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

Term Report of Work Done for
Winter Quarter 2026

By Ryan Isaacson

Master of Science in Computer Science & Software Engineering

University of Washington
March 20th, 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 [20] 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 post-graph-creation editing features that were not previously available within the system. 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, as well as analyze the declarative-to-imperative query translation process and how its engineering within a MASS-based graph DB system compares to alternative libraries.

The Fall quarter of 2025's work began with a necessary and preparatory foundational analysis for how to begin this work. The initial goal was to better understand the framework's composition in order to decidedly articulate and implement what changes would be necessary within the codebase in order to add Cypher query language [3] writing clause functionality types including DELETE [21] and SET [22]. These operations offer editability of a particular vertex within a sample graph database in order to manipulate data properties after the initial graph network's construction.

The Winter quarter of 2026 has contributed concrete implementation steps for DELETE and it's integral DETACH DELETE [21] functionality. DELETE offers the removal of an unconnected vertex in the graph DB, whereas DETACH DELETE offers a decoupling of edge relationships attached to specified vertices before performing a structural removal of the specified vertices from the graph. This work successfully extends the declarative-to-imperative translation pipeline by enabling agents to not only query graph data, but to also perform structural modifications through controlled vertex and relationship removal. These operations allow for the graph structure to evolve more dynamically over time, however supporting these features adds additional complexity compared to read-only queries. In particular, deleting a vertex requires careful coordination to ensure that all associated relationships are properly removed across potentially distributed nodes, while maintaining consistency and avoiding orphaned edge relationships.

Within the MASS query execution model, distributed computation is structured around two core abstractions: Places and Agents. Places represent graph vertices while Agents act as mobile execution engines performing localized computation at those vertices and they migrate between Places in phased movement. Building on these primitives, the MASS graph DB layer introduces GraphPlaces, which extend Places in order to manage collections of VertexPlace objects representing graph vertices. GraphAgents extend Agents to support traversal-based computation over the graph structure. Finally, to support Cypher-compatible semantics, these components are further extended through PropertyGraphPlaces, PropertyVertexPlace, and PropertyGraphAgent respectively to incorporate labeled vertices, typed relationships, and property-aware state required for declarative clause execution. Together, this layered

architecture provides the underlying mechanisms that enable Cypher-based queries to be translated into coordinated agent-based operations over the distributed property graph.

This work introduces a distributed deletion mechanism driven by autonomous agents rather than relying on a fully centralized coordination system. Specifically leveraging PropertyGraphAgent instances to execute deletion logic locally at each vertex. When a DELETE or DETACH DELETE clause is encountered, agents are deployed to the target vertices identified during earlier MATCH [23] and/or WHERE [24] query stages. For DETACH DELETE operations, a two-phase process is employed involving the deployment of an initial parent agent which removes local relationships on the vertex it was deployed to. Then the parent agent sets up child agents to spawn at adjacent vertices to remove corresponding edge references back to the vertex the parent agent was initially deployed to. This design aligns with the decentralized philosophy of MASS while enabling safe and consistent graph mutation.

The implementation of DELETE and DETACH DELETE represents a significant milestone in transforming the MASS Java graph query engine from a read-only system into one capable of full graph manipulation. This advancement not only expands the functional completeness of the system, but also provides a foundation for evaluating how declarative graph queries can be translated into distributed, agent-based execution patterns for Agent-Based Modeling (ABM) methods. This key achievement enables future exploration into performance, scalability, and correctness when compared to established graph database systems such as Neo4j and ArangoDB.

The contributions of this Winter quarter's work are threefold. First, a functional DELETE and DETACH DELETE execution pipeline has been designed and implemented within the MASS framework. Second, the system introduces a distributed edge detachment strategy that ensures safe removal of relationships across vertices without requiring full global synchronization. Third, the implementation has been validated through a small series of controlled experiments on a micro-scale dataset to demonstrate correct behavior for both isolated vertex deletion and relationship-aware DETACH DELETE operations. This report presents the design, implementation, and evaluation of these features.

2. Background & Motivation

Graph databases have emerged as a powerful alternative to traditional relational database systems for modeling highly connected data. Rather than representing data in tables joined through foreign keys, graph databases store data as vertices and relationships, which enables more natural representations of real-world entities such as social networks, recommendation systems, and knowledge graphs. Query languages such as Cypher which provide a declarative interface for expressing graph traversals and pattern matching operations allow users to describe what results they want without specifying exactly how to compute them and instead leave that imperative detail to be managed under-the-hood of the graph query framework.

Within this paradigm, much of the existing work in the MASS graph framework has focused on supported read-oriented query clauses such as CREATE, MATCH, and WHERE, which enable pattern discovery and filtering across distributed graph structures. These operations map well to the agent-based execution model of MASS, where autonomous agents traverse the graph and evaluate conditions at each vertex. Implementing write operations such as DELETE and DETACH DELETE however, introduces a fundamentally different set of challenges compared to read-only queries.

In traditional graph database systems such as Neo4j, DELETE operations are handled through tightly controlled transactional mechanisms that ensure consistency across the graph. For example, a vertex cannot be deleted if it still has existing relationships unless DETACH DELETE is explicitly stated. These guarantees are enforced through centralized coordination mechanisms which coorelate to its use of the RAFT consensus algorithm [13].

In contrast, the MASS framework follows a decentralized, agent-based execution model where computation is distributed across multiple compute nodes. This design introduces unique challenges when attempting to support graph mutation. Specifically, deleting a vertex requires ensuring that all associated relationships are safely removed not only locally, but also across neighboring vertices that may reside on different compute nodes. Without proper coordination, this can lead to inconsistent graph state, such as orphaned edges or race conditions caused by concurrent modifications.

The motivation for this work is to extend MASS beyond read-only query capabilities and enable safe, distributed graph mutation through DELETE and DETACH DELETE operations. This requires designing a system that preserves the decentralized nature of MASS while introducing sufficient coordination to maintain graph consistency. In particular this work focuses on developing a distributed deletion mechanism that avoids global synchronization as much as possible while ensuring correctness through localized agent behavior and structured execution phases. This work aims to bridge the gap between declarative graph query languages and imperative agent-based execution models in a functionally scalable and decentralized methodology.

3. Related Work

The development of distributed graph database systems and query execution frameworks has been an active area of research, particularly in the context of scaling graph operations across large datasets and distributed environments. Traditional graph database systems such as Neo4j have established a strong foundation for declarative graph querying through the Cypher query language, providing support for both read and write operations including MATCH, WHERE, DELETE, DETACH DELETE, and many others [3][5][21]. Those systems rely on transactional guarantees and centralized coordination mechanisms to ensure consistency during graph mutation.

In contrast, MASS adopts a fundamentally different approach by leveraging an agent-based execution model to perform distributed computation across graph structures [1]. Rather than

relying on centralized query planning and execution, MASS distributes logic across autonomous agents that operate locally at each vertex. This design aligns with ABM principles and enables scalable parallel execution, but introduces additional complexity when extending beyond read operations.

Prior work within the MASS graph framework has focused on incrementally expanding support for Cypher query clauses. Dea [6] introduced support for the CREATE clause which enabled graph construction, while Cao [8] and Ma [9] contributed further to enhancements in distributed graph representation and query execution. More recently, Prajapati [10][11] implemented support for the WHERE clause, which allowed for conditional filtering logic. These contributions and the mechanisms underlying them establish a declarative-to-imperative translation pipeline for read-oriented queries. The basics of the pipeline including their build up to current DELETE and DETACH DELETE operations can be seen outlined in Figure 1 below.

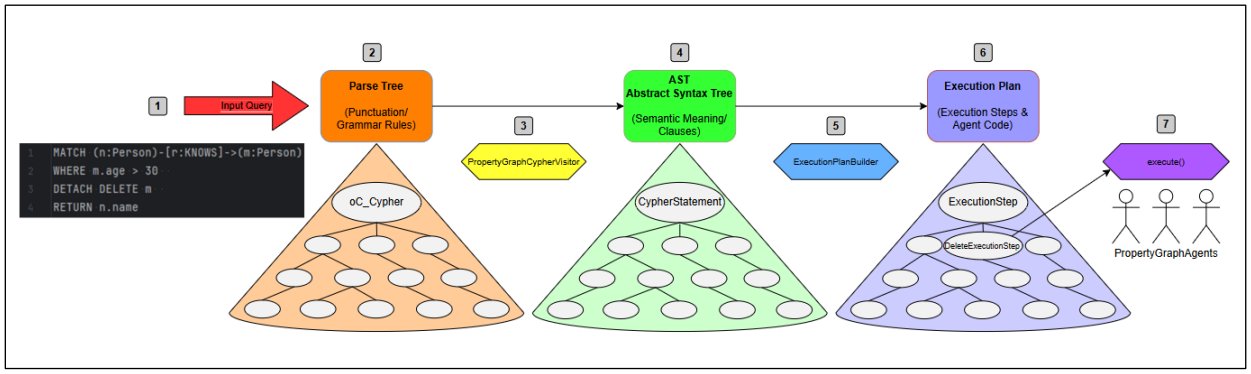

Figure 1: Query translation pipeline from input to agent distribution and execution.

Work has been limited on supporting graph mutation operations that require coordinated updates across multiple vertices and across multiple compute nodes. DELETE and DETACH DELETE introduce challenges that were not present in earlier work which include the need to safely remove relationships from endpoints and ensure no residual hanging references remain after query completion.

From a broader perspective, distributed systems research has explored various strategies for maintaining consistency during concurrent updates, including consensus algorithms like RAFT [13] and while that sort of approach can offer strong consistency guarantees, it relies deeply on centralized coordination and synchronization mechanisms that do not directly correlate to the decentralized design philosophy of MASS.

This work builds on prior MASS extensions by introducing an agent-based approach to graph deletion that avoids global synchronization while preserving correctness. By leveraging local agent behavior, context propagation, and structured execution phases, the implementation provides a novel method for performing DELETE and DETACH DELETE operations within a distributed graph environment.

4. Implementation

4.1 Dataset and Graph Representation

To validate the functionality of the DELETE and DETACH DELETE clauses, a micro-scale dataset themed for the movie The Matrix was constructed consisting of Person and Movie vertices attached via directed ACTED_IN relationships. The dataset includes actors such as Carrie-Anne Moss, Keanu Reeves, and Laurence Fishburne which are each connected to a Movie vertex representing The Matrix. An additional isolated vertex for the actor Tom Hanks who does not star in the movie is included to test DELETE behavior on vertices without relationships. Figure 2 below represents the initial graph structure.

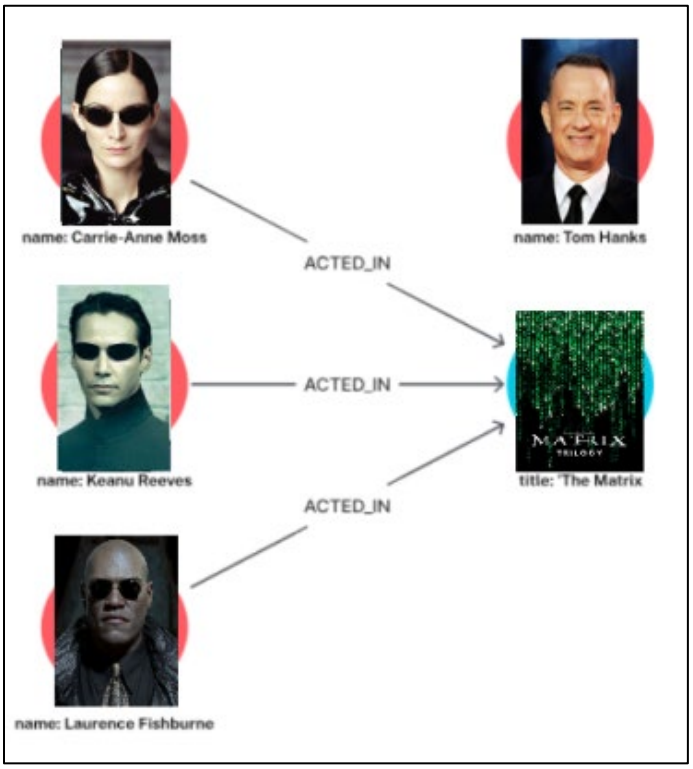

Figure 2: Initial graph structure for The Matrix dataset showing Person-to-Movie relationships.

This dataset provides a controlled environment for testing and demonstrating both simple deletion cases and relationship-aware detachment operations. With this dataset having a small size and clear structure, it allows for step-by-step visualization of how agents interact with the graph during execution.

4.2 DELETE Execution Pipeline

The DELETE clause was implemented as an extension of the existing declarative-to-imperative query execution pipeline. Figure 3 highlights the example query used here for demonstration purposes where we will perform a DELETE operation on the detached actor Tom Hanks.

```
Enter queries: MATCH (p:Person) WHERE p.name = "Tom Hanks" DELETE p;
```

Figure 3: Example query to test DELETE operation on vertex with no relationships attached.

After the MATCH and WHERE clauses identify and constrain a set of target vertices to delete, these targets are passed into the DeleteExecutionStep's execution phase via PropertyGraphCypherResult rows. These rows of data are returned and passed through each ExecutionStep into subsequent ExecutionStep's through the recursive traversal of the ExecutionPlan tree. Lines 1-8 in Listing 4 illustrate sifting through the PropertyGraphCypherResult rows returned from the MATCH/WHERE execution phase and passing the vertex IDs to delete into the DELETE execution phase.

Listing 4: Lines 1-8 showcase data transfer from MATCH/WHERE to DELETE for which vertices to delete.

```
1  source.forEach(row -> { // PropertyGraphCypherResult
2      for (String var : this.deleteTargets) {
3          Object value = row.get(var);
4          if (value != null) {
5              vertexNodeIDsToDelete.add((String) value);
6          }
7      }
8  });
```

Next, the AgentInitArgs class is created which has its own constructor setup specifically for DELETE and DETACH DELETE PropertyGraphAgents outlined in Table 5 below. The parameters passed include whether we are in a DETACH case or not, the target vertices to delete, and the originNodeID which will be explained in full within section 4.2 for DETACH DELETE, but it essentially allows for an agent to know if it's a first generation or second-generation instantiation of a delete-oriented PropertyGraphAgent.

Table 5: AgentInitArgs parameters for DELETE & DETACH DELETE PropertyGraphAgents.

Type	Variable	Remarks
Boolean	isDetachClause	Whether the DELETE clause included DETACH or not
Set<String>	deleteTargets	The set of node IDs to be deleted
String	originNodeID	Indicates stage 1 parent agent or stage 2 child agent

In Figure 6 below the agentInitializer with its newly packaged AgentInitArgs data (line 94) is handed off to the agent creation phase (line 95). For customization the agents will receive their class name (line 97), the target graph (line 99) and the number of vertices in the graph (line 100).

Listing 6: AgentInitArgs object are created and agents are instantiated.

```
24  public class DeleteExecutionStep extends ExecutionStepWithChildren {
50      public PropertyGraphCypherResult execute(PropertyGraphCypherQueryContext ctx, PropertyGraphCypherResult source) {
94          this.agentInitializer = new AgentInitArgs(this.isDetachClause, vertexNodeIDsToDelete, null);
95          Agents agents = new Agents(
96              0,                                  // Base agent ID
97              PropertyGraphAgent.class.getName(), // Agent class name
98              this.agentInitializer,              // Initialization data for each agent
99              ctx.getGraph(),                     // Graph (Places object)
100             graphSize                           // Deploy to every vertex node in the graph
101         );
```

Figure 7 then showcases agents being deployed across all vertices in the graph. Each PropertyGraphAgent determines whether it has been deployed to a valid delete target vertex or not by checking if the current vertex the PropertyGraphAgent is deployed to is within the deleteTargets set of unique vertex identifier strings. If the agent is not responsible for deletion, it terminates early. If the PropertyGraphAgent is deployed at a vertex belonging to the deleteTargets, it proceeds to mark the current vertex for structural removal from the graph later on in the DeleteExecutionStep. In this case, Figure 8 shows Tom Hanks being marked for structural deletion.

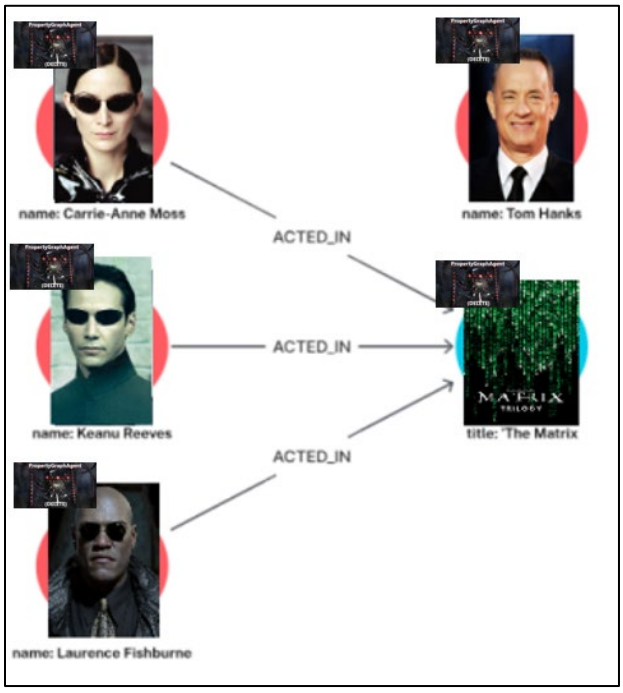

Figure 7: Initial agent distribution to all vertices during DELETE.

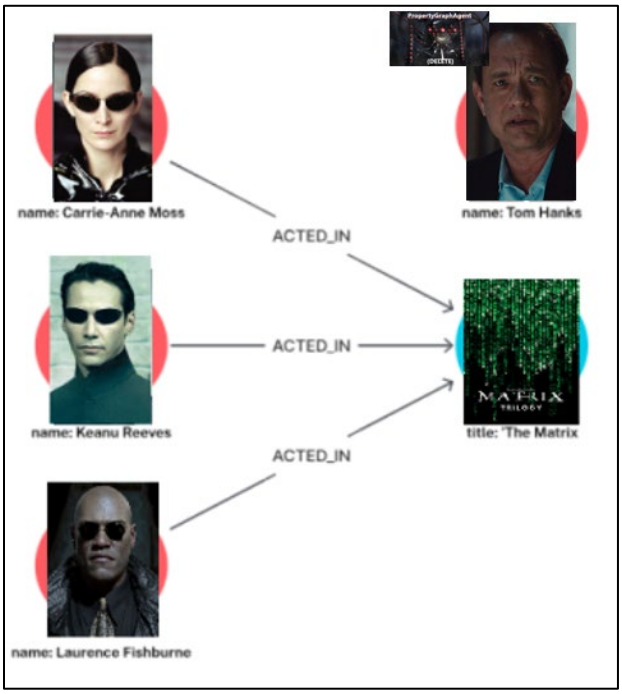

Figure 8: Graph showing DELETE agent targeting the query specified Tom Hanks vertex without relationships.

For standard DELETE operations, the system enforces a constraint that only vertices without relationships can be removed. When this condition is satisfied, the vertex is marked for structural deletion from the graph structure without requiring additional coordination at the end of the DeleteExecutionStep phase.

4.3 DETACH DELETE Algorithm

The DETACH DELETE clause introduces additional complexity by requiring the removal of both a vertex and all of its associated relationships. Figure 9 exhibits the example query specified for demonstration where the Movie object titled The Matrix will be detached and deleted.

```
Enter queries: MATCH (p:Person) WHERE m.title = "the matrix" DETACH DELETE m;
```

Figure 9: Example query to test DETACH DELETE query on vertex with existing relationships attached.

To support the valid removal of the vertex in the case of detachment, a distributed two-phase algorithm was implemented using agent-based execution to position agents with the role of decoupling edge connection points at the target vertices. Initially agents are dispersed across the entire graph as seen in Figure 10 below, with agents that spawned on non-target vertices choosing to self-terminate without action. Next, any agents left alive are by design positioned at a target vertex for detachment. These agents are counted as parent agents and are responsible for detaching all local relationship endpoints from the target vertex. This includes removing entries from both outgoing relationships owned by the PropertyVertexPlace, as well as incoming relationships. The parent agent identifies all neighboring vertices connected through these relationships and before self-termination, sets up the child agents to spawn adjacently with additional initialization arguments which include setting the originNodeID field so that future child agents that spawn will know the ID of their parent agent's origin vertex. This is shown in Figure 11 below.

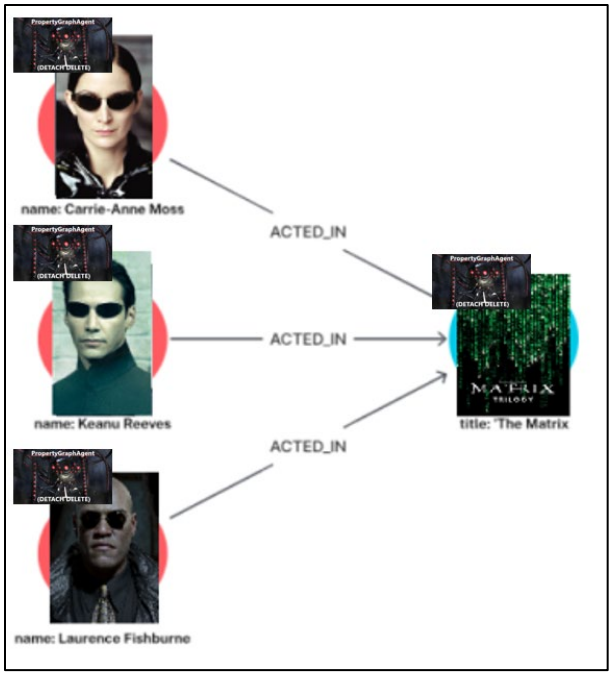

Figure 10: Initial agent deployment to all available vertices.

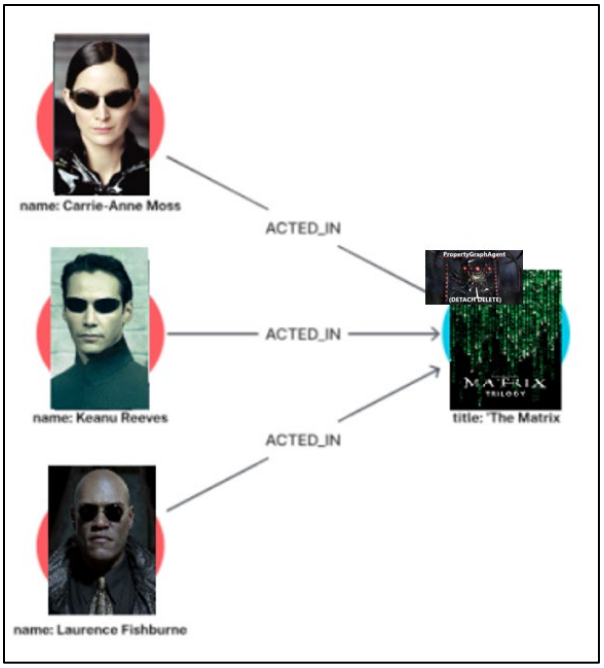

Figure 11: Parent agent located at The Matrix. Parent agent detaches local relationships, sets originNodeID, and detaches local relationship endpoints.

To support coordinated edge removal across distributed vertices, contextual information about the delete target origin vertex ID is propagated from parent agents to child agents during execution. Specifically, the originNodeID field is assigned by the parent agent to represent the identifier of the vertex being deleted. This value is passed to all adjacent child agents during instantiation through modifications made in the AgentsBase class, where agent initialization arguments were extended to include necessary delete execution state as seen in Listing 12 below.

Listing 12: Lines 1632-1635 specifically ensure spawned child DETACH DELETE agents are notified of the parent originNodeID.

```
66   public class AgentsBase {
1571     private void ManageLifeCycleEventsForPropertyGraph( int tid ) {
1629                         if (currAgent.getIsDeleteClause() && currAgent.getIsDetachClause()) {
1630                             addAgent = (PropertyGraphAgent) objectFactory.getInstance(
1631                                 className,
1632                                 new AgentInitArgs(
1633                                     currAgent.getIsDetachClause(),
1634                                     null,                          // DETACH DELETE child does not need deleteTargets
1635                                     currAgent.getOriginNodeID()    // Only needs originNodeID
1636                                 )
1637                             );
1638                         }
```

Upon migration and spawning on neighboring vertices, in the child agents phase they use the originNodeID to identify and remove the corresponding relationships on the adjacent vertex that directly connect back to the origin vertex node as represented in Figure 13 and Figure 14. This design avoids the need for direct agent-to-agent communication or shared global state, as all necessary context is embedded within the agent itself at the time of its instantiation.

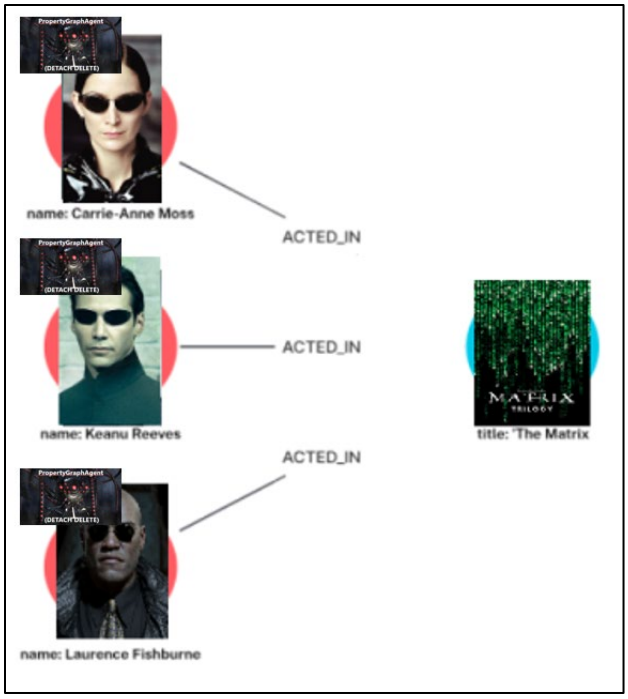

Figure 13: Child agents spawn at adjacent vertices to the originNodeID vertex.

Figure 14: Child agents remove reverse relationships associated with originNodeID vertex.

4.4 Edge Detachment Implementation (PropertyVertexPlace)

To support the removal of relationships during DETACH DELETE logic in Section 4.3, a dedicated method detachIncomingOrOutgoingEdge was implemented within the PropertyVertexPlace class. This method provides a controlled mechanism for modifying the adjacency structures that store both outgoing (toRelationship) and incoming (fromRelationship) edges associated with a vertex.

The method supports multiple modes of operation, allowing for selective removal of incoming, outgoing, or both directional types of relationships depending on which arguments are passed to the function. Within the DETACH DELETE execution pipeline, this functionality is used in two distinct phases. First, parent agents invoke the method to remove all local relationships from the target vertex. This ensures that the vertex is no longer connected to any neighbors form its own perspective. Second, child agents invoke the same method on their respective neighboring vertices to remove the corresponding reverse references to the origin vertex.

To ensure correctness in a distributed execution environment, the detachment logic is implemented in an idempotent manner. Before removing any relationship, the method checks for the existence of the relationship within the corresponding adjacency map. This prevents errors in cases where multiple agents attempt to remove the same relationship or when operations are executed in overlapping or potentially non-deterministic order.

This design ensures that edge removal is both safe and consistent across all vertices involved in the deletion process, forming a critical foundation for the correctness of DETACH DELETE operations.

4.5 Structural Vertex Deletion (DeleteExecutionStep)

While the preceding sections describe how vertices are identified and how relationships are safely detached in a distributed environment, the final stage of the DELETE and DETACH DELETE execution pipeline is the structural removal of the vertices from the graph. This functionality is handled at the end of the DeleteExecutionStep after all agent-based operations have finished.

Rather than allowing agents to directly delete vertices during execution, a deferred deletion strategy is employed. During earlier phases, agents mark vertices for deletion once the required conditions have been satisfied. For standard DELETE operations this requires that the vertex has no remaining relationships. For DETACH DELETE operations this occurs only after all relationships have been removed through the idempotent parent and child 2-stage detachment algorithm described in Section 4.3 and 4.4. This phase is responsible for finalizing the deletion of vertices that have been marked as successfully detached and marked for structural deletion during agent execution. During the previous DELETE and DETACH DELETE phases, agents effectively mark the vertex for deletion after validating that all necessary conditions have been satisfied.

Once all agents have completed execution, the DeleteExecutionStep performs a cleanup phase in which all vertices marked for deletion are removed from the underlying PropertyGraphPlaces data structure. This ensures that structural modifications to the graph occur in a controlled and consistent manner rather than during intermediate agent execution steps. During the course of this work, improvements were also made to the agent lifecycle and execution model within the MASS framework to address potential synchronization issues. In earlier iterations, agents were managed within a more globally shared structure, allowing multiple threads to process agents associated with different vertices without strict locality guarantees. This created the possibility of concurrent modifications to shared vertex data structures, such as adjacency lists representing incoming and outgoing relationships, which would require synchronization mechanisms to ensure correctness.

To mitigate this, the agent management model was refined by other students working on MASS this Winter to enforce stronger locality between agents and their corresponding PropertyVertexPlace. Under this updated design, agents are grouped and processed within an AgentBag covering a localized execution context. As a result, concurrent access to the same vertex is avoided, eliminating the need for fine-grained synchronization when modifying relationship structures. This improvement was particularly important as it relates to DELETE and DETACH DELETE implementations presented in this work because edge detachment operations directly modify shared adjacency maps. This guarantee of localized execution ensures that these modifications can be performed safely and efficiently through dedicated agents and therefore

without introducing race conditions. This design aligns with the broader goal of enabling scalable and decentralized graph mutation within the MASS framework.

Shown below in Listing 15 on line 159, a series of functions are called that are found within the PropertyGraphPlaces class. These are the deleteLocalVertex method which removes the vertex from the local data structure, sendDeleteRemoteVertex which propogates deletion requests to remote compute nodes depending on vertex ownership if necessary, deleteVertexInternal which finalizes the cleanup of internal references, and removeVertexFromPlaces which updates the global graph structure. This process will culminate in the vertex having no hanging relationships and the PropertyVertexPlace itself being a null value that can be recycled later as needed by MASS.

By deferring structural deletion until after all agent operations have completed, the system avoids race conditions and ensures that no agent attempts to access or modify a vertex that has already been removed. This design is critical for maintaining correctness in a distributed graph execution model.

Listing 15: Line 159 performs a nullification of the vertex to be deleted, leaving a null graph hole to be recycled for re-use later.

```
24   public class DeleteExecutionStep extends ExecutionStepWithChildren {
50       public PropertyGraphCypherResult execute(PropertyGraphCypherQueryContext ctx, PropertyGraphCypherResult source) {
156          for (String vertexItemID : vertexNodeIDsToDelete) {
159              ctx.getGraph().deleteVertex(vertexItemID, this.isDetachClause);
160          }
```

5. Evaluation

5.1 Execution of DELETE

To evaluate the correctness of the DELETE and DETACH DELETE implementations, tests were done via the algorithms discussed in Section 4 on The Matrix dataset. These tests were introduced to verify that vertex removal behaved correctly under both isolated deletion conditions and relationship-aware detachment scenarios. The initial state of the graph as printed to the command line interface (CLI) when running the program are shown in Figure 16 below.

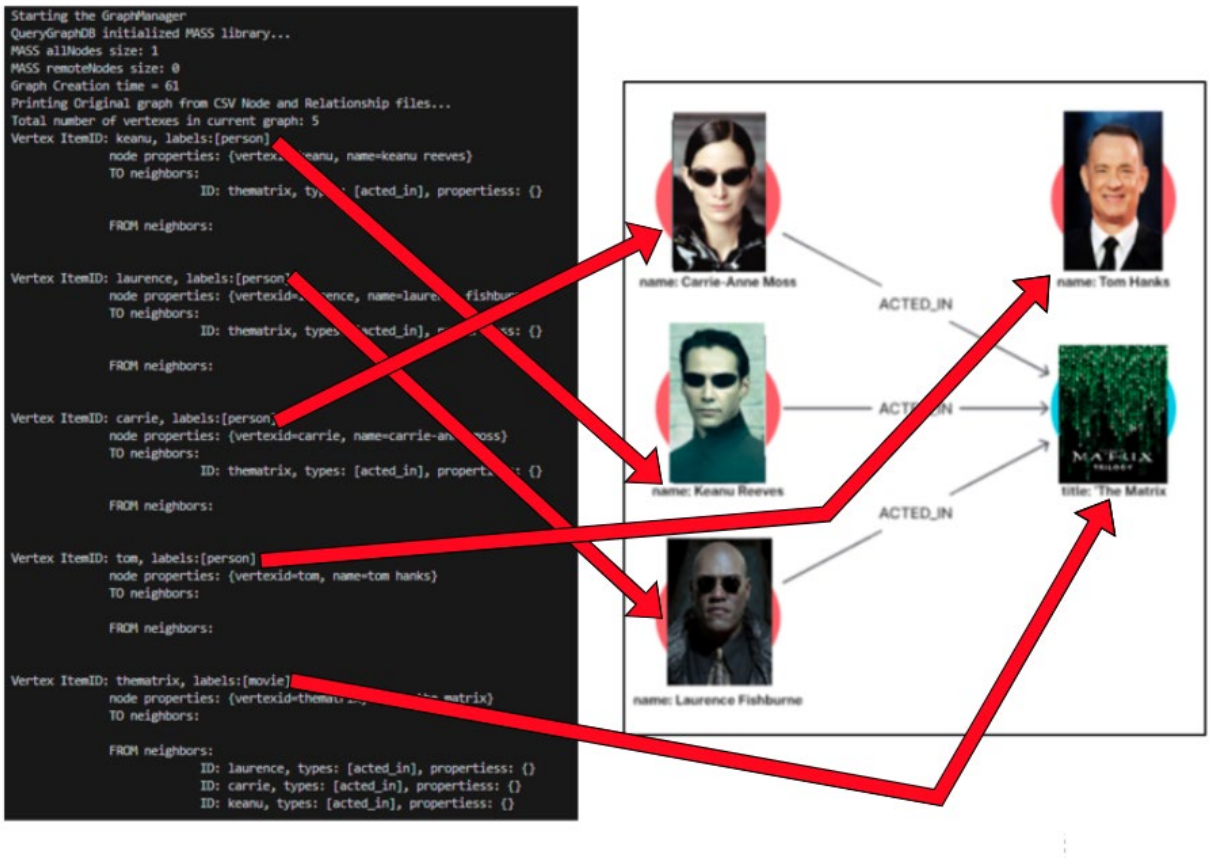

Figure 16: Initial graph state CLI print out on left, including original Figure 2 at right.

After running the command to DELETE the Tom Hanks PropertyGraphVertex, the logs behind the scenes show a successful delete are below in Figure 17, finishing on line 391.

```
381   21:01:24.084 [main] [DEBUG] [DeleteExecutionStep-execute] Beginning structural delete phase
382   21:01:24.084 [main] [DEBUG] [DeleteExecutionStep-execute] Calling deleteVertex(tom, detach= false)
383   21:01:24.085 [main] [DEBUG] [PropertyGraphPlaces-remoteVertexFromPlaces] Vertex deleted successfully, itemID= tom, internalVertexID= 3, localIndex= 3
384   21:01:24.085 [main] [DEBUG] [DeleteExecutionStep-execute] Structural delete phase complete
385   21:01:24.085 [main] [DEBUG] [DeleteExecutionStep-execute] Printing graph after DELETE
386   21:01:24.085 [main] [DEBUG] Printing the graph with node properties and relationship information: ============
387   21:01:24.085 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: keanu, toRelationship: 1, fromRelationship: 0
388   21:01:24.085 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: laurence, toRelationship: 1, fromRelationship: 0
389   21:01:24.085 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: carrie, toRelationship: 1, fromRelationship: 0
390   21:01:24.085 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: thematrix, toRelationship: 0, fromRelationship: 3
391   21:01:24.087 [main] [DEBUG] [DeleteExecutionStep-execute] DELETE step completed successfully.
```

Figure 17: Logs for DELETE case showing successful structural delete of Tom Hanks.

In Figure 18 below, the CLI does a final print out after the DELETE query to confirm that compared to the original graph state, the Tom Hanks vertex is indeed no longer in the graph.

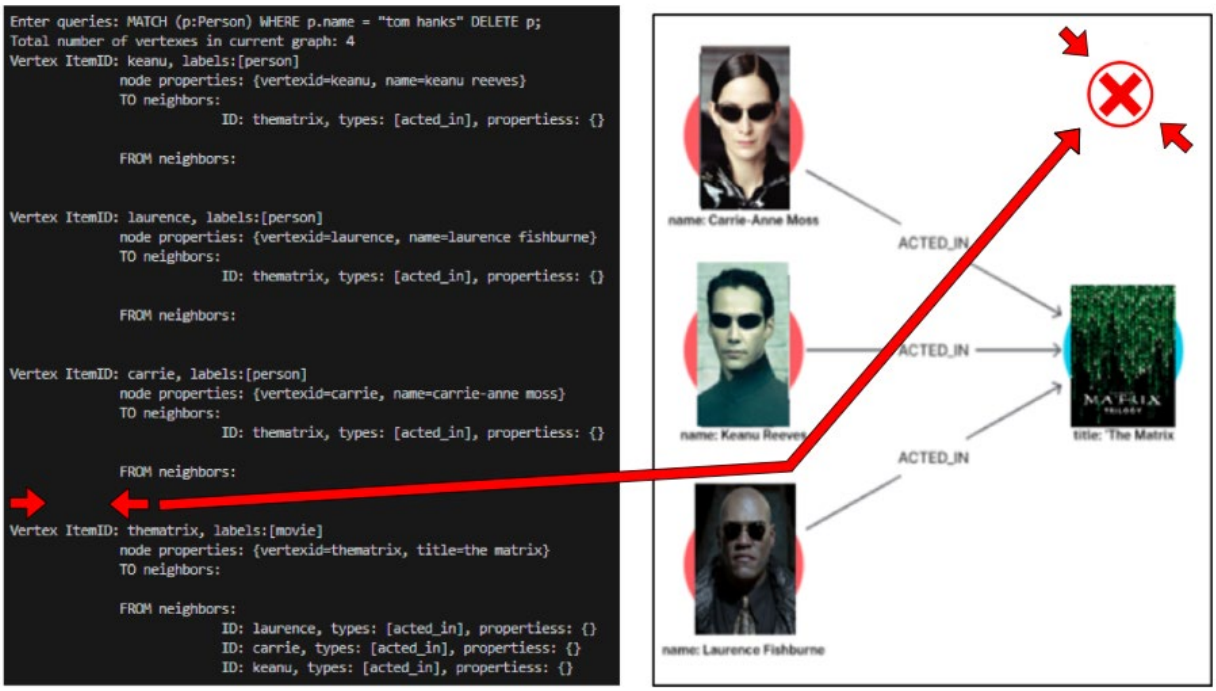

Figure 18: Successful DELETE of Tom Hanks PropertyGraphVertex.

5.2 Execution of DETACH DELETE

Now that Tom Hanks has been successfully removed to showcase the DELETE scenario, a test for DETACH DELETE will be shown. As seen in Figure 19 below, the logs show that we are in the parent-child lifecycle and the parent agent has been correctly deployed to The Matrix vertex.

```
600   21:04:51.721 [main] [DEBUG] [PropertyGraphAgent-executeDelete] agent=3 place=thematrix originNodeID=null deleteTargets=[thematrix]
601   21:04:51.721 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Starting executeDelete at vertex node: thematrix
602   21:04:51.721 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Parent agent at delete target thematrix
603   21:04:51.721 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] vertex=thematrix neighbor=null direction=both toCount=0 fromCount=3
604   21:04:51.721 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] Removing ALL edges at vertex thematrix
605   21:04:51.721 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] END vertex=thematrix toCount=0 fromCount=0
```

Figure 19: Logs showing DETACH DELETE parent agent at The Matrix PropertyGraphVertex.

Next, the logs show in Figure 20 that the child agents have been subsequently spawned at adjacently connected actor vertices that have a directed relationship with The Matrix. The PropertyVertexPlace's detachIncomingOrOutgoingEdge successfully decouples the relationships from the child vertices for each of the 3 actors.

```
21:04:51.729 [main] [DEBUG] [PropertyGraphAgent-executeDelete] agent=0 place=keanu originNodeID=thematrix deleteTargets=null
21:04:51.729 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Starting executeDelete at vertex node: keanu
21:04:51.729 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Child agent detaching from origin thematrix at vertex keanu
21:04:51.729 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] vertex=keanu neighbor=thematrix direction=both toCount=1 fromCount=0
21:04:51.729 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] Detached OUTGOING edge from thematrix at keanu
21:04:51.729 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] END vertex=keanu toCount=0 fromCount=0
21:04:51.729 [main] [DEBUG]                         MASSBase PropertyGraph notNull setReturns at index: 1, with results: []
21:04:51.729 [main] [DEBUG] Thread [0]: (1) has called its method;
21:04:51.729 [main] [DEBUG] At AgentsBase callAll, agent ID: 2
21:04:51.729 [main] [DEBUG] [PropertyGraphAgent-executeDelete] agent=2 place=laurence originNodeID=thematrix deleteTargets=null
21:04:51.729 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Starting executeDelete at vertex node: laurence
21:04:51.729 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Child agent detaching from origin thematrix at vertex laurence
21:04:51.729 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] vertex=laurence neighbor=thematrix direction=both toCount=1 fromCount=0
21:04:51.729 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] Detached OUTGOING edge from thematrix at laurence
21:04:51.729 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] END vertex=laurence toCount=0 fromCount=0
21:04:51.729 [main] [DEBUG]                         MASSBase PropertyGraph notNull setReturns at index: 0, with results: []
21:04:51.730 [main] [DEBUG] Thread [0]: (0) has called its method;
21:04:51.730 [main] [DEBUG] At AgentsBase callAll, agent ID: 1
21:04:51.730 [main] [DEBUG] [PropertyGraphAgent-executeDelete] agent=1 place=carrie originNodeID=thematrix deleteTargets=null
21:04:51.730 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Starting executeDelete at vertex node: carrie
21:04:51.730 [main] [DEBUG] [PropertyGraphAgent-executeDelete] Child agent detaching from origin thematrix at vertex carrie
21:04:51.730 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] vertex=carrie neighbor=thematrix direction=both toCount=1 fromCount=0
21:04:51.730 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] Detached OUTGOING edge from thematrix at carrie
21:04:51.730 [main] [DEBUG] [PropertyVertexPlace-detachIncomingOrOutgoingEdge] END vertex=carrie toCount=0 fromCount=0
```

Figure 20: Logs showing DETACH DELETE child agent phase at actors starring in The Matrix.

Lastly, in Figure 21 the structural delete phase begins after all agents have finished working and what results is the log file showing only the 3 remaining actors, without The Matrix movie to be seen.

```
21:04:51.731 [main] [DEBUG] [DeleteExecutionStep-execute] Beginning structural delete phase
21:04:51.731 [main] [DEBUG] [DeleteExecutionStep-execute] Calling deleteVertex(thematrix, detach= true)
21:04:51.732 [main] [DEBUG] [PropertyGraphPlaces-remoteVertexFromPlaces] Vertex deleted successfully, itemID= thematrix, internalVertexID= 4, localIndex= 4
21:04:51.732 [main] [DEBUG] [DeleteExecutionStep-execute] Structural delete phase complete
21:04:51.732 [main] [DEBUG] [DeleteExecutionStep-execute] Printing graph after DELETE
21:04:51.732 [main] [DEBUG] Printing the graph with node properties and relationship information: ============
21:04:51.732 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: keanu, toRelationship: 0, fromRelationship: 0
21:04:51.732 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: laurence, toRelationship: 0, fromRelationship: 0
21:04:51.732 [main] [DEBUG] At PropertyGraphModel, addPropertyModel, ID: carrie, toRelationship: 0, fromRelationship: 0
21:04:51.732 [main] [DEBUG] [DeleteExecutionStep-execute] DELETE step completed successfully.
```

Figure 21: Logs showing DETACH DELETE completed successfully for The Matrix movie.

As seen from the CLI, the graph is finally printed back out to the application user, which verifies what was shown in the log file, which in Figure 22 shows that only the 3 Matrix actors remain within the graph after both DELETE and DETACH DELETE operations took place in succession.

```
Enter queries: MATCH (m:Movie) WHERE m.title = "the matrix" DETACH DELETE m;
Total number of vertexes in current graph: 3
Vertex ItemID: keanu, labels:[person]
              node properties: {vertexid=keanu, name=keanu reeves}
              TO neighbors:

              FROM neighbors:


Vertex ItemID: laurence, labels:[person]
              node properties: {vertexid=laurence, name=laurence fishburne}
              TO neighbors:

              FROM neighbors:


Vertex ItemID: carrie, labels:[person]
              node properties: {vertexid=carrie, name=carrie-anne moss}
              TO neighbors:

              FROM neighbors:
```

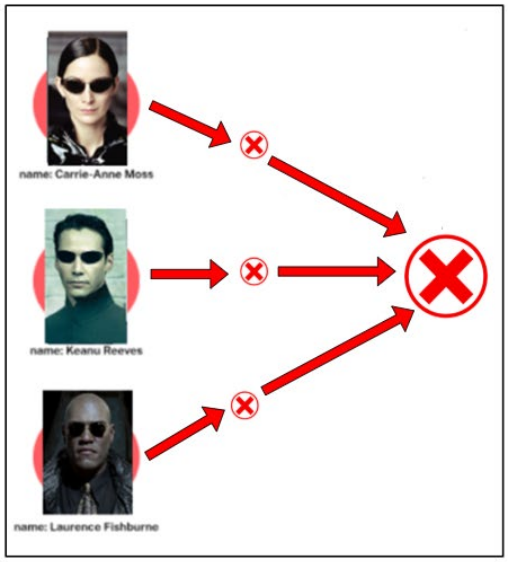

Figure 22: Successful DETACH DELETE of The Matrix PropertyGraphVertex.

6. Conclusion

This work so far during Winter quarter 2026 has extended the MASS graph query framework by introducing support for the DELETE and DETACH DELETE clauses within the declarative-to-imperative Cypher query execution pipeline. Prior to this implementation the system supported only read-oriented query operations such as MATCH and WHERE. The addition of deletion functionality represents a significant step toward enabling full graph mutation capabilities within the MASS environment.

To support these operations, a decentralized deletion strategy was developed using agent-based execution. Parent agents deployed at target vertices remove local relationships and propagate deletion context to neighboring vertices through spawning of child agents during graph management lifecycles. Structural vertex removal is then performed during the cleanup phase of the DeleteExecutionStep, ensuring correctness across the distributed graph.

While these simple tests showcase a successful implementation to finish out the quarter, future work will need to accomplish more tasks within DELETE and DETACH DELETE to confirm a more polished and set-in-stone final state. Tests will need to be run across multiple compute nodes in varying formats to ensure that communication between parent and child vertices along the edges of compute node boundaries are communicating properly. Similarly, additional tests on larger datasets where known agent conflictions on DETACH DELETE operations involving where 2^{nd} hop child spawning occurs will need to confirm that the current setup of idempotent edge cleanup operations rightly do not conflict with each other. There is success in these results, but further work remains to be done.

Beyond finishing and polishing the work for DELETE and DETACH DELETE, documentation will need to be written and published to the code repositories as well as peer review of the code implementation to help ensure that its written structure is cleanly coded and sufficient for logical peer comprehension and understanding. After those steps have been completed, work can begin on setting up the design and implementation of SET functionalities for the MASS parallelization library and the large undertaking to mimic the effective functionalities of the SET clause as outlined in Neo4j's Cypher documentation [22] while customizing the under-the-hood declarative-to-imperative translations and operations so that the graph can be fully manipulated and written to via distributed ABM methodologies.

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).

[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.

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