Graph Mutation in a Distributed Multi-Agent Spatial Simulation

Term Report of Work Done for
Spring Quarter 2026

By Ryan Isaacson

Master of Science in Computer Science & Software Engineering

University of Washington
June 12th, 2026

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

1. Introduction

This thesis work involving the Multi-Agent Spatial Simulation (MASS) library aims to accomplish a two-part incremental enhancement of the MASS-based graph database (DB) system. First, the implementation of write operation functionality to the agent-based system is paramount. This will introduce new graph mutation features that were not previously available within the system such as the ability to delete vertices, detether edges between them, as well as set new values for the attributes they contain. Then, after this functionality has been implemented, a comparison of these new features and their relative efficiencies can be compared to industry standard and pioneering alternative graph DB systems such as Neo4j and ArangoDB. This work will functionally implement and then analyze the underlying differences between the MASS-based declarative-to-imperative query translation process and compare results to industry leading competitors.

The previous Winter 2026 quarter established the baseline operations for DELETE and DETACH DELETE of vertices successfully within the MASS graph database, however there remained a gap in the codebase which prevented support for surgical deletion of an edge or relationship specified in the original user-generated query itself. This was because while the work of the previous quarter enabled graph mutation via deletion of vertex node entities, the underlying query pipeline still lacked a mechanism for preserving relationship identity through the previous MATCH and WHERE execution phases. As a result, relationships could be traversed by agents during pattern matching but could not be directly returned, manipulated, or deleted in later execution stages.

As each clause-affiliated phase of the query completes execution, the resulting data is packaged and transferred via `CypherResultRow` objects to later execution stages. In order to enable direct relationship mutation in subsequent phases, it became necessary to modify the declarative-to-imperative query pipeline. How the pipeline moved data forward would need to be altered to retain contextual relationship information. Prior to this work, the traversal information generated during MATCH and WHERE execution primarily consisted of vertex-oriented path data but no edge-oriented data, which resulted in the loss of relationship-specific context once traversal completed.

To address this limitation, the Spring 2026 quarter focused on refactoring the internal representation of graph relationships and redesigning how traversal information is preserved throughout the query pipeline. Relationships that were previously represented through anonymous Object[] array list structures were migrated to a dedicated Edge class, providing a clearly defined and more maintainable representation of graph relationships. In parallel, a new `TraversalData` class was introduced to encapsulate both vertex and relationship traversal history, allowing agents to preserve not only which vertices were visited during pattern matching, but also which specific relationships were traversed between them.

These modifications enabled relationship information to be packaged into `CypherResultRow` objects and transferred to later execution phases alongside vertex data. As a result,

relationships became explicitly represented query entities that are capable of being preserved through earlier clause evaluation phases such as MATCH and WHERE and then made available for future graph mutation operations such as direct relationship deletion. In addition to such a mutation phase, these relationship values can also be returned directly to the user.

This report discusses the motivations behind these architectural changes, the implementation details of the `Edge` and `TraversalData` refactors, the result of modifications to the MATCH and WHERE execution pipeline, and how these enhancements prepare the MASS graph database for future support of direct relationship mutation functionality.

2. Background & Motivation

The MASS graph database processes Cypher queries through a declarative-to-imperative execution pipeline in which clauses such as MATCH, WHERE, and DELETE are translated into distributed `PropertyGraphAgent` graph traversals as seen in Figure 1 below. Stage 1 at the application layer takes in a declarative input query from the user that describes *what* the user is looking to achieve within the DB and then the following imperative operations of the pipeline determine *how* that query will be actualized. Stage 2 uses ANTLR (ANotherTool for Language Recognition version 4) to assess the grammatical correctness of the query and then construct a parse tree, or concrete syntax tree composed of `oC_Cypher` rule structures that hold lexical portions of the original input query. Stage 3 then has the `PropertyGraphCypherVisitor` class recursively traverse the parse tree and construct the Abstract Syntax Tree (AST). Stage 4's AST holds the semantic meaning of the original input query using `CypherStatement` clause nodes, which are then used in the following stage. The `ExecutionPlanBuilder` in stage 5 recursively traverses the AST and constructs the `ExecutionPlan`, which in stage 6 is a tree comprised of `ExecutionStep` objects. These `ExecutionSteps` are recursively called within the application `GraphManager` to recursively perform their execute function, which deploys stage 7, the deployment of `PropertyGraphAgents` to spawn in the property graph and perform their designated logic. During query execution, agents migrate between graph vertices, evaluate constraints, and construct intermediate results that are passed forward to later ExecutionStep stages such as DELETE, DETACH DELETE, and RETURN using `CypherResultRow` objects to pass forward pertinent information.

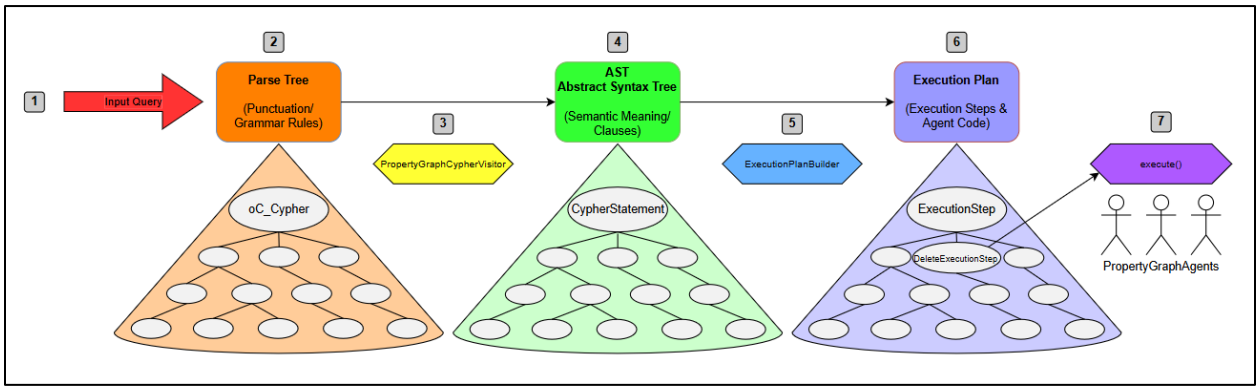

Figure 1: Declarative-to-Imperative Query Pipeline Diagram.

Prior quarter's work established the foundational infrastructure for graph mutation through the implementation of DELETE and DETACH DELETE operations. While these features enabled deletion of vertices and detachment of associated relationships for those specified vertices, a significant architectural limitation remained regarding relationship identity preservation through the entire query pipeline. Specifically, the query execution pipeline was primarily designed around vertex-oriented traversal results, which caused relationship-specific traversal information to be lost once `PropertyGraphAgent` traversal completed for a given clause phase.

At the time, traversal results were represented through a `pathResult` structure containing only the sequence of vertices visited during agent migration. For example, an agent traversing from a Person vertex representing actor Keanu Reeves to a Movie vertex representing the movie The Matrix would preserve the visited vertices within the `pathResult`, but the ACTED_IN relationship responsible for that traversal would not be retained. Consequently, later execution stages could determine vertices where an agent had traveled but would not be able to sufficiently determine which specific relationship enabled each step of that traversal. This can be seen in a representation of an initial trial query below in Figure 2 where you can see that the relationship value was not retained from previous clause phases and therefore cannot properly be displayed after results are returned to the user.

```
MATCH (p:Person)-[r]->(m:Movie)
RETURN p, r, m;

Printing Query Results:
p = keanu        r = null      m = thematrix
p = laurence     r = null      m = thematrix
p = carrie       r = null      m = thematrix
```

Figure 2: Relationship information is not preserved and passed to the RETURN phase.

The limitation became increasingly problematic as development progressed toward supporting direct relationship manipulation. Query clause segments such as "DELETE r" or "RETURN r" require the execution engine to preserve relationship identity throughout the entire query pipeline. Without this information, relationships could be traversed during pattern matching but could not be returned, filtered, or directly modified during subsequent execution stages. Without information provided on the specifics of a relationship, `PropertyGraphAgents` are not able to perform specific edge mutations as they find the information they've been provided only contains null information for the associated relationships. Resolving this issue requires change to the packaging of path data via the introduced `TraversalData` class and is covered further in section 3 regarding implementation.

An additional limitation beyond relationship information preservation involved the internal representation of graph relationships themselves. The codebase previously represented

relationship information using generic `Object[]` array list structures containing relationship metadata and property maps. While functional, this approach reduced readability, required extensive type casting, and complicated maintenance of the codebase in calls to reference such objects. More importantly, the `Object[]` representation stored relationship type and property metadata but did not provide a dedicated relationship object that could be easily preserved, referenced, and manipulated throughout later query execution stages. While sufficient for traversal, this structure complicated scenarios involving multiple relationship instances between the same pair of vertices and made later relationship-oriented operations more difficult to implement. As graph datasets become more complex, preserving the identity of individual relationships becomes increasingly important for accurate query evaluation and data representation including for future graph mutation operations. This issue was addressed as part of the implementation of the `Edge` class to replace this previously used `Object[]` array list. Listing 1 below showcases the structure of the previous `Object[]` edge version.

Listing 1: Relationship structure with Object[] structure.

```
// ============================================================
// BEFORE: Object[] Relationship Representation
// ============================================================
//
// Type Structure:
//
// Map<Object, List<Object[]>>
//
// neighbor itemID -> [
//       Object[] {
//            Set<String> relationshipTypes,
//            Map<String, String> relationshipProperties
//       }
// ]
//
// Example:
//
// toRelationship = {
//
//      "thematrix" : [
//
//            Object[] {
//                 relationship[0] = {"ACTED_IN"},
//                 relationship[1] = {"role":"'neo'"}
//       ]
// };
//
```

To address these limitations of relationship preservation and multiple relationships between the same pair of vertices, the Spring 2026 quarter focused on redesigning how relationships are represented and propagated throughout the MASS graph database execution pipeline. This work introduced a dedicated `Edge` class to replace the previous `Object[]` array representation and implemented a new `TraversalData` structure capable of preserving both vertex and relationship traversal history. Together, these modifications established the foundation necessary for relationship-aware query processing and future support of direct relationship mutation operations.

3. Implementation

The Spring 2026 quarter focused on extending the MASS graph database execution pipeline to preserve relationship identity throughout the query execution. While previous work established foundational graph mutation operations for vertex deletion, relationship information was not retained beyond distributed graph traversal phases. As a result, relationships could participate in MATCH and WHERE evaluation but could not be directly returned or manipulated by subsequent execution stages.

To address this limitation, modifications were implemented at three primary layers of the system architecture. First, the internal representation of graph relationships was refactored from anonymous `Object[]` array list structures into dedicated Edge objects. Second, traversal state management was redesigned through the introduction of the `TraversalData` class, enabling simultaneous preservation of both vertex and relationship traversal history. Finally, the MATCH and WHERE execution pipeline was modified to package relationship information into `CypherResultRows`, making relationships available to later execution stages such as RETURN and future graph mutation operations.

3.1 Relationship Representation Refactor

The first stage of implementation focused on redesigning the internal representation of graph relationships.

Historically, relationship information was stored using Object[] array structures embedded within adjacency maps. While functional for traversal operations, this representation required frequent type casting and provided no explicit object representation for a relationship. Relationship metadata and properties could be accessed, but individual relationships themselves were not represented as explicit entities within the codebase. To improve maintainability and establish a more robust foundation for future relationship-oriented operations, a dedicated Edge class was introduced. The Edge class encapsulates relationship-specific information including source vertex IDs, destination vertex IDs, relationship type information, and associated properties as seen in Listing 2 below.

Listing 2: Relationship structure with new Edge structure.

```
// ============================================================
// AFTER: Edge Relationship Representation
// ============================================================
//
// Type Structure:
//
// Map<Object, List<Edge>>
//
// neighbor itemID -> [
//      Edge {
//             String   edgeID,
//             String   fromVertexID,
//             String   toVertexID,
//             String   edgeType,
//             Map<String, String> properties
//      }
// ]
//
// Example:
//
// toRelationship = {
//
//     "thematrix" : [
//          Edge {
//               String edgeID        = "",
//               String fromVertexID  = "keanu",
//               String toVertexID    = "thematrix",
//               String edgeType      = "ACTED_IN",
//               Map<String, String>  properties = {"role":"neo"}
//          }
//     ]
// }
// ============================================================
```

By introducing an explicit relationship object, relationship data could now be passed throughout the execution pipeline without relying on anonymous array structures. This redesign improved readability, reduced type-casting requirements, and established a clear representation for future operations involving relationship-specific mutation and retrieval.

3.2 `TraversalData` Refactor

Following implementation of the Edge class, traversal state management required redesign to retain relationship information generated during distributed graph traversal.

Prior to this work, `PropertyGraphAgents` stored traversal history using a `pathResult` structure containing only vertex IDs. While this allowed later execution stages to determine which vertices had been visited, no information was retained regarding which specific relationships had been traversed between those vertices.

As a result, relationship identity was discarded once agent traversal completed. Subsequent execution stages could determine that a traversal occurred between two vertices but could not identify the specific relationship responsible for that traversal.

To address this limitation, a new `TraversalData` class was introduced. `TraversalData` encapsulates two complementary forms of traversal information:

- `nodePath`, which stores visited vertex IDs
- `edgePath`, which stores traversed `Edge` objects

As `PropertyGraphAgents` migrate throughout the graph, both structures are populated simultaneously. Visited vertices continue to be recorded within `nodePath` while traversed relationships are added within `edgePath`. Listing 3 below shows what the `pathResult` looked like before this work was implemented compared to the after state which additionally holds Edge information explicitly.

Listing 3: Before, only vertex information was stored. After, Edge information is preserved.

```
// ================================================
// Before
// ================================================
// ArrayList<String> pathResult
//     [keanu, thematrix]
// ================================================

// ================================================
// After
// ================================================
// TraversalData
//      ArrayList<String> nodePath
//           [keanu, thematrix]
//
//      ArrayList<Edge> edgePath
//      [
//           Edge {
//                edgeID        = ""
//                fromVertexID  = "keanu"
//                toVertexID    = "thematrix"
//                edgeType      = "ACTED_IN"
//                properties    = {"role":"neo"}
//           }
//      ]
// ================================================
```

This redesign enables the system to keep track of not only where an agent traveled but also how it traveled. As a result, relationship identity is maintained throughout query execution rather than being discarded once traversal completes.

3.3 Relationship-Aware Query Pipeline

The final implementation stage involved modifying the query execution pipeline to transfer the newly packaged relationship information into `CypherResultRows` in the MATCH and WHERE phase that are then handed off to subsequent clause phases.

Prior to this work, the MATCH and WHERE execution phase produced `CypherResultRows` containing only vertex-oriented traversal information derived from `pathResult`. Consequently, relationship variables such as r in a query pattern such as an example MATCH (p)-[r]->(m) could not be returned or manipulated during later execution stages because no relationship object information had been preserved. The introduction of the `TraversalData` class enabled both `nodePath` and `edgePath` information to be available at the completion of a given graph traversal clause phase. `MatchPatternPartExecutionStep` was subsequently modified to package matched `Edge` object information into `CypherResultRows` alongside traditional vertex results.

This enhancement allows relationship information to survive the transition between execution step phases. Queries containing relationship variables can now access specific relationship instances after MATCH and WHERE processing completes.

Evaluation of these modifications was performed using a custom graph dataset containing Person and Movie vertices connected through ACTED_IN relationships. An example query displayed in Figure 3 below shows that relationship information is now capable of being preserved throughout the entire query pipeline as needed as compared to similar queries shown in Figure 2 previously where the r value would have been null.

```
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)

WHERE r.role = "neo"

RETURN p, r, m;


Printing Query Results:

p = keanu

r = Edge{edgeId='', fromVertexId='keanu', edgeType='acted_in',
        toVertexId='thematrix', properties={salary=20000000, role=neo}}

m = thematrix
```

Figure 3: Sample query showing successful return of Edge information back to the user through a MATCH -> WHERE -> RETURN query pipeline.

These results demonstrate that relationship identity can now be propagated throughout the query pipeline, establishing the architectural foundation necessary for future implementation of direct relationship mutation operations.

4. Conclusion

The Spring 2026 quarter focused on addressing a key architectural limitation within the MASS graph database execution pipeline, which was the inability to preserve relationship identity beyond clause-oriented `PropertyGraphAgent` graph traversal phases. While previous work established functional DELETE and DETACH DELETE operations for vertex mutation, relationships themselves could not be directly returned, referenced, or mutated because relationship information was discarded once MATCH and WHERE evaluation completed.

To address this limitation, the internal representation of graph relationships was redesigned through the introduction of a dedicated Edge class, which replaced the simple `Object[]` array list structure. This refactor improved code readability, reduced type-casting complexity, and established an explicit relationship object capable of being propagated throughout the query execution pipeline. In addition, the newly introduced `TraversalData` structure extended traversal state preservation beyond vertex-only path information by maintaining both `nodePath` and `edgePath` traversal history simultaneously.

Modifications to `MatchPatternPartExecutionStep` and associated query execution components enabled matched `Edge` object information to be packaged into `CypherResultRow` objects alongside vertex IDs. Evaluation results demonstrated that relationship variables can now survive MATCH and WHERE processing and be successfully returned to the user through subsequent execution stages. This establishes relationships as identifiable entities within the MASS graph database architecture and removes a major obstacle toward future relationship-oriented graph mutation operations.

It is important to note that this work does not yet introduce globally unique relationship IDs comparable to the vertex IDs utilized in the current system's distributed hash map associated with `PropertyVertexPlace` nodes. Future work may explore distributed ID allocation mechanisms to support direct relationship mutation by unique edge ID. The primary objective of the current quarter was to preserve relationship identity through the query pipeline itself. By maintaining relationship context through source and destination vertices, relationship types, and associated properties, the system now possesses the information necessary to support targeted relationship mutation operations without requiring globally assigned edge IDs.

These enhancements provide the architectural foundation for the next stage of development, which will focus on extending DELETE and DETACH DELETE functionality to support direct relationship deletion in Summer quarter 2026 while functional comparison to Neo4j and ArangoDB will subsequently take place in Fall quarter 2026. Future work will necessarily investigate how DELETE and DETACH DELETE operations should now interpret inbound

`CypherResultRow` data which now contains both vertex and relationship data simultaneously packaged within the same result information passed between `ExecutionStep phases`.

These efforts move the MASS graph DB framework closer to having consistent and reliable deterministic deletion capabilities for both nodes and relationships across a distributed multitude of agents acting in parallel within the DB system. Once finished, these mutation features will bring the MASS DB framework closer to feature parity with mature graph DB platforms while maintaining its unique underlying distributed multi-agent execution model.

5. 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).

[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/

[20] R. Isaacson, *"Analysis of Declarative-to-Imperative Cypher Query Translations in MASS - A Distributed Graph Query Framework,"* Term Report, M.S. Computer Science & Software Engineering, University of Washington Bothell, Dec. 2025. Available: https://depts.washington.edu/dslab/MASS/reports/RyanIsaacson_au25.pdf

[21] Neo4j Inc., *"DELETE — Cypher Manual,"* Neo4j Documentation. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/clauses/delete/. Accessed: Mar. 6, 2026.

[22] Neo4j Inc., *"SET — Cypher Manual,"* Neo4j Documentation. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/clauses/set/. Accessed: Mar. 13, 2026.

[23] Neo4j Inc., *"MATCH — Cypher Manual,"* Neo4j Documentation. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/clauses/match/. Accessed: Mar. 14, 2026.

[24] Neo4j Inc., *"WHERE — Cypher Manual,"* Neo4j Documentation. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/clauses/where/. Accessed: Mar. 20, 2026.

[25] R. Isaacson, *"Analysis of Declarative-to-Imperative Cypher Query Translations in MASS - A Distributed Graph Query Framework,"* Term Report, M.S. Computer Science & Software Engineering, University of Washington Bothell, March. 2026. Available: https://depts.washington.edu/dslab/MASS/reports/RyanIsaacson_wi25.pdf

6. 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