An Agent-based Graph Database

Harshit Rajvaidya

A white paper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2023

Project Committee:

Prof. Munehiro Fukuda, Chair

Prof. Bill Erdly, Member

Prof. Min Chen, Member

University of Washington

Abstract

Agent based Graph Database

Harshit Rajvaidya

Chair of the Supervisory Committee:

Dr.Munehiro Fukuda

Computing & Software Systems

Graph databases are a type of NoSQL databases that use graph structures to store and represent data. Unlike traditional relational databases that use tables and rows to represent data, graph databases use nodes and edges to represent relationships between data items. This allows for more flexible and efficient querying of complex and connected data items. Graph databases provide us with functional capabilities of querying a large number of interconnected data schemas, such as social networks and biological networks. In this project, we aim to build a graph database using the MASS (Multi-Agent Spatial Simulation) library that relies on Places and Agents as the core components. The MASS library has already supported graph data structure (GraphPlaces) which is distributed on a cluster of computing nodes. However, the current implementation worked on specific graph types. This project implements graph creation using CSV files as generic inputs as possible. We also implement a query-parsing engine that takes OpenCypher queries as inputs and parses it to method calls of MASS GraphPlaces. On top of that we have implemented four types of queries (including where clause, aggregate type, and multi relationship queries) in order to perform verification of the graph database and to perform query benchmarks. Each benchmark measures the query latency, graph creation times, and spatial scalability of all the queries. The performance measurements are performed on a cluster of eight computing nodes, and the spatial scalability is measured using a Twitch monthly dataset, which contains more than 7k nodes and more than 20k edges. The research presents significant improvements in query latency and spatial scalability as we increase the number of computing nodes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1. Introduction

In the dynamic world of big data analytics, traditional approaches have predominantly relied on data streaming frameworks such as MapR, Apache Spark, and Apache Storm to process and analyze massive volumes of information. These frameworks have been instrumental in facilitating real-time data processing and scalable analytics pipelines. However, it is crucial to recognize that big data computing encompasses more than just data streaming. There also exists another big-data computing domain that is equally important and evolving increasingly, that is, the analytics of complex data structures such as graphs and arrays.

While data streaming frameworks excel in handling continuous data flows, they may not be optimally suited for the analysis of interconnected data points, where relationships play a pivotal role. Graph databases, in particular, have gained popularity as a practical big data computing domain, as they provide a robust foundation for representing and analyzing relationships between entities. However, constructing and analyzing such complicated data structures over distributed memory while ensuring smooth operations and reusable methodologies can pose significant challenges.

To address these challenges and unlock the full potential of graph databases and other data structures, we propose the adoption of agent-based graph computing. This novel paradigm leverages the power of distributed memory and intelligent agent dispatching to enable efficient construction, analysis, and reuse of complex data structures. By combining the strengths of agent-based systems and graph computing, we aim to demonstrate the numerous advantages and transformative capabilities that this approach offers.

In this white paper, we delve into the concept of agent based computing and explore how agent migrations can overcome the limitations of traditional graph data storage.
The subsequent sections of this white paper will provide a comprehensive analysis of agent-based graph computing, its underlying principles, and its implementation as a standalone database. We will present the different types of queries that have been implemented, and how performance is affected. We will also dive into the MASS library that provides the entire skeleton to this project.

1.1. Background

MASS [1] (Multi-Agents Spatial Simulation) is an agents-based parallel programming library to do computation over a cluster of nodes . It provides an intuitive programming framework to do big data processes and can simulate a lot of real-life problems including bioinformatics, climate change, social networking, etc. A cluster of computing nodes fork a process at each node, and all the processes communicate with each other via TCP connections. There are two key concepts in the MASS library: Places and Agents. Places is a distributed array of place elements over a cluster of computing nodes. It is managed by a set of global indices and each place element can be identified with an index. Data can be saved in a specific place and exchanged between Places instances, (each simply referred to as a place).

Agents are a set of execution instances that can migrate over places. Each agent object (referred to as an agent in the following discussion) can specify the next place's index to indicate where to migrate next. When an agent resides in a place, it can manipulate the data saved on that place. Agents are grouped into bundles. On each computing node, multiple threads check-in each agent one by one and process each agent's request. On top of the Places class, MASS also supports other data structures including Binary Tree, QuadTree, Continuous Space [1], and Graph [3]. Those agent-navigable can better meet users' various computing needs.

1.2. Motivation

Graph is an abstraction of relationships in nature, which has direct application to many real world problems such as biological networks, social networks and neural networks. Therefore, the study of graphs, having the ability to extract information from any graph has a very strong and practical significance. This significance is the major motivation of this project, to create a Graph Database capable of understanding and interpreting the queries.

With a tremendous growth of ML and AI, the volume of associated data in industry as well as in academia is growing manifolds, often reaching to the level of petabytes (1024 terabytes) and exabytes (1024 petabytes). The data can include billions and trillions of records of people, including their habits, their details, which can transform to generation of graphs with millions of vertices and edges. Analyzing and fetching information from these graphs has become very important. The principle task of fetching information from any datasource is to provide an ease of interaction with the data. It is very challenging to process and compute the rapidly growing datasets in a reasonable amount of time, as well as provide with the ability to fetch data on the fly with the limited computing and memory resources available. In order to provide the performance and immediate data delivery, there have been a few implementations of graph databases such as Neo4j [4] and RedisGraph [5].

Based on the purpose of comparing the performance and programmability of different implementations of the graph database, we have come up with an Agent Based Graph Database Model. The model is based on MASS Java, and employs all the features of the MASS library. The Agent Based Graph will draw parallels from Neo4j and RedisGraph for comparing query translations, and will do a performance and data accuracy analysis with different types of applications.

1.3. Goal

Based on the motivations above, the project goals for this capstone work are defined as follows:

- Design and implement a generic data parser, where when provided with CSV data files, it reads and translates the given rows into Graph Nodes and Edges.
- Design and implement a query parser engine that can interpret OpenCypher queries.
- Design and implement basic graph queries that give us an understanding of general operations that can be performed on a graph.

The rest of the white paper is organized in the following manner: Chapter 2 discusses the background and the related work done by existing applications such as Neo4j. Chapter 3 gives a detailed understanding of the execution process and the multiple components involved from when the application is initialized to returning of results. Chapter 4 dives deep into the implementation and the business logic behind every query that the graph database currently supports. Chapter 5 evaluates the performance, scalability and verification of the graph database application. Chapter 6 summarizes the conclusions and future improvements of this white paper.

2. Related Work

2.1. Neo4j

Neo4j is a leading graph database that enables users to efficiently model, store, and query complex data relationships. As a graph database, it leverages the inherent flexibility of graphs to represent data in a way that closely reflects real-world relationships. With its powerful query language, Cypher, and support for multiple programming languages, Neo4j helps build applications in a much quicker and faster manner that leverage the full power of graph data.
The core element of the Neo4j architecture is the storage engine, which is responsible for persisting the data to the disk and provides optimization for the storage.
The storage engine also enables the functionality of scaling a neo4j cluster to many parallel running machines.
One of the key advantages of Neo4j is its scalability. It is designed to handle large and complex datasets, making it a popular choice for enterprise-level applications. With its support for clustering and sharding, Neo4j can easily scale horizontally to meet the needs of high-traffic applications. However, the core problem with scalability in a Neo4j cluster is that since the data is stored on to the disk, the retrieval of data is slower for heavy queries. This serves as a bottleneck in large scale applications where the query needs to respond quickly.

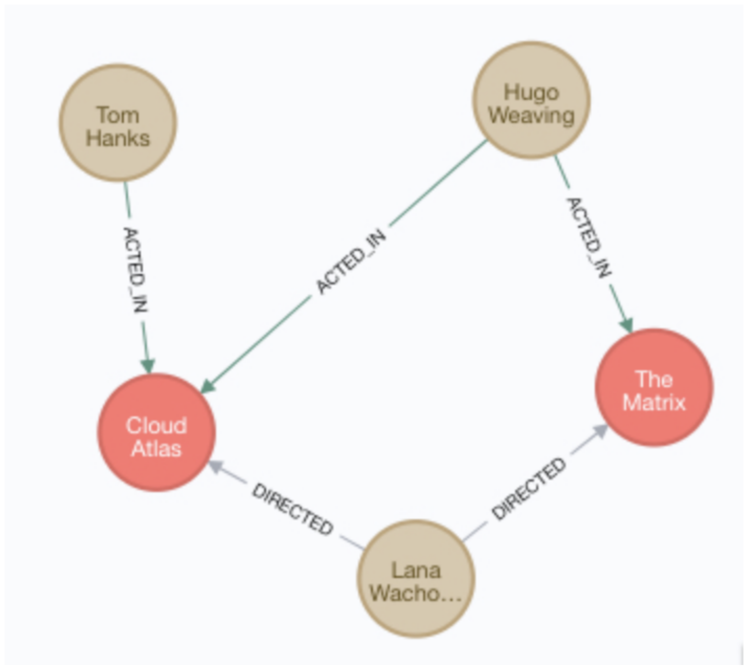


Figure 1: Visual representation of a Neo4j Graph

2.2. RedisGraph

RedisGraph is a high-performance graph database module that is built on top of Redis, the popular in-memory data structure store. It is designed to handle large and complex datasets with high performance and accuracy. With its ability to leverage the power of Redis, RedisGraph offers a fast, scalable, and flexible solution for building graph-based applications. RedisGraph is highly optimized for graph queries and is ideal for use cases that require real-time graph analysis, such as fraud detection, recommendation engines, and social networks.

One of the key benefits of RedisGraph is its ease of use. It offers a simple and intuitive query language that is easy to learn and use. This makes it easy for developers to get started with graph databases and start building powerful applications. Additionally, RedisGraph offers a powerful set of APIs for accessing and manipulating graph data, making it easy to integrate with other technologies and tools.

Another important aspect of RedisGraph is its scalability. It is designed to handle large datasets and high volumes of queries with ease, making it a good choice for mission-critical applications that require high performance and scalability. RedisGraph leverages the power of Redis to offer features such as clustering and sharding, which enable vertical scaling and ensure high availability and fault tolerance. With its ability to handle complex graph queries in real-time, RedisGraph is a powerful solution for building applications that require advanced data modeling and analysis.

However, the core problem with vertical scalability is that the cost incurred to increase the size of a single machine would be extremely high, and it also adds an upper bound limit to the size of data it can handle. Since, redis graph is a memory store single machine graph database, it cannot scale vertically.

2.3. Summary

Neo4j and RedisGraph are very popular and widely used graph databases and they provide a huge variety of analytics tools for processing large amounts of data. However, they do come with limitations. RedisGraph works on a memory model and a key value storage system, hence it provides faster retrieval for the data stored. But due to the memory model architecture, it is not scalable for very large datasets. Ne04j, on the other hand, works on a disk based model which makes it an ideal candidate for large datasets, but due to the disk based architecture all reads and writes are slow in comparison.

With an Agent Based Graph database, which is based on MASS architecture we aim to mitigate both of these limitations. MASS architecture enables us to define a distributed memory over a set of compute nodes. Each graph node that resides on this distributed space provides us with a constant access time, thus mitigating the slow retrieval issue by neo4j. Additionally, with the MASS architecture, it is possible for us to scale the computing nodes horizontally, hence mitigating the scale issue by RedisGraph.

3. Execution Mode

This chapter aims to provide an in-depth overview of the MASS Java library, the current graph implementation on which this project's work is based. It also talks about the format of data that we have used for successful completion of this project and also shed some light on the system architecture and design decisions for this project.

3.1. MASS Library

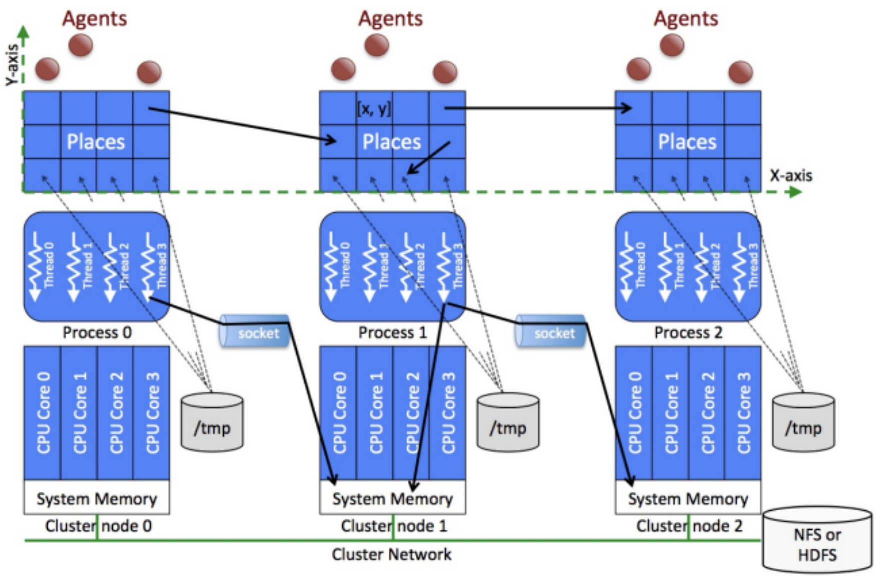


Figure 2: MASS Library Data Model

MASS is an agent-based parallel computing library for multi-agent and spatial simulation over a cluster of computing nodes. MASS started as a research project at the of Distributed Systems Laboratory (hereinafter refers to as “DSLab”) led by Professor Munehiro Fukuda of University of Washington Bothell in 2010, currently available in several languages, including Java [6], C++ [7], and CUDA [8]. Over the years, many of the ideas behind the system were presented in various journals and research papers over the years. MASS provides an intuitive programming framework for big data processing that can simulate many real-world problems, such as bioinformatics, social networks, geographic information systems and climate analysis . MASS also supports other data structures, including arrays, binary trees, quadtrees, continuous spaces, which can better meet the needs of users in various situations.

3.2. GraphPlaces

The MASS library has already supported the graph data structure. It was originally implemented by Justin Gilroy [9] and refactored by Brain Luger [3]. In the MASS library, the graph data structure is called GraphPlaces. GraphPlaces is an extension of the existing MASS Places class with the capability to support simulations. Each GraphPlaces consists of vertices. Vertices with the graph are distributed across all nodes as shown in Figure 3. When we add vertices to the graph, we will follow a round-robin fashion so that the vertices are balanced across the entire cluster. Each vertex was represented by a VertexPlace object and the VertexPlace class is an extension of the MASS Place class. The vertex contains a list of outgoing edges and information about the vertex itself.

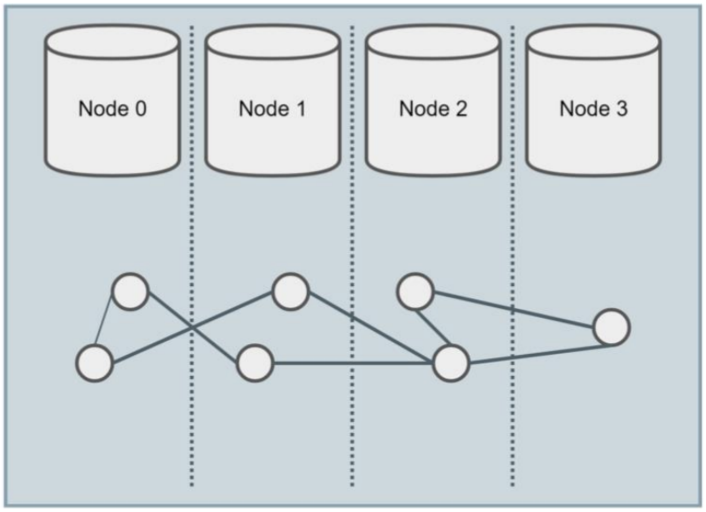


Figure 3:Visual representation of a distributed graph

When constructing a graph, users can load the graph from a supported graph format including Hippie, Matsim, UW-Bothell proprietary DSL, and SAR. Users can also choose to add vertices and edges manually by calling Addvertex() and AddEdge() methods. On each computing node, vertices are stored in a single vector as shown in Figure 4. The single vector can grow dynamically with the size of the graph so that we don't have to know the graph size in advance. The implementation also has a removal queue as shown in the red rectangle of Figure 4. It was used to recycle the removed vertices so that we don't have to shift all the elements in the container. When a vertex is removed, it will be added to the queue. On the next call to addVertex(), the recycled vertices will be dequeued. After constructing a graph, agents can migrate over an edge from one vertex to another.

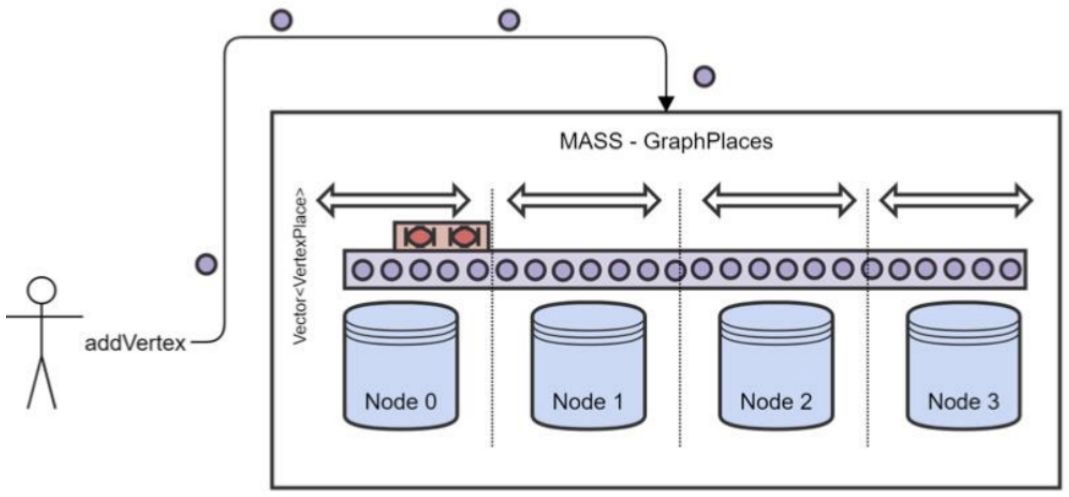


Figure 4: Vertex container and distribution [3]

MASS Basic Library	MASS Graphic Library
`Places`	⇦ `GraphPlaces`
Contains ⇩	*Extends* *Contains* ⇩
`Place`	⇦ `VertexPlace`
`Agents`	
Contains ⇩	*Extends*
`Agent`	⇦ `GraphAgent`

Table 1:Relationship between MASS Java classes

3.3. Data Format

On a limited dataset with only eight nodes, as shown in Figure 5 and Table 2 ,we diligently carried out a number of preliminary benchmarks and query verification methods early in our evaluation process. We were able to establish a strong basis for evaluating the system's performance and verifying the accuracy and dependability of query processing thanks to this methodical methodology. We made sure the system could give accurate and consistent results by putting it through these first tests, creating a solid starting point for further investigation.

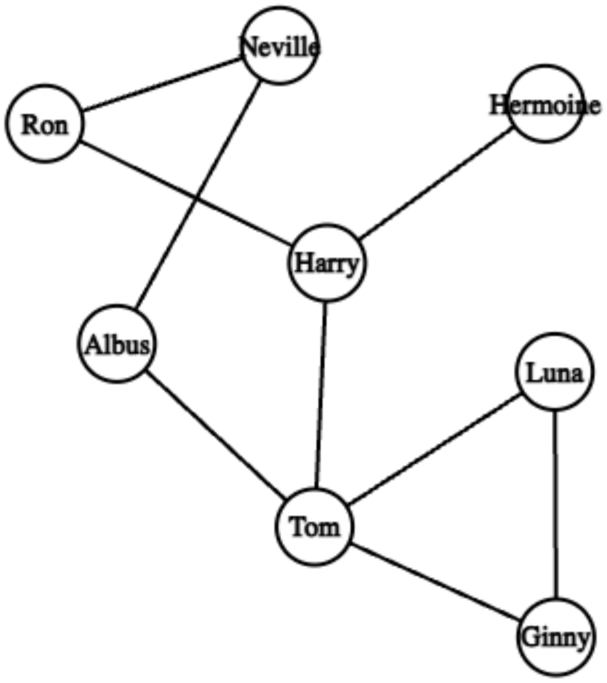
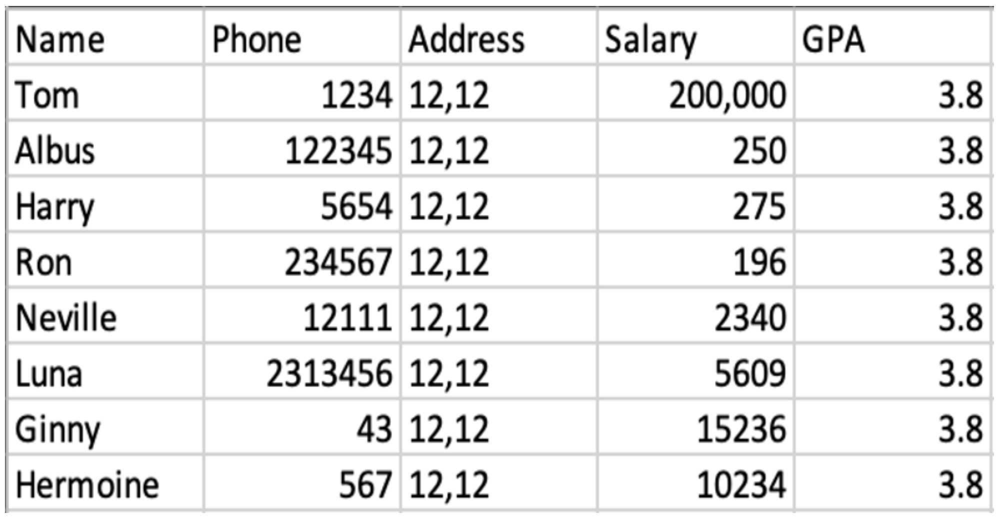


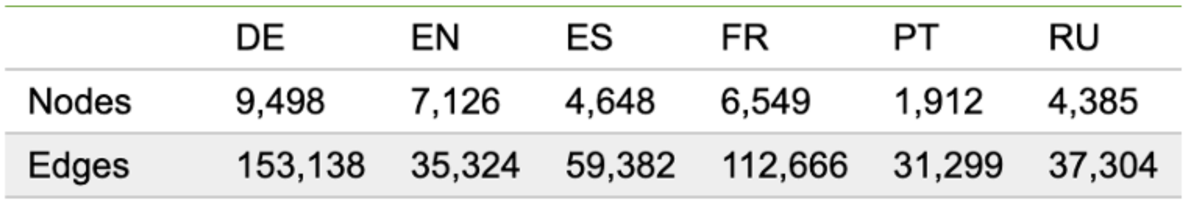
Figure 5: Preliminary benchmark data



Name	Phone	Address	Salary	GPA
Tom	1234	12,12	200,000	3.8
Albus	122345	12,12	250	3.8
Harry	5654	12,12	275	3.8
Ron	234567	12,12	196	3.8
Neville	12111	12,12	2340	3.8
Luna	2313456	12,12	5609	3.8
Ginny	43	12,12	15236	3.8
Hermoine	567	12,12	10234	3.8

Table 2: Preliminary Benchmark data

Further, we decided to take advantage of the vast resources made available by the well-known Stanford Large Network Dataset Collection, more often referred to as SNAP [10], in order to conduct a more thorough and rigorous evaluation of the system's performance. This priceless collection includes a huge variety of datasets, each of which has distinctive dimensions, complexities, and application to the actual world. We plan to reproduce and simulate numerous real-world scenarios using suitable datasets from the SNAP collection. This will allow us to fully explore the system's scalability, efficiency, and general efficacy in managing various and difficult data environments.



	DE	EN	ES	FR	PT	RU
Nodes	9,498	7,126	4,648	6,549	1,912	4,385
Edges	153,138	35,324	59,382	112,666	31,299	37,304

Table 3: Twitch dataset for users in different languages

The SNAP datasets offer an unrivaled chance to put the system through rigorous stress tests and performance analysis because they have been extensively curated and widely used by researchers and practitioners in the field. We can learn more about the behavior, resource usage, and flexibility of the system by investigating datasets with a range of sizes, complexities, and topologies. For this project we have taken a Twitch dataset, as shown in Table 3, which depicts a social-network graph containing details of user-to-user relationship in different languages. The first row of Table 3 depicts the language, and row 2 and 3 shows the number of vertices and edges in each graph.

3.4. System Architecture

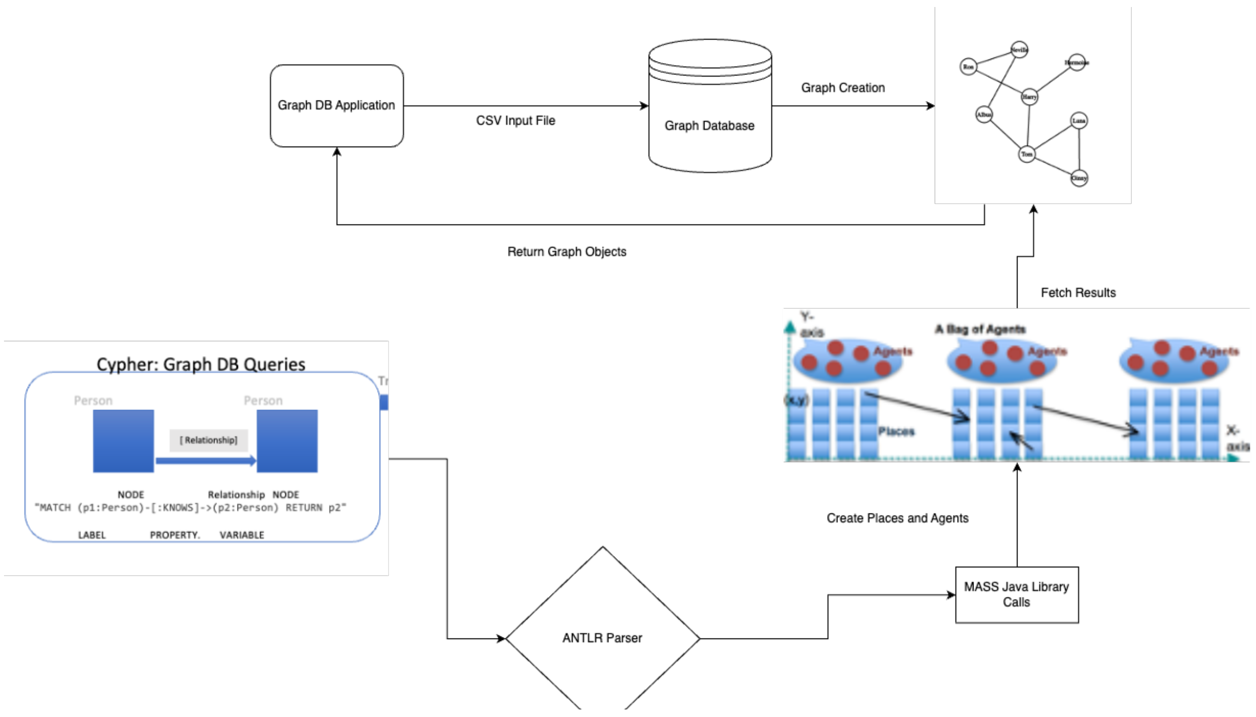


Figure 6: Architecture of an Agent Based Graph Database

The architecture of the graph database is divided into 2 sub-systems. The first system reads the CSV files and parses them into graph node objects and corresponding edges. This system relies heavily on the OpenCSV[11] library to read and understand the set of CSV files given.

The second sub system is responsible for reading and understanding the Cypher queries sent to it. This system relies on ANTLR parser to determine which place and agent to invoke to get the query results.

4. Implementation

This section discusses the design and implementation details of the generic graph parser, the six type of queries (see Table 4) implemented as part of the completion of this project and the utilization of a grammar parser tool to create a cypher query parser engine.

```
1. "MATCH (p:Person) WHERE p.Name = \"Tom\″ RETURN p"
2. “MATCH (p:Person) RETURN min(p.Salary)″
3. “MATCH (p:Person) RETURN max(p.Salary)″
4. “MATCH (p:Person) RETURN avg(p.Age)″
5. "MATCH (p1:Person)-[:KNOWS]->(p2:Person) WHERE p1.name = \"Tom\″
   RETURN p2"
6. "MATCH (p1:Person)-[:KNOWS]->(p2:Person)[:KNOWS]->(p3:Person)  WHERE
   p1.name = \"Tom\″ RETURN p3"
```

Table 4: Example list of all the types of queries implemented.

4.1. ANTLR and OpenCypher

OpenCypher[12] is the most widely adopted, fully-specified, and open query language for property graph databases that was introduced by Neo4j in 2015. Since its inception, opencypher has been adopted by other graph database vendors including RedisGraph, AgnesGraph etc.

OpenCypher provides a unified and standardized syntax for querying and manipulating graph data irrespective of the data engine working behind it. It is highly derived from syntax similar to SQL and has in-built support for all varieties of graph operations, such as creating and modifying graph nodes, creating and finding relationships, searching, aggregating and filtering results. The most unique selling point of OpenCypher is that the design is meant to be extensible, which means that anyone can utilize the language in any form and any operations and it will still maintain the compatibility with the predefined functions.

In order to read and understand an OpenCypher query, we need a mechanism to break it down into legible values that our agent-based database understands. We then try to understand what each token signifies and to get what each query means. In order to help understand each query and its grammar, we have utilized ANTLR[13] (Another Tool For Language Recognition) as our query analyzer. Among other different options like JFlex[14], JavaCC[15], Coco/R[16] we chose to go with ANTLR for two reasons. The first one is that ANTLR is intuitive and it has very streamlined integration with all the popular languages like C++ and Python, making it
ANTLR is a powerful parser generator that uses a set of predefined rules in order to parse and interpret any given text and derive legible context from it. ANTLR is widely used in generating parsers, interpreters, compilers and even for generating codes.

ANTLR generates a parser that is based on a formal grammar specification provided to the tool. It also provides the functionality to create custom grammar files to create custom parsers. The generated parser uses the input provided to it, and then creates an abstract syntax tree. In the syntax tree, there contains tokens, which are texts derived from the inputs. These tokens can be further processed to generate code, execute commands and perform other varieties of actions.
In order to parse an OpenCypher query, we have used an open source grammar file[17] that contains rules to parse all supported queries.

Upon compilation, the parser returns a enter and exit rule which can be overridden. We will override each of these methods as and when we need to implement any given query.

We have taken *MATCH* type query to start off our query parser engine. Among several available queries we chose to start with a *MATCH* type query, and the reason behind it was that most of our information retrieval for a graph database uses *MATCH*. We decided it would be most feasible to begin with this as it would provide a very strong foundation for implementation of other query types.

For the given query, ***`“MATCH (tom:Person {name: ‘Tom’})”`*** we will discuss a step by step algorithm that will help us implement this query in our Agent Based Graph Database.

In this case, we have a *MATCH* type query with a *where clause* provided to us. In order to parse this Match query we will perform the following steps:

1. Tokenize the query. Tokenization process uses a CommonTokenStream class to separate the keywords in the query.
2. Create a parse tree based on the key words. Antlr uses ParseTree class to create the tree nodes. Figure 7 depicts the resultant tree.
3. Create a Listener to walk through the tree nodes created.
4. Override the MatchOC rule method and extract the required information.

    For our case, we need to understand that the query is on a node of the type Person and we need to understand that the attribute key is ‘name’ and the attribute value in that case is ‘Tom’.

Figure 7 depicts the results when a Match query is provided as an input.

5. With this information we can determine the required GraphPlaces methods and use them to process the information.

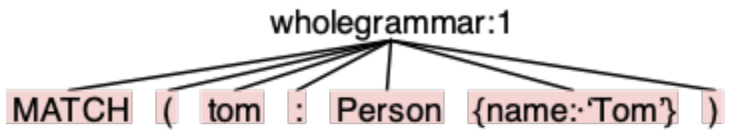


Figure 7: Parsed Tree for Match Query

4.2. CSV Parser

In previous iterations of the MASS library, there had been implementations to create a graph structure using different kinds of source files, namely HIPEE and MATSIM [9]**.** Both of these data formats for graphs have a wide range of usefulness and implementations in graph computing. However, these data formats do not provide us with the ability to fully utilize the power of Open Cypher graph queries. There are two major reasons for that. First, the data format for each HIPEE and MATSIM has to follow a strict standard, which takes away the ability for our graph node to have different identifiable attributes. Second, we cannot perform aggregates and variable relationships between the graph nodes. In order to query a graph database, we needed to implement a graph structure that can have multiple fields and each of those fields could be queried.

To overcome these two challenges, we came up with the idea of creating a GenericGraph class, which is an extension to the GraphPlaces (as mentioned in ch3.2), that can take input in the form of a generic CSV file. The file will contain information for the nodes, where each row would represent the attributes related to the node. For example, if the graph is for a group of people on a social networking website, the attributes can be name, address, etc.
The GenericGraph class has two members, Generic Vertex which is an expansion of VertexPlace class and consists of properties and attributes relating to a node, and GenericEdge class, which stores all the properties of an edge, such as weight and relationship.

The parser uses OpenCSV library to read and parse the csv file. The code snippet in Listing 1 provides information on reading a csv file and creation of MASS Graph Structure.

The loadFromFile function accepts the path of the graph datafiles as an input and calls the populate_graph function. The populate_graph uses OpenCsv library to open and read the contents of the file.

```
public void populate_graph(String nodes, String edges){
   HashMap<String, String> nodeNameId = new HashMap<>();
   try (CSVReader csvReader = new CSVReader(new FileReader(nodes));) {


       String[] headers = null;
       Object[] values = null;


       headers = csvReader.readNext();





       int vertex_id = 0;
       while ((values = csvReader.readNext()) != null) {


try (CSVReader csvReader = new CSVReader(new FileReader(edges));) {
            String[] headers = null;
            Object[] values = null;
            headers = csvReader.readNext();
            while ((values = csvReader.readNext()) != null) {
                if (values.length < 2){
                    continue;
                }
22                  int source =
23  Integer.parseInt(nodeNameId.get(values[0]));
24                  int dest = Integer.parseInt(nodeNameId.get(values[1]));
25                  cacheEdge(source, dest, 0);    }
```

Listing 1: Code Snipper for CSV file parser

The populate_graph() method from Listing 1 takes a node attribute file, and an edge attribute file as an input. It then starts parsing each line of the attribute file and adds it to GenericVertex objects. GenericVertex class has a properties attribute which is a key value pair map, which is defined in order to make the Graph node attributes as general as possible. It also creates an internal map of vertex id and node primary attribute and is cached into the MASS memory. These cached nodes are used for accessing the data on remote nodes. Lines 33 through 52 in Listing 1 perform the similar file read operations on the file containing edge related information.

4.3. Where Clause Query

A where clause in an opencypher query draws similarity with the one in any SQL-based query. When we have a group of nodes in a graph, where each node has some defining attributes, we employ a where clause query to find a certain node with a certain unique attribute. As in the case of query1, we want to find the node, if it exists, where the name is Tom.

Query 1:

```
"MATCH (p:Person WHERE p.Name = \"Tom\" RETURN p"
```

A MATCH query with the WHERE clause is implemented by using the functionality of Places and their interaction with each of the graph nodes. The way GraphPlaces[5] is constructed is that each graph node resides on a Place. So when we implement an OpenCypher query that requires us to find a node where a particular property matches a required value, we can directly do a Places.callAll() and provide the value as an argument. When CallAll() happens, each of the Place executes the method simultaneously and there we have overridden the business logic to return the Place that matches our argument query. Listing 2 provides a detail of the code written to execute the where query. Lines 4 to 10 perform a string break down of the argument which contains the information of the where clause parameter. Lines 11 and 12 gathers information from the given place and then does a string match to return the result.

```
String whereVar = ctx.oC_Where().oC_Expression().getText();
String patternVar = ctx.oC_Pattern().getText();
network.callAll(GraphDBNode.computeMatch, (Object) whereVar);
public Object computeMatchQuery(Object argument){
       String args = (String) argument;
       String[] fromArgs = args.split("=");
       String property = fromArgs[0];
       String value = fromArgs[1];
       String[] property2 = property.split("\\.");
       String attr = (String)
this.properties.get(property2[1].trim());
       if (value1.equals(attr)){
               return attr;
       }
       return attr;
   }
```

Listing 2: Code Snippet for Processing Graph Queries

4.4. Aggregate Queries

An Aggregate type of query is very similar to general aggregations that we do in statistical analysis of any given data. When we have a group of nodes in a graph, where each node has a set of defining attributes, we employ the use of aggregate queries when we want to fetch the results and analyze the graph from a top-view point in frame.

As shown in Query 2 and Query 3, where we are trying to find the node with the minimum and maximum salary respectively. Apart from min and max, we have also implemented queries that will calculate the count and average of the given attributes.

Query 2:

```
"MATCH (p:Person) RETURN min(p.Salary)"
```

Query 3:

```
"MATCH (p:Person) RETURN max(p.Salary)"
```

To implement aggregate type of queries, we take into consideration two important factors. First, since the aggregation we perform using the queries are on the properties of the graph node and since each graph node is located on each of the places, we can make use of the CallAll method for each place in order to find the given aggregation. Second, since we can only get the attribute value at each place, we also need to get the attribute from each node and then aggregate it manually at the business layer.

In Listing 3, we are using the query rule from the Cypher grammar that we have created. As all the rules we have created in the grammar are hierarchical in nature, we can parse any query through multiple rules. The parser recognizes all types of queries under the first rule of the parser hierarchy, that is the *OC_query* context rule. We then determine what the query wants to return by fetching the node that contains the return variable. The next step is to identify the aggregator for the query as well as the property at which the aggregation needs to be performed, for which we use basic substring matching available for us.

```
@Override
     public void enterOC_Query(CypherParser.OC_QueryContext ctx) {
     CypherParser.OC_RegularQueryContext c = ctx.oC_RegularQuery();


     String returnVariable =
c.oC_SingleQuery().oC_SinglePartQuery().oC_Return().oC_ProjectionBody()
.getText();
     if(returnVariable.contains(MIN) || returnVariable.contains(MAX) ||
returnVariable.contains(AVG) || returnVariable.contains(SUM)){
          isMatch = 1;
            String[] fromArgs = returnVariable.split("\\(");
          String aggregator = fromArgs[0];
          Object property =
(Object)fromArgs[1].split("\\.")[1].replace(")", "");
          Oject[] res = network.callAll(GraphDBNode.computeAgg, args);
```

Listing 3: Code snippet for parsing the aggregate type queries

In the implementation shown in Listing 3, we do a single level aggregation for the given attribute on each place individually, and then send that data to the main program for the place level aggregation. Another way to possibly do this would be to use the agents and migrate agents to each place and let agents do the aggregation and return. Both solutions are very network heavy and can cause congestion in the network with a lot of message exchanges happening together. This creates a huge strain on the network as each place will send its data back to the main node. To solve this problem, we have come up with a two-level aggregation method by using the help of lambdas.

The lambda (see Listing 4) is implemented with a strategy to reduce the number of messages that are sent to the main node. We have implemented a Lambda class that extends GraphPlaces and each lambda instance has access to a set of local Vertex place objects, which contains all the information regarding the attributes of the given graph node. We then iterate over these local places and aggregate the results for the given set. Each of these instances then sends the result back to the main function. So now, instead of each place sending their local results to the main node, we have a smaller set of results on which we can then perform our next level aggregation.

```
public class Lambdas extends GraphPlaces {
   @Override
   public Object[] callAll(int functionId, Object[] argument) {

       String property = (String) argument[0];
```

```
        String aggregator = (String) argument[1];

        HashMap<String, Object> attributeValues = new HashMap<>();
        for (int i = 0; i < g.getGraphPlaces().size(); i++) {
            switch (aggregator) {
                case "min"
                    int mn_val = Integer.MAX_VALUE;
                    for (String k : attributeValues.keySet()) {
  int value = Integer.parseInt((String) attributeValues.get(k));
                    if (value < mn_val) {
                        mn_val = value;
                        u = k;
                    }
            }
            val = (Object) mn_val;
            user = (Object) u;
            break;

            case "max":
                int mx_val = Integer.MIN_VALUE;
                for (String k : attributeValues.keySet()) {
   int value = Integer.parseInt((String) attributeValues.get(k));
```

```
28                  if (value > mx_val) {
29                      mx_val = value;
30                      u = k;
31                  }
31              }
32              val = (Object) mx_val;
33              user = (Object) u;
34              break;
35          Object[] res = {user, val};
36          return res;
37      }
```
Listing 4: Code snippet for Lambdas implementation

Listing 4 shows the implementation of Lambda class. The class extends to GraphPlaces and each instance of Lambda would have access to all the locally created places for the current compute node. From line 7 to 28 we iterate over the local places and perform a reduction based on the arguments received from the callAll method dispatch.

4.5. Relationship Queries

A relationship query is very unique to a Graph database aspect. When we have a group of nodes in a graph, where each node has some defining attributes, and each node is connected to one or more nodes within that graph. We employ the relationship query when we try to find connections between the nodes based on the relationship they share. If the two nodes are people, as given the Query 4, we can find the nodes that "Tom" knows within the Graph. Another example of a simple use-case would be, if there are two types of nodes in the graph, actors and movies, and we want to determine all the movies the particular actor has been a part of. In this case, the relationship would be "*ACTED*" instead of "*KNOWS"*.

Query 4:

```
"MATCH (p1:Person)-[:KNOWS]->(p2:Person) WHERE p1.name = \"Tom\″ RETURN
p2"
```

Any relationship query is defined in 2 sections, the first section determines how to reach the node, which in our case is by specifying a "Knows" relationship. This idea can be explored into different nodes having different relationships between them. For example, we can have a node "person" and a node "city", where the relationship between the two is that the *person* "lives" in the *city*

The second section of the query to find the node with which we are searching for a relationship, which essentially means a starting node for our search within the graph.

In Query 4, we can determine the starting node p1 by utilizing the where conditional parsing we have implemented in our application. For getting the relationship, we use the OpenCypher Grammar method to fetch the exact relationship.

After we have determined the relationship and the starting node, we employ the use of Agent based migration to perform a breadth first search to fetch all the neighboring nodes and return the nodes where the relationship matches with the required one.

```
if ( smartPlace.footprint == -1 ) {
                smartPlace.footprint = 1; //This place has been visited
                kill( );
            }
            // Set my next node before spawning children.
            nextNode =
                 ( neighbors[0] != prevNode ) ? neighbors[0] :
                   neighbors[1];
            migrate( nextNode ); //Migrate to the next Node
            SmartArgs2Agents[] args
                    = new SmartArgs2Agents[( getAgentId( ) == 0 && getPlace(
).getIndex( )[0] == 0 && ( ( SmartPlace )getPlace( ) ).footprint == -1 ) ?
                    neighbors.length - 1:
                    neighbors.length - 2];
            for ( int i = 0, j = 0; i < neighbors.length; i++ ) {
                if ( neighbors[i] == nextNode || neighbors[i] == prevNode )
// skip the parent's next node or previous node
                    continue;
                args[j++] = new SmartArgs2Agents( neighbors[i],getPlace(
).getIndex( )[0]);
21              }
22              spawn( args.length, args );
23              //Before the Agent moves or Spawns set the prevNode
24              prevNode = getPlace( ).getIndex( )[0];
25              smartPlace.footprint = 1; //This place has been visited
26          }     level--;
```

Listing 5: Code Snippet for multi-level relation migration

Listing 5 depicts the implementation of a multi level agent migration for a given graph. Lines 1 to 4 perform a check that current vertex has been visited. Lines 5 to 20, perform the following two things. First, they determine all the neighbors of the current vertex and then create a set of migration to a set of nodes, and spawn child agents to traverse all the other neighbors.

5. Evaluation

This chapter presents experimental results on the MASS implementation of the graph database for the following queries.

```
"MATCH (p:Person) WHERE p.Name = \"Tom\" RETURN p"

"MATCH (p:Person) RETURN max(p.Salary)"
```

```
"MATCH (p1:Person)-[:KNOWS]->(p2:Person) WHERE p1.name = \"Tom\″ RETURN
   p2"
"MATCH(p1:Person)-[:KNOWS]->(p2:Person)-[:KNOWS]->(p3:Person)WHERE
   p1.name = \"Tom\″ RETURN p3"
```

Table 5: List of evaluated queries

5.1. Environment Set-up

The applications were executed on the HERMES cluster and the CSSMPI cluster at the University of Washington Bothell. Those two clusters have 24 computing nodes in total. The detailed information about the computing nodes is shown in Table 4.1. When executing applications, the first 12 computing nodes are from the HERMES cluster, and the rest 12 computing nodes are from the CSSMPI cluster.

# Computing Nodes	# Logical CPU Cores	CPU Model	Memory	Cluster
3	4	Intel Xeon 5150 @ 2.66 GHz	16 GB	HERMES
4	8	Intel Xeon E5410 @ 2.33 GHz	16 GB	HERMES
5	4	Intel Xeon Gold 5220R @ 2.20 GHz	16 GB	HERMES
12	4	Intel Xeon Gold 6130 @ 2.10 GHz	16 GB	CSSMPI

Table 6: Execution environments of HERMES and CSSPMI clusters

5.2. Evaluation Scenario

We have established a comprehensive set of three distinct metrics that serve as evaluation criteria for our study. These metrics enable us to thoroughly assess the performance and effectiveness of our proposed solution. The first metric, known as Query Verification, focuses on validating the accuracy and correctness of queries processed by our system. We meticulously analyze and verify the results obtained from these queries to ensure their reliability.

The second metric we employ is Query Round Trip Time, which encompasses the entire journey of queries within our system. We collect an extensive range of data values for this metric, specifically targeting a dataset comprising 7,000 nodes and 35,000 edges. By measuring the time it takes for queries to travel through the system and return with results, we gain insights into the efficiency and responsiveness of our solution.

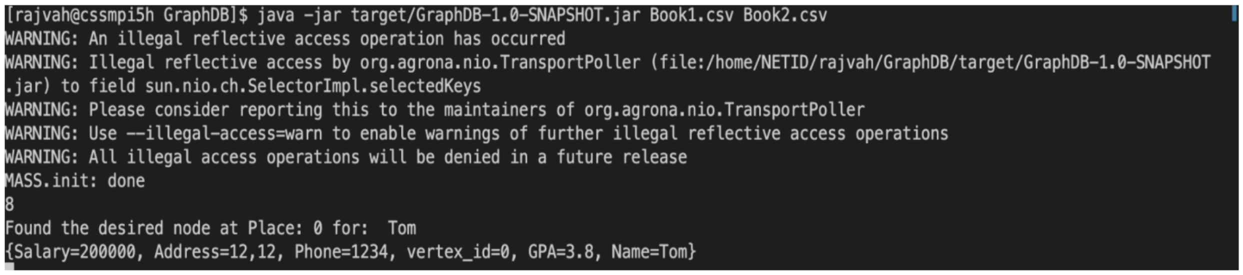
Lastly, we evaluate the spatial scalability of our system through the Spatial Scalability metric. We meticulously assess the system's performance when operating on different computing nodes, specifically focusing on a range of 1,900 to 9,500 nodes. This analysis allows us to gauge how well our solution adapts and scales in response to varying computational resources, providing valuable insights into its potential to handle larger datasets and workloads.

By employing these three metrics in our evaluation process, we can thoroughly scrutinize the capabilities and performance of our proposed solution, ensuring its effectiveness, reliability, and scalability in real-world scenarios.

5.2.1. Verification of Query

Figure 8 depicts the result for query 1, (ie., a WHERE clause query), where we are trying to fetch the node where name equals the value, Tom. Based on the RETURN type, the graph database can return the entire node, or certain attributes.

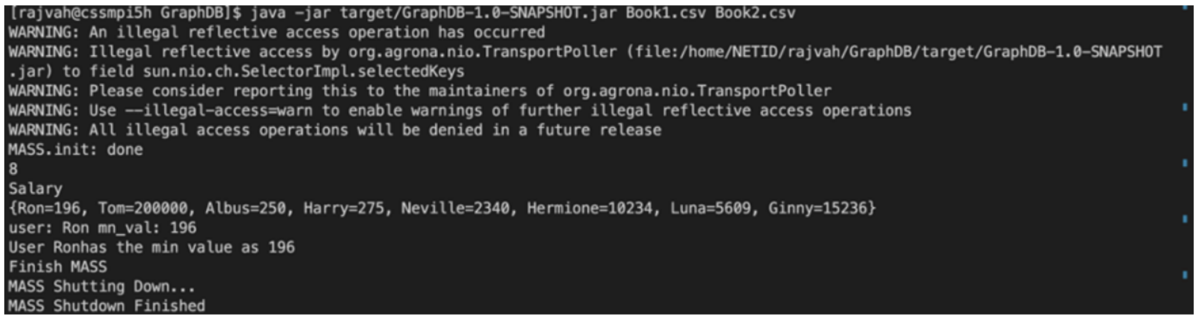
We can return the node and print all the properties that the node has.



```
[rajvah@cssmpi5h GraphDB]$ java -jar target/GraphDB-1.0-SNAPSHOT.jar Book1.csv Book2.csv
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller (file:/home/NETID/rajvah/GraphDB/target/GraphDB-1.0-SNAPSHOT
.jar) to field sun.nio.ch.SelectorImpl.selectedKeys
WARNING: Please consider reporting this to the maintainers of org.agrona.nio.TransportPoller
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
MASS.init: done
8
Found the desired node at Place: 0 for:  Tom
{Salary=200000, Address=12,12, Phone=1234, vertex_id=0, GPA=3.8, Name=Tom}
```

Figure 8:Verification Results of Where Clause

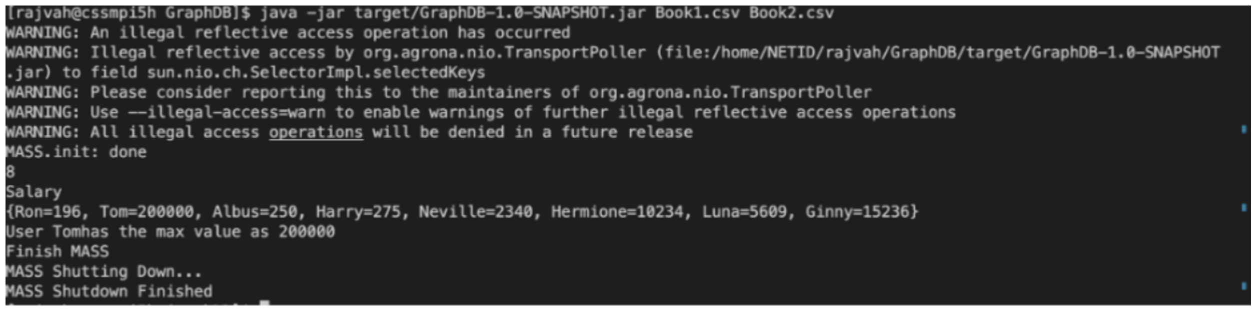
The Figure 9 is from query 2, where we are trying to find the minimum "Salary" for the given set of nodes. For the sake of convenience, I also printed out the salaries of each of the "*Person"* that we have in our graph.



```
[rajvah@cssmpi5h GraphDB]$ java -jar target/GraphDB-1.0-SNAPSHOT.jar Book1.csv Book2.csv
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller (file:/home/NETID/rajvah/GraphDB/target/GraphDB-1.0-SNAPSHOT
.jar) to field sun.nio.ch.SelectorImpl.selectedKeys
WARNING: Please consider reporting this to the maintainers of org.agrona.nio.TransportPoller
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
MASS.init: done
8
Salary
{Ron=196, Tom=200000, Albus=250, Harry=275, Neville=2340, Hermione=10234, Luna=5609, Ginny=15236}
user: Ron mn_val: 196
User Ronhas the min value as 196
Finish MASS
MASS Shutting Down...
MASS Shutdown Finished
```

Figure 9: Verification of Aggregate for minimum

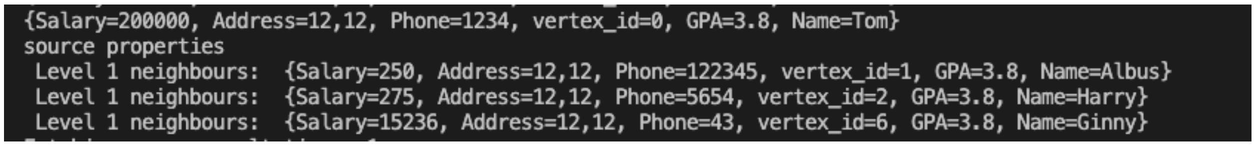
The Figure 10 depicts the results for query 3, where we wanted to get the maximum of an attribute and return the resulting node. We can perform these types of aggregations on any of the properties that the node possesses. However, the application will throw an exception if the property is not of the type where aggregation can happen, for example "address".



```
[rajvah@cssmpi5h GraphDB]$ java -jar target/GraphDB-1.0-SNAPSHOT.jar Book1.csv Book2.csv
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller (file:/home/NETID/rajvah/GraphDB/target/GraphDB-1.0-SNAPSHOT
.jar) to field sun.nio.ch.SelectorImpl.selectedKeys
WARNING: Please consider reporting this to the maintainers of org.agrona.nio.TransportPoller
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
MASS.init: done
8
Salary
{Ron=196, Tom=200000, Albus=250, Harry=275, Neville=2340, Hermione=10234, Luna=5609, Ginny=15236}
User Tomhas the max value as 200000
Finish MASS
MASS Shutting Down...
MASS Shutdown Finished
```

Figure 10: Verification of Aggregate for maximum

The Figure 11 depicts the results for query 3, where we wanted to get the maximum of an attribute and return the resulting node. We can perform these types of aggregations on any of the properties that the node possesses. However, the application will throw an exception if the property is not of the type where aggregation can happen, for example “address”.



```
{Salary=200000, Address=12,12, Phone=1234, vertex_id=0, GPA=3.8, Name=Tom}
source properties
 Level 1 neighbours:  {Salary=250, Address=12,12, Phone=122345, vertex_id=1, GPA=3.8, Name=Albus}
 Level 1 neighbours:  {Salary=275, Address=12,12, Phone=5654, vertex_id=2, GPA=3.8, Name=Harry}
 Level 1 neighbours:  {Salary=15236, Address=12,12, Phone=43, vertex_id=6, GPA=3.8, Name=Ginny}
```

Figure 11: Verification of Relationship query

5.2.2. Spatial Scalability

Given that MASS Places operates on a highly efficient distributed memory model, the process of expanding the number of graph nodes within the system becomes synonymous with the act of augmenting the quantity of places. This seamless expansion, in turn, leads to a direct augmentation in the accessibility of each individual place for the agent, ensuring that the time required to access any given place remains constant and unchanged.

In order to evaluate the impact of dataset size on performance, a notable experiment was conducted wherein the dataset was incrementally expanded from 1,000 nodes to a substantially larger scale of 9,000 nodes. The dataset used for this purpose was sourced from the well-regarded SNAP datasets. Remarkably, this substantial increase in dataset size had little to no discernible effect on the overall performance of the system, further attesting to the robustness and scalability of MASS Places.

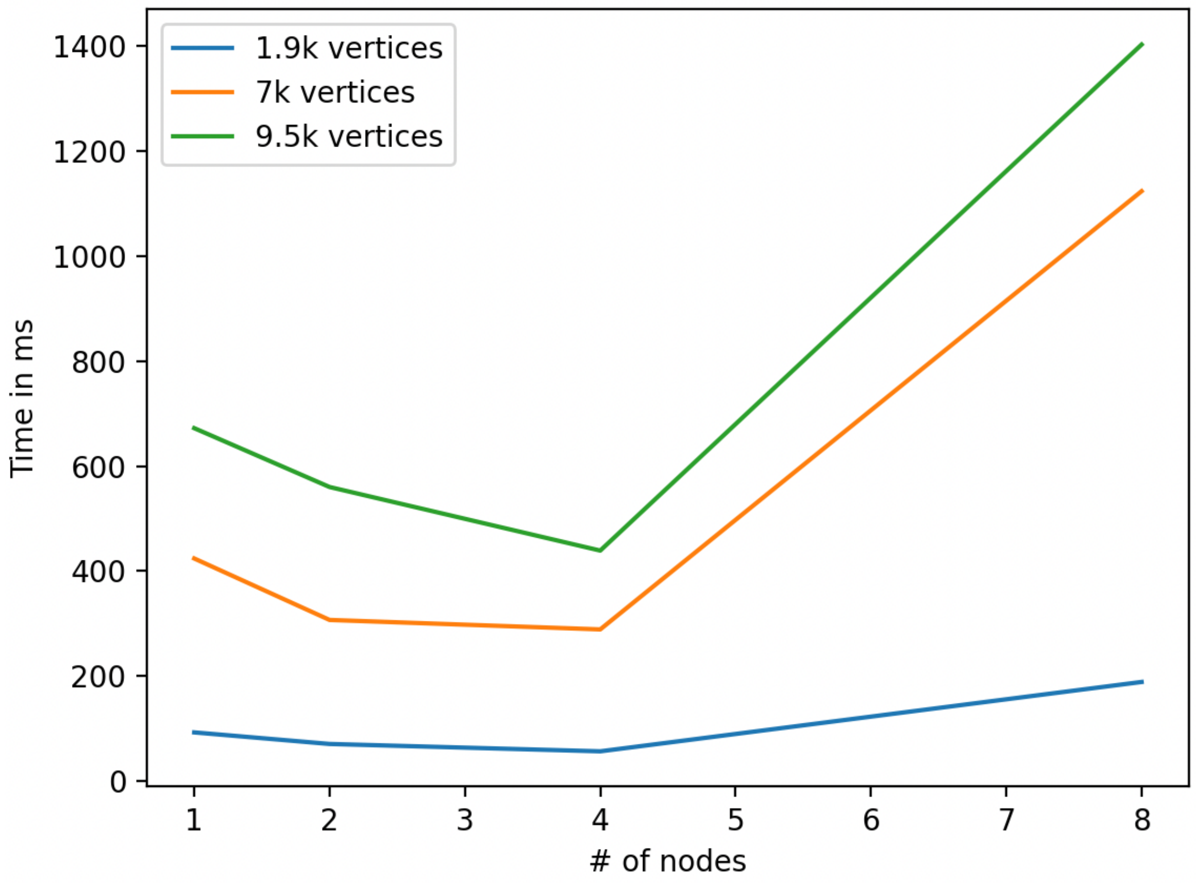


Figure 12: Query time for different data files

5.2.3. Query Round Trip Time

Figure 13 through 18 provide a graphical representation of the time taken for graph creation and time taken by each query as we increase the number of computing nodes. All these query results are performed on a social networking twitch dataset provided by SNAP [12], a Stanford open source group that generates and provides graphical data for solving different types of problems and performing analysis.

From the Figure 13 we can observe that graph creation is a linearly increasing graph. The reason behind this is the creation of a graph is a sequential process and increasing the number of vertices and edges will cause the time taken to increase. In addition to that, as we increase the number of compute nodes, the communication overhead increases as well.

A solution to fixing these time creations is to use Graph Streaming in MASS [13] that was implemented by Yan. The implementation divides the graphs into smaller subgraphs and performs the creation of vertices and edges in parallel. This would significantly improve the results for large datasets.

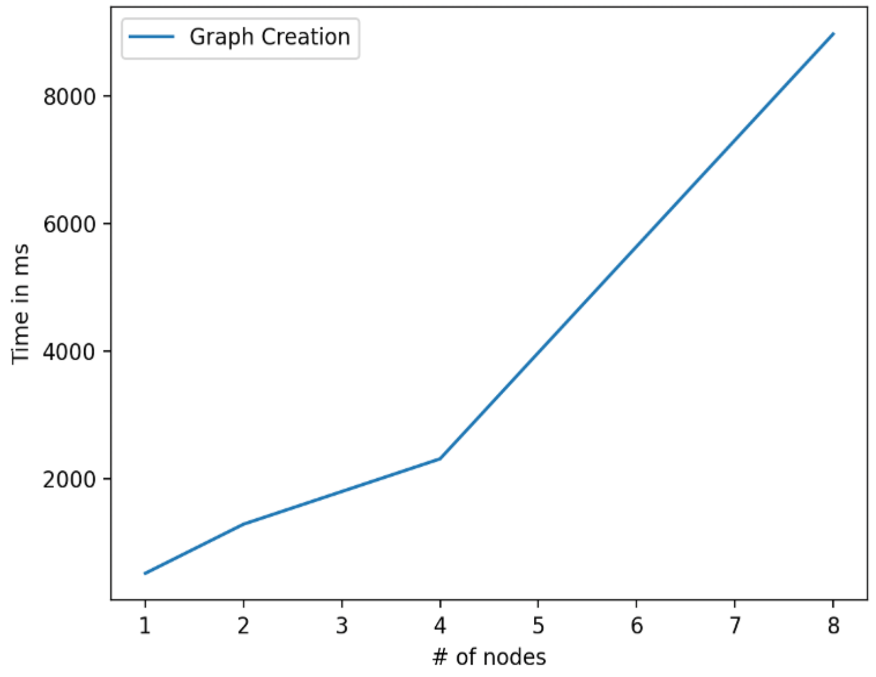


Figure 13: Query Round trip time for Twitch Dataset

Figure 14 through 18 follow the same trend as we increase the number of computing nodes. We observe a significant improvement in the query return time up until we increase the number of compute nodes to four. The reason is that with two or four computing nodes, the number of places and agents within these places the computation happens in parallel and with the number of compute nodes, the communication overhead is not significant enough to cause any slowdowns. However, with the number of compute nodes increased to eight, the number of messages passed between the nodes increases as the total number of vertices stored on each computing node decreases, which also results in a decrease in number of agents.

Therefore, for each agent migration between places would require communication over the network and thus increase the communication overhead by manifolds.

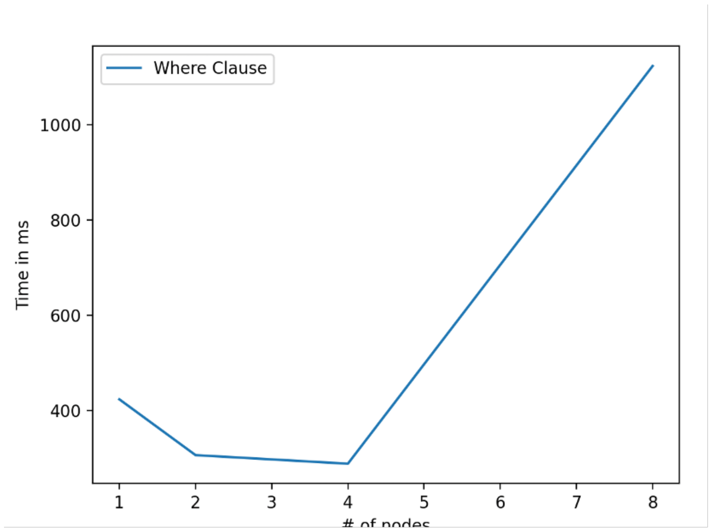


Figure 14:Query Round trip time for Twitch Dataset

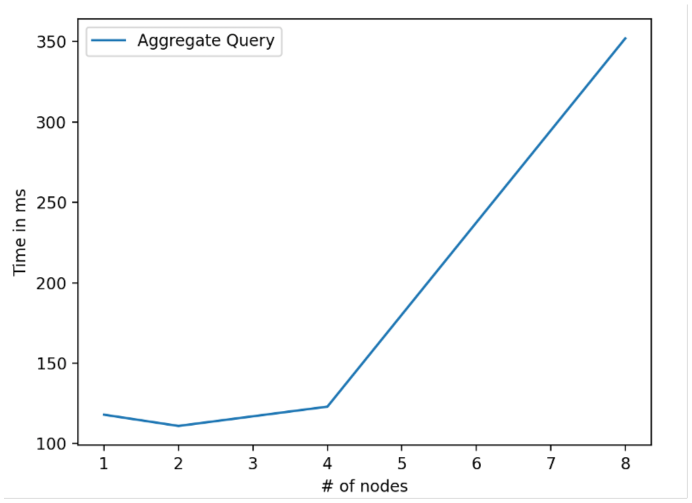


Figure 15:Query Round trip time for Twitch Dataset

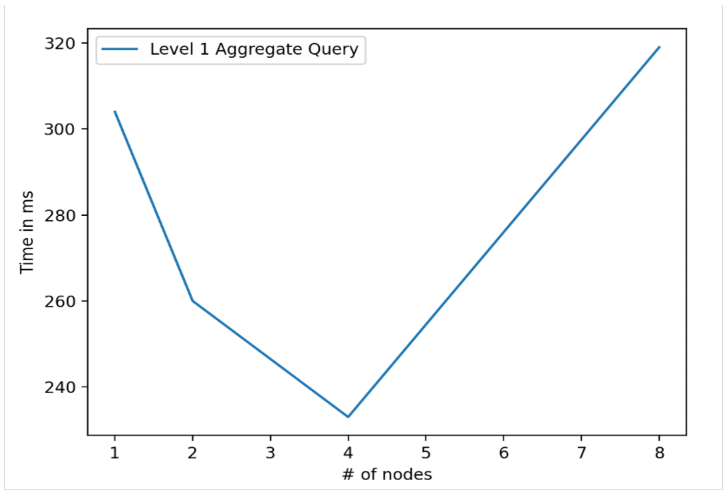


Figure 16: Query Round trip time for Twitch Dataset

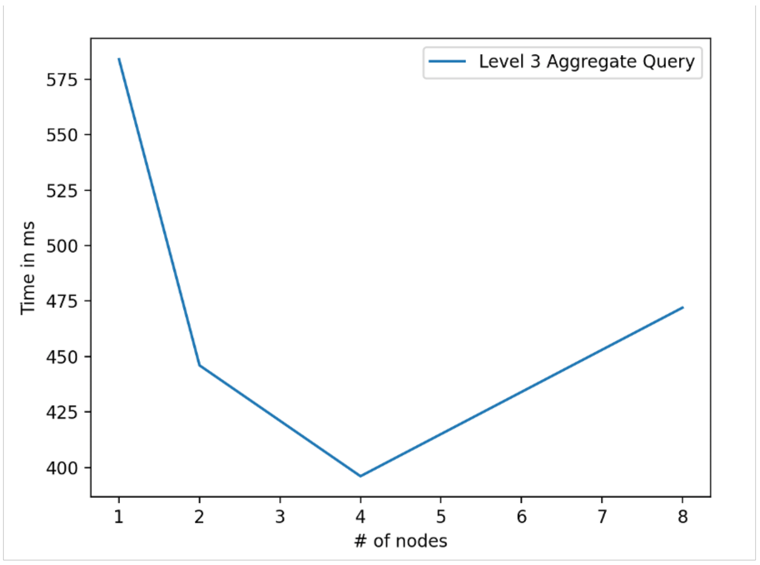


Figure 17: Query Round trip time for Twitch Dataset

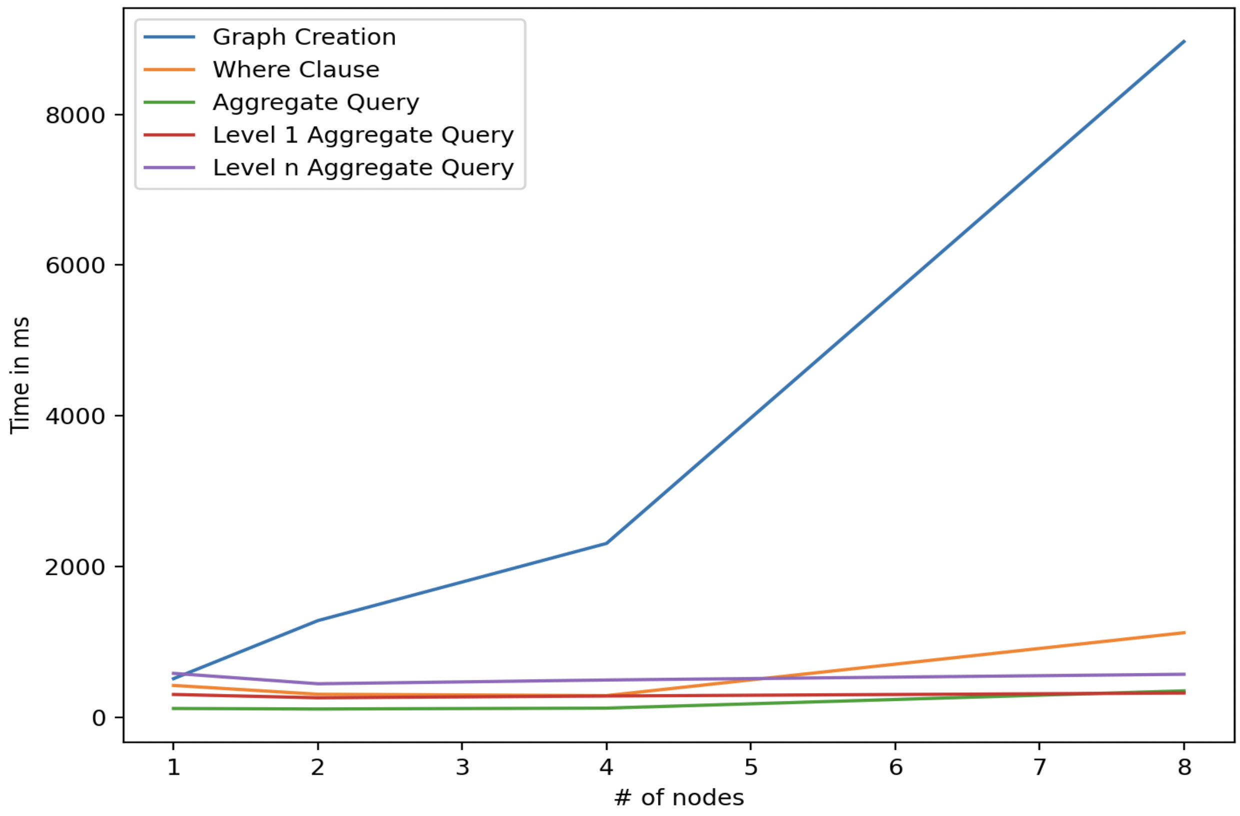


Figure 18: Query Round trip time for Twitch Dataset with 9.5k nodes

6. Conclusion

This project completed the implementation for creating a CSV data file parser, an opencypher query parser, and successful implementations of where type graph query, aggregate type graph queries, and relationship type graph queries. We also performed the scalability and performance testing using 1, 2, 4 and 8 compute nodes.

The execution results show that for both small and large datasets, our Agent based graph database scales horizontally without having an impact on performance. This project also verifies the correctness of a graph query for both small and large datasets.

For future works, several improvements can be considered to enhance the existing queries and overall functionality of the system:

1. Generalization of "where" queries: Introduce the capability to encompass a broader range of query requirements by incorporating comparators that can return not only a single node but also a comprehensive list of nodes. This expansion would enable users to specify queries that retrieve a range of graph nodes based on specific criteria.
2. Augmentation of aggregate queries: Enhance the scope and versatility of aggregate queries by incorporating additional functionalities. This expansion would empower users to perform more complex aggregations, enabling them to gain deeper insights and derive more meaningful results from their data.

3. Extension of CRUD operations: Extend the capabilities of the system by incorporating Create and Update operations through Cypher queries. This addition would facilitate seamless data management within the graph database, allowing users to effortlessly create and modify graph nodes and edges.
4. Performance benchmarking: Establish a comprehensive benchmarking framework to assess the performance of the graph database. Conduct comparative analyses with industry-leading solutions like Neo4j and RedisGraph to gauge the system's efficiency, scalability, and effectiveness in handling large-scale datasets and complex queries.
5. Development of a Generic Parser class: Create a versatile and adaptable parser class capable of reading any data file format and converting it into corresponding objects such as GenericVertex and GenericEdge. This class would enhance the system's flexibility, allowing users to seamlessly import and process data from various sources, further expanding its applicability.
6. Exploration of diverse application domains: Conduct extensive research and experimentation in different domains, such as financial datasets or network systems. By exploring these domains, the system can be adapted and optimized to cater specifically to the unique requirements and challenges posed by different application contexts, thereby expanding its potential range of applications and use cases.
7. We are also planning on adding multi user support to MASS in order to perform transactional database queries.

Bibliography

[1] MASS: A Parallel Library for Multi-Agent Spatial Simulation, “Accessed on: May 1, 2023. [online]. available: http://depts.washington.edu/dslab/mass/.”

[2] Yuna Guo, Construction of Agent-navigable Data Structure from Input Files, 2022, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/YunaGuo_whitepaper.pdf

[3] Brian Luger, Distributed Data Structures, 2021, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/BrianLuger_su21.pdf

[4] Neo4j, “Accessed on: May 10, 2023[Online]. Available: https://neo4j.com/

[5] RedisGraph, “Accessed on: May 12, 2023. [Online]. Available: https://redis.io/docs/stack/graph/.”

[6] DSLab, "MASS Java Manual," University of Washington, Bothell, 31 Aug 2016. [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Technical%20Manual.pdf.

[7] DSLab, "MASS C++: Parallel-Computing Library for Multi-Agent Spatial Simulation," University of Washington, Bothell, [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MassCpp.pdf.

[8] DSLab, "MASS CUDA: Parallel-Computing Library for Multi-Agent Spatial Simulation," [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MassCuda.pdf.

[9] J. Gilroy, "Dynamic Graph Construction and Maintenance," DSLab, [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/JustinGilroy_whitepaper.pdf.

[10] Twitch Social Networks,“Accessed on: March 03, 2023. [Online].Available: https://snap.stanford.edu/data/twitch-social-networks.html

[11] OpenCSV,“Accessed on: May 04, 2023. [Online].Available:

https://opencsv.sourceforge.net/

[12] openCypher, “Accessed on: November 17, 2022”. [Online]. Available:

https://opencypher.org.”

[13] ANTLR, “Accessed on: January 15, 2023”.[Online].Available: https://www.antlr.org/

[14] JFlex, “Accessed on: January 15, 2023”.[Online].Available: https://www.jflex.de/

[15] JavaCC,“Accessed on: January 15, 2023” [Online].Available:

https://javacc.github.io/javacc/

[16] Coco/R, “Accessed on: January 15, 2023” [Online].Available:

https://ssw.jku.at/Research/Projects/Coco/

[17] Cypher Grammar, “Accessed on: January 20, 2023” [Online]. Available:

https://github.com/antlr/grammars-v4/tree/master/cypher