AN INCREMENTAL ENHANCEMENT OF AGENT-BASED GRAPH DATABASE

SHENYAN CAO

A Capstone Project Term Report
submitted in partial fulfillment of the
requirements of the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

Dec 13, 2023

Project Committee:

Professor Munehiro Fukuda, Committee Chair
Professor Clark Olson, Committee Member
Professor Wooyoung Kim, Committee Member

1. Project Overview

With the rise of big data and the need to analyze interconnected data, graph databases have gained popularity as a valuable tool for managing and exploring large and complex datasets (Robinson et al., 2013). A graph database is a type of database that uses graph structures to represent and store data. In a graph database, data is organized as nodes and edges (Robinson et al., 2013). Nodes represent entities, and edges represent the relationships between those entities (Robinson et al., 2013). This graph-based representation allows for efficient querying and analysis of complex relationships and connections within the data. Graph databases support the storage of large-scale graphs with billions of nodes and edges, and they often provide specialized query languages (such as Cypher for Neo4j) that facilitate expressive and efficient traversal of the graph to discover patterns, identify paths, and perform graph algorithms (Raj, 2015; Robinson et al., 2013).

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 (Eadline, 2018). These tools have been instrumental in enabling real-time data processing and building scalable analytics pipelines (Eadline, 2018). Their support for big-data computing and data sciences, with data formats primarily in text, has made them widely adopted tools (Eadline, 2018). 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 (Eadline, 2018). 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 structured dataset like a graph in big-data computing, it is more logical to retain the structure in memory and deploy agents within it, as opposed to breaking down the structure and streaming its data to traditional analytical tools like Spark (Li & Fukuda, 2023). Agents hold significant potential in supporting the analysis of data structures due to their ability to be deployed repetitively within data structures mapped over distributed memory. This capability becomes particularly promising in domains such as graph databases, where the construction of graphs over distributed disks or, preferably, distributed memory is required. With agent-based graph database, it can achieve efficient graph construction and analysis, facilitating the processing of highly interconnected data (Hong & Fukuda, 2022).

Throughout previous years, the Distributed Systems Laboratory (DSL) at University of Washington has come up with an Agent Based Graph Database Model, which is based on the MASS (multi-agent spatial simulation) Java framework and makes full use of the comprehensive features provided by the MASS library. Their approach builds graphs over a cluster system, deploy numerous reactive agents to datasets, and task these agents with computing data attributes or shapes (Mohan et al., 2022). Nevertheless, the current database system is structured to accommodate nodes and edges with specific properties. Each node in the graph is defined by its unique identifier, and edges are characterized by the identifiers of connected nodes along with a weight associated with the relationship. This simplicity in attribute structure may be suitable for scenarios where the relationships between entities can be adequately represented using only these three attributes. However, it's essential to recognize that more complex data models with additional attributes may be required for applications where richer information about nodes and relationships is necessary. This capstone project seeks to reengineer the existing system to enhance its capability to handle data with richer information about nodes and relationships in accordance with the Property Graph Model. Property Graph Model is well-suited for various use cases, including social networking, recommendation engines, knowledge graphs, fraud detection, and any application where understanding and leveraging relationships are crucial.

But even with this enhancement, our graph database model will still be in the form of diagrams. Diagrams are great for describing graphs outside of any technology context, but when it comes to using a database, we need some other mechanism for creating, manipulating, and querying data. We need a query language. OpenCypher is an open query language specifically designed for querying graph databases. It was initially developed by Neo4j, a popular graph database management system, but has since gained broader adoption and is used by various graph database systems (Raj, 2015; Robinson et al., 2013). OpenCypher provides a standardized syntax and set of operations for querying and manipulating graph data (openCypher, 2017). This capstone project seeks to integrate OpenCypher with MASS-based graph database by implementing Create, Match, Return and Delete clauses. With these enhancements, the agent-based graph database will become more feature-rich, offering users with a broader range of query options and empowering agents to effectively match user queries with relevant vertices and edges in the MASS-based graph database system. The implementation of the enhancements requires a deep understanding of graph databases, query languages, and distributed computing. By successfully implementing the enhancements, it showcases my proficiency in these areas and the ability to work with complex technologies.

2. Goals

Goal 1: Reengineer the existing system to enhance its capability to handle data with various property information about nodes and relationships in accordance with the Property Graph Model.

The Property Graph Model is a type of graph database model that represents data as a graph, consisting of nodes, relationships, and properties (Raj, 2015; Robinson et al., 2013). In this model:

- Nodes are the entities in the graph.
- Nodes can be tagged with labels, representing their different roles in your domain. (For example, Employee in Figure 1).
- Nodes can hold any number of key-value pairs, or properties. (For example, name in Figure 1)
- Relationships provide directed, named, connections between two node entities (e.g. Person LOVES Person).
- Relationships always have a direction, a type, a start node, and an end node, and they can have properties, just like nodes.
- Nodes can have any number or type of relationships without sacrificing performance.
- Although relationships are always directed, they can be navigated efficiently in any direction.

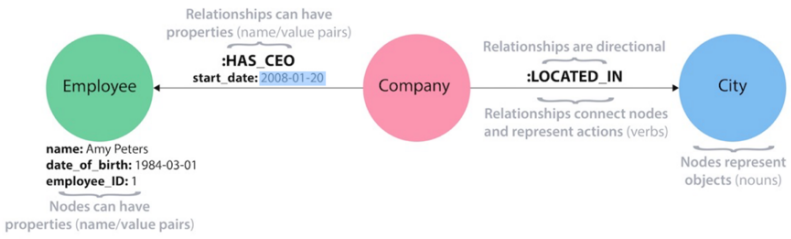


Figure 1: Node and Relationship demonstration in Property Graph Data Model.

For domains with inherently connected and interdependent data, such as social networks, supply chains, or hierarchical structures, the Property Graph Model provides a natural and efficient way to model and represent complex relationships between entities. Nodes, relationships, and properties allow for a fine-grained representation of the real-world scenario.

Goal 2: Integrate OpenCypher into MASS-based graph database by implementing Create, Match, Return and Delete clauses.

OpenCypher stands out as a graph database query language that is both expressive and concise. While it is tailored for Neo4j, its alignment with our tendency to depict graphs through diagrams makes it exceptionally well-suited for accurately and programmatically describing graphs (Robinson et al., 2013). Integrating OpenCypher with MASS-based property graph database allows for a consistent and widely adopted query interface. It simplifies the learning curve for users and promotes interoperability, as they can leverage their existing knowledge of OpenCypher to interact with the property graph database within the MASS framework.

The cypher clauses to be implemented are:

- CREATE: Creates nodes and relationships in the graph.
- MATCH: Specifies the pattern to search for in the graph (example as shown in Figure 2).
- DELETE: Removes nodes, relationships, or properties from the graph.
- RETURN: Specifies what data to include in the query results.

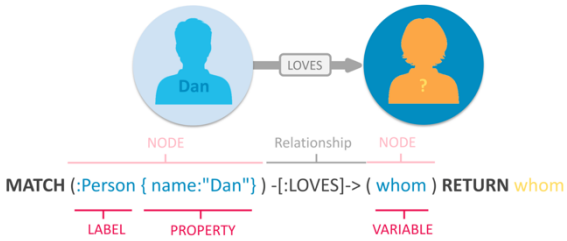


Figure 2: Example of Match cypher clause.

Goal 3: Conduct comprehensive testing on the enhanced MASS-based Property Graph Database Management System to ensure its functionality, performance, and reliability meet the desired standards.

3. Achievements

3.1 Main Design for the whole project

As shown in Figure 3, the main design of the whole project follows the three-tier architecture patter, with presentation layer, logic layer and data access layer.

- **Presentation Layer (Tier 1):**
  - This is the main program of the interface that users interact with.
  - Users have no direct access to the underlying graph data or knowledge of the database structure.
  - Users make requests to the system through the GraphManager component.
- **Logic Layer (Tier 2):**
  - This layer serves as an intermediary between the presentation layer and the data access layer.
  - This layer contains the GraphManager class, which exposes CRUD methods (Create, Read, Update, Delete) to users, providing a high-level interface for interaction with the MASS-based property graph database.
- **Data Access Layer (Tier 3):**
  - This is where the actual MASS-based property graph database resides.
  - Users, through the GraphManager in the application layer, interact with the database using standard CRUD operations.
  - Graph Database is the underlying database that stores the graph data.
  - Query Handler is responsible for handling OpenCypher queries. It interprets CRUD requests received from the GraphManager class and translates them into appropriate OpenCypher queries to be executed against the graph database.

- PropertyGraphCypherQueryContext provides the interface for Query Handler to executed queires and interact with GraphHandler.
- The underlying structures of the graph database and query handler are abstracted from users, ensuring they only need to be concerned with the high-level CRUD operations, while the Graph Database and Cypher Handler manages the low-level details of graph database and query parsing and execution.

The 3-tier architecture provides modularity, scalability, and a clear separation of concerns, allowing for easier maintenance and development of the system.

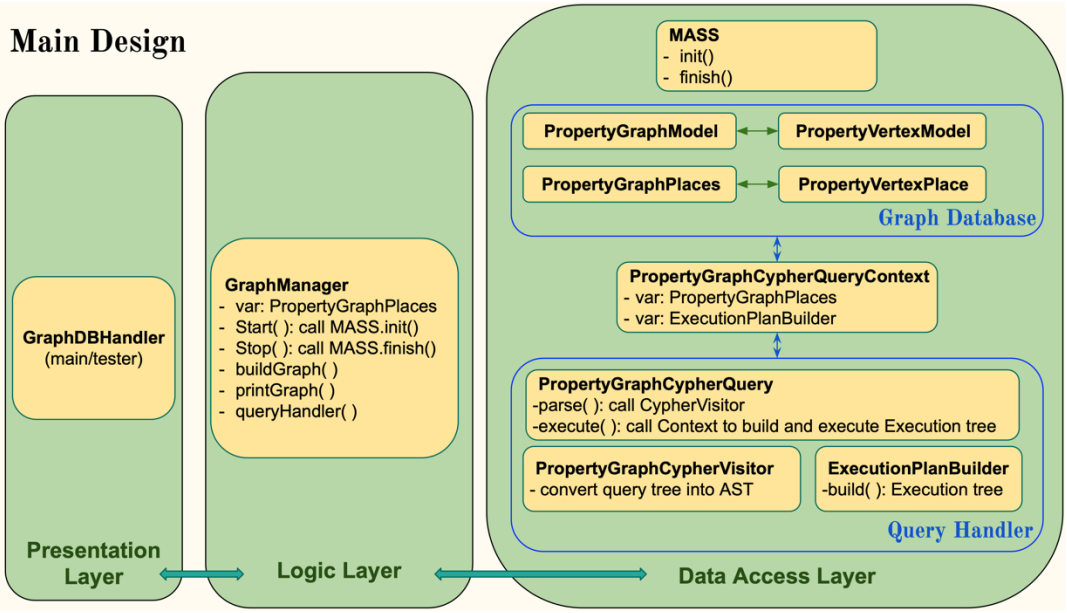


Figure 3: Main design for the whole project.

3.2 Design and Implementation for enhancing the existing MASS-based graph database

The existing MASS-based graph database mainly consists of GraphPlaces class and VertexPlace class, as shown in Figure 4. GraphPlaces extends MASS Places while VertexPlace extends MASS Place. The VertexPlace stores the node information of Vertex ID, Neighbor Vertex ID, and relationship weight. GraphPlaces stores a vector of VertexPlace, which serves as the lower-level graph database management interface.

To enhance the capacity of the existing system in accordance with the property graph model, the PropertyGraphPlaces class is designed and implemented to extend GraphPlaces, while PropertyVertexPlace class is designed and implemented to extend VertexPlace. The variables declared in PropertyVertexPlace and PropertyGraphPlaces are shown in Figure 5. In PropertyVertexPlace, the variable labels uses a Set to store node labels, which the variable nodeProperties uses a Map to store node property type-value pairs. The toRelationship stores the outgoing neighbor relationship information with Map<neighbor ItemID, relationshipInformation>. The relationshipInformation is an Object array which stores the relationship types in Set and relationship properties in Map. The fromRelationship stores the incoming relationship information with the similar way as toRelationship. These enhancements provide a more property graph-oriented structure, allowing for the storage and retrieval of properties associated with nodes and relationships in property graph model.

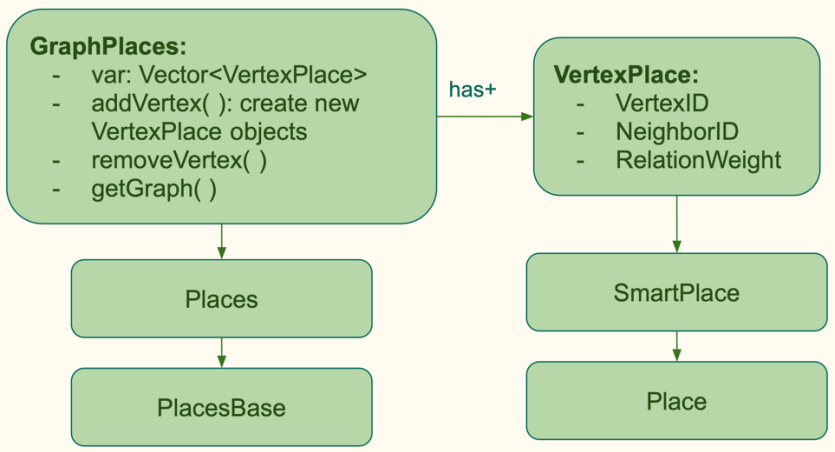


Figure 4: Existing MASS-based graph database framework based on MASS 1.4.3-SNAPSHOT version (develop branch).

```
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 5: PropertyVertexPlace variables.

3.3 Design and Implementation for integrating OpenCypher with MASS-based Property Graph Database

The process to integrate OpenCypher with MASS-based Property Graph Database is shown in Figure 6. Query text is first parsed to AST structure, and then executed with execution plan, and finally get query results.

Figure 6: Query flow process.

3.3.1 Parser generator

To seamlessly integrate Cypher with the MASS-based Property Graph Database, the initial and pivotal step is to select an adept language parser for translating Cypher queries into a Java-compatible format. Given the inherent complexity of the OpenCypher query grammar, the parsing strategy employed becomes crucial. In this context, ANTLR4 (ANother Tool for Language Recognition, fourth version) emerges as an optimal choice. ANTLR4 is a powerful parser generator, and its parsing strategy is renowned for its exceptional flexibility, making it well-suited for handling the intricate structures of the OpenCypher language (Parr, 2013). Beyond its technical merits, ANTLR4 boasts a broad adoption across diverse language communities, attesting to its reliability and versatility. Most importantly, there is an existing ANTLR4 grammar file for OpenCypher (i.e. Cypher.g4). The first step to building a language application is always to create a grammar that describes a language's syntactic rules. ANTLR4 uses grammars defined in .g4 files to generate lexer and parser code for a particular language (Parr, 2013). The Cypher grammar file (i.e. Cypher.g4) in ANTLR4 can be readily used to generate lexer and parser code for Cypher queries. Furthermore, as we are using Maven in our project, we can use the ANTLR4

Maven Plugin to integrate ANTLR4 into our project. From Cypher.g4, once we build the application with ANTLR4 plugin, then ANTLR automatically generates lots of code files corresponding to Cypher.g4, as shown in Figure 7. These code files are the base to parse query text to AST.

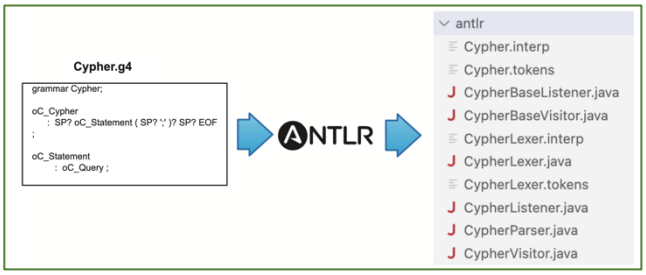


Figure 7: ANTLR4 autogenerates code files from Cypher.g4.

In our project, query string is converted to CharStream, then lexer and parser are generated accordingly. Then we manually convert the parse tree to AST with CypherVisitor mechanism. Figure 8 shows the code to parse query string and call PropertyGraphCypherVisitor to handler parser tree. PropertyGraphCypherVisitor extends CypherBaseVisitor, which implements CypherVisitor.

```
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 and create a parser that feeds off the tokens buffer
    CypherParser parser = new CypherParser(new CommonTokenStream(lexer));
    parser.setErrorHandler(new ParserErrorHandler(queryString));
    CypherParser.OC_CypherContext tree = parser.oC_Cypher();// begin parsing at oC_Cypher
    String treeText = tree.getText();

    if (treeText.endsWith("<EOF>")) {
        treeText = treeText.substring(0, treeText.length() - "<EOF>".length());
    }
    if (!treeText.equals(queryString)) {
        throw new PropertyGraphCypherSyntaxErrorException("Parsing error, \"" + queryString.substring(treeTe
    }
    return new PropertyGraphCypherVisitor().visitOC_Cypher(tree);
}
```

Figure 8: Code to parse query string and handle with Cypher Visitor.

3.3.2 Parsing query text to AST

Cypher query parsing with ANTLR4 is the process of analyzing a sequence of symbols to determine its grammatical structure (Parr, 2013). This involves breaking down the input into its constituent parts according to the Cypher.g4 gramma file. Figure 11 Shows the grammatical structure for the query text of “CREATE (charlie:Person:Actor {name: 'Charlie Sheen'})”.

```
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 9: The grammatical structure of a CREATE query text.

An Abstract Syntax Tree (AST) is a hierarchical tree structure that represents the abstract syntactic structure of source code or a query. Each node in the tree corresponds to a syntactic construct in the code, and the edges represent the relationships between them. To translate Cypher queries into a Java-compatible format, we need to parse query text to AST, which involves transforming the parser tree into a structured AST representation. This representation captures the essential syntactic elements and their relationships, making it easier to analyze and manipulate the query code programmatically.

```java
public class PropertyGraphCypherVisitor extends CypherBaseVisitor<CypherAstBase> {
    private final CypherCompilerContext compilerContext;

    public PropertyGraphCypherVisitor() {
        this(new CypherCompilerContext());
    }

    public PropertyGraphCypherVisitor(CypherCompilerContext compilerContext) {
        this.compilerContext = compilerContext;
    }

    @Override
    public CypherStatement visitOC_Statement(CypherParser.OC_StatementContext ctx) {
        return new CypherStatement(visitQuery(ctx.oC_Query()));
    }

    public CypherAstBase visitQuery(CypherParser.OC_QueryContext ctx) {
        return visitRegularQuery(ctx.oC_RegularQuery());
    }

    public CypherAstBase visitRegularQuery(CypherParser.OC_RegularQueryContext ctx) {
        CypherQuery left = visitSingleQuery(ctx.oC_SingleQuery());
        if (ctx.oC_Union().size() > 0) {
            return visitUnions(left, ctx.oC_Union());
        }
        return left;
```

Figure 10: The grammatical structure of a CREATE query text.

In our project, we use CypherAstBase, an abstract class, to serve as the base for the structured AST representation. PropertyGraphCypherVisitor transforms the cypher parser tree to AST structure. Then for each syntactic construct in cypher parse tree, various classes that extends CypherAstBase class would be used to store information accordingly. For example, CypherStatement extends CypherAstBase and is return by the visitOC_Statement() function. Figure 10 shows partial implementation of the PropertyGraphCypherVisitor class. The detailed function flow of PropertyGraphCypherVisitor is shown in Appendix B1. AST structure is shown in Appendix B2.

3.3.3 Planning for Execution

In this project, we used the ExecutionPlanBuilder to produce logical execution plans which describe how a particular query is going to be executed (i.e. steps for execution). This execution plan is essentially a binary tree of operators. An operator is, in turn, a specialized execution module that is responsible for some type of transformation to the data before passing it on to the next operator, until the desired graph pattern has been matched. The execution plans produced by the planner thus decide which operators will be used and in what order they will be applied to achieve the aim declared in the original query.

```java
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) {
```

```java
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;
}
```

```java
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());
        }
```

Figure 11: The code to add execution steps to the execution plan.

Figure 11 shows the code to add execution steps to the executionPlan tree. In visitQuery() function, SeriesExecutionStep is the execution plan to store the execution steps in a tree structure. For example, for the create node clause, executionPlan will add a child execution step by calling visitCreateClause() function. In visitCreateClause() function, for each CypherPatterPart, it will call visitCreateClausePatternPart() function to create a CreatePatterExecutionStep, which contains information about node pattern creation steps (e.g. CreateNodePatternExecutionStep) and relationship pattern creation steps. With the execution steps returned from various functions, the executionPlan will store execution steps in a tree structure. A detailed tree structure for execution steps are shown in Appendix B3.

3.3.4 Execution to results

Figure 12 shows the code for execute() function of CreateNodePatternExecutionStep. Upon execute, it extracts labels and properties from the Cypher query, and call PropertyGraphPlaces's addPropertyVertex( ) function to create new PropertyVertexPlace with labels and properties.

```
@Override
public PropertyGraphCypherResult execute(PropertyGraphCypherQueryContext ctx, PropertyGraphCypherResult source) {
    MASS.getLogger().error("At createNodePatternExecutionStep, Item Name:" + name);
    source = super.execute(ctx, source);

    return source.peek(row -> {
        Map<String, String> properties = new HashMap<String, String>();
        for (String propertyResultName : propertyResultNames) {
            Object value = row.get(propertyResultName);
            if (value instanceof CypherAstBase) {
                throw new PropertyGraphCypherNotImplemented("Unhandled type: " + value.getClass().getName());
            }
            if (value != null) {
                String sValue = (String) value;
                properties.put(propertyResultName.trim().toLowerCase(), sValue.trim().toLowerCase());
            }
        }

        List<String> labels = new ArrayList<>();
        for(int i = 0; i < labelNames.size(); i++) {
            labels.add(labelNames.get(i).trim().toLowerCase());
        }

        // store PropertyVertexPlace in MASS library,
        // vertexId is the internal reference in MASS library
        int vertexId = ctx.getGraph().addPropertyVertex(name, labels, properties); // set node properties
        if(vertexId == -1) {
            // MASS.getLogger().error("At createNodePatternExecutionStep: Failed at adding Vertex to Graph.  Item N
            return;
        }

    });
```

Figure 12: Code of CREATE execution step's execution() function.

Figure 13 shows the code for execute() function of MatchPatternPartExecutionStep. Upon execute, it creates an Agents instance and call Agent.PropertyGraphDoAll( ) to fetch Match results from graph database. Agents initially initialize one PropertyGraphAgent on each PropertyVertexPlace. Then upon Agents.callAll( ) method, each agent will carry the current Match node and/or relationship information from the Match clause and then proceed to verify if the current PropertexVertexPlace is the correct node that satisfies the node matching information. If validated, then add current PropertyVertexPlace's itemID to the results list and further determine the neighboring PropertyVertexPlaces to migrate to. Results list would be returned to main server at the end of CallAll( ) method. After that, Agent.manageAll( ) would be called, and new Agent would be spawn based on the neighbor vertex information, and then migrated to the destinated PropertyVertexPlace. This cycle continues until all Match clause criteria are fulfilled. In this way, agents transmit the gathered results to the main server.

Agents' callAll( ) function calls each PropertyGraphAgent's callMethod( ), which based on the functionID would call each agent's executeMatch( ) function. The executeMatch( ) function, shown in Figure 14, processes the matchArgs parameter, which contains information about node labels, node properties, relationship direction, relationship types, and relationship properties. It checks if the current place (PropertyVertexPlace) has the specified node labels and properties. If the node labels and properties match, the current place's item ID is added to the pathResult. This indicates a successful match. Depending on the specified direction, it sets the next vertex (neighbor vertexes) based on the given relationship types and properties and updates the newChildren count of the agent based on the number of next vertices set. If the direction is "NULL," it clears the next vertex, indicating no further traversal in that direction.

```
@Override
public PropertyGraphCypherResult execute(PropertyGraphCypherQueryContext ctx, PropertyGraphCypherResult originalSource) {

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

    Agents agents = new Agents(handle:0, PropertyGraphAgent.class.getName(), argument:null, ctx.getGraph(), thisInitPopulation);

    Object[] results = (Object[]) agents.PropertyGraphDoAll(functionId:0, this.arguments, nodeNumber);

    MASS.getLogger().debug("At MacthPatternPartExecutionStep: Agents size " + agents.nAgents() );

    LinkedHashSet<String> columnNames = new LinkedHashSet<String>(pathResultNames);

    Stream<CypherResultRow> rows = Arrays.stream(results)
                                   .filter(result -> ((ArrayList<String>) result).size() == nodesResultNames.size())
                                   .map(result -> {
                                       ArrayList<String> elements = (ArrayList<String>) result;
                                       Map<String, Object> elementMap = new HashMap<>();
                                       AtomicInteger i = new AtomicInteger(0); // Using AtomicInteger for indexing within forEach
                                       for(String x : elements) {
                                           elementMap.put(nodesResultNames.get(i.getAndIncrement()), (Object) x);
                                       }
                                       return (CypherResultRow) new DefaultCypherResultRow(columnNames, elementMap);
                                   });

    PropertyGraphCypherResult source = new PropertyGraphCypherResult(rows, columnNames);

    return source;
}
```

Figure 13:Code of MATCH execution step's execution() function

```
public Object executeMatch(Object matchArgs) {
    MASS.getLogger().debug("At propertyGraphAgent's executeMatch, path result before executing: " + (pathResult == null ? "null" : pathRes
    PropertyVertexPlace place = (PropertyVertexPlace) this.getPlace();
    MASS.getLogger().debug("                                    current place: " + place.getItemID());
    MASS.getLogger().debug("                                    place's nextVertex size: " + place.getNextVertexSize());

    int m = (int) pathResult.size();
    MASS.getLogger().debug("                                    current pathResult size: " + m);

    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];
    MASS.getLogger().debug("                                    thisArg: " + nodeLabels + ", " + nodeProperties);

    if(place.hasLabelsProperties(nodeLabels, nodeProperties)) {
        pathResult.add(place.getItemID());
        String direction = (String) thisArg[2];
        MASS.getLogger().debug("At PropertyGraphAgent executeMatch, relationship direction: " + direction);
        if(!direction.equals("NULL")) {
            Set<String> relTypes = (Set<String>) thisArg[3];
            Map<String, String> relProperties = (Map<String, String>) thisArg[4];
            place.setNextVertex(direction, relTypes, relProperties);
            this.setNewChildren(place.getNextVertexSize());
            MASS.getLogger().debug("                                        relTypes: " + relTypes);
            MASS.getLogger().debug("                                        relProperties: " + relProperties);
            MASS.getLogger().debug("                                        place's nextVertex size: " + place.getNextVertexSize());
            MASS.getLogger().debug("                                        this agent's new children: " + this.getNewChildren());
        } else {
            place.clearNextVertex();
            this.setNewChildren(newChildren:0);
        }
```

Figure 14: PropertyGraphAgent's executeMatch( ) function.

4. Results

4.1 Execution and verification of building MASS-based Property Graph Database

With the queries shown in Appendix C1, the graph database was built and printed out as shown Appendix C2. The graph was printed out correctly based on the Create query clause, and Figure 15 is the illustration of the current graph.

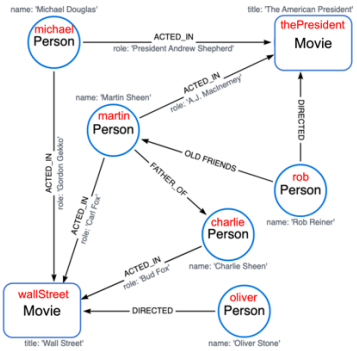


Figure 15: Graph created with Create Clauses

4.2 Execution and verification of executing the Match query clauses.

OpenCypher Match clauses tested are shown in Appendix C1. Appendix C3 shows the screenshots of the results for Match query clauses tested on the created graph shown in Figure 13. The Match query results are fetched from graph database using Agents. All the results from Match query clauses are verified to be correct and in line with Figure 15.

5. Spring Quarter Project Plan

Timeline	Tasks
Spring 2024	
Week 1 & 2	- Finish MATCH and DELETE query clause. - Schedule final defense.
Week 3 & 4	- Finalize code package. - Start writing white paper.
Week 5 & 6	- Writing white paper.
Week 7 & 8	- Writing white paper.
Week 9 & 10	- Final defense. - Final deliverables, including white paper and code package.

6. Summary

In current project, we have successfully extended the MASS-based graph database to handle complex node and relationship property information and executed OpenCypher CREATE and MATCH node and relationship queries accurately. Future tasks are outlined for further development during Spring Quarter. DELETE clauses will be integrated into our MASS-based property graph database to enhance its query capabilities and support a broader range of functionalities.

Reference

Eadline, D. (2018). Hadoop and Spark fundamentals : LiveLessons. Pearson.

Hong, Y., & Fukuda, M. (2022). Pipelining Graph Construction and Agent-based Computation over Distributed Memory. 2022 IEEE International Conference on Big Data (Big Data), 4616–4624. https://doi.org/10.1109/BigData55660.2022.10020903

Li, A., & Fukuda, M. (2023). Agent-Based Parallelization of a Multi-Dimensional Semantic Database Model. IRI, 64–69. https://doi.org/10.1109/IRI58017.2023.00019

Mohan, V., Potturi, A., & Fukuda, M. (2022). Automated agent migration over distributed data structures. In Proceedings of the 15th International Conference on Agents and Artificial Intelligence, 1.

openCypher. (2017, November). Cypher Query Language Reference, Version 9. Https://Github.Com/Opencypher/OpenCypher/Blob/Master/Docs/OpenCypher9.Pdf.

Parr, T. (2013). The Definitive ANTLR 4 Reference (2nd ed.). Pragmatic Bookshelf.

Raj, Sonal. (2015). Neo4j High Performance : design, build, and administer scalable graph database systems for your applications using Neo4j. Packt Publishing.

Robinson, I., Webber, James., & Eifrem, E. (2013). Graph databases (First edition.). O'Reilly Media, Inc.

Appendix A

A1. Code package:

Code package can be downloaded from Bitbucket:

1) mass_java_appl, QueryGraphDB branch:
   https://bitbucket.org/mass_application_developers/mass_java_appl/src/QueryGraphDB/
2) mass_java_core, QueryGraphDB bracnch:
   https://bitbucket.org/mass_library_developers/mass_java_core/src/QueryGraphDB/

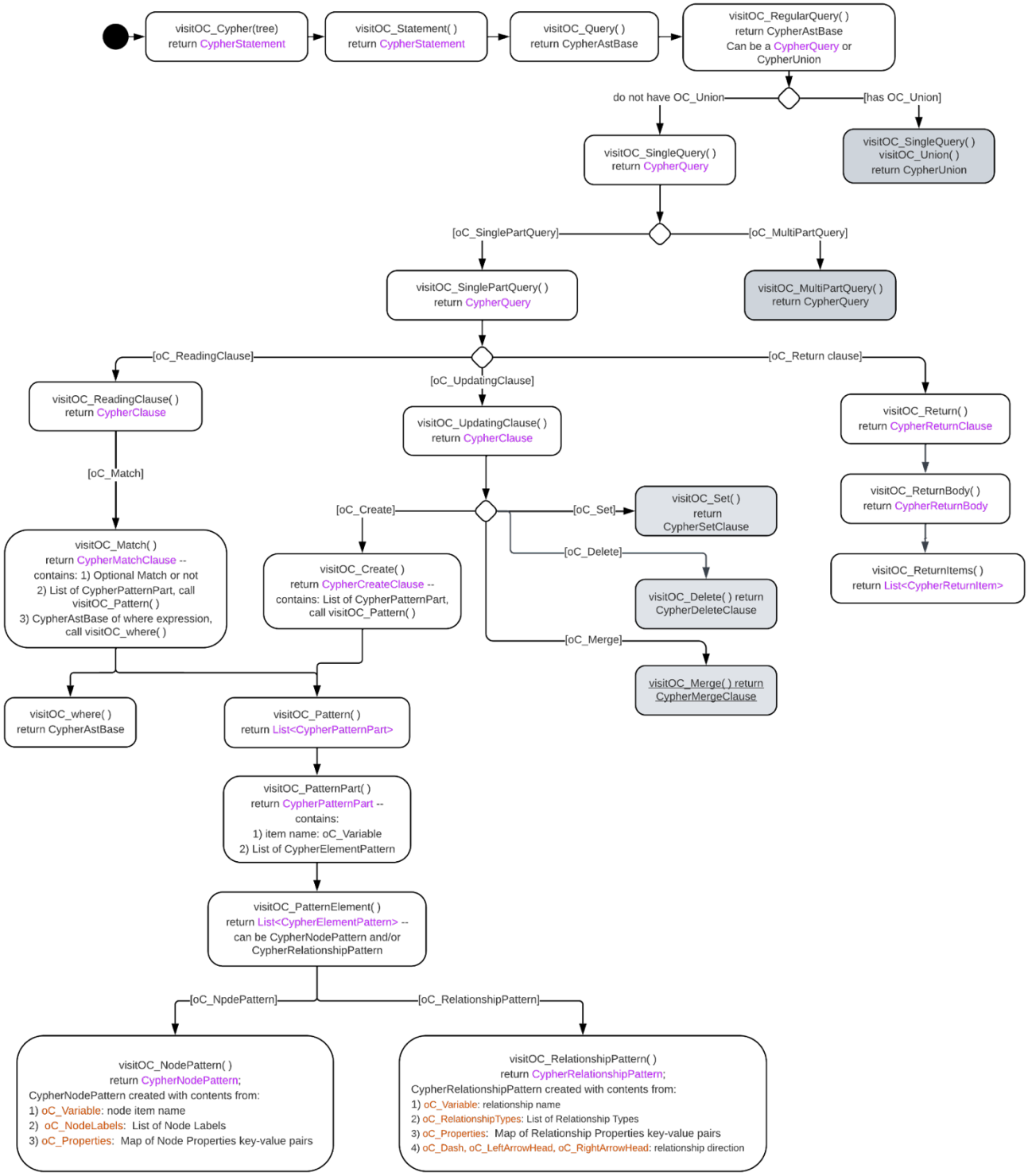
For more information of the files and folders containing the code that implemented in this project, it can be found at mass_java_appl, QueryGraphDB branch, QueryGraphDB folder, README.md file.

A2. Build and Run:

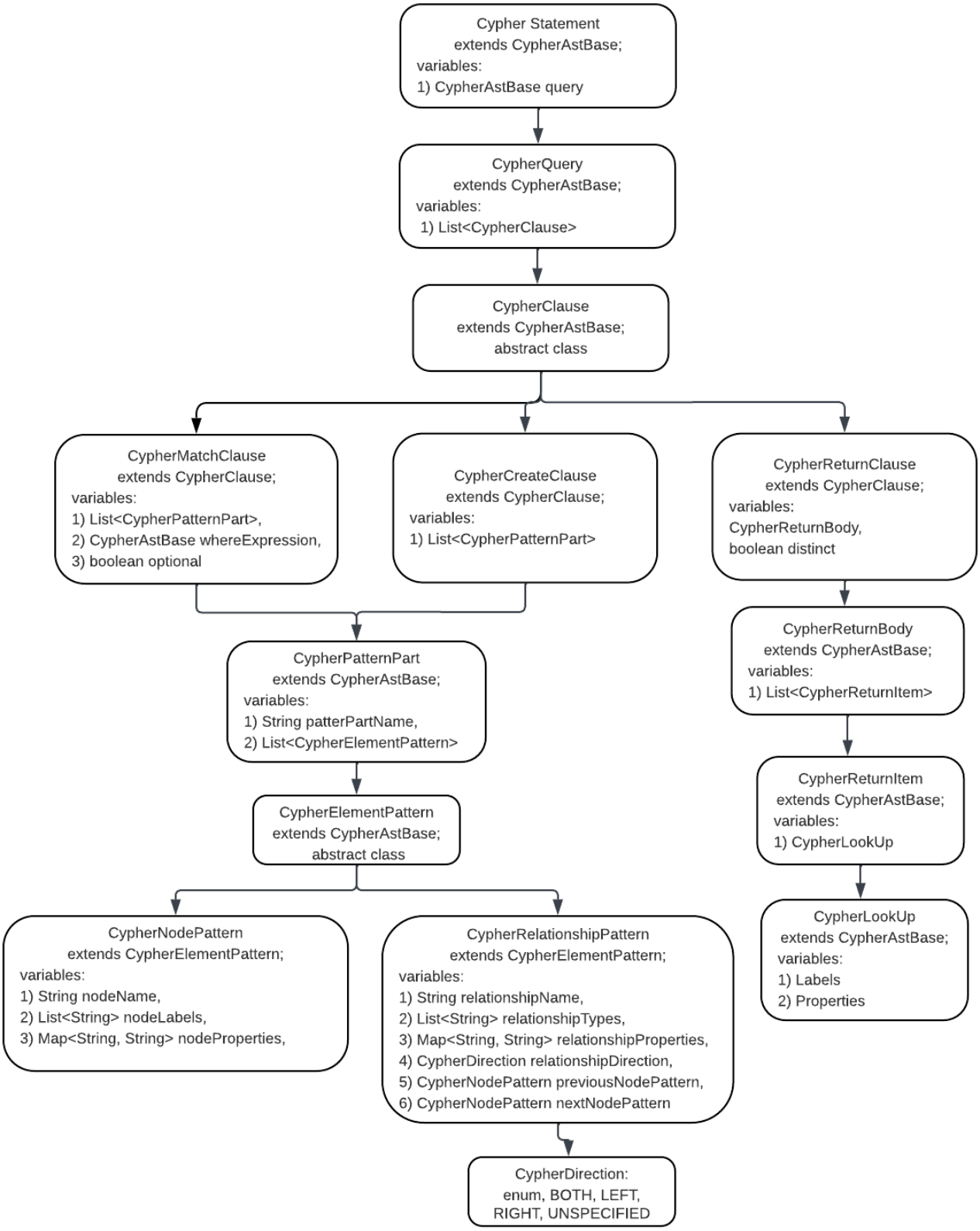
1. to make changes to MASS_java_core and rebuild, go to ‘~/mass_java_appl/QueryGraphDB’ folder and then run ‘sh build_mass.sh’
2. to rebuild and run the QueryGraphDB application, go to ‘~/mass_java_appl/QueryGraphDB’ folder and then run ‘sh build_run.sh’

Appendix B

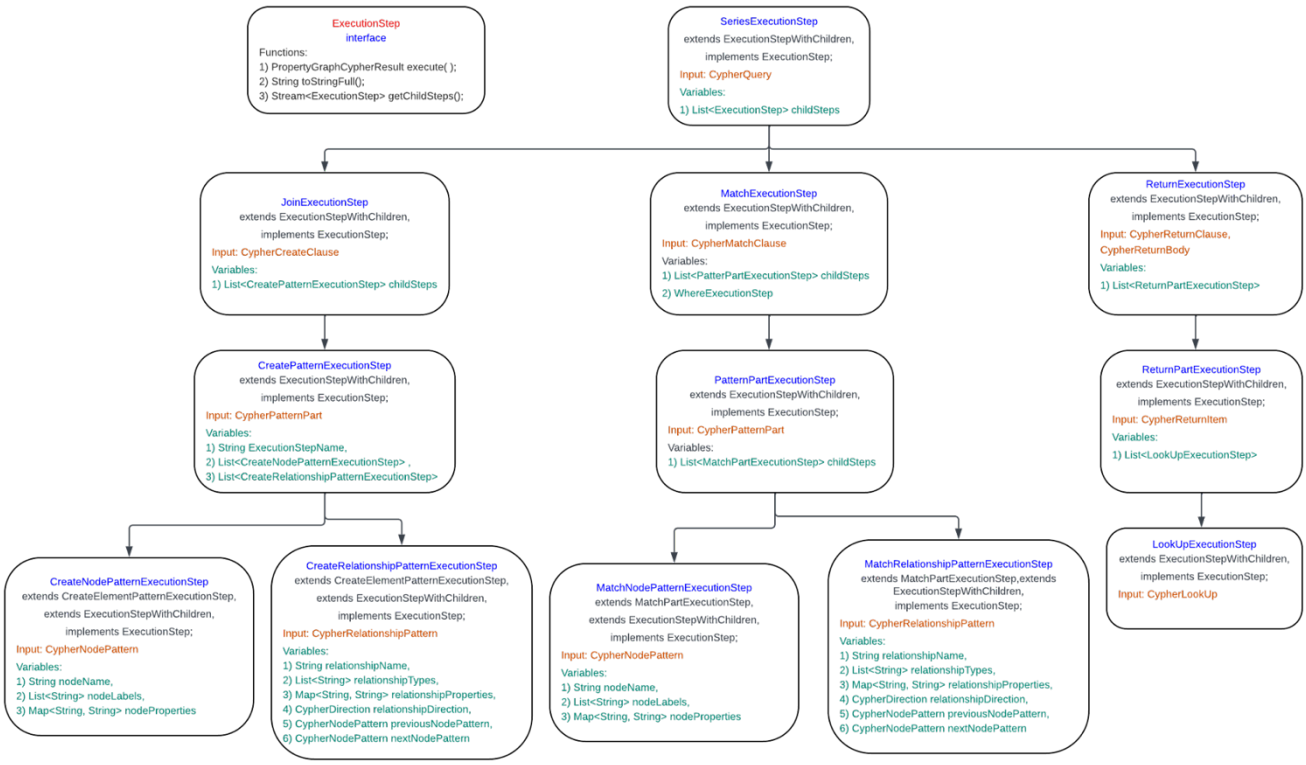
B1. Function flow of PropertyGraphCypherVisitor



B2. Abstract Syntax Tree (AST)



B3. ExecutionStep tree



Appendix C

C1. OpenCypher 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, differnt 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");
```

C2. Printed Graph after 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}
```

C3. 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
```

C4. Printed graph after the second 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}
```

C5. Printed result for Match clause on new 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
```