

CSS 595: An Enhancement of Distributed Graph Queries in an Agent-Based Graph Database

Term Report (SP25)

1. Introduction

Graph databases are a specialized type of database designed to handle highly interconnected data by leveraging graph theory principles. Unlike traditional databases, graph databases store nodes (entities) and edges (relationships) natively to enable efficient traversal and querying of connected data. These databases are particularly well-suited for applications like social networks, recommendation systems, and fraud detection, where understanding relationships is critical. There are different approaches to graph database modelling, including the RDF (resource description framework) model and the property graph model [3], which are the two most widely followed models. RDF model uses triplets (subject-predicate-object) to represent data, while the property graph model allows nodes and relationships to carry properties as key-value pairs. The RDF model can be rigid due to its structure and complex for certain application, while the property graph model is more flexible and is ideal for a dynamic use case requiring frequent schema changes and detailed relationship modelling.

The agent-based graph database system follows the property graph model, abstracting the underlying distributed layer built using the MASS (Multi-agent spatial simulation) Java library. In the distributed environment, the system operates across multiple computing nodes with the data split across the nodes and maintained in-memory for efficient access and manipulation. The MASS Java library excels at managing simulations where entities, represented as Agents, dynamically interact with each other and their environment, depicted as Places [5]. The graph database system is developed by extending the MASS Java library to allow data representation using the property graph model and querying using the agent entities.

Filters are an essential component of a database system, enabling users to extract data matching certain criteria while also reducing the computational load on the systems during reading operations. To fulfil this aspect in the current version of the agent-based graph database, my aim for this quarter has been the implementation of the filtering mechanism for reading operations in the graph database.

2. Background and Challenges

The current version of the MASS graph database provides basic functionality to the users in the form of creating, updating, and fetching results based on simple query patterns. It successfully incorporates the following aspects of the Cypher query language – CREATE, RETURN and MATCH clauses. The system follows a three-tier architecture pattern, as shown in Figure-1, separating the presentation, logic, and data access layers to provide high levels of

abstraction to the users such that they can interact with the database without any in-depth knowledge of the underlying structure. [1]

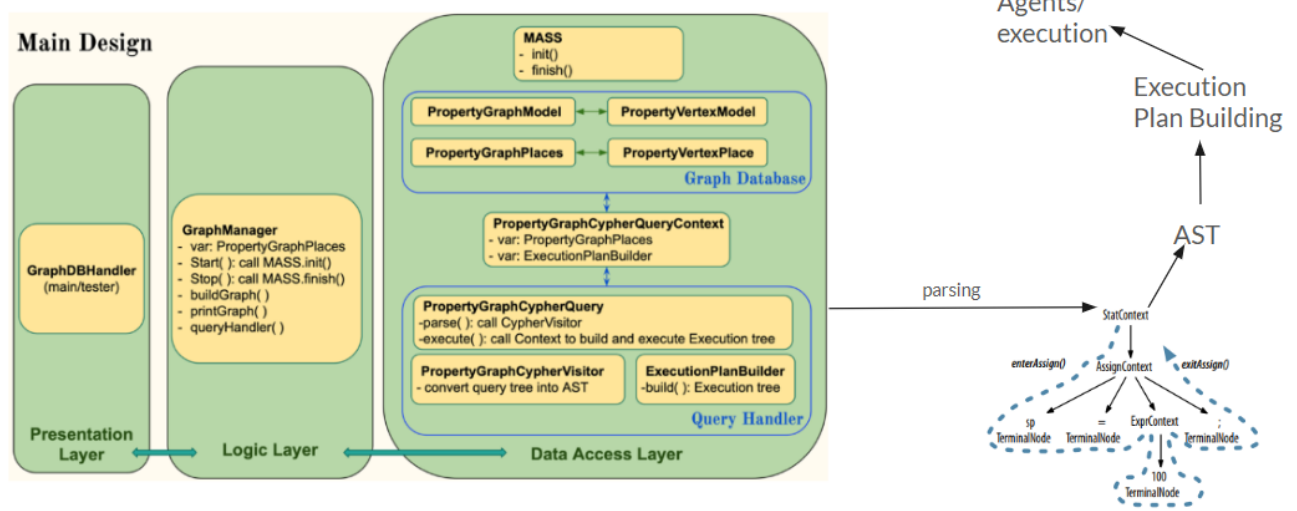


Figure-1: Three tier design for Graph Database System

The presentation layer acts as the point of interaction with the users, where users can build the graph based on input node and entity files and query the graph, upon successful graph creation. The logic layer is responsible for executing the graph creation and further interacting with the MASS system for fetching results for reading queries. The data access layer where the raw string inputs are parsed and converted into an abstract syntax tree (AST) representation in Java, which is eventually translated into agent code which manipulates the agent behaviours based on the input query. This layer is also responsible for fetching the results and providing them to the logic and presentation layers.

This system adopts the property graph model by extending the core MASS concepts, as shown in Figure-2, such as Places (serving as a spatial framework) and Agents (dynamic entities that interact within the spatial framework defined by Places) . Places is initially extended to support simple graph applications by GraphPlaces and further modified to support multiple properties for entities and relationships (property-graph model) by PropertyGraphPlaces.

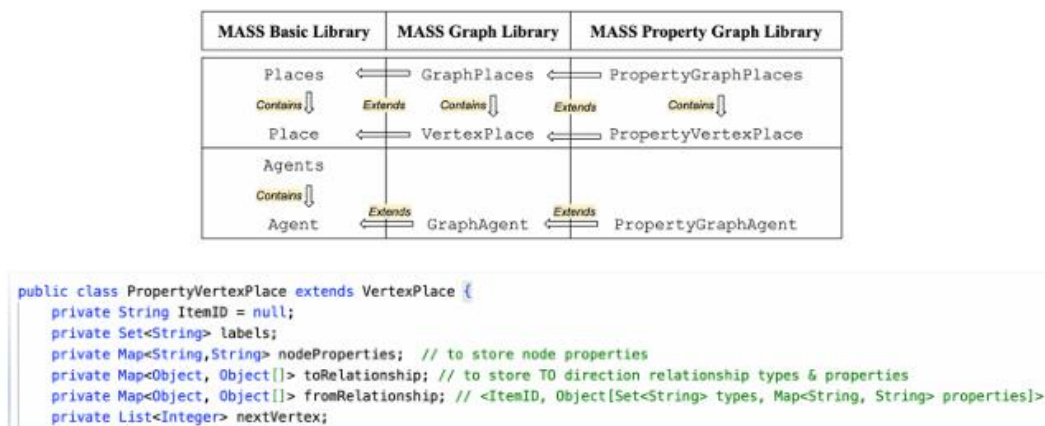


Figure-2: Adopting the property graph model in MASS

The query execution flow for the current system is based on a three-step process: **building the abstract syntax tree (AST) representation** of the cypher query, **defining the execution plan for agents** based on query content followed by **the execution and result aggregation**. The (AST) representation is built using the **ANTLR parsing tool** which makes it easy to customize how the query tree gets parsed and necessary information gets stored for generating the agent code [2]. The AST module is built to map the syntax specific to cypher grammar, which follows a hierarchy as shown in *Figure-3*, to Java classes with similar class hierarchy. Using the ANTLR’s visitor mechanism, the query tree is parsed and the important data (e.g. label names, property lookups, comparison operators etc.) gets stored in the respective AST class. The following figure provides a visualization of the parse tree that needs to be traversed for a cypher clause using the ANTLR parsing tool.

Figure-3: Parse tree structure for a CypherOL query (e.g. MATCH clause with a WHERE component)

Once the entire query is parsed and the data stored in AST format, we move onto the execution plan building phase. This phase defines how the extracted data gets utilized and converted into arguments for agents. The execution plan is essentially a tree of execution steps, which is required to fulfil the query efficiently. The mapping of classes and respective methods different steps of the query execution flow (AST parsing, Execution Plan building and the query execution by Agents), is shown in the Table-1 as it connects the parse tree rules (*from Fig-3*), with class hierarchy of execution plan builder (*Fig-4*).

Table - 1: Mapping between the parse-tree rules, AST classes, and execution plan builder classes

Parse Tree	CypherVisitor function	AST	ExecutionPlanBuilder function	Execution Step
oC_CREATE	visitOC_Create()	CypherCreateClause	visitCreateClause()	JoinExecutionStep
oC_PatternPart	visitOC_PatternPart()	CypherPatternPart	visitCreateClausePatternPart()	CreatePatternExecutionStep
oC_NodePattern	visitOC_NodePattern()	CypherNodePattern	visitCreateNodePattern()	CreateNodePatternExecutionStep
oC_RelationshipPattern	visitOC_RelationshipPattern()	CypherRelationshipPattern	visitCreateRelationshipPattern()	CreateRelationshipPatternExecutionStep

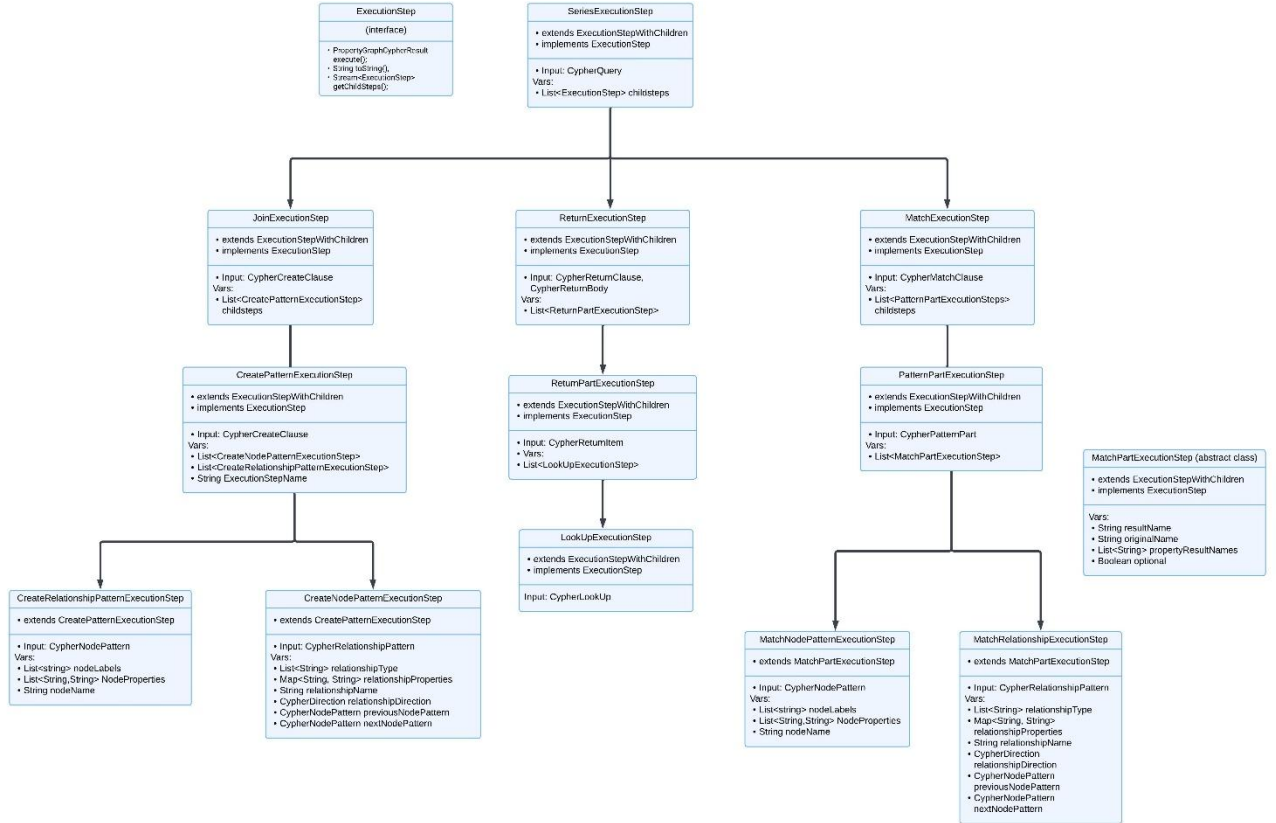


Figure-4: Execution step tree determining the steps in an execution plan

The MASS graph database system implements various Cypher clauses, each requiring distinct agent behaviors. Clauses like CREATE (for node/relationship creation) operate independently, while others like MATCH and RETURN function as interdependent components within a single query. This variation in clause functionality necessitates a structured execution framework.

Figure-4 illustrates the hierarchical execution plan that guides agent behavior during query processing. This hierarchy serves as a critical blueprint, defining the precise sequence of operations agents must perform when interpreting different clause combinations. The structured class relationship shown in the diagram ensures that agents can systematically process complex queries by following predefined execution paths, while maintaining the specific operational requirements of each clause type. This architectural approach enables the

system to handle both standalone operations and multi-clause query patterns with consistent behavior.

The final step in the process is the execution of the plan by agents, as depicted visually in *Figure-5*. Based on the execution step, agents are provided with corresponding attributes to verify and migrate or terminate. PropertyGraphAgents carry the pathResult attribute to store the path traversal results and inherit them from the parent agent (if any). To facilitate coordination agents, multiple iterations of callAll() and manageAll() are invoked for verifying if the current Place satisfies the node pattern information. If validated, the current place's ID is added to its pathResults list, and neighbor places are determined via the relationship pattern and updated in its nextVertex field, facilitating agents to migrate to its neighbor places. Subsequently, manageAll() is called to coordinate the lifecycle of agents, encompassing the stages of spawn, kill, and migration.

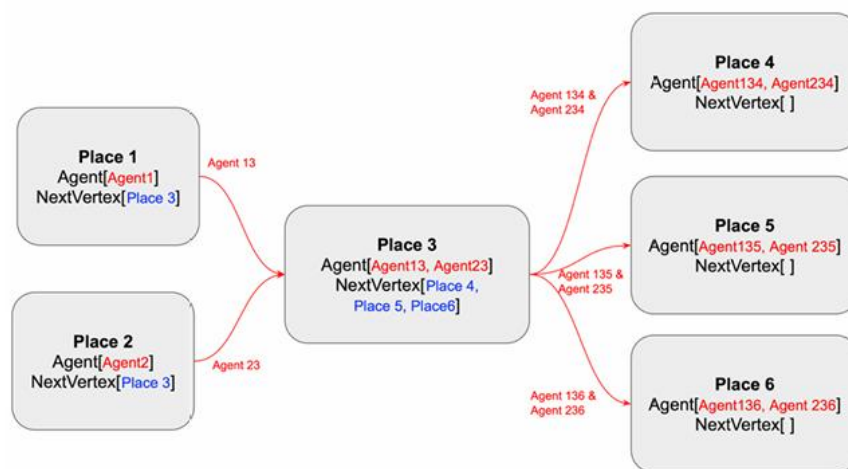


Figure-5: Agent propagation in MASS PropertyGraphPlaces

The current state of MASS graph database system allows for CREATE, MATCH and RETURN clauses, but has some limitations in terms of the constraints that can be put for the queries. In the current state it also does not allow for deletion and does not support the “filters” for a query, which are essential for a database system.

The challenges associated with building a filter mechanism for the MASS graph database lies in determining the approach for checking constraints for different entities and relationships based on the MATCH pattern. For example, in the following pattern:

```
MATCH (p:person)-[r:acted_in]->(m:movie) WHERE p.age > 35 AND m.rating >=4.0
RETURN m,p
```

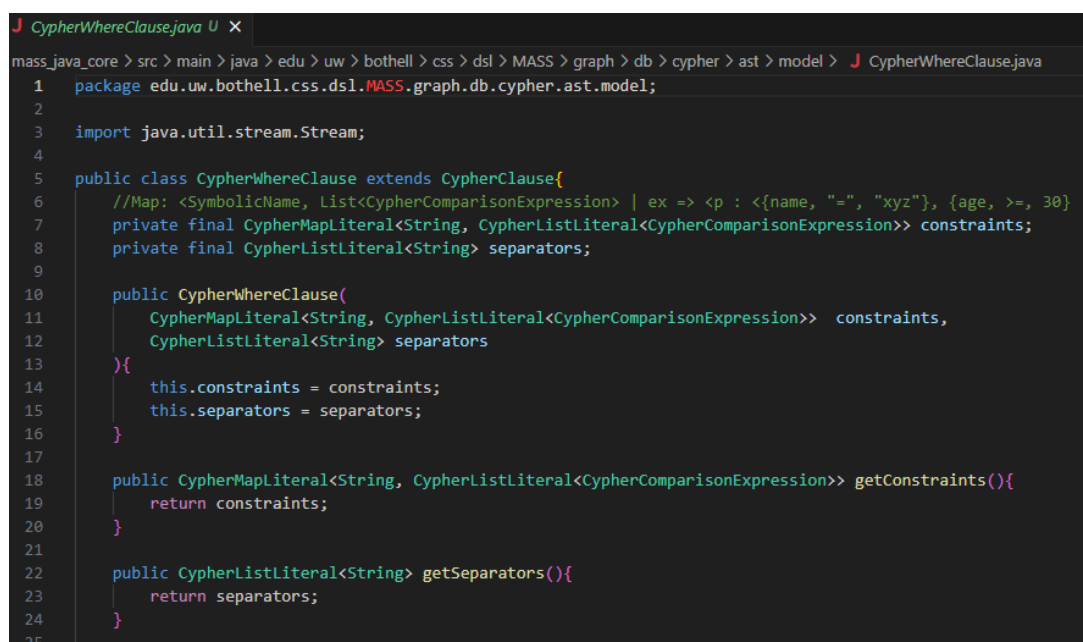
- Agents **only have access to one node and the corresponding relationship information** at a given point of time
- **Using constraints after traversing all the paths** with the MATCH pattern might provide the correct results to the user, but **would inherently not provide any optimization/reduce computational load on the system** while doing reading operations.

3. Autumn Quarter Implementation (AU24)

My focus for this quarter was working on the WHERE clause implementation. WHERE clause in Cypher is used to filter query results by adding conditions to patterns specified in the MATCH or OPTIONAL MATCH clauses. It functions similarly to the WHERE clause in SQL, allowing users to refine their queries by applying constraints on node properties, relationships, or patterns [4]. For example, it can be used to filter nodes based on property values, check for the existence of properties, or apply multiple conditions using logical operators like 'AND', 'OR', and 'XOR'. This makes it a powerful tool for retrieving specific data from a graph database.

The approach for incorporating the WHERE clause in the MASS graph database is as follows:

1. AST representation: Within the Java AST module, building a new class to represent the structure of cypher WHERE clause. Splitting the constraints to a List of expressions and separators (Boolean operators – AND, OR, NOT), as shown in *Figure-6*, (line 5 – 7).
2. Execution plan building: Making changes within the parent MatchPartExecutionStep, to provide the agents with more attributes, namely: constraints (list of expressions), separators (list of Boolean operators) and partial_results (a list of Booleans, storing the corresponding results of constraints after verification by agents)
3. Agent code: In addition to pathResult, the PropertyGraphAgents will also pass along the partial_results to the children instance.
4. To optimize constraint checking, implementing a 'short-circuit' evaluation strategy for Boolean logic.



```
1 package edu.uw.bothell.css.dsl.MASS.graph.db.cypher.ast.model;
2
3 import java.util.stream.Stream;
4
5 public class CypherWhereClause extends CypherClause{
6     //Map: <SymbolicName, List<CypherComparisonExpression> | ex => <p : <{name, "=", "xyz"}, {age, >=, 30}>
7     private final CypherMapLiteral<String, CypherListLiteral<CypherComparisonExpression>> constraints;
8     private final CypherListLiteral<String> separators;
9
10    public CypherWhereClause(
11        CypherMapLiteral<String, CypherListLiteral<CypherComparisonExpression>> constraints,
12        CypherListLiteral<String> separators
13    ){
14        this.constraints = constraints;
15        this.separators = separators;
16    }
17
18    public CypherMapLiteral<String, CypherListLiteral<CypherComparisonExpression>> getConstraints(){
19        return constraints;
20    }
21
22    public CypherListLiteral<String> getSeparators(){
23        return separators;
24    }
25 }
```

Figure-6: AST representation for WHERE clause, initial version

The flowchart in *Figure-7* illustrates a comprehensive process for extending Cypher clauses in the MASS graph database system, including the development of the Cypher WHERE clause. This approach begins with building the Abstract Syntax Tree (AST) representation of the clause, followed by modifying the PropertyGraphCypherVisitor class methods based on the parse tree. The data is stored using the AST module class. Execution steps classes are then constructed for each clause, with necessary adjustments made in the ExecutionPlanBuilder class methods. The process also involves mapping these changes to agent behavior, such as adding new attributes or methods, and optimizing the AST representation. This incremental and systematic methodology ensures effective feature development and functionality enhancement.

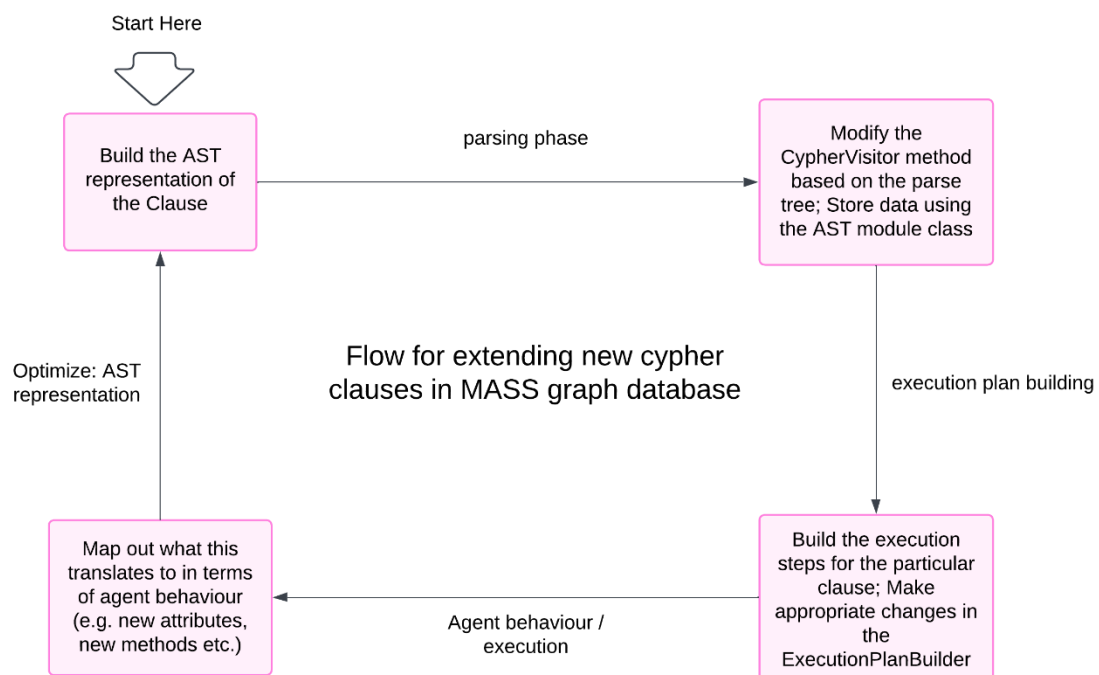


Figure-7: Flow for incrementally building new features/extending cypher clauses in MASS graph database

5. Winter Quarter Implementation (WI25)

This quarter's focus was working on the WHERE clause, extending the previous quarter's implementation along with some changes made to incorporate the evaluation approach for the WHERE clause expression. The following implementation entails the details for the completion of parsing phase (using ANTLR tool) of Cypher Where clause for MASS graph database system.

1. Abstract Syntax Tree (AST)

Taking into consideration the arguments expected for providing additional set of constraints to Agents during the Match clause execution, the AST representation of CypherWhereClause was modified to accommodate the postfix notation of the expression along with the constraints mapped based on symbolic names, as shown in *Fig-8 (lines 7-12)*.

```

1 package edu.uw.bothell.css.dsl.MASS.graph.db.cypher.ast.model;
2
3 import java.util.stream.Stream;
4 import java.util.List;
5
6 public class CypherWhereClause extends CypherClause{
7     //Map: <SymbolicName, List<String> | ex => < Symbolic_name : <{"prop_name", "comparator_op", "value", "exp_id", "truth_val"}
8     //>> .... e.g. {p:{age, >=, 30, exp1, false}}
9     // -> (e1 AND (e2 OR e3)) => e1 e2 e3 OR AND
10    //expressions = [e1,e2,e3,OR,AND]
11    private final CypherMapLiteral<String, CypherListLiteral<List<String>>> constraints;
12    private final CypherListLiteral<String> expression_list;
13
14    public CypherWhereClause(
15        CypherMapLiteral<String, CypherListLiteral<List<String>>> constraints,
16        CypherListLiteral<String> expression_list){
17        this.constraints = constraints;
18        this.expression_list = expression_list;
19    }

```

Figure-8: AST representation for WHERE clause, modified version

To build the constraint map along with the unique expression ID (representing each sub-expression / the smallest individual expression) while parsing the input query's where-expression, another helping class – CypherWhereContext was built. This class generates unique expression IDs to keep track of the sub-expressions in WHERE clause and store them in the constraint map for efficient retrieval, as shown in *Figure-9 (lines 8 – 19)*. This class object is used as a shared resource and passed downstream in the parsing tree based on CypherQL's Grammar (*more details in the appendix section-a*).

```

1 package edu.uw.bothell.css.dsl.MASS.graph.db.cypher.ast.model;
2
3 import java.util.HashSet;
4 import java.util.List;
5 import java.util.Set;
6
7 public class CypherWhereContext {
8     //Map: <SymbolicName, List<String> | ex => < Symbolic_name : <{"prop_name", "comparator_op", "value", "exp_id", "truth_val"} >>
9     private CypherMapLiteral<String, CypherListLiteral<List<String>>> constraints_map;
10    private CypherListLiteral<String> expression_list;
11    private Integer expressionIDCounter;
12    private final Set<String> generatedExpIDs;
13
14    public CypherWhereContext(CypherMapLiteral<String, CypherListLiteral<List<String>>> constraint_map,
15        CypherListLiteral<String> expression_list){
16        this.constraints_map = constraint_map;
17        this.expression_list = expression_list;
18        this.expressionIDCounter = 0;
19        this.generatedExpIDs = new HashSet<>();
20    }

```

Figure-9: Helper AST class – CypherWhereContext

2. Testing with different cases:

Fig-10 to Fig-12 shows the contents of the constraints map and expression list attributes of the CypherWhereClause AST class (*Fig-8*), after the parsing phase is completed.

- Simple Expression:

WHERE p.age> 25 OR m.rating > 4

=> Infix: "exp0 | exp1"
=> Postfix: "exp0 exp1 |"

```
[aatmanrp@hermes01 ~]$ /usr/bin/env /usr/lib/jvm/java-11-openjdk-11.0.22.0.7-1.el7_9.x86_64/bin/java @/tmp/cp_e8crhb0ibbbuhonv87wxzbow.a
leParsing
LHS tree returned UID: exp0
In the RHS tree updating UID: exp0
Fetching RHS
LHS tree returned UID: exp1
In the RHS tree updating UID: exp1
Fetching RHS
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 25, exp0, TRUTH_VALUE], m: [rating, >, 4, exp1, TRUTH_VALUE]}
expression_list: exp0, exp1, |
[aatmanrp@hermes01 ~]$
```

Figure-10: WHERE clause parser test case-1

- Complex Nested Expression:

WHERE p.age > 25 AND (m.rating > 4 OR m.budget < 500000)

=> Infix: "exp0 & (exp1 | exp2)"
=> Postfix: "exp0 exp1 exp2 | &"

```
LHS tree returned UID: exp0
In the RHS tree updating UID: exp0
Fetching RHS
Fetching RHS
LHS tree returned UID: exp1
In the RHS tree updating UID: exp1
Fetching RHS
LHS tree returned UID: exp2
In the RHS tree updating UID: exp2
Fetching RHS
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 25, exp0, TRUTH_VALUE], m: [rating, >, 4, exp1, TRUTH_VALUE], [budget, <, 500000, exp2, TRUTH_VALUE]}
expression_list: exp0, exp1, exp2, |, &
[aatmanrp@hermes01 ~]$
```

Figure-11: WHERE clause parser test case-2

- Parenthesized expression with multiple constraints on the same key:

WHERE m.release_year = 1995 OR (m.rating > 4 AND m.budget > 100000)

=> Infix: "exp0 | (exp1 & exp2)"
=> Postfix: "exp0 exp1 exp2 & |"

```
[aatmanrp@hermes01 ~]$ /usr/bin/env /usr/lib/jvm/java-11-openjdk-11.0.22.0.7-1.el7_9.x86_64/bin/java @/tmp/cp_e8crhb0ibbbuhonv87wxzbow.a
leParsing
LHS tree returned UID: exp0
In the RHS tree updating UID: exp0
Fetching RHS
Fetching RHS
LHS tree returned UID: exp1
In the RHS tree updating UID: exp1
Fetching RHS
LHS tree returned UID: exp2
In the RHS tree updating UID: exp2
Fetching RHS
====>>>CypherWhereClause content:
WHERE
constraints-map: {m: [release_year, =, 1995, exp0, TRUTH_VALUE], [rating, >, 4, exp1, TRUTH_VALUE], [budget, >, 100000, exp2, TRUTH_VALUE]}
expression_list: exp0, exp1, exp2, &, |
[aatmanrp@hermes01 ~]$
```

Figure-12: WHERE clause parser test case-3

Based on the above implementation, the following is the approach towards integrating the data stored in the AST representation with the agent code.

3. Expression Evaluation Approach and Integration:

The graph database WHERE conditions are to be evaluated using a **stack-based approach**. As agents traverse the graph, they populate the constraint-map with truth value results for individual constraints (e.g., "prop_name > 30"). The core of this evaluation relies on the representation of the WHERE expression in postfix notation and then using a stack to process operators and operands sequentially. During postfix evaluation, operands (truth values from constraints) are pushed onto the stack, while operators pop the required number of operands from the stack, apply the logical operation (AND, OR, NOT, XOR), and push the result back onto the stack. This stack-based method efficiently handles complex boolean expressions with proper operator precedence (NOT > AND > XOR > OR), allowing the WHERE clause to correctly filter graph paths by evaluating nested conditions without requiring recursive parsing. The result is obtained when all tokens are processed and only a single boolean value remains on the stack, determining whether the current path satisfies all conditions.

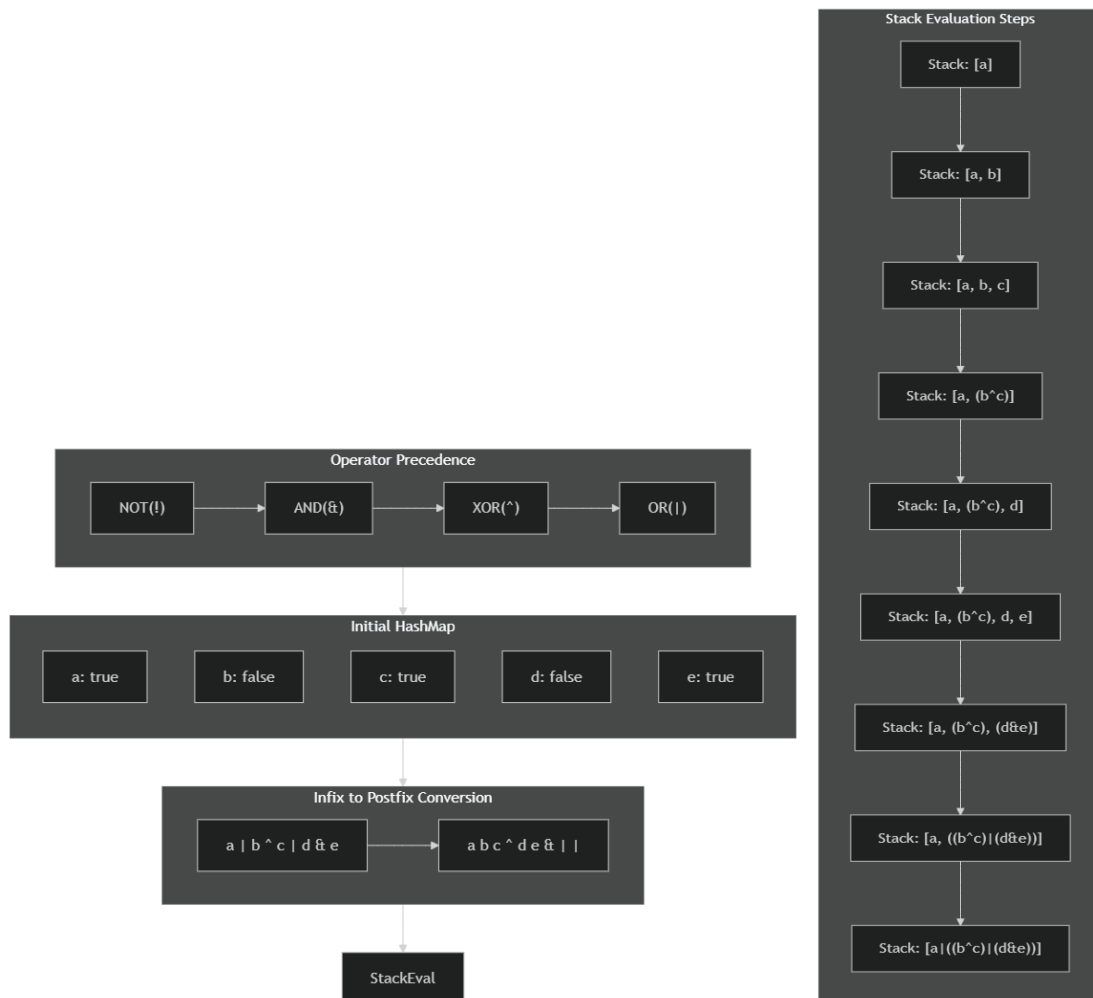


Figure-13: A visual representation of the stack-based expression evaluation approach

6. Spring Quarter Implementation (SP25)

This quarter's focus was on completing the implementation of WHERE clause in MASS graph database by integrating the above-mentioned expression evaluation mechanism with agent migration. The outputs of WHERE clause were verified for correctness and benchmarking was performed using the IMDB movies dataset. The details are as follows: -

1. WHERE clause integration changes:

The changes made to the existing implementation, on the execution planning and agent code execution phase include:

- PropertyGraphAgent & Agent arguments:

To provide agents with the context of the WHERE clause, modifications were made to the PropertyGraphAgent class itself and additional set of data added to the agent arguments array. These are the arguments/attributes that enable the PropertyGraphAgents to evaluate if they are on the correct vertex on the graph according to the details extracted from the query patterns.

Two new attributes – **constraintsMap** (a hashmap containing the Where clause constraints based on the generated AST) and **expressionList** (a postfix notation representing the logical expression in the WHERE clause), were added to the PropertyGraphAgent class (lines 48 to 66).

```
46 public class PropertyGraphAgent extends Agent {
47
48     protected HashMap<String, List<List<String>>> constraintsMap;
49     protected List<String> expressionList;
50     protected ArrayList<String> pathResult;
51
52     // PropertyGraphAgent need to carry information about MATCH query pathResult
53     // The parameter args is the pathResult of its parent Agent (if any)
54     // Object args format - [pathResult, constraintsMap, expressionList]
55     @SuppressWarnings("unchecked")
56     public PropertyGraphAgent(Object args) {
57         super();
58         //TODO - Changes for WHERE Clause
59         if(args instanceof AgentInitArgs) {
60             this.pathResult = (((AgentInitArgs)args).pathResult != null) ? new ArrayList<String>(((AgentInitArgs)args).pathResult) : new ArrayList<String>();
61             this.constraintsMap = (((AgentInitArgs)args).constraintsMap != null) ? deepCopyConstraintsMap(((AgentInitArgs)args).constraintsMap) : null;
62             this.expressionList = (((AgentInitArgs)args).expressionList != null) ? new ArrayList<String>(((AgentInitArgs)args).expressionList) : null;
63         } else {
64             this.pathResult = new ArrayList<String>();
65             this.constraintsMap = null;
66             this.expressionList = null;
67         }
68     }
}
```

Figure-14: Modified PropertyGraphAgent class

Two new arguments – **symbolicName pair** (array containing current node and relationship's symbolic names) and **iteration number pair** (array containing the current iteration and total iterations based on query pattern), were added to the existing list of five agent arguments (lines 67 to 77).

```

37 public MatchPatternPartExecutionStep(String resultName, CypherWhereClause whereExpression, MatchPartExecutionStep... matchPartExecutionSteps) {
45 //preparing initializer args for the Agent
46 if(whereExpression!=null){
47     this.agentInitializer = new AgentInitArgs(pathResult:null, convertCypherMapToHashMap(whereExpression.getConstraintsMap()), convertCypherListToList(whereExpression.getExpressionList()));
48 }else{
49     this.agentInitializer = new AgentInitArgs(pathResult:null, constraintsMap:null, expressionList:null);
50 }
51
52 // prepare arguments send to MASSBase for Agents to fetch match clause results
53 for(int i = 0; i < matchPartExecutionSteps.length; i++) {
54     MatchNodePartExecutionStep nodeStep = (MatchNodePartExecutionStep) matchPartExecutionSteps[i];
55     nodeNumber++;
56     nodesResultNames.add(nodeStep.getResultName());
57     i++;
58     MatchRelationshipPartExecutionStep relStep = i < matchPartExecutionSteps.length ? (MatchRelationshipPartExecutionStep) matchPartExecutionSteps[i] : null;
59
60     //changes for WHERE clause : added args - symbolic names pair [nodeSymName, relSymName]; iteration number pair [currentIter, nodeNumber]
61     Object[] currArgs = new Object[7];
62     currArgs[0] = (Object) nodeStep.getLabelNames();
63     currArgs[1] = (Object) nodeStep.getNodeProperties();
64     currArgs[2] = relStep == null? (Object) "NULL" : (Object) relStep.direction;
65     currArgs[3] = relStep == null? (Object) "NULL" : (Object) relStep.relTypes;
66     currArgs[4] = relStep == null? (Object) "NULL" : (Object) relStep.relProperties;
67     String nodeSymbolicName = nodeStep.getSymbolicName();
68     String relSymbolicName = relStep == null? "EMPTY" : relStep.getSymbolicName();
69     String[] symbolicNames = {nodeSymbolicName, relSymbolicName};
70     currArgs[5] = (Object) symbolicNames;
71     arguments.add(currArgs);
72 }
73 //add the last arg: iteration num pair [currentIteration, totalIterations(==nodeNumber)]
74 for(int i=0;i<arguments.size();i++){
75     Integer[] matchIterations = {i+1, nodeNumber};
76     arguments.get(i)[6] = (Object) matchIterations;
77 }
78 }

```

Figure-15: Modified set of arguments for agents

- Updating the constraintsMap:**
 As the agent perform validation checks for each vertex and edge to determine if it should migrate or terminate, the constraintsMap update the truth values of sub-expressions based on the current node and relationship. Once the entire constraintsMap is populated with the truth values, we perform the stack-based evaluation described above using the trigger mechanism.
- Trigger mechanism for WHERE clause evaluation:**
 Once the agents reach the destination node, there needs to be an evaluation to determine whether the retrieved path passes the conditions of the WHERE clause. This evaluation gets triggered when the current_iteration equals the total_iterations in the agent arguments. Based on the outcome of the stack-based evaluator, the current pathResult is aggregated to results only if the outcome is 'true'.
- Execution phase visualization:**
 The execution phase for handling queries with WHERE clauses involves both argument preparation and agent migration, as shown in the figure below:
 - Argument-Preparation:**
 The execution plan builder decomposes the Cypher query into multiple stages. For each stage, it assigns:
 - The symbolic name of the node/relationship.
 - The node labels, relationship type and properties (if specified).
 - Any directional information required for relationship traversals.
 - constraintsMap and expressionList from the CypherWhereClause class in parsing phase.
 These arguments are packaged into agent payloads before dispatch.
 - Agent-Migration:**
 As agents traverse through MASS Places (nodes), they carry these arguments and constraints. At each place:

- Agents verify the correct path using node/edge properties & keep updating the constraintsMap.
- Evaluate applicable constraints once they reach the final node based on query pattern.
- Clear the pathResult if the evaluator output is 'false'.

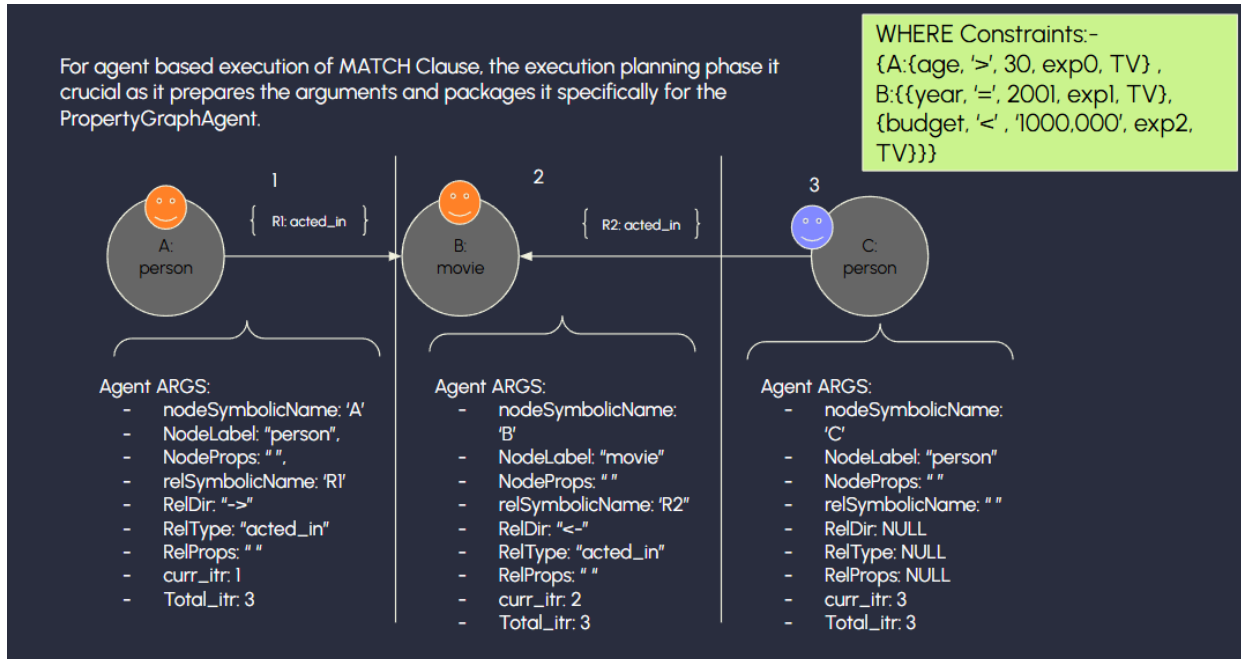


Figure-16: Visualization of agent argument preparation

2. Verifying the correctness of the implementation:

```
Enter queries: MATCH (m:movie) WHERE m.year = 1996 RETURN m
Printing Query Results:
m = movie_61
m = movie_63
m = movie_64
m = movie_65
m = movie_66
m = movie_70
m = movie_74
m = movie_79
m = movie_86
m = movie_87
m = movie_88
m = movie_92
m = movie_94
m = movie_95
m = movie_100
Query handling time: 50
```

Figure-17: Execution example-1 of WHERE clause query

```
Enter queries: MATCH (m:movie)<-[r:acted_in]-(p:person) WHERE m.year < 1996 AND p.bornin = 'australia' RETURN m,p
Printing Query Results:
m = movie_2    p = person_8537
Query handling time: 89
Enter queries:
```

Figure-18: Execution example-2 of WHERE clause query

3. Initial benchmarks using the IMDB movies dataset:

The IMDB movies dataset supports diverse query types: simple lookups, property filters, and nested logical conditions. It was transformed into the MASS graph database CSV format and here is a brief overview of it:

Node Types:

- *Person*: Name, person_imdbid, born
- *User*: Name
- *Movie*: Title, Year, movie_imdbid, budget, runtime, imdb_votes, released, imdb_rating

Relationship Types:

- *Acted_in* (Role),
- *Directed*,
- *Rated* (rating (out of 5), Timestamp)

Dataset Size:

- ~1,100 Nodes
- ~8,130 Relationships

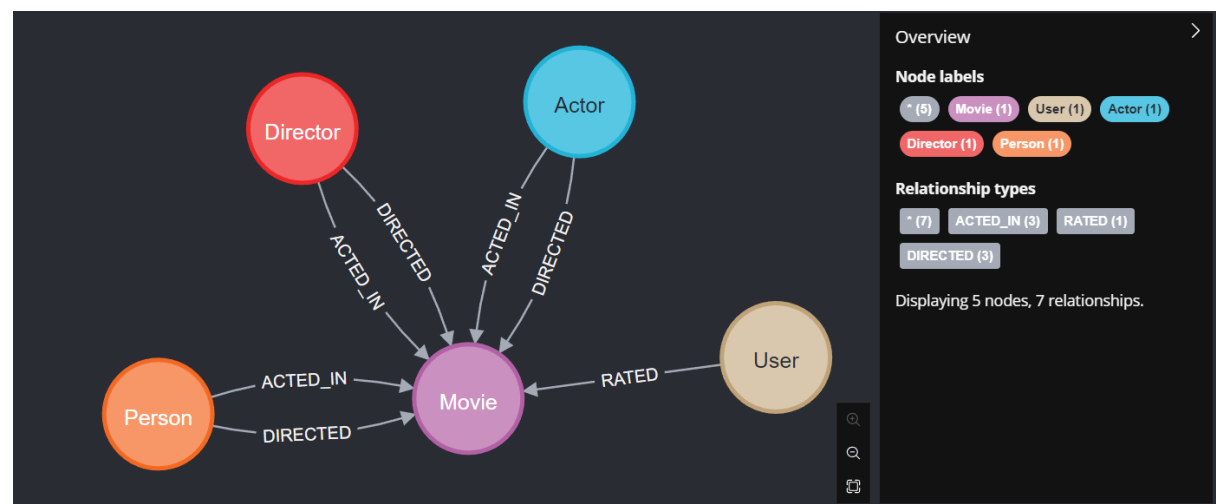


Fig-19: IMDB dataset visualization

Initial Benchmarking was performed using the following query types:

- Single Entity (no filters): `MATCH (m:movie) RETURN m`
- Single Entity (1 constraint): `MATCH (m:movie) WHERE m.year = 1996 RETURN m`
- Single Entity (Multiple Nested Constraints): `MATCH (m:movie) WHERE m.imdbrating > 7 AND (m.year >= 1995 AND m.year < 2000) RETURN m`

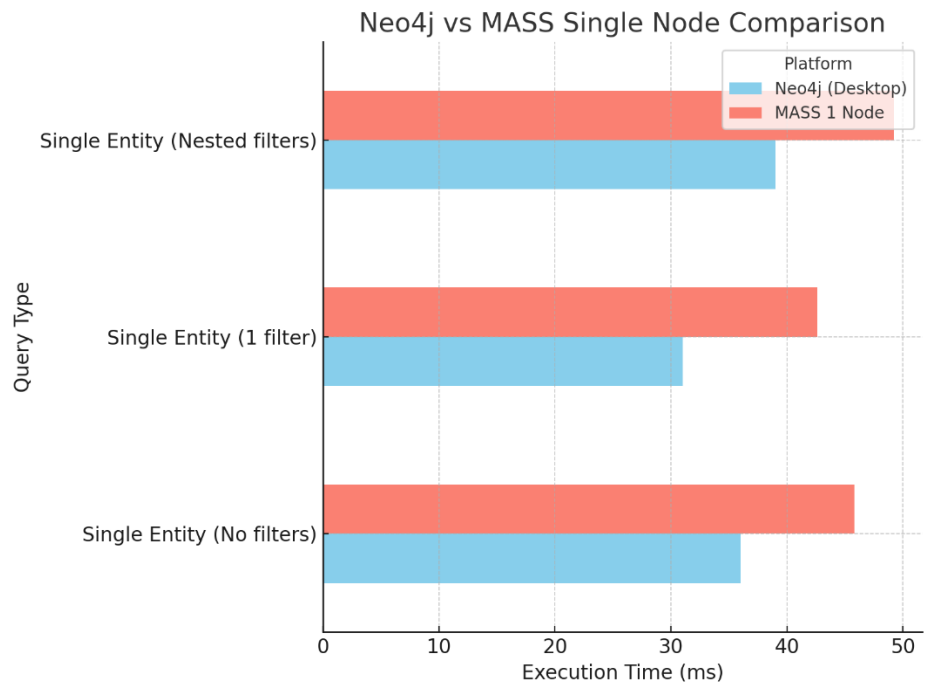


Fig-20: Comparison of average execution times on Neo4j vs MASS Graph DB (single compute node)

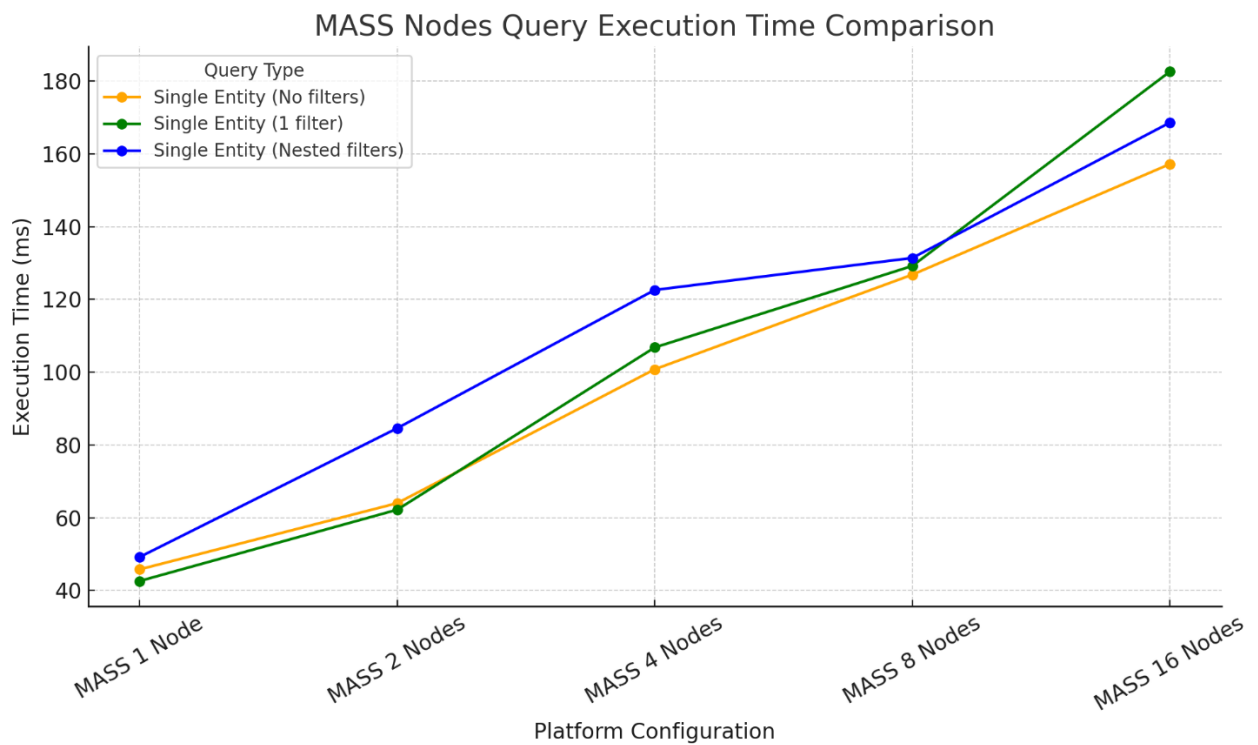


Fig-21: Average execution times of queries in MASS Graph DB over different number of compute nodes

Since systems like Neo4j only work on a single compute instance, we compared its query execution times with MASS Graph Database on a single compute node. As per the visualization in figure-20, we can infer that both systems have quite comparable execution times (< 10 ms of difference) which are consistent along all three query types. The slightly quicker execution observed in Neo4j can be attributed to its caching mechanisms and optimized querying using Anchor Nodes (appendix – Section a, 1). MASS Graph DB does not utilize any such caching or optimizations currently.

As MASS Graph Database is a distributed graph database, it can create and maintain a graph across multiple cluster nodes. This is especially useful when the graph sizes increase significantly (i.e. larger than the physical memory) and we exhaust a single machine's memory space. In such cases systems like Neo4j cannot store the graph, yet alone querying it. Our aim with MASS Graph Database is to support querying under such scenarios along with the advantage to scale horizontally by adding more compute nodes. Additionally, when systems like Neo4j get many simultaneous queries beyond its capabilities we expect MASS Graph Database to outperform it due to its agent-based querying mechanism.

The query execution times for agent-based querying on a distributed graph database are visualized in figure-21. We find that as we keep doubling the number of compute nodes, the execution times tend to increase but converges at 8 nodes configuration. This increase in execution times is likely due to an overhead from communication and synchronization which is required when there are agents migrating and accumulating results across multiple compute nodes.

We can observe that the number of constraints on the query do affect the execution times as the nested query has a slightly slower execution compared to other queries but well within the range of other queries. This proves that the stack-based evaluation of WHERE clause does not add any significant overheads on the MATCH clause while improving the expressiveness of queries in MASS Graph Database.

Takeaways:

- There is scope for improving the agent-based querying with some optimization strategies.
- Benchmarking over larger datasets is required to compare how performance is affected by graph size and node degrees.
- Benchmarking over different types of queries at different depths to see how the execution times are changing.

4. Plan for Summer 25 quarter implementation

The focus during Summer 2025 quarter is performing additional benchmarking tasks over larger datasets and observing how the performance scales with graph size. Another area of focus is to find ways to optimize the filtering performance by improving how agents can terminate quickly using dynamic evaluation while migrating.

Spring 2025 Timeline	Tasks
Week 1	WHERE Clause benchmarks: Running benchmarks over bigger dataset
Week 2	WHERE Clause: Evaluating possible optimizations
Week 3	WHERE Clause: Trying the short-circuit evaluation pattern
Week 4	WHERE Clause: Testing the correctness of above implementation
Week 5	WHERE Clause: Benchmarking across MASS, Neo4j and ArangoDB
Week 6	Delete Clause: Ideation and possible implementation plan
Week 7	Completing the White-paper
Week 8	Documentation and Code clean-up in the development branch
Week 9	Final preparations
Week 10	Thesis Defence?
Week 11	Thesis Defence?

5. References

- [1] L. Cao, "An Incremental Enhancement of the Agent-based Graph Database (Received from DS-Lab)," 2024.
- [2] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.
- [3] C. Willemsen and L. Misquitta, *Neo4j: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2025. ISBN: 9781098165659.
- [4] "Clauses," *Cypher Manual*, Neo4j, Dec. 11, 2024. [Online]. Available: <https://neo4j.com/docs/cypher-manual/current/clauses/>. [Accessed: Dec. 11, 2024].
- [5] M. Fukuda, C. Gordon, U. Mert and M. Sell, "An Agent-Based Computational Framework for Distributed Data Analysis," in *Computer*, vol. 53, no. 3, pp. 16-25, March 2020, doi:10.1109/MC.2019.2932964.

6. Code

1. mass_java_appl, aatmanrp/graph-database branch:
https://bitbucket.org/mass_application_developers/mass_java_appl/src/QueryGraphDB/B/
2. mass_java_core, aatmanrp/mass-refactoring branch:
https://bitbucket.org/mass_library_developers/mass_java_core/src/QueryGraphDB/

7. Appendix

a. Definitions:

1. Anchor Nodes: a set of nodes to start from that are selected first, which are used as the starting point for graph traversals.

b. Use of CypherWhereContext class in parsing the WHERE clause sub-tree

PropertyGraphCypherVisitor – the parsing interface class built using ANTLR tool for parsing the CypherQL queries. It consists of visitor methods for grammar rules based on the following structure in *Figure-15*.

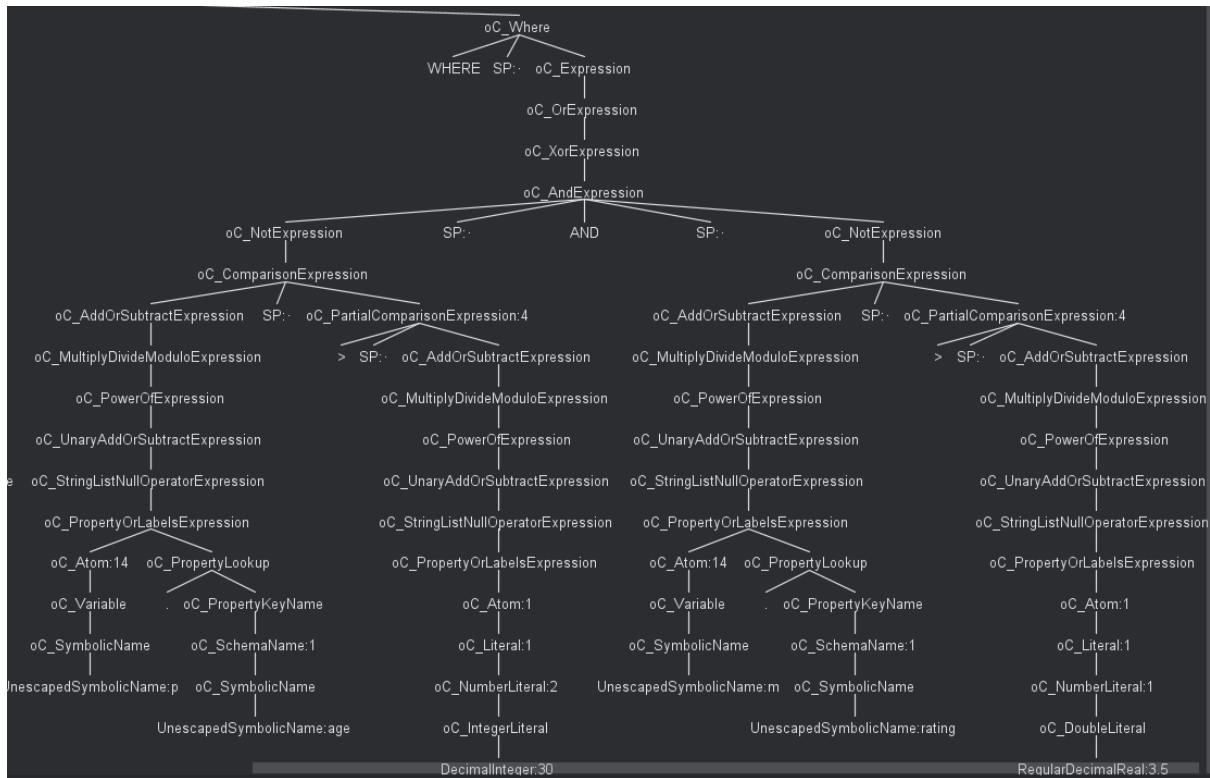


Figure-15: WHERE clause parse tree used in the parsing and AST building phase

The following code snippets represent important visitor methods for rules important to building the AST of WHERE clause (i.e. generating the constraint map and postfix expression), which depict the CypherWhereContext object – “*whereCtx*” being passed down the subtree.

```

@Override
public CypherWhereClause visitOC_Where(CypherParser.OC_WhereContext ctx) {
    if (ctx == null) {
        return null;
    }
    CypherMapLiteral<String, CypherListLiteral<List<String>>> constraints_map = new CypherMapLiteral<>();
    CypherListLiteral<String> expression_list = new CypherListLiteral<>();
    //new obj to pass downstream
    CypherWhereContext whereCtx = new CypherWhereContext(constraints_map, expression_list);
    //returns the final CypherWhereClause obj => return visitOC_Expression(ctx.oC_Expression)
    visitOC_Expression_(ctx.oC_Expression(), whereCtx);
    return new CypherWhereClause(whereCtx.getConstraintsMap(), whereCtx.getExpressionList());
}

```

Figure-15: Visitor method for visitOC_Where rule

```

//WhereClauseParsing
private void visitOC_OrExpression_(CypherParser.OC_OrExpressionContext ctx, CypherWhereContext whereCtx){
    List<CypherParser.OC_XorExpressionContext> children = ctx.oC_XorExpression();
    if (children.size() == 1) {
        visitOC_XorExpression_(children.get(0), whereCtx);
    }else{
        //visit sub-tree / children nodes
        visitOC_XorExpression_(children.get(0), whereCtx);
        visitOC_XorExpression_(children.get(1), whereCtx);
        whereCtx.getExpressionList().addElement(element:" | ");
    }
}

```

Figure-15: Visitor method for visitOC_OrExpression rule

```

//WhereClauseParsing
private void visitOC_ComparisonExpression_(CypherParser.OC_ComparisonExpressionContext ctx, CypherWhereContext whereCtx){
    List<CypherParser.OC_PartialComparisonExpressionContext> partialComparisonExpressions = ctx.oC_PartialComparisonExpression();
    //With Parenthesis Case
    if(partialComparisonExpressions.size() == 0){
        visitOC_AddOrSubtractExpression_(ctx.oC_AddOrSubtractExpression(), whereCtx, EMPTY_STRING);
    }else{
        //splits
        //left-subTree: gets the symbolic name, builds the constraint_map element, returns UID of the current expression
        String curr_UID = visitOC_AddOrSubtractExpression_(ctx.oC_AddOrSubtractExpression(), whereCtx, EMPTY_STRING);
        if(curr_UID.equals(EMPTY_STRING)){
            throw new PropertyGraphCypherException(msg:"Expression returned an empty unique expression ID");
        }
        //right sub-Tree: uses the UID and adds the RHS of the expression to the constraint_map element
        visitOC_PartialComparisonExpression_(partialComparisonExpressions, whereCtx, curr_UID);

        //adds the expression to the expression_list
        whereCtx.getExpressionList().addElement(curr_UID);
    }
}

```

Visitor method for visitOC_ComparisonExpression rule

```

//WhereClauseParsing
private String visitOC_PropertyOrLabelsExpression_(CypherParser.OC_PropertyOrLabelsExpressionContext ctx,
CypherWhereContext whereCtx, String curr_UID){
    CypherParser.OC_AtomContext atom = ctx.oC_Atom();
    List<CypherParser.OC_PropertyLookupContext> propertyLookups = ctx.oC_PropertyLookup();
    CypherParser.OC_NodeLabelsContext nodeLabels = ctx.oC_NodeLabels();

    //case for RHS value fetch or a parentheized sub-expression case
    if((propertyLookups==null || propertyLookups.size() == 0)
    && (nodeLabels == null || nodeLabels.oC_NodeLabel() == null || nodeLabels.oC_NodeLabel().size() == 0)){
        if (ctx.children.size() != 1) {
            throw new PropertyGraphCypherSyntaxErrorException("invalid expression \"" + ctx.getText() + "\"");
        }
        System.out.println("Fetching RHS");
        return visitOC_Atom_(atom, whereCtx, curr_UID);
    }

    // update the symbolicName & gets the generated unique expression ID
    curr_UID = visitOC_Atom_(atom, whereCtx, curr_UID);
    System.out.println("LHS tree returned UID: " + curr_UID);

    if(curr_UID.equals(EMPTY_STRING)){
        throw new PropertyGraphCypherException(msg:"OC_Atom rule returned an empty expression ID");
    }

    //update the propertyName
    visitOC_PropertyLookup_(propertyLookups, whereCtx, curr_UID);
    return curr_UID;
}

```

Figure-15: Visitor method for visitOC_PropertyOrLabelsExpression rule

```

856 private String visitOC_Atom_(CypherParser.OC_AtomContext ctx, CypherWhereContext whereCtx, String curr_UID){
857
858     //when variable context is not null -> new expression & ID being generated
859     if(!ctx.oC_Variable().isEmpty()){
860         String keyName = visitVariableString(ctx.oC_Variable());
861         curr_UID = whereCtx.generateExpressionID(whereCtx.getIDCounter(), whereCtx.getGeneratedIDSet());
862         if(curr_UID!=null){
863             List<String> ListItem = Arrays.asList("PROPERTY_NAME","COMPARATOR","VALUE",curr_UID,"TRUTH_VALUE");
864             if(whereCtx.getConstraintsMap().containsKey(keyName)){
865                 //add to the corresponding list
866                 CypherListLiteral<List<String>> key_constraint_list = whereCtx.getConstraintsMap().get(keyName);
867                 key_constraint_list.addElement(ListItem);
868             }else{
869                 CypherListLiteral<List<String>> tempValueList = new CypherListLiteral<>();
870                 tempValueList.addElement(ListItem);
871                 whereCtx.getConstraintsMap().addElement(keyName, tempValueList);
872             }
873         }
874     }
875     return curr_UID;
876 }else if(ctx.oC_Literal().isEmpty()){ //RHS value fetch and update to the curr_UID
877     if(curr_UID.equals(EMPTY_STRING)){
878         System.out.println(whereCtx.getConstraintsMap().toString());
879         throw new PropertyGraphCypherException(msg:"oC_Atom Rule got an empty UID");
880     }
881     String rhs_value = visitOC_Literal(ctx.oC_Literal()).getValue().toString();
882     CypherMapLiteral<String, CypherListLiteral<List<String>>> constraints_map = whereCtx.getConstraintsMap();
883     for(String key: constraints_map.getKeys()){
884         CypherListLiteral<List<String>> currKeyOuterList = constraints_map.get(key);

```

Figure-15: Visitor method for visitOC_Atom rule