

©Copyright 2025

Aatman Rajeshkumar Prajapati

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

Aatman Rajeshkumar Prajapati

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2025

Committee:

Munehiro Fukuda

Min Chen

Annuska Zolyomi

Program Authorized to Offer Degree:
Computer Science and Software Engineering

University of Washington

Abstract

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

Aatman Rajeshkumar Prajapati

Chair of the Supervisory Committee:

Munehiro Fukuda

School of Science, Technology, Engineering & Mathematics

Graph databases have become essential in domains requiring real-time querying of highly interconnected data. This work is an enhancement of the querying capabilities of a distributed agent-based graph database. Built using the MASS (Multi-Agent Spatial Simulation) Java library, it combines the computing paradigm of Agent-based Modeling (ABM) with rich data modeling of the property graph model. This system leverages autonomous agents for querying a distributed in-memory graph across multiple computing nodes, enabling scalable and parallel graph operations. This work focuses on integrating the Cypher WHERE clause, a critical feature that enables data filtering functionality, into the current system. To implement this functionality, we adopted a modular approach that begins with an abstract syntax tree (AST) for parsing Boolean expressions. On top of this, we developed two evaluation methods designed to handle constraints efficiently. These extensions improve both the flexibility and performance of read operations, while retaining the system's agent-based execution model. As a result, the system's practical scope is broadened, and a basis is established for supporting more complex query patterns and future mutation operations.

TABLE OF CONTENTS

	Page
List of Figures	iii
List of Tables	v
Chapter 1: Introduction	1
1.1 Motivation	1
1.2 Research Objective	2
Chapter 2: Background	3
2.1 MASS Java Library	3
2.2 MASS Graph Database System: Initial Implementation	3
2.3 MASS Graph Database System: Initial Benchmarking	6
Chapter 3: Related Work	7
3.1 Data models for graph databases	7
3.2 Neo4j	8
3.2.1 Architecture	8
3.2.2 Query Language and processing	9
3.3 ArangoDB	10
3.3.1 Architecture	10
3.3.2 Query Language and processing	10
3.4 Query Processing	11
3.4.1 Fundamental Processing Techniques	11
3.4.2 Index-based and Index-free approaches	12
3.4.3 Distributed Query Processing	12
Chapter 4: Implementation	13

4.1	Parsing and Abstract Syntax Tree Generation	14
4.2	Execution Plan Building Phase	16
4.3	Agent-based Execution	17
4.3.1	Evaluation Mechanisms	18
Chapter 5:	Results	23
5.1	Functionality Testing	23
5.1.1	Parsing phase verification	23
5.1.2	Interactive queries verification	24
5.2	Benchmarking and Dataset Overview	25
5.2.1	Dataset-1: IMDB-1K Dataset	26
5.2.2	Dataset-2: BioSNAP-10K Dataset	26
5.2.3	Dataset-3: Shipments-30K Dataset	28
5.3	Benchmarks: MASS Graph Database	29
5.3.1	CPU Scalability	29
5.3.2	Stack-based VS. Dynamic Evaluation performance	31
5.4	Comparative Benchmarks	35
5.4.1	Single Compute Node: MASS vs. Neo4j vs. ArangoDB	35
5.4.2	Multiple Compute Nodes: MASS vs. ArangoDB	40
Chapter 6:	Conclusion	46
6.1	Contributions	46
6.2	Limitations	47
6.3	Future work	47
Chapter 7:	Appendix	52
7.1	Code and Benchmarking queries	52

LIST OF FIGURES

Figure Number	Page
2.1 MASS Graph Database Overview.	4
2.2 Architecture of the MASS Graph Database.	5
4.1 Walking a parse tree (simplified).	15
4.2 Generating an AST for WHERE clause.	16
4.3 Stack-based expression evaluator.	19
4.4 Optimization cases in three-valued systems (highlighted in green)	21
4.5 Tree-based expression evaluator	22
5.1 Verification of constraints-hashmap and postfix expression formation.	24
5.2 Test dataset nodes.	24
5.3 Test dataset relationships.	25
5.4 Verification of interactive queries with WHERE clause.	25
5.5 IMDB Dataset schema.	27
5.6 BioSNAP dataset schema.	27
5.7 Shipments dataset schema.	28
5.9 MASS graph database execution times for Shipments-30K.	29
5.8 MASS graph database execution times for BioSNAP-10K.	30
5.10 MASS graph database execution times for IMDB-1K.	31
5.11 IMDB-1K - Stack vs Dynamic evaluation.	32
5.12 IMDB-1K - Stack vs Dynamic performance ratios	33
5.15 Shipments-30K - Stack vs Dynamic evaluation.	33
5.13 BioSNAP-10K - Stack vs Dynamic evaluation.	34
5.16 Shipments-30K - Stack vs Dynamic performance ratios.	34
5.14 BioSNAP-10K - Stack vs Dynamic performance ratios.	35
5.17 IMDB-1K - Single compute node performance	36
5.18 IMDB-1K - Heatmap - single compute node execution performance	37
5.19 BioSNAP-10K - Single compute node performance	37

5.20	BioSNAP-10K -Heatmap - single compute node execution performance . . .	38
5.21	Shipments-30K - Single compute node execution performance	39
5.22	Shipments-30K - Heatmap - single compute node execution performance . .	40
5.23	IMDB-1K: Multiple compute node performance	41
5.24	IMDB-1K - Heatmap - multiple compute node execution performance	42
5.25	BioSNAP-10K: Multiple compute node performance	43
5.27	Shipments-30K: Multiple compute node execution performance	43
5.26	BioSNAP-10K - Heatmap - multiple compute node execution performance .	44
5.28	Shipments-30K - Heatmap - multiple compute node execution performance .	45

LIST OF TABLES

Table Number	Page
5.1 Dataset parameters	28

ACKNOWLEDGMENTS

I am extremely grateful to my advisor and committee chair, Professor Fukuda, for his guidance, encouragement, and high standards throughout this work. Special thanks to my fellow researchers at the Distributed Systems Lab and my friends for their valuable insights, thoughtful discussions, and brainstorming sessions throughout the course of this work. Finally, to my family - thank you for your love, patience, and for the opportunity to pursue my master's; your constant support made this possible.

Chapter 1

INTRODUCTION

1.1 *Motivation*

Agent-based modeling (ABM) is a computational modeling paradigm that helps to simulate complex systems through the interactions of autonomous entities called agents. Agents operate with their own set of properties, rules, and internal states. These interactions are critical for generating patterns on a system-level which cannot be extrapolated from the individual agent's behavior itself. Fundamentally, the ABM architecture consists of three components - agents, an interaction environment, and a topology defining how these entities are connected. During a simulation, as agents navigate through the system, the internal states maintained keep updating and enable the agents to decide how they traverse and interact with the environment. The natural alignment between ABM and graph data structures has been extensively demonstrated across diverse domains [1].

Graphs provide an intuitive framework with vertices and directed edges (as well as undirected) that can represent agent-agent and agent-environment interactions. Previous research on graph traversal approached using ABM demonstrates improved parallelization and computational efficiency [2]. The effectiveness of ABM on graph structures is further evidenced by applications in interdisciplinary domains such as ecological simulations [3].

In contemporary times, with the adoption of data intensive technologies like Machine learning and other data-driven decision making approaches such as knowledge graphs, the demand for quality rich data has been on the rise. This has led to not only an increase in data volume but also data complexity as data becomes interconnected. Relational databases experience significant performance degradation dealing with highly interconnected data, as costly JOIN operations lead to exponential runtime complexity. In recent years, graph

databases have emerged as the predominant solution for managing complex, interconnected data due to the efficiency of data retrieval due to their underlying graph-based data model. Key application domains include fraud detection and analytics, real-time recommendation engines, knowledge graphs, and social network analytics. [4]

Industry standard graph databases, such as Neo4j, face scalability and performance issues in scenarios where data volumes and complexity grows as they maintain all the data essentially on a single compute node. Motivated by these developments, researchers at the Distributed Systems Lab (University of Washington - Bothell) have developed a graph database using the MASS Java library [5]. This development addresses the gap of data centrality in modern graph databases by building a distributed graph database along with a novel querying approach using agents.

1.2 Research Objective

This project builds on the above mentioned implementation of an agent-based graph database, using the MASS Java library, extending the system’s query capabilities by integrating support for the Cypher WHERE clause, a critical feature for filtering and refining data retrieval. The objective of this research is as follows:

1. Enabling the support for constraint-based filtering (using expressions and logical operations over sub-expressions) in MASS Graph Database, thereby extending the set of read queries overall.
2. Exploring different evaluation mechanisms to better align with agent-based querying and performing a comparative analysis on how it affects query execution performance.
3. Benchmarking the performance of MASS Graph Database compared to Neo4j and ArangoDB for data-retrieval using filtering queries.

Chapter 2

BACKGROUND

2.1 *MASS Java Library*

The **Multi-Agent Spatial Simulation** (MASS) Java library is a parallel computing library that allows simulations involving multi-agents, with a spatial environment, over a distributed computing cluster. Its core architecture centers around two fundamental abstractions:

- **Places:** Places are multi-dimensional distributed arrays, hosted on specific compute nodes of a cluster, that help define the spatial environments for multi-agent simulations.
- **Agents:** Agents are autonomous entities that interact and occupy Places, are capable of performing computations using the locally stored data and migrate across the cluster during the course of the simulation.

MASS performs computation through coordinated cycles: the `callAll` phase initiates agents to perform required computations and determine migration destination; the `manageAll` phase handles all lifecycle (spawning, migration and termination of agents) and synchronization events across the cluster. MASS uses multi-threading to achieve parallel execution within each node and TCP-based messaging for inter-node communication, thereby supporting horizontal scalability for simulations.

2.2 *MASS Graph Database System: Initial Implementation*

The initial implementation significantly extended MASS Java library’s capabilities and built the foundation for an interactive graph database system, as shown in Figure 2.1 . It involved extending the MASS Java library’s abstractions to support the property graph model and querying support with CypherQL [5]. This transformation introduced several key features:

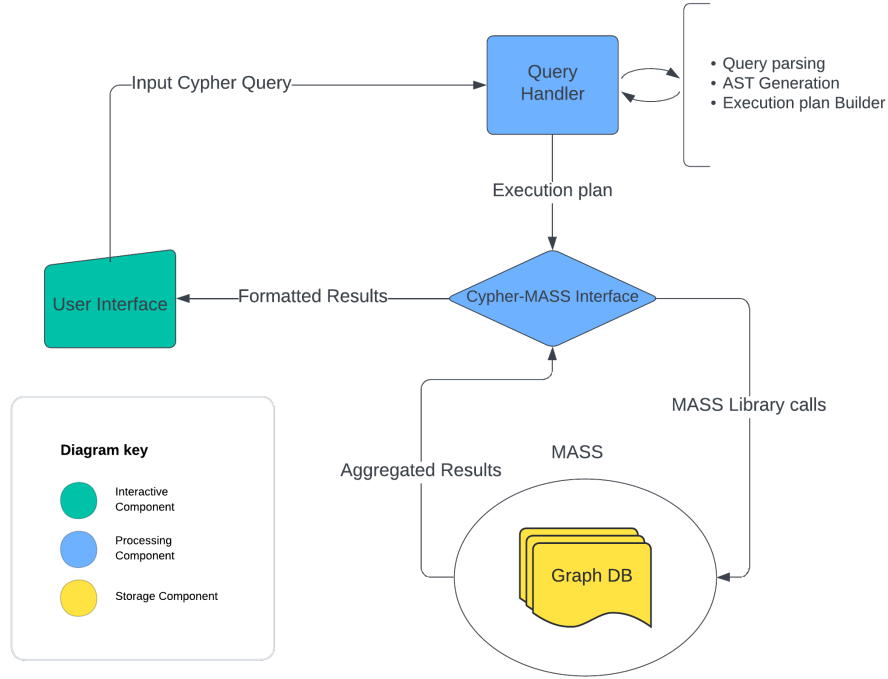


Figure 2.1: MASS Graph Database Overview.

- Property Graph Model:** MASS Graph Database adopted this data model for storing the in-memory graph, enabling graph vertices and edges to include labels and key-value based attribute pairs. MASS java library supports this by extending the previously implemented `Places` and `VertexPlace` classes with `PropertyGraphPlaces` (a collection of `PropertyVertexPlace` instances) and `PropertyVertexPlace` (stores an individual graph vertex coupled with its edges). These classes facilitate agents to perform data validations during traversals.
- CypherQL Integration:** To support querying to MASS Graph Database using CypherQL, a customized parser was built using the ANTLR-v4 parsing tool. It performs the conversion of CypherQL graph queries into an abstract syntax tree representation (AST) and execution plans, enabling sophisticated graph pattern querying.

- **Agent-Based Query Execution:** In order to model the graph traversal patterns with MASS agents, the `PropertyGraphAgent` class was designed. Queries are executed as `PropertyGraphAgents` traverse the distributed graph, performing parallel computations and pattern matching.
- **In-Memory Distributed Storage:** MASS maintains the entire graph in memory, which is logically distributed throughout the cluster (depending on the number of compute nodes used). This development was a part of improving the storage of graph data within distributed memory by extending the implementation of `GraphPlaces` class, allowing rapid data access compared to traditional disk-based solutions [5].

An architectural representation of these changes is shown in Figure 2.2.

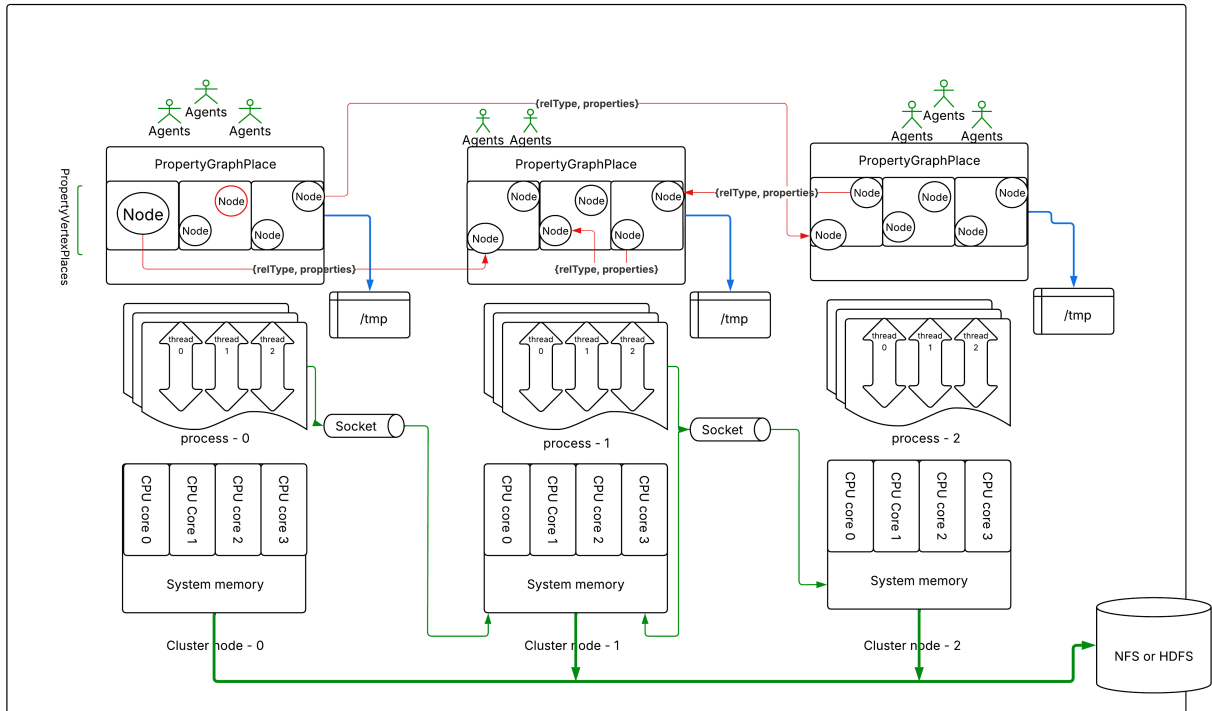


Figure 2.2: Architecture of the MASS Graph Database.

2.3 MASS Graph Database System: Initial Benchmarking

A standardized benchmarking was performed to evaluate MASS Graph Database’s performance relative to industry standard graph databases - ArangoDB [6] and Neo4j [7].

The key highlights of the benchmarking approach were as follows:

- **Query Patterns:** Basic pattern-based queries, involving graph traversals at varying depths (1,2 and 3), were used during this benchmarking. These queries are a foundational test for graph database performance and were implemented CypherQL(for Neo4j and MASS Graph Database) and AQL(for ArangoDB).
- **Datasets:** Random homogeneous graphs were generated with a controlled node degree (typically 2-3) based on real-world datasets from social networks, product co-purchase graphs, or transaction/fraud detection scenarios. Evaluations used had graphs sizes up to 30,000 vertices, as larger sizes led to memory issues.
- **Results:** Results suggest that for larger graphs, MASS could outperform ArangoDB for depth-3 traversals and was competitive with Neo4j under some workloads, although traversal times for small depths and in the single-node configuration were not competitive compared to Neo4j and ArangoDB [5].

The above benchmarking concluded that agent distribution and in-memory data structures led to specific trade-offs in traversals using the agent-based querying model. The communication overheads, due to synchronization, and memory capacity are things to be considered for future development and design decisions.

Chapter 3

RELATED WORK

In recent years, graph databases have been utilized for real-time analytics in domains where data is found to be highly interconnected, such as - fraud detection, drug discovery, and social networks [8]. The choice of data model used, storage architecture, and query processing strategy has a direct impact on scalability and performance, particularly in the case of queries involving deep traversals and complex pattern matching requirements. This section reviews data models used in graph databases, architectural implications with case studies of Neo4j and ArangoDB, and explores the query processing techniques and their implications for distributed environments.

3.1 Data models for graph databases

Modern graph database systems are primarily based on two data models (sometimes subtle variations of them), which dominate the field : the **Property graph model** (also known as labeled property graph) and the **RDF (Resource Description Framework)** model. Each model has distinct properties and approaches towards representing the graph information.

The Property graph model has become the predominant model for most modern graph databases due to its ability to represent real-world data inherently. It allows for rich data modeling due to four core properties - representing individual entities using nodes, relationships between nodes as directed edges, allowing any number of key-value based attributes (known as properties) for nodes and relationships alike, and categorizing nodes and relationships using labels. Graph databases like Neo4j and ArangoDB utilize slight variants of the property graph model to represent data [9]. In addition to the above-mentioned core proper-

ties, Neo4j allows for any number of labels associated to each vertex (node) but allows only one label per edge (relationship). ArangoDB, in order to maintain its underlying document structure effectively, allows only one label per vertex and edge.

The Resource Description Framework (RDF) is a different approach to graph data modeling and places a strong emphasis on interoperability. Graphs in RDF are usually a collection of triplets. Triplets in RDF follow a subject-predicate-object structure, where each component is usually either a URI(Uniform Resource Identifiers) or a literal. The model is designed with the intention of providing a graph-based description for heterogeneous data. The key advantage of this model is the ability to generate knowledge graphs that can integrate information from multiple sources while maintaining a semantic meaning [9].

Due to the fundamental difference in the above mentioned data models, each has its own set of advantages in different use cases. While RDF offers the potential for dealing with heterogeneous data sources, property graph model suits the case of agent-based querying better. This is because it allows the for a schema-driven and rich data model, where the nodes and relationships are first-class entities, allowing for an agent’s state changes and traversals to occur in a seamless manner. Hence, it was chosen as the data model for MASS graph database system.

As the graph database systems - Neo4j and ArangoDB utilize the property graph data model, an architectural overview is required to better understand the implications.

3.2 Neo4j

3.2.1 Architecture

Neo4j is one of the most widely used graph databases in the industry and it popularized the property graph model as the standard for graph databases. Additionally, Neo4j maintains both relationships and nodes as first-class entities and places a restriction of directional relationships between any two nodes. This architectural choice is made to implement ”native

graph processing” which enables traversals without the use of additional index lookup (also known as index-free adjacency). This means that each node contains direct pointers to its adjacent relationships (stored as a linked-list), eliminating the need for index lookup during traversal operations. This ensures that the traversal performance in Neo4j is solely dependent on local connectivity of nodes rather than the overall size of the database [10]. A major downside of this approach is that it does not scale horizontally if the graph is partitioned over a cluster of computing nodes.

3.2.2 Query Language and processing

Querying in Neo4j is done using the Cypher Query Language (commonly known as CypherQL). CypherQL’s development was led by Neo4j as an open-source language and has seen adoption by existing databases due to its simplicity. CypherQL is a declarative graph query language that uses pattern-matching syntax to express graph patterns, allowing users to specify nodes, relationships, and properties in an intuitive, ASCII-art-like format. This query language is inspired by the popular database query language SQL, and some expression matching is inspired by SPARQL. The intuitive querying format for graph traversal along with open-source support makes it an ideal choice for MASS graph database [11].

On the surface, Neo4j’s query processing follows the standard procedure of parsing, semantic analysis, query planning and execution. Most of the optimization occurs during the planning phase, where the Cypher planner generates a logical plan in the form of a tree-like structure of operators. The execution proceeds in a bottom-up manner from the leaf-node operators and transformations happen as the data flows upstream [10]. The optimizations are based on graph statistics and metadata, including index usage patterns and relationship density information.

Neo4j uses different types of global indexing techniques for different use cases. It uses b-trees for indexing nodes for sequential access, token-lookup indexes for labels based queries which help with efficient traversal starting points. Neo4j’s performance advantages are particularly evident in deep traversal queries and pattern matching operations. Recent studies

support this claim for querying over connected data as Neo4j consistently outperforms traditional RDBMS systems like MySQL [12]. This architecture helps with deep traversal queries without the exponential performance degradation of join operations.

3.3 ArangoDB

3.3.1 Architecture

ArangoDB is a fundamentally different graph database in terms of the design architecture as it supports multi-modal capabilities along with graph traversals. It adopts this multi-modal approach by storing data as JSON documents in a unified storage layer. In addition to graph, it supports documents and key-value access patterns. This allows users to utilize the same data store by simply modifying the queries (based on the data access strategy) without data restructuring or switching to a different database system altogether.

ArangoDB’s graph implementation follows the property graph model with a document-oriented approach. It stores vertices of the same type (i.e. label) in document collections without maintaining the information of its edges. It provides an advantage with edge creation and deletion being isolated from the associated vertex, allowing such changes without modifying the vertex document. Edges are stored using special edge collections with `_from` and `_to` attributes, referring to the vertex document IDs. The presence of edge collections enables ArangoDB to prevent frequent vertex document reads leading to faster query execution [9].

3.3.2 Query Language and processing

ArangoDB Query Language (AQL) is the query language used for multi-modal querying. AQL is primarily declarative, but it does have aspects which enable the users to define the execution flow. AQL’s multi-model nature allows for complex queries that can combine graph traversals with document searches, aggregations in a single statement. Given the multi-modal nature of ArangoDB, the query processing techniques differ significantly from

other graph databases. The query engine supports both declarative and imperative type of queries, typically with an extensive use of the FOR loop construct. Users can define the ways in which data gets accessed, filtered, and the stage for data transformations and aggregations. The optimizations mostly happen around leveraging the indexing mechanisms to push operations to appropriate storage engines. ArangoDB demonstrates its strengths in horizontal scaling in multi-modal scenarios where the data access patterns can vary based on different requirements [6].

3.4 Query Processing

3.4.1 Fundamental Processing Techniques

Graph databases and relational databases require fundamentally different query processing methodologies due to the difference in data access patterns (based on data models) and a traversal-based approach for navigating the database. The challenge lies in maintaining the performance using different traversal techniques as the graph sizes (both the number of vertices and edges) increase. Query processing typically involves four main stages: parsing, translation to the required representation (varies based on technologies used), optimization, and evaluation.

Query processing in graph databases is fundamentally a graph traversal with additional complexity. Traversal algorithms such as breadth-first search (BFS) and depth-first search (DFS) serve as the primary building blocks. BFS is best for shorter path traversal queries as it explores all neighboring vertices before moving onto the next hop/level of traversal. DFS explores the paths as far as possible before backtracking and is suitable for deep-traversal queries with additional complexity. Approaches like aDFS explore how hybrid execution using both, DFS and BFS, can provide performance improvements for certain types of queries in a distributed workload using an imperative language like PGQL [13].

Modern graph databases go beyond only optimizing for traversal approaches. Graph-specific optimizations include query rewriting to transform queries into more efficient equiv-

alent forms, index utilization to reduce traversal starting points, and cost-based optimization that considers graph topology and connectivity patterns. A study on query optimizations found that topology-based optimizations are good for smaller datasets and approaches with better pruning mechanisms lead to better overall execution performance [14].

3.4.2 Index-based and Index-free approaches

Index-free adjacency has been a significant point of discussion in graph database architecture, with Neo4j pioneering this approach. This approach essentially structures nodes with direct references to their adjacent relationships without requiring frequent index-lookup, theoretically decoupling traversal times from the graph size [10]. This approach contrasts with index-based systems that require intermediate lookups to navigate between connected nodes, utilized extensively by databases like ArangoDB (and Neo4j for specific types of data). The effectiveness of either approach depends on particular use cases, graph topology, and query patterns.

3.4.3 Distributed Query Processing

Distributed query processing becomes an important factor when considering systems that can scale horizontally, as datasets become larger. Existing literature has approaches that perform distributed graph processing by offloading parts of processing to relational data stores (DG-SPARQL [15], but no such approach exists for native in-memory distributed graphs. Specifically for graph databases, there are unique challenges pertaining effective graph partitioning, maintaining subgraphs in the same cluster for data locality, and coordinating across multiple compute nodes.

The challenge for distributed graph processing lies in minimizing network communication while maintaining query performance. Effective partitioning strategies must consider the graph structure to minimize cross-partition edges, while query optimization must account for the increased latency of distributed operation. Additionally, for the MASS graph database, we must consider the approaches that suit agent-based querying.

Chapter 4

IMPLEMENTATION

This section looks at the implementation of the Cypher WHERE clause in the MASS graph database. It covers the scope of the implementation, the entire query processing pipeline (from parsing to execution) in detail, along with the two evaluation mechanisms explored.

The WHERE clause in Cypher is a mechanism to filter the results obtained from the Cypher MATCH clause. It allows users to provide a set of constraint expressions on graph elements such as nodes (vertices) and relationships (edges) based on their properties. The role and integration of the WHERE Clause in MASS Graph Database is as follows:

- **Functionality:** WHERE clause operates as a filter by providing additional constraints on the read queries with expressions. A simple constraint, for example, `MATCH (n: Person) WHERE n.age > 30`, finds only those with an age property greater than 30.
- **Types of Queries:** WHERE clause supports a wide variety of conditions, including property comparisons (string, numerical, etc.) and complex logical combinations over constraint expressions using `AND`, `OR`, `NOT`, and `XOR`.

Query processing in the MASS graph database is a three-step process. It starts with parsing the CypherQL query (raw string input) and building a logical hierarchy/representation known as the abstract syntax tree (AST). Next, we have the execution planning phase, characterized by producing a step-by-step plan to execute the particular query. Additionally, it also performs any query pre-processing or data transformations necessary for the final

step. Finally, the plan is sent to the execution engine, which fetches the required results. The finer details of each of the steps mentioned above are as follows:

4.1 Parsing and Abstract Syntax Tree Generation

In the parsing phase, the input string query is first processed to generate an Abstract Syntax Tree (AST), representing the logical structure of the WHERE clause. Parsing is conducted using ANTLR (ANother Tool for Language Recognition) [16], a robust parser generator, tailored specifically to the grammar rules of Cypher grammar. This phase involves:

1. **Parse tree generation:** The raw input string gets fed to the lexer generated by ANTLR for the Cypher grammar file. This essentially tokenizes the input character stream into a sequence of tokens which form the foundational blocks of a language (i.e. keywords, identifiers, literals etc.). Next is syntactic analysis (parsing), during which the rules of the language are applied to build the structure of the input known as the parse tree.
2. **Walking the parse tree:** This is the crucial step during the parsing phase, as it helps us essentially "walk" through the generated parse tree using ANTLR's visitor interface. This interface consists of customized methods that get invoked based on the rule being visited on the parse tree. The `PropertyGraphCypherVisitor` class consists of all the customized methods that enable us to perform the top-down recursive parsing. For each rule present, we move up and down the parse tree and store what needs to be extracted from that particular rule into our abstract syntax tree, as visualized in Figure 4.1.
3. **Abstract Syntax Tree:** The AST module is a package of Java classes, built to store the data extracted from parsing hierarchically and logically. Since the WHERE clause essentially provides us with a set of constraints, a logical representation was required to

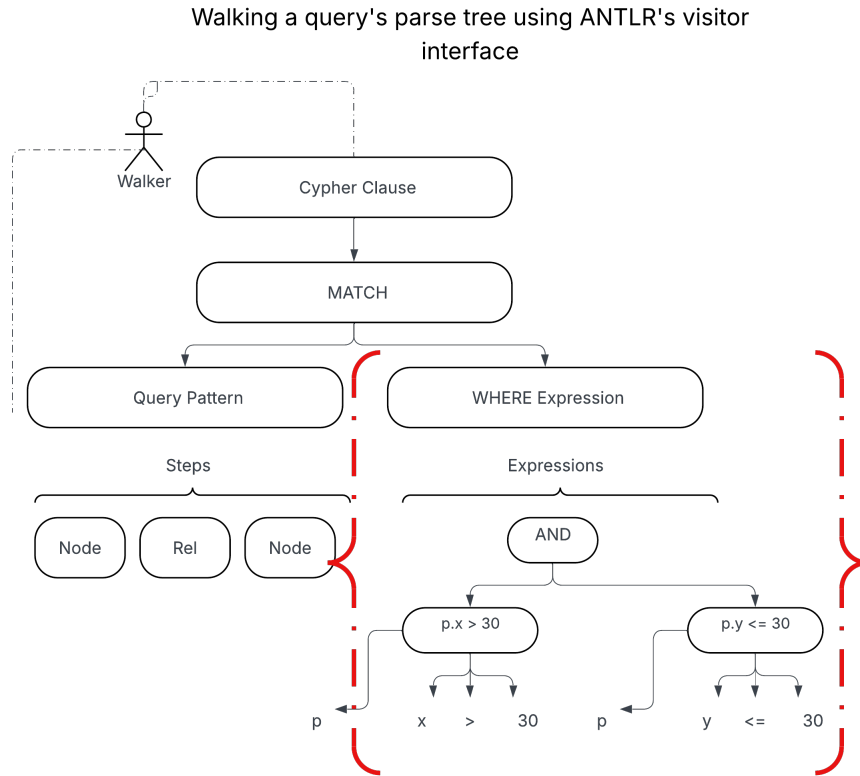


Figure 4.1: Walking a parse tree (simplified).

maintain the context of the constraints, which is fulfilled by the `CypherWhereClause` class in the AST module.

4. **CypherWhereClause Structure:** The `CypherWhereClause` class was built based on the utilities needed for the evaluation mechanisms - a hashmap of constraints and a postfix notation defining how the constraints (i.e., sub-expressions) are structured logically. During the parsing phase, each constraint is added to the hashmap along with a unique identifier (called `ExpressionID`) value. The mapping structure of the constraints hashmap is as follows:

- **Key:** Symbolic names for entities encountered in the input query.
- **Property Value:** a list of lists consisting - {property name, comparator, left-hand-side (LHS) value, TruthValue, ExpressionID}

The outcome of this parsing phase is the generation of a **CypherWhereClause** object, which stores the constraints and logical relationships defined in the WHERE clause. A visualization is presented in Figure 4.2, using an example query.

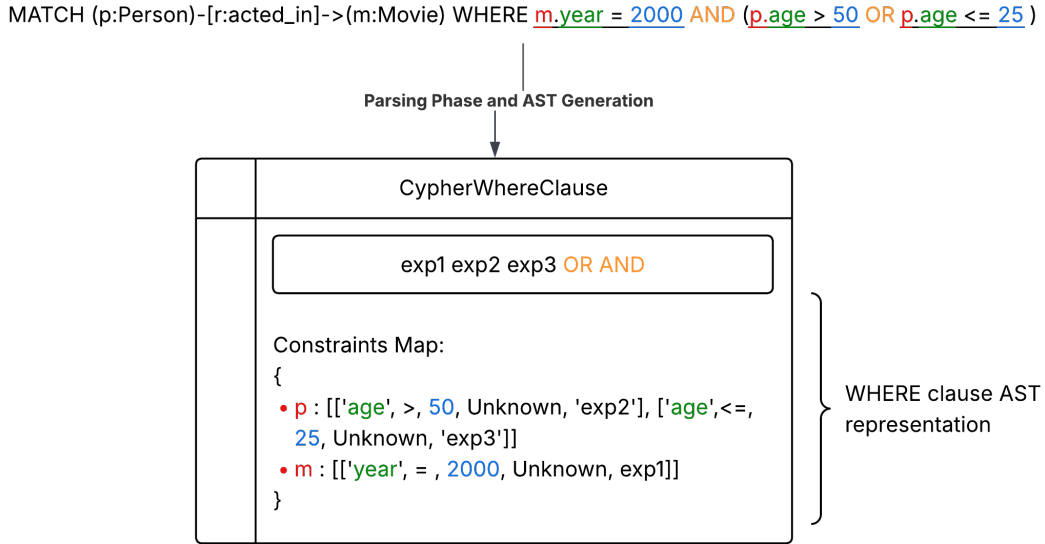


Figure 4.2: Generating an AST for WHERE clause.

4.2 Execution Plan Building Phase

Following AST generation, we proceed to the execution planning phase, wherein the structured AST is leveraged to prepare for distributed execution across the MASS agent-based graph database. This phase includes the following:

1. **Determination of Execution Steps:** In this phase, the assembled AST and corresponding parse tree are used to plan the order and structure of query execution.

2. **Argument Packaging for Agents:** The execution plan prepares the arguments that will be supplied to `MASS PropertyGraphAgents`. These arguments include the hashmap of constraints and any supporting variables necessary to reproduce the logical structure of the original WHERE clause. The plan segments the query into agent-executable "chunks" determined by the number of graph nodes/vertices in the MATCH clause pattern (i.e., the number of hops or depth of traversal).

These packaged arguments are critical for agent-based execution, clearly breaking down the computational tasks for each iteration and enabling distributed querying using MASS agents.

4.3 *Agent-based Execution*

In the agent-based execution phase, `MASS PropertyGraphAgents` are instantiated with pre-processed arguments to traverse the distributed in-memory graph representation. The agents traverse by iteratively performing two critical functions:

- `callAll()`: This is responsible for validating if the agents are at the correct vertex (and the corresponding edge based on the input query), along with updating the constraints hashmap's truth-values. If validated, it determines the number of child-agents to spawn based on the number of edges. This step also flags agents for termination if the path is deemed incorrect.
- `manageAll()`: Oversees the lifecycle management of agents, with three events - Spawn (create new child-agents), Migrate (transfers an agent to its next step according to its vertex location in the cluster), and Kill (terminates the execution of agents flagged to terminate during `callAll()`).

The MASS graph database maintains a distributed, shared map of vertices (`PropertyVertexPlace`), which agents utilize to validate property values and constraint conditions, regardless of the

physical node location in the distributed cluster.

4.3.1 *Evaluation Mechanisms*

During the execution phase, constraints represented in the `CypherWhereClause` can be evaluated using different approaches. This project explored two distinct approaches: **Stack-based Evaluation** and **Dynamic (tree-based) Evaluation**.

1. **Stack-based Evaluation:**

- **Requirements:** This method of evaluation requires all truth-values of the constraint expressions to be updated in the constraints map prior to evaluation.
- **Methodology:** The stack-based evaluator uses two parameters to evaluate the final expression value - a hashmap of truth-values and a postfix notation of expressions.
 - The truth-values hashmap uses the `ExpressionID` as key, mapped to a boolean value (true or false), derived from our constraints hashmap.
 - It goes through the postfix notation left to right, pushes the `ExpressionID` into the stack and evaluates the topmost values (one or two based on the type of logical operator) and pushes the new value back onto the stack.
 - This process is repeated till it reaches the end of the postfix notation and the topmost value in the stack is the final output. A visualization of this approach is shown using a simple example expression [`exp1 AND (exp2 OR exp3)`] in Figure 4.3.
- **Implications:** With this evaluation method, the results of the query are filtered after all possible paths have already been traversed. It filters the results but does not eliminate redundant computations. The expected runtime with this approach is similar to or worse than the case without using the `WHERE` clause.

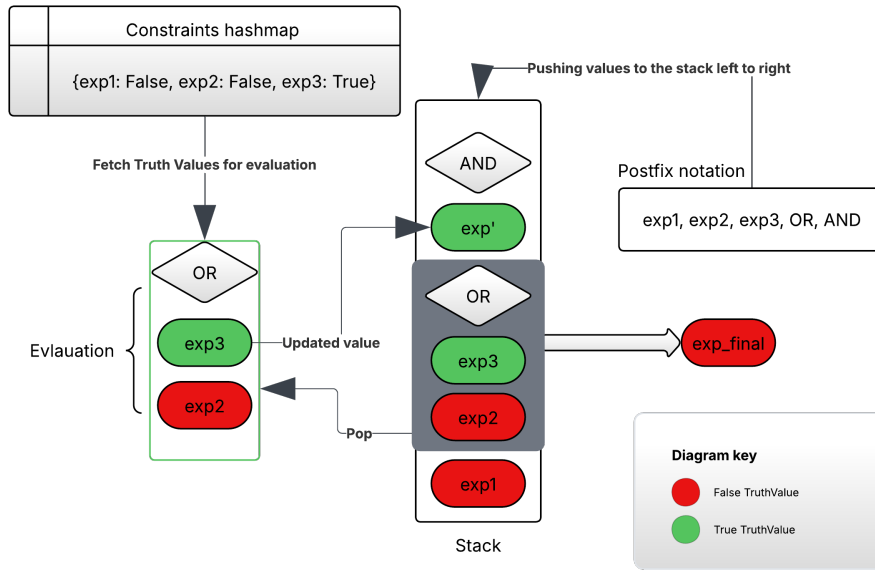


Figure 4.3: Stack-based expression evaluator.

2. **Dynamic Evaluation (Using Trees and three-valued logic):** This approach operates in a different logic system than the traditional Boolean logic. It uses the three-valued logic system [17], where constraint expressions can have one of three possible states at a particular moment: true, false, or unknown. This logic brings certain advantages by providing a value for the final expression at every step of the traversal. If the final value is true or false, we do not need to perform further computations, thereby reducing redundant computations.

- **Requirements:** This method of evaluation requires two parameters to evaluate the final expression value - a hashmap of truth-value and a tree-based notation of the constraints expression.
- **Methodology:** This evaluation requires a tree data structure, which stores operators (unary and binary) as non-leaf nodes and ExpressionIDs as leaf-nodes of a tree. The evaluation occurs by traversing the tree in an iterative post-order traversal.

sal, evaluating the expression at each step, and the resultant value gets pushed back to help the evaluation upstream. A visualization of the truth values for optimized cases is found in Figure 4.4. The Figure 4.5 shows an example where we can determine the final value of an example expression: [`exp1 AND (exp2 OR exp3)`] where some truth-values are unknown at the time of evaluation. Finally, when the root node gets evaluated, we have the final answer, which determines what the agent will do :

- Terminate itself: if the final value is false (F), the current agent gets terminated and the path is not explored further.
 - Defer Evaluation: if the final value is unknown (UNK), the evaluation gets deferred to the next step of the traversal.
 - No further computations: if the final value is true (T), the current agent is deemed to be on the correct path and no evaluation for constraints is required.
- Implications: With this evaluation method, the results get computed during the traversal itself. We probabilistically determine the outcome for around 60% of the cases (based on Figure 4.4, reducing redundant computations. Furthermore, we can dynamically prune the incorrect paths, which reduces the number of active agents and leads to less computational overhead.

Diagram key

- T = True
- F = False
- UNK = Unknown state
- Case being acted upon

x	y	x OR y	x AND y
T	T	T	T
T	F	T	F
T	UNK	T	UNK
F	T	T	F
F	F	F	F
F	UNK	UNK	F
UNK	T	T	UNK
UNK	F	UNK	F
UNK	UNK	UNK	UNK

Figure 4.4: Optimization cases in three-valued systems (highlighted in green)

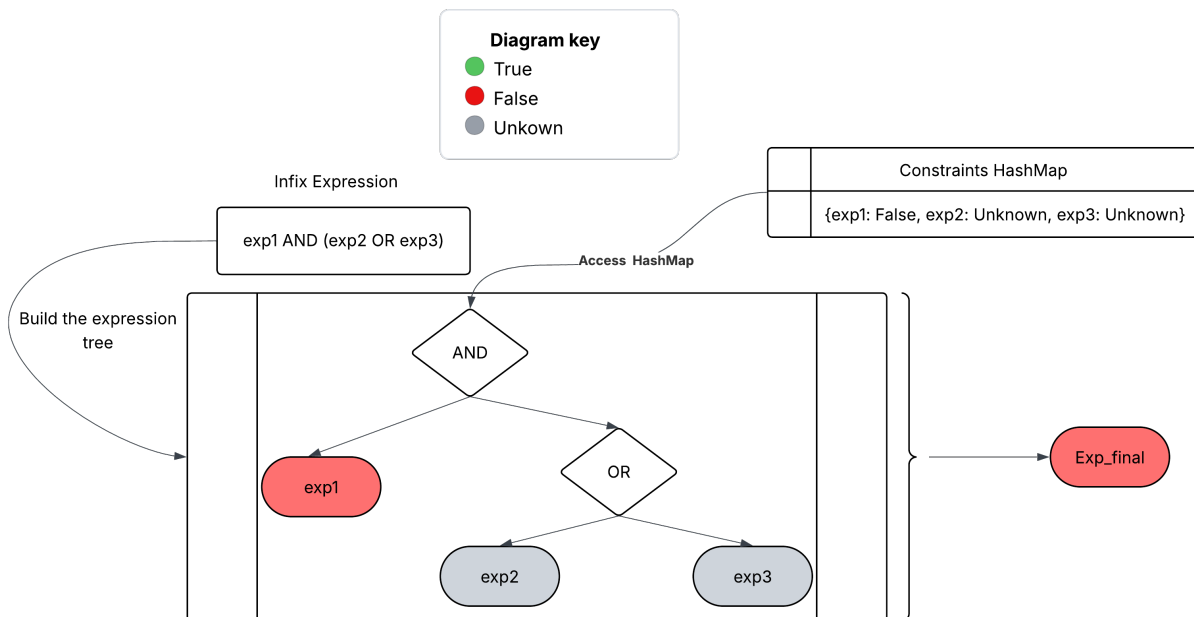


Figure 4.5: Tree-based expression evaluator

Chapter 5

RESULTS

This section begins with the functionality testing of the WHERE clause implementation, followed by the results for MASS Graph database benchmarking performed using three datasets, an overview of which is presented below.

5.1 *Functionality Testing*

5.1.1 *Parsing phase verification*

This involves testing the parser with various queries of different complexity, the goal of which is to check the formation of the constraints hashmap and the postfix notation of constraint expressions with expressionIDs (generated during the parsing phase). The following cases cover all possible scenarios and the resultant constraints map formation is shown in Figure 5.1:

- WHERE p.age > 30
- WHERE p.age > 30 AND m.title = 'xyz'
- WHERE p.age > 30 OR (m.budget >= 100000 AND m.year < 2000)
- WHERE p.age > 30 OR m.budget >= 100000 OR m.year < 2000
- WHERE NOT(p.age > 30 OR m.budget >= 100000) XOR m.year < 2000

```
[aatmanrp@cssmp12 mass_code]$ /usr/bin/env /usr/lib/jvm/java-11-openjdk-11.0.25.0.9-3.el9.x86_64/bin/java @/tmp/cp_1eyn12zznlt7dmg2pjnh025y.argfi
.uw.bothell.css.dsl.MASS.graph.db.WhereRuleParsing
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 30, exp0, TRUTH_VALUE]}
expression_list: exp0
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 30, exp0, TRUTH_VALUE], m: [title, =, xyz, exp1, TRUTH_VALUE]}
expression_list: exp0, exp1, &
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 30, exp0, TRUTH_VALUE], m: [budget, >=, 100000, exp1, TRUTH_VALUE], [year, <, 2000, exp2, TRUTH_VALUE]}
expression_list: exp0, exp1, exp2, & , |
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 30, exp0, TRUTH_VALUE], m: [budget, >=, 100000, exp1, TRUTH_VALUE], [year, <, 2000, exp2, TRUTH_VALUE]}
expression_list: exp0, exp1, | , exp2, |
====>>>CypherWhereClause content:
WHERE
constraints-map: {p: [age, >, 30, exp0, TRUTH_VALUE], m: [budget, >=, 100000, exp1, TRUTH_VALUE], [year, <, 2000, exp2, TRUTH_VALUE]}
expression_list: exp0, exp1, | , ! , exp2, ^
[aatmanrp@cssmp12 mass_code]$
```

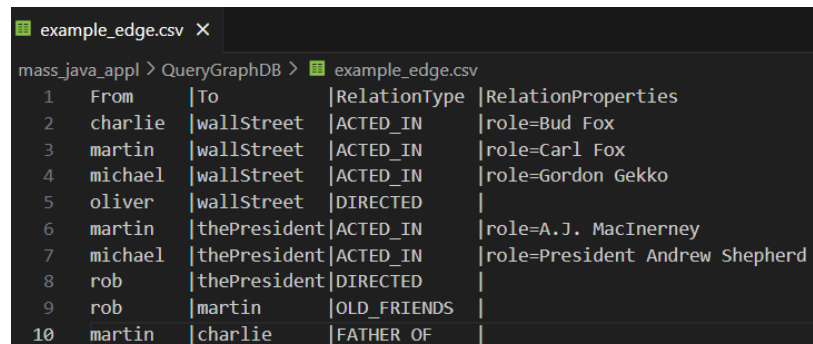
Figure 5.1: Verification of constraints-hashmap and postfix expression formation.

5.1.2 Interactive queries verification

Using a small test dataset, as in Figure 5.2 and Figure 5.3, we can see that the WHERE clause is successfully integrated in the MASS graph database. It provides the correct results as expected (shown in Figure 5.4), thereby improving the expressiveness of queries to the MASS graph database.

example_object2.csv		
mass_java_appl > QueryGraphDB > example_object2.csv		
1	itemUniqueName	Labels Properties
2	charlie	Person name=Charlie Sheen, age=35
3	martin	Person name=Martin Sheen, age=12
4	michael	Person name=Michael Douglas, age=40
5	oliver	Person name=Oliver Stone, age=50
6	rob	Person name=Rob Reiner, age=21
7	wallStreet	Movie title=Wall Street
8	thePresident	Movie title=The American President
9		

Figure 5.2: Test dataset nodes.



From	To	RelationType	RelationProperties
charlie	wallStreet	ACTED_IN	role=Bud Fox
martin	wallStreet	ACTED_IN	role=Carl Fox
michael	wallStreet	ACTED_IN	role=Gordon Gekko
oliver	wallStreet	DIRECTED	
martin	thePresident	ACTED_IN	role=A.J. MacInerney
michael	thePresident	ACTED_IN	role=President Andrew Shepherd
rob	thePresident	DIRECTED	
rob	martin	OLD_FRIENDS	
martin	charlie	FATHER_OF	

Figure 5.3: Test dataset relationships.

```

FROM neighbors:
  ID: michael, types: [acted_in], propertiess: {role=president andrew shepherd}
  ID: rob, types: [directed], propertiess: {}
  ID: martin, types: [acted_in], propertiess: {role=a.j. macinerney}

Enter queries: MATCH (p:person) WHERE p.age > 30 RETURN p
Printing Query Results:
p = charlie
p = michael
p = oliver
Query execution time: 196
Enter queries: MATCH (p:person)-[r:acted_in]->(m:movie) WHERE p.age <= 40 AND m.title = 'wall street' RETURN p,m
Printing Query Results:
p = charlie    m = wallstreet
p = martin    m = wallstreet
p = michael    m = wallstreet
Query execution time: 54
Enter queries: MATCH (p:person) WHERE p.age > 21 AND p.age <= 40 RETURN p
Printing Query Results:
p = charlie
p = michael
Query execution time: 18
Enter queries: MATCH (p)-[r:acted_in]->(m) WHERE r.role = 'bud fox' RETURN p,m
Printing Query Results:
p = charlie    m = wallstreet
Query execution time: 22

```

Figure 5.4: Verification of interactive queries with WHERE clause.

5.2 Benchmarking and Dataset Overview

The purpose of the benchmarking is to provide a baseline execution performance of MASS Graph Database under varying datasets (each with different topology, size, and other graph characteristics). In order to standardize the benchmarking, the three sets of queries are compared at varying levels of traversal depths, according to each of the datasets.

These consist of the following types (along with an example):

1. No constraints: Only consists of the MATCH clause with query pattern.
E.g. `MATCH (s:shipment)-[r:assigned_to]->(c:carrier) RETURN s`
2. Single constraint: MATCH clause pattern with only 1 constraint expression.
E.g. `MATCH (s:shipment)-[r:assigned_to]->(c:carrier) WHERE r.pathindex = 9 RETURN s`
3. Multiple constraints: MATCH clause pattern with multiple constraint expressions, along with logical operators. E.g. `MATCH (s:shipment)-[r:assigned_to]->(c:carrier) WHERE r.pathindex = 9 AND (r.effectiveminutes > 30 AND r.effectiveminutes <= 55) RETURN s`

The actual queries used for every dataset can be found in the appendix section 7.1.

5.2.1 Dataset-1: IMDB-1K Dataset

This is a widely used dataset by Neo4j and is also present as a starter directly in their certification modules [18]. This dataset represents a simplified version of data extracted from the popular website IMDB, and consists of three types of nodes - **Person (sub-categories: Actor and Director)**, **Movie** and **User**; and three types of relationships - **ACTED_IN**, **DIRECTED**, **RATED**. Dataset parameters are listed in Table 5.1 and the schema is shown in Figure 5.5.

5.2.2 Dataset-2: BioSNAP-10K Dataset

This is a drug side-effect association network that contains information on side effects caused by drugs that are on the U.S. market. Nodes represent **Compounds (drugs) and side effects**, and edges indicate recorded adverse drug reactions. This data is derived from the Stanford Biomedical Network Dataset Collection (BioSNAP) [19]. Furthermore, this dataset

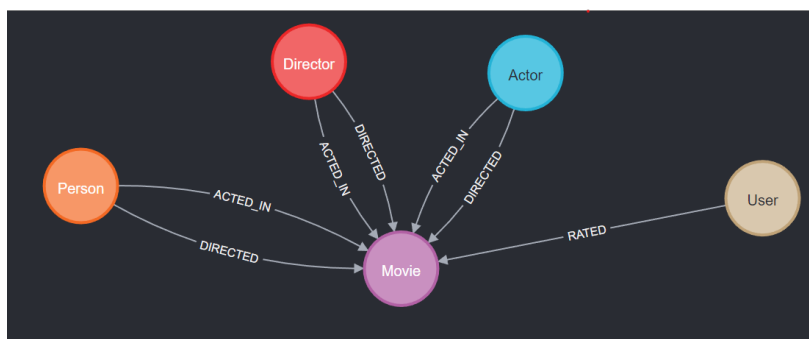


Figure 5.5: IMDB Dataset schema.

has been modified (without any changes to the dataset topology) such that the nodes have more attributes to be used as constraints and the edges are randomly distributed among the following three types - **CAUSES_SIDE_EFFECT**, **POSSIBLE_INTERACTION**, **ASSOCIATED_WITH**. Dataset parameters are listed in Table 5.1 and the schema is shown in Figure 5.6.



Figure 5.6: BioSNAP dataset schema.

Dataset	No. of Vertices	No. of Edges	Avg. node degrees(by type)
IMDB-1K	1,097	8,130	Movie(43), Person(1.6), User(6.4)
BioSNAP-10K	10,825	174,978	Compound(273.8), SideEffect(17.8)
Shipments-30K	30,000	40,000	Airport(1), Shipment(3.75), Flightleg(2.85), Carrier(3.98)

Table 5.1: Dataset parameters

5.2.3 Dataset-3: Shipments-30K Dataset

This dataset is inspired by supply chain logistics for cargo shipments (C2k) [20]. This data has been synthetically generated using Python libraries, and it consists of **4 types of nodes** - **Shipment**, **Airport**, **Carrier**, **Flightleg**. It has four types of relationships - **LOCATED_AT**, **PART_OF**, **OPERATED_BY**, **ASSIGNED_TO**. Dataset parameters are listed in Table 5.1 and the schema is shown in Figure 5.7.

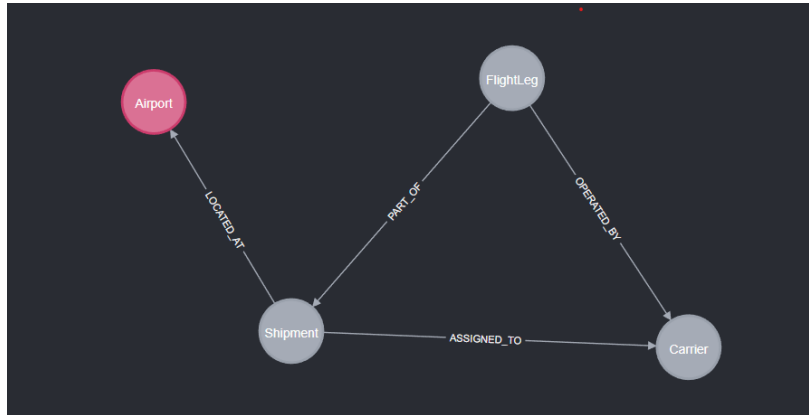


Figure 5.7: Shipments dataset schema.

5.3 Benchmarks: MASS Graph Database

This section covers the benchmarking performed on the MASS graph database system. It explores the CPU scalability that can be attained with multiple compute nodes in a cluster and how it affects the query execution times. Another aspect of this section is to analyze the effect of the evaluation mechanisms used during the execution phase. This is done by comparing the queries using filtering conditions across all traversal depths. The following are the benchmarks for each dataset with different configurations of the compute cluster.

5.3.1 CPU Scalability

Parallelism is achieved for MASS graph database over multiple compute nodes when the traversal depths increase and queries become more complex (i.e. multiple constraints). This is evident for datasets - BioSNAP-10K (5.2.2) and Shipments-30K (5.2.3), based on the benchmarks in Figure 5.8 and Figure 5.9.

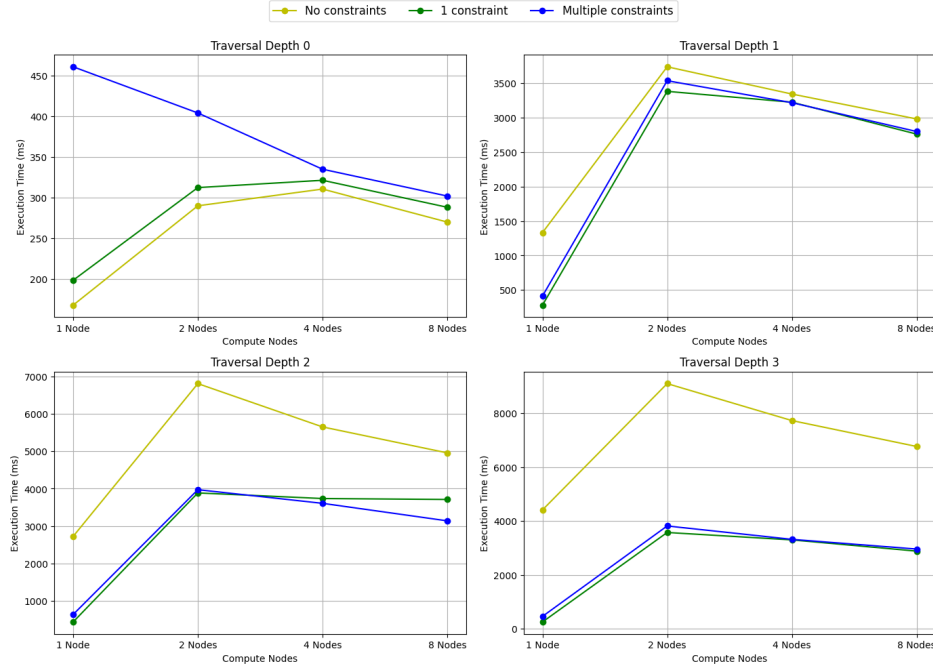


Figure 5.9: MASS graph database execution times for Shipments-30K.

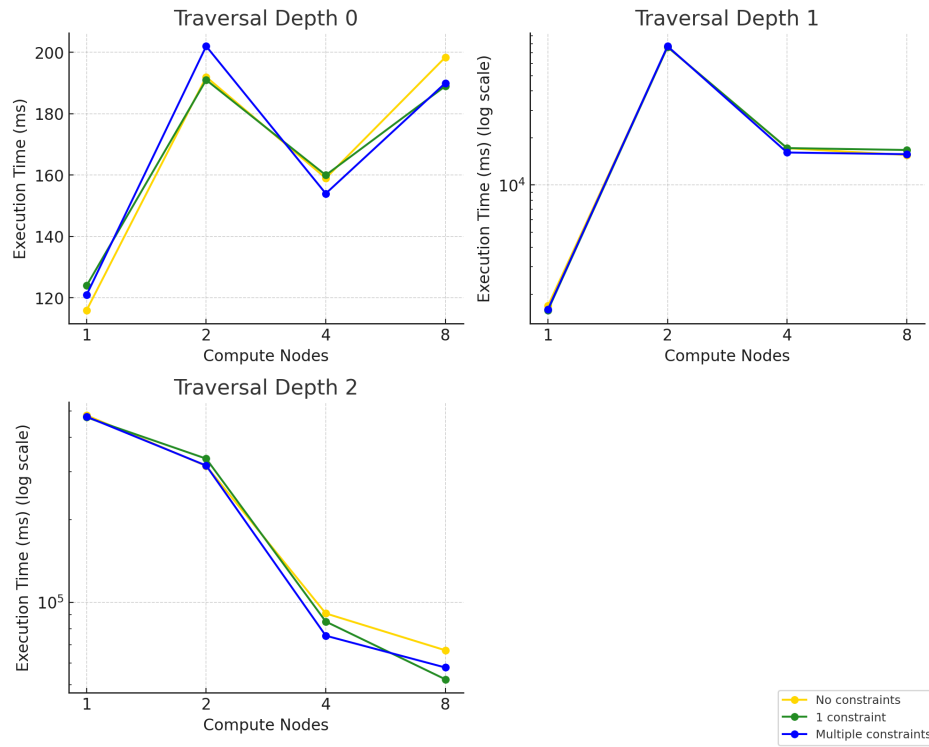


Figure 5.8: MASS graph database execution times for BioSNAP-10K.

Although, that is not the case for IMDB-1K (5.2.1), attributing to its smaller size, where communication overhead affects the runtime significantly as shown in Figure 5.10.

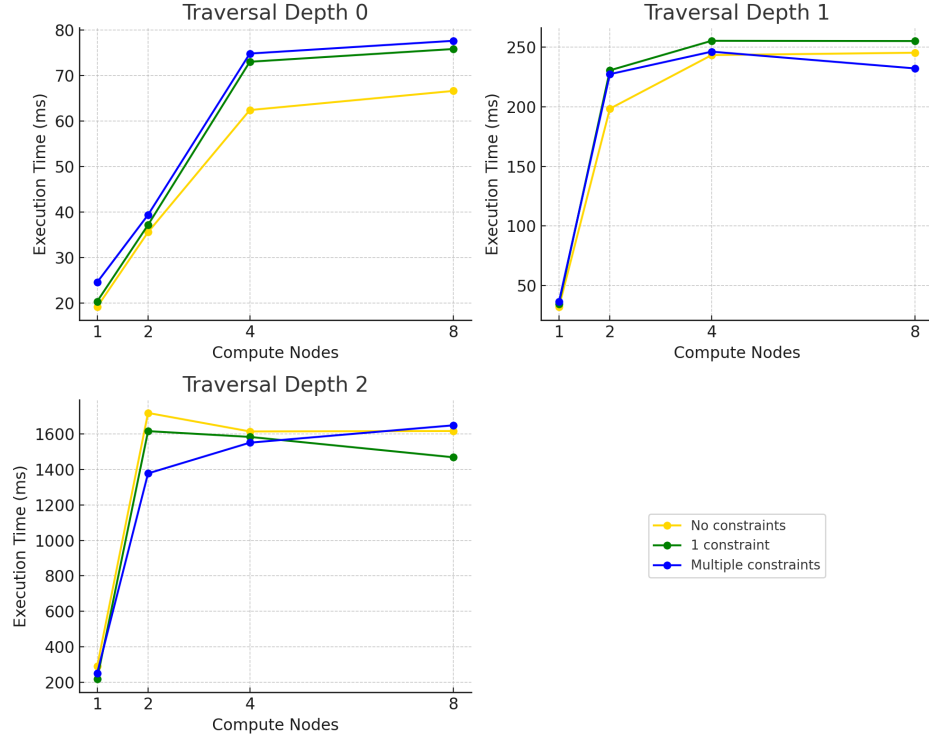


Figure 5.10: MASS graph database execution times for IMDB-1K.

5.3.2 Stack-based VS. Dynamic Evaluation performance

The bar-plot visualizations in this section use the blue color for dynamic (tree-based) execution and red for the stack-based execution mechanism. For the heatmap visualizations, blue tones indicate that the stack-based approach works better and red tones indicate the opposite (since the performance ratios calculated are stack/dynamic execution time). Neutral tones (white/off-white) imply that similar performance was achieved by both approaches.

The stack-based approach for expression evaluation works better than the dynamic (tree-based) approach for IMDB-1K (as shown in Figure 5.11 and performance ratios in Figure 5.12) and for single and two compute node configurations for BioSNAP-10K (as shown in the Figure 5.13 and performance ratios in Figure-5.14). This is attributed to the fact that the number of computations required in the stack-based approach is proportional to the number

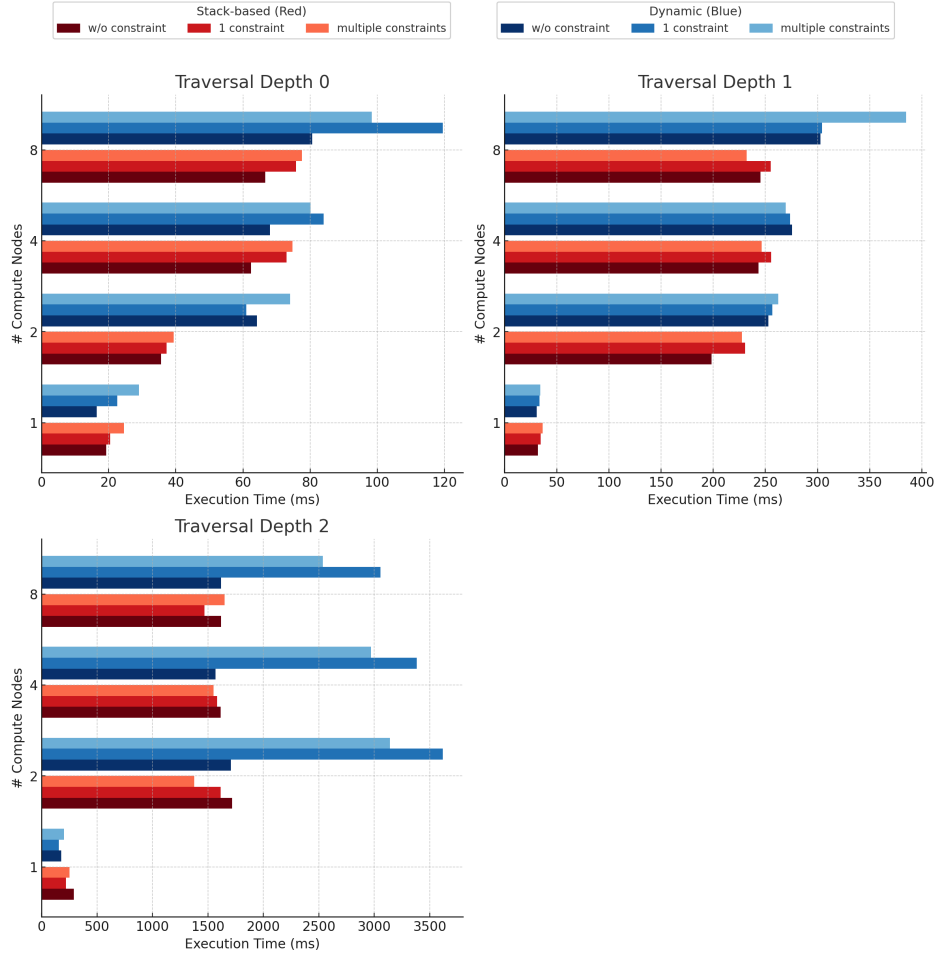


Figure 5.11: IMDB-1K - Stack vs Dynamic evaluation.

of vertices present in a graph, contrary to the dynamic approach which works better with a larger number of vertices in a graph.

The dynamic approach outperforms the stack-based approach for multiple constraints queries and deeper traversal depths for Shipments-30K (as shown in Figure 5.15 and performance ratios in Figure 5.16). This is due to the fundamental advantage provided by dynamically pruning agents on the incorrect paths, which prevents unnecessary computations.

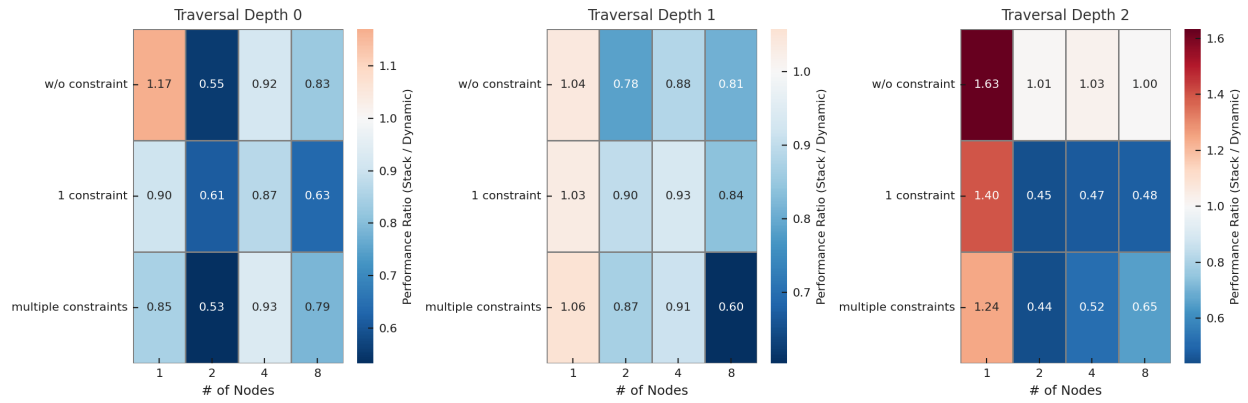


Figure 5.12: IMDB-1K - Stack vs Dynamic performance ratios

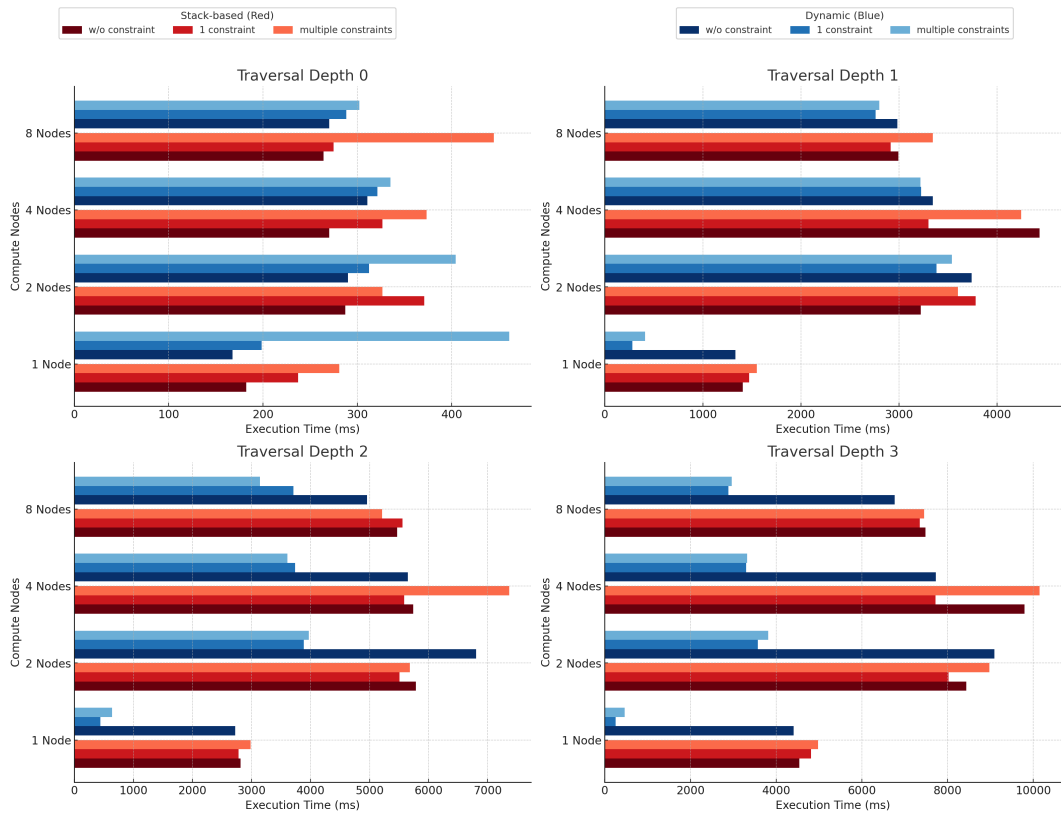


Figure 5.15: Shipments-30K - Stack vs Dynamic evaluation.

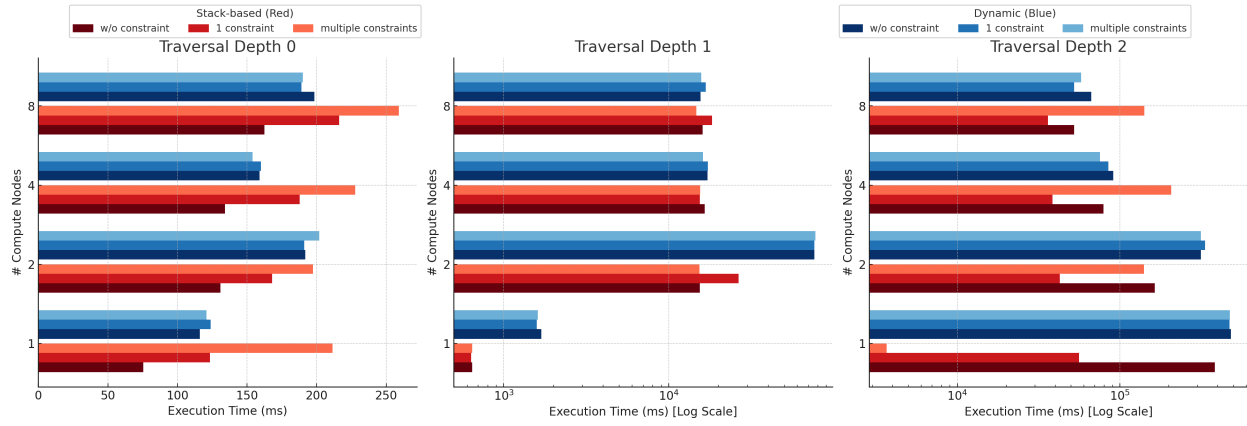


Figure 5.13: BioSNAP-10K - Stack vs Dynamic evaluation.

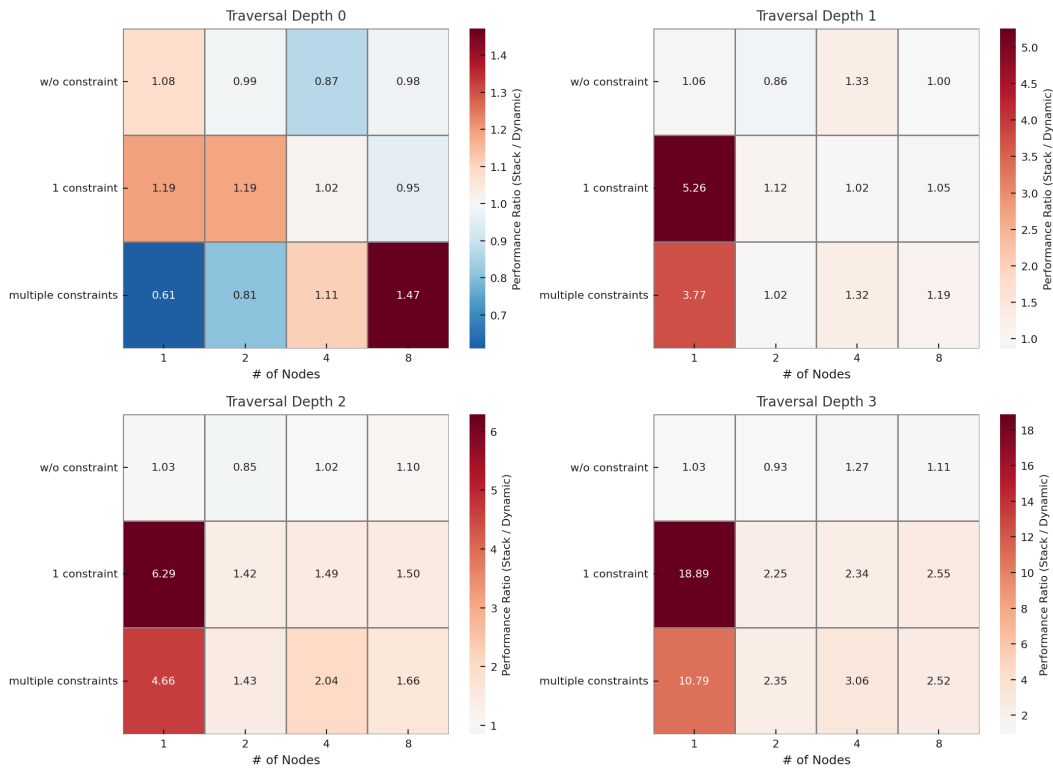


Figure 5.16: Shipments-30K - Stack vs Dynamic performance ratios.

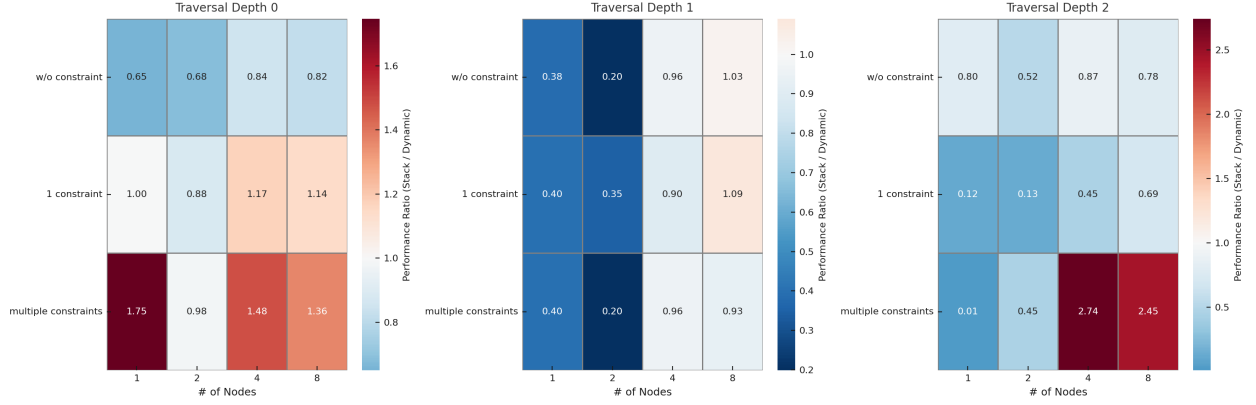


Figure 5.14: BioSNAP-10K - Stack vs Dynamic performance ratios.

5.4 Comparative Benchmarks

5.4.1 Single Compute Node: MASS vs. Neo4j vs. ArangoDB

This section covers the comparative benchmarks of three database systems running over a single compute node. These benchmarks are performed to compare the execution performance of the MASS graph database with two industry-standard graph databases - Neo4j and ArangoDB.

The visualizations in the form of heatmaps (shown in Figure 5.18, Figure 5.20 and, Figure 5.22) compare the ratio of performance of each of these systems compared to the MASS graph database. Blue tones indicate the cases where MASS performs better and red tones for the opposite case.

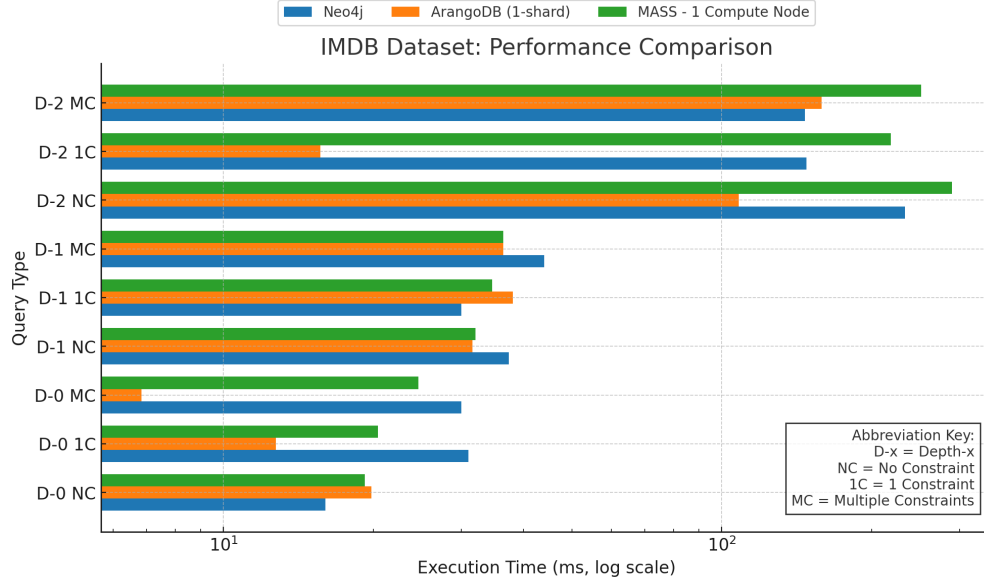


Figure 5.17: IMDB-1K - Single compute node performance

In case of single compute node, the MASS graph database performs better than other databases for IMDB-1K, in some cases, at Depth-0 and Depth-1 traversals (as shown in Figure 5.17 and performance ratios in Figure 5.18) and for Shipments-30K at Depth-3 traversal with single-constraint filtering (refer Figure 5.21).

Benchmarking results for the BioSNAP-10K dataset suggest that the MASS graph database suffers in performance as the average node degree increases (based on Figure 5.19 and Figure 5.20). In the MASS graph database, each agent is a separate execution thread. As dataset sizes increase (particularly the number of vertices), the number of active agents at a single compute node also increases, which leads to performance degradation once the optimal thread count limit is reached for the CPU.

One consideration for the competitiveness of the MASS graph database in single compute node performance is that it currently operates without any indexing or caching optimizations, which both Neo4j and ArangoDB employ.

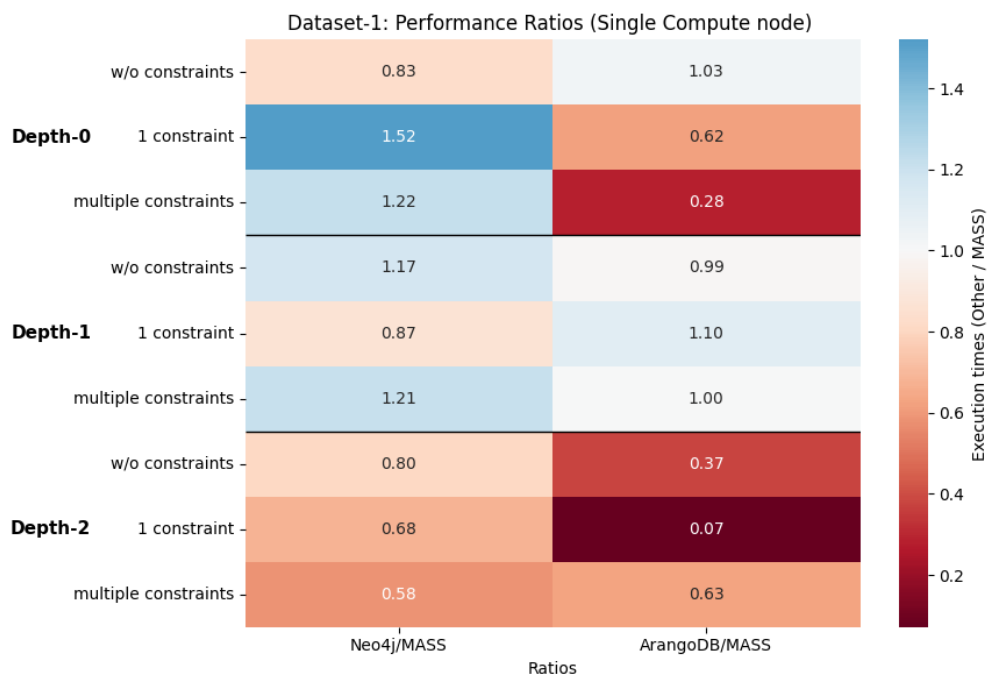


Figure 5.18: IMDB-1K - Heatmap - single compute node execution performance

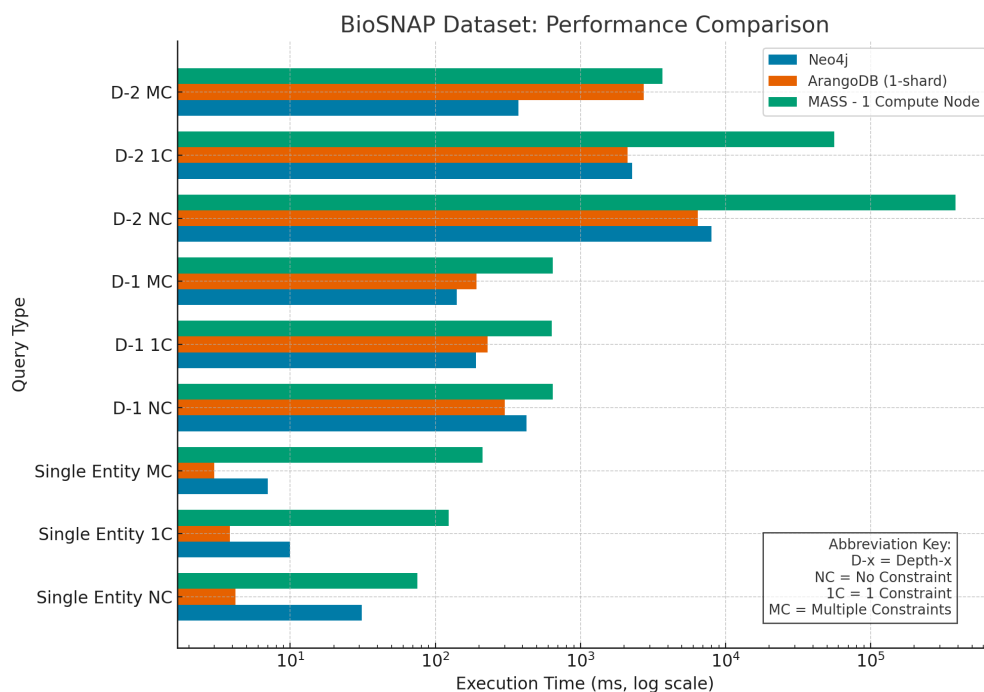


Figure 5.19: BioSNAP-10K - Single compute node performance

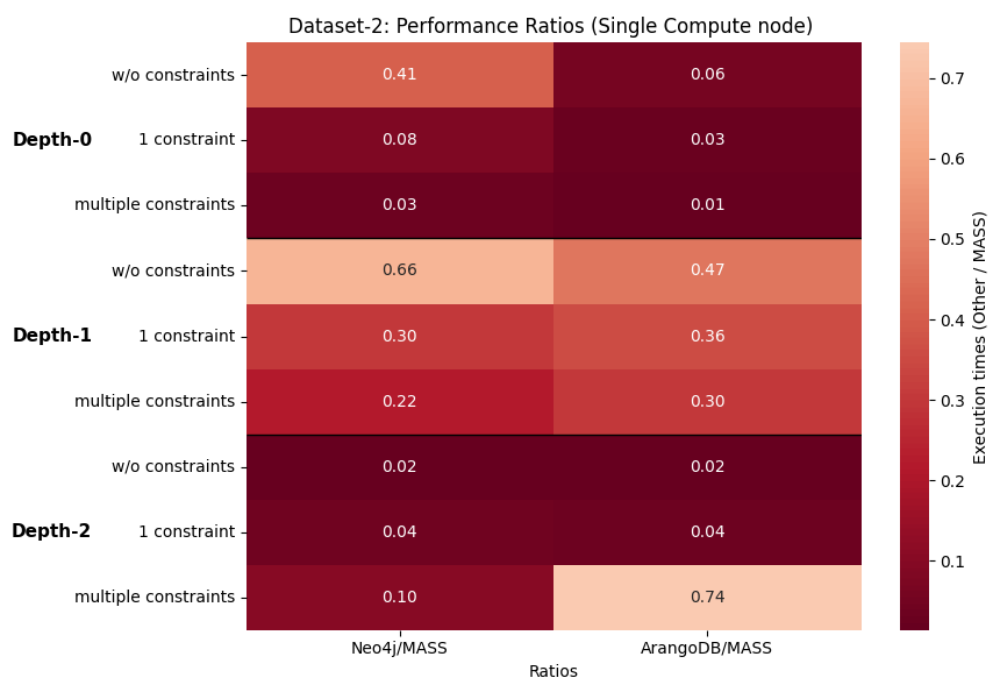


Figure 5.20: BioSNAP-10K -Heatmap - single compute node execution performance

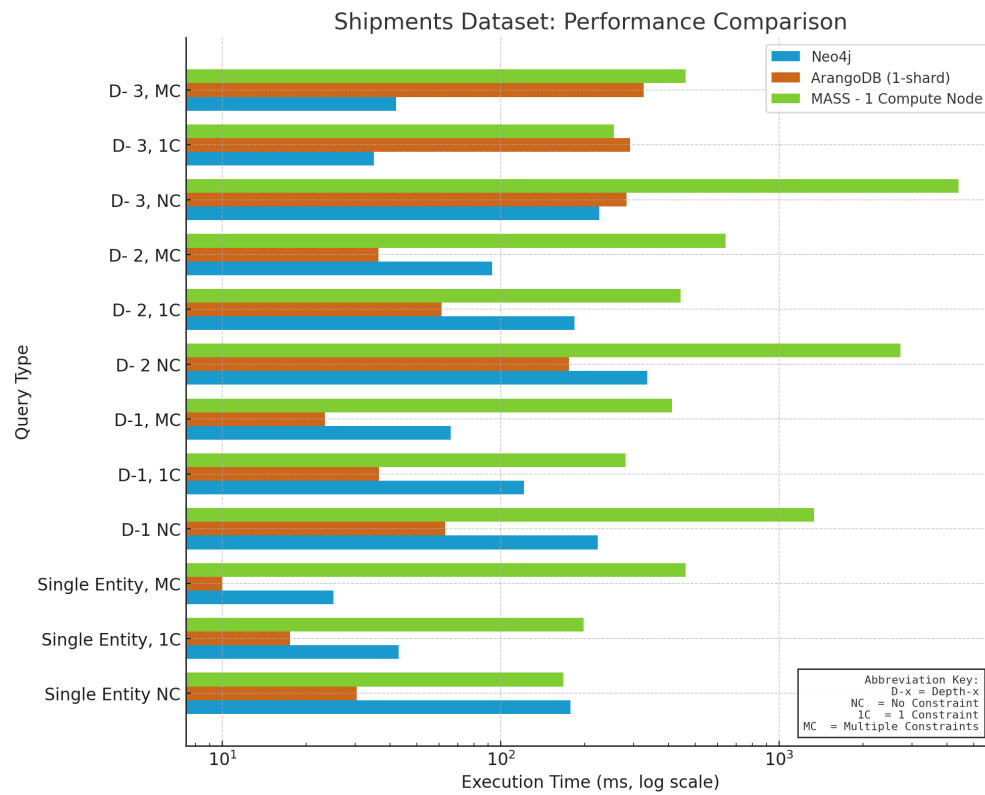


Figure 5.21: Shipments-30K - Single compute node execution performance

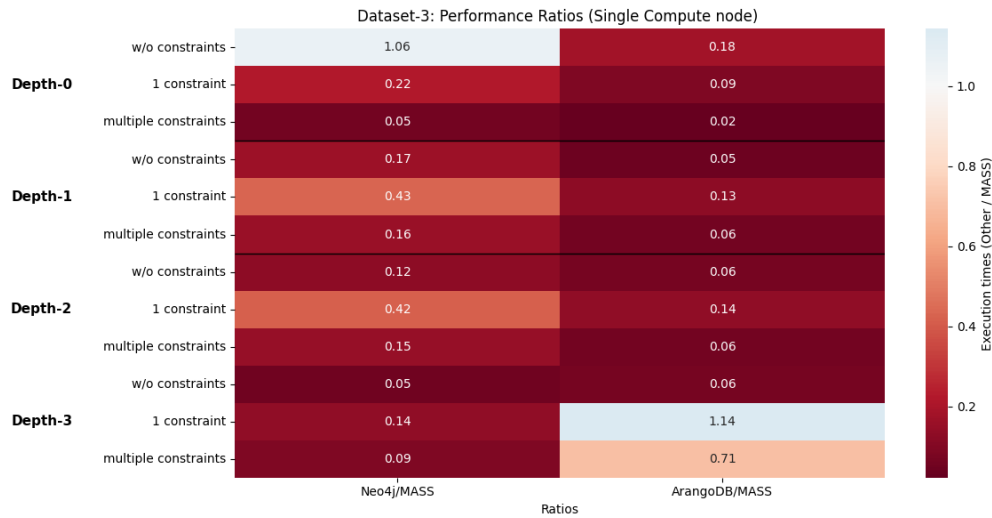


Figure 5.22: Shipments-30K - Heatmap - single compute node execution performance

5.4.2 Multiple Compute Nodes: MASS vs. ArangoDB

This section covers the comparative benchmarks of CPU scalability for the MASS graph database and ArangoDB. Important considerations here are that ArangoDB's benchmarks are performed over a cloud instance and MASS graph database over the cssmpi cluster.

The visualizations in the form of heatmaps (shown in Figure 5.24, Figure 5.26 and, Figure 5.28) compare the ratio of performance of each of ArangoDB compared to the MASS graph database. Blue tones indicate the cases where MASS performs better, and red tones indicate the opposite case.

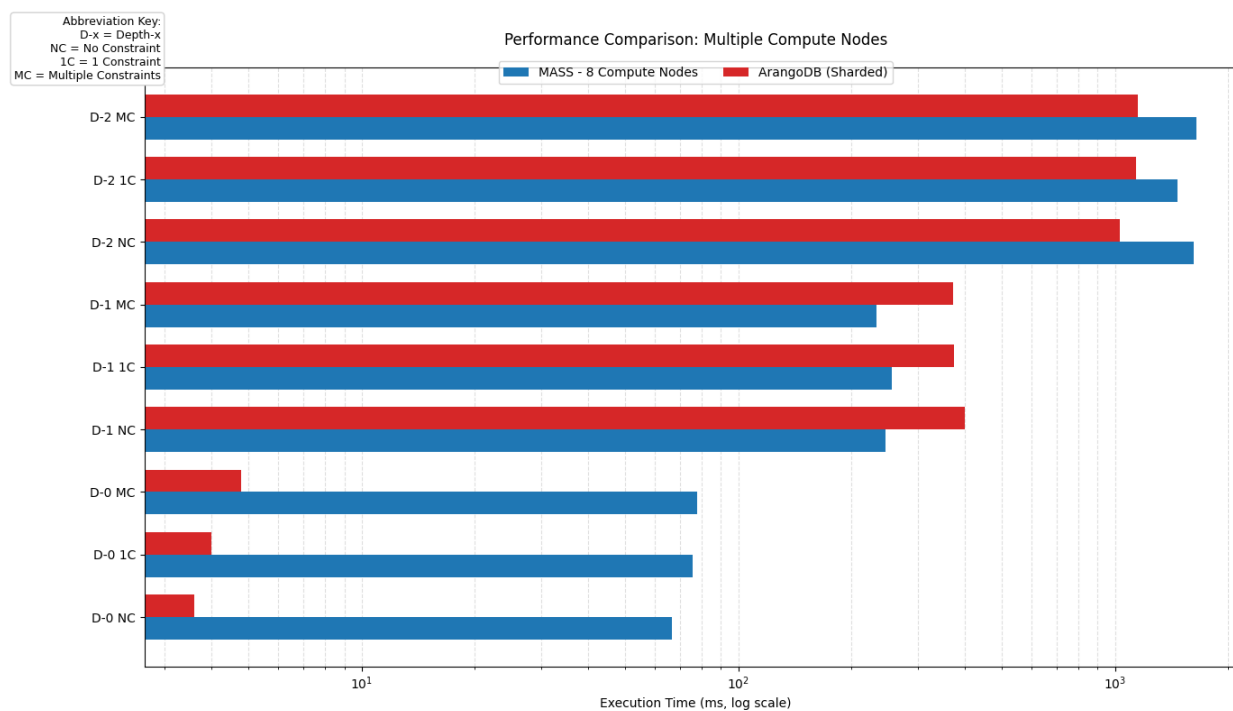


Figure 5.23: IMDB-1K: Multiple compute node performance

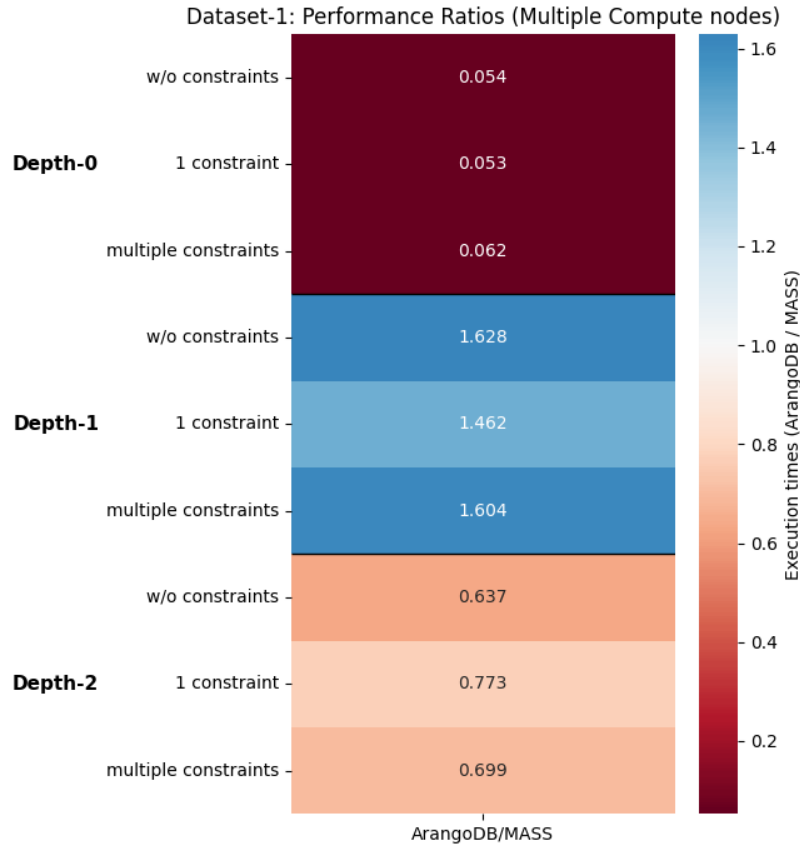


Figure 5.24: IMDB-1K - Heatmap - multiple compute node execution performance

MASS graph database outperforms ArangoDB consistently at Depth-1, Depth-2 and Depth-3 traversals for the Shipments-30K (as shown in Figure-5.27 and Figure 5.28), at Depth-1 traversals for IMDB-1K (as shown in Figure 5.23 and Figure 5.24) and at Depth-2 traversals (apart from multiple constraints case) for BioSNAP-10K (as shown in Figure 5.25 and Figure 5.26). This is due to the agent-based traversal approach complemented by the dynamic constraints evaluation approach.

Due to the indexing mechanisms in place for ArangoDB, it performs better than MASS at all depth-0 traversals. This optimization from indexing decreases as the traversal depths increase and data needs to be fetched from a different computing node.

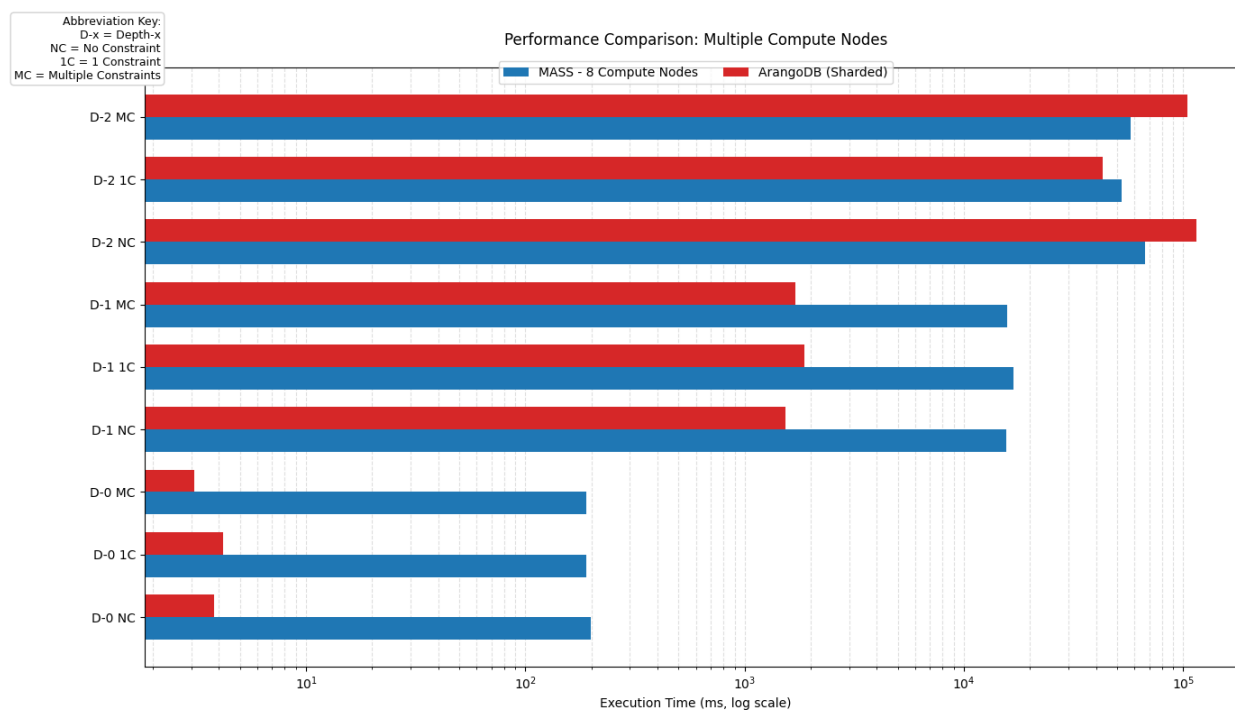


Figure 5.25: BioSNAP-10K: Multiple compute node performance

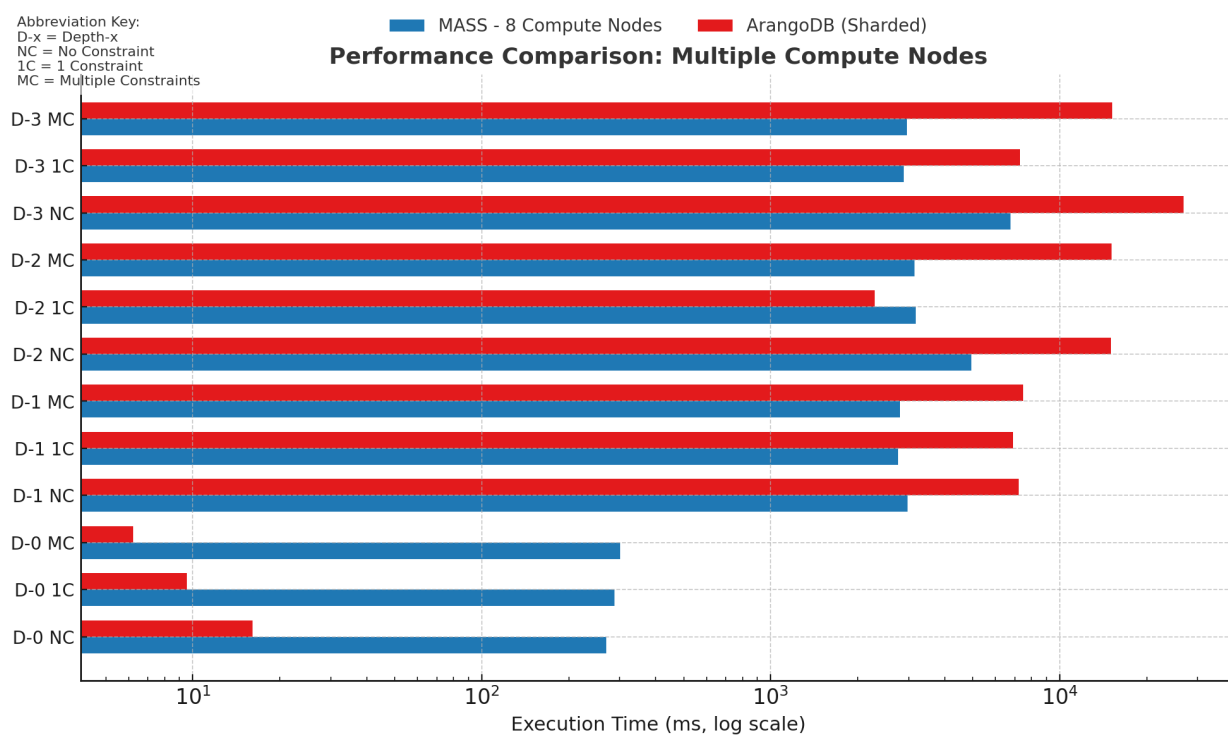


Figure 5.27: Shipments-30K: Multiple compute node execution performance

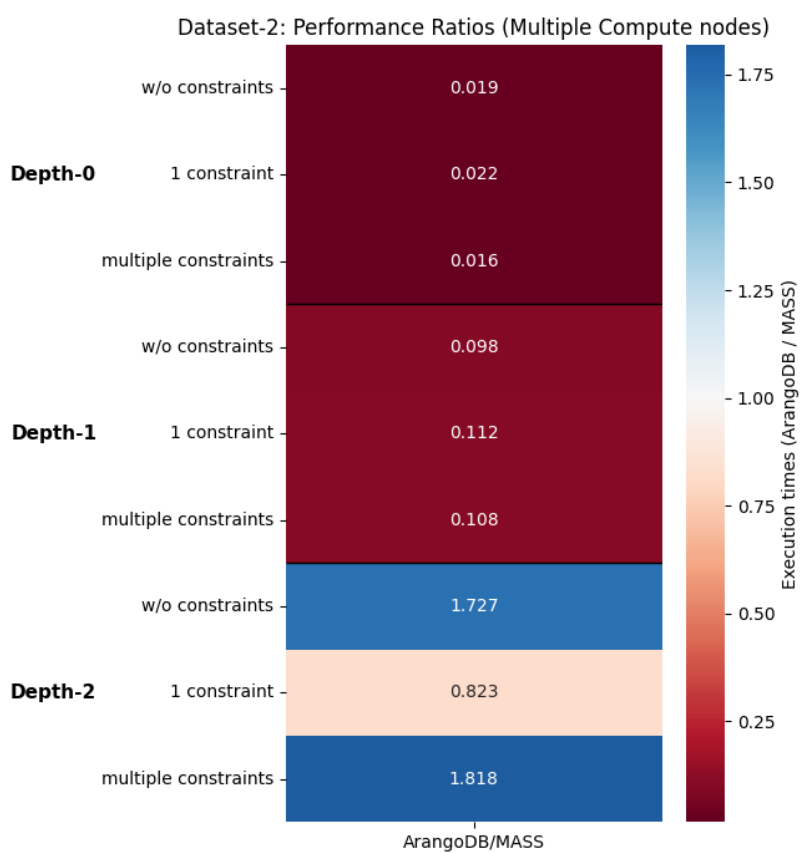


Figure 5.26: BioSNAP-10K - Heatmap - multiple compute node execution performance

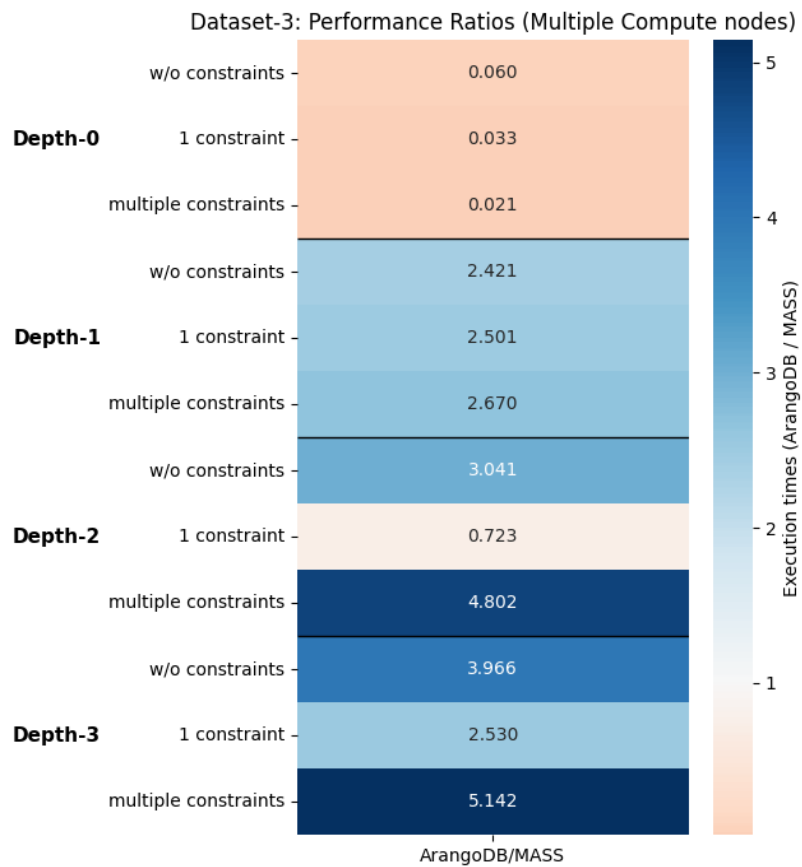


Figure 5.28: Shipments-30K - Heatmap - multiple compute node execution performance

Chapter 6

CONCLUSION

In this project, we embarked on a thorough exploration of enhancing the querying capabilities of our existing agent-based graph database system. The objective was not only to improve the range and expressiveness of the queries possible, but also to explore the fit for a declarative query language like CypherQL with agent-based modeling and execution mechanisms employed by the MASS Java Library.

6.1 *Contributions*

Below is the summary of the work done in this project:

- A comprehensive review was conducted into the existing works on query processing and the mechanisms used for distributed systems to identify any limitations and improve upon it. Furthermore, the impact of query complexity on execution performance was identified, which led to the exhaustive approach to benchmarking with varying types of constraints.
- To obtain generalizable results and to explore the strengths and weaknesses of our graph database system, the benchmarking was performed over datasets having a diverse set of topologies - ranging from bi-partite graphs to complex flow of supply chain logistics.
- Lead the end-to-end development of the WHERE Clause integration into the MASS graph database system. Designed and integrated two evaluation mechanisms - stack-based and dynamic (utilizing three-valued logic with trees), both of which provided the required filtering capabilities while improving the performance of the graph database system.

6.2 Limitations

Despite the progress achieved with this project, two limitations persist, which require the following improvements. Firstly, while the implementation successfully incorporates the WHERE clause functionality for the scope of this project, there remains a need to improve the way graph traversals are initiated. Currently, agents are being spawned across all `PropertyGraphPlace` instances which leads to an overhead to begin with and can be resolved by employing metadata-based measures like anchor nodes (i.e., determining the efficient starting vertices for graph traversals using the graph metadata). Secondly, there also exists a master-worker mode of control transfer during the iterative `callAll()` and `manageAll()` execution cycle, where agents need to be migrated and synchronized across all cluster nodes. Providing true autonomy to `PropertyGraphAgents` along with optimized graph traversals using anchor nodes would lead to less communication overhead and better querying performance.

6.3 Future work

The future works from this project include:

- Determining the limit for the number of constraints for the WHERE clause to avoid performance issues.
- CRUD Operations in MASS Graph Database: From an application developer's standpoint, the development of DELETE clause, which also relies on the results of MATCH and WHERE clauses, will facilitate efficient removal of nodes and relationships from the graph. Furthermore, improving the RETURN clause implementation, which enables formatting of retrieved results, making the MASS Graph Database system more robust and adoptable.
- Aggregation functionality: In many graph analytics scenarios, being able to perform aggregations for attributes within the database is a powerful utility. Cypher provide

clauses such as COUNT, COLLECT, SUM, etc. which can prove to be beneficial for future developers and user of the MASS Graph Database System alike.

BIBLIOGRAPHY

- [1] B. Zhao, K. Kumar, G. Casey, and K. Soga. Agent-based model (abm) for city-scale traffic simulation: A case study on san francisco. In *International Conference on Smart Infrastructure and Construction 2019 (ICSIC): Driving data-informed decision-making*. ICE Publishing, 01 2019.
- [2] Vishnu Mohan, Anirudh Potturi, and Munehiro Fukuda. Automated agent migration over distributed data structures. In *ICAART (1)*, pages 363–371, 2023.
- [3] Taylor M. Anderson and Suzana Dragičević. Network-agent based model for simulating the dynamic spatial network structure of complex ecological systems. *Ecological Modelling*, 389:19–32, 2018.
- [4] Yuanyuan Tian. The world of graph databases from an industry perspective. *SIGMOD Rec.*, 51(4):60–67, January 2023.
- [5] Yuan Ma, Michelle Dea, Lilian Cao, and Munehiro Fukuda. Toward implementing an agent-based distributed graph database system. In *2024 IEEE International Conference on Big Data (BigData)*, pages 3456–3465, 2024.
- [6] ArangoDB Documentation. About arangodb. <https://docs.arangodb.com/stable/about-arangodb/>. Accessed: 2025-08-14.
- [7] Neo4j Documentation. What is neo4j? <https://neo4j.com/docs/getting-started/whats-neo4j/>. Accessed: 2025-08-14.
- [8] Hypermode Blog. Top use cases for graph databases. <https://hypermode.com/blog/use-case-graph-database/>, Jul 9 2024. Accessed: 2025-08-15.

- [9] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefer. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Comput. Surv.*, 56(2), September 2023.
- [10] Graphable Blog. Neo4j performance architecture explained & 6 tuning tips. <https://www.graphable.ai/blog/neo4j-performance/>, September 2021. Published 3.9 years ago; Accessed: 2025-08-17.
- [11] Neo4j Documentation. Introduction — cypher manual (v 4.4). <https://neo4j.com/docs/cypher-manual/4.4/introduction/>. Accessed: 2025-08-17.
- [12] Johan Sandell, Einar Asplund, Workneh Yilma Ayele, and Martin Duneld. Performance comparison analysis of arangodb, mysql, and neo4j: An experimental study of querying connected data, 2024.
- [13] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost Depth-First-Search distributed Graph-Querying system. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 209–224. USENIX Association, July 2021.
- [14] Hamed Dinari. A survey on graph queries processing: techniques and methods. *International Journal of Computer Network and Information Security*, 9(4):48, 2017.
- [15] Omar Batarfi, Radwa Elshaw, Ayman Fayoumi, Ahmed Barnawi, and Sherif Sakr. A distributed query execution engine of big attributed graphs. *SpringerPlus*, 5(1):665, 2016.
- [16] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

- [17] Wikipedia contributors. Three-valued logic. https://en.wikipedia.org/wiki/Three-valued_logic. Last edited: 26 July 2025; Accessed: 18 August 2025.
- [18] Neo4j Documentation. Common datasets — neo4j graph data science client. <https://neo4j.com/docs/graph-data-science-client/current/common-datasets/>. Accessed: 2025-08-19.
- [19] SNAP (Stanford Network Analysis Project). Drug side-effect association network (chse-decagon). <https://snap.stanford.edu/biodata/datasets/10018/10018-ChSe-Decagon.html>. Accessed: 2025-08-19.
- [20] Chris Crawford – Kaggle. Cargo 2000 dataset. <https://www.kaggle.com/datasets/crawford/cargo-2000-dataset>. Accessed: 2025-08-19; A transport and logistics case study dataset from Cargo 2000 system tracking events over five months (as described on Kaggle).

Chapter 7

APPENDIX

7.1 *Code and Benchmarking queries*

The actual queries used for benchmarking, along with the datasets and import instructions are stored in this drive **link**.

MASS Java Library has two repositories: -

1. **mass-java-core**: This repository contains all the core aspects of the database engine and the PropertyGraphPlaces, PropertyGraphAgent and PropertyVertexPlace abstractions. Furthermore, the code for different evaluation mechanisms is maintained in two different branches. The link to the repository is present **here**.
 - **aatmanrp/mass-refactoring**: this branch contains the stack-based evaluation mechanism for WHERE clause.
 - **aatmanrp/where-optimization**: this branch contains the dynamic (tree-based) evaluation mechanism for WHERE clause.
2. **mass-java-appl**: This repository contains the MASS Graph Database Application. It contains the interactive part of the application, allowing users to query the graph database and perform benchmarks. The changes for my implementation are present in the **aatmanrp/graph-database** branch. The link to the repository is present **here**.

Run instructions for MASS Graph Database:

- Build the MASS core snapshot in the mass-java-core directory using: `mvn -DskipTests clean package install` command.

- Next, build the MASS GraphDB application by switching to the mass-java-appl directory at: `/home/NETID/netid/mass_code/mass_java_appl/QueryGraphDB`
- Now setup the nodes.xml file with the required configuration for MASS cluster and make sure to have the masshome tag set to the location you want to run the application from. In my case, it is set to the path mentioned above.
- Use the bash scripts to run the graph database on the dataset of your choice. If you put the command 'benchmark' after the run command, it will run the graph database in benchmarking mode, otherwise it runs in the interactive mode.