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 06, 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, Delete, Set, Return, With, and Where 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.

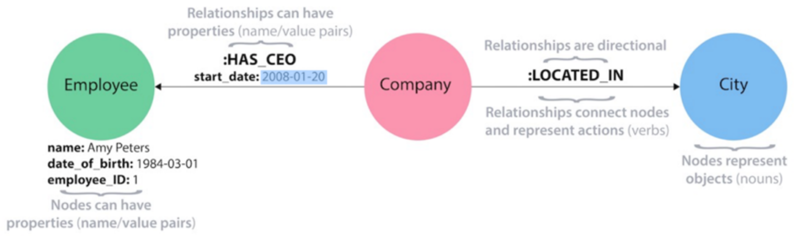

Relationships can have properties (name/value pairs)
:HAS_CEO
start_date: 2008-01-20
Employee
Company
Relationships are directional
:LOCATED_IN
City
Relationships connect nodes and represent actions (verbs)
name: Amy Peters
date_of_birth: 1984-03-01
employee_ID: 1
Nodes can have properties (name/value pairs)
Nodes represent objects (nouns)

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, Delete, Set, Return, With, and Where 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.
- SET: Modifies properties of nodes or relationships.
- RETURN: Specifies what data to include in the query results.
- WITH: Chains multiple clauses together, passing results from one to the next.
- WHERE: Filters the results based on specified conditions.

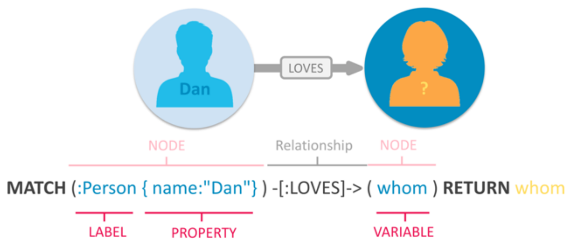

Dan
LOVES
?
NODE
Relationship
NODE
MATCH (:Person { name:"Dan"}) -[:LOVES]-> (whom) RETURN whom
LABEL
PROPERTY
VARIABLE

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.
  - The process flow of main program is shown in Figure 4.
  - 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.
  - The functions implemented in GraphManager are shown in Figure 5.
  - GraphManager 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.
- The underlying structures of the graph database are abstracted from users, ensuring they only need to be concerned with the high-level graph data model.
- PropertyGraphCypherQuery parses query text into an Abstract Syntax Tree (AST) using the Cypher Visitor mechanism and then uses ExecutionPlanBuilder's ExecutionPlan to build and execute the query statement stored in AST following various execution steps. In Figure 3, the CreateNodePatterExecutionStep is an example of execution step.

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

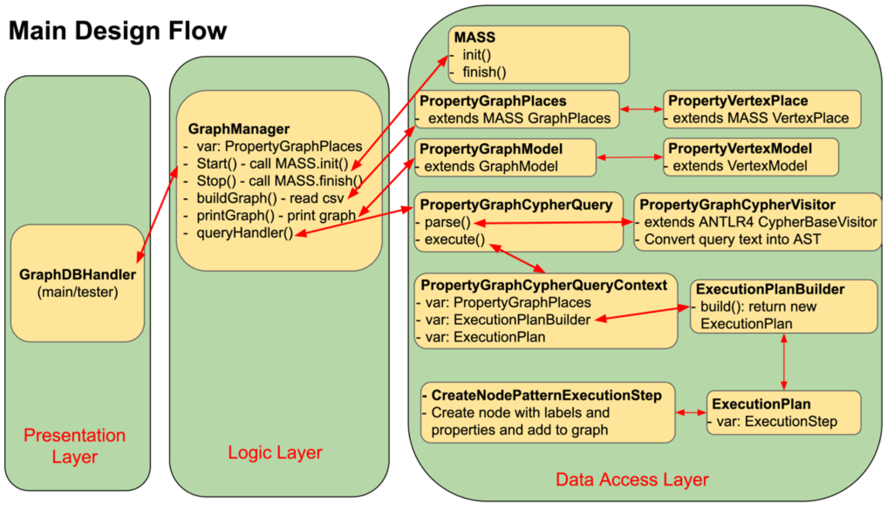

Main Design Flow
GraphDBHandler
(main/tester)
Presentation Layer
GraphManager
- var: PropertyGraphPlaces
- Start() - call MASS.init()
- Stop() - call MASS.finish()
- buildGraph() - read csv
- printGraph() - print graph
- queryHandler()
Logic Layer
MASS
- init()
- finish()
PropertyGraphPlaces
- extends MASS GraphPlaces
PropertyVertexPlace
- extends MASS VertexPlace
PropertyGraphModel
- extends GraphModel
PropertyVertexModel
- extends VertexModel
PropertyGraphCypherQuery
- parse()
- execute()
PropertyGraphCypherVisitor
- extends ANTLR4 CypherBaseVisitor
- Convert query text into AST
PropertyGraphCypherQueryContext
- var: PropertyGraphPlaces
- var: ExecutionPlanBuilder
- var: ExecutionPlan
ExecutionPlanBuilder
- build(): return new ExecutionPlan
- CreateNodePatternExecutionStep
- Create node with labels and properties and add to graph
ExecutionPlan
- var: ExecutionStep
Data Access Layer

Figure 3: Main design flow for the whole project.

GraphDBHandler.java - main/tester
1. Run main program with argument of csv node file name and edge file name.
2. In main()
    - Construct GraphManager object
    - Call GraphManager.start()
    - Upon successful start, then call GraphManager.buildGraph(String csvPropertyFileName, String csvRelationshipFileName)
    - Read OpenCypher query input from user
    - Call GraphManager to handle query instructions
    - Get results from GraphManger and print it out.
    - Keep on reading OpenCypher query till user typed exit.
    - Call GraphManager.stop() to stop the program.

Figure 4: Process flow in GraphDBHandler main program.

GraphManager.java
1. Constructor: GraphManager()
2. start(): init MASS
3. stop(): stop MASS
4. buildGraph(String csvPropertyFileName, String csvRelationshipFileName):
    i. Create new PropertyGraphPlaces object
    ii. Read csv node file contents
    iii. Call PropertyGraphPlaces to addVertex()
    iv. Call PropertyGraphPlaces to set vertex properties
    v. Read relationships csv file and call PropertyGraphPlaces to addRelationEdge with relationship Information

5. printGraph(): create property graph model with PropertyGraphModel and PropertyVertexModels, then print graph model

6. queryHandler(): call PropertyGraphCypherQuery to handle Cypher Query

Figure 5: Functions implemented in GraphManager.

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 6. 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.

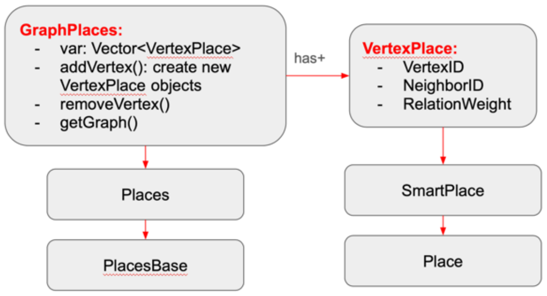


Figure 6: Existing MASS-based graph database framework based on MASS 1.4.3-SNAPSHOT version (develop branch).

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 and functions declared in PropertyVertexPlace and PropertyGraphPlaces are shown in Figure 7. In PropertyVertexPlace, the variable nodeProperties uses a map to store node property type-value pairs, while the variable relationProperties uses a map of map to store neighbor ID and its relationship property key-value pairs Map<neighbor node ID, Map<property type, property value>>. 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.

PropertyVertexPlace extends VertexPlace

```
private Map<Object, Object> nodeProperties = new HashMap<>();  // to store node properties
private Map<Object,Map<Object,Object>> relationProperties = new HashMap<>(); // to store relationship
properties

public void setProperties(Map<Object, Object> properties);
public Map<Object, Object> getProperties();
public Map<Object,Map<Object,Object>> getAllNeighborProperties();
public Map<Object,Object> getNeighborProperties(Object neighborId);
public void setNeighborProperties(Object neighborVertexId, Map<Object,Object> newRelationProperty);
```

PropertyGraphPlaces extends GraphPlaces

```
@Override
public boolean addVertex();

public boolean setProperties(Object id, Map<Object,Object> properties);
public Map<Object,Object> getProperties(Object id);
public boolean setRelationEdge(Object FromID, Object ToID,  Map<Object,Object>
relationProperties)
public Map<Object,Object> getRelationEdge(Object FromID, Object ToID)
```

Figure 7: Variables and functions declared and implemented in PropertyVertexPlace and PropertyGraphPlaces.

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 8. Query text is first parsed to AST structure, and then executed with execution plan, and finally get query results.

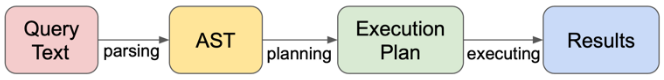
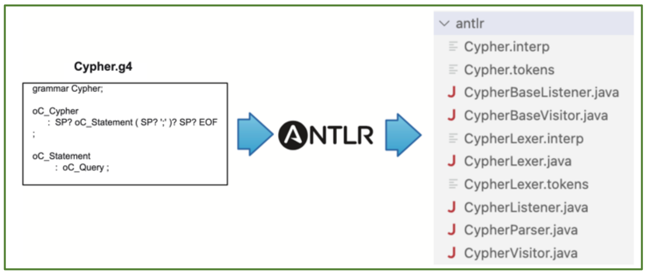


Figure 8: Query flow process.

3.3.1 Parser generator

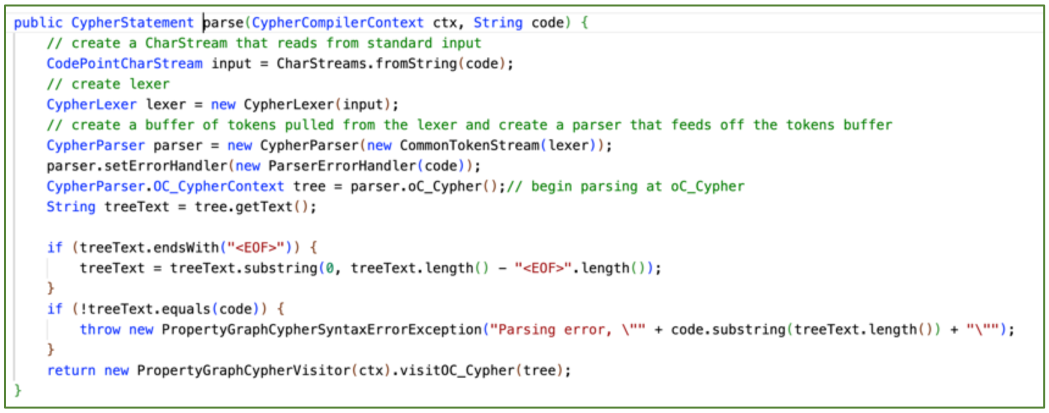
To seamlessly integrate Cypher with the MASS-based Property Graph Database, the initial and pivotal step is to select an adept language parser for translating Cypher queries into a Java-compatible format. Given the inherent complexity of the OpenCypher query grammar, the parsing strategy employed becomes 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 9. These code files are the base to parse query text to AST.



```
grammar Cypher;

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

oC_Statement
       :  oC_Query ;
```

Figure 9: 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 10 shows the code to parse query string and call PropertyGraphCypherVisitor to handler parser tree. PropertyGraphCypherVisitor extends CypherBaseVisitor, which implements CypherVisitor.

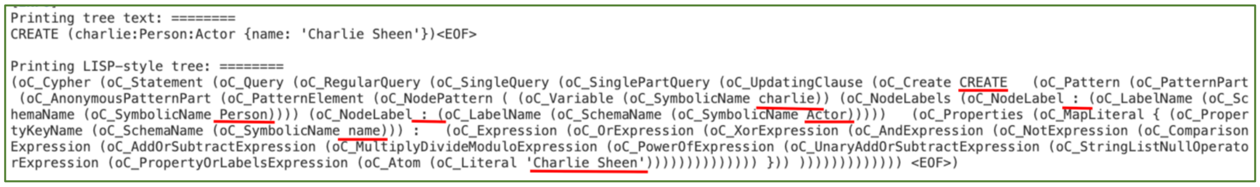


```java
public CypherStatement parse(CypherCompilerContext ctx, String code) {
    // create a CharStream that reads from standard input
    CodePointCharStream input = CharStreams.fromString(code);
    // create lexer
    CypherLexer lexer = new CypherLexer(input);
    // create a buffer of tokens pulled from the lexer and create a parser that feeds off the tokens buffer
    CypherParser parser = new CypherParser(new CommonTokenStream(lexer));
    parser.setErrorHandler(new ParserErrorHandler(code));
    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(code)) {
        throw new PropertyGraphCypherSyntaxErrorException("Parsing error, \"" + code.substring(treeText.length()) + "\"");
    }
    return new PropertyGraphCypherVisitor(ctx).visitOC_Cypher(tree);
}
```

Figure 10: Code to parse query string and handle with Cypher Visitor.

3.3.2 Parsing query text to AST

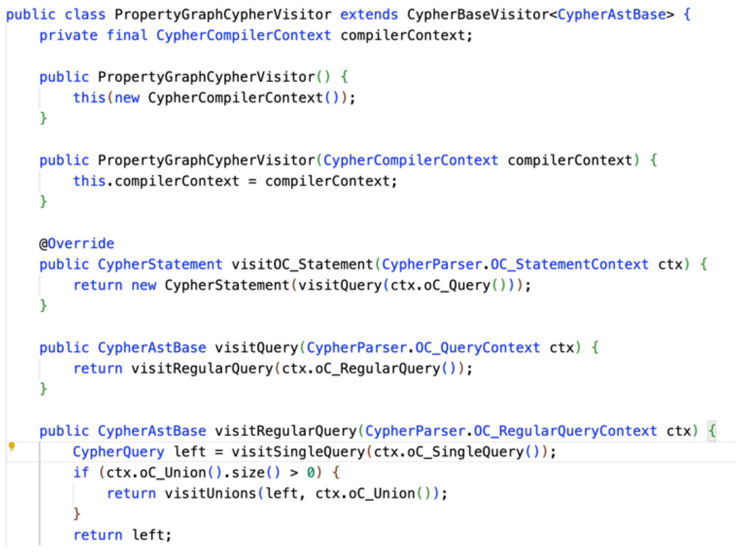
Cypher query parsing with ANTLR4 is the process of analyzing a sequence of symbols to determine its grammatical structure (Parr, 2013). This involves breaking down the input into its constituent parts according to the Cypher.g4 gramma file. Figure 11 Shows the grammatical structure for the query text 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 11: 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.



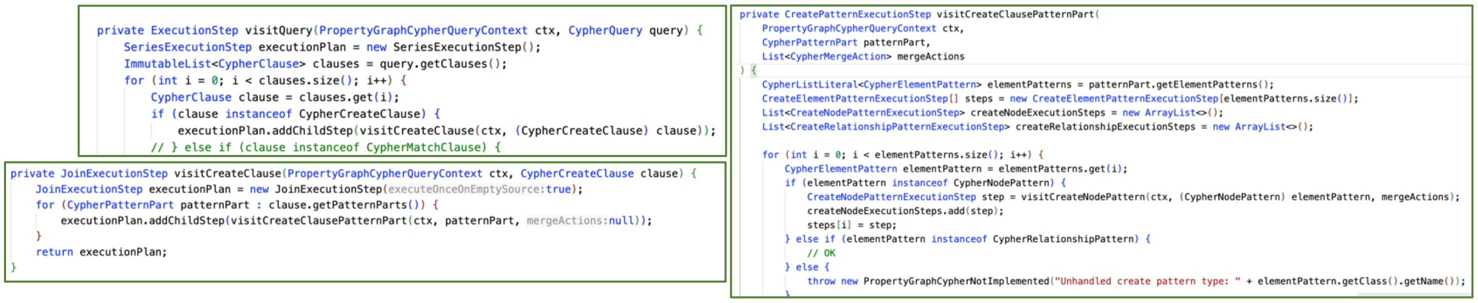
```java
public class PropertyGraphCypherVisitor extends CypherBaseVisitor<CypherAstBase> {
    private final CypherCompilerContext compilerContext;

    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 12: 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 12 shows partial implementation of the PropertyGraphCypherVisitor class.

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;
}
```

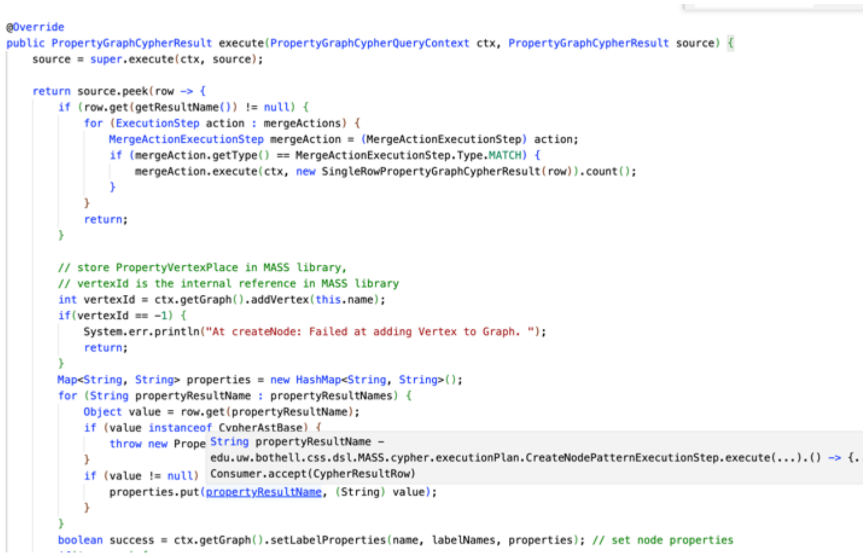
```java
private CreatePatternExecutionStep visitCreateClausePatternPart(
    PropertyGraphCypherQueryContext ctx,
    CypherPatternPart patternPart,
    List<CypherMergeAction> mergeActions
) {
    CypherListLiteral<CypherElementPattern> elementPatterns = patternPart.getElementPatterns();
    CreateElementPatternExecutionStep[] 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 13: The code to add execution steps to the execution plan.

Figure 13 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.

3.3.4 Execution to results

Figure 14 shows the code for execute() function of CreateNodePatternExecutionStep. Upon execute, it first invokes the superclass's execute method and then processes the result using the peek method. If the result already exists (row.get(getResultName()) != null), and merge action step is MATCH, then it executes merge actions of type MATCH. If the result does not exist, it creates a new node in the Property Graph Database, extracts properties from the Cypher query result, sets node properties, and prints information about the added vertex.



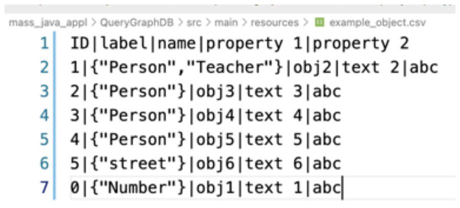
```java
@Override
public PropertyGraphCypherResult execute(PropertyGraphCypherQueryContext ctx, PropertyGraphCypherResult source) {
    source = super.execute(ctx, source);

    return source.peek(row -> {
        if (row.get(getResultName()) != null) {
            for (ExecutionStep action : mergeActions) {
                MergeActionExecutionStep mergeAction = (MergeActionExecutionStep) action;
                if (mergeAction.getType() == MergeActionExecutionStep.Type.MATCH) {
                    mergeAction.execute(ctx, new SingleRowPropertyGraphCypherResult(row)).count();
                }
            }
            return;
        }

        // store PropertyVertexPlace in MASS library,
        // vertexId is the internal reference in MASS library
        int vertexId = ctx.getGraph().addVertex(this.name);
        if(vertexId == -1) {
            System.err.println("At createNode: Failed at adding Vertex to Graph. ");
            return;
        }
        Map<String, String> properties = new HashMap<String, String>();
        for (String propertyResultName : propertyResultNames) {
            Object value = row.get(propertyResultName);
            if (value instanceof CypherAstBase) {
                throw new Prope String propertyResultName –
                                edu.uw.bothell.css.dsl.MASS.cypher.executionPlan.CreateNodePatternExecutionStep.execute(...).() -> {.
            }                   
            if (value != null)  Consumer.accept(CypherResultRow)
                properties.put(propertyResultName, (String) value);
            }
        }
        boolean success = ctx.getGraph().setLabelProperties(name, labelNames, properties); // set node properties
```

Figure 14: Sample code of execution step's execution() function.

4. Results

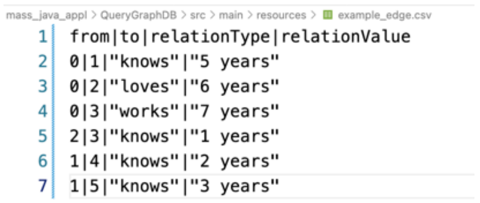
4.1 Execution and verification of building MASS-based Property Graph Database

The content of the input csv node file and edge file is shown in Figure 15 and Figure 16. After running the program with these two input files, the graph database was built and printed out as shown in Figure 17. The graph was printed out correctly based on the two input files.



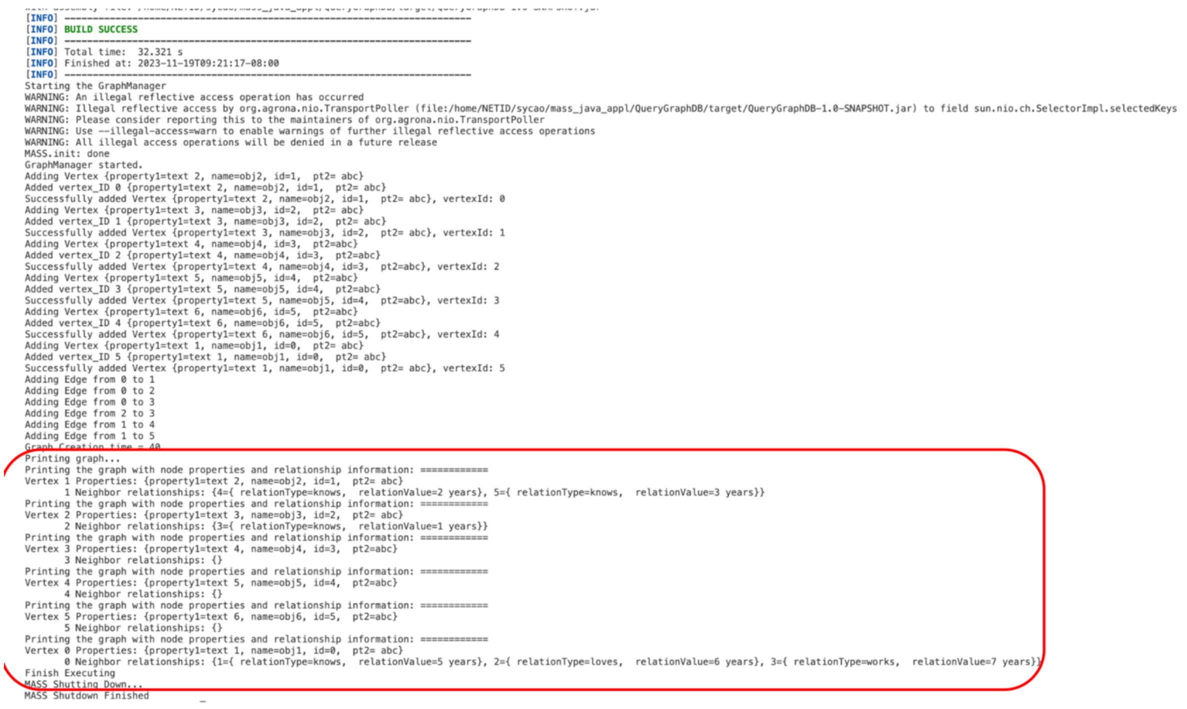
```
mass_java_appl > QueryGraphDB > src > main > resources > example_object.csv
ID|label|name|property 1|property 2
1|{"Person","Teacher"}|obj2|text 2|abc
2|{"Person"}|obj3|text 3|abc
3|{"Person"}|obj4|text 4|abc
4|{"Person"}|obj5|text 5|abc
5|{"street"}|obj6|text 6|abc
0|{"Number"}|obj1|text 1|abc
```

Figure 15: Node csv file.



```
mass_java_appl > QueryGraphDB > src > main > resources > example_edge.csv
from|to|relationType|relationValue
0|1|"knows"|"5 years"
0|2|"loves"|"6 years"
0|3|"works"|"7 years"
2|3|"knows"|"1 years"
1|4|"knows"|"2 years"
1|5|"knows"|"3 years"
```

Figure 16: Edge csv file.

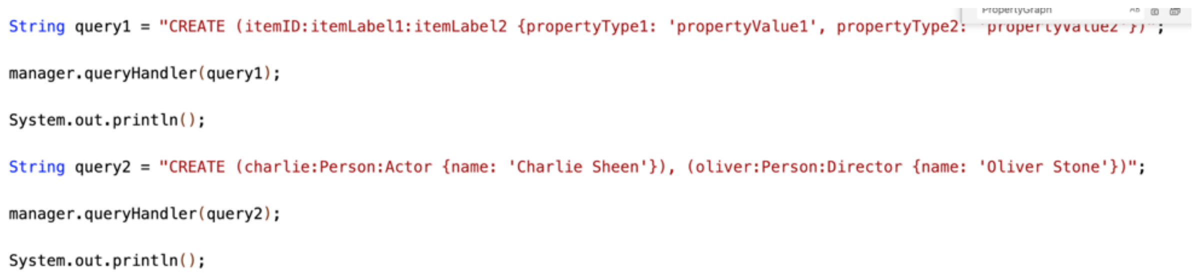


```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  32.321 s
[INFO] Finished at: 2023-11-19T09:21:17-08:00
[INFO] ------------------------------------------------------------------------
Starting the GraphManager
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller (file:/home/NETID/sycao/mass_java_appl/QueryGraphDB/target/QueryGraphDB-1.0-SNAPSHOT.jar) to field sun.nio.ch.SelectorImpl.selectedKeys
WARNING: Please consider reporting this to the maintainers of org.agrona.nio.TransportPoller
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
MASS.init: done
GraphManager started.
Adding Vertex {property1=text 2, name=obj2, id=1,  pt2= abc}
Added vertex_ID 0 {property1=text 2, name=obj2, id=1,  pt2= abc}
Successfully added Vertex {property1=text 2, name=obj2, id=1,  pt2= abc}, vertexId: 0
Adding Vertex {property1=text 3, name=obj3, id=2,  pt2= abc}
Added vertex_ID 1 {property1=text 3, name=obj3, id=2,  pt2= abc}
Successfully added Vertex {property1=text 3, name=obj3, id=2,  pt2= abc}, vertexId: 1
Adding Vertex {property1=text 4, name=obj4, id=3,  pt2=abc}
Added vertex_ID 2 {property1=text 4, name=obj4, id=3,  pt2=abc}
Successfully added Vertex {property1=text 4, name=obj4, id=3,  pt2=abc}, vertexId: 2
Adding Vertex {property1=text 5, name=obj5, id=4,  pt2=abc}
Added vertex_ID 3 {property1=text 5, name=obj5, id=4,  pt2=abc}
Successfully added Vertex {property1=text 5, name=obj5, id=4,  pt2=abc}, vertexId: 3
Adding Vertex {property1=text 6, name=obj6, id=5,  pt2=abc}
Added vertex_ID 4 {property1=text 6, name=obj6, id=5,  pt2=abc}
Successfully added Vertex {property1=text 6, name=obj6, id=5,  pt2=abc}, vertexId: 4
Adding Vertex {property1=text 1, name=obj1, id=0,  pt2= abc}
Added vertex_ID 5 {property1=text 1, name=obj1, id=0,  pt2= abc}
Successfully added Vertex {property1=text 1, name=obj1, id=0,  pt2= abc}, vertexId: 5
Adding Edge from 0 to 1
Adding Edge from 0 to 2
Adding Edge from 0 to 3
Adding Edge from 2 to 3
Adding Edge from 1 to 4
Adding Edge from 1 to 5
Graph Creation time = 40
Printing graph...
Printing the graph with node properties and relationship information: ============
Vertex 1 Properties: {property1=text 2, name=obj2, id=1,  pt2= abc}
       1 Neighbor relationships: {4={ relationType=knows,  relationValue=2 years}, 5={ relationType=knows,  relationValue=3 years}}
Printing the graph with node properties and relationship information: ============
Vertex 2 Properties: {property1=text 3, name=obj3, id=2,  pt2= abc}
       2 Neighbor relationships: {3={ relationType=knows,  relationValue=1 years}}
Printing the graph with node properties and relationship information: ============
Vertex 3 Properties: {property1=text 4, name=obj4, id=3,  pt2=abc}
       3 Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 4 Properties: {property1=text 5, name=obj5, id=4,  pt2=abc}
       4 Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 5 Properties: {property1=text 6, name=obj6, id=5,  pt2=abc}
       5 Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 0 Properties: {property1=text 1, name=obj1, id=0,  pt2= abc}
       0 Neighbor relationships: {1={ relationType=knows,  relationValue=5 years}, 2={ relationType=loves,  relationValue=6 years}, 3={ relationType=works,  relationValue=7 years}}
Finish Executing
MASS Shutting Down...
MASS Shutdown Finished
```

Figure 17: Print graph build from csv files.

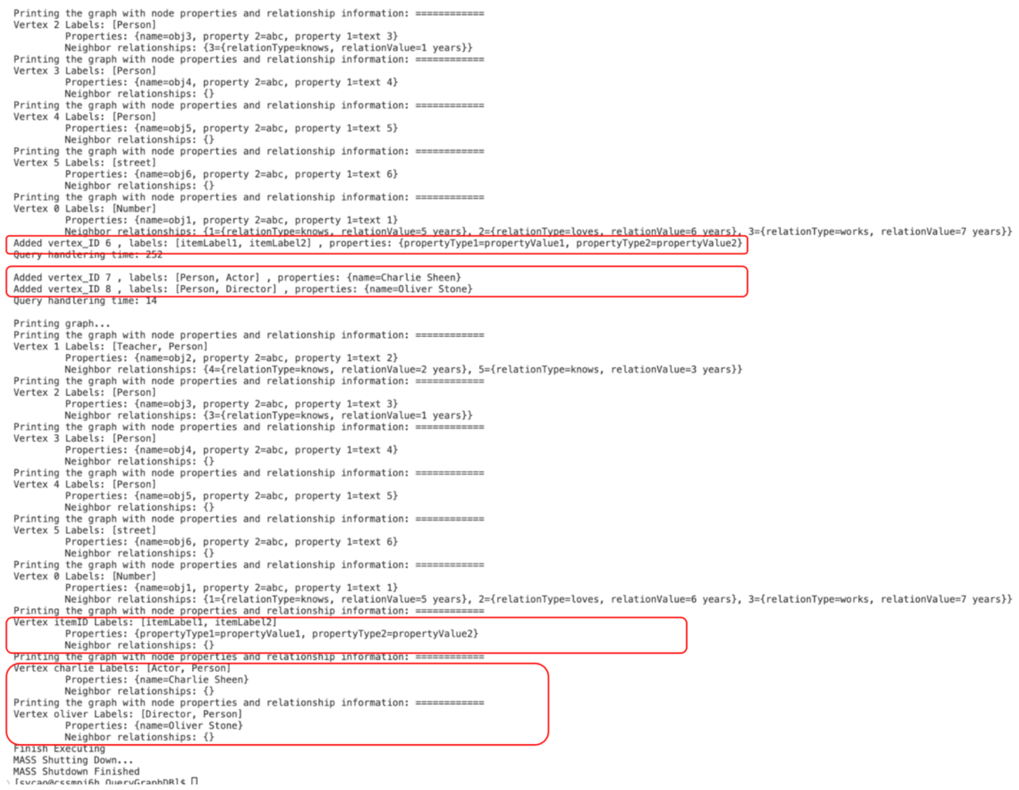
4.2 Execution and verification of executing the node creation query clauses.

Figure 18 shows the two node creation query clauses that we used for testing the system. Currently implementation is to integrate the node creation query clause into our MASS-based Property Graph Database system. The first tested query clause contains information for one new node, while the second tested query clause contains information for two new nodes. The updated graph was printed and shown in Figure 19. The parts highlighted in red rectangles are about the new nodes added the existing graph database, serving as confirmation of the accuracy of the implementation.



```
String query1 = "CREATE (itemID:itemLabel1:itemLabel2 {propertyType1: 'propertyValue1', propertyType2: 'propertyValue2'})";

manager.queryHandler(query1);

System.out.println();

String query2 = "CREATE (charlie:Person:Actor {name: 'Charlie Sheen'}), (oliver:Person:Director {name: 'Oliver Stone'})";

manager.queryHandler(query2);

System.out.println();
```

Figure 18: Node creation cypher query clauses in testing.



```
Printing the graph with node properties and relationship information: ============
Vertex 2 Labels: [Person]
         Properties: {name=obj3, property 2=abc, property 1=text 3}
         Neighbor relationships: {3={relationType=knows, relationValue=1 years}}
Printing the graph with node properties and relationship information: ============
Vertex 3 Labels: [Person]
         Properties: {name=obj4, property 2=abc, property 1=text 4}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 4 Labels: [Person]
         Properties: {name=obj5, property 2=abc, property 1=text 5}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 5 Labels: [street]
         Properties: {name=obj6, property 2=abc, property 1=text 6}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 0 Labels: [Number]
         Properties: {name=obj1, property 2=abc, property 1=text 1}
         Neighbor relationships: {1={relationType=knows, relationValue=5 years}, 2={relationType=loves, relationValue=6 years}, 3={relationType=works, relationValue=7 years}}
Added vertex_ID 6 , labels: [itemLabel1, itemLabel2] , properties: {propertyType1=propertyValue1, propertyType2=propertyValue2}
Query handlering time: 252

Added vertex_ID 7 , labels: [Person, Actor] , properties: {name=Charlie Sheen}
Added vertex_ID 8 , labels: [Person, Director] , properties: {name=Oliver Stone}
Query handlering time: 14

Printing graph...
Printing the graph with node properties and relationship information: ============
Vertex 1 Labels: [Teacher, Person]
         Properties: {name=obj2, property 2=abc, property 1=text 2}
         Neighbor relationships: {4={relationType=knows, relationValue=2 years}, 5={relationType=knows, relationValue=3 years}}
Printing the graph with node properties and relationship information: ============
Vertex 2 Labels: [Person]
         Properties: {name=obj3, property 2=abc, property 1=text 3}
         Neighbor relationships: {3={relationType=knows, relationValue=1 years}}
Printing the graph with node properties and relationship information: ============
Vertex 3 Labels: [Person]
         Properties: {name=obj4, property 2=abc, property 1=text 4}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 4 Labels: [Person]
         Properties: {name=obj5, property 2=abc, property 1=text 5}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 5 Labels: [street]
         Properties: {name=obj6, property 2=abc, property 1=text 6}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex 0 Labels: [Number]
         Properties: {name=obj1, property 2=abc, property 1=text 1}
         Neighbor relationships: {1={relationType=knows, relationValue=5 years}, 2={relationType=loves, relationValue=6 years}, 3={relationType=works, relationValue=7 years}}
Printing the graph with node properties and relationship information: ============
Vertex itemID Labels: [itemLabel1, itemLabel2]
         Properties: {propertyType1=propertyValue1, propertyType2=propertyValue2}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex charlie Labels: [Actor, Person]
         Properties: {name=Charlie Sheen}
         Neighbor relationships: {}
Printing the graph with node properties and relationship information: ============
Vertex oliver Labels: [Director, Person]
         Properties: {name=Oliver Stone}
         Neighbor relationships: {}
Finish Executing
MASS Shutting Down...
MASS Shutdown Finished
[sycao@cssmpi6h QueryGraphDB]$ 
```

Figure 19: Print graph after executing node creation cypher query clauses.

5. Winter Quarter Project Plan

Timeline	Tasks
Winter 2024	
Week 1 (Jan 3 – Jan 7)	Implement Node Match query clause and Return query clause.
Week 2 & 3 (Jan 8 – Jan 21)	Implement Node Match query clause with With and Where query clauses.
Week 4 & 5 (Jan 22 – Feb 4)	Implement Relationship Create query clause.
Week 6 & 7 (Feb 5 – Feb 18)	Implement Relationship Match query clause and Return query clause.
Week 8 & 9 (Feb 19 – Mar 3)	Implement Relationship Match query clause with With and Where query clauses.
Week 10 & 11(Mar 4 – Mar 17)	Implement Node and Relationship Set query clauses.

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 node queries accurately. Future tasks are outlined for further development during Winter Quarter. More query 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

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'
```