Analysis of Declarative-to-Imperative Cypher Query Translations in MASS - A Distributed Graph Query Framework

Term Report of Work Done for
Autumn Quarter 2025

By Ryan Isaacson

Master of Science in Computer Science & Software Engineering

University of Washington
December 10th, 2025

Project Committee:
Professor Munehiro Fukuda, Committee Chair
Professor Wooyoung Kim, Committee Member
Professor Dong Si, Committee Member

1 Introduction

The Distributed Systems Laboratory (DSL) at the University of Washington Bothell has long pursued the vision of scalable agent-based computation across large-scale graph networks to handle real-world problem solving. In order to address a multitude of parallelization challenges left unaddressed from various shared-memory programming environments, the DSL began creation of the parallelization library known as the Multi-Agent Spatial Simulation (MASS) in 2010 [1]. Multi-agent parallelized systems are useful to analyze emergent properties in complex systems that involve entities (agents) operating concurrently with independent sets of instructions that each agent carries with it [2]. Applications leveraging such novel techniques for parallelization seek to prioritize scalability, locality, and decentralized control which effectively allow for exceptionally efficient computation speeds as datasets and their affiliated properties grow in number.

While MASS is simultaneously expanding its Agent-Based Modeling (ABM) methodologies to support C++ and CUDA operations, the work done here will focus primarily on the Java-based library which allows autonomous agents to spawn, move, interact, and compute operations across a distributed memory cluster. Each variant of the library is intended for more nuanced purposes. The CUDA library is intended for ultra-high performance in massively parallel ABM simulations. The C++ library is purposed for large-scale ABM simulations that include operations that CUDA cannot perform. Finally, the Java-specific library is tailored to handle interactive computation on distributed data structures and while preferring optimized parallel performance on large data sets, it is not explicitly dedicated to ultra-high performance compared C++ or CUDA.

In order to practicalize computation based on distributed data structures, the MASS library is applied toward its own graph database. The MASS library supports input queries similar to other industry-standard graph databases like Neo4j and ArangoDB by utilizing declarative input query grammar inspired by Neo4j's Cypher query language [3]. In essence, declarative query grammar rules allow a user to input simple and understandable statements describing "what" they are looking for, as opposed to imperative methods which explicitly include the step-by-step details for "how" the database will create, acquire, or manipulate the specified data [4]. MASS aims to support a familiar declarative input syntax for ease of use by the querier that is thereafter translated into imperative instruction sets to deploy agents and execute in parallel across the compute node cluster autonomously.

Prior work in recent years for the DSL has implemented several core Cypher clause functionalities [5] along with other criticalities for a distributed graph network. Michelle Dea worked on identifying similar databases for performance comparison and benchmarking [6, 7]. Shenyan (Lilian) Cao provided support for the MATCH clause which specifies the patterns to search for within the database [7, 8]. Yuan (Chris) Ma focused on upgrading the graph storage layer to be distributed and support multi-user access [7, 9]. Aatman Prajapati afterward implemented key functionality for adding filtering constraints to the MATCH clause with 2 underlying variants of the WHERE clause [10, 11]. These important additions allow for search, filtering logic, and data return of the graph network but the ability to update vertex nodes and their respective relationship edges through updating clauses such as DELETE and UPDATE remain to be implemented.

Ensuring these update functionalities are addressed would allow for expanded capability of the MASS library while allowing for a more comprehensive analysis of its translation pipeline which highlights declarative query input to imperative parallelized agent execution. To determine MASS's framework and methodologies as viable in the greater academic and industry space, it is called for to perform an examination of similar queries across industry-standard alternatives such as Neo4j which leverages underlying usage of the RAFT consensus protocol [12, 13, 14] and ArangoDB which uses alternative ACID (Atomicity, Consistency, Isolation, Durability) compliant single-node transaction protocols [14, 15]. The MASS framework currently prioritizes scalability and distributed local agent-execution over that of strict global coordination and consistency methods. While current work on DELETE intends to focus on data availability and query processing progress, future work in MASS may explore extensions to incorporate fault-tolerance or consensus mechanisms similar to Neo4j and ArangoDB.

It is the primary objective of this research to implement new functionality to the MASS library through functioning update clauses such as DELETE and SET. Then it will be prudent to compare the functionality of these implemented clauses against similar queries in Neo4j and ArangoDB in order to illustrate the comparative operations between MASS and commercially deployed alternatives. An analysis will need to be performed in order to showcase whether MASS stands ahead or behind its competitor's for ABM simulations within graph databases. Furthermore, an implementation of a CALL clause which would allow for pre-defined methods and subqueries to be deployed within a query would be both helpful and fully encompassing.

2 Background & Motivation

2.1 Graph Databases vs. Relational Databases

Before assessing MASS and its competing database frameworks, we can look at why they each utilize an underlying property graph architecture. Traditional relational databases have long served as the backbone of data management; however, their performance efficiency begins to degrade in highly connected datasets like social networks and scientific simulations such as biological interaction networks. One of the primary challenges regarding efficiency in relational databases are due to heavy reliance on join operations, which are used to reconstruct relationships by matching keys across multiple data tables. While join operations can be effective in very particularly structured datasets, they become increasingly expensive as data complexity and connectivity grow. Each join operation requires scanning, hashing, and sorting large tables to establish links and pairing. As a result, work involving multi-hop graph traversals can incur a substantial computational overhead and usage of available memory. With this in mind, graph databases can be better suited for these highly connected data types. Asplundh and Sandell [17] showed that with semi-structured and densely linked data that graph databases outperformed relational databases. Furthermore, Jain, Khanchandani, and Rodrigues analyzed Neo4j as a representative of graph databases and MySQL as a representative of relational databases and found that the graph database outperformed the relational database by up to 146 times when querying complex and large datasets.

2.2 Challenges in Graph Databases

In graph databases by storing relationship edges as first-class architectural components, traversal across vertex nodes can be performed without the use of join operations. This advantage has made platforms like Neo4j and ArangoDB popular for applications involving social networks, biological interaction networks, recommendation systems, and large-scale knowledge graphs. While these systems provide efficient graph network centered data models that use declarative query languages on the front-end, they still rely on centralized or tightly coordinated transaction protocols such as Neo4j's use of RAFT consensus for cluster replication, or ArangoDB's single-leader transaction manager to maintain consistency [12-15]. While both of these coordination methods are essential for ACID compliance and correctness, they can impose synchronization overhead that can hinder scalability in massively parallel or compute-intensive environments.

2.3 Agent-Based Solutions

This inefficiency of a heavy communication overhead is one of the primary motivating factors for the implementation of the MASS parallelization library. MASS aims to avoid monolithic database architecture to specialize in highly connected and large-scale dataset handling. Rather than focusing on a coordination-heavy transactional process, MASS's architecture relies on swarms of lightweight agents executing imperative instructional logic autonomously across the distributed memory space. Through utilizing control of targeted agent deployment and minimizing global communicative synchronization, MASS aims to exploit fine-grained parallelism, decentralized decision-making, and dynamic adaptability. As seen in table 1 below, MASS's architecture development stands to place parallelism and scalability as primary goals and optimizations for implementation.

Feature	Neo4j	ArangoDB	MASS
Language	Cypher (Declarative)	AQL (Declarative + Imperative)	Cypher Declarative-to-Imperative + agents
Execution	Centralized	Centralized + Hybrid	Fully Distributed
Model	Property Graph	Property Graph/ Multi-Model	Property Graph
Parallelism	Low	Medium -> High	High
Scalability	Medium	Medium -> High	High

Table 1: High-Level Comparison of Neo4j, ArangoDB, & MASS

Like Neo4j, MASS will support and mimick a declarative query interface that uses Cypher syntax for CREATE, MATCH, WHERE, and soon DELETE, SET, and CALL clauses. After translating the declarative query into an imperative instruction set called an execution plan, distribution of agents permits MASS to combine the simple usability of modern graph networks while leveraging the gains of parallel performance in High-Performance Computing (HPC) oriented simulation frameworks. Work to extend MASS's ability to handle updating clauses is not simply a necessity for being a feature-complete library, but also essential for understanding and comparing how a highly-parallelized agent-based architecture can compare with industry alternatives for both design and ultimately execution performance.

3 Previous Work

Shenyen Cao previously implemented work for CREATE and MATCH clause functionality which enabled pattern matching across the distributed property graph. In her whitepaper [8] Cao designed mechanisms for translating Cypher-like declarative patterns into executable agent-based searches, particularly focusing on label matching, variable bindings, and edge traversal logic across vertex nodes. This allowed the system to interpret user queries in the Cypher form MATCH (a)-[r]->(b) which shows a vertex node a and b with a directionally outbound relationship pointing from vertex node a to vertex node b. The framework then systematically deploys agents that explore graph neighborhoods in a parallelized fashion and then collects intermediate resultant data between execution step phases that satisfy the matching query conditions.

Cao's work was further solidified in a joint publication presented at KNBigData 2024, coauthored with Michelle Dea, Yuan Ma, and Professor Munehiro Fukuda. In that paper, Cao's MATCH clause functionality was demonstrated as an essential stage in building an end-to-end pipeline from Cypher query input to agent execution. This contribution established the earliest executable components of MASS's declarative-to-imperative pipeline model.

Building directly on Cao's MATCH clause functionality, Aatman Prajapati added support for the WHERE clause to enable constraint filtering logic. His implementation accounted for multiple variants of logical expressions such as boolean comparison operators. Aatman notably designed and created a stack-based parsing algorithm to evaluate nested logical expressions by decomposing the WHERE clause into postfix notation which is otherwise known as Reverse Polish Notation. His initial work enabled efficient expression evaluation during agent execution.

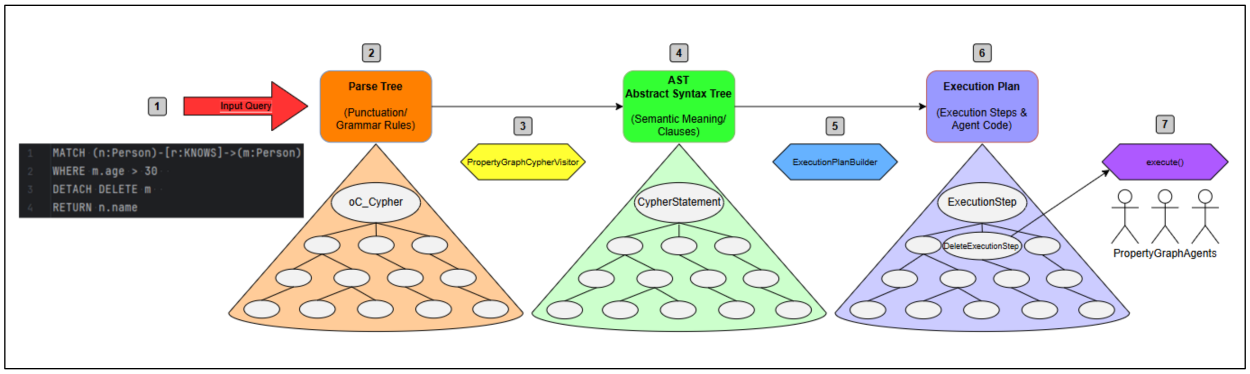
After developing the stack-based algorithm, Prajapati extended his work to implement an optimized tree-based filtering system, which constructed an abstract syntax tree (AST) from the WHERE clause and traversed it recursively to evaluate expressions. The tree-based approach which leveraged a significantly improved three-value logic system allows for the dynamic pruning of incorrect paths for agent traversal, which reduced the number of active agents and therefore lead to less overall computational overhead [11]. This critical work by Prajapati constrains the overall query search and importantly focuses computation to intermediately return vertex information that successfully matches the provided query.

The 2024 KNBigData publication by Ma, Cao, Dea, and Fukuda collectively framed the MASS framework's foundational architecture as a declarative-to-imperative pipeline, with each researcher improving critical functionality. Their paper introduced a multi-stage framework that included query parsing, AST generation, execution planning, and agent deployment. This broad architecture set the context for future expansion to support updating clauses like DELETE and SET, which remain to be fully implemented. The authors discussed future possibilities of enabling deletion via a hypothetical PropertyGraphPlaces.deleteVertex() method, although that functionality was not implemented at the time of the papers writing.

Cao's whitepaper also briefly discussed design concerns for updating clauses which acknowledged that supporting deletion in a distributed context would require careful handling of concurrency and data consistency during agent operations [8]. These insights lay the conceptual groundwork for the job being undertaken in this report for implementing DELETE clause functionality. This thesis will involve adding accurate query actions of DELETE and SET functionalities to the graph database. To do this will require consideration for accurate and up-to-date data representation in vertex nodes and their relationship edge pairings among agents in local neighborhoods while also controlling for mutual exclusion such that agents will not work on the same vertex node simultaneously. A careful consideration of methods to craft this functionality will need to be considered accordingly to ensure correctness and avoid race conditions. A synchronization strategy will need to be articulated that will prevent multiple agents from concurrently modifying the same vertex or its edges. Once an agent enters a vertex node to perform a deletion, other agents must be temporarily blocked from access. Either atomic operations or a locking mechanism will be critical to maintain data integrity and ensure accurate updates across the graph network.

4 Implementation

To understand and contribute to the MASS framework, an investigation into its declarative-to-imperative underlying translation pipeline was a necessity. As was stated previously, MASS uses a multi-stage pipeline to transform declarative Cypher-style queries into a parallel agent-based execution plan. The process is illustrated in Figure 1 below, and has been broken down into 7 key stages.



```
MATCH (n:Person)-[r:KNOWS]->(m:Person)
WHERE m.age > 30
DETACH DELETE m
RETURN n.name
```

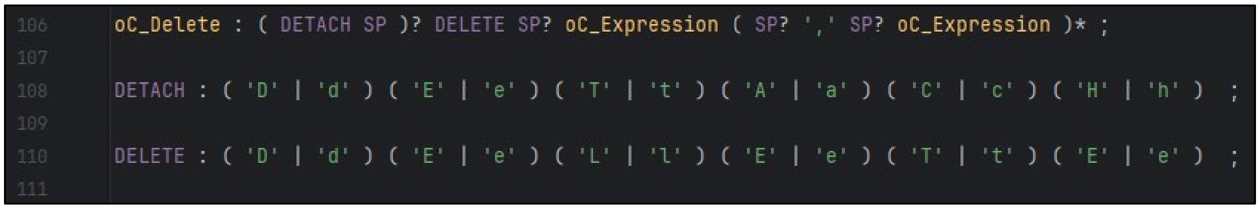
Figure 1: Query Translation Pipeline from Input to Execution

Stage 1: Input Query

A user at the application layer submits a declarative Cypher-style query such as the example MATCH (n:Person)-[r:KNOWS]->(m:Person) WHERE m.age > 30 DETACH DELETE m RETURN n.name. Parenthesis surround vertex nodes while square brackets surround relationship edges and arrow notation showcases directionality of the relationship. This notation describes the declarative "what" the querier is looking for but leaves it to MASS for "how" that imperative logic will be orchestrated under-the-hood.

Stage 2: Parse Tree Creation

MASS in it's current form utilizes ANTLR4 (ANother Tool for Language Recognition version 4) to break down the query into lexical components and is parsed correlating to specific Cypher language grammar rules. What is generated is a parse tree, otherwise known as a concrete syntax tree, which includes each and every token of the original input including punctuation, carriage line returns, space characters, and special markers including but not limited to the End of File (EOF). This phase ensures that the query was entered in a legitimately recognized format for Cypher statements and that there are no errors. An example formatted rule for what a DELETE entry's syntax must abide by is seen in Figure 2.



```
oC_Delete : ( DETACH SP )? DELETE SP? oC_Expression ( SP? ',' SP? oC_Expression )* ;

DETACH : ( 'D' | 'd' ) ( 'E' | 'e' ) ( 'T' | 't' ) ( 'A' | 'a' ) ( 'C' | 'c' ) ( 'H' | 'h' )  ;

DELETE : ( 'D' | 'd' ) ( 'E' | 'e' ) ( 'L' | 'l' ) ( 'E' | 'e' ) ( 'T' | 't' ) ( 'E' | 'e' )  ;
```

Figure 2: Cypher.g4 File Grammar Rule for DELETE

Each node in the parse tree is correlated to a specially formatted Cypher rule and the root node is the oC_Cypher rule, where “oC” in these rules stands for the “openCypher” standard. This accepted format for “oC” rules is an open-source project that helps spread the correct format of the Cypher language in each respective graph database system [18]. Each token will establish a node in the parse tree, and while it can seem cluttered, the tree will hold the accurate tokenized version of the entire representation of the query. Figure 3 below is a visualized representation of a sample parse tree highlighting the DELETE clause associated specifics within the red circle.

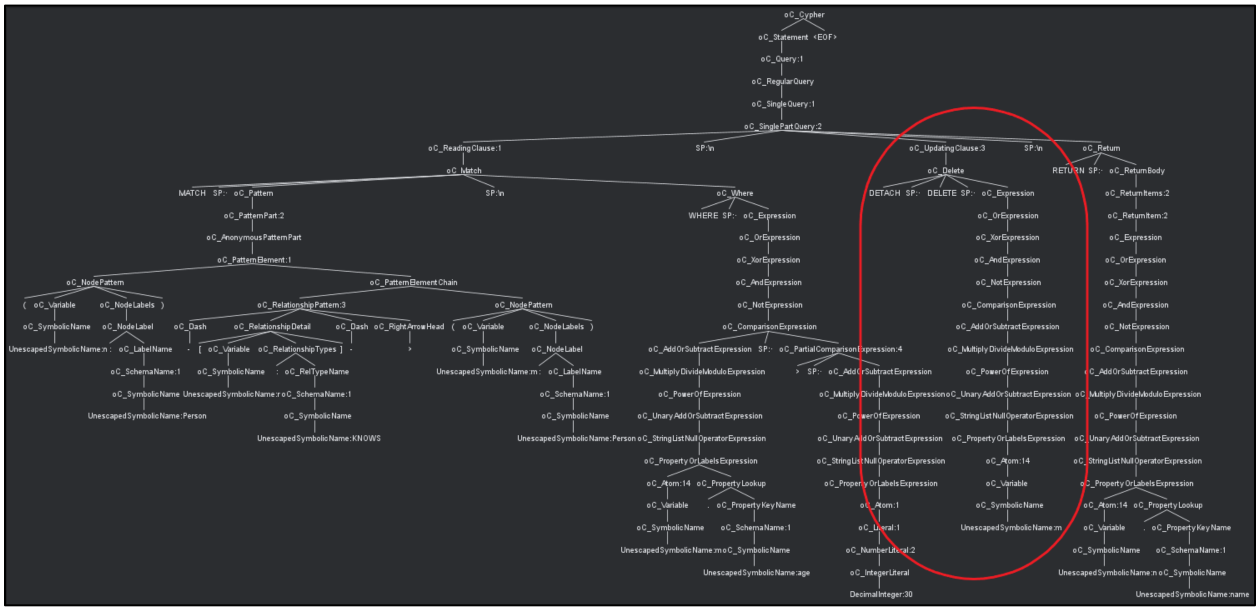
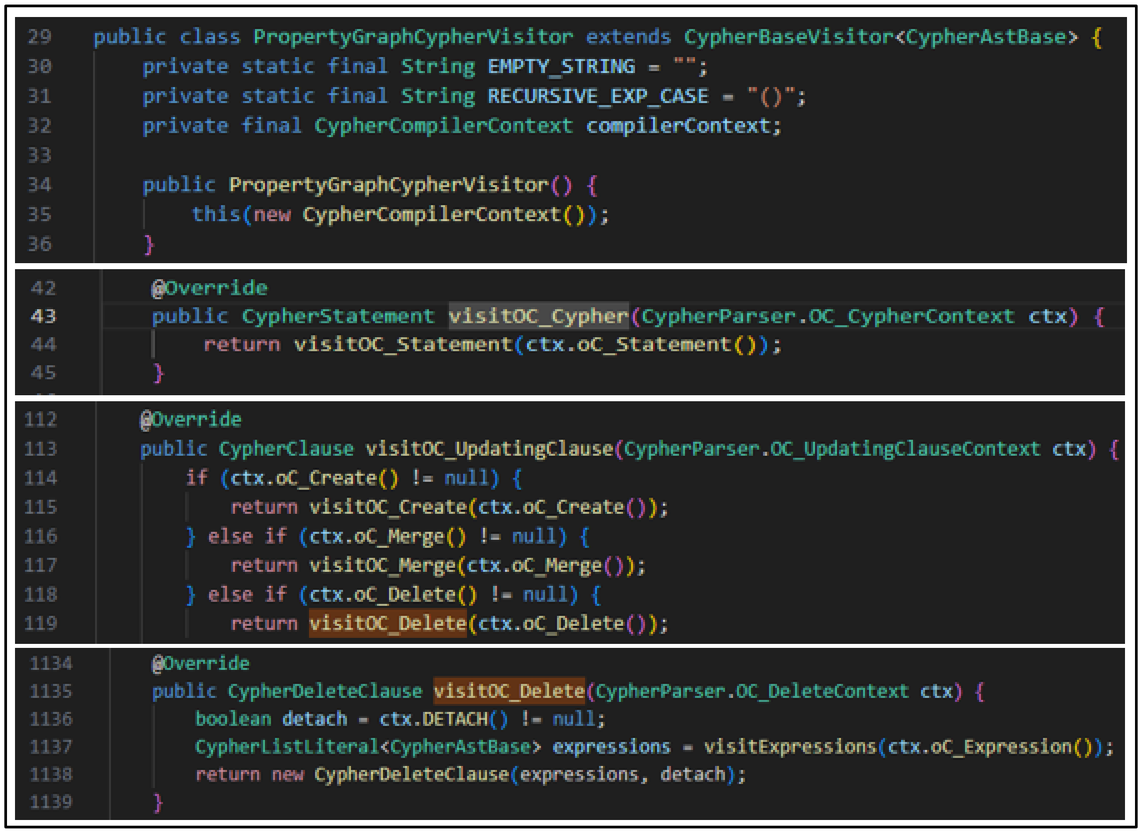


Figure 3: Parse Tree (Concrete Syntax Tree) with DELETE Branch Circled

Stage 3: PropertyGraphCypherVisitor Traverses Parse Tree

The PropertyGraphCypherVisitor (PGVC) class implements ANTLR’s visitor interface. It traverses the parse tree that was created in stage 2 recursively and builds the abstract syntax tree (AST). The visitor will ignore nodes related to superficial grammar and extracts the semantic meaning of the original query. This can be thought of as trimming out the excess grammatical components to get at the heart of the query meaning. The PGVC carries with it the “ctx” or context of the parse tree branch that remains to be recursively traversed for that section of the tree. Figure 4 below shows the PGVC constructor with an example visitation to its root node of oC_Cypher in addition to a visitation of an updating clause of which DELETE is one of. When visiting these rule encounters, logic may dictate to visit an alternative local method, but the end result will be the return of a newly generated type of AST node such as the CypherDeleteClause.



```
public class PropertyGraphCypherVisitor extends CypherBaseVisitor<CypherAstBase> {
    private static final String EMPTY_STRING = "";
    private static final String RECURSIVE_EXP_CASE = "()";
    private final CypherCompilerContext compilerContext;

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

```
    @Override
    public CypherStatement visitOC_Cypher(CypherParser.OC_CypherContext ctx) {
        return visitOC_Statement(ctx.oC_Statement());
    }
```

```
    @Override
    public CypherClause visitOC_UpdatingClause(CypherParser.OC_UpdatingClauseContext ctx) {
        if (ctx.oC_Create() != null) {
            return visitOC_Create(ctx.oC_Create());
        } else if (ctx.oC_Merge() != null) {
            return visitOC_Merge(ctx.oC_Merge());
        } else if (ctx.oC_Delete() != null) {
            return visitOC_Delete(ctx.oC_Delete());
```

```
    @Override
    public CypherDeleteClause visitOC_Delete(CypherParser.OC_DeleteContext ctx) {
        boolean detach = ctx.DETACH() != null;
        CypherListLiteral<CypherAstBase> expressions = visitExpressions(ctx.oC_Expression());
        return new CypherDeleteClause(expressions, detach);
    }
```

Figure 4: PropertyGraphCypherVisitor Methods Return AST Nodes

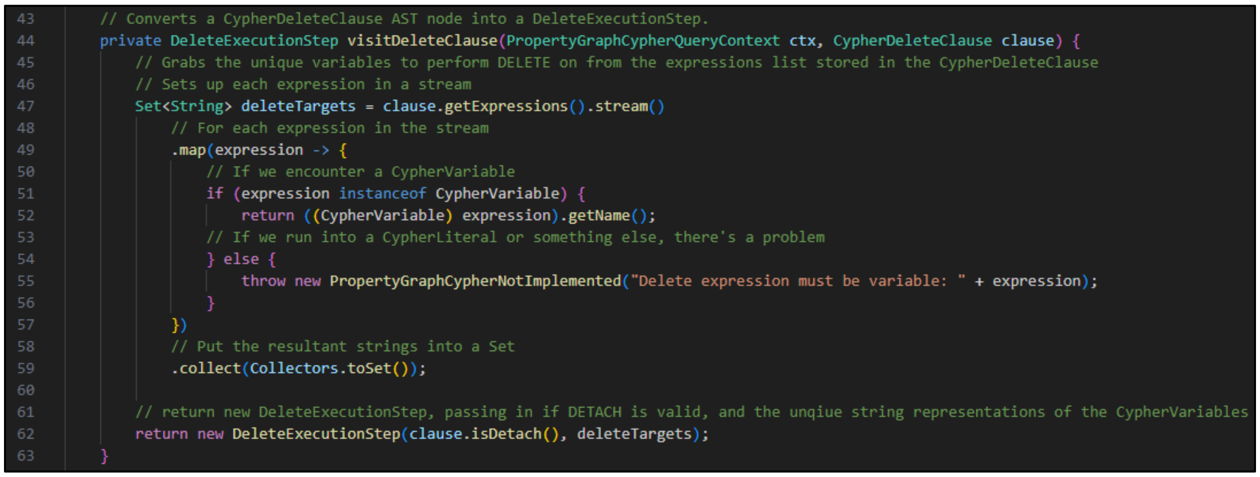
Stage 4: Abstract Syntax Tree

The AST holds the core semantic context of the original query. At its root is the CypherStatement class which extends the CypherAstBase class. This CypherStatement class holds children that are the other AST clause nodes generated via the PGVC parse tree traversal. The CypherStatement lives within the PropertyGraphCypherQuery wrapper class which acts as a higher-level controller for building and executing the corresponding execution plan. The purpose of these AST nodes are to simplify each clause concept into more easily translatable operations for later execution stages.

Stage 5: ExecutionPlanBuilder Traverses AST

Within the PropertyGraphCypherQueryVisitorContext wrapper class, the ExecutionPlanBuilder is created, generates a new ExecutionPlan, and then traverses the AST producing a resultant ExecutionStep node in the ExecutionPlan corresponding to the clause node being visited in the AST. The following code in figure 5 was drafted in the ExecutionPlanBuilder visitation of the

CypherDeleteClause in the AST. The ctx argument holds the shared state of the actual property graph, a carrying forward of the ExecutionPlanBuilder itself, as well as the ExecutionPlan being operated on and constructed. It specifies the collection of the variable expressions queried for as a part of the Cypher rule shown previously in figure 2.



```java
    // Converts a CypherDeleteClause AST node into a DeleteExecutionStep.
    private DeleteExecutionStep visitDeleteClause(PropertyGraphCypherQueryContext ctx, CypherDeleteClause clause) {
        // Grabs the unique variables to perform DELETE on from the expressions list stored in the CypherDeleteClause
        // Sets up each expression in a stream
        Set<String> deleteTargets = clause.getExpressions().stream()
            // For each expression in the stream
            .map(expression -> {
                // If we encounter a CypherVariable
                if (expression instanceof CypherVariable) {
                    return ((CypherVariable) expression).getName();
                // If we run into a CypherLiteral or something else, there's a problem
                } else {
                    throw new PropertyGraphCypherNotImplemented("Delete expression must be variable: " + expression);
                }
            })
            // Put the resultant strings into a Set
            .collect(Collectors.toSet());

        // return new DeleteExecutionStep, passing in if DETACH is valid, and the unqiue string representations of the CypherVariables
        return new DeleteExecutionStep(clause.isDetach(), deleteTargets);
    }
```

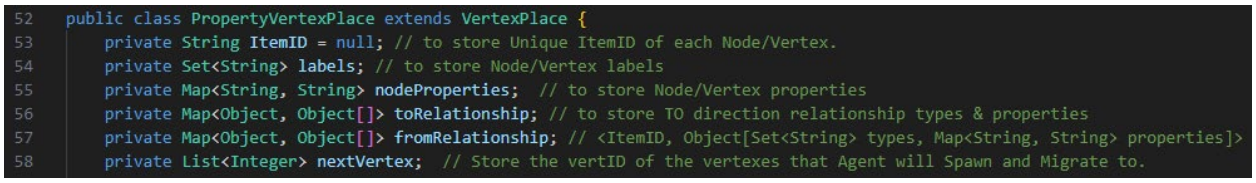
Figure 5: ExecutionPlanBuilder Method to Create DeleteExecutionStep

Stage 6: ExecutionPlan

The root of the ExecutionPlan is a base node type of ExecutionStep which is a non-instantiable interface class. Multiple types of ExecutionSteps are derived from this base class. Notably the DefaultExecutionStep is an abstract class used for leaf nodes that don't have children while the ExecutionStepWithChildren class obviously has child nodes respectively. A subtype of the ExecutionStepWithChildren class is the concrete class SeriesExecutionStep which is generated when the children steps necessarily need to run in series. For example, the matching and filtering of MATCH and WHERE clause stages need to complete before DELETE and RETURN logic occurs, and therefore the root node would likely implement a SeriesExecutionStep for that reason. Data is passed through these steps as a PropertyGraphCypherResult object which effectively allows for intermediate collections of data that can then be transferred between stages and then modified or extended.

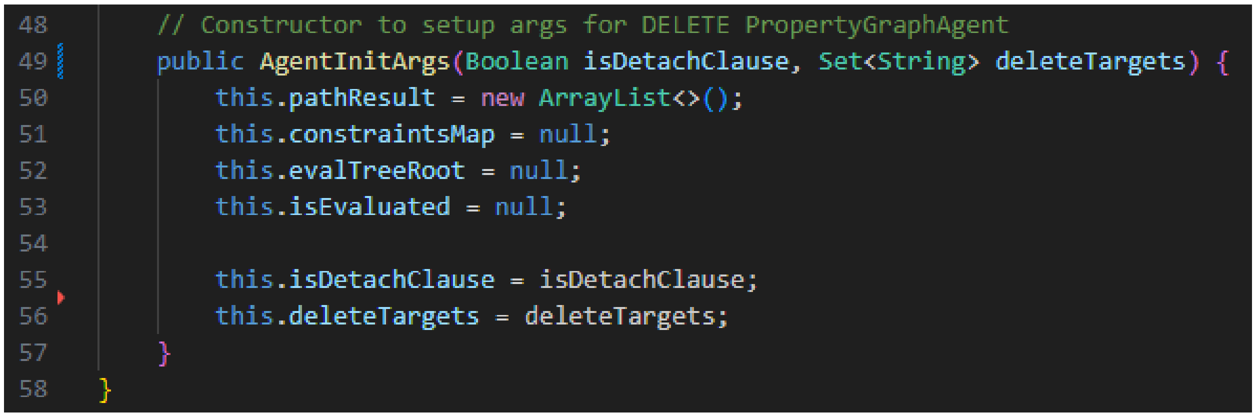
Stage 7: PropertyGraphAgents

PropertyGraphAgents are distributed across the shared memory of the property graph which is partitioned across a compute node cluster. Depending on the type of ExecutionStep that would currently be executing, a different variant group of the PropertyGraphAgent class would be deployed. The PropertyGraphAgent has different motives to complete when deployed for a DELETE operation compared to MATCH or WHERE and would rely on a different make-up of the AgentInitArgs class which is handed to it. In DELETE ideally the AgentInitArgs is setup for containing the node ID's (ItemID) of the PropertyVertexPlace's that have been traveled to in the MATCH/WHERE phase through the intermediate returnPath result.



```java
public class PropertyVertexPlace extends VertexPlace {
    private String ItemID = null; // to store Unique ItemID of each Node/Vertex.
    private Set<String> labels; // to store Node/Vertex labels
    private Map<String, String> nodeProperties;  // to store Node/Vertex 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;  // Store the vertID of the vertexes that Agent will Spawn and Migrate to.
```

Figure 6: PropertyVertexPlace shows a Unique ID Available for All Vertex Nodes



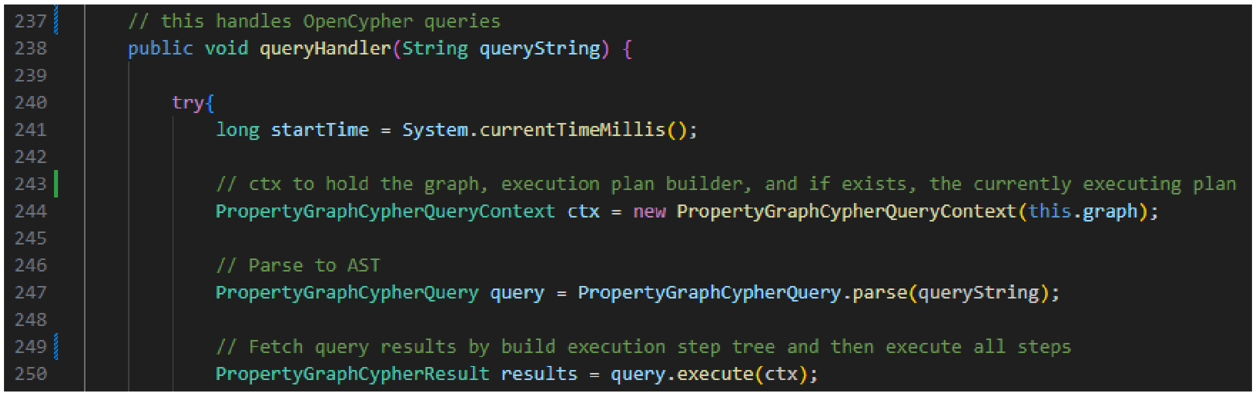
```java
    // Constructor to setup args for DELETE PropertyGraphAgent
    public AgentInitArgs(Boolean isDetachClause, Set<String> deleteTargets) {
        this.pathResult = new ArrayList<>();
        this.constraintsMap = null;
        this.evalTreeRoot = null;
        this.isEvaluated = null;

        this.isDetachClause = isDetachClause;
        this.deleteTargets = deleteTargets;
    }
}
```

Figure 7: AgentInitArgs Class for DELETE PropertyGraphAgents Will Carry Target ID's to Perform DELETE Work

Stages 2-7: Declarative-To-Imperative Pipeline

To sum up the pipeline, in the GraphManager class outside of the mass_java_core engine on the application side of the framework, the GraphManager.queryHandler function effectively begins the chain reaction of the declarative-to-imperative pipeline. In figure 8 below, line 244 encompasses stages 2-4 which creates a new context to carry the PropertyGraphPlaces graph, the ExecutionPlanBuilder, and the current form of the ExecutionPlan. Line 247 encompasses stages 5-7 which then return the result of the query which is formatted as rows of information defined in the PropertyGraphCypherResult class.



```java
237     // this handles OpenCypher queries
238     public void queryHandler(String queryString) {
239 
240         try{
241             long startTime = System.currentTimeMillis();
242 
243             // ctx to hold the graph, execution plan builder, and if exists, the currently executing plan
244             PropertyGraphCypherQueryContext ctx = new PropertyGraphCypherQueryContext(this.graph);
245 
246             // Parse to AST
247             PropertyGraphCypherQuery query = PropertyGraphCypherQuery.parse(queryString);
248 
249             // Fetch query results by build execution step tree and then execute all steps
250             PropertyGraphCypherResult results = query.execute(ctx);
```

Figure 8: GraphManager queryHandler Core Function Aspects

5 Evaluation

In order to evaluate the DELETE clause implementation in MASS, there will be reuse of three successfully utilized datasets previously by Prajapati [10, 11] to test execution time for speed as well as memory usage to monitor overall thread communication workloads in closely identically property graph network environments. The datasets are stored as CSV files containing vertex nodes and relationship edges which can be uploaded to MASS thanks to the prior work by Dea [6] and Cao [8]. The datasets scale across 10^3 for the IMDB movie dataset and 10^4 for the BioSnap and Shipments datasets. The BioSnap dataset is a smaller scale biomedical dataset while the Shipments dataset is a synthetically crafted dataset by Prajapati for larger scale that fits the needs of the testing environment with greater vertex node and relationship quantities.

Dataset	Vertex Nodes	Relationship Edges	Node Magnitude	Relationship Magnitude	Fabrication
IMDB	1,097 nodes	8,130 edges	10^3	10^3	Non-synthetic
BioSNAP	10,825 nodes	174,978 edges	10^4	10^5	Non-synthetic
Shipments	30,000 nodes	40,000 edges	10^4	10^4	Synthetic

Figure 9: Three Datasets - IMDB, BioSNAP, & Shipments

While a dataset representing 10^5 has not yet been selected, the SNAP Datasets via Stanford Large Network Dataset Collection [19] stands as an excellent repository to search through for data groupings that can accommodate the kind of densely linked structure at the quantity of scale necessary to put the MASS framework to the test. Finding a suitable dataset in the 10^5 order of magnitude will allow for a more comprehensive scalability test for update clause operations such as DELETE and SET respectively. After ensuring correct functionality, the use of initial test data will allow for the algorithm to be updated as necessary and finally directly compared to the identical datasets in Neo4j and ArangoDB for execution speed and memory overhead comparisons.

6 Conclusion

There are still core aspects of the DELETE clause pipeline that need to be implemented, but the Autumn 2025 quarter has set off to a great start. A large amount of research time was needed on the MASS declarative-to-imperative pipeline initially to assess how it's designed and how it functions. This research enabled the depth of understanding of the complex framework and layers of translation and interdependencies between classes that are required to make knowledgeable edits and contributions to the MASS library.

This project has deepened my understanding of distributed systems, query processing pipelines, and the architectural complexity behind multiple stages of query translations. Through extensive exploration of the MASS framework, I've learned how MASS Cypher queries are parsed, translated into abstract syntax trees, and ultimately executed via a parallel agent distribution plan. Gaining this foundational insight has been crucial in preparing me to further implement and plan for update clauses in a way that will accurately perform the task at hand, while respecting the complexities of concurrent operations at scale. The challenges faced so far have seemed aggressively present, but in knowledgeable reflection appear surmountable.

The next step for Winter Quarter 2026 will be definitive code addition and implementation initially for the completion of singular vertex node deletion. To do that correctly an accuracy inspection of the deletion target ItemID's passed from the MATCH/WHERE phase to the DELETE phase during the intermediate data transfer will be mandatory. Furthermore, the conceptual design will need to be drafted for a more complicated DELETE execution. This can involve multiple hops or agent traversal across vertex nodes where agents may encounter concurrency issues for acting on the same vertex node. An algorithm for confirming exclusive rights to operate on that node will be necessary. Lastly, the underlying theme in developing an algorithm will refrain from costly global consensus measures. Instead, the aim will be to necessarily support high resource availability through localized decision-making for responding to individual agent requests for data acces. This methodology should ensure that the progress for query processing is not hindered and is instead systematically resolved.

7 **References**

[1] DSLab, "MASS: A Parallelizing Library for Multi-Agent Spatial Simulation," University of Washington Bothell, [Online]. Available: https://depts.washington.edu/dslab/MASS/

[2] Distributed System Laboratory, "Distributed Systems Laboratory (DSL)", University of Washington Bothell, [Online]. Available: https://depts.washington.edu/dslab/

[3] Neo4j, "Cypher – Graph Query Language", Neo4j Inc, [Online]. Available: https://neo4j.com/product/cypher-graph-query-language/

[4] Neo4j, "Cypher and GQL: Imperative vs Declarative Query Languages", Neo4j, [Online]. Available: https://neo4j.com/blog/cypher-and-gql/imperative-vs-declarative-query-languages/

[5] Neo4j Documentation, "Clauses — Cypher Manual," Accessed on: July 20, 2025 [Online]. Available: https://neo4j.com/docs/cypher-manual/current/clauses/

[6] M. Dea. (2024). Implementing CREATE Clause in a Distributed Graph Database Using the MASS Library. White Paper, Distributed Systems Lab, University of Washington Bothell. Available: https://depts.washington.edu/dslab/MASS/reports/MichelleDea_whitepaper.pdf

[7] Y. Ma, M. Dea, S. Cao. & M. Fukuda. (2024). Toward Implementing an Agent-based Distributed Graph Database System. In Proceedings of the IEEE International Conference on Knowledge and Big Data (KNBigData). Available:

[8] S. Cao, *"An Incremental Enhancement of Agent-Based Graph Database System"*, UW Bothell Distributed Systems Lab, 2024. Accessed on: July 20, 2025 [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/LilianCao_whitepaper.pdf

[9] Y. Ma, An Implementation of Multi-User Distributed Shared Graph, White Paper (M.S. capstone), University of Washington, Bothell, 2024. Available: https://depts.washington.edu/dslab/MASS/reports/ChrisMa_whitepaper.pdf

[10] A. R. Prajapati, *Adding WHERE Clause Support to the MASS Graph Query Engine*, UW Bothell Distributed Systems Lab, Spring 2025. Accessed on: July 20, 2025 [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/AatmanPrajapati_sp25.pdf

[11] A.R. Prajapati, An Enhancement of Distributed Graph Queries in an Agent-Based Graph Database, M.S. Thesis, University of Washington Bothell, 2025. Available: https://depts.washington.edu/dslab/MASS/reports/AatmanPrajapati_thesis.pdf

[12] A. Bowman, "Understanding Causal Cluster Size Scaling", Neo4j Knowledge Base, [Online]. Available: https://neo4j.com/developer/kb/understanding-causal-cluster-size-scaling/

[13] The Secret Lives of Data, "Raft", [Online]. Available: https://thesecretlivesofdata.com/raft/

[14] M. Fukuda, "CSR: RUI: An Agent-Based Graph Database System", University of Washington Bothell, Bothell, WA, Internal Grant Proposal, Unpublished, 2025.

[15] ArangoDB GmbH, "Feature List of the ArangoDB Core Database System", ArangoDB Documentation, 2025. [Online]. Available: https://docs.arango.ai/arangodb/stable/features/list/

[16] C. Rodrigues, M. R. Jain, and A. Khanchandani, "Performance Comparison of Graph Database and Relational Database", May 2023, doi: 10.13140/RG.2.2.27380.32641. Available: https://www.researchgate.net/publication/370751317_Performance_Comparison_of_Graph_Database_and_Relational_Database

[17] A. Asplund and R. Sandell, "Comparison of graph databases and relational databases performance" B.S. Thesis, Stockholm University, Sweden, 2023. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1784349/FULLTEXT01.pdf

[18] OpenCypher, "OpenCypher – An Open Source Project", [Online]. Available: https://opencypher.org/

[19] J. Leskovec and A.Krevl, "SNAP Datasets", Stanford University, [Online]. Available: https://snap.stanford.edu/data/

8 Appendix

Committed work for DELETE clause can be found in the **ryanmi/ryan-delete-clause** branch of mass_java_core.

https://bitbucket.org/mass_library_developers/mass_java_core/branch/ryanmi/ryan-delete-clause