An Agent-Based Graph Database Benchmarking Program

Michelle Dea

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington, Bothell

2024

University of Washington

Abstract

An Agent-Based Graph Database Benchmarking Program

Michelle Dea

Chair of the Supervisory Committee:
Dr. Munehiro Fukuda
Computing & Software Systems

Graphs can be used to represent complex relationships between different entities. These are stored as edges and nodes in a graph database system. This type of data makes it easier and more flexible to query connected data items, and identify insights from those relationships. Two common graph database systems are Neo4j and ArangoDB. These systems store graph data differently as Neo4j is a native graph database system, and ArangoDB is a multi-model database. Both database system rely on storing data in the disk, which can be slow for data retrieval when the data is not in memory. A graph database system using the Multi-Agent Spatial Simulation (MASS) library is being implemented to pursue CPU and spatial scalability by leveraging distributed memory to store graph data. This project aims to provide a benchmarking protocol for performance testing of MASS Graph Database system compared to Neo4j and ArangoDB. This will identify the current

strengths and weaknesses of MASS, and provide a standard benchmarking tool for future researchers to use. The work includes using data pulled from real-world applications as the foundation of a random graph generator that takes user input regarding the topology and size of the graph that will be generated, a set of common queries both in Cypher and AQL, a manual for testing in Neo4j, and a script for testing in ArangoDB. The graph sizes used in testing are 1K nodes, 10K nodes, 20K nodes, and 30K nodes for spatial scalability evaluation via graph traversal. CPU scalability of MASS is performed on a cluster of eight computing nodes using a 10K node graph.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEDICATION

To Bob, thank you for everything.

Chapter 1. INTRODUCTION

1.1 MASS

The Multi-Agent Spatial Simulation (MASS) library is a parallel programming library using agent-based modeling to simulate a number of collective behaviors, ex: biological agents, and computational geometry algorithms [1]. MASS at its core is comprised of two main components, Places and Agents. Places represents a matrix of elements dynamically allocated over a cluster of nodes. These elements can exchange information with any other element, i.e. Place. Agents can perform computations, and can migrate between different Places, allowing them to interact with other Agents and Places.

When considering big data computing, frameworks such as MapReduce and Spark are some of the more common tools being used to process large volumes of data. However, these tools mainly deal with data in the form of text. For more complex data structures, such as graphs, where the data points are interconnected, a more suitable approach is to use a graph database system. Specifically with MASS, agents can be leveraged to support analysis of graphs, as they can be deployed into data structures mapped over distributed memory.

1.2 GRAPH DATABASE SYSTEM

Database systems were first introduced in the 1960s, with some systems, like IBM’s IMS, supporting tree-like structures [27]. These relational database systems were ACID (atomicity, consistency, isolation, durability) compliant, and stored data in a tabular format with rows and columns. As the size and structure of data evolved, the need for a different storage and querying solution paved the way for a new way to organize this data. In the mid 2000s, the first commercial graph database systems were made available to the public. A graph database system is a collection of data where vertices, commonly referred to as nodes, hold information related to the entity, and edges, commonly referred to as relationships, describe the connection between two nodes. This type of database system places an emphasis on the relationships between different entities. Medium sized graph datasets usually range from 1 million to 10 million graph entities (nodes and relationships), and large graphs usually range from 10 million to 100 million graph entities [8]. Two common graph database systems include Neo4j and ArangoDB.

Neo4j is one of the leaders in graph database systems, and it is a native graph database system. This popular graph database system uses Cypher as its query language, and has a storage engine that optimizes storing graph data on the disk.

ArangoDB is another popular graph database system that is a multi-model database solution, meaning it stores and queries data in more than one format. This system stores data in the form of a document, which is essentially a JSON object. These documents are then organized into collections, and can be queried using the ArangoDB Query Language (AQL). For data storage, ArangoDB uses a RocksDB storage engine where data is stored in the disk.

1.3 Graph Database System Use Cases

The primary driver behind storing data as graphs is to explore the relationship between different entities. These complex relationships are difficult to translate into standard relational databases, and the use of a graph database system allows for flexibility and efficiency when querying connected data points. There are a number of use cases for storing data in a graph database system. The most popular use cases include recommendation engines, social network analysis, and fraud detection [15,16,17,18,21,22]. Recommendation engines are used to provide recommendations, an example of this is recommending products for purchase based on previous purchasing history or products typically ordered together. Social networks typically have many interconnected entities such as friends, interests, and followers. A graph database system provides a way to store and query these intertwined relationships. Lastly, fraud detection usually refers to financial transactions. Because graph database systems allow for faster querying of connected data points, institutions are able to detect fraud through relationship patterns more quickly than if using a relational database.

1.4 GOALS

The main goal of this project is to create a benchmarking dataset and tools to simplify benchmarking the MASS Graph Database system (Graph DB system) against popular graph database systems. The motivation behind this is to create tools such that future researchers can devote their time to improving the MASS Graph DB system, rather than spending resources on investigating methods to conduct performance or execution testing. This work will also standardize how future MASS Graph DB system changes will be evaluated, establishing consistency and improving accuracy when it comes to performance testing. To further outline the specific goals for this project, here are the explicit tasks I will accomplish:

- Create a tool to randomly generate several graph datasets that can be loaded into MASS Graph DB system, Neo4j, and ArangoDB. These datasets are modeled after graphs pulled from real world systems based on the use cases mentioned above. The randomization ensures fairness in our evaluation.
- Create common queries for each of the datasets that can be run across each graph database system for measuring performance.
- Identify the strengths and weaknesses of the MASS Graph DB system compared to Neo4j and ArangoDB.

- Create an efficient process for benchmarking performance of MASS Graph DB system compared to Neo4j and ArangoDB for future researchers.

Chapter 2. BACKGROUND

The growth of big data gave rise to the development of data streaming frameworks such as Hadoop, and Spark for analysis of these larger datasets. These frameworks primarily support data in the format of text, with the approach of dividing the data into chunks and processing those individual chunks. This limits their ability to analyze data in more complex data structures, such as a graph, as the division, streaming, and reconstruction of a graph for processing can be challenging.
Instead, using a graph database system to store and query this kind of data makes both storage and graph traversal more efficient. Two popular graph database systems, Neo4j and ArangoDB, store data on the disk, with retrieval of data possibly being slow if it is not already available in memory. An alternative is to construct the graph on distributed memory, using agents to traverse the graph. The use of MASS for this approach not only stores the data in memory, allowing for faster access of the data, but also allows query processing to be done in parallel via agents.

2.1 Previous Work

The MASS Graph DB system was originally implemented by Harshit Rajvaidya [2], with enhancements being implemented this year by Lilian Cao. MASS Graph DB system stores the graph data in Place objects. More specifically, with this year's enhancements, the data is stored in PropertyVertexPlace, which is an extension of Place. The PropertyVertexPlace object has two hashmaps that store directed edge information. When querying data, agents are deployed at each node, traverse the graph by checking a hashmap of its neighbors, and return the results of the query. Previously, spatial scalability testing of MASS Graph DB system used graphs with 1.9K nodes, 7K nodes, and 9.5K nodes. Compared to data from real-world applications, these graphs are relatively small in size. The queries used for testing were also based on testing the validity of results, rather than emulating queries in realistic scenarios. There was also no testing done to compare the performance of these queries in MASS against existing graph database systems. As the original intention of the MASS Graph DB system implementation was to improve performance by using distributed memory to store and traverse the graph, it is important to compare MASS' implementation to disk-based solutions.

2.2 MOTIVATION

There has been little work done to provide a standard protocol for evaluating the capabilities of the MASS Graph DB system implementation. In this evaluation, we want to compare MASS Graph DB system to other commercially available graph database systems to identify the strengths and weaknesses of using distributed memory to create and analyze graphs, and to gain insight into the spatial and CPU scalability of using MASS Graph DB system. In addition, we want to use data from real-world applications for this performance testing, to emulate realistic scenarios when using a graph database system. This project's focus is to help future MASS graph database system researchers by establishing a framework for benchmarking their enhancements against Neo4j and ArangoDB.

2.3 DATABASE SYSTEMS

To evaluate the performance of the MASS graph database system against commercially available tools we use the following criteria to select an appropriate database:

- Cost: free or relatively low cost (capped at $10 per month which is relatively affordable for students)
- Cloud options: agnostic of provider
- OS options: agnostic of OS
- Community: a larger user base/active forum for asking questions and searching for answers

The above criteria will make it easier for future researchers to continue using these systems for benchmarking since the systems will not incur a cost, deployment is flexible, and common questions can be found/answered within the community forum. Based on these criteria, we have selected Neo4j and ArangoDB as the systems to compare MASS against.

Neo4j is an open-source graph database system that is one of the leading software systems in the graph database space [8,14]. It stores data using a node and relationship format, such that a node represents an entity in the graph, and a relationship represents an edge between two entities. It is cloud and operating system (OS) agnostic, meaning it can be deployed with any cloud provider and on any OS. It has a free edition as it is open-source, and a strong community base since it is one of the most popular graph database systems. Neo4j uses Cypher, an open-source query language developed by Neo4j. The basis of the language is similar to that of SQL, making it easier for developers who are used to querying relational databases to translate their queries into Cypher. Cypher is supported by other vendors as well.

ArangoDB, like Neo4j, is an open-sourced database system that also has a strong community of users. In addition to supporting graph storage, ArangoDB is a multi-model database that uses a document key-value model to store data [9]. It is cloud and OS agnostic, has a free edition available for development, and uses its own query language called ArangoDB Query Language (AQL). AQL is a declarative language that is meant to be human-readable. Its structure is similar to using a forloop in most coding languages, making it fairly straightforward to adopt for users who are familiar with common software development languages.

As Neo4j and ArangoDB also store data differently, the former is a native graph database, and the latter is a multi-model database, using these two systems for comparison will also show how the MASS Graph DB system performs against different data storage solutions.

2.4 Datasets

Graph data is useful for representing complex relationships between connected entities. For this project, we wanted to use real-world data to emulate storing and querying data in a realistic scenario. When determining what types of datasets to use for this benchmarking project, we explored common use cases for graph databases. Some recurring use cases include [15,16,17,18, 21,22] Recommendation Engine: recommending something based on the graph data, ex: recommending products to purchase with other items. Social Network: analyzing social media networks, ex: most followed users, or connections based on friendships. Fraud Detection: detecting fraudulent activities based on transaction data, ex: identifying all the transactions stemming from known fraudulent entities

Based on these use cases, we are using the Amazon Co-Purchased Products dataset [5] as the data for a recommendation engine use case, the Twitch [4] dataset as the data for a social network use case, and the Elliptic [7] dataset as the data for the fraud detection use case. These datasets are further described in section 4.

Chapter 3. RELATED WORKS

3.1 Graph Dataset Benchmarking Programs

The importance of having a benchmarking protocol is to standardize performance comparisons when evaluating a graph database system. In the domain of graph datasets, there have been several projects related to establishing such protocols. The studies described below utilize different methods and datasets for their benchmarking programs, such as using heterogenous data, creating hybrid graphs, and defining a benchmark protocol. Notably, all of these studies have a focus of using graph datasets for benchmarking machine learning and graph algorithms. Because of this, these studies do not generate data for performance testing, rather data is loaded and split for training purposes. Also, compared to the common graph database use cases [15,16,17,18,21,22] described in section 2 (recommendation engine, social network, and fraud detection), these projects explore other data domains outside of common real-world applications.

The Open Graph Benchmark (OGB) project offers a graph benchmarking framework that covers a variety of domains, including social network, with the emphasis on evaluation of machine learning on graphs [12]. OGB was developed to address the variability in experimental protocol when reproducing graph machine learning research. This work has a main focus on creating realistic benchmark datasets for machine learning. With a focus on machine learning training and testing, random graph generation is not explored in this work.

In real-world datasets, graph data is useful for representing complex relationships that cannot be easily stored in a typical relational database. Analyses of graph data typically employ graph learning algorithms, such as graph neural networks. Training these models usually require real-world datasets for learning and fine tuning [10]. Kumarasinghe, et. al’s worked on creating the largest public heterogenous graph dataset for malicious domain classification [10]. In their research, researchers used VirusTotal to collect the malicious domains and extracted hosting infrastructure information from a DNS repository to create a heterogenous graph with the intention of improving graph neural network (GNN) models. They discovered that larger heterogenous datasets performed better for classification tasks, however they did not outperform their homogenous dataset counterparts. This heterogenous graph also falls outside of the common industry use cases for graph data, with the project's primary focus on improving data for GNNs. The data is also loaded in and split accordingly for GNN training and testing, there is no random graph generation component.

A separate work introduces hybrid graph generation for GNN evaluation [11]. The conceptualization of a hybrid graph as a representation of higher-order graphs where nodes are connected to groups of nodes, rather than just other nodes. This research defines the groupings and provides both datasets and an evaluation framework for hybrid graph creation and GNN testing. This work focuses on three graph domains: social network, biology, and e-commerce in the form of product reviews. While this research addresses a common industry graph dataset use case, social network, this project's main focus is to provide a benchmark for GNN. Similarly, to the heterogenous graph study, this work also does not offer random graph generation.

3.2 OVERVIEW

Graph generation in the graph dataset benchmarking programs above have a primary interest in machine learning and graph learning algorithms. They do not take into consideration user input for graph size, nor random graph generation, which are both important for comprehensive execution and performance analysis of a graph database system. These works have a focus on evaluating and advancing graph learning algorithms, not an evaluation of graph database performance. The graphs created in these works are not appropriate for comparative analysis of the MASS graph database system against commercially available products. The domains for these projects also do not necessarily reflect common real-world use cases for graph data. Our approach to graph dataset creation accepts user input for the size of the graph. At MASS' current state, it may not be able to handle immensely large datasets, and being able to specify the size of the graph is required for performance evaluation. The resulting graph will be randomized while maintaining subgraph degree distribution using the network motif algorithm. The randomization of the graph is important for fairness in our evaluation to reduce the chances of hardcoding query results to give an illusion of performance increase. Lastly, our benchmarking tool creates the following graph types described below in Table 1. These graph types correspond to common industry use cases and the data used for graph generation is extracted from real-world datasets [4,5,7].

Table 1: Graph Use Cases and Corresponding Dataset

Type	Graph Use Case	Corresponding Real-World Dataset
Products	Recommendation Engine	Amazon Co-Purchased Products
Social	Social Network	Twitch
Transaction	Fraud Detection	Bitcoin Transactions

Chapter 4. METHOD

This section describes the design decisions and implementation details of the benchmarking protocols. We will describe the selected datasets and how they relate to the common graph database use cases, the implementation of the random graph generator, common queries for the different datasets, and the testing strategy for each of the graph database systems, Neo4j, ArangoDB, and MASS.

4.1 DATASETS AND RANDOM GRAPH GENERATOR

The datasets we chose correspond to a common graph use case described in section 1. These use cases include recommendation engine, social network, and fraud detection. For recommendation engine, we wanted to use a dataset that could be used to recommend products for purchasing. The Amazon Co-Purchased Products dataset [5] contains information regarding the different products, and edge information to indicate products that were purchased together. Based on this, we would be able to query the data to see co-purchased products to simulate a recommendation scenario. Twitch is a social network with a focus on live streaming gameplay. Like most social network platforms, there is a concept of following other users. The Twitch Social Network dataset [4] can be used for social network analysis, such as identifying the most followed user. For fraud detection, we are using the Elliptic dataset [7]. This is an anonymized Bitcoin transaction dataset with entities labeled as illicit (fraudulent), licit, and unknown. To mimic a fraud detection scenario, we can query this data to identify all transactions made with illicit entities. The size of each dataset is described in Table 2.

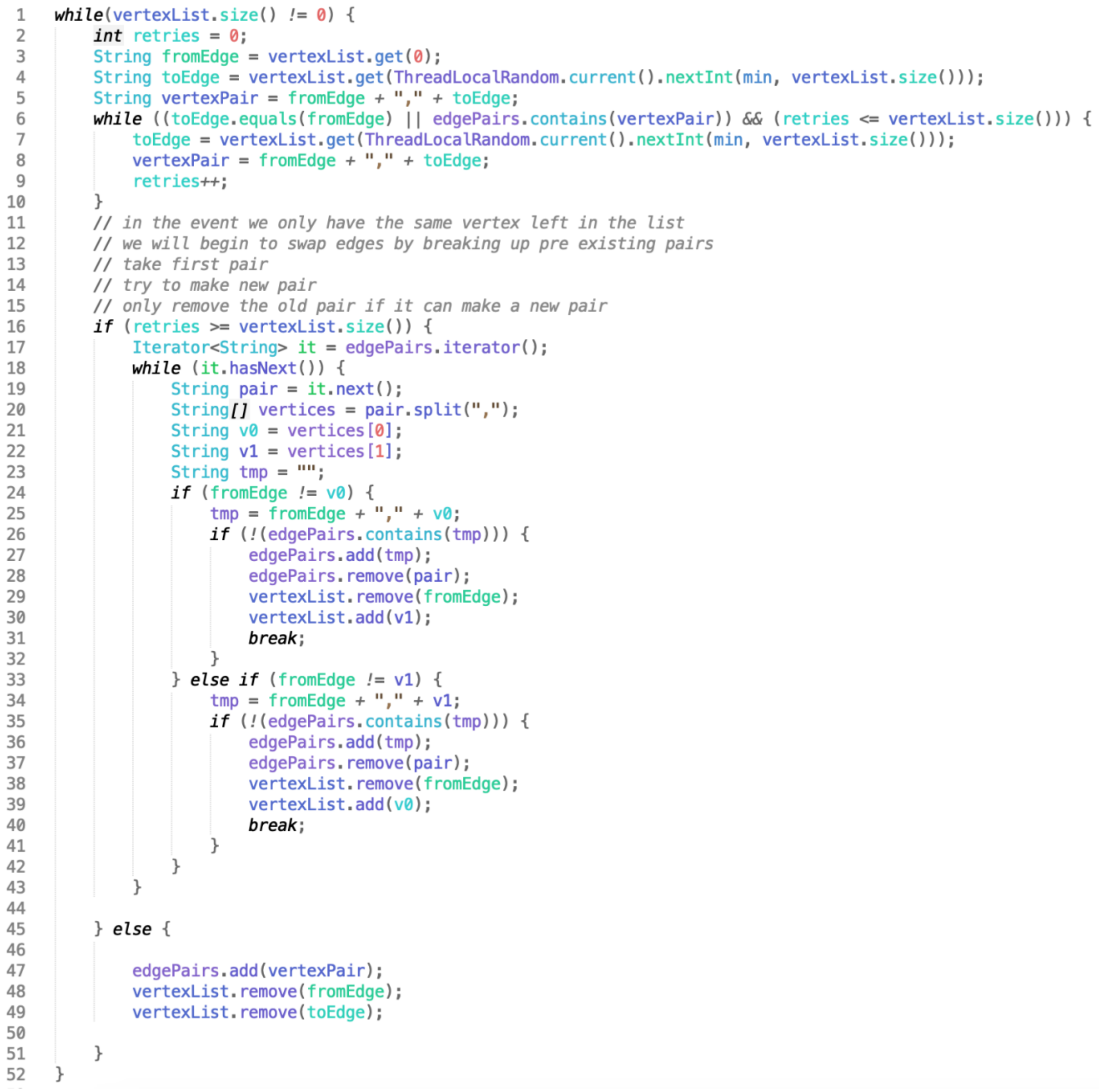
Table 2: The size of each graph dataset.

Dataset	Number of Nodes	Number of Edges
Amazon	548,552	3,387,388
Twitch	168,114	6,797,557
Elliptic	203,769	234,355

For performance evaluation, we decided to using random graphs to guarantee fairness in our testing. The random graphs use data pulled from the datasets described in section 4.1. The randomness comes from the rearrangement of the edges between different nodes. As the relationship between edges in a graph are important, we wanted to maintain any subgraph patterns that might be of interest for future analysis. Figure 1 shows a code snippet of how the degree distribution algorithm will behave in the event that the vertices in the list cannot be used to create new edges (line 16-43).

To generate a random graph while maintaining the original degree distribution we use the following logic [6]:

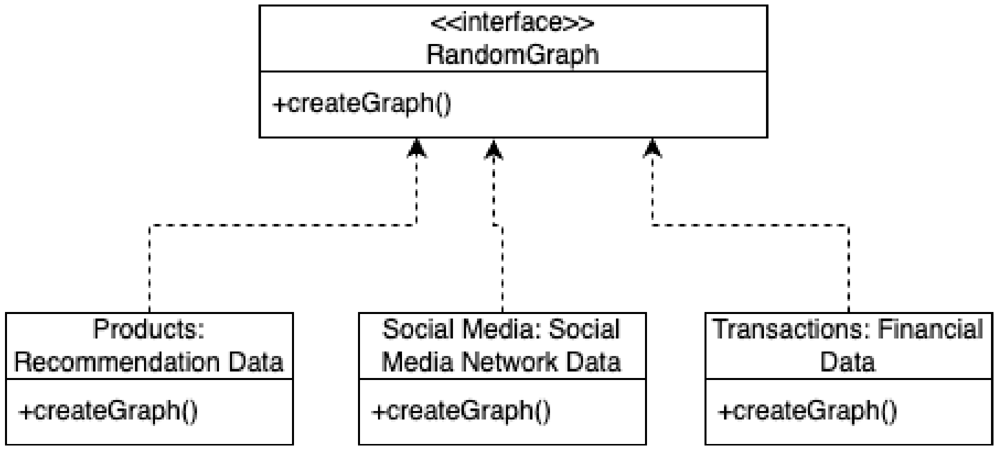
- For an edge, add each vertex ID to a list. (stored in vertexList in Figure 1)
- Pick two vertex IDs from the list to form an edge. (lines 3-10 in Figure 1)
- Check if this edge already exists in our set of edges. If it does, try to form a new edge from the vertex list. (line 6 in Figure 1)



```java
while(vertexList.size() != 0) {
    int retries = 0;
    String fromEdge = vertexList.get(0);
    String toEdge = vertexList.get(ThreadLocalRandom.current().nextInt(min, vertexList.size()));
    String vertexPair = fromEdge + "," + toEdge;
    while ((toEdge.equals(fromEdge) || edgePairs.contains(vertexPair)) && (retries <= vertexList.size())) {
        toEdge = vertexList.get(ThreadLocalRandom.current().nextInt(min, vertexList.size()));
        vertexPair = fromEdge + "," + toEdge;
        retries++;
    }
    // in the event we only have the same vertex left in the list
    // we will begin to swap edges by breaking up pre existing pairs
    // take first pair
    // try to make new pair
    // only remove the old pair if it can make a new pair
    if (retries >= vertexList.size()) {
        Iterator<String> it = edgePairs.iterator();
        while (it.hasNext()) {
            String pair = it.next();
            String[] vertices = pair.split(",");
            String v0 = vertices[0];
            String v1 = vertices[1];
            String tmp = "";
            if (fromEdge != v0) {
                tmp = fromEdge + "," + v0;
                if (!(edgePairs.contains(tmp))) {
                    edgePairs.add(tmp);
                    edgePairs.remove(pair);
                    vertexList.remove(fromEdge);
                    vertexList.add(v1);
                    break;
                }
            } else if (fromEdge != v1) {
                tmp = fromEdge + "," + v1;
                if (!(edgePairs.contains(tmp))) {
                    edgePairs.add(tmp);
                    edgePairs.remove(pair);
                    vertexList.remove(fromEdge);
                    vertexList.add(v0);
                    break;
                }
            }
        }

    } else {

        edgePairs.add(vertexPair);
        vertexList.remove(fromEdge);
        vertexList.remove(toEdge);

    }
}
```

Figure 1: Code snippet of degree distribution algorithm when edge pairs need reassignment

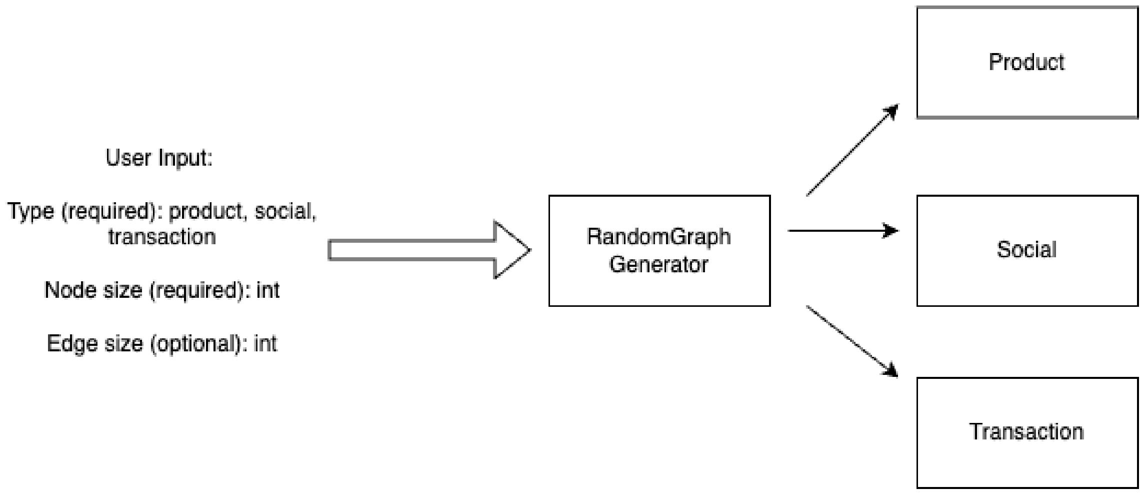
The random graph generator takes in user input for both the type and size of the graph to generate. There are three types of graphs that can be generated, Product, Social, and Transaction as shown in Figure 2. Product data is extracted from the Amazon dataset. Social data is extracted from the Twitch dataset. Transaction data is extracted from the Elliptic dataset. These are RandomGraph objects that implement a createGraph method that generates 4 csv files. These csv files will be used for graph creation in Neo4j, ArangoDB, and MASS.



<<interface>>
RandomGraph
+createGraph()
Products: Recommendation Data
+createGraph()
Social Media: Social Media Network Data
+createGraph()
Transactions: Financial Data
+createGraph()

Figure 2: RandomGraph Objects

Based on user input for both the type and size of the graph, the random graph generator will then generate the appropriate csv files, shown in Figure 3. The size of the graph is limited to the size of the corresponding real-world dataset described in Table 2.



User Input:
Type (required): product, social, transaction
Node size (required): int
Edge size (optional): int
RandomGraph Generator
Product
Social
Transaction

Figure 3: RandomGraphGenerator takes user input to create the appropriate csv files

4.2 COMMON QUERIES

For the common queries, we wanted to use queries that simulate realistic scenarios based on the type of dataset being queried. In general, we want to test out CRUD (Create, Read, Update, Delete) operations that all databases support, and graph traversal which all graph databases support. In addition, we have specific queries for each dataset, such as finding the most followed user using social network data. These dataset specific queries are meant to compare how MASS performs in a realistic situation when querying graph data. The set of common queries shown in Table 3 describe the queries we will use for benchmarking MASS.

Table 3: Common queries per dataset for benchmarking

Type	Query	Reasoning
General	Create new node Create new relationship between new node and an existing node Match on criteria to return new node Delete relationship Match on criteria to see if deleted relationship returns any result Delete node Match on criteria to see if deleted node is returned Graph Traversal (Depths 1-3)	Testing the general functionality of the graph database: • CRUD operations (Create, Read, Update, Delete) • Graph Traversal
Products	Match on products that are purchased together Most popular products purchased together	Recommend new products for people to purchase
Social Network	Match on follows of people you follow (can do this by connections of connections, and second connections) Match on the most popular user (most follows)	This is to recommend new people for someone to follow
Transactions	Match on the illicit nodes	To identify all known fraudulent transactions, we can use a match query to return all the illicit transactions

4.3 MASS Benchmarking Program

The benchmarking program in MASS will take user input to generate the 4 csv files for graph creation. MASS will use the files that contain “MASS” in the name, the other two files will be used for Neo4j and ArangoDB testing. MASS is initiated and then the graph is created using the

GraphManager. From there, a list of Cypher queries is run using the queryHandler function. Graph creation and query execution times are printed to the console.

4.4 NEO4J MANUAL

For Neo4j testing, we created a set of instructions for conducting analysis. Originally, we planned on using a python script that executes the queries and logs the execution time, however the executeQuery call includes overhead outside of the query runtime that greatly inflates the execution time. Because of this, we will run the queries directly in the Neo4j user interface. The manual includes instructions on set up, how to load data, how to create a graph, and the queries written in Cypher for testing.

4.5 ARANGODB SCRIPT

ArangoDB uses AQL for querying data. There are two parts to ArangoDB testing. The importing of data into collections is done using the arangoimport command. We created a shell script with these commands, with the execution time being printed to the console once collection creation is complete. We then call arangosh to open the arango shell environment. We use a script, arango_test.js, to create the graph using the newly created collections and conduct query testing. The queries in the script are the Cypher queries used in MASS and Neo4j testing translated into AQL. To execute the script, use the command require("internal").load("arango_test.js"). The execution times are printed to the console.

Chapter 5. EVALUATION

We are using social graphs of size 1K nodes, 10K nodes, 20K nodes, and 30K nodes for spatial scalability testing. During our initial evaluation, we used a 50K, and 100K node graph for testing graph traversal which resulted in an out of memory error within MASS Graph DB system. We then adjusted the size of the graphs for testing using 10K node increments that successfully returned query results in MASS Graph DB system. This testing includes graph creation, and graph traversal of depths 1 – 3 using Neo4j, ArangoDB, and MASS Graph DB system on a single computing node. For the graph database systems evaluation, we focused on spatial scalability testing using different sized graphs on a single machine to better understand the strengths and limitations of using an in-memory data storage solution compared to disk-based systems. Depth traversal through levels 1 – 3 is a common graph database query use case and has been used in other studies for performance testing of database systems [23]. The computing node has 16 GB of memory. For CPU scalability testing, we are using a social graph of size 10K nodes on the Hermes cluster, using 8 computing nodes. We chose a 10K graph because it is large enough to showcase the benefits and limitations of MASS for both graph creation and query execution during spatial scalability testing, and allows us to further explore the effects of CPU scalability. This testing includes graph creation, and graph traversal of depth 1 using MASS.

Due to the limitations of MASS Graph DB system, we are mainly using social graphs for testing, as the current implementation can only process Create and simple Match queries.

5.1 PERFORMANCE ANALYSIS

5.1.1 *Spatial Scalability*

We want to evaluate the spatial scalability of each graph database system, which references how each system performs as the size of the data increases. To evaluate this metric, we are capturing the time it takes to create a graph via csv importing, and the query execution time when traversing the graph from depth 1 to depth 3 [. We are using social graphs of sizes 1K nodes and 257 edges, 10K nodes and 24K edges, 20K nodes and 92K edges, and 30K nodes and 204K edges. Table 4 shows the queries used for graph traversal. The queries are written in both Cypher and AQL.

Table 4: Queries used for graph traversal

Description	Cypher	AQL
Graph traversal depth 1	MATCH (a)-->(b) RETURN b	FOR vertex in socialNodes FOR vertices, edges, paths in 1..1 OUTBOUND vertex GRAPH social RETURN vertices
Graph traversal depth 2	MATCH (a)-->()-->(b) RETURN b	FOR vertex in socialNodes FOR vertices, edges, paths in 2..2 OUTBOUND vertex GRAPH social RETURN vertices
Graph traversal depth 3	MATCH (a)-->() -->()--> (b) RETURN b	FOR vertex in socialNodes FOR vertices, edges, paths in 3..3 OUTBOUND vertex GRAPH social RETURN vertices

In Figure 4, we show the time it takes for each graph database system to create graphs of different sizes. Graph creation for all graph database systems was done via import of csv files. For larger graphs, 20K and 30K, MASS Graph DB system outperforms both Neo4j and ArangoDB for graph creation. Generally, Neo4j is the slowest for graph creation.

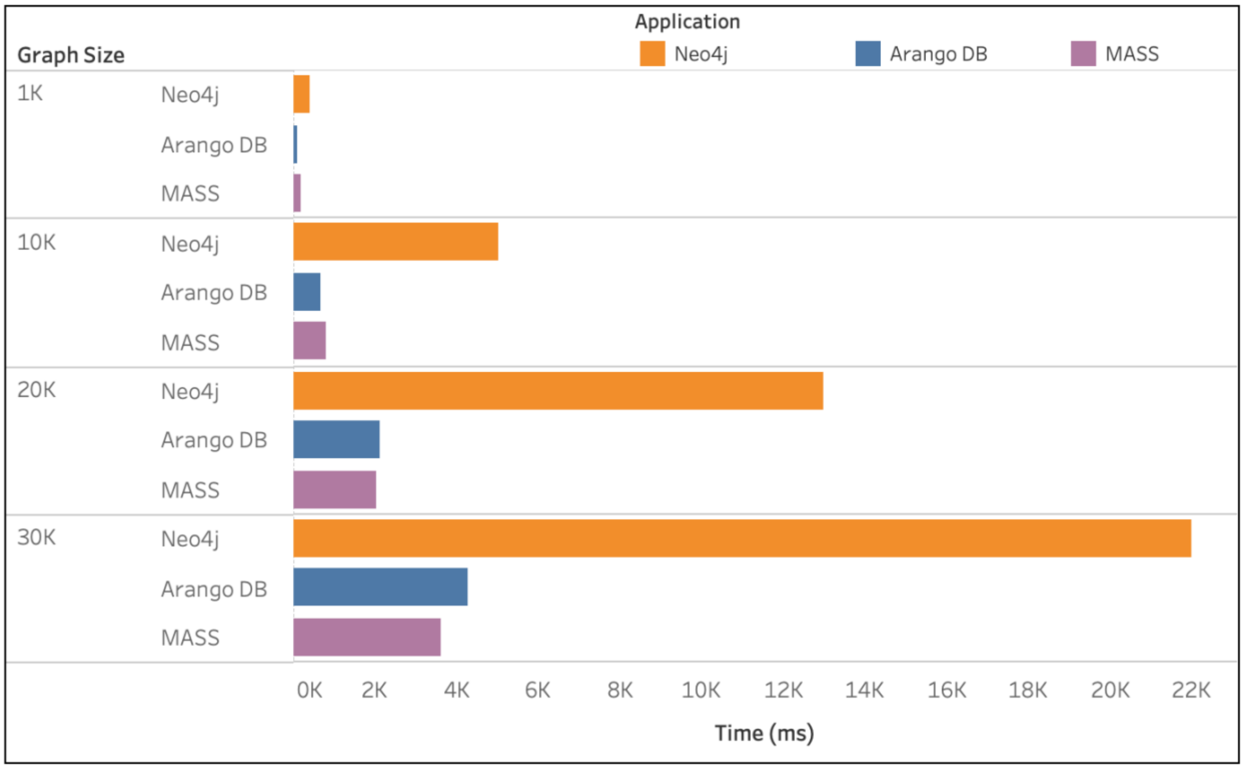


Figure 4: Graph creation time across graph database systems

Figures 5, 6, and 7 demonstrate the query execution time for each graph database system when traversing depth 1, depth 2, and depth 3 of the various graphs. In Figure 5, we see that as the graph size increases, the depth 1 graph traversal in MASS Graph DB system increases, with 144x time increase in traversal time between the 1K graph and 30K graph. ArangoDB is the most performant for depth 1 traversal. In Figure 6, we see MASS Graph DB system is more performant than ArangoDB for depth 2 traversal. In Figure 7, again MASS Graph DB system outperforms ArangoDB for depth 3 traversal as ArangoDB runs out of memory. Overall, Neo4j's performance is relatively consistent and is generally the most performant for graph traversal of any depth. This is expected, as Neo4j data storage uses index-free adjacency.

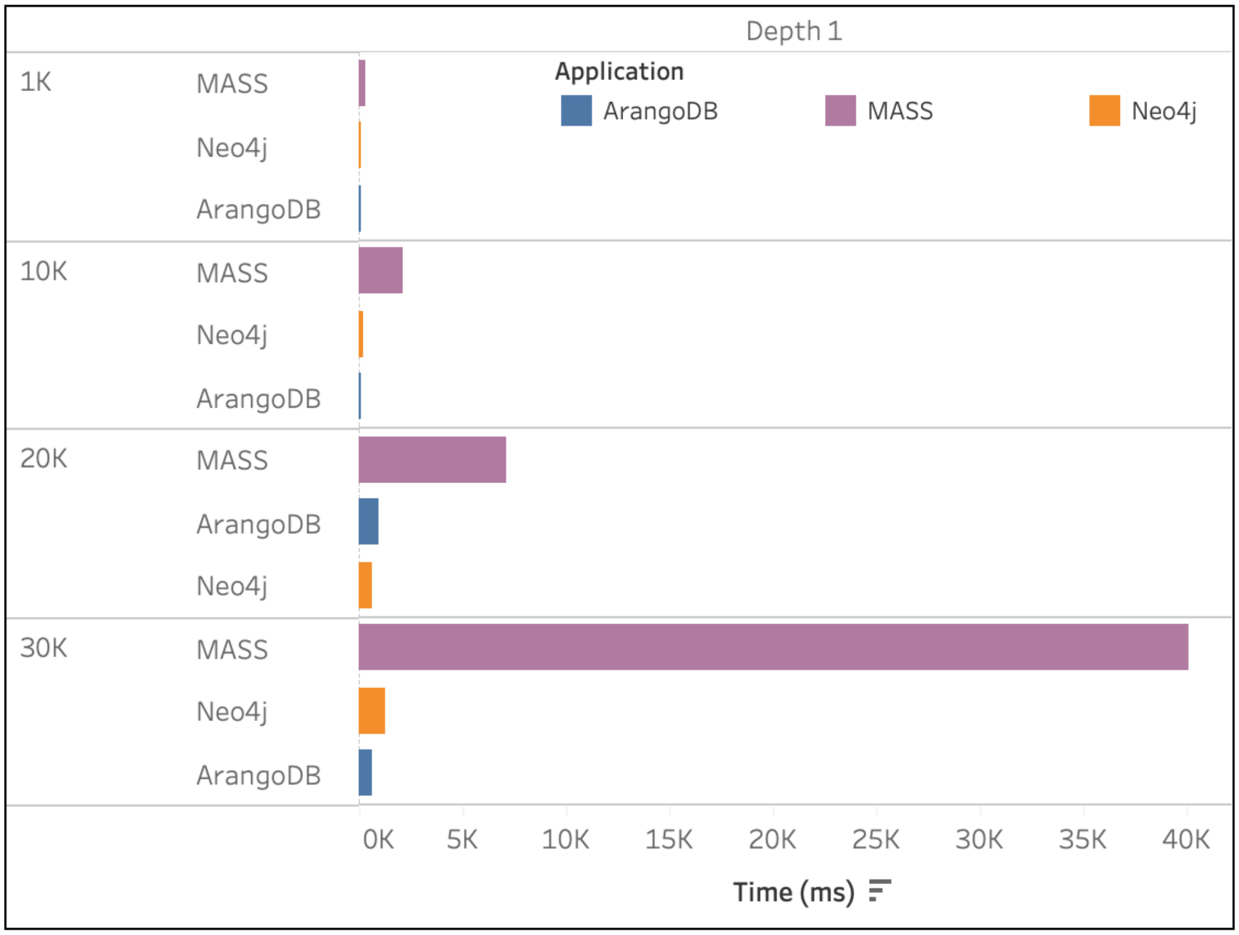


Figure 5: Depth 1 traversal across graph database systems

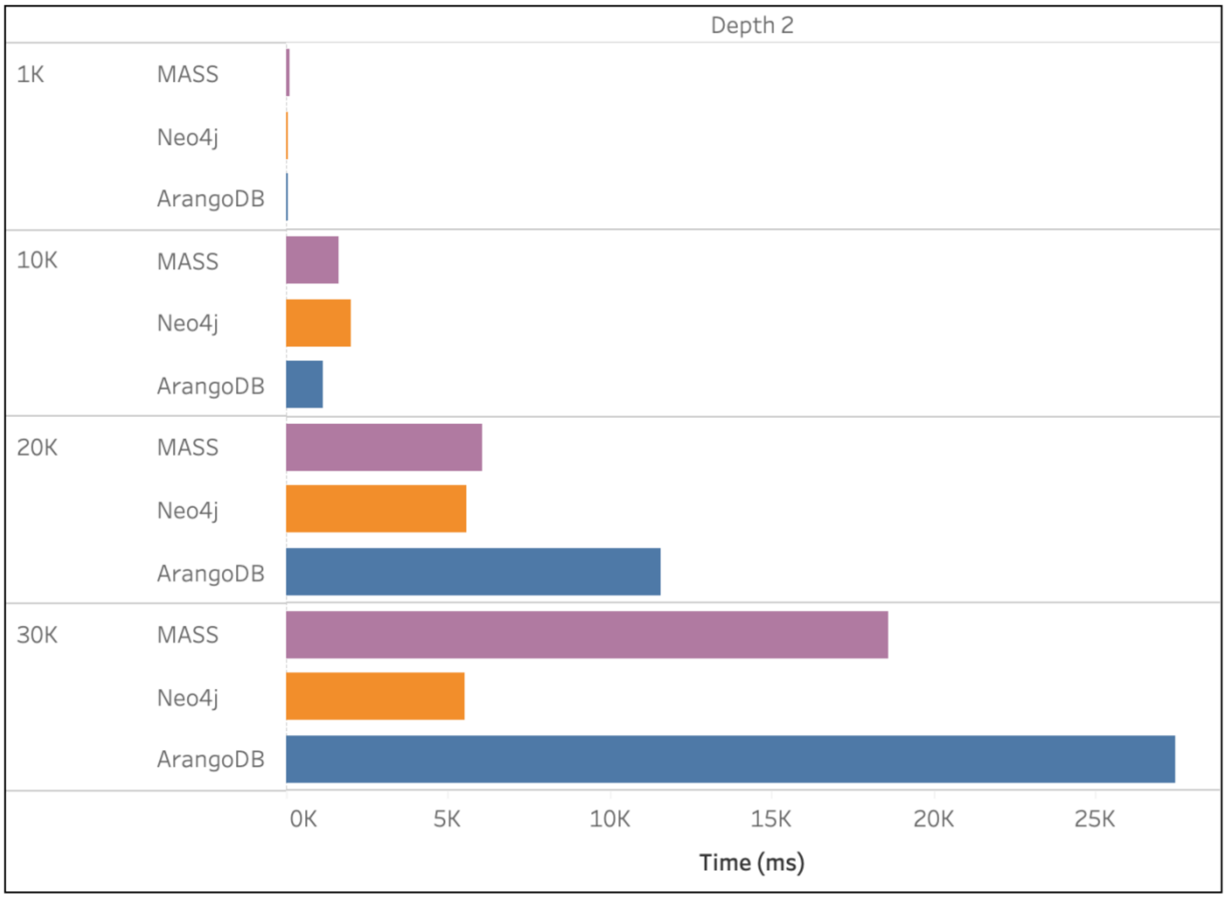


Figure 6: Depth 2 traversal across graph database systems

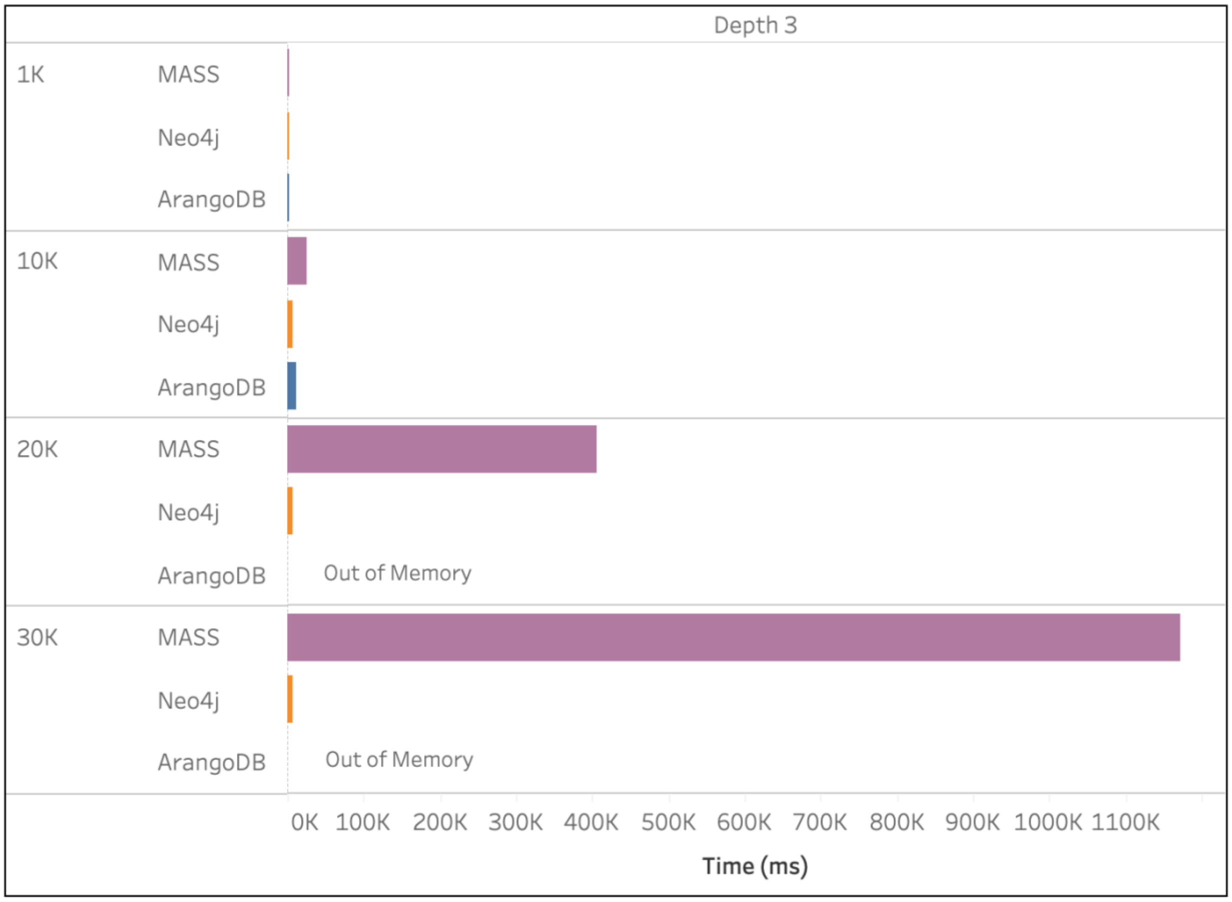


Figure 7: Depth 3 traversal across graph database systems

5.1.2 *CPU Scalability*

MASS’ computing architecture leverages multiple computing nodes to achieve parallelization. To evaluate the CPU scalability of MASS Graph DB system, we will be using a 10K node social graph for graph creation and graph traversal of depth 1 using 1-8 computing nodes.

Figure 8 shows the graph creation time across 1-8 computing nodes. We see that the time substantially increases between one and two computing nodes. The time appears to plateau and stabilize around 5-8 computing nodes. This time difference between using 1 and 2 nodes is due to the sequential nature of graph creation in MASS.

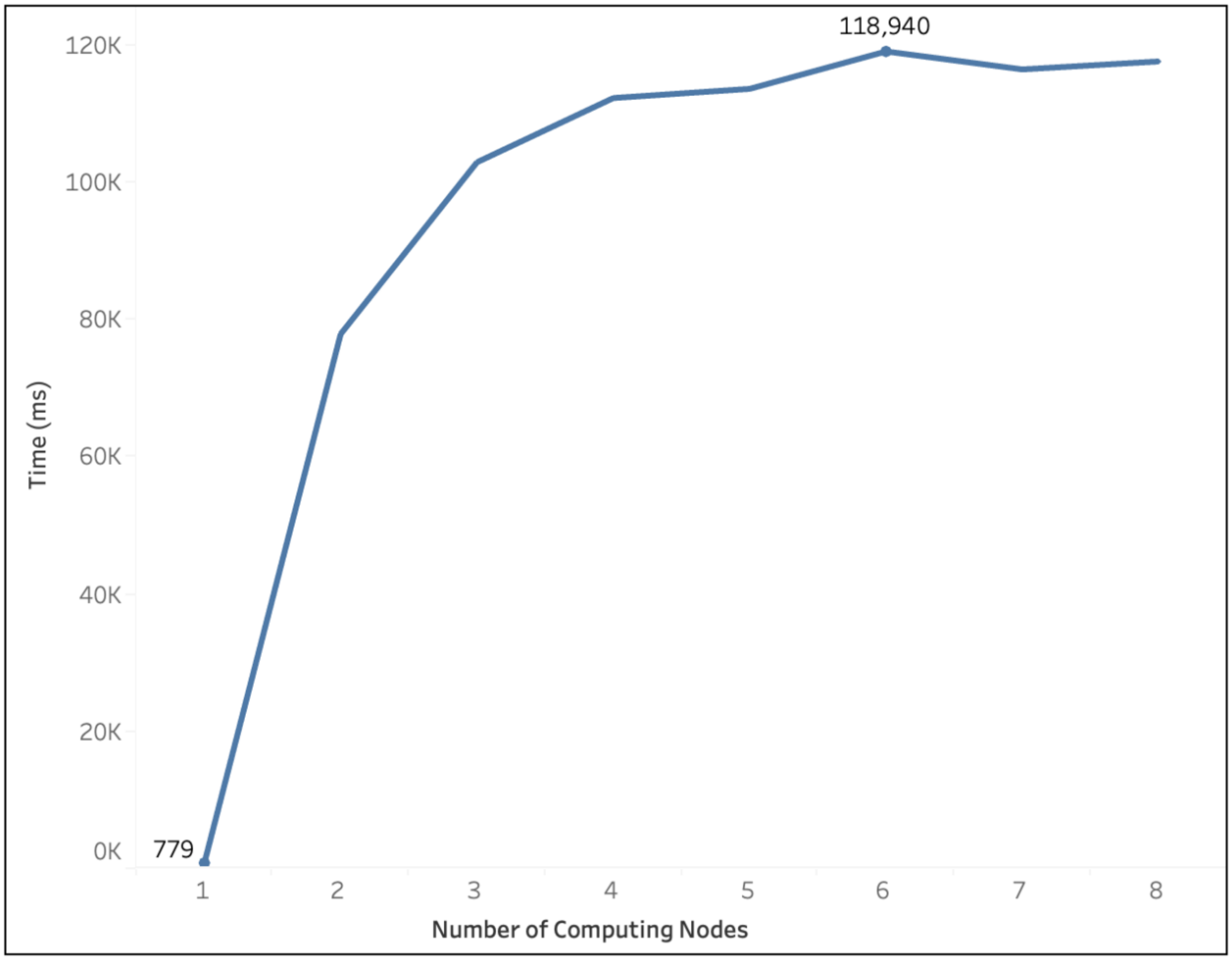


Figure 8: Graph creation time from 1-8 computing nodes

Figure 9 displays the graph traversal of depth 1 query time when using 1-8 computing nodes. We see a spike in the use of two computing nodes, with a downward trend as we continue to add computing nodes. This increase in time is due to communication overhead from additional computing nodes.

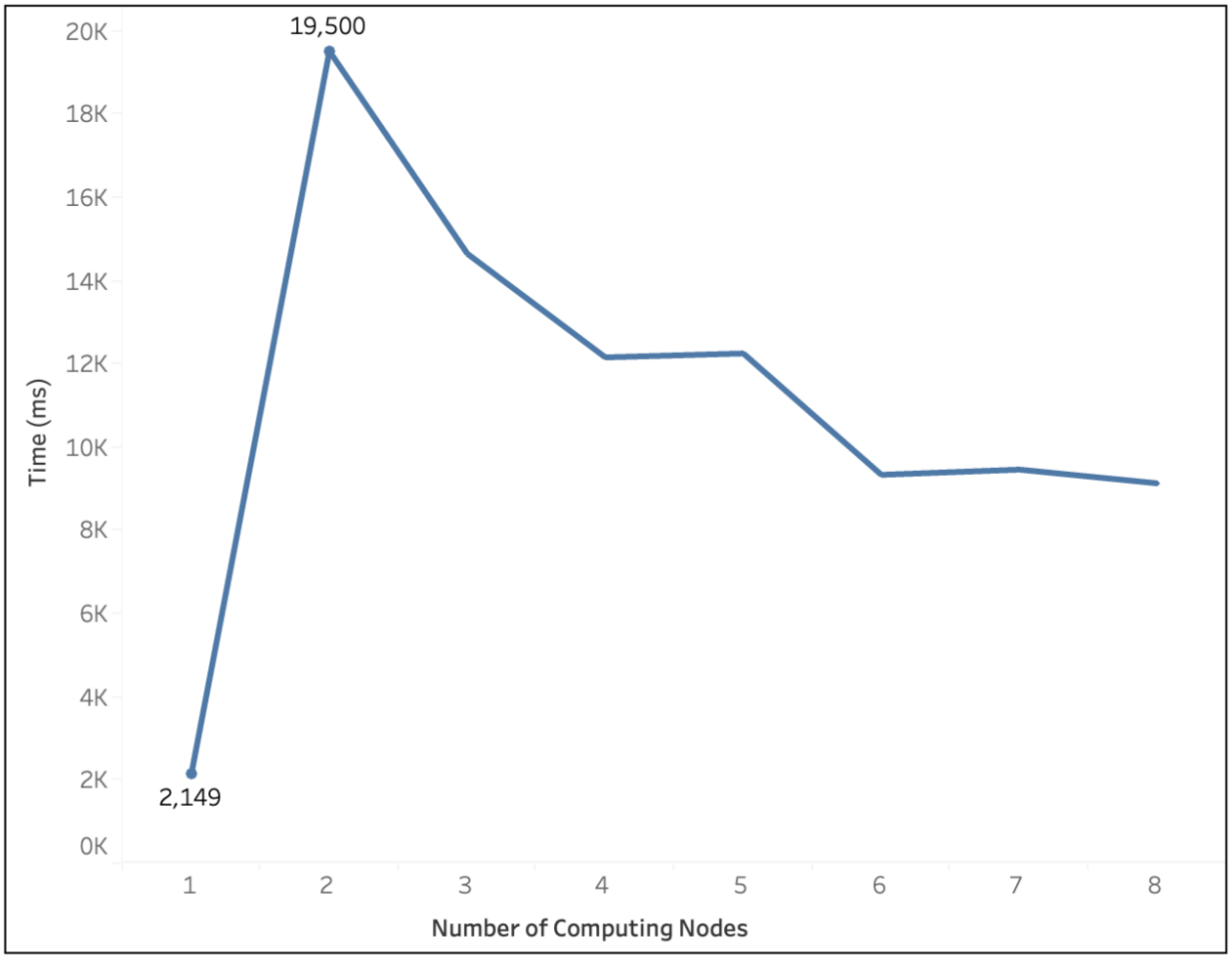


Figure 9: Depth 1 graph traversal across 1-8 nodes

5.2 Discussion

5.2.1 *Spatial Scalability Graph Creation*

In our evaluation of spatial scalability, we identified MASS Graph DB system as the fastest system for creating large graphs via csv import. Neo4j was the slowest for graph creation, followed by ArangoDB. This is likely due to the way we store node data in each system. For Neo4j, data is stored on the disk as a linked list, with each node pointing to its next neighbor. Node and relationship data are stored in separate database files [24]. In ArangoDB, data is also stored on the disk but as a JSON object referred to as a document [25]. Documents are organized into collections. Nodes and edges are stored as separate collections of documents. MASS Graph DB system stores nodes as a PropertyVertexPlace object, with two separate hashmaps to capture directed edge information. This data is stored in memory, in contrast with the other two systems which store data in the disk. The lengthier execution time for graph creation in Neo4j is related to this linked list storage method.

5.2.2 *Spatial Scalability Graph Traversal*

The storage of data using this strategy of linked lists is known as index-free adjacency, where instead of a traversing a graph via an index, it uses a pointer instead. Unlike a relational database where performance is usually dependent on the size of the tables, performance in Neo4j is dependent on the connectedness of the nodes in a traversal. This results in Neo4j having fast and consistent query execution times when traversing a graph of any size at any depth level. ArangoDB's graph traversal slows down for larger graph sizes at depths greater than one. Graph traversal uses the edge collection, which are documents stored as key-value pairs. MASS Graph DB system also sees a slowdown with increased depth traversal, especially for depth 3. MASS' graph system stores its neighbor information in hashmaps as part of the PropertyVertexPlace object. Because of the creation the graph nodes in a round robin fashion, the placement of nodes and their corresponding hashmaps can cause a slowdown during traversal as the agents need to read through the different maps and travel to the appropriate neighbors. This travel is not optimized, as there is no logical grouping of node storage based on the edges within the graph. Traversal requires the reading of both hashmaps to identify the appropriate agent migration pattern. For depths 1 and 2, MASS Graph DB system graph traversal actually decreases in execution time. This is due to the storage of data in the cache, with data for depth 3 needing to be accessed from memory. A limitation to consider when comparing MASS' in-memory graph database system to Neo4j and ArangoDB's disk-based database systems, is that while accessing data held in memory is generally faster than disk retrieval, other applications that require memory may also be competing for resources with the MASS Graph DB system. This scenario may also result in a slowdown for the MASS Graph DB system.

5.2.3 *Graph Traversal Baseline*

Based on our findings regarding graph traversal across all graph database systems, we wanted to establish a baseline of graph traversal. For the baseline, we decided to capture graph traversal execution times when starting from a random node. By starting from a random node, the data should not be available in cache or memory for the disk-based solutions. This helps highlight the differences between disk-based (Neo4j and ArangoDB) and memory-based (MASS Graph DB system) query times. For graphs with 10K nodes, 20K nodes, and 30K nodes, we traversed the graph from depths 1-3 starting from a random vertex, as shown in Figure 10. MASS performs the worst for depth 1 and depth 2 traversals. For 20K and 30K graphs, at depth 3 traversal, we see that MASS outperforms both Neo4j and ArangoDB. This is likely due to both Neo4j and ArangoDB needing to access the disk to return data at depth 3 traversal. The Neo4j and ArangoDB execution times for depths 1 and 2 will serve as the performance goal for MASS as more enhancements and optimizations are made to the Agents for graph traversal.

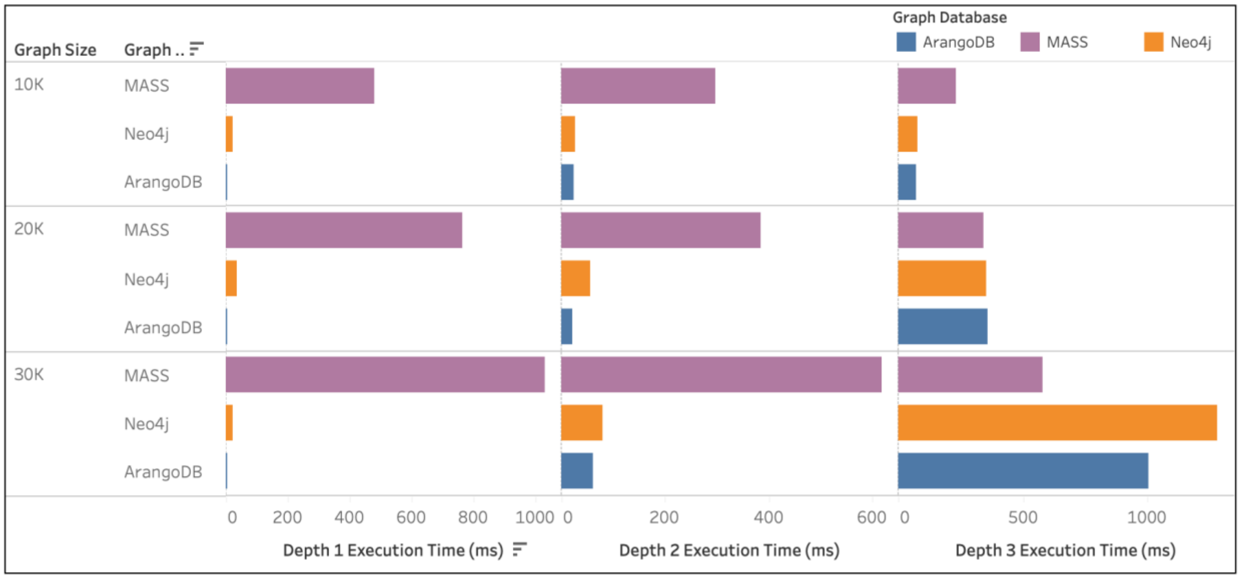


Figure 10: Graph traversal depths 1-3 from a starting vertex

5.2.4 *CPU Scalability*

Our evaluation of CPU scalability for MASS Graph DB system used a 10K graph and 1-8 computing nodes on the HERMES cluster. When using additional computing nodes, we anticipated an increase in execution times because of the communication overhead between the additional computing nodes and main computing node. MASS uses a round robin technique to evenly distribute the creation of nodes across the available computing nodes. The increase in execution time for graph creation is expected as graph creation is done sequentially. We noticed a plateau occurring in execution time around 5-8 computing nodes. This is likely due to the overlap between the computing nodes sending messages about which graph nodes to create, while simultaneously also creating the appropriate nodes. To confirm this behavior, we should consider logging the behavior at each computing node to see if the timestamps overlap between messages being sent, and graph node creation.

With graph traversal of depth 1 using 1-8 computing nodes, we noticed that adding a second computing node introduced a significant spike in the execution time. The communication overhead is responsible for this increase. As we continue to add more computing nodes, the traversal time decreases. Here we are able to demonstrate that the traversal is being parallelized across the nodes via agent migration, as communication overhead remains constant despite adding more nodes. This downward trend is a sign that MASS Graph DB system is working as expected and is able to scale horizontally to improve performance for large data sources. At this stage, since the single node execution is faster than multi-node execution, optimization of communication is key to better leveraging MASS' capabilities for a more performant graph database system.

Chapter 6. CONCLUSION

In this project we were able to identify two commercial graph database systems, Neo4j and ArangoDB, to use for performance comparison of MASS Graph DB system. Benchmarking datasets were selected for performance testing based on the common industry use cases of recommendation engine, social network, and fraud detection. We implemented a random graph generator that allows users to select the type and size of the graph to generate. A set of common queries for each dataset was created both in Cypher and AQL. Performance testing using different sized graphs was done on all three graph database systems to demonstrate spatial scalability, and CPU scalability testing was conducted for the MASS Graph DB system using 1-8 computing nodes. For MASS testing, a main test program can be used to both generate the graph and run the appropriate queries. For Neo4j testing, we created a manual with instructions of set up and Cypher queries. For ArangoDB testing, we developed a shell script for graph creation, and a JavaScript script for running queries on the database.

Overall, MASS handles graph creation for larger datasets better than the other two applications, and outperforms graph traversal compared to ArangoDB for datasets with 20K nodes or more at depths greater than one. Graph traversal in Neo4j is faster than the other two applications, due to its index free addressing.

For future work and improvements:

1. Once MASS Graph DB system has implemented more functionality, test the rest of the common queries using the different graph types to gauge the topology of datasets MASS would be appropriate for.

2. ArangoDB can also be set up in a cluster system for CPU scalability comparison. This would be interesting to see how this affects larger dataset graph traversal. However, parallelization of graph traversal is only available for the enterprise edition. This test would be mainly for CPU scalability comparison.
3. Neo4j queries are currently run manually in the web console because using the Neo4j driver to execute queries causes significant overhead. Future researchers should consider looking into options for gathering query execution time without using the executeQuery call.
4. Expand the benchmarking protocol to include ML testing. Many graph applications incorporate ML analyses to uncover insights in the data, such as detecting fraudulent transactions. This would be interesting to explore now that ML agents have been implemented.
5. Add a UI component to display the graph. Both Neo4j and ArangoDB display the graphs when returning results for a query.

REFERENCES

[1] “MASS: A parallelizing library for multi-agent spatial simulation.” [Online]. Available: http://depts.washington.edu/dslab/MASS/

[2] Rajvaidya, Harshit “An Agent-based Graph Database” White Paper. [Email]. Available: https://depts.washington.edu/dslab/MASS/reports/HarshitRajvaidya_whitepaper.pdf

[3] Leskovec, J., & Krevl, A. (2014, June). SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data

[4] Twitch Social Networks. [Online]. Available: https://snap.stanford.edu/data/twitch-social-networks.html

[5] Product co-purchasing networks. [Online]. Available: https://snap.stanford.edu/data/#amazon

[6] Kim, Wooyoung “Network Motif” Lecture Slides. [Email]. Available: via Professor Wooyoung Kim.

[7] Mark Weber, Daniel Karl I. Weidele, Giacomo Domeniconi, Claudio Bellei, Charles E. Leiserson, Jie Chen, and Tom Robinson. 2019. Anti-money laundering in bitcoin: “Experimenting with graph convolutional networks for financial forensics.” arXiv (2019). Issue 10. https://arxiv.org/abs/1908.02591

[8] Neo4j [Online]. Available: https://neo4j.com/

[9] ArangoDB [Online]. Available: https://arangodb.com/

[10] Kumarasinghe, U., Deniz, F., & Nabeel, M. (2022, March 15). “PDNS-net: A large heterogeneous graph benchmark dataset of network resolutions for graph learning.” arXiv.org. https://arxiv.org/abs/2203.07969

[11] Li, Z., Zhao, X., Shen, M., Stan, G.-B., Liò, P., & Zhao, Y. (2024, February 20). “Hybrid graph: A unified graph representation with datasets and benchmarks for complex graphs.” arXiv.org. https://arxiv.org/abs/2306.05108

[12] Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., & Leskovec, J. (2021, February 25). “Open graph benchmark: Datasets for machine learning on graphs.” arXiv.org. https://arxiv.org/abs/2005.00687

[13] Tsui, Caroline “Agent-based Graph Applications in MASS Java and Comparison with Spark” Term Paper. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/CarolineTsui_whitepaper.pdf

[14] Disney, Andrew “How to choose a graph database: we compare 6 favorites” [Online]. Available: https://cambridge-intelligence.com/choosing-graph-database/

[15] Smith, Daniel Alexander “Use cases for graph databases” [Online]. Available: https://6point6.co.uk/insights/use-cases-for-graph-databases/

[16] Neo4j “Graph database use cases” [Online]. Available: https://neo4j.com/use-cases/

[17] Neo4j “Use Cases: Social Media and Social Network Graphs” [Online]. Available: https://neo4j.com/use-cases/social-network/

[18] Amazon “What are the use cases of graph databases” [Online]. Available: https://aws.amazon.com/nosql/graph/

[19] Sandell, J., Asplund, E., Ayele, W. Y., & Duneld, M. (2024, January 30). “Performance comparison analysis of arangodb, mysql, and neo4j: An experimental study of querying connected data.” arXiv.org. https://arxiv.org/abs/2401.17482

[20] M. Macak, M. Stovcik, B. Buhnova and M. Merjavy, "How well a multi-model database performs against its single-model variants: Benchmarking OrientDB with Neo4j and MongoDB," *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, Sofia, Bulgaria, 2020, pp. 463-470, doi: 10.15439/2020F76.

[21] Oracle “17 use cases for graph databases and Graph Analytics.” [Online]. Available: https://www.oracle.com/a/ocom/docs/graph-database-use-cases-ebook.pdf

[22] Yousry, A. “7 graph database use cases that will change your mind.” Medium. https://medium.com/technology-hits/7-graph-database-use-cases-that-will-change-your-mind-699e92437523

[23] Oliveira, F. R., & del Val Cura, L. “Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications.” [Online]. Avaiable: https://dl-acm-org.offcampus.lib.washington.edu/doi/abs/10.1145/2938503.2938518

[24] *The Evolution of Big Data and the Future of the Data Platform*. (2022). Oracle.

[25] Rocha, Jose “Understanding Neo4j’s data on disk” [Online]. Available: https://neo4j.com/developer/kb/understanding-data-on-disk/

[26] ArangoDB “Data Structure” [Online]. Available: https://docs.arangodb.com/3.10/concepts/data-structure/

[27] Dryburgh, Alastair, “Growth Stories: The Magical Power Of A Name.” Forbes [Online]. Available: https://www.forbes.com/sites/alastairdryburgh/2017/03/22/growth-stories-the-magical-power-of-a-name/?sh=7de542fd6db9

APPENDIX

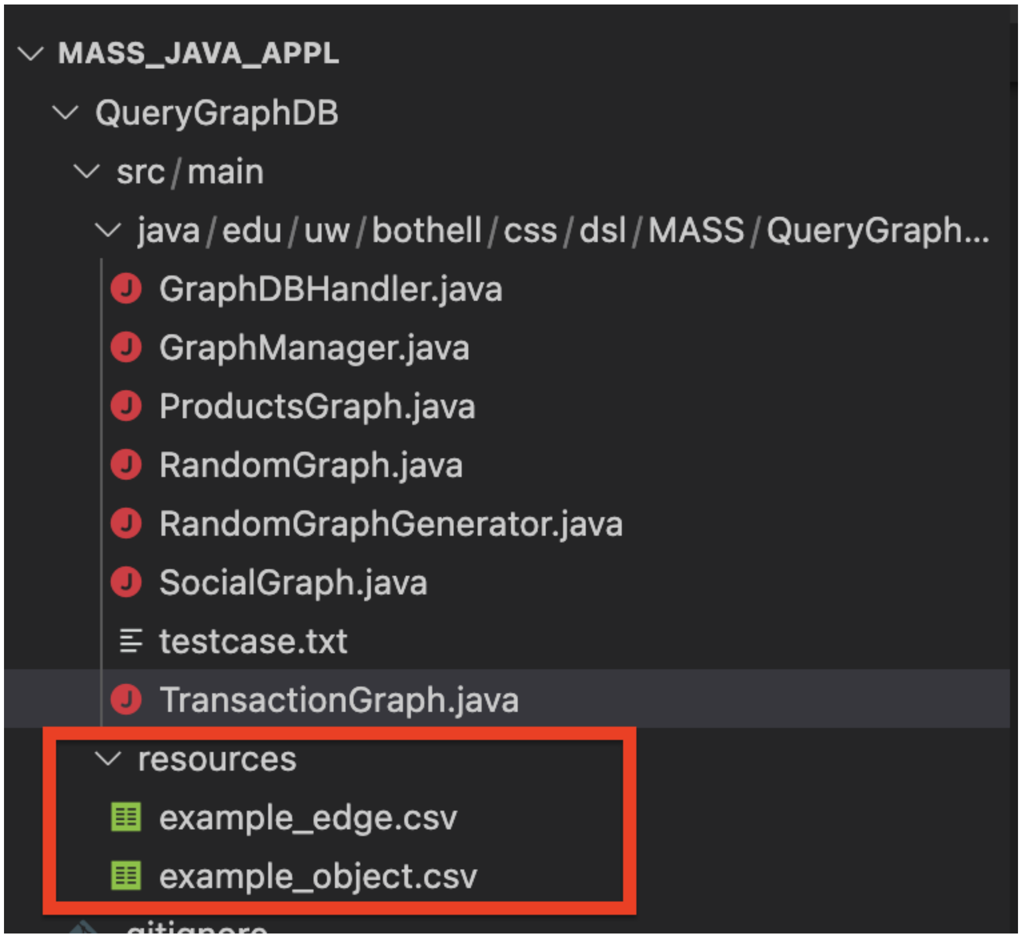
You can find the graph datasets in the dslabs google drive Work > Michelle Dea Graph Database Benchmarking:

https://drive.google.com/drive/u/2/folders/1MBTIGkaIMWnkIs21Qb4XrwC888REYCuI

Code can be found on the dslab bitbucket under the dea_develop branch

To run the MASS benchmarking program:

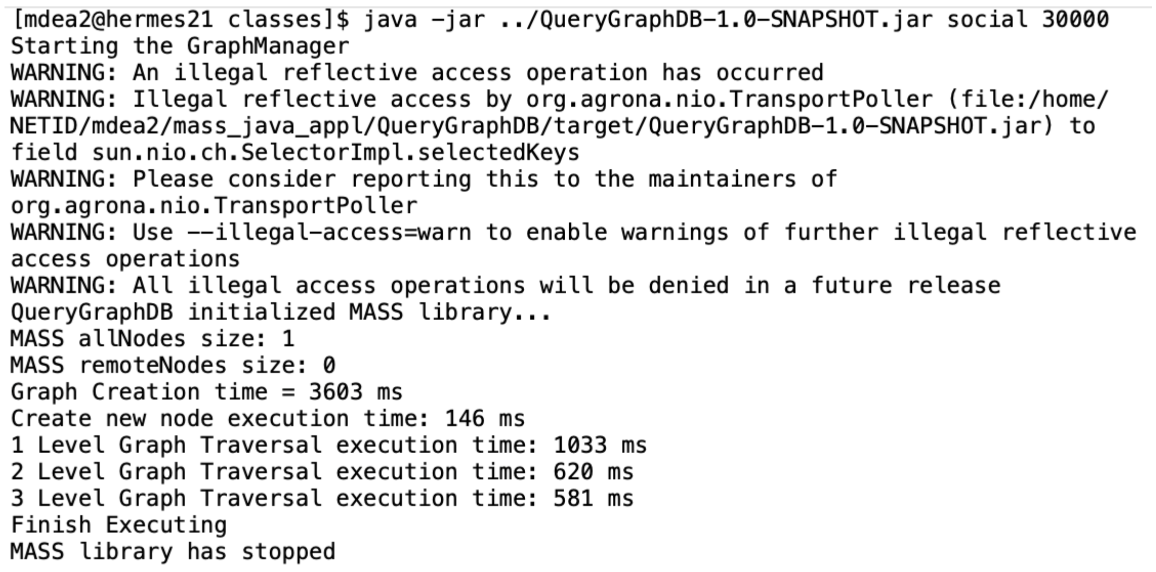
1. Add the data files to QueryGraphDB / src/main/resources



```
MASS_JAVA_APPL
  QueryGraphDB
    src / main
      java / edu / uw / bothell / css / dsl / MASS / QueryGraph...
        GraphDBHandler.java
        GraphManager.java
        ProductsGraph.java
        RandomGraph.java
        RandomGraphGenerator.java
        SocialGraph.java
        testcase.txt
        TransactionGraph.java
      resources
        example_edge.csv
        example_object.csv
```

2. When executing the jar file, specify the type and size of graph you want

a. For an example in mass_java_appl > QueryGraphDB > build_run.sh



```
[mdea2@hermes21 classes]$ java -jar ../QueryGraphDB-1.0-SNAPSHOT.jar social 30000
Starting the GraphManager
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.agrona.nio.TransportPoller (file:/home/
NETID/mdea2/mass_java_appl/QueryGraphDB/target/QueryGraphDB-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
QueryGraphDB initialized MASS library...
MASS allNodes size: 1
MASS remoteNodes size: 0
Graph Creation time = 3603 ms
Create new node execution time: 146 ms
1 Level Graph Traversal execution time: 1033 ms
2 Level Graph Traversal execution time: 620 ms
3 Level Graph Traversal execution time: 581 ms
Finish Executing
MASS library has stopped
```

3. When generating larger graphs, if you feel it is taking a while, you can add a print statement in the graph specific java file to see the edges being created – this part is what takes the longest. Ex: You can add these print statements to between line 197 and 198 in SocialGraph.java

   System.out.println("social vertexList size = " + vertexList.size());

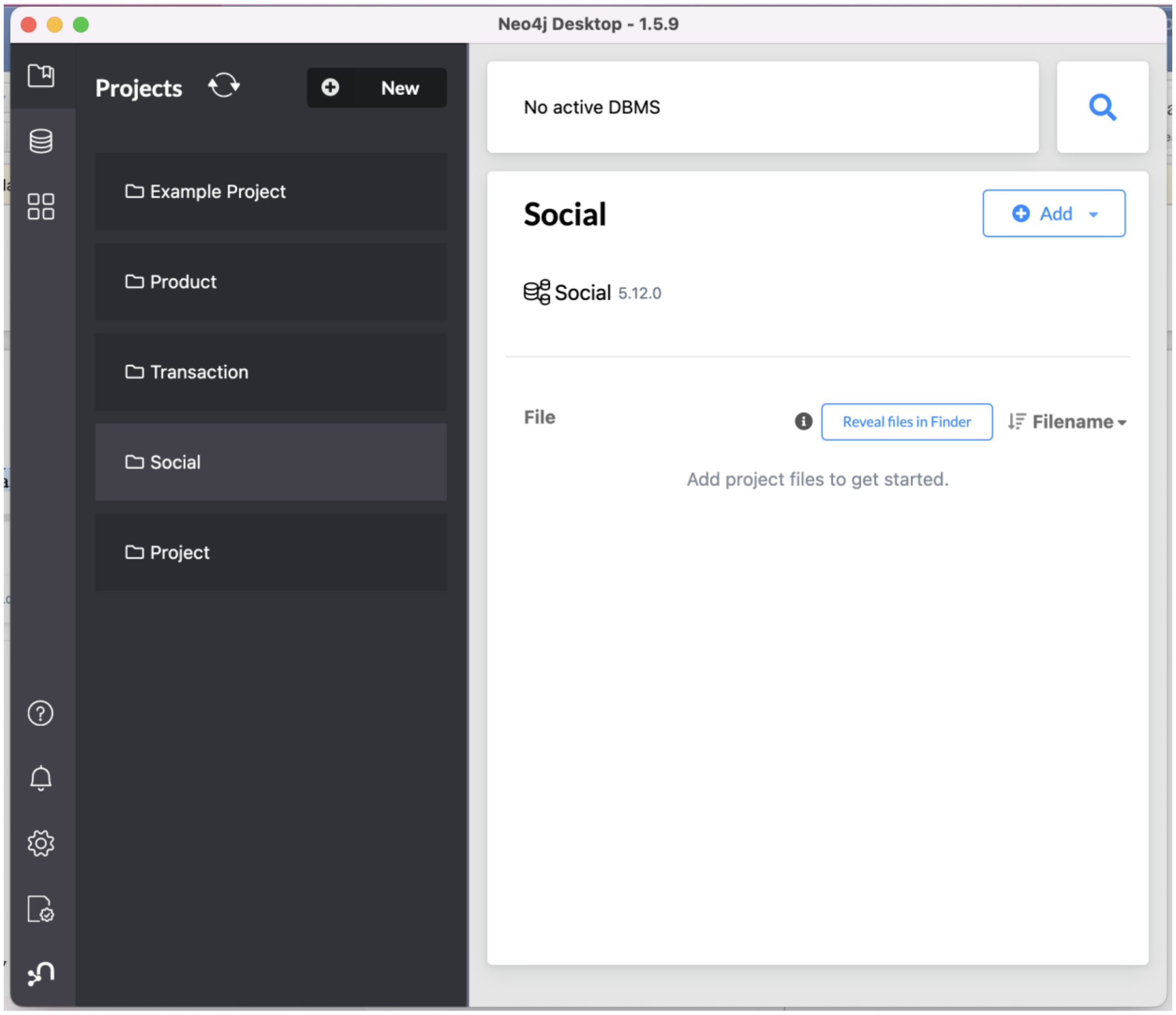
   System.out.println("social edgePairs size = " + edgePairs.size());

4. Execution times will be printed to console

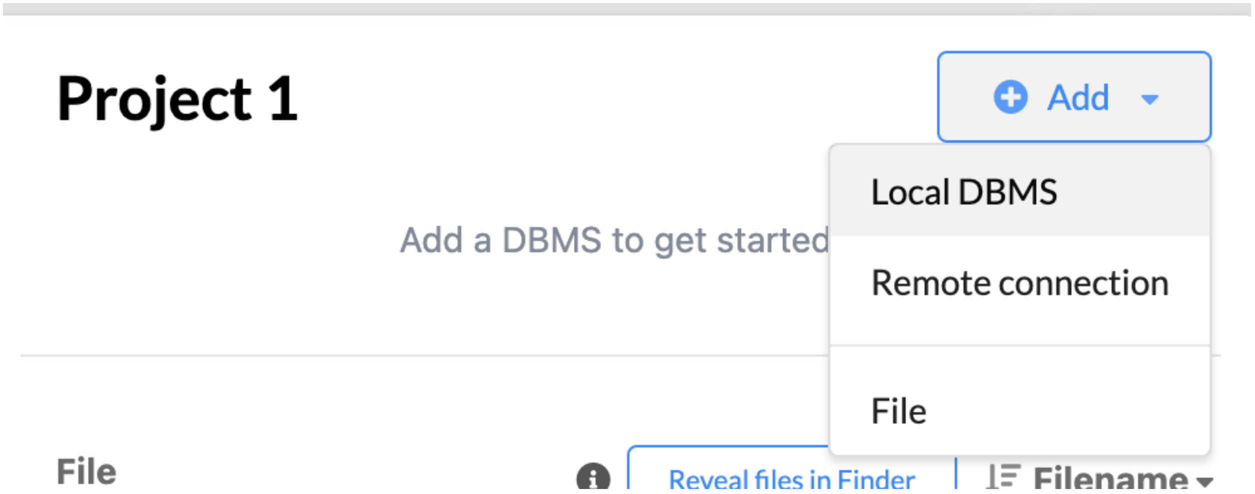
Neo4j:

Once you have downloaded Neo4j Desktop it is time to create your database and import data.

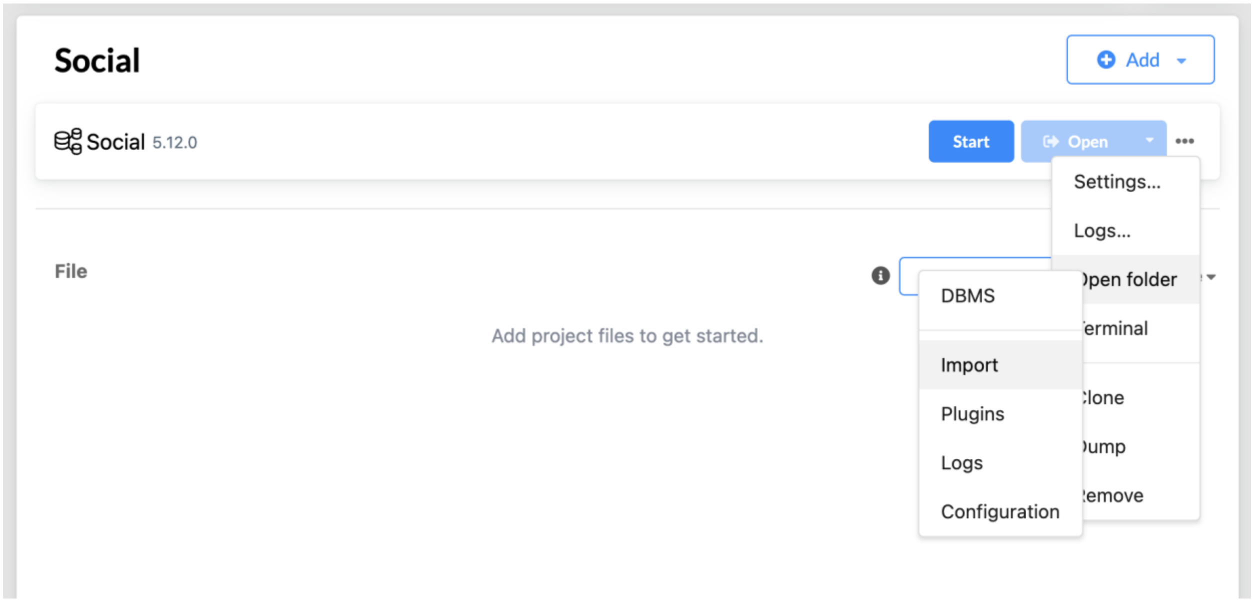
Open Neo4j Desktop:



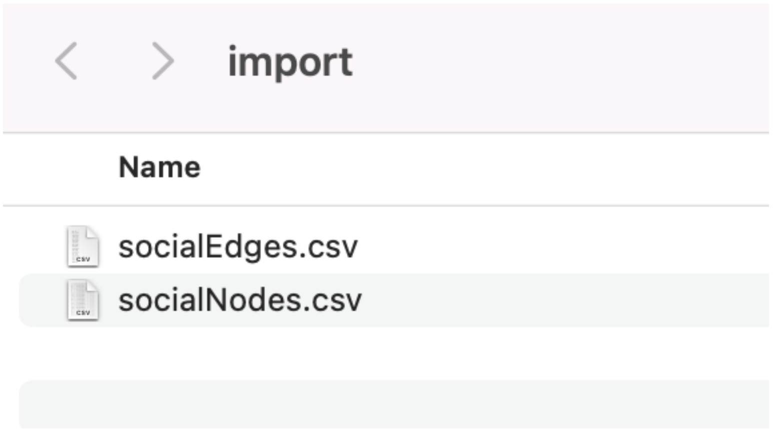
Create a new project using the New button in the top left > Rename your project and use the Add button to add a local DBMS



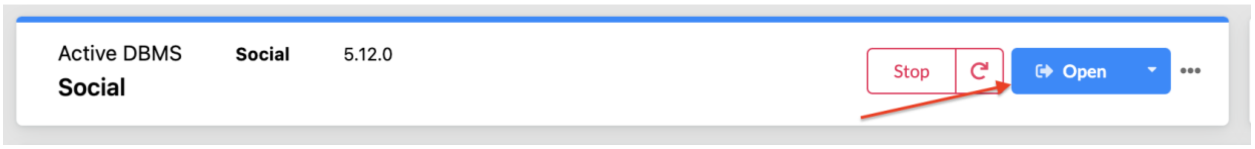
To import data, click on the three dots next to Open > Open Folder > Import



Copy your csv files that you want to import into the import folder:


import
Name
socialEdges.csv
socialNodes.csv

Now you will start your database and then open the Neo4j Browser:



Once in the browser you can run the following commands to generate your graph:

LOAD CSV WITH HEADERS FROM 'file:///socialNodes.csv' AS row

CREATE(s:Social {id:row.numeric_id})

set s += row

LOAD CSV WITH HEADERS FROM 'file:///socialEdges.csv' as row

match (s:Social {id: row.from})

match (s2: Social {id: row.to})

create (s)-[rel:FOLLOWS]->(s2);

For Cypher queries, see the capstone queries excel sheet in the google drive folder:

https://drive.google.com/drive/u/2/folders/1MBTIGkaIMWnkIs21Qb4XrwC888REYCuI

For ArangoDB:

Download docker, you will be using a docker image:

```
docker pull arangodb:3.10
```

```
docker run \
 -e ARANGO_ROOT_PASSWORD=password \
 -p 8529:8529 \
 -v /arangodb:/var/lib/arangodb3 \
 -v /arangodb/logs/:/var/log/arangodb3 \
 --restart always \
 --name arangodb \
 -d arangodb:3.10 \
 --log.level warning \
```

localhost:8529

user:root

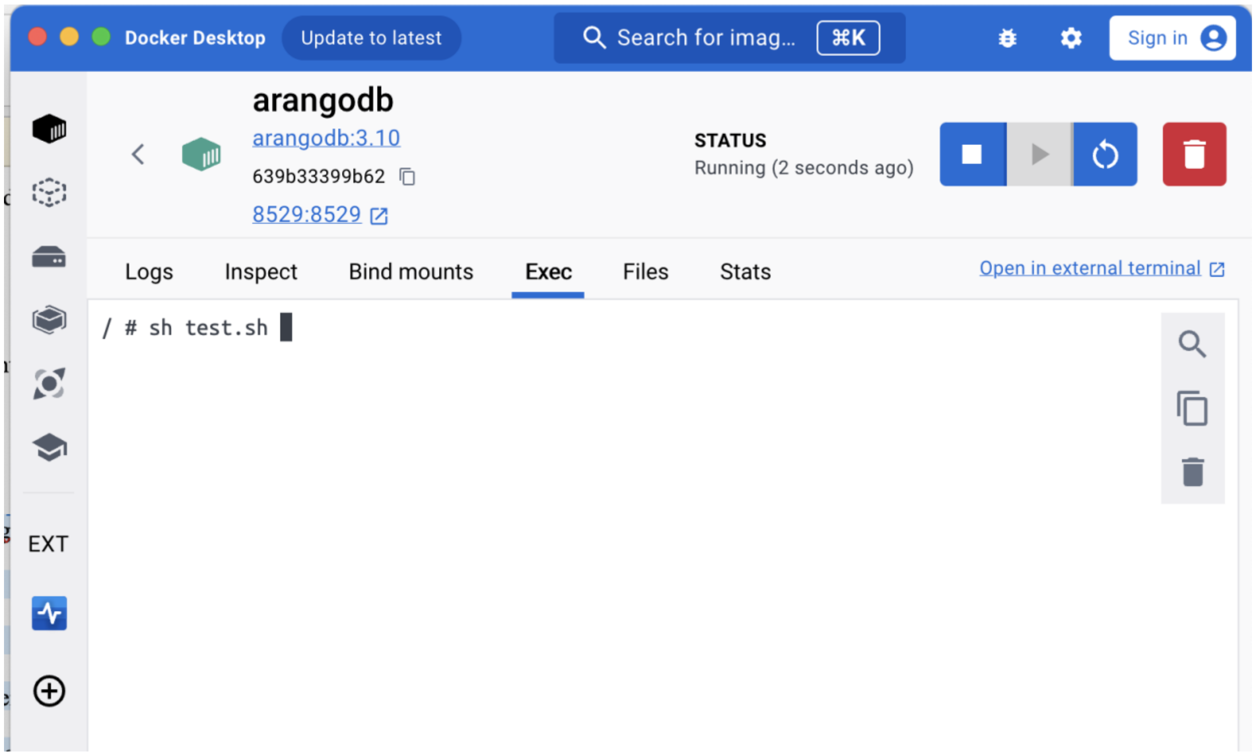
password: whatever password you want

- Note please update the password in the arango scripts once you have decided on your password

After you have the csv files, you will need to copy them to your docker container:

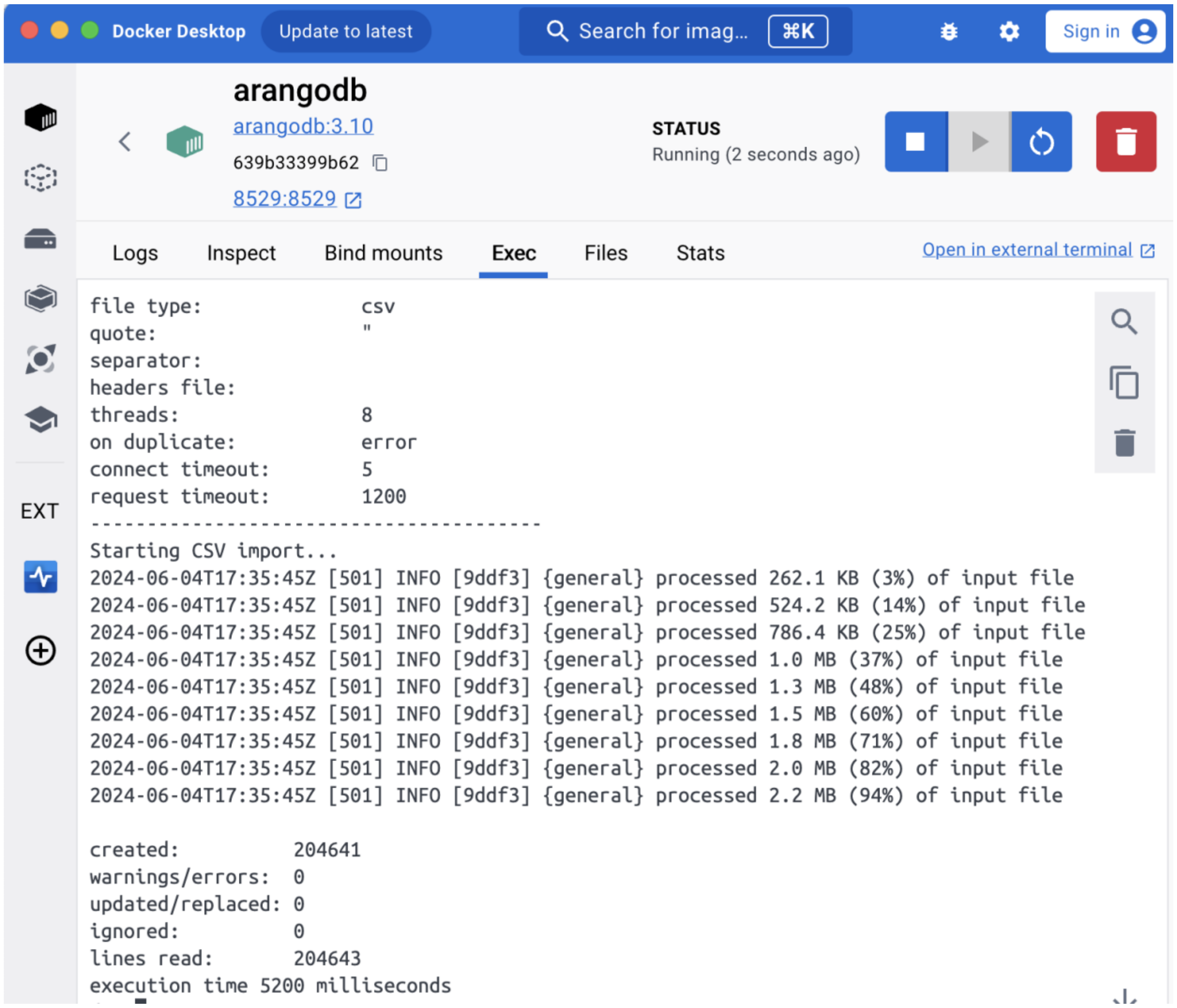
docker cp edges.csv <CONTAINERID>:/edges.csv

Once you have started the container, you can now use the appropriate arango script to import the data into collections. The scripts are labeled type_test.sh. Here is an example of how to run the shell script:



```
/ # sh test.sh
```

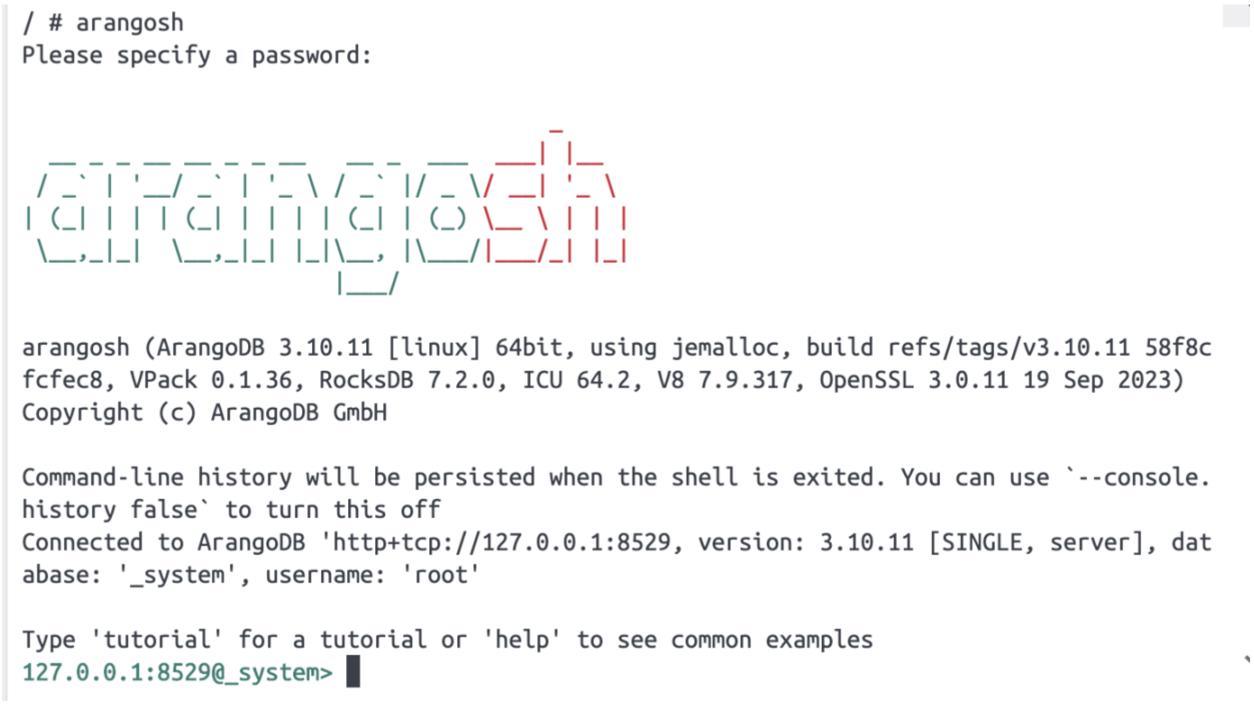
Execution time for loading data is printed to console:



```
file type:              csv
quote:                  "
separator:
headers file:
threads:                8
on duplicate:           error
connect timeout:        5
request timeout:        1200
----------------------------------------
Starting CSV import...
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 262.1 KB (3%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 524.2 KB (14%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 786.4 KB (25%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 1.0 MB (37%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 1.3 MB (48%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 1.5 MB (60%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 1.8 MB (71%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 2.0 MB (82%) of input file
2024-06-04T17:35:45Z [501] INFO [9ddf3] {general} processed 2.2 MB (94%) of input file

created:             204641
warnings/errors:     0
updated/replaced:    0
ignored:             0
lines read:          204643
execution time 5200 milliseconds
```

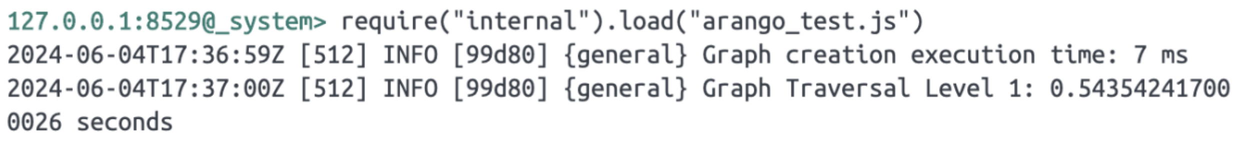
Once you have run the shell script, type arangosh into the terminal, this opens arango_shell. Now update arango_test.js with the queries and graph name you want to run the queries on. Copy arango_test.js into your docker container.

```
/ # arangosh
```



```
/ # arangosh
Please specify a password:

                                    _
  __ _ _ __ __ _ _ __   __ _  ___ ___| |__
 / _` | '__/ _` | '_ \ / _` |/ _ \/ __| '_ \
| (_| | | | (_| | | | | (_| | (_) \__ \ | | |
 \__,_|_|  \__,_|_| |_|\__, |\___/|___/_| |_|
                       |___/

arangosh (ArangoDB 3.10.11 [linux] 64bit, using jemalloc, build refs/tags/v3.10.11 58f8c
fcfec8, VPack 0.1.36, RocksDB 7.2.0, ICU 64.2, V8 7.9.317, OpenSSL 3.0.11 19 Sep 2023)
Copyright (c) ArangoDB GmbH

Command-line history will be persisted when the shell is exited. You can use `--console.
history false` to turn this off
Connected to ArangoDB 'http+tcp://127.0.0.1:8529, version: 3.10.11 [SINGLE, server], dat
abase: '_system', username: 'root'

Type 'tutorial' for a tutorial or 'help' to see common examples
127.0.0.1:8529@_system>
```

To run the JavaScript file from arangosh use: require("internal").load("arango_test.js")



```
127.0.0.1:8529@_system> require("internal").load("arango_test.js")
2024-06-04T17:36:59Z [512] INFO [99d80] {general} Graph creation execution time: 7 ms
2024-06-04T17:37:00Z [512] INFO [99d80] {general} Graph Traversal Level 1: 0.54354241700
0026 seconds
```

Execution times will be printed to console