PropertyVertexPlace class include 'labels' and 'nodeProperties' for denoting node label and property information, along with 'toRelationship' and 'fromRelationship' for retaining relationship details. The ‘nextVertex’ is list to store the vertex IDs of nodes where an agent is expected to migrate to.

Figure 4.6 is an illustration of the PropertyGraphPlaces instances and the PropertyVertexPlace instances in a cluster network with three computing nodes. The PropertyGraphPlaces instance has a collection of the PropertyVertexPlace instances, which are distributed evenly across the cluster network. With its methods, such as addPropertyVertex(), setRelationEdge(), propertyGraphCallAll(), and getPropertyGraph(), it is tailored to streamline the creation, modification, and examination of property graphs.

Table 4.1: Enhanced property graph DB system leveraging MASS Java library.





```
public class PropertyVertexPlace extends VertexPlace {
    private String ItemID = null;
    private Set<String> labels;
    private Map<String,String> nodeProperties;  // to store node properties
    private Map<Object, Object[]> toRelationship; // to store TO direction relationship types & properties
    private Map<Object, Object[]> fromRelationship; // <ItemID, Object[Set<String> types, Map<String, String> properties]>
    private List<Integer> nextVertex;
```

Figure 4.5: Attributes in the PropertyVertexPlace class.

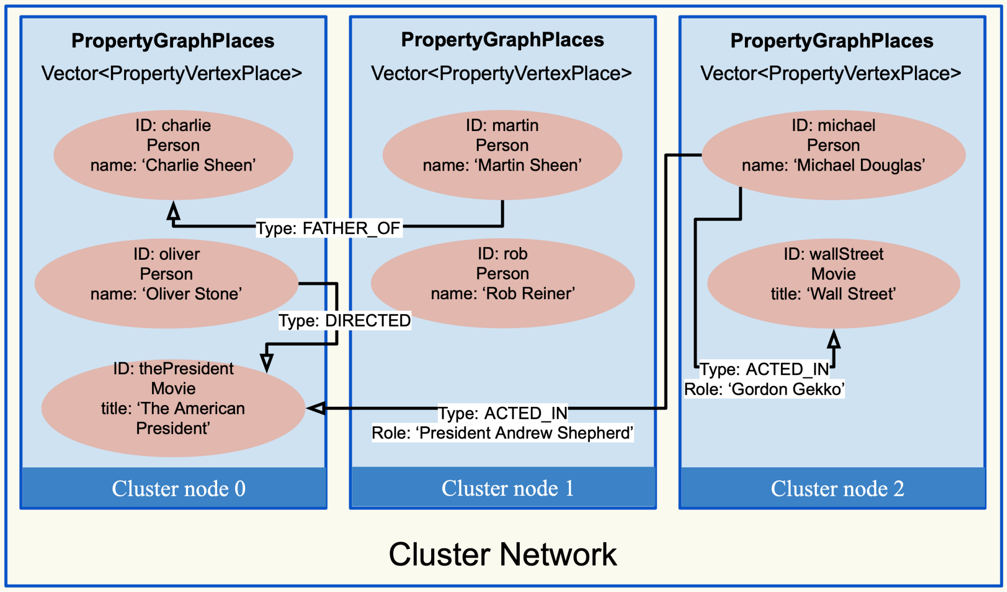


Figure 4.6: An illustration of PropertyGraphPlaces and PropertyVertexPlace in a cluster network.

4.3 Design Principle 2: Supporting Cypher Query Language

The existing agent-based graph DB system does not support any query languages. Cypher is a declarative query language specifically designed for querying and manipulating graph data stored property graph DB systems, such as Neo4j and RedisGraph [1]. This project aims to integrate Cypher with our agent-based graph DB system to provide users with a standard query interface. This subsection provides an overview of the key patterns and principles in the Cypher query language. Subsequently, it delves into detailing the Cypher processing pipeline to integrate Cypher with our agent-based graph DB system.

4.3.1 Cypher Query Language

Cypher provides a user-friendly and expressive syntax for interacting with nodes, relationships, and attributes within the property graph model, making it an ideal tool for navigating and analyzing complex data structures like those found in the film industry example. In illustrations of the property graph model, nodes are represented as circles, and relationships as arrows. In Cypher, these circles are denoted by a pair of parentheses, referred to as node patterns, while the arrows are represented as dashes accompanied by either a greater-than or less-than symbol, referred to as relationship patterns [16].

In Cypher, users can target specific nodes in the graph with node pattern by specifying criteria such as labels, properties, or a combination of both. Every graph pattern contains at least one node pattern. Figure 4.7 is a detailed illustration of a MATCH clause applied to a single node pattern, which is targeting an entity in a film industry graph database. In this demonstration, the node comprises the following components: a label denoted as ‘Movie’, a property with key of 'title' and value of 'Wall Street', and an assigned variable, denoted as ‘mv’, which enables the referencing of designated nodes in subsequent clauses, aiding in the construction of complex queries. The initial MATCH clause locates all nodes labeled as ‘Movie’ within the graph possessing the ‘title’ property set to ‘Wall Street’, associating them with the variable mv. Subsequently, this variable, mv, is propagated to the subsequent RETURN clause, which retrieves and presents the information associated with the same node.

```
MATCH (mv:Movie {title: 'Wall Street'})
RETURN mv
```

Figure 4.7: A demonstration of a MATCH clause applied to a single node pattern.

Relationship is represented in Cypher with dashes and/or an arrow to indicate its direction. The simplest possible relationship pattern is a pair of dashes (‘--‘). This pattern specifies a relationship with any direction and does not filter on any relationship type or property. To specify a particular direction, an arrow '<' or '>' is added to the left or right-hand side of dashes respectively (‘-->’ or ‘<--‘). Figure 4.8 includes three detailed MATCH queries applied with different directional relationship patterns, targeting the relationships in a film industry graph database. These query examples match for relationships of type ‘ACTED_IN’ and property with key of ‘role’ and value of ‘Bud Fox’. Unlike nodes, information within a relationship pattern must be enclosed by square brackets. The first query does not specify the direction of the relationships, so it matches relationships in any direction between node a and node b. The second query specifically retrieves relationships with direction from node a to node b. The third query specifically retrieves relationships with direction from node b to node a.

```
01.  MATCH (a)-[r:ACTED_IN {role: 'Bud Fox'}]-(b)
     RETURN r
02.  MATCH (a)-[r:ACTED_IN {role: 'Bud Fox'}]->(b)
     RETURN r
03.  MATCH (a)<-[r:ACTED_IN {role: 'Bud Fox'}]-(b)
     RETURN r
```

Figure 4.8: A demonstration of three MATCH clauses applied to a single relationship pattern.

In the property graph model, a path represents a sequence of nodes and relationships that form a connected chain within the graph. Path patterns in Cypher mirror this structure, encompassing a sequence of node patterns and relationship patterns. A path pattern must include at least one node pattern and also initiate and terminate with a node pattern. It must alternate between node and relationship patterns. By specifying start and end nodes, as well as optional conditions along a path pattern, users can construct complex queries to extract valuable insights from the graph data. Figure 4.9 is a visual way of matching a path pattern in Cypher. This path pattern includes two node patterns and one relationship pattern. This MATCH clause seeks to fetch path results within a graph DB where a node representing a person named 'Dan' is connected by an outgoing 'LOVES' relationship to another node representing a person named 'Ann'.

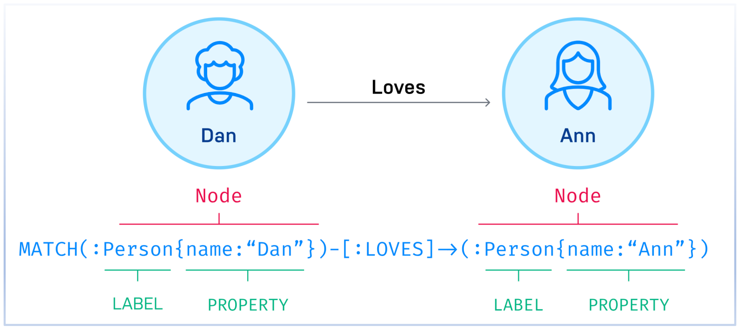


Figure 4.9: A visual way of matching path pattern in Cypher.

4.3.2 Implementation of Cypher processing pipeline

Figure 4.10 describes the Cypher query processing pipeline. It begins with the input of a Cypher query text, which specifies the query clause with patterns to be performed on the graph data. Once the Cypher query is received, it undergoes parsing into a parse-tree, a hierarchical representation of its syntactic structure, and then being translated into an Abstract Syntax Tree (AST), which captures the essential semantic structure of the parse-tree. Upon the construction of AST, it is transformed into an execution plan, which is essentially a tree of execution steps required to fulfill the query efficiently. Once the execution plan is generated, it is executed against the agent-based property graph DB and the results of the query execution are retrieved. Throughout this pipeline, our project leverages optimization techniques and algorithms to ensure efficient query processing and timely retrieval of results from the graph DB system, thereby facilitating effective data querying and analysis.

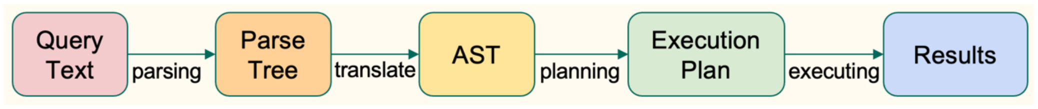
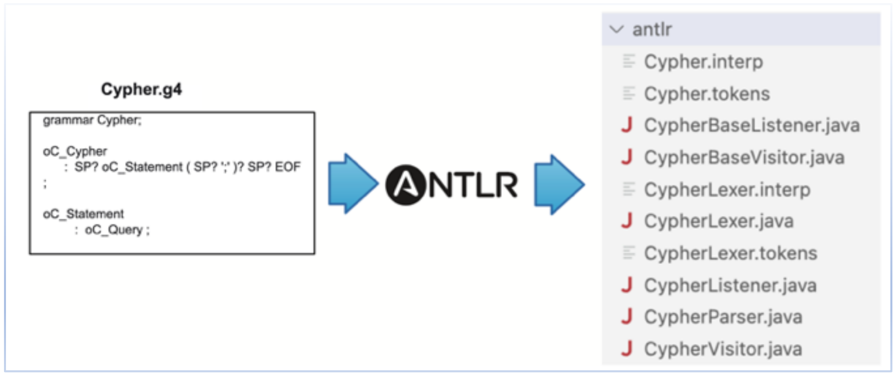


Figure 4.10: Cypher processing pipeline.

For converting query text into a parse-tree, ANTLR4 (ANother Tool for Language Recognition, version four) stands out as an optimal choice due to its capabilities as a powerful parser generator [17]. There exists an ANTLR4 grammar file specifically tailored for Cypher, named Cypher.g4.

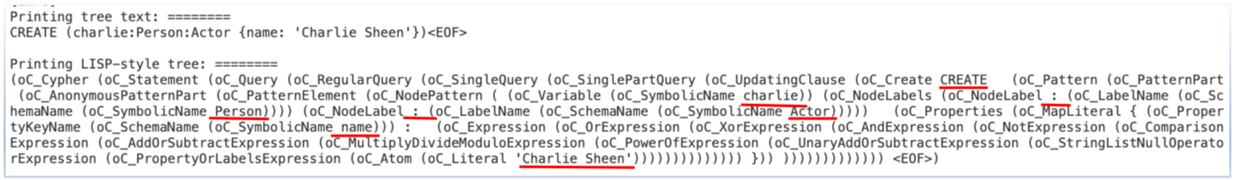
Upon incorporating Cypher.g4 into the application with the ANTLR4 plugin, ANTLR automatically generates numerous code files corresponding to Cypher.g4, as illustrated in Figure 4.11. The input query text first undergoes conversion to CharStream, followed by the generation of lexer with CypherLexer and parse-tree by CypherParser sequentially. Appendix C1 presents the code snippet for parsing the query string to parse-tree.

In a parse-tree, each node corresponds to a syntactic element of the query, such as keywords, identifiers, operators, or expressions. Analyzing the parse-tree with LISP-style tree structure could make it easier to understand and analyze the grammatical structure of the parse-tree. Figure 4.12 shows the LISP-style tree structure for the parse-tree of the query clause "CREATE (charlie:Person:Actor {name: 'Charlie Sheen'})". Each node in this tree structure corresponds to a specific syntactic construct in accordance with the methods defined in the CypherVisitor interface. By aligning the tree structure with the CypherVisitor interface's methods, we can establish a direct correspondence between the two. This alignment facilitates the translation of parse-tree into AST using the CypherVisitor mechanism.



```
grammar Cypher;

oC_Cypher
    : SP? oC_Statement ( SP? ';' )? SP? EOF
;

oC_Statement
    : oC_Query ;
```

Figure 4.11: ANTLR4 autogenerates code files from Cypher.g4.



```
Printing tree text: ========
CREATE (charlie:Person:Actor {name: 'Charlie Sheen'})<EOF>

Printing LISP-style tree: ========
(oC_Cypher (oC_Statement (oC_Query (oC_RegularQuery (oC_SingleQuery (oC_SinglePartQuery (oC_UpdatingClause (oC_Create CREATE   (oC_Pattern (oC_PatternPart
 (oC_AnonymousPatternPart (oC_PatternElement (oC_NodePattern ( (oC_Variable (oC_SymbolicName charlie)) (oC_NodeLabels (oC_NodeLabel : (oC_LabelName (oC_Sc
hemaName (oC_SymbolicName Person)))) (oC_NodeLabel : (oC_LabelName (oC_SchemaName (oC_SymbolicName Actor)))))   (oC_Properties (oC_MapLiteral { (oC_Proper
tyKeyName (oC_SchemaName (oC_SymbolicName name))) :   (oC_Expression (oC_OrExpression (oC_XorExpression (oC_AndExpression (oC_NotExpression (oC_Comparison
Expression (oC_AddOrSubtractExpression (oC_MultiplyDivideModuloExpression (oC_PowerOfExpression (oC_UnaryAddOrSubtractExpression (oC_StringListNullOperato
rExpression (oC_PropertyOrLabelsExpression (oC_Atom (oC_Literal 'Charlie Sheen')))))))))))))) })) ))))))))))))) <EOF>)
```

Figure 4.12: The grammatical structure of a CREATE query text.

Table 4.2 enumerates the key nodes in the parse-tree that require translation to the AST using specific CypherVisitor functions. These nodes are then transformed into execution steps through distinct ExecutionPlanBuilder functions. For instance, the oC_CREATE node in the parse tree is converted to a CypherCreateClause in the AST using the visitOC_Create() method of the CypherVisitor. Similarly, the oC_PatternPart node becomes a CypherPatternPart in the AST after being processed by the visitOC_PatternPart() method. Once in the AST, the CypherCreateClause node is handled by the ExecutionPlanBuilder's visitCreateClause() method, resulting in the creation of a JoinExecutionStep. Likewise, the CypherPatternPart node in the AST is processed by the ExecutionPlanBuilder's visitCreateClausePatternPart() method, generating a CreatePatternExecutionStep. In our project, the PropertyGraphCypherVisitor class is designed to implement the CypherVisitor interface. The function flow of PropertyGraphCypherVisitor is explained in Appendix B1, and its corresponding AST is shown in Appendix B2. Appendix B3 presents a detailed tree structure of execution steps based on our current project implementation. The code snippet for the PropertyGraphCypherVisitor class can be found in Appendix C2. Appendix C3 includes a Java code snippet illustrating the construction of an execution plan for a CREATE clause.

Table 4.2: The relation between parse-tree, AST and execution steps.

Parse Tree	CypherVisitor function	AST	ExecutionPlanBuilder function	Execution Step
oC_CREATE	visitOC_Create()	CypherCreateClause	visitCreateClause()	JoinExecutionStep
oC_PatternPart	visitOC_PatternPart()	CypherPatternPart	visitCreateClausePatternPart()	CreatePatternExecutionStep
oC_NodePattern	visitOC_NodePattern()	CypherNodePattern	visitCreateNodePattern()	CreateNodePatternExecutionStep
oC_RelationshipPattern	visitOC_RelationshipPattern()	CypherRelationshipPattern	visitCreateRelationshipPattern()	CreateRelationshipPatternExecutionStep

4.4 Design Principle 3: In-Memory Distributed Graph Structure

In the four industrial graph DB systems studied in this section, scalability issues arise from single computing node setups, while high latency and performance degradation stem from disk read and write operations in cluster networks. This project aims to enhance the agent-based graph DB system by adhering to its existing in-memory distributed graph structure, which will ensure high efficiency in reading and writing data from memory and address the scalability issue with a cluster network. The in-memory distributed graph structure of the existing agent-based graph DB system was shown in Figure 2.2, with GraphPlaces stored in the memory of each computing node and distributed across a cluster network. The enhanced graph DB system also maintains such structure. Figure 4.13 illustrates the distribution of three PropertyGraphPlaces instances across a cluster network comprising three computing nodes. Each PropertyGraphPlaces instance stores a segment of graph data and resides in the memory of one computing node in the cluster network.

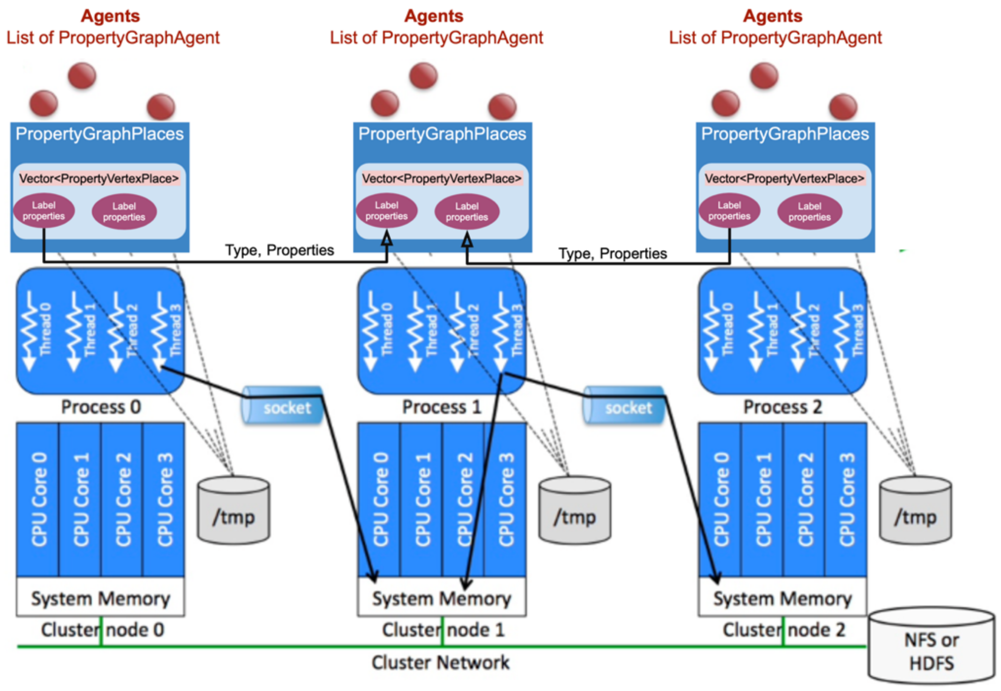


Figure 4.13: The enhanced agent-based graph DB system with PropertyGraphPlaces.

4.5 Design Principle 4: Agents and PropertyGraphAgent

Neo4j, JanusGraph and ArangoDB all face the challenge of performance degradation due to inter-node communication in the distributed environment. In this project, agents are deployed to explore the graph DB, aiming to minimize inter-node communication and improve system performance. Besides, the PropertyGraphAgent class empowers agents to carry path results as they navigate and explore the graph DB. This subsection will outline the initialization process of agents in the enhanced agent-based graph DB system. It will then elaborate on the process of agents traversing the graph DB to retrieve results upon the execution of MATCH query clauses.

4.5.1 Initialization of Agents and PropertyGraphAgent

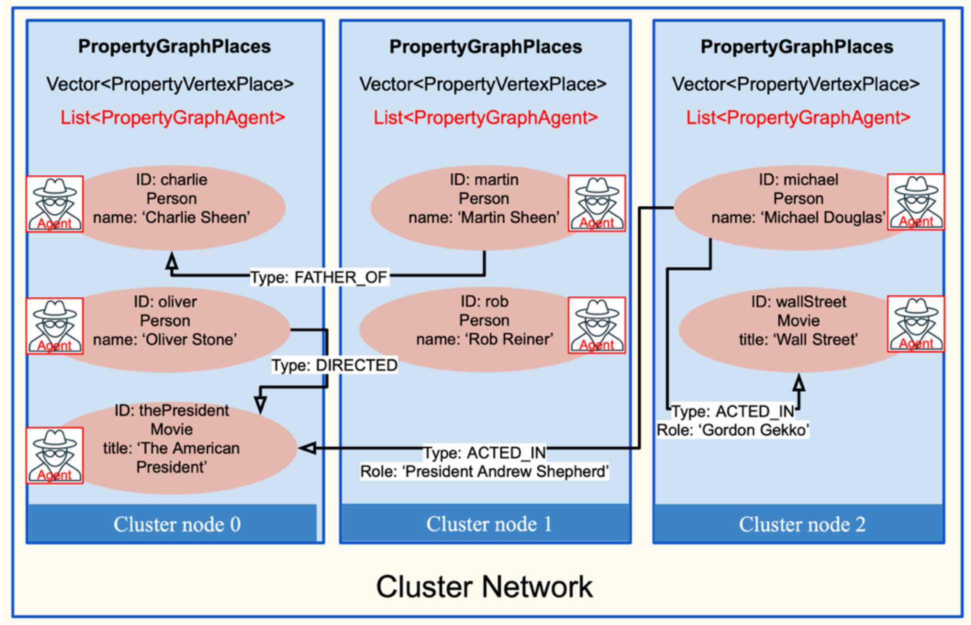
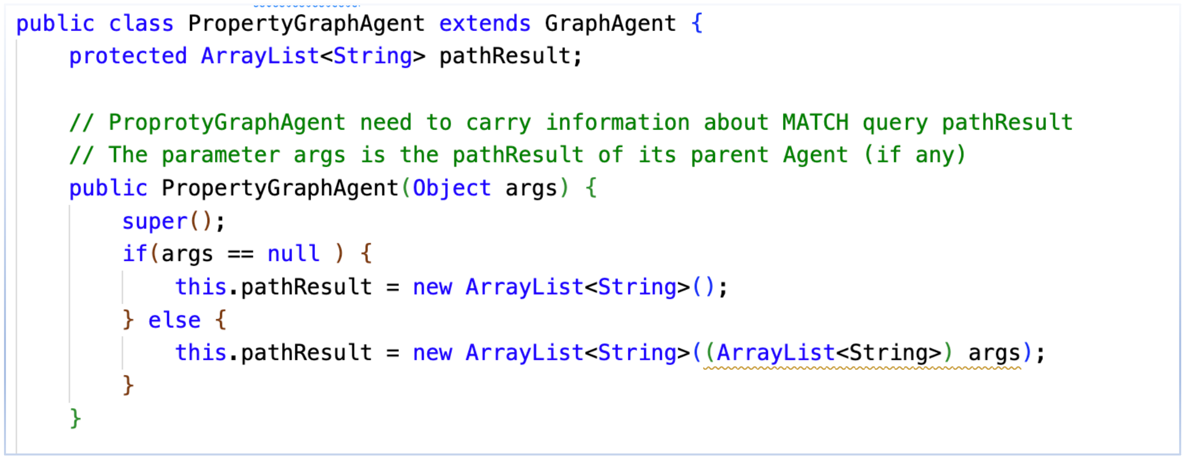


Figure 4.14: A demonstration of agents initialized in a cluster network.

In this project, agents are responsible for traversing the graph and fetching results for the MATCH Cypher clause. As demonstrated in Figure 4.14, an Agents instances will be initialized on each computing node in the system, and each contains a bag of PropertyGraphAgent instances. In the enhanced system, upon initialization, one PropertyGraphAgent instance will reside in each place. The PropertyGraphAgent class extends the Agent class and introduces a new attribute called pathResult, as presented in Figure 4.15. In graph traversal operations, agents often need to navigate through the graph and collect information about the paths they traverse. The pathResult attribute serves as a container to store these traversal results. The constructor of PropertyGraphAgent takes an argument, which is expected to be the pathResult of its parent agent, if available. Upon instantiation, the constructor checks if the argument's content is null. If it is null, then there is no parent agent and the current PropertyGraphAgent instance initializes pathResult as an empty list. Otherwise, the current PropertyGraphAgent instance has a parent and thus initializes its pathResult with a copy of the provided argument to carry forward this information. This mechanism ensures that the PropertyGraphAgent instances can inherit and utilize path information from their parent agents, if available, facilitating seamless continuation of path traversal operations within the property graph.



```
public class PropertyGraphAgent extends GraphAgent {
    protected ArrayList<String> pathResult;

    // ProprotyGraphAgent need to carry information about MATCH query pathResult
    // The parameter args is the pathResult of its parent Agent (if any)
    public PropertyGraphAgent(Object args) {
        super();
        if(args == null ) {
            this.pathResult = new ArrayList<String>();
        } else {
            this.pathResult = new ArrayList<String>((ArrayList<String>) args);
        }
    }
```

Figure 4.15: The code snippet of PropertyGraphAgent's constructor.

4.5.2 Agents' callAll() and manageAll()

In the agent-based graph DB system, callAll() and manageAll() are methods that facilitate communication and coordination among agents. As shown in Figure 4.16, agents will undergo several iterations of the callAll() and manageAll() cycle to retrieve MATCH query results from the graph DB. The number of iterations depends on the number of node patterns in the MATCH query clause. Initially, callAll() is invoked to trigger all agents within the system to verify if the current place where it resides satisfies the node pattern information. If validated, the current place's ID is added to its pathResults list, and neighbor places are determined via the relationship pattern and updated in its nextVertex field, facilitating agents to migrate to its neighbor places. Subsequently, manageAll() is called to coordinate the lifecycle of agents, encompassing the stages of spawn, kill, and migration. In the spawn stage, new agents are created based on the nextVertex information and assigned to specific places within the system. In the kill stage, agents that reside in the places that do not match the current node pattern are terminated. In the migration stage, agents that are allowed to migrate will be moved to the designated places. Migration can occur either locally within the same computing node or remotely between different nodes. This cycle repeats until all MATCH clause criteria are met. The results list is then returned to the main server at the end of the callAll() method.

Figure 4.17 illustrates the process of agents exploring the graph DB when executing the MATCH clause, such as "MATCH (a)-[r1]->(b)-[r2]->(c)". Each place maintains a list to gather all the agents residing in it, along with another list to store neighbor vertex IDs. Initially, one agent resides in each place, such as Agent1 in Place 1 and Agent2 in Place 2. Upon the first iteration of callAll(), Places 1 and 2 are identified as the correct vertices according to node pattern 'a'. Based on relationship 'r1', both Places 1 and 2 have corresponding relationships with Place 3. Consequently, Agent1 updates Place 1's NextVertex list with Place 3, and Agent2 updates Place 2's list similarly. During the first iteration of manageAll(), new agents are spawned based on NextVertex information. In this stage, only Place 3 is listed, so no additional agents are spawned. Agent1 and Agent2 migrate to Place 3, becoming Agent13 and Agent23, respectively. In the subsequent iteration of callAll(), Place 3 is confirmed as the correct vertex for node pattern 'b'. Considering relationship 'r2', Places 4, 5, and 6 are identified as corresponding neighbor vertices to Place 3. Agent13 or Agent23 updates Place 3's NextVertex list with these places. During the subsequent manageAll() iteration, new agents are spawned for each Agent13 and Agent23 and migrated to new places, with six agents in total, labeled Agent134, Agent234, Agent135, Agent235, Agent136, and Agent236. The third iteration of callAll() verifies if Places 4, 5, and/or 6 satisfy node pattern 'c'. At this stage, all MATCH clause criteria are fulfilled, and the results lists carried by each agent are returned to the main server. During the third iteration of manageAll(), no new agents are spawned or migrated; instead, all existing agents are terminated.

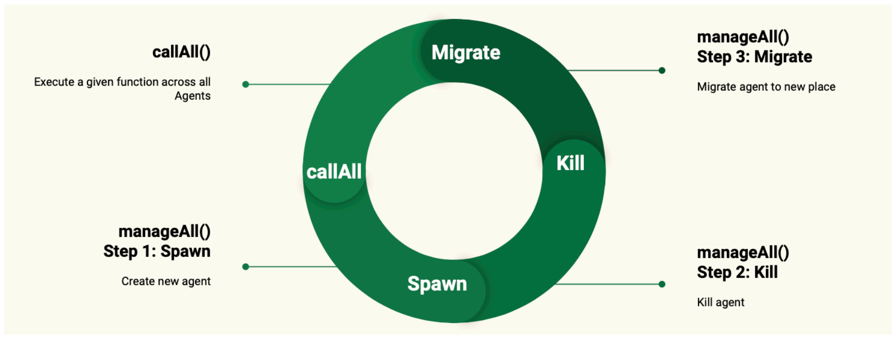


Figure 4.16: The iteration cycle of callAll() and manageAll().

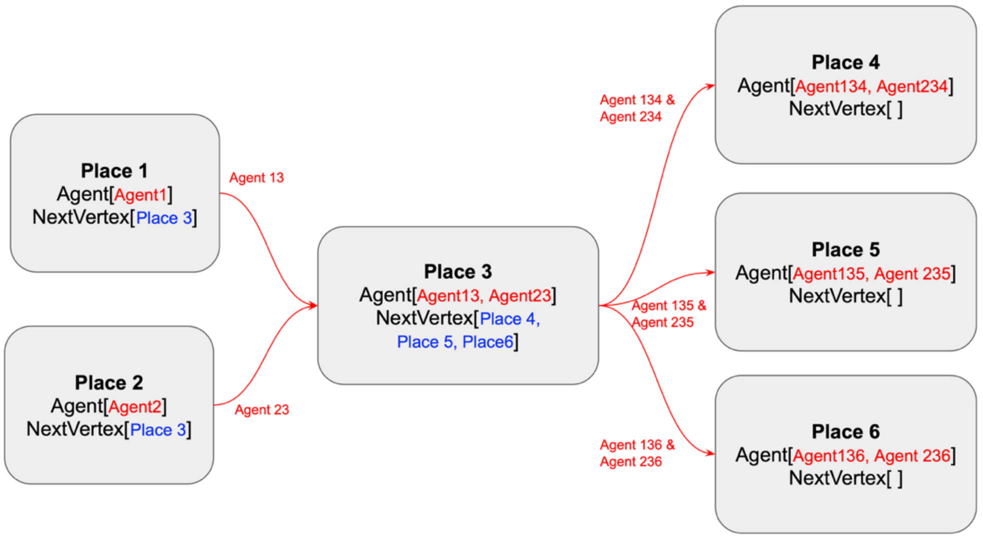


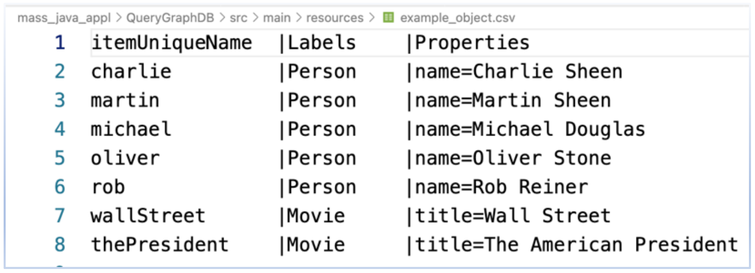
Figure 4.17: An example of agents traversing the enhanced agent-based graph DB.

5. Evaluation

This section aims to verify the functionality of the re-engineered system from the viewpoint of three key aspects: 1) the construction of a property graph from CSV files within a distributed environment; 2) the ability to update the graph structure using CREATE Cypher clauses; and 3) the utilization of agents to access and traverse the graph, leveraging the MATCH Cypher clause to fulfill query requirements. The dataset in the test mimics the film industry.

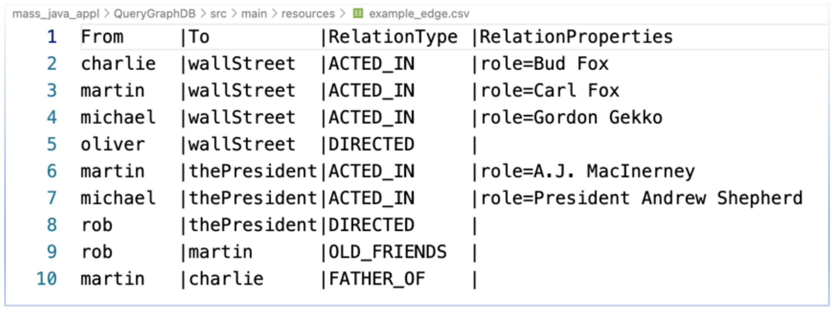
5.1 Verification of building graph from CSV files

Figure 5.1 provides an example of a CSV file containing vertex information, while Figure 5.2 illustrates a CSV file containing relationship information. Utilizing data from these two files, a graph was constructed and partially visualized in Figure 5.3. This graph spans across three server nodes and comprises seven vertices. The vertex IDs, node labels, node properties, and relationship direction, types, and properties have been validated to align accurately with the corresponding data in the CSV files. In this way, the re-engineered graph DB system has demonstrated its capability to handle complex data and relationships in accordance with the property graph model.


mass_java_appl > QueryGraphDB > src > main > resources > example_object.csv

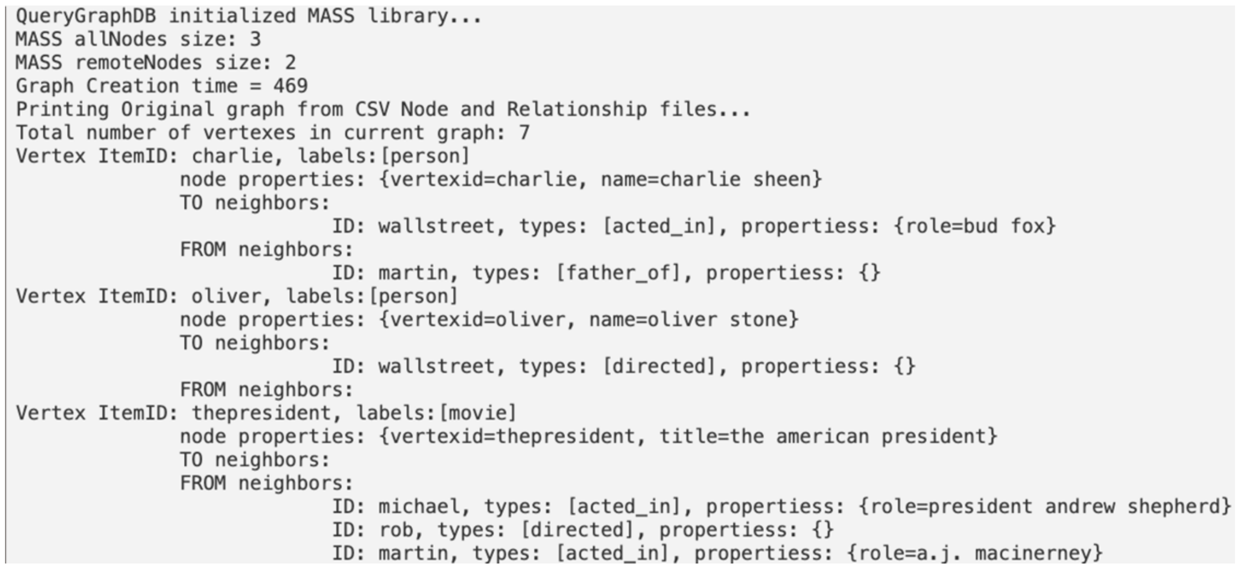
```
itemUniqueName   |Labels     |Properties
charlie          |Person     |name=Charlie Sheen
martin           |Person     |name=Martin Sheen
michael          |Person     |name=Michael Douglas
oliver           |Person     |name=Oliver Stone
rob              |Person     |name=Rob Reiner
wallStreet       |Movie      |title=Wall Street
thePresident     |Movie      |title=The American President
```

Figure 5.1: An example of node csv file.


mass_java_appl > QueryGraphDB > src > main > resources > example_edge.csv

```
From     |To          |RelationType |RelationProperties
charlie  |wallStreet  |ACTED_IN     |role=Bud Fox
martin   |wallStreet  |ACTED_IN     |role=Carl Fox
michael  |wallStreet  |ACTED_IN     |role=Gordon Gekko
oliver   |wallStreet  |DIRECTED     |
martin   |thePresident|ACTED_IN     |role=A.J. MacInerney
michael  |thePresident|ACTED_IN     |role=President Andrew Shepherd
rob      |thePresident|DIRECTED     |
rob      |martin      |OLD_FRIENDS  |
martin   |charlie     |FATHER_OF    |
```

Figure 5.2: An example of relationship csv file.



```
QueryGraphDB initialized MASS library...
MASS allNodes size: 3
MASS remoteNodes size: 2
Graph Creation time = 469
Printing Original graph from CSV Node and Relationship files...
Total number of vertexes in current graph: 7
Vertex ItemID: charlie, labels:[person]
              node properties: {vertexid=charlie, name=charlie sheen}
              TO neighbors:
                           ID: wallstreet, types: [acted_in], propertiess: {role=bud fox}
              FROM neighbors:
                           ID: martin, types: [father_of], propertiess: {}
Vertex ItemID: oliver, labels:[person]
              node properties: {vertexid=oliver, name=oliver stone}
              TO neighbors:
                           ID: wallstreet, types: [directed], propertiess: {}
              FROM neighbors:
Vertex ItemID: thepresident, labels:[movie]
              node properties: {vertexid=thepresident, title=the american president}
              TO neighbors:
              FROM neighbors:
                           ID: michael, types: [acted_in], propertiess: {role=president andrew shepherd}
                           ID: rob, types: [directed], propertiess: {}
                           ID: martin, types: [acted_in], propertiess: {role=a.j. macinerney}
```

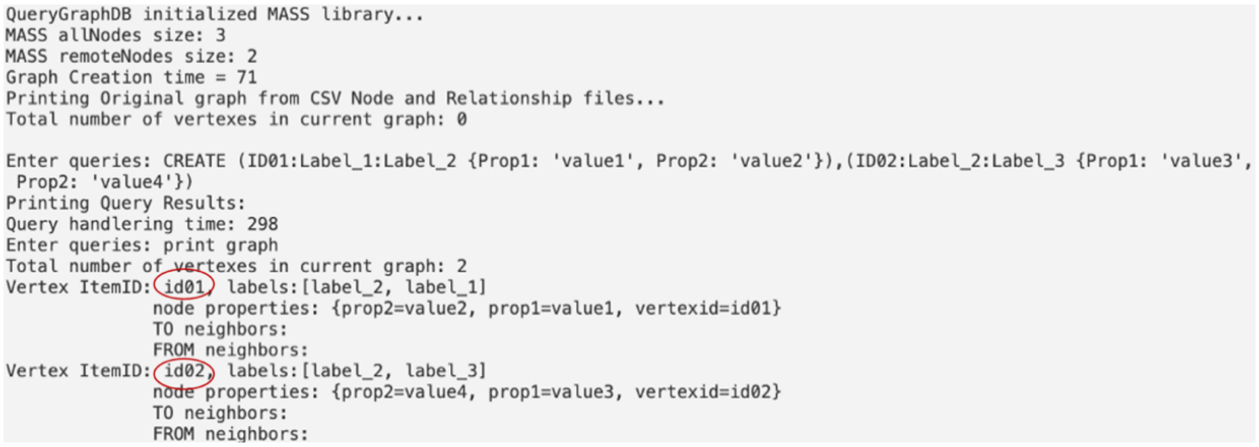
Figure 5.3: The graph constructed from csv files.

5.2 Verification of executing CREATE Cypher clauses

CREATE Cypher clauses serve to generate new nodes or relationships within the graph DB. In this subsection, the functionality and performance of supporting CREATE clauses are evaluated through four test cases, facilitating the update of the graph DB. All the evaluations are conducted within a clustered environment comprising three server nodes.

5.2.1 Test case 1: Creating two nodes with labels and properties in a single clause.

This test case aims to evaluate the functionality of creating multiple nodes in a single CREATE clause, ensuring that the graph DB can be updated properly. Before executing the CREATE query, an empty graph is built across the distributed environment. The resultant graph following the execution of this query clause is depicted in Figure 5.4. Upon executing the aforementioned query, two nodes with ID as id01 and id02 were instantiated and integrated into the cluster network, each endowed with its unique identifier, labels, and properties. Subsequent validation confirms the accuracy of these attributes, aligning perfectly with the query's specifications.



```
QueryGraphDB initialized MASS library...
MASS allNodes size: 3
MASS remoteNodes size: 2
Graph Creation time = 71
Printing Original graph from CSV Node and Relationship files...
Total number of vertexes in current graph: 0

Enter queries: CREATE (ID01:Label_1:Label_2 {Prop1: 'value1', Prop2: 'value2'}),(ID02:Label_2:Label_3 {Prop1: 'value3',
 Prop2: 'value4'})
Printing Query Results:
Query handlering time: 298
Enter queries: print graph
Total number of vertexes in current graph: 2
Vertex ItemID: id01, labels:[label_2, label_1]
              node properties: {prop2=value2, prop1=value1, vertexid=id01}
              TO neighbors:
              FROM neighbors:
Vertex ItemID: id02, labels:[label_2, label_3]
              node properties: {prop2=value4, prop1=value3, vertexid=id02}
              TO neighbors:
              FROM neighbors:
```

Figure 5.4: The output of the CREATE clause test case 1.

5.2.2 Test case 2: Establishing a relationship between two existing nodes.

This test case aims to assess the functionality of updating relationships between existing nodes using the CREATE Cypher clause. This query clause was executed sequentially after the previous CREATE clause. Upon execution, a relationship was built from Vertex id01 directed to Vertex id02. Relationship information is added to the 'toRelationship' attribute of Vertex id01, indicating an outgoing relationship, and meanwhile it is also added to the 'fromRelationship' attribute of Vertex id02, indicating an incoming relationship. The relationship attributes include neighbor ID, relationship type ‘ACTED_IN’, and properties (role=’nothing’). The updated graph is depicted in Figure 5.5, validating the correct execution of this CREATE clause.

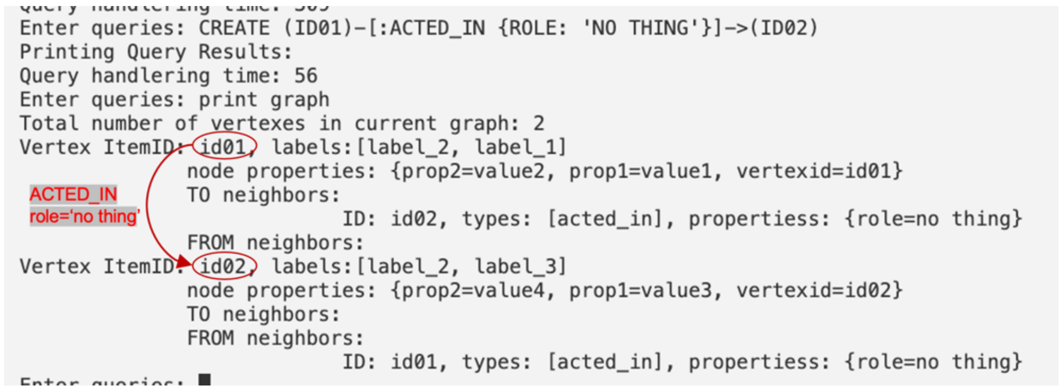

```
Enter queries: CREATE (ID01)-[:ACTED_IN {ROLE: 'NO THING'}]->(ID02)
Printing Query Results:
Query handlering time: 56
Enter queries: print graph
Total number of vertexes in current graph: 2
Vertex ItemID: id01, labels:[label_2, label_1]
                      node properties: {prop2=value2, prop1=value1, vertexid=id01}
                      TO neighbors:
                                    ID: id02, types: [acted_in], propertiess: {role=no thing}
                      FROM neighbors:
Vertex ItemID: id02, labels:[label_2, label_3]
                      node properties: {prop2=value4, prop1=value3, vertexid=id02}
                      TO neighbors:
                      FROM neighbors:
                                    ID: id01, types: [acted_in], propertiess: {role=no thing}
```

Figure 5.5: The output of the CREATE clause test case 2.

5.2.3 Test case 3: Creating multiple nodes and relationships with one path pattern.

Figure 5.6 validates the graph constructed with the "CREATE (a)-[:KNOWS]->(b)-[:KNOWS]->(c)<-[:KNOWS]-(d)" clause. Vertex 'a' has a relationship to Vertex 'b' with the relationship type of 'KNOWS'. Similarly, Vertex 'b' and Vertex 'd' have relationships to Vertex 'c' with the relationship type of 'KNOWS'.

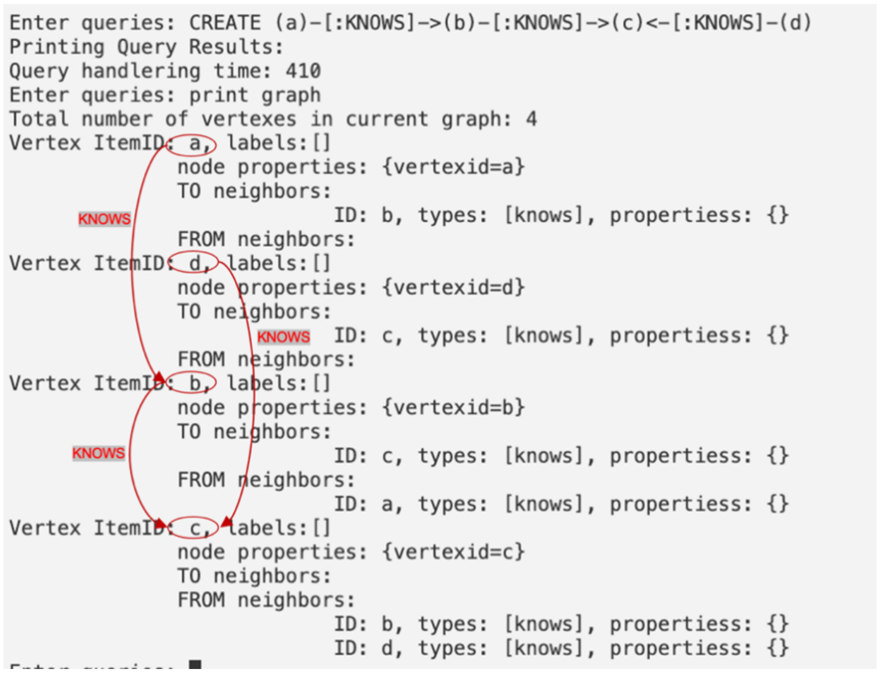

```
Enter queries: CREATE (a)-[:KNOWS]->(b)-[:KNOWS]->(c)<-[:KNOWS]-(d)
Printing Query Results:
Query handlering time: 410
Enter queries: print graph
Total number of vertexes in current graph: 4
Vertex ItemID: a, labels:[]
                      node properties: {vertexid=a}
                      TO neighbors:
                                    ID: b, types: [knows], propertiess: {}
                      FROM neighbors:
Vertex ItemID: d, labels:[]
                      node properties: {vertexid=d}
                      TO neighbors:
                                    ID: c, types: [knows], propertiess: {}
                      FROM neighbors:
Vertex ItemID: b, labels:[]
                      node properties: {vertexid=b}
                      TO neighbors:
                                    ID: c, types: [knows], propertiess: {}
                      FROM neighbors:
                                    ID: a, types: [knows], propertiess: {}
Vertex ItemID: c, labels:[]
                      node properties: {vertexid=c}
                      TO neighbors:
                      FROM neighbors:
                                    ID: b, types: [knows], propertiess: {}
                                    ID: d, types: [knows], propertiess: {}
```

Figure 5.6: The output of the CREATE clause test case 3.

5.2.4 Test case 4: Creating nodes and relationships with multiple path patterns.

A graph as shown in Figure 5.3 can be created using a complex CREATE clause as shown in Appendix D1.

5.3 Verification of executing MATCH Cypher clauses

MATCH Cypher clauses are designed not only to match nodes or relationships but also to fetch graph traversing results. In this subsection, the functionality and performance of supporting MATCH clauses are assessed through seven test cases, which enable the access and traversal of the graph constructed in Section 5.1 and depicted in Figure 5.3. For enhanced visualization, a corresponding graphical representation is provided in Figure 5.7. All evaluations are conducted within a cluster network comprising three computing nodes.

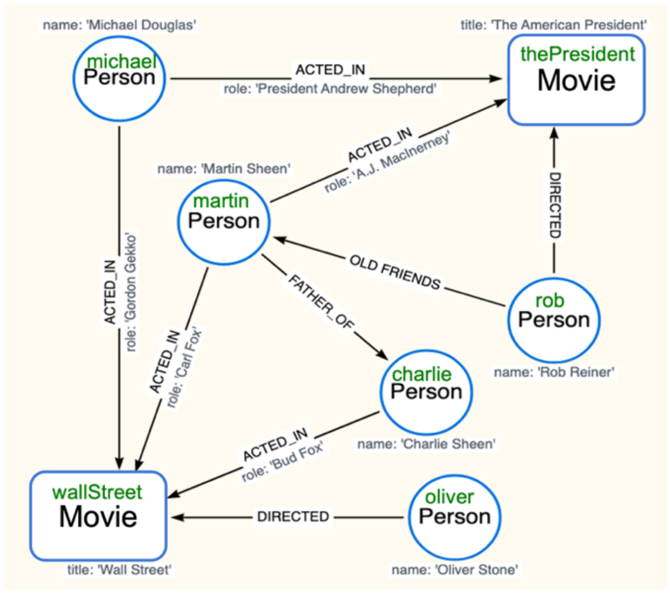
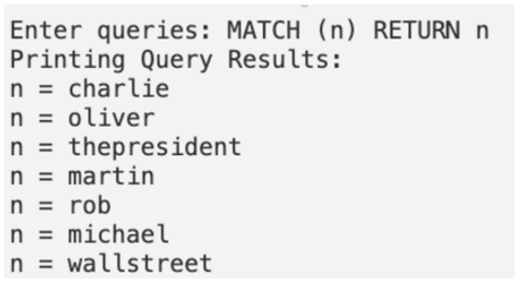


Figure 5.7: Graphical representation of the graph constructed from csv files in Section 5.1.

5.3.1 Test case 1: Matching all nodes in a graph.

The ‘MATCH (n)’ clause involves executing a MATCH query to identify and retrieve every node present within the graph structure. This test assesses the ability of the MATCH Cypher clause to comprehensively traverse the entire graph, ensuring that no node is omitted from the results. In the graph shown in Figure 5.7, there are seven vertices. The output of this MATCH clause is displayed in Figure 5.8, with seven nodes retrieved from the graph, which confirms the accuracy of the execution of this MATCH clause.

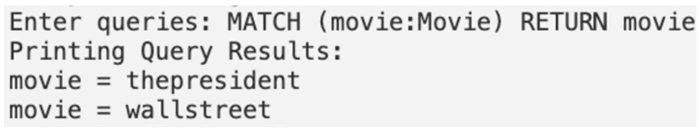


```
Enter queries: MATCH (n) RETURN n
Printing Query Results:
n = charlie
n = oliver
n = thepresident
n = martin
n = rob
n = michael
n = wallstreet
```

Figure 5.8: The Output of the MATCH clause test case 1.

5.3.2 Test case 2: Matching nodes with a label.

This test case is to retrieve a node from the graph with a label of ‘Movie’. It evaluates the capability of the MATCH Cypher clause to accurately identify vertices that satisfy the label condition. The output of executing this MATCH clause is shown in Figure 5.9. After retrieving the vertices with the IDs of 'thepresident' and 'wallstreet' from the graph, the output is confirmed to be consistent with the graph.

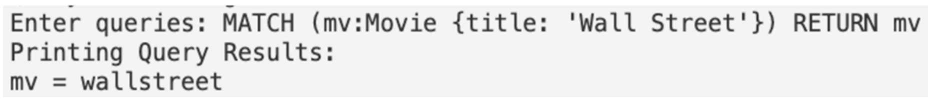


```
Enter queries: MATCH (movie:Movie) RETURN movie
Printing Query Results:
movie = thepresident
movie = wallstreet
```

Figure 5.9: The output of the MATCH clause test case 2.

5.3.3 Test case 3: Matching nodes with a label and a property.

Test Case 3 involves matching nodes within the graph based on both their label and a specific property, which evaluates the capability of the MATCH Cypher clause to accurately identify vertices that satisfy both the specified label and property conditions. In the graph, there is only one vertex with the ID of ‘wallStreet’ that has the label of ‘Movie’ and the property of ‘title = Wall Street’. The output demonstrated in Figure 5.10 is accurate.

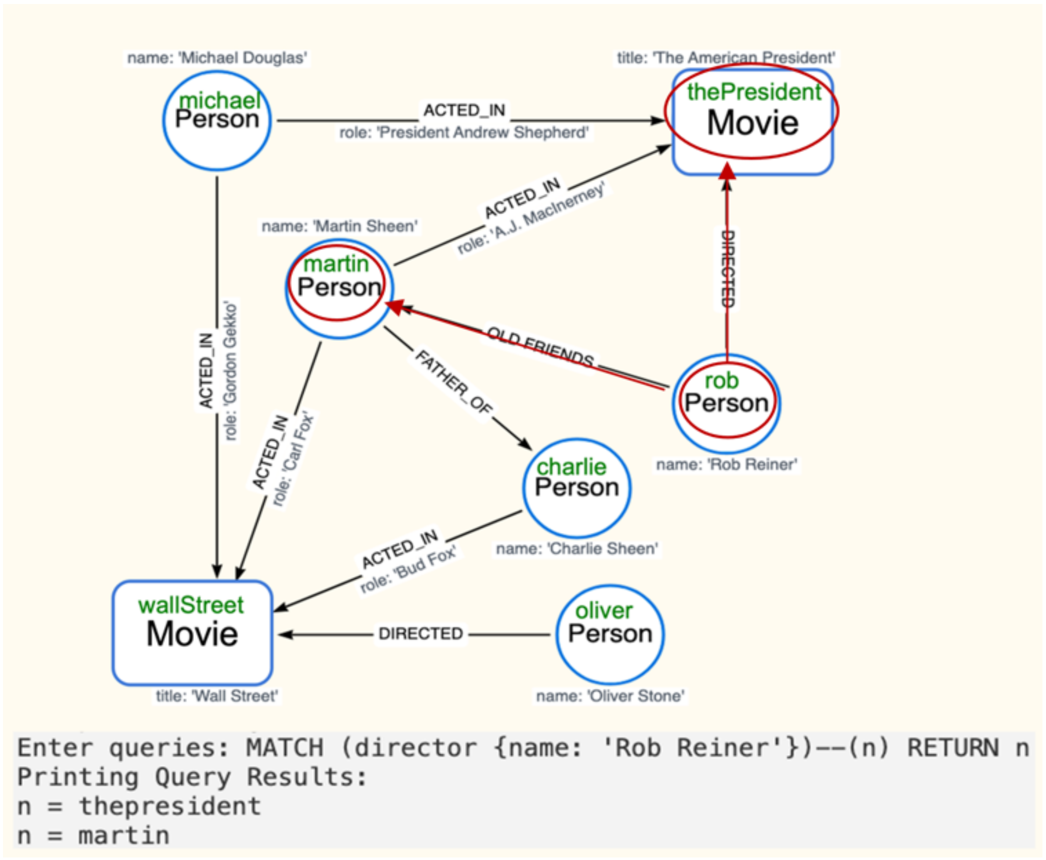


```
Enter queries: MATCH (mv:Movie {title: 'Wall Street'}) RETURN mv
Printing Query Results:
mv = wallstreet
```

Figure 5.10: The output of the MATCH clause test case 3.

5.3.4 Test case 4: Matching all nodes with a relationship to a specific node.

Test Case 4 targets all nodes that are connected to a node labeled ‘director’ with the property ‘name’ as ‘Rob Reiner’. This test assesses the capability of the MATCH clause to traverse relationships effectively and retrieve nodes connected to a specified starting node. As depicted in Figure 5.11, the vertex labeled 'director' with the name 'Rob Reiner' is uniquely identified as 'rob' in the graph. Additionally, two vertices, identified by the IDs 'thePresident' and 'martin', are connected to 'rob'. The output of the MATCH clause confirms the presence of these two vertices ('thePresident' and 'martin'), thus validating the execution of this test case.

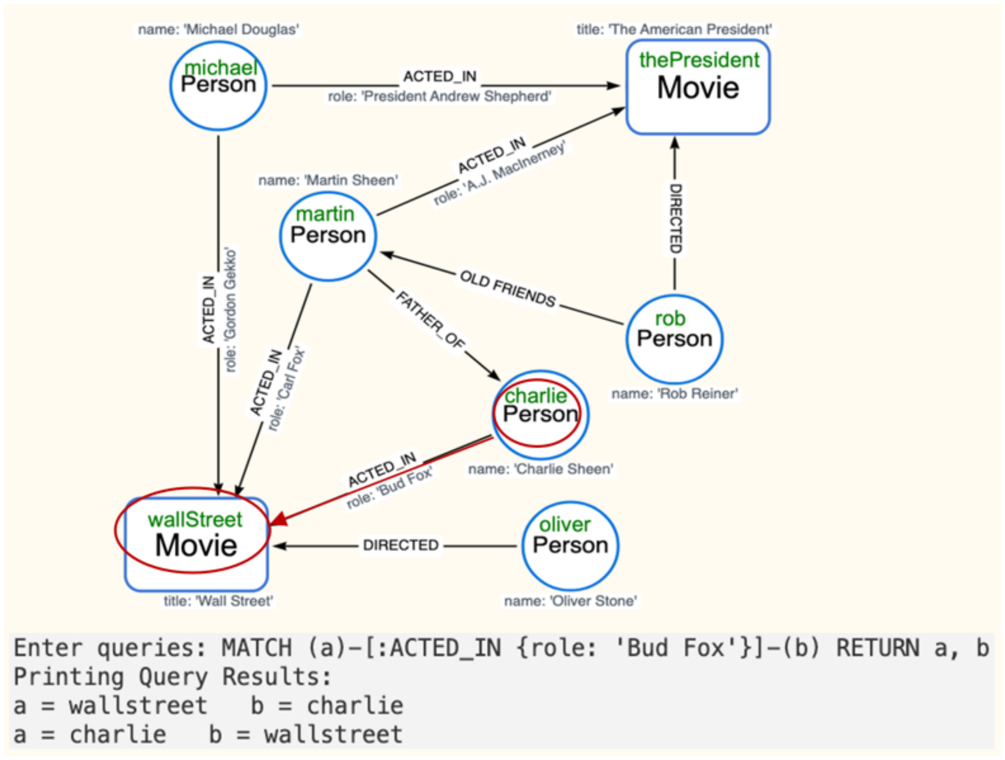


```
Enter queries: MATCH (director {name: 'Rob Reiner'})--(n) RETURN n
Printing Query Results:
n = thepresident
n = martin
```

Figure 5.11: The output of the MATCH clause test case 4 with graphical illustration.

5.3.5 Test case 5: Matching nodes with specific relationship type and property.

Test Case 5 focuses on matching nodes in a graph based on a specific relationship type 'ACTED_IN' and a property of 'role = Bud Fox'. This test assesses whether the query successfully filters nodes based on both relationship type and property criteria, ensuring that only relevant nodes are included in the output. The output of this test case is demonstrated in Figure 5.12. In this graph, ‘charlie’ and ‘wallStreet’ are two vertices connected via the specified relationship. Since the direction of the relationship is not specified in the query, either variable 'a' or 'b' can be assigned to represent 'charlie', while the other represents 'wallStreet'.

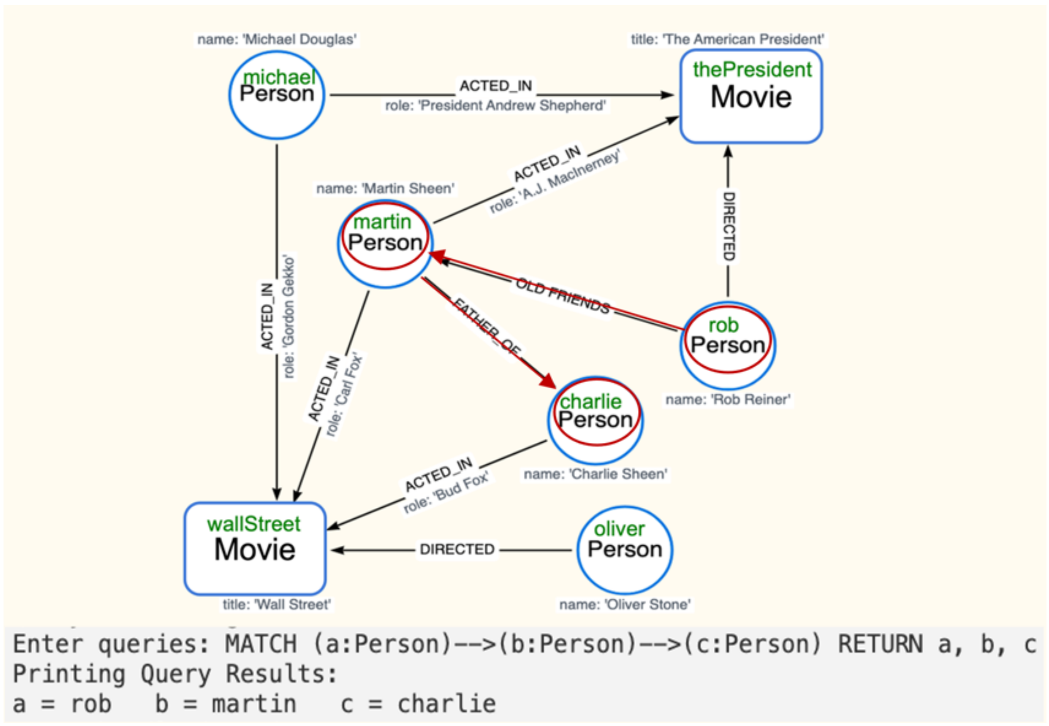


```
Enter queries: MATCH (a)-[:ACTED_IN {role: 'Bud Fox'}]-(b) RETURN a, b
Printing Query Results:
a = wallstreet    b = charlie
a = charlie    b = wallstreet
```

Figure 5.12: The output of the MATCH clause test case 5 with graphical illustration.

5.3.6 Test case 6: Matching nodes along a path with simple relationships.

Test Case 6 involves matching nodes along a path with simple relationships in a graph. The query in the test seeks to identify sequences of nodes labeled as 'Person' connected by simple relationships, where each node is connected to the next one. This test evaluates the system’s ability to use agents to traverse paths within the graph by following relationships between nodes. The output of this test case is demonstrated in Figure 5.13. There is one path identified with the input query, which is validated in accordance with the graph.

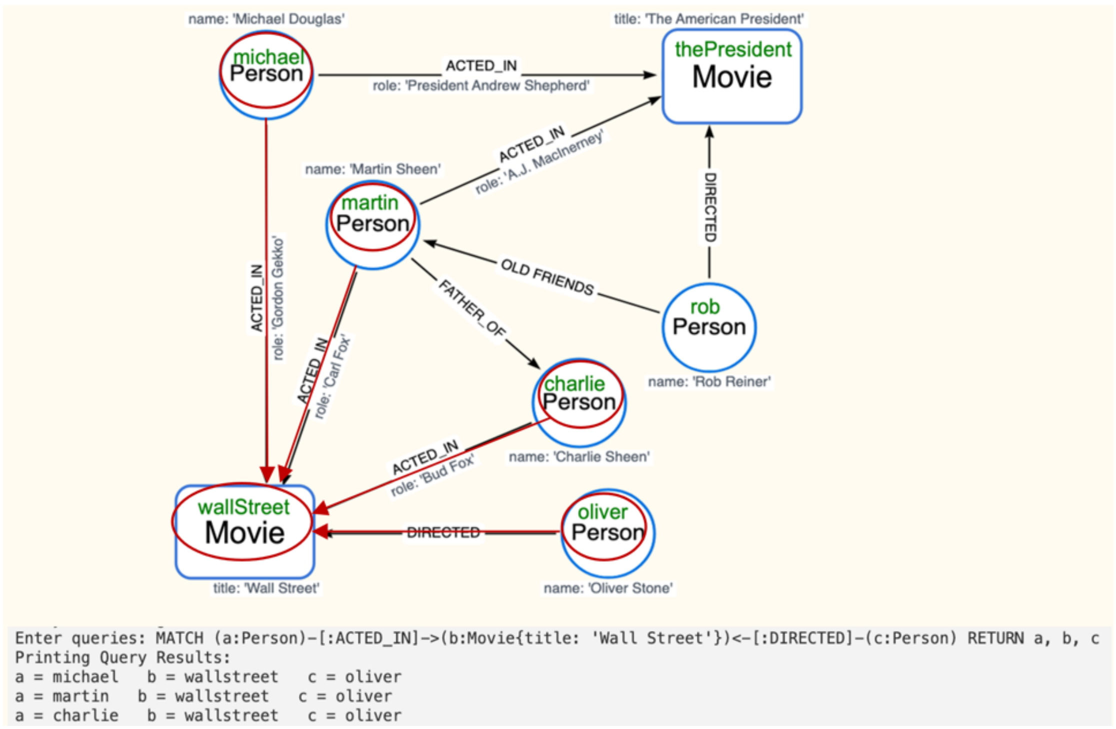


```
Enter queries: MATCH (a:Person)-->(b:Person)-->(c:Person) RETURN a, b, c
Printing Query Results:
a = rob   b = martin   c = charlie
```

Figure 5.13: The output of the MATCH clause test case 6 with graphical illustration.

5.3.7 Test case 7: Matching nodes along a path with complex relationships.

Test Case 7 involves matching nodes along a path with complex relationships in a graph. The tested query clause seeks to identify sequences of nodes labeled as 'Person' connected by complex relationships, involving both directed and property-constrained relationships. Specifically, the query aims to find nodes 'a' and 'c' that are connected to a node 'b' representing a movie titled 'Wall Street'. Node 'a' must be connected to 'b' via an 'ACTED_IN' relationship, while 'c' must be connected to 'b' via a 'DIRECTED' relationship. The output is shown in Figure 5.14, with three vertices identified to ‘ACTED_IN’ the ‘Wall Street’ movie and one vertex identified to “DIRECTED” that movie.



```
Enter queries: MATCH (a:Person)-[:ACTED_IN]->(b:Movie{title: 'Wall Street'})<-[:DIRECTED]-(c:Person) RETURN a, b, c
Printing Query Results:
a = michael   b = wallstreet   c = oliver
a = martin   b = wallstreet   c = oliver
a = charlie   b = wallstreet   c = oliver
```

Figure 5.14: The output of the MATCH clause test case 7 with graphical illustration.

5.4 Summary of evaluation

Through the comprehensive evaluation of all test cases, we successfully validated the functionality of the enhanced graph DB system. This validation extends to its effectiveness in managing complex datasets within a property graph model and its support for Cypher CREATE and MATCH clauses within an in-memory distributed graph structure. Notably, the system's capability to traverse the graph using agents to retrieve desired results for MATCH clause execution underscores its ability to seamlessly handle complex queries.

6. Conclusion

In this project, we embarked on a thorough exploration aimed at refining and augmenting the capabilities of our existing agent-based graph DB system. Below is the summary of the work done in this project: 1) a comprehensive review was conducted, encompassing both previous endeavors and related research, to identify any limitations and propose corresponding remedies; 2) integrating valuable insights from this review into our project's scope, we proceeded to re-engineer the system to accommodate the handling of complex vertex and relationship properties in alignment with the property graph model; 3) we further enhanced the functionality of the system by implementing a Cypher processing pipeline, thereby rendering it compatible with the Cypher query language; 4) this enhancement was seamlessly integrated into the existing in-memory distributed graph database structure; and 5) leveraging PropertyGraphAgent, we enabled efficient traversal of the graph database to fetch results when executing MATCH clauses.

Despite the significant advancements achieved in this project, two limitations persist, necessitating the following improvements. Firstly, while the implementation successfully incorporates CREATE and MATCH clauses, there remains a need to extend support to other essential clauses such as DELETE, which facilitates the removal of nodes or relationships, and RETURN, enabling the retrieval of results in various formats. Moreover, to enhance the power of the MATCH clause, the implementation and integration of WHERE and WITH clauses are necessary. Secondly, the current dataset utilized for testing purposes is relatively small, which poses a limitation in accurately gauging the system's performance. To address this, future work entails procuring and employing larger datasets to conduct thorough benchmarking exercises. Initiatives like Michelle Dea's Benchmark program, aimed at testing the system with large real-world datasets and comparing its performance with other graph DB systems like ArangoDB and Neo4j, will be instrumental in validating and refining the system's efficacy and scalability.

Bibliography

[1] I. Robinson, James. Webber, and E. Eifrem, *Graph databases*, First edition. Sebastopol, CA: O'Reilly Media, Inc., 2013.

[2] Sonal. Raj, *Neo4j High Performance : design, build, and administer scalable graph database systems for your applications using Neo4j*. in Community Experience Distilled. Packt Publishing, 2015.

[3] D. Eadline, "Hadoop and Spark fundamentals : LiveLessons," Pearson, 2018.

[4] A. Li and M. Fukuda, "Agent-Based Parallelization of a Multi-Dimensional Semantic Database Model," in *IRI*, IEEE, 2023, pp. 64–69. doi: 10.1109/IRI58017.2023.00019.

[5] Y. Hong and M. Fukuda, "Pipelining Graph Construction and Agent-based Computation over Distributed Memory," in *2022 IEEE International Conference on Big Data (Big Data)*, 2022, pp. 4616–4624. doi: 10.1109/BigData55660.2022.10020903.

[6] V. Mohan, A. Potturi, and M. Fukuda, "Automated agent migration over distributed data structures," in *In Proceedings of the 15th International Conference on Agents and Artificial Intelligence*, 2022.

[7] "MASS Java Manual," Aug. 2016, Accessed: Apr. 16, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Technical%20Manual.pdf

[8] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," in *Big Data*, IEEE, 2020, pp. 2957–2966. doi: 10.1109/BigData50022.2020.9378298.

[9] Brian Luger and Munehiro Fukuda, "Wiki Graph Programming with MASS Java." Accessed: Apr. 14, 2024. [Online]. Available: https://bitbucket.org/mass_library_developers/mass_java_core/wiki/GraphPlaces%20Class%20and%20Cytoscape%20Integration

[10] "Graph Database Concepts." Accessed: Apr. 14, 2024. [Online]. Available: https://neo4j.com/docs/getting-started/appendix/graphdb-

concepts/#:~:text=The%20Neo4j%20property%20graph%20database,node%20and%20a%20target%20node.

[11] Emil Eifrem, “Meet openCypher.” Accessed: Apr. 14, 2024. [Online]. Available: https://neo4j.com/blog/open-cypher-sql-for-graphs/?utm_source=Google&utm_medium=PaidSearch&utm_campaign=Evergreenutm_content=AMS-Search-SEMCE-DSA-None-SEM-SEM-NonABM&utm_term=&utm_adgroup=DSA-use-cases&gad_source=1&gclid=CjwKCAjwoPOwBhAeEiwAJuXRh3FPxkkDEQ-WrFs5zy06iQ65lckn4TPxJiICKa3J9LdaodMgZ7aX3BoCP-sQAvD_BwE

[12] “RadisGraph”, Accessed: Apr. 26, 2024. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/stack-with-enterprise/deprecated-features/graph/

[13] “JanusGraph”, Accessed: Apr. 18, 2024. [Online]. Available: https://janusgraph.org/

[14] D. Fernandes and J. Bernardino, *Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB*. 2018. doi: 10.5220/0006910203730380.

[15] “What is ArangoDB?”, Accessed: Apr. 19, 2024. [Online]. Available: https://docs.arangodb.com/stable/about-arangodb/

[16] “Cypher Manual”, Accessed: Apr. 28, 2024. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/introduction/?utm_medium=PaidSearch&utm_source=Google&utm_campaign=Evergreen&utm_adgroup=DSA&utm_content=AMS-Search-SEMCE-DSA-None-SEM-SEM-NonABM&gclid=Cj0KCQjwir2xBhC_ARIsAMTXk85_2bL9xuwQz86a_Z-TkrJ89idlrfBtHpmGTLAuaMzMLfftsPMjtEkaAhwcEALw_wcB

[17] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

Appendix A

A1. Code package:

Code package can be downloaded from Bitbucket:

mass_java_appl, QueryGraphDB branch:
https://bitbucket.org/mass_application_developers/mass_java_appl/src/QueryGraphDB/

mass_java_core, QueryGraphDB bracnch:
https://bitbucket.org/mass_library_developers/mass_java_core/src/QueryGraphDB/

For more information of the files and folders containing the code that implemented in this project, it can be found at mass_java_appl, QueryGraphDB branch, QueryGraphDB folder, README.md file.

A2. Build and Run:

1. to make changes to MASS_java_core and rebuild, go to '~/mass_java_appl/QueryGraphDB' folder and then run 'sh build_mass.sh'

2. to rebuild and run the QueryGraphDB application, go to '~/mass_java_appl/QueryGraphDB' folder and then run 'sh build_run.sh'

A3. Tips for running PropertyGraphPlaces and PropertyGraphAgent in remote nodes:

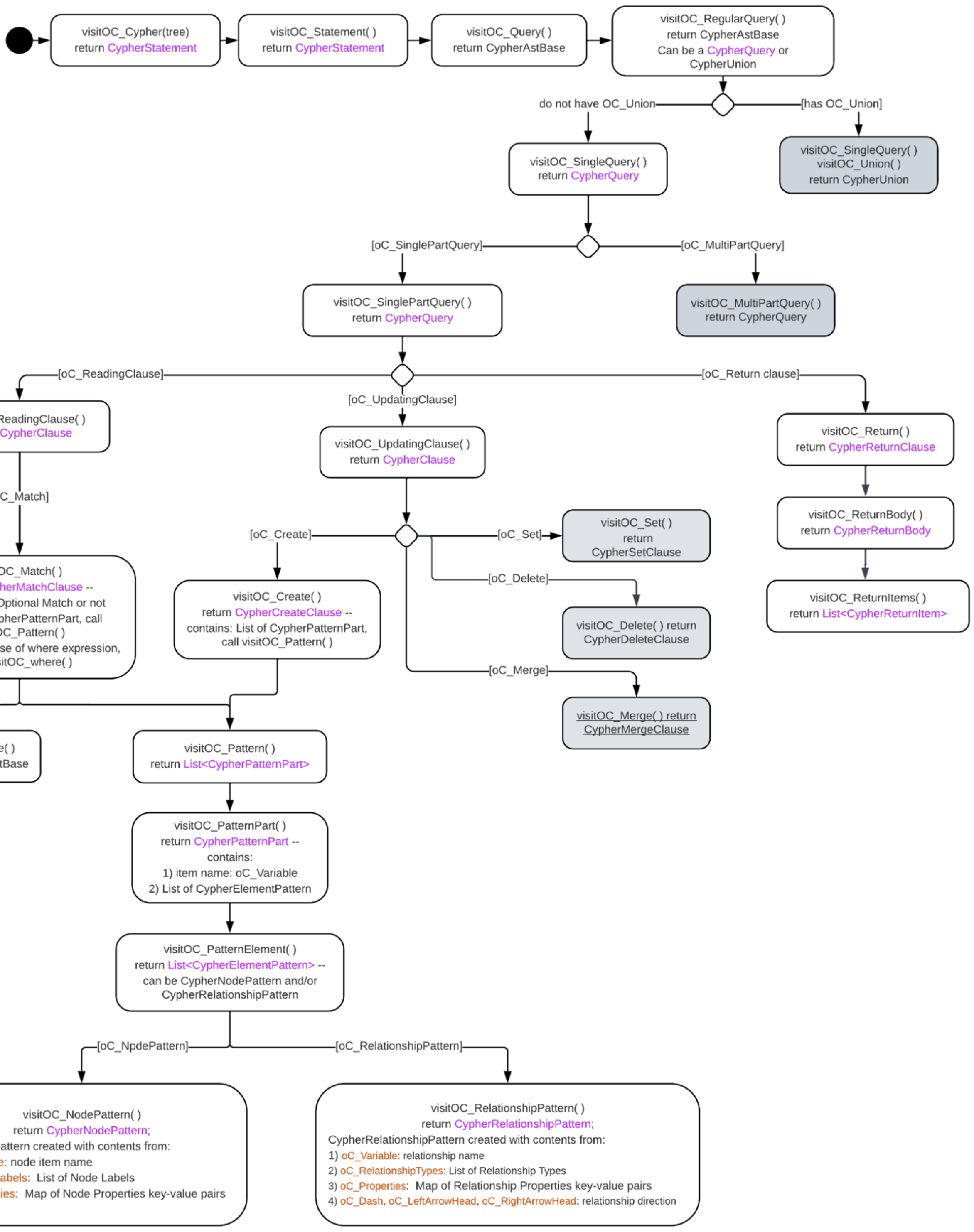
1. Local node communicates with remote nodes via message passing, with code in three classes: Message.java, MThread.java, MProcess.java.

2. To implement a new function in PropertyGraphPlaces, needs to send message to remote nodes. Then update Message.java's message type, MThread.java's status, MProcess's start() function to listen to the message and handle it.

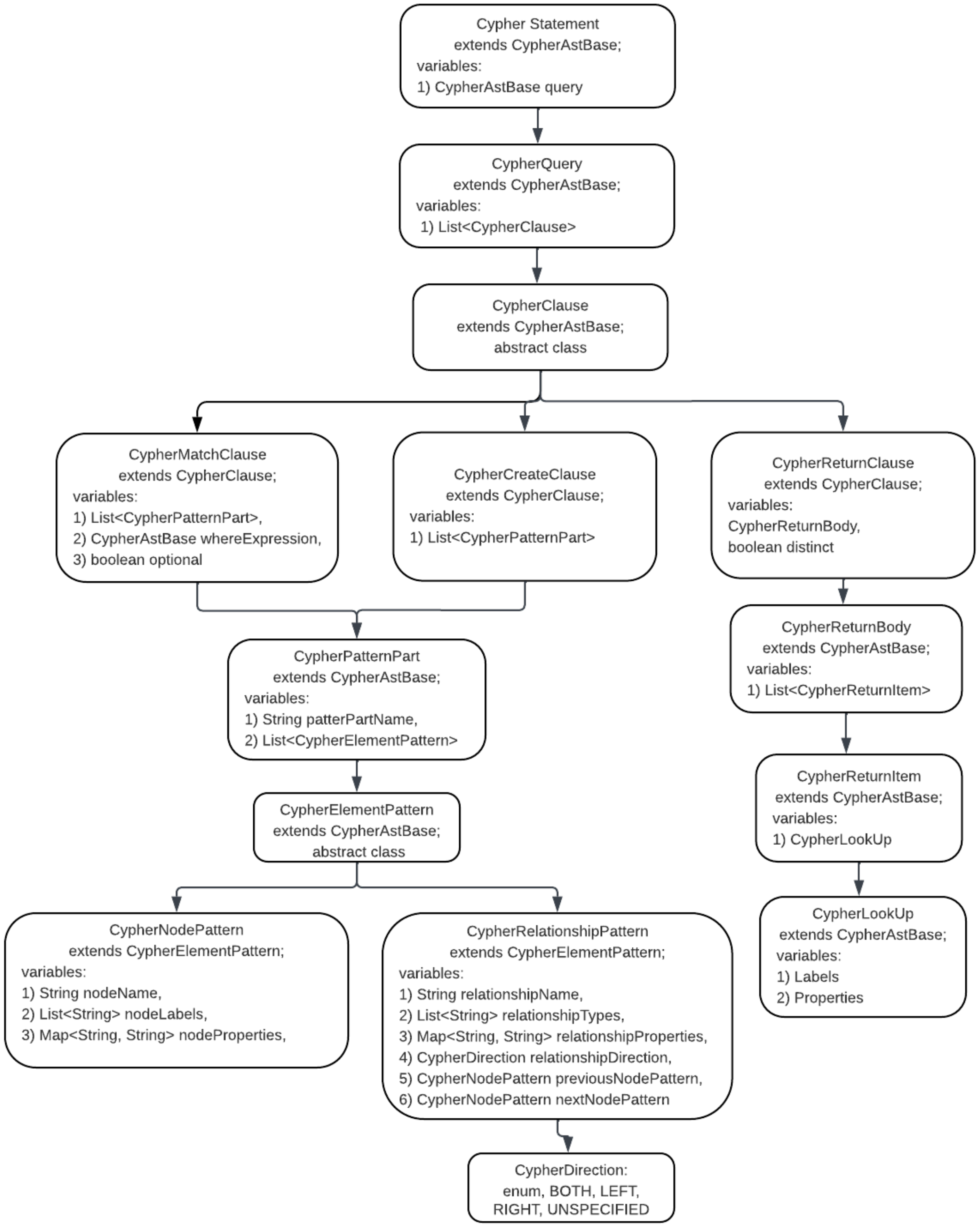
3. If remote nodes cannot handle PropertyGraphPlaces or PropertyGraphAgent properly, go through the code to see if there are any casting issues (may need to cast Places to PropertyGraphPlaces, Place to PropertyVertexPlace, or Agent to PropertyGraphAgent).

Appendix B

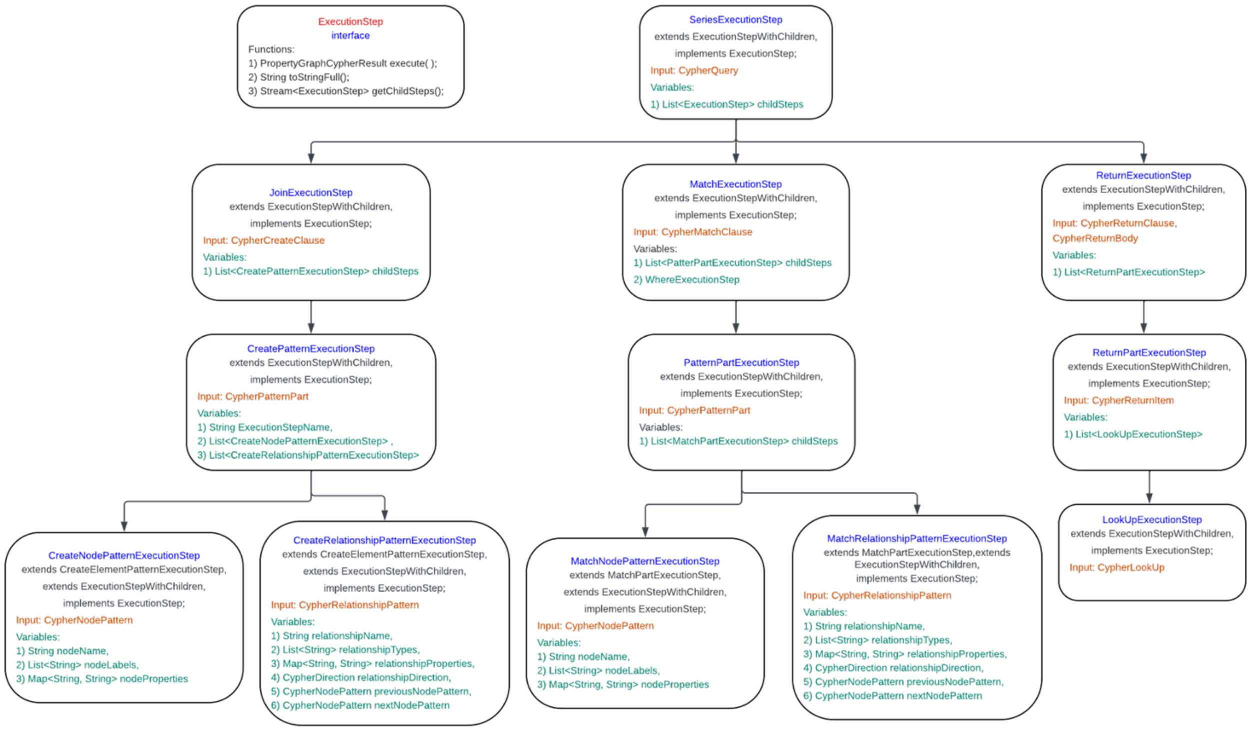
B1. Function flow of PropertyGraphCypherVisitor


visitOC_Cypher(tree) return CypherStatement
visitOC_Statement( ) return CypherStatement
visitOC_Query( ) return CypherAstBase
visitOC_RegularQuery( ) return CypherAstBase Can be a CypherQuery or CypherUnion
do not have OC_Union
[has OC_Union]
visitOC_SingleQuery( ) return CypherQuery
visitOC_SingleQuery( ) visitOC_Union( ) return CypherUnion
[oC_SinglePartQuery]
[oC_MultiPartQuery]
visitOC_SinglePartQuery( ) return CypherQuery
visitOC_MultiPartQuery( ) return CypherQuery
[oC_ReadingClause]
[oC_UpdatingClause]
[oC_Return clause]
visitOC_ReadingClause( ) return CypherClause
visitOC_UpdatingClause( ) return CypherClause
visitOC_Return( ) return CypherReturnClause
visitOC_ReturnBody( ) return CypherReturnBody
visitOC_ReturnItems( ) return List<CypherReturnItem>
[oC_Match]
visitOC_Match( ) return CypherMatchClause -- contains: 1) Optional Match or not 2) List of CypherPatternPart, call visitOC_Pattern( ) 3) CypherAstBase of where expression, call visitOC_where( )
[oC_Create]
[oC_Set]
[oC_Delete]
[oC_Merge]
visitOC_Create( ) return CypherCreateClause -- contains: List of CypherPatternPart, call visitOC_Pattern( )
visitOC_Set( ) return CypherSetClause
visitOC_Delete( ) return CypherDeleteClause
visitOC_Merge( ) return CypherMergeClause
visitOC_where( ) return CypherAstBase
visitOC_Pattern( ) return List<CypherPatternPart>
visitOC_PatternPart( ) return CypherPatternPart -- contains: 1) item name: oC_Variable 2) List of CypherElementPattern
visitOC_PatternElement( ) return List<CypherElementPattern> -- can be CypherNodePattern and/or CypherRelationshipPattern
[oC_NpdePattern]
[oC_RelationshipPattern]
visitOC_NodePattern( ) return CypherNodePattern; CypherNodePattern created with contents from: 1) oC_Variable: node item name 2) oC_NodeLabels: List of Node Labels 3) oC_Properties: Map of Node Properties key-value pairs
visitOC_RelationshipPattern( ) return CypherRelationshipPattern; CypherRelationshipPattern created with contents from: 1) oC_Variable: relationship name 2) oC_RelationshipTypes: List of Relationship Types 3) oC_Properties: Map of Relationship Properties key-value pairs 4) oC_Dash, oC_LeftArrowHead, oC_RightArrowHead: relationship direction

B2. Abstract Syntax Tree (AST)



B3. ExecutionStep tree



ExecutionStep
interface
Functions:
1) PropertyGraphCypherResult execute( );
2) String toStringFull();
3) Stream<ExecutionStep> getChildSteps();
SeriesExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherQuery
Variables:
1) List<ExecutionStep> childSteps
JoinExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherCreateClause
Variables:
1) List<CreatePatternExecutionStep> childSteps
MatchExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherMatchClause
Variables:
1) List<PatterPartExecutionStep> childSteps
2) WhereExecutionStep
ReturnExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherReturnClause,
CypherReturnBody
Variables:
1) List<ReturnPartExecutionStep>
CreatePatternExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherPatternPart
Variables:
1) String ExecutionStepName,
2) List<CreateNodePatternExecutionStep> ,
3) List<CreateRelationshipPatternExecutionStep>
PatternPartExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherPatternPart
Variables:
1) List<MatchPartExecutionStep> childSteps
ReturnPartExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherReturnItem
Variables:
1) List<LookUpExecutionStep>
LookUpExecutionStep
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherLookUp
CreateNodePatternExecutionStep
extends CreateElementPatternExecutionStep,
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherNodePattern
Variables:
1) String nodeName,
2) List<String> nodeLabels,
3) Map<String, String> nodeProperties
CreateRelationshipPatternExecutionStep
extends CreateElementPatternExecutionStep,
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherRelationshipPattern
Variables:
1) String relationshipName,
2) List<String> relationshipTypes,
3) Map<String, String> relationshipProperties,
4) CypherDirection relationshipDirection,
5) CypherNodePattern previousNodePattern,
6) CypherNodePattern nextNodePattern
MatchNodePatternExecutionStep
extends MatchPartExecutionStep,
extends ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherNodePattern
Variables:
1) String nodeName,
2) List<String> nodeLabels,
3) Map<String, String> nodeProperties
MatchRelationshipPatternExecutionStep
extends MatchPartExecutionStep,extends
ExecutionStepWithChildren,
implements ExecutionStep;
Input: CypherRelationshipPattern
Variables:
1) String relationshipName,
2) List<String> relationshipTypes,
3) Map<String, String> relationshipProperties,
4) CypherDirection relationshipDirection,
5) CypherNodePattern previousNodePattern,
6) CypherNodePattern nextNodePattern

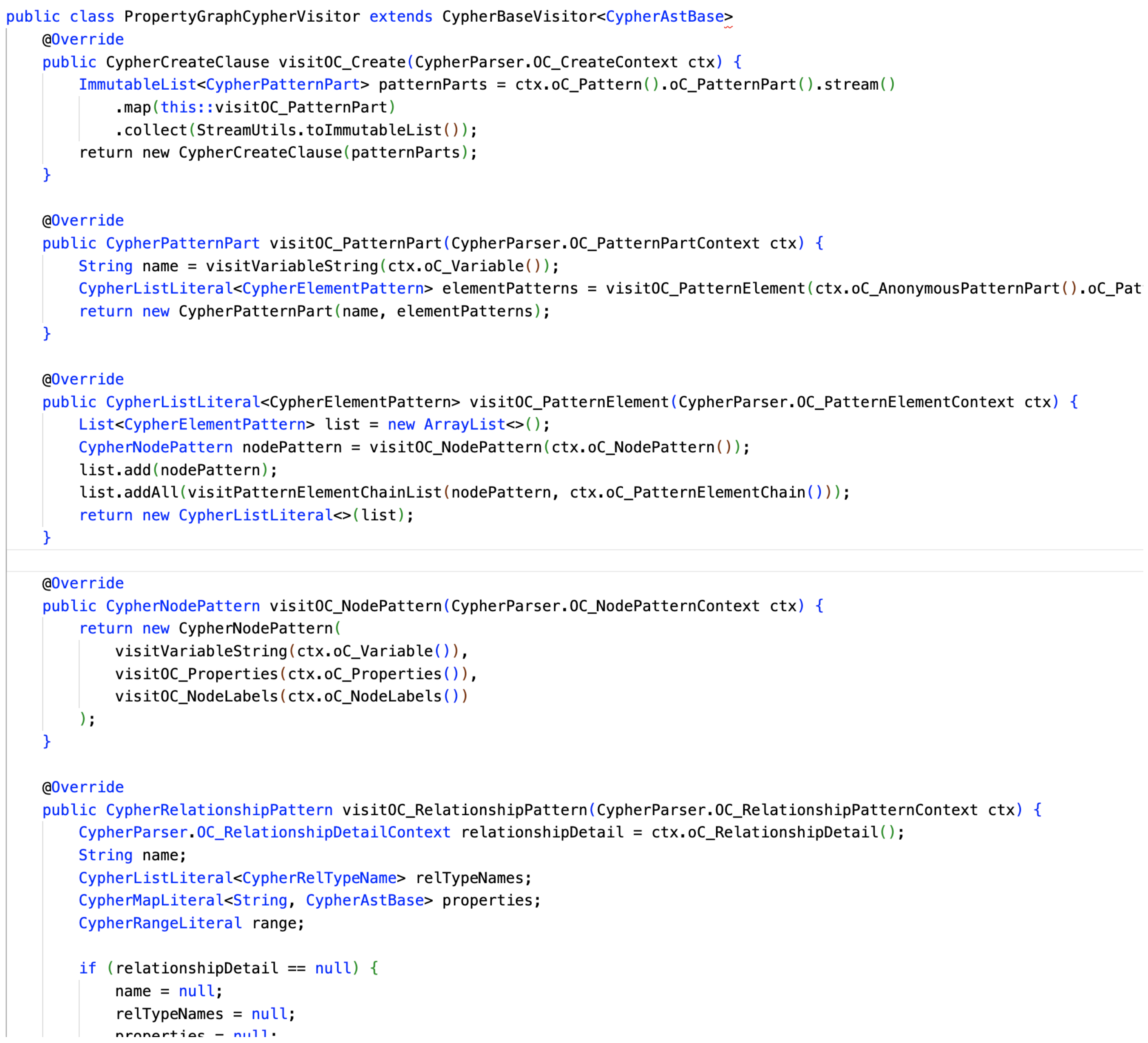
Appendix C

C1: Java code snippet for parsing query string and process it with Cypher Visitor

```java
public CypherStatement parse(String queryString) {
    // create a CharStream that reads from standard input
    CodePointCharStream input = CharStreams.fromString(queryString);
    // create lexer
    CypherLexer lexer = new CypherLexer(input);
    // create a buffer of tokens pulled from the lexer
    // create a parser that feeds off the tokens buffer
    CypherParser parser = new CypherParser(new CommonTokenStream(lexer));
    // begin parsing at oC_Cypher to get parse tree
    CypherParser.OC_CypherContext tree = parser.oC_Cypher();
    return new PropertyGraphCypherVisitor().visitOC_Cypher(tree);
}
```

The parse() method starts by converting the input query string into a CodePointCharStream, which is then processed by a CypherLexer to generate tokens. These tokens are subsequently fed into a CypherParser to construct a parse tree starting from the root rule oC_Cypher. A PropertyGraphCypherVisitor object is then constructed to traverse the parse tree following the process flow outlined in Appendxi B1. This traversal produces and returns a CypherStatement object that represents the AST form of the query.

C2: Java code snippet for the PropertyGraphCypherVisitor class with visitOC_Create() as an example.



```java
public class PropertyGraphCypherVisitor extends CypherBaseVisitor<CypherAstBase>
    @Override
    public CypherCreateClause visitOC_Create(CypherParser.OC_CreateContext ctx) {
        ImmutableList<CypherPatternPart> patternParts = ctx.oC_Pattern().oC_PatternPart().stream()
            .map(this::visitOC_PatternPart)
            .collect(StreamUtils.toImmutableList());
        return new CypherCreateClause(patternParts);
    }

    @Override
    public CypherPatternPart visitOC_PatternPart(CypherParser.OC_PatternPartContext ctx) {
        String name = visitVariableString(ctx.oC_Variable());
        CypherListLiteral<CypherElementPattern> elementPatterns = visitOC_PatternElement(ctx.oC_AnonymousPatternPart().oC_Pat
        return new CypherPatternPart(name, elementPatterns);
    }

    @Override
    public CypherListLiteral<CypherElementPattern> visitOC_PatternElement(CypherParser.OC_PatternElementContext ctx) {
        List<CypherElementPattern> list = new ArrayList<>();
        CypherNodePattern nodePattern = visitOC_NodePattern(ctx.oC_NodePattern());
        list.add(nodePattern);
        list.addAll(visitPatternElementChainList(nodePattern, ctx.oC_PatternElementChain()));
        return new CypherListLiteral<>(list);
    }

    @Override
    public CypherNodePattern visitOC_NodePattern(CypherParser.OC_NodePatternContext ctx) {
        return new CypherNodePattern(
            visitVariableString(ctx.oC_Variable()),
            visitOC_Properties(ctx.oC_Properties()),
            visitOC_NodeLabels(ctx.oC_NodeLabels())
        );
    }

    @Override
    public CypherRelationshipPattern visitOC_RelationshipPattern(CypherParser.OC_RelationshipPatternContext ctx) {
        CypherParser.OC_RelationshipDetailContext relationshipDetail = ctx.oC_RelationshipDetail();
        String name;
        CypherListLiteral<CypherRelTypeName> relTypeNames;
        CypherMapLiteral<String, CypherAstBase> properties;
        CypherRangeLiteral range;

        if (relationshipDetail == null) {
            name = null;
            relTypeNames = null;
            properties = null;
```

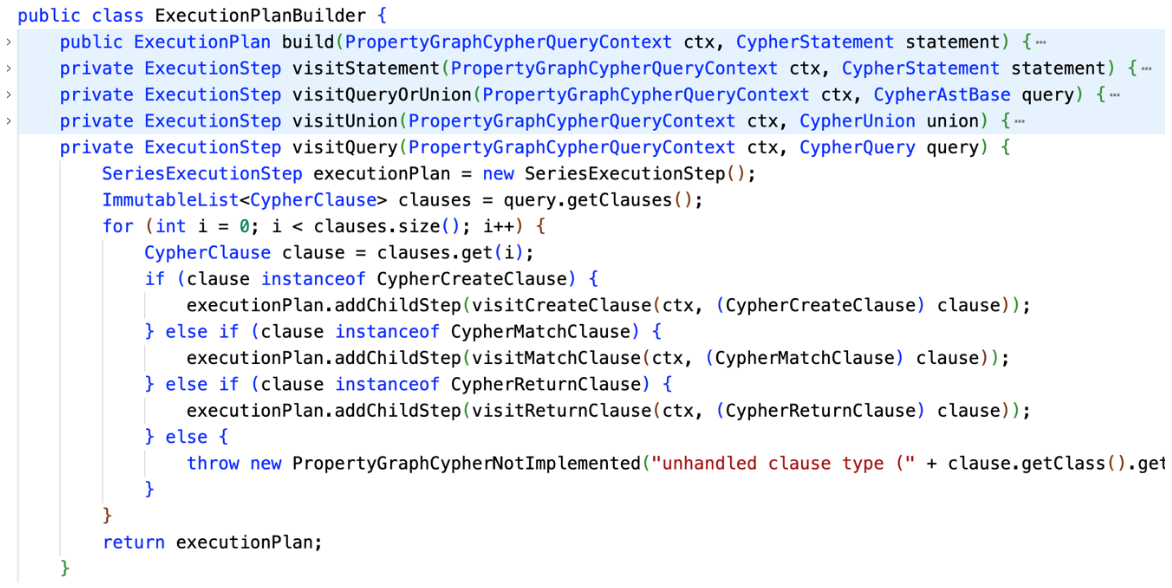
The PropertyGraphCypherVisitor class extends CypherBaseVisitor<CypherAstBase> and implements ANTLR4's CypherVisitor interface. This visitor pattern implementation overrides specific methods to handle various parts of the Cypher grammar. For example:

1) The visitOC_Create() method processes CREATE clauses by visiting the pattern parts and collecting them into an immutable list of CypherPatternPart objects, which are then used to construct and return a CypherCreateClause.

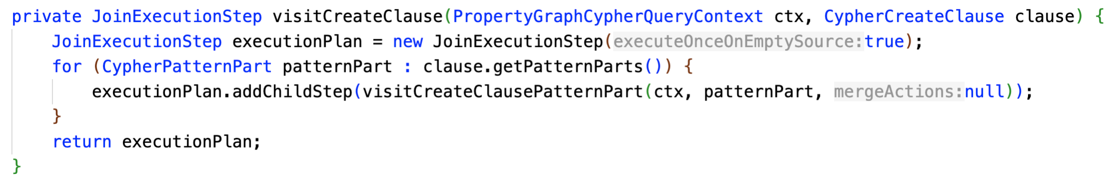
2) The visitOC_PatternPart() method handles pattern parts by extracting the variable and visiting the pattern elements, combining them into a CypherPatternPart.
3) The visitOC_PatternElement() method manages pattern elements by initializing a list of CypherElementPattern, visiting node patterns and relationship patterns sequentially.
4) The visitOC_NodePattern() method processes node patterns by creating a new CypherNodePattern with the variable, properties, and labels extracted from the context.
5) The visitOC_RelationshipPattern method addresses relationship patterns, extracting relationship direction, types and properties, and constructs a CypherRelationshipPattern based on these components.

More methods are outlined in Appendix B1. Each of these methods ensures the correct extraction and construction of the AST nodes demonstrated in Appendix B2, adhering to the Cypher query language's grammar and semantics.

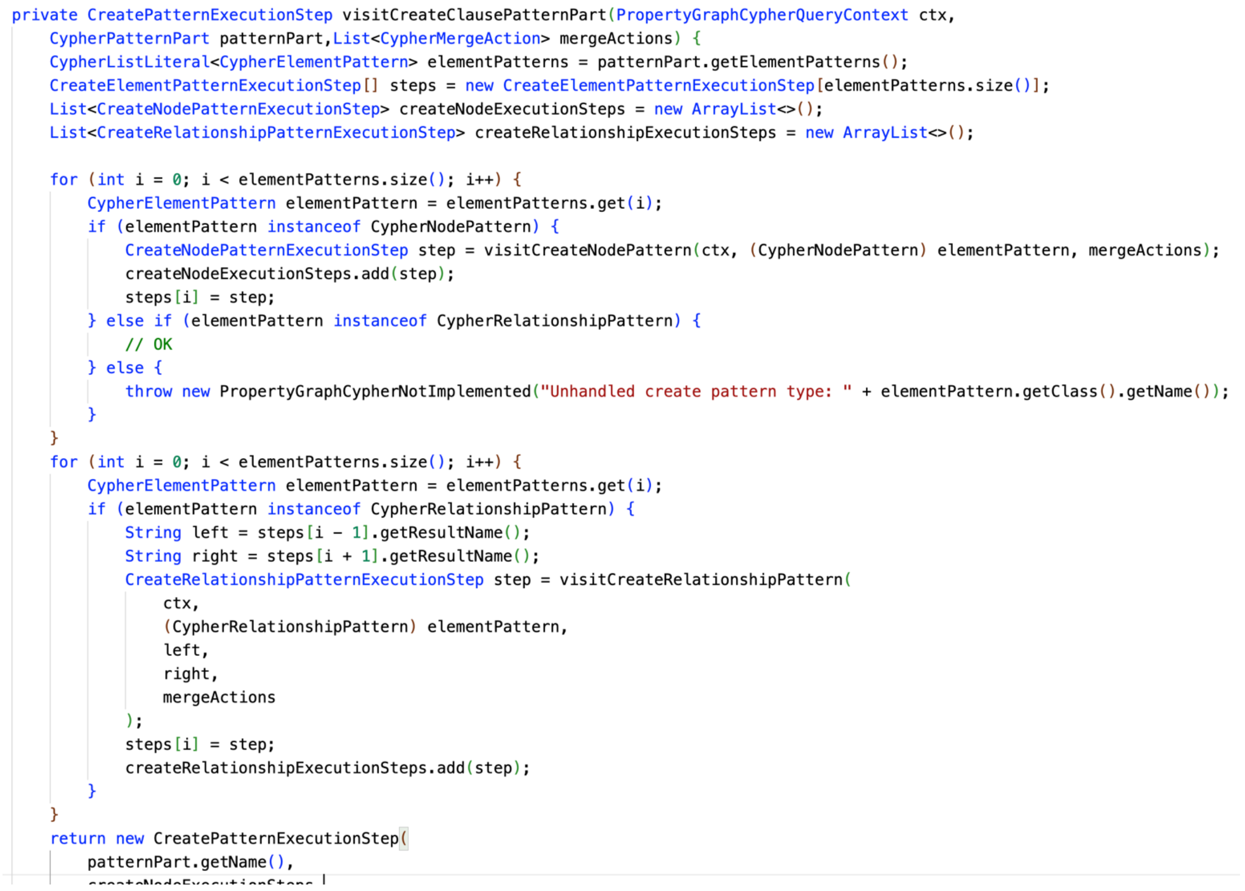
C3: Java code snippet for ExecutionPlanBuilder class with visitCreateClause() as an example.



```
public class ExecutionPlanBuilder {
    public ExecutionPlan build(PropertyGraphCypherQueryContext ctx, CypherStatement statement) {…
    private ExecutionStep visitStatement(PropertyGraphCypherQueryContext ctx, CypherStatement statement) {…
    private ExecutionStep visitQueryOrUnion(PropertyGraphCypherQueryContext ctx, CypherAstBase query) {…
    private ExecutionStep visitUnion(PropertyGraphCypherQueryContext ctx, CypherUnion union) {…
    private ExecutionStep visitQuery(PropertyGraphCypherQueryContext ctx, CypherQuery query) {
        SeriesExecutionStep executionPlan = new SeriesExecutionStep();
        ImmutableList<CypherClause> clauses = query.getClauses();
        for (int i = 0; i < clauses.size(); i++) {
            CypherClause clause = clauses.get(i);
            if (clause instanceof CypherCreateClause) {
                executionPlan.addChildStep(visitCreateClause(ctx, (CypherCreateClause) clause));
            } else if (clause instanceof CypherMatchClause) {
                executionPlan.addChildStep(visitMatchClause(ctx, (CypherMatchClause) clause));
            } else if (clause instanceof CypherReturnClause) {
                executionPlan.addChildStep(visitReturnClause(ctx, (CypherReturnClause) clause));
            } else {
                throw new PropertyGraphCypherNotImplemented("unhandled clause type (" + clause.getClass().get
            }
        }
        return executionPlan;
    }
```



```
private JoinExecutionStep visitCreateClause(PropertyGraphCypherQueryContext ctx, CypherCreateClause clause) {
    JoinExecutionStep executionPlan = new JoinExecutionStep(executeOnceOnEmptySource:true);
    for (CypherPatternPart patternPart : clause.getPatternParts()) {
        executionPlan.addChildStep(visitCreateClausePatternPart(ctx, patternPart, mergeActions:null));
    }
    return executionPlan;
}
```



```
private CreatePatternExecutionStep visitCreateClausePatternPart(PropertyGraphCypherQueryContext ctx,
    CypherPatternPart patternPart,List<CypherMergeAction> mergeActions) {
    CypherListLiteral<CypherElementPattern> elementPatterns = patternPart.getElementPatterns();
    CreateElementPatternExecutionStep[] steps = new CreateElementPatternExecutionStep[elementPatterns.size()];
    List<CreateNodePatternExecutionStep> createNodeExecutionSteps = new ArrayList<>();
    List<CreateRelationshipPatternExecutionStep> createRelationshipExecutionSteps = new ArrayList<>();

    for (int i = 0; i < elementPatterns.size(); i++) {
        CypherElementPattern elementPattern = elementPatterns.get(i);
        if (elementPattern instanceof CypherNodePattern) {
            CreateNodePatternExecutionStep step = visitCreateNodePattern(ctx, (CypherNodePattern) elementPattern, mergeActions);
            createNodeExecutionSteps.add(step);
            steps[i] = step;
        } else if (elementPattern instanceof CypherRelationshipPattern) {
            // OK
        } else {
            throw new PropertyGraphCypherNotImplemented("Unhandled create pattern type: " + elementPattern.getClass().getName());
        }
    }
    for (int i = 0; i < elementPatterns.size(); i++) {
        CypherElementPattern elementPattern = elementPatterns.get(i);
        if (elementPattern instanceof CypherRelationshipPattern) {
            String left = steps[i - 1].getResultName();
            String right = steps[i + 1].getResultName();
            CreateRelationshipPatternExecutionStep step = visitCreateRelationshipPattern(
                ctx,
                (CypherRelationshipPattern) elementPattern,
                left,
                right,
                mergeActions
            );
            steps[i] = step;
            createRelationshipExecutionSteps.add(step);
        }
    }
    return new CreatePatternExecutionStep(
        patternPart.getName(),
        createNodeExecutionSteps,
```

The ExecutionPlanBuilder class constructs an execution plan for processing Cypher queries in our agent-based graph database. It provides methods to traverse and convert different components of a Cypher query into executable steps. For example,

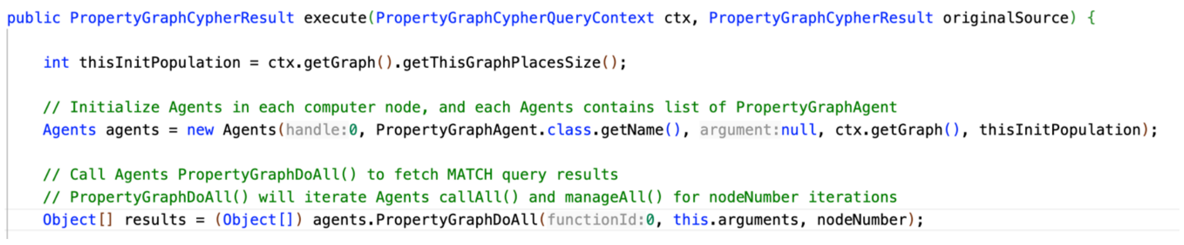
1) The build() method initiates the process by invoking visitStatement, which in turn calls visitQueryOrUnion() to handle either a single query or a union of queries.

2) The visitQuery() method iterates through the clauses of a Cypher query (such as MATCH, CREATE, RETURN), converting each clause into corresponding execution steps.

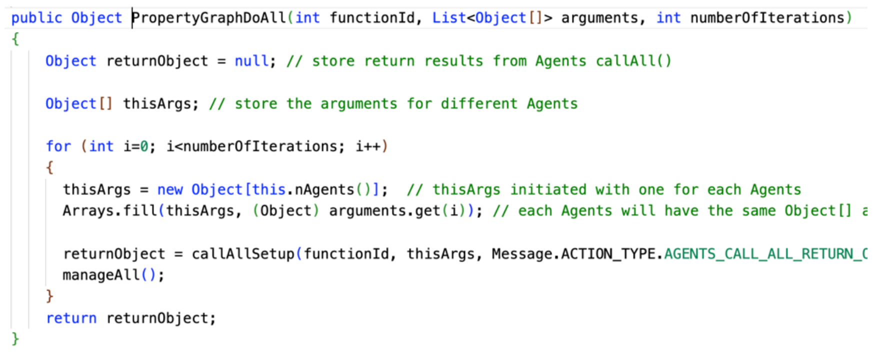
3) Specialized methods like visitCreateClause() and visitCreateClausePatternPart() methods break down CREATE clauses into more granular steps, handling patterns of nodes and relationships.

More methods are outlined in Appendix B3. Each of these methods ensures the correct extraction and construction of the ExecutionStep nodes in the execution plan tree.

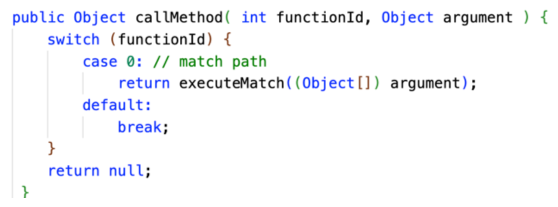
C4: Java code snippet for Agents' callAll() and PropertyGraphAgent's callMethod().



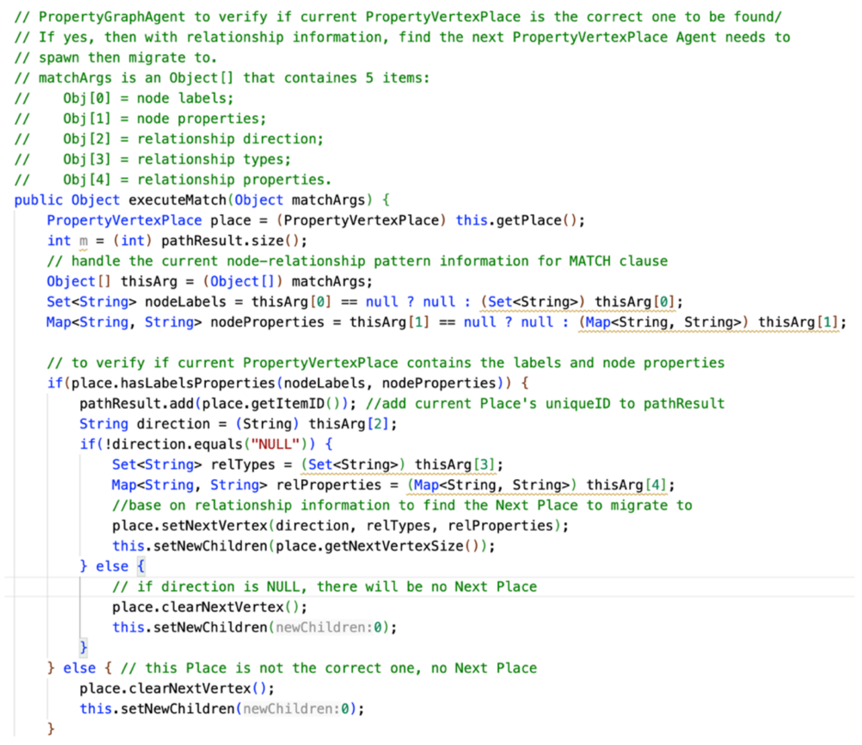
```java
public PropertyGraphCypherResult execute(PropertyGraphCypherQueryContext ctx, PropertyGraphCypherResult originalSource) {

    int thisInitPopulation = ctx.getGraph().getThisGraphPlacesSize();

    // Initialize Agents in each computer node, and each Agents contains list of PropertyGraphAgent
    Agents agents = new Agents(handle:0, PropertyGraphAgent.class.getName(), argument:null, ctx.getGraph(), thisInitPopulation);

    // Call Agents PropertyGraphDoAll() to fetch MATCH query results
    // PropertyGraphDoAll() will iterate Agents callAll() and manageAll() for nodeNumber iterations
    Object[] results = (Object[]) agents.PropertyGraphDoAll(functionId:0, this.arguments, nodeNumber);
```



```java
public Object PropertyGraphDoAll(int functionId, List<Object[]> arguments, int numberOfIterations)
{
    Object returnObject = null; // store return results from Agents callAll()

    Object[] thisArgs; // store the arguments for different Agents

    for (int i=0; i<numberOfIterations; i++)
    {
      thisArgs = new Object[this.nAgents()];  // thisArgs initiated with one for each Agents
      Arrays.fill(thisArgs, (Object) arguments.get(i)); // each Agents will have the same Object[] a

      returnObject = callAllSetup(functionId, thisArgs, Message.ACTION_TYPE.AGENTS_CALL_ALL_RETURN_C
      manageAll();
    }
    return returnObject;
}
```



```java
public Object callMethod( int functionId, Object argument ) {
    switch (functionId) {
        case 0: // match path
            return executeMatch((Object[]) argument);
        default:
            break;
    }
    return null;
 }
```



```java
// PropertyGraphAgent to verify if current PropertyVertexPlace is the correct one to be found/
// If yes, then with relationship information, find the next PropertyVertexPlace Agent needs to
// spawn then migrate to.
// matchArgs is an Object[] that containes 5 items:
//    Obj[0] = node labels;
//    Obj[1] = node properties;
//    Obj[2] = relationship direction;
//    Obj[3] = relationship types;
//    Obj[4] = relationship properties.
public Object executeMatch(Object matchArgs) {
    PropertyVertexPlace place = (PropertyVertexPlace) this.getPlace();
    int m = (int) pathResult.size();
    // handle the current node-relationship pattern information for MATCH clause
    Object[] thisArg = (Object[]) matchArgs;
    Set<String> nodeLabels = thisArg[0] == null ? null : (Set<String>) thisArg[0];
    Map<String, String> nodeProperties = thisArg[1] == null ? null : (Map<String, String>) thisArg[1];

    // to verify if current PropertyVertexPlace contains the labels and node properties
    if(place.hasLabelsProperties(nodeLabels, nodeProperties)) {
        pathResult.add(place.getItemID()); //add current Place's uniqueID to pathResult
        String direction = (String) thisArg[2];
        if(!direction.equals("NULL")) {
            Set<String> relTypes = (Set<String>) thisArg[3];
            Map<String, String> relProperties = (Map<String, String>) thisArg[4];
            //base on relationship information to find the Next Place to migrate to
            place.setNextVertex(direction, relTypes, relProperties);
            this.setNewChildren(place.getNextVertexSize());
        } else {
            // if direction is NULL, there will be no Next Place
            place.clearNextVertex();
            this.setNewChildren(newChildren:0);
        }
    } else { // this Place is not the correct one, no Next Place
        place.clearNextVertex();
        this.setNewChildren(newChildren:0);
    }
```

The provided Java code snippets demonstrate the usage of Agents for executing MATCH Cypher clauses in our graph database system. Firstly, the execute method initializes an Agents instance in each computing node in the cluster network. These Agents instances initialize and allocate one PropertyGraphAgent to each PropertyVertexPlace. PropertyGraphAgent instances are responsible for executing specific tasks related to the MATCH clause across the distributed graph.

The heart of the execution lies in the PropertyGraphDoAll() method. This method orchestrates the distributed execution by iterating over a specified number of iterations. During each iteration, it invokes the callAllSetup() method, which locally executes the callAll() method on the current node and sends a callAll message to remote nodes, passing the necessary arguments for executing the MATCH query.

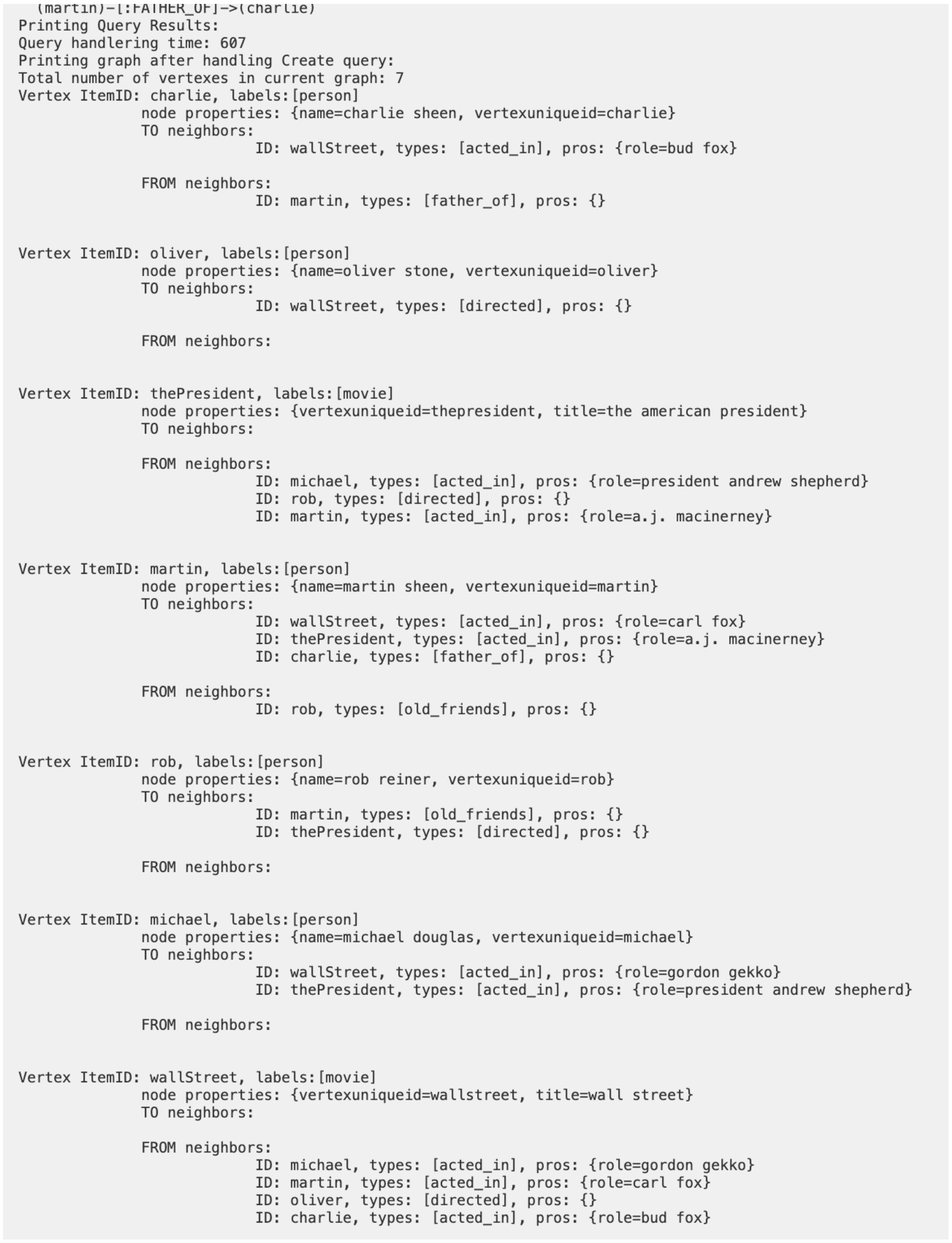
Within the callAll() method, each agent's callMethod() function is invoked with the appropriate functionId. The functionId serves as a key to determine the specific operation to be performed by the agent. For example, when functionId is 0, indicating a MATCH operation, the agent executes the executeMatch() method. This method is responsible for traversing the local portion of the graph assigned to the agent and identifying patterns that match the criteria specified in the Cypher MATCH clause. As the agents complete their computations, the results are collected and aggregated to form a comprehensive response to the MATCH query. The callAll() method manages this aggregation process, ensuring that the results from each agent are combined effectively.

Appendix D

D1. Cypher queries

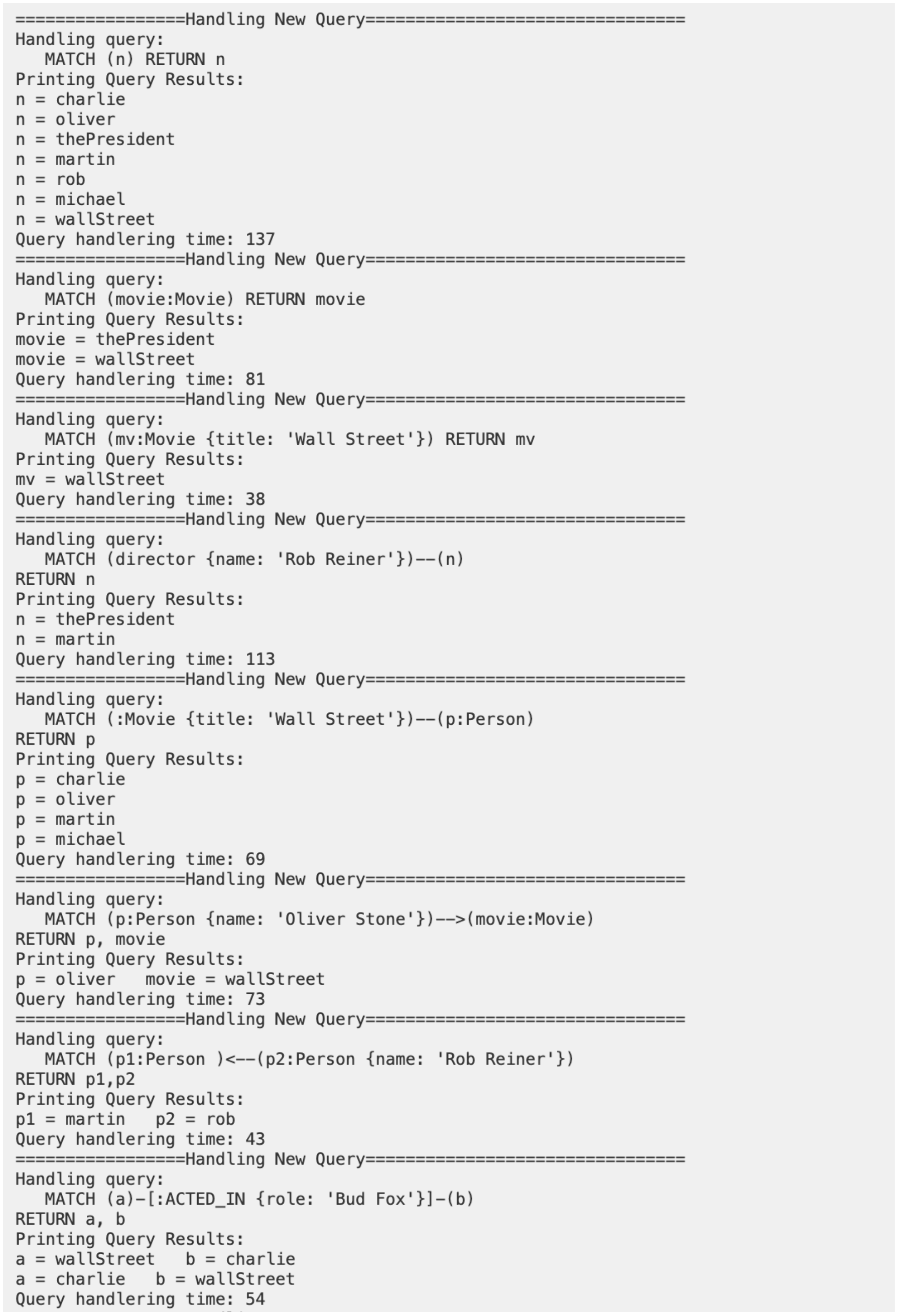
```
// 1. create graph
queries.add("CREATE\n" + //
            "  (charlie:Person {name: 'Charlie Sheen'}),\n" + //
            "  (martin:Person {name: 'Martin Sheen'}),\n" + //
            "  (michael:Person {name: 'Michael Douglas'}),\n" + //
            "  (oliver:Person {name: 'Oliver Stone'}),\n" + //
            "  (rob:Person {name: 'Rob Reiner'}),\n" + //
            "  (wallStreet:Movie {title: 'Wall Street'}),\n" + //
            "  (charlie)-[:ACTED_IN {role: 'Bud Fox'}]->(wallStreet),\n" + //
            "  (martin)-[:ACTED_IN {role: 'Carl Fox'}]->(wallStreet),\n" + //
            "  (michael)-[:ACTED_IN {role: 'Gordon Gekko'}]->(wallStreet),\n" + //
            "  (oliver)-[:DIRECTED]->(wallStreet),\n" + //
            "  (thePresident:Movie {title: 'The American President'}),\n" + //
            "  (martin)-[:ACTED_IN {role: 'A.J. MacInerney'}]->(thePresident),\n" +  //
            "  (michael)-[:ACTED_IN {role: 'President Andrew Shepherd'}]->(thePresident),\n" + //
            "  (rob)-[:DIRECTED]->(thePresident),\n" + //
            "  (rob)-[:OLD_FRIENDS]->(martin),\n" + //
            "  (martin)-[:FATHER_OF]->(charlie)");
// 2. get all nodes
queries.add("MATCH (n) RETURN n");
// 3. get all nodes with a label (e.g. Movie)
queries.add("MATCH (movie:Movie) RETURN movie");
// 4. get all nodes with labels and properties
queries.add("MATCH (mv:Movie {title: 'Wall Street'}) RETURN mv");
// 5. get related nodes with undirected relationship
queries.add("MATCH (director {name: 'Rob Reiner'})--(n)\n" + // n = martin & thePresident
            "RETURN n");


// 6. Match undirected relationship
queries.add("MATCH (:Movie {title: 'Wall Street'})--(p:Person)\n" + // p = oliver, charlie, martin, michael
            "RETURN p");
// 7. Match directed relationship
queries.add("MATCH (p:Person {name: 'Oliver Stone'})-->(movie:Movie)\n" + // wallStreet
            "RETURN p, movie");
queries.add("MATCH (p1:Person )<--(p2:Person {name: 'Rob Reiner'})\n" + // P1: martin P2: rob
            "RETURN p1,p2");
// 8. Match undirected relationship with relation type and property
queries.add("MATCH (a)-[:ACTED_IN {role: 'Bud Fox'}]-(b)\n" + // can be either charlie, wallStreet
            "RETURN a, b");
// 9. Match directed relationship with relation type
queries.add("MATCH (wallstreet:Movie {title: 'Wall Street'})<-[:ACTED_IN]-(actor)\n" + // actor =charlie, martin, michael
            "RETURN actor");
// 10. Match on multiple relationship, diffent directions
queries.add("MATCH (a:Person)-->(b:Person)-->(c:Person)\n" + // rob, martin, charlie
            "RETURN a, b, c");
queries.add("MATCH (a:Person)<--(b:Person)<--(c:Person)\n" + // charlie, martin, rob
            "RETURN a, b, c");
queries.add("MATCH (a:Person)-[:ACTED_IN]->(b:Movie{title: 'Wall Street'})<-[:DIRECTED]-(c:Person)\n" + // a: michael, ma
            "RETURN a, b, c");
queries.add("MATCH (a:Movie {title: 'Wall Street'})<--(b:Person)-->(c:Movie {title: 'The American President'})\n" + // b:
            "RETURN a, b, c");
// 11. Match by Vertex UniqueID
queries.add("MATCH (a {VertexUniqueID: 'thePresident'})<-[:DIRECTED]-(b:Person)\n" + // b: michael & martin
            "RETURN a, b");
// retest create
queries.add("CREATE (thePresident)<-[:DIRECTED]-(Tesh:Person)");
queries.add("MATCH (a {VertexUniqueID: 'thePresident'})<-[:DIRECTED]-(b:Person)\n" + // b: michael & martin
            "RETURN a, b");
```

D2. Printed Graph 1st CREATE query clause

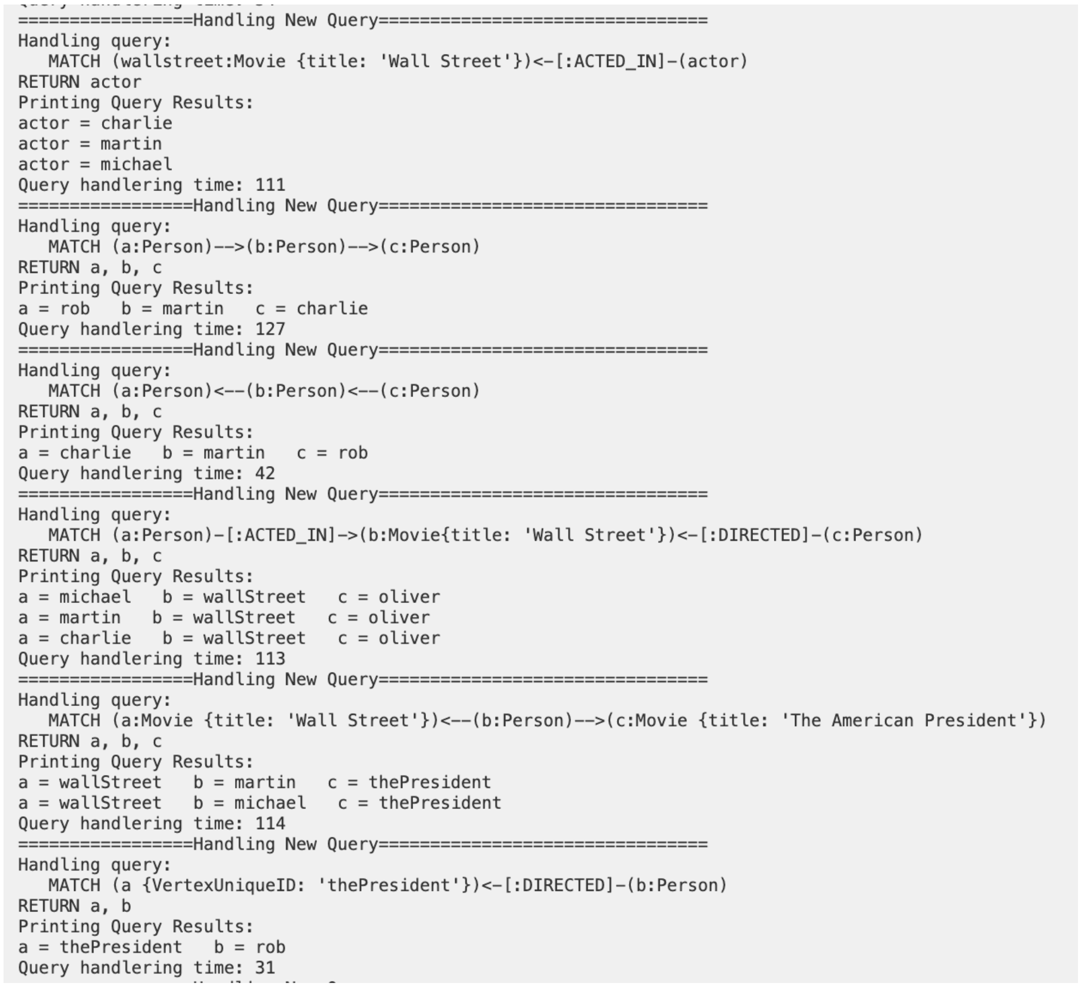


```
  (martin)-[:FATHER_OF]->(charlie)
Printing Query Results:
Query handlering time: 607
Printing graph after handling Create query:
Total number of vertexes in current graph: 7
Vertex ItemID: charlie, labels:[person]
              node properties: {name=charlie sheen, vertexuniqueid=charlie}
              TO neighbors:
                           ID: wallStreet, types: [acted_in], pros: {role=bud fox}

              FROM neighbors:
                           ID: martin, types: [father_of], pros: {}


Vertex ItemID: oliver, labels:[person]
              node properties: {name=oliver stone, vertexuniqueid=oliver}
              TO neighbors:
                           ID: wallStreet, types: [directed], pros: {}

              FROM neighbors:


Vertex ItemID: thePresident, labels:[movie]
              node properties: {vertexuniqueid=thepresident, title=the american president}
              TO neighbors:

              FROM neighbors:
                           ID: michael, types: [acted_in], pros: {role=president andrew shepherd}
                           ID: rob, types: [directed], pros: {}
                           ID: martin, types: [acted_in], pros: {role=a.j. macinerney}


Vertex ItemID: martin, labels:[person]
              node properties: {name=martin sheen, vertexuniqueid=martin}
              TO neighbors:
                           ID: wallStreet, types: [acted_in], pros: {role=carl fox}
                           ID: thePresident, types: [acted_in], pros: {role=a.j. macinerney}
                           ID: charlie, types: [father_of], pros: {}

              FROM neighbors:
                           ID: rob, types: [old_friends], pros: {}


Vertex ItemID: rob, labels:[person]
              node properties: {name=rob reiner, vertexuniqueid=rob}
              TO neighbors:
                           ID: martin, types: [old_friends], pros: {}
                           ID: thePresident, types: [directed], pros: {}

              FROM neighbors:


Vertex ItemID: michael, labels:[person]
              node properties: {name=michael douglas, vertexuniqueid=michael}
              TO neighbors:
                           ID: wallStreet, types: [acted_in], pros: {role=gordon gekko}
                           ID: thePresident, types: [acted_in], pros: {role=president andrew shepherd}

              FROM neighbors:


Vertex ItemID: wallStreet, labels:[movie]
              node properties: {vertexuniqueid=wallstreet, title=wall street}
              TO neighbors:

              FROM neighbors:
                           ID: michael, types: [acted_in], pros: {role=gordon gekko}
                           ID: martin, types: [acted_in], pros: {role=carl fox}
                           ID: oliver, types: [directed], pros: {}
                           ID: charlie, types: [acted_in], pros: {role=bud fox}
```

D3. Printed results for various Match query clauses

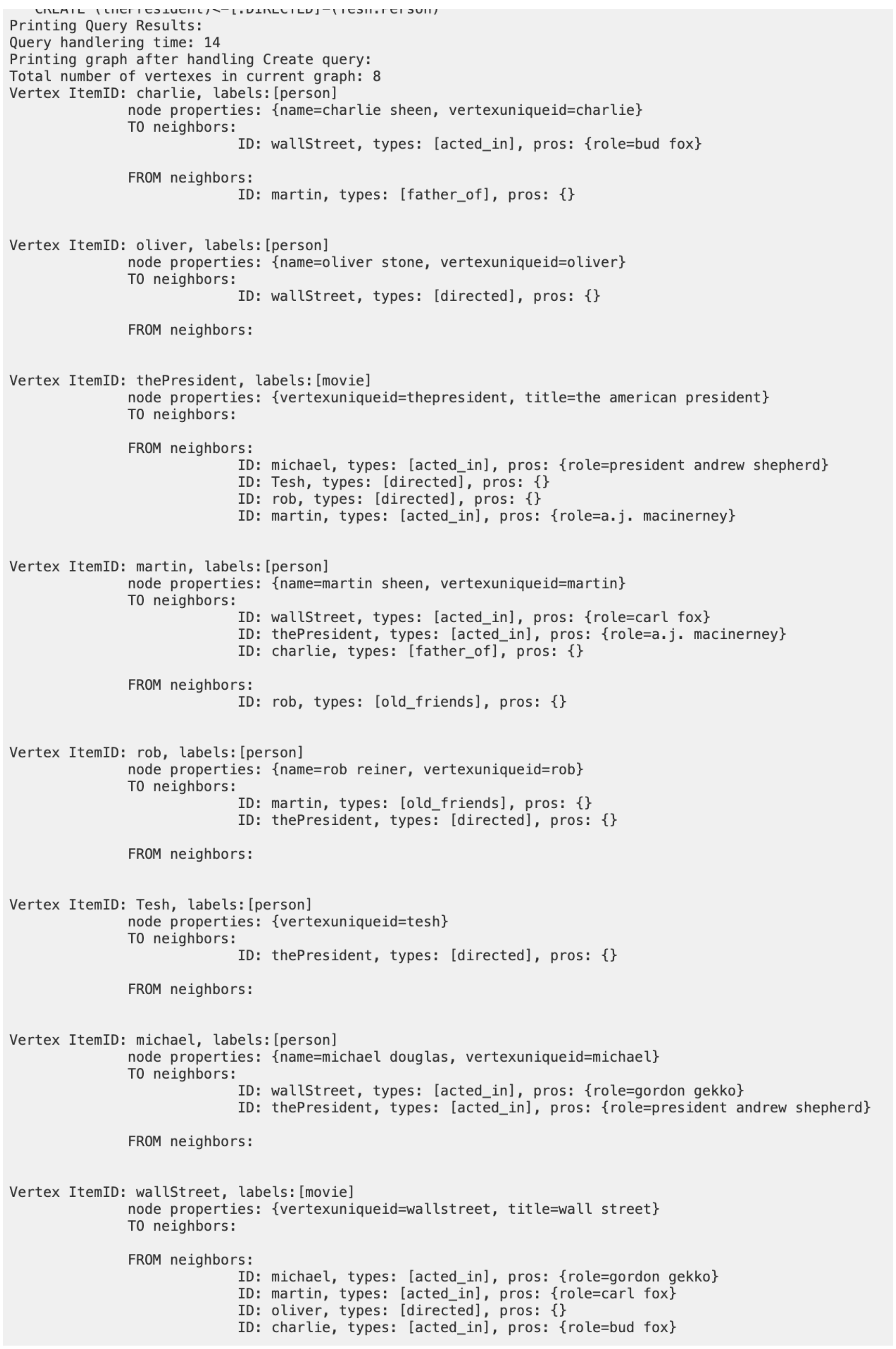


```
=================Handling New Query================================
Handling query:
   MATCH (n) RETURN n
Printing Query Results:
n = charlie
n = oliver
n = thePresident
n = martin
n = rob
n = michael
n = wallStreet
Query handlering time: 137
=================Handling New Query================================
Handling query:
   MATCH (movie:Movie) RETURN movie
Printing Query Results:
movie = thePresident
movie = wallStreet
Query handlering time: 81
=================Handling New Query================================
Handling query:
   MATCH (mv:Movie {title: 'Wall Street'}) RETURN mv
Printing Query Results:
mv = wallStreet
Query handlering time: 38
=================Handling New Query================================
Handling query:
   MATCH (director {name: 'Rob Reiner'})--(n)
RETURN n
Printing Query Results:
n = thePresident
n = martin
Query handlering time: 113
=================Handling New Query================================
Handling query:
   MATCH (:Movie {title: 'Wall Street'})--(p:Person)
RETURN p
Printing Query Results:
p = charlie
p = oliver
p = martin
p = michael
Query handlering time: 69
=================Handling New Query================================
Handling query:
   MATCH (p:Person {name: 'Oliver Stone'})-->(movie:Movie)
RETURN p, movie
Printing Query Results:
p = oliver   movie = wallStreet
Query handlering time: 73
=================Handling New Query================================
Handling query:
   MATCH (p1:Person )<--(p2:Person {name: 'Rob Reiner'})
RETURN p1,p2
Printing Query Results:
p1 = martin   p2 = rob
Query handlering time: 43
=================Handling New Query================================
Handling query:
   MATCH (a)-[:ACTED_IN {role: 'Bud Fox'}]-(b)
RETURN a, b
Printing Query Results:
a = wallStreet   b = charlie
a = charlie   b = wallStreet
Query handlering time: 54
```



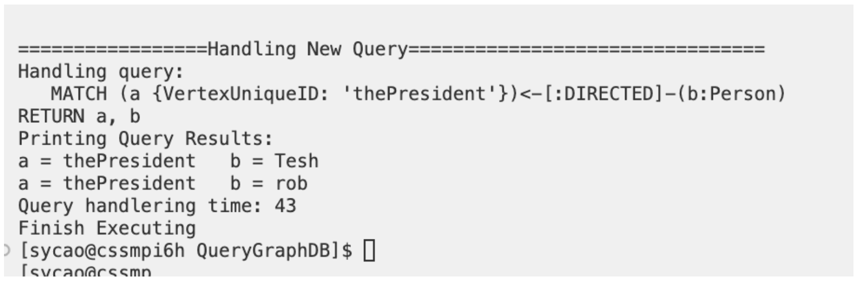
```
=================Handling New Query================================
Handling query:
   MATCH (wallstreet:Movie {title: 'Wall Street'})<-[:ACTED_IN]-(actor)
RETURN actor
Printing Query Results:
actor = charlie
actor = martin
actor = michael
Query handlering time: 111
=================Handling New Query================================
Handling query:
   MATCH (a:Person)-->(b:Person)-->(c:Person)
RETURN a, b, c
Printing Query Results:
a = rob   b = martin   c = charlie
Query handlering time: 127
=================Handling New Query================================
Handling query:
   MATCH (a:Person)<--(b:Person)<--(c:Person)
RETURN a, b, c
Printing Query Results:
a = charlie   b = martin   c = rob
Query handlering time: 42
=================Handling New Query================================
Handling query:
   MATCH (a:Person)-[:ACTED_IN]->(b:Movie{title: 'Wall Street'})<-[:DIRECTED]-(c:Person)
RETURN a, b, c
Printing Query Results:
a = michael   b = wallStreet   c = oliver
a = martin   b = wallStreet   c = oliver
a = charlie   b = wallStreet   c = oliver
Query handlering time: 113
=================Handling New Query================================
Handling query:
   MATCH (a:Movie {title: 'Wall Street'})<--(b:Person)-->(c:Movie {title: 'The American President'})
RETURN a, b, c
Printing Query Results:
a = wallStreet   b = martin   c = thePresident
a = wallStreet   b = michael   c = thePresident
Query handlering time: 114
=================Handling New Query================================
Handling query:
   MATCH (a {VertexUniqueID: 'thePresident'})<-[:DIRECTED]-(b:Person)
RETURN a, b
Printing Query Results:
a = thePresident   b = rob
Query handlering time: 31
```

D4. Printed graph after 2nd CREATE clause



```
   CREATE (thePresident)<-[:DIRECTED]-(Tesh:Person)
Printing Query Results:
Query handlering time: 14
Printing graph after handling Create query:
Total number of vertexes in current graph: 8
Vertex ItemID: charlie, labels:[person]
              node properties: {name=charlie sheen, vertexuniqueid=charlie}
              TO neighbors:
                           ID: wallStreet, types: [acted_in], pros: {role=bud fox}

              FROM neighbors:
                           ID: martin, types: [father_of], pros: {}


Vertex ItemID: oliver, labels:[person]
              node properties: {name=oliver stone, vertexuniqueid=oliver}
              TO neighbors:
                           ID: wallStreet, types: [directed], pros: {}

              FROM neighbors:


Vertex ItemID: thePresident, labels:[movie]
              node properties: {vertexuniqueid=thepresident, title=the american president}
              TO neighbors:

              FROM neighbors:
                           ID: michael, types: [acted_in], pros: {role=president andrew shepherd}
                           ID: Tesh, types: [directed], pros: {}
                           ID: rob, types: [directed], pros: {}
                           ID: martin, types: [acted_in], pros: {role=a.j. macinerney}


Vertex ItemID: martin, labels:[person]
              node properties: {name=martin sheen, vertexuniqueid=martin}
              TO neighbors:
                           ID: wallStreet, types: [acted_in], pros: {role=carl fox}
                           ID: thePresident, types: [acted_in], pros: {role=a.j. macinerney}
                           ID: charlie, types: [father_of], pros: {}

              FROM neighbors:
                           ID: rob, types: [old_friends], pros: {}


Vertex ItemID: rob, labels:[person]
              node properties: {name=rob reiner, vertexuniqueid=rob}
              TO neighbors:
                           ID: martin, types: [old_friends], pros: {}
                           ID: thePresident, types: [directed], pros: {}

              FROM neighbors:


Vertex ItemID: Tesh, labels:[person]
              node properties: {vertexuniqueid=tesh}
              TO neighbors:
                           ID: thePresident, types: [directed], pros: {}

              FROM neighbors:


Vertex ItemID: michael, labels:[person]
              node properties: {name=michael douglas, vertexuniqueid=michael}
              TO neighbors:
                           ID: wallStreet, types: [acted_in], pros: {role=gordon gekko}
                           ID: thePresident, types: [acted_in], pros: {role=president andrew shepherd}

              FROM neighbors:


Vertex ItemID: wallStreet, labels:[movie]
              node properties: {vertexuniqueid=wallstreet, title=wall street}
              TO neighbors:

              FROM neighbors:
                           ID: michael, types: [acted_in], pros: {role=gordon gekko}
                           ID: martin, types: [acted_in], pros: {role=carl fox}
                           ID: oliver, types: [directed], pros: {}
                           ID: charlie, types: [acted_in], pros: {role=bud fox}
```

D5. Printed result for MATCH clause on the updated Graph



```
=================Handling New Query================================
Handling query:
   MATCH (a {VertexUniqueID: 'thePresident'})<-[:DIRECTED]-(b:Person)
RETURN a, b
Printing Query Results:
a = thePresident   b = Tesh
a = thePresident   b = rob
Query handlering time: 43
Finish Executing
[sycao@cssmpi6h QueryGraphDB]$ 
[sycao@cssmp
```