Toward Implementing an Agent-based Distributed Graph Database System

Yuan Ma, Michelle Dea, Lilian Cao, and Munehiro Fukuda
Computing and Software Systems
University of Washington Bothell
Bothell, WA 98011
{yuanma, mdea2, sycao, mfukuda}@uw.edu

***Abstract*—**
Graph database (DB) systems are increasing their popularity in big-data analysis and machine learning particularly in the areas of e-commerce recommendation, fraud detection, and social-media analytics. Speed-up and spatial scalability of their DB transactions are pursued with various techniques such as index-free access to graph components in Neo4j, graph sharding over a cluster system in ArangoDB, and graph DB construction over distributed memory in AnzoGraph. However, these techniques have their respective challenges: difficulty in expanding an index-free graph over distributed memory, slow-down in accessing distributed disks, and a bottleneck incurred by repetitive master-to-worker distributions of query pipelines.

As a solution to these problems, we are applying multi-agent technologies to distributed graph DB construction: multiple user processes over a cluster system maintain portion of distributed graph in their cache space; their cache contents are synchronized through a software-snooped write-back and write-update protocol; and a DB user from any cluster node dispatches an agent that handles an independent graph query through navigating over distributed graph. To follow current trends in graph DB standardization, we adopt the Cypher language whose queries are translated into agent code.

This paper presents a new distributed hash-map implementation and its application to our graph DB system; differentiates it from Hazelcast from the viewpoints of its memory coherency and access speed; describes our translator that generates agent code from Cypher queries; and examines graph DB creation and traversal performance of agents in comparison with Neo4j and ArangoDB.

***Index Terms*—distributed shared graph, distributed hash map, parallel graph computing, multi-agent systems, graph database systems**

I. Introduction

Emergent uses of graph database (DB) systems are being highlighted for big-data analysis and graph-based machine learning particularly in financial technology, marketing/retail, and biological/health-care sciences [1], [2]. Graph DB has been ranked as one of the fastest growing categories since 2013 [3]. Many existing and multi-model DB systems have participated in the openCypher community[1] to accept Cypher graph queries. Speed-up and spatial scalability of their DB transactions are pursued with various implementation techniques such as index-free access to graph components in Neo4j[2], graph sharding over a cluster of distributed disks in ArangoDB[3], and graph DB construction over distributed memory in AnzoGraph[4]. However, these techniques have their respective challenges in distributed graph computing. Index-free graph traversals need a complex naming resolution of remote vertices. Distributed disk accesses need an efficient disk-cache mechanism. A master-worker model (taken by AnzoGraph) may have a transactional bottleneck at the master node that repeats dispatching each of query pipelines to the corresponding worker.

As a solution to these problems, we are applying multi-agent technologies to distributed graph DB construction. Our research motivation is based on some findings of agents' potential to analyze distributed data structures [4]. In contrast to data streaming that is optimized to keep processing input data with parallel and multithreaded compute instances, (e.g., lambda expressions), agent-based approach sometimes works better on finding topological attributes of a distributed data structure, in particular graph by dispatching many agents over the structure and having them collaborate toward a solution [5]. Needless to say, agents do not always perform faster on parallel graph analysis. Entire graph traversals and statistical analyses such as biological network motif identifications cause an exponential increase of agent population, thus wasting memory space [6]. However, limited graph traversals such as triangle count of a social network [7] moderate rapid agent propagation, (e.g., three edge traversals in triangle count), which can complete agent-based computation much quicker. We expect that most graph DB queries are centered around up to several edge traversals, each starting from different vertices. (An example is finding the common ancestor of different user-associated vertices.)

With this motivation, our research aims to address the above-mentioned challenges of graph DB systems by deploying agents as DB queries over a distributed in-memory graph structure. We use our multi-agent spatial simulation (MASS) library[5] as an agent execution platform. MASS distinguishes two classes: Agents and Places. The former populates a collection of Agent instances with navigational and behavioral autonomy, whereas the latter creates a distributed

[1] https://opencypher.org
[2] https://neo4j.com/
[3] https://arangodb.com/
[4] https://cambridgesemantics.com/anzograph/
[5] http://depts.washington.edu/dslab/MASS/

array of Place instances over a cluster system. Our agent-based graph DB system takes the following two approaches: (1) re-implementing MASS Places as GraphPlaces that makes a graph shared from and cached by multiple cluster-computing users and (2) extending MASS Agent to GraphAgent that navigates agents over GraphPlaces in response to Cypher queries. While the project is halfway through to its ultimate goal toward a construction of fault-tolerant and multi-user DB system with graph data-science features, this paper demonstrates two technical contributions to big graphs. One is an implementation technique of high-performance graph structure with a distributed hash map. Multiple user processes consistently maintain portion of shared graph in their cache space, using a software-snooped write-back and write-update protocol. The other is a cypher query translation into agent code that handles an entire query of multiple pipelines with an agent and its spawned descendants. This technique allows each user to submit an independent query agent from any cluster computing node, thus eliminating the master node.

The rest of the paper is organized as follows: Section II differentiates our agent-based graph DB construction from graph streaming, distributed cache, and conventional graph DB systems; Section III gives technical details of the current MASS GraphPlaces and GraphAgent implementation; Section IV demonstrates the promising performance of our graph DB system in comparison with Hazelcast, Neo4j, and ArangoDB; Section V concludes the discussion as identifying our next work.

II. Related Work

This section reviews the current trend of distributed graph DB systems in the following three categories: graph streaming, distributed cache, and parallelization of conventional graph DB systems.

A. Graph Streaming

One straightforward attempt is to apply data-streaming tools to big-graph computing. Running on top of Hadoop[6], Apache Giraph [8] solves graph problems using iterative MapReduce. Gradoop [9] is based on Hadoop as well to stream and process graph data with Flink[7]. CAPS [10] not only uses Spark SQL but also accepts Cypher queries to conduct graph analysis. However, these graph-streaming tools, in response to each graph query, focus on performing a batched machine-learning task on an entire graph data set such as Spark RDD rather than handling simultaneous queries from multiple users.

B. Distributed Cache

As a key-value hash table is used as a traditional graph implementation where a key indexes a vertex and its value represents an adjacency list, distributed hash tables are a natural extension to implementations of a distributed graph. Hazelcast[8] and Oracle Coherence [11] are used to facilitate multi-model distributed key-value DB and thus are potential to construct distributed graph DB. In fact, RedisGraph [12], Memcached [13], and Terracotta with Ehcache [14] take this implementation approach. However, their challenges are: RedisGraph is limited to single-machine uses, whereas Memcached and Ehcache maintain one copy of each vertex object at one computing node [15], thus no backup copy stored for fault tolerance purposes.

Since Hazelcast demonstrates its faster performance on benchmarks than Oracle Coherence [16], we consider it as a competitor against our graph DB system's infrastructure and therefore compare the basic graph-manipulating performance between these two in Section IV.

C. Graph Database Systems

Being recognized for 17 years, Neo4j speeds up graph transactions with two optimization techniques. One is index-free adjacency that directly references from a given graph vertex, (i.e., called node in Neo4j) to its neighbors. The other is anchoring a frequently referred graph node whose edges (called relationships in Neo4j) are minimal as compared to the other nodes, which reduces unnecessary graph traverses. While Neo4j covers distributed-computing, cloud-service environments as aura DB, the main objective is availability through its replica management rather than graph distribution.

Contrary to that, ArangoDB and AnzoGraph DB pursue graph scalability over a cluster system. They both take master-worker parallelization. ArangoDB divides a big graph into multiple shards, each maintained by a different computing node. SQL-based queries named AQL are first applied to the leader and thereafter copied to all followers for parallel transactions [17]. On the other hand, AnzoGraph DB focuses more on high-speed and parallel execution of multiple queries, using its distributed in-memory graph DB implementation. It accepts RDF queries named SPARQL at the leader computing node that parses each query, assembles multiple steps into a stream, and copies it to all workers. They take charge of processing those steps associated with their triples in parallel.

In Section IV, we compare Neo4j and ArangoDB with our agent-based graph DB in graph construction and graph traversal queries. The former measures disk-based versus in-memory DB construction, whereas the latter observes different strategies in graph traversals: those with index-free adjacency in Neo4j, those through hash accesses to document collections in ArangoDB, and those with agent propagation over a graph in our approach.

D. Challenges and Approach

In summary, graph streaming benefits one-time, batched machine-learning computation but does not always support multi-user, simultaneous graph DB transactions. Index-free graph structures realize the fastest search for graph components but have difficulty in extending to cluster computing. Query decomposition and distribution from the leader to followers cause serialization and bottleneck overheads inherent to the master-worker architecture.

[6]https://hadoop.apache.org
[7]https://flink.apache.org
[8]https://hazelcast.com/

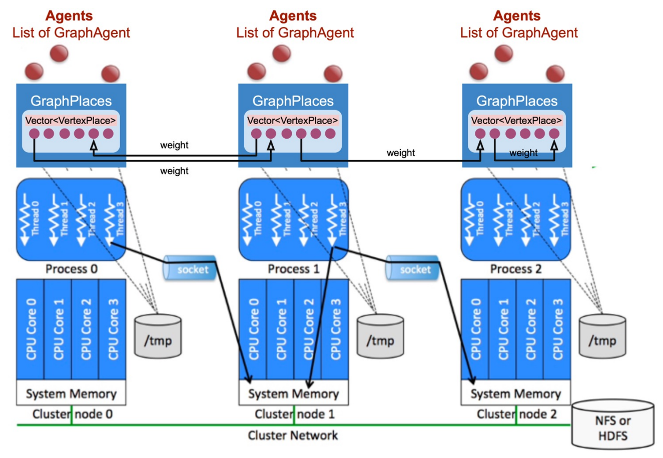


Fig. 1. The MASS library with graph-computing features

.

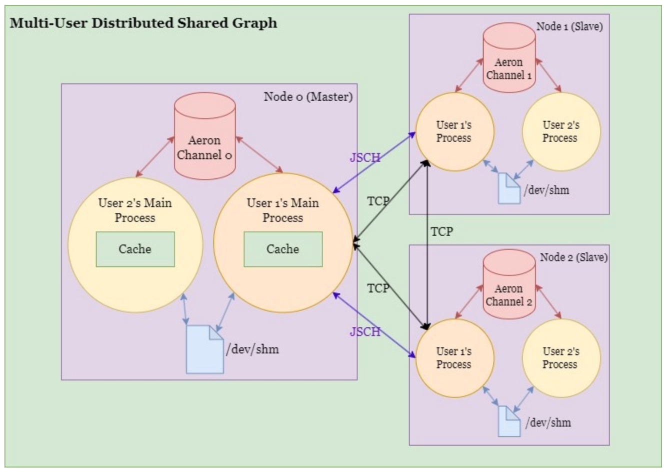


Fig. 2. A multi-user distributed shared graph

To address these challenges, we take the following two strategies. One allocates a different portion of a shared hash-based graph to each computing node's /dev/shm shared memory (or shared files in JVM), allows each user process to cache recently accessed graph vertices in its memory, and maintains their cache consistency with a software-controlled write-back write-update cache protocol. The other strategy translates each graph query into agent code and handles a whole query by navigating an agent and its descendents over a graph. In the next section, we explain the details of these implementation strategies.

III. Agent-Based Graph Database Infrastructure

After a brief explanation on the MASS library, we give technical discussions on GraphPlaces as a multi-user distributed shared graph and GraphAgent instances as query-executing agents.

A. *MASS Platform*

The MASS library lines up three language versions: Java, C++, and CUDA. For the parallelization purposes of agent-based models (ABMs), MASS instantiates array elements from the Places class, maps them over a cluster system or CUDA device memory, populates mobile objects from the Agents class, and observes their interactions. We use MASS Java for our agent-based DB project. As illustrated in Figure 1, MASS Java (simply referred to as MASS in the following discussions) alleviates array-based graph simulation and enables run-time graph modification by extending Places to GraphPlaces that instantiates vertex elements from the VertexPlace class and stores them in vectors, each maintained by a different computing node [7]. The library also removes manual operations on agent migration by extending Agent to the GraphAgent class that automates each agent's propagation along multiple graph edges [18].

While MASS demonstrates parallel performance of graph algorithms such as triangle count and disconnected components [7], it is intended for single-user graph analysis. To apply MASS to a multi-user graph DB infrastructure, we need to facilitate multi-user access to GraphPlaces, to enhance its availability and reliability features, and to have GraphAgent understand graph DB queries.

B. *Multi-User Distributed Shared Graph*

Our implementation takes the following three techniques: (1) mapping GraphPlaces onto /dev/shm that can be shared among multi-user processes and that remains until the system shut downs, (2) caching portion of GraphPlaces into each user process and maintaining their cache consistency at software level, and (3) allowing not only DB queries to identify a vertex with its property but also agents to traverse over vertices with their integer IDs.

As shown in Figure 2, each user process from its *main()* launches remote processes with JSCH[9] and establishes a TCP network among all these processes. More specifically, this initialization is achieved by an *MASS.init()* call. The *main()* function then calls the GraphPlaces constructor that has all the processes read an existing instance from the local /dev/shm space or maps an empty instance to this space. While the GraphPlaces implementation is based on a distributed key-value hash map, its manipulation is made through typical graph vertex and edge operations: *loadNNNFile(filePath)*, *getVertex(property)*, *addVertex(property)*, and *deleteVertex(property)*, each loading an NNN-formatted graph file data into GraphPlaces, identifying, adding, and deleting a vertex with a given property.

Although the easiest graph-sharing implementation is having each process directly read and write GraphPlaces on /dev/shm, Java's interface to /dev/shm is available only through file I/O operations in contrast to native languages such as C++ and C that map the space to shared memory. As direct I/O accesses to /dev/shm cannot outperform conventional distributed caches, (e.g., Hazelcast), GraphPlaces allows each user process

[9]http://www.jcraft.com/jsch/

to maintain recently accessed vertices into its private memory cache. This however results in cache inconsistency among user processes. A potential solution takes write-through and write-invalidation cache protocols, each forcing a process to directly modify the original copy of a vertex on /dev/shm and to invalidate all the copies that the other processes maintain. The main obstacle is that all must be done with file I/Os, which would cancel out effects obtained from vertex caching.

As a solution, we adopt a software-controlled write-back and write-update approach. During the course of execution, a user process broadcasts to the others of its write action to a specific vertex and the new value. We use Aeron[10] for FIFO-ordered reliable multicast messaging (See Figure 2). Note that such a write-update message should be delivered only within the same machine. This is because our agent-based approach dispatches agents to remote vertices instead of allowing a user process to cache remote copies. Assuming that a process can hold an entire copy of a sub-graph mapped to its local /dev/shm, our current implementation prompts a process to write back all its dirty copies to /dev/shm when it gets terminated.

A special attention must be paid when a user launches new MASS processes through *MASS.init()*. It is not sufficient to read vertex information only from the local /dev/shm, anticipating that newer vertex copies may remain in other processes. To solve this initialization problem, we use the finite state machine depicted in Figure 3. When a process invokes the GraphPlaces constructor, it reads the corresponding graph instance from /dev/shm into the private memory space, as making the state transition from INVALID to MAYBE_OLD. The process simultaneously broadcasts an initRequest message to all the others on the same machine. The receivers enclose their up-to-date vertex information in an initResponse message and send it back to the process in initialization. This process then updates its vertex copies with the first initResponse, as moving to the UP_TO_DATE state. All the later messages are simply discarded. Note that no initResponse messages leave the process in MAYBE_OLD. While the process stays in either MAYBE_OLD or UP_TO_DATE, it repeats broadcasting a write-update message to the other processes for coherence-maintenance purposes. Eventually, *MASS.finish()* moves all the processes to the INVALID state as writing back their cache to the local /dev/shm.

Graph DB in general indexes a vertex with a user-defined label and additional properties, which means that such a vertex ID should be of any data type. From this viewpoint, GraphPlaces refers to each vertex, (i.e., a VertexPlace object), using a Java Object ID. However, when agents migrate over a cluster system, they need to uniquely identify their destination vertices. Therefore, we should map a machine-local Java Object ID to a system-wide integer ID. As shown in Figure 4, MASS internally uses a system-wide distributed hash map together with machine-local vectors, where a Java Object ID, (e.g., the one obtained from a string) is mapped to a system-

[10]https://aeron.io

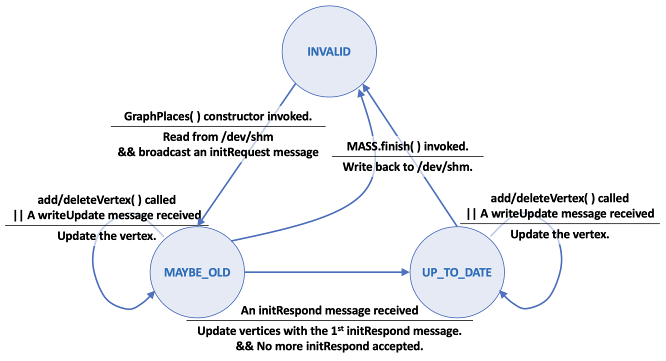

INVALID
GraphPlaces( ) constructor invoked.
Read from /dev/shm
&& broadcast an initRequest message
MASS.finish( ) invoked.
Write back to /dev/shm.
add/deleteVertex( ) called
|| A writeUpdate message received
Update the vertex.
add/deleteVertex( ) called
|| A writeUpdate message received
Update the vertex.
MAYBE_OLD
UP_TO_DATE
An initRespond message received
Update vertices with the 1st initRespond message.
&& No more initRespond accepted.

Fig. 3. A hash-map coherency protocol

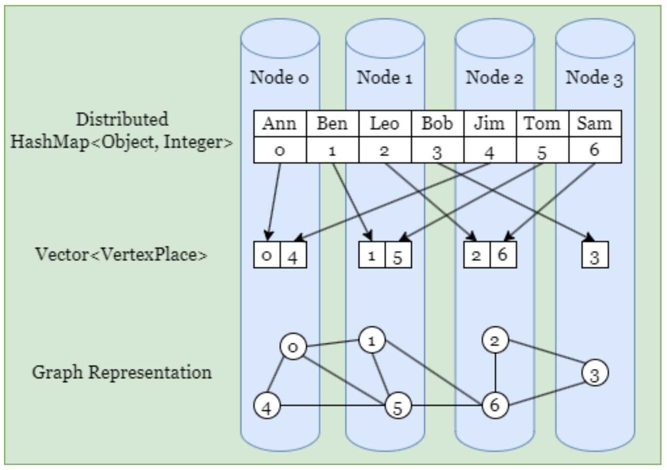


Fig. 4. A graph construction from a hash-map

wide integer ID that is then used to index a VertexPlace object, using Formula 1:

$$Index\ in\ Vector = \lfloor \frac{Integer\ ID}{\#Computing\ Node} \rfloor \quad (1)$$

This two-step indexing mechanism allows agents to not only jump to a VertexPlace with a user-provided label but also compute graph algorithms only using integer IDs. For instance, the number of triangles in a social network is counted by walking agents twice from the current VertexPlace to its neighbors with a lower integer ID and having their third walk search for an edge back to their original VertexPlace. Figure 5 illustrates that two agents successfully identify triangles along $5 \rightarrow 4 \rightarrow 2$ and $4 \rightarrow 3 \rightarrow 2$ respectively, whereas another agent that traverses along $3 \rightarrow 2 \rightarrow 1$ fails in finding any edge from vertex 1 back to 3. The main merit of this agent-based triangle count is preventing double counts of the same triangle.

C. Agent-Based Graph Query

To follow the concept of property graphs widely used in graph DB systems, we further extend GraphPlaces to PropertyGraphPlaces that instantiates vertices from PropertyVertex-Place, an extension of VertexPlace and populates agents from

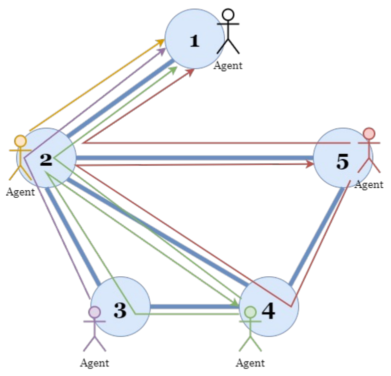


Fig. 5. Agent-based triangle count algorithm

TABLE I
AN ENHANCED PROPERTY GRAPH STRUCTURE LEVERAGING MASS

MASS Basic Library		MASS Graph Library		MASS Property Graph Library
Places	⇐	GraphPlaces	⇐	PropertyGraphPlaces
Contains ⇓	Extends	Contains ⇓	Extends	Contains ⇓
Place	⇐	VertexPlace	⇐	PropertyVertexPlace
Agents				
Contains ⇓	Extends		Extends	
Agent	⇐	GraphAgent	⇐	PropertyGraphAgent

PropertyGraphAgent extended from the GraphAgent class. Table I summarizes our enhanced property graph structure leveraging the MASS library.

PropertyVertexPlace not only stores the original VertexPlace data members - an integer ID, a label, and an integer ID list of neighbors but also includes: vertex properties, outgoing relationships, (i.e. edges to neighbors), and incoming relationships, (i.e., edges from neighbors). On the other hand, PropertyGraphAgent introduces a new data member named pathResult. It is used for agents to collect information about the paths they traversed.

Figure 6 illustrates an example of PropertyGraphPlaces that instantiates seven PropertyVertexPlace vertices and populates a PropertyGraphAgent instance on each vertex. These agents get started with an empty pathResult list. They build up their pathResult as navigating over the graph, clone themselves when encountering branches, and dispatch each clone to a different edge with a duplicated pathResult.

Our graph DB system uses the Cypher query language. Each Cypher query text eventually populates a group of new instances from the PropertyGraphAgent class and deploys them over PropertyGraphPlaces. This agent execution is planned through the following Cypher processing pipeline:

1) **Parsing Phase:** converts a Cypher query text into a parse tree, using ANTLR4 (ANother Tool for Language Recognition, version four) [19] and its grammar file tailored for Cypher, named Cypher.g4. In the parse tree, each node represents a given query's syntactic element such as keywords, identifiers, operators, or expressions.

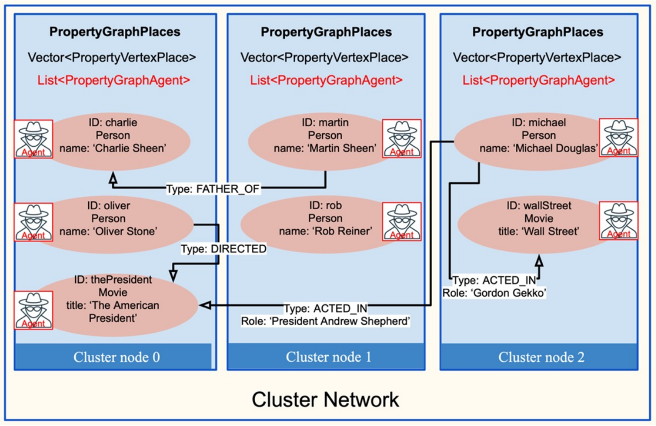


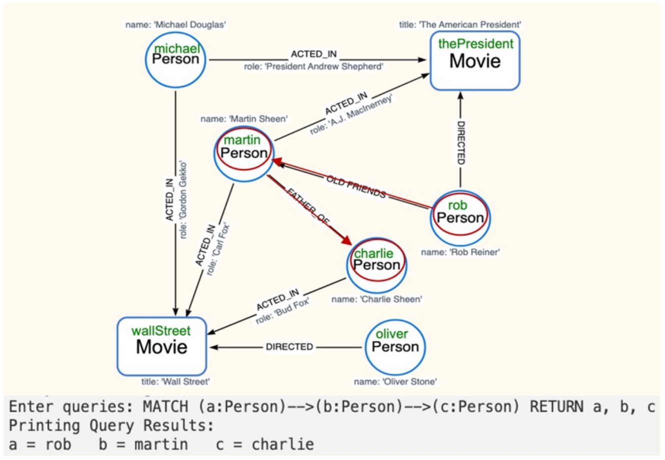
Fig. 6. A property graph representation for making a graph DB

2) **Translation Phase:** uses the openCypher Transpiler library[11] and its CypherVisitor interface for a parse-tree translation into AST (abstract syntax tree). Each AST tree node is transformed into one of the execution steps, each corresponding to a different Cypher operator such as CREATE, JOIN, MATCH, and RETURN.
3) **Planning Phase:** currently focuses on the MATCH, CREATE, and DELETE Cypher clauses as well as their execution plans. MATCH generates agent code to deploy agents over a graph. CREATE and DELETE are translated into *PropertyGraphPlaces.addVertex()* and *deleteVertex()*. The agent code consists of the PropertyGraphAgent constructor call and repetitive *callAll()* and *manageAll()* invocations. The latter two functions are MASS Agents' base methods, each respectively planning a next agent behavior and committing agent cloning, termination, and migration.
4) **Execution Phase:** repeats the following agent-deployment cycle until all the agents finish their graph traversals. Each cycle gets started with *callAll()* to have all agents check if their current vertices satisfy a given MATCH query, followed by three *manageAll()* invocations: the first call spawns the same number of children as that of relationships emanating from the current vertex; the second call kills those whose relationship doesn't match the query; and the third call migrates each agent along its assigned relationship.

Figure 7 gives an example of agent deployment that executes a Cypher query: MATCH (A:PERSON) $\rightarrow$ (B:PERSON) $\rightarrow$ (C:PERSON). Agents populated at vertices with the Person property survive for further graph navigation but only one of them completes a successful traverse on A = ROB, B = MARTIN, and C = CHARLIE.

Note that a complete explanation on our query-to-agent translation is given in [20].

[11]https://github.com/microsoft/openCypherTranspiler



```
Enter queries: MATCH (a:Person)-->(b:Person)-->(c:Person) RETURN a, b, c
Printing Query Results:
a = rob    b = martin    c = charlie
```

Fig. 7. Agent migration over a property graph

IV. Evaluation

For the purpose of forecasting the promising performance of our agent-based graph DB system, we focus on two benchmark tests. One checks what graph operations make MASS superior to Hazelcast as both implementations are based on distributed cache. The other explores the depth of graph DB traversals for benefiting MASS parallelization as compared to Neo4j and ArangoDB.

Our performance evaluation is conducted over a cluster of 20 computing nodes, all connected to a 1Gbps LAN. Their configurations are summarized in Table II. Among them, 12 computing nodes are VMs, each with four CPU cores and 16GB memory, virtualized from Xeon Gold 6130.

TABLE II
System configuration for performance evaluation

# computing nodes	CPU model	machine type	# CPU cores	memory
12	Xeon 6130 @ 2.10GHz	virtual	4	16GB
4	Xeon E5410 @ 2.33GHz	physical	4	16GB
3	Xeon E5150 @ 2.66GHz	physical	4	16GB
1	Xeon 5220R @ 2.2GHz	virtual	4	16GB

A. *Comparison with Hazelcast in Basic Graph Operations*

Our performance comparison between MASS and Hazelcast covers (1) vertex manipulations, (2) attribute retrievals, and (3) graph loading and computing.

1) Vertex additions and retrievals: are measured for 3,000 versus 5,000 operations on top of one to four computing nodes. The measurements are summarized in Figures 8 and 9 respectively. Regardless of the number of operations, we observe the same performance trend in MASS and Hazelcast.

For vertex additions, while Hazelcast performs twice faster than MASS for single-machine operations, it spends more than twice time as MASS does with four computing nodes. On a single machine, Hazelcast simply creates new vertices in its single process, whereas a MASS process needs to broadcast

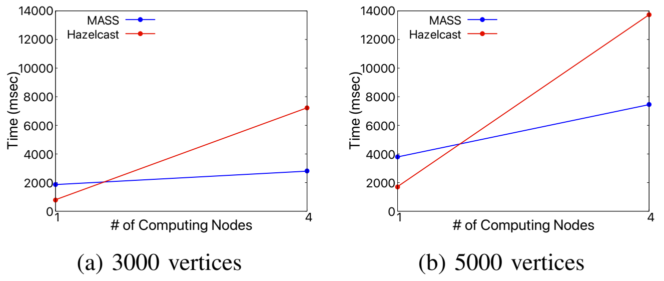


Fig. 8. Vertex addition performance with one to four computing nodes

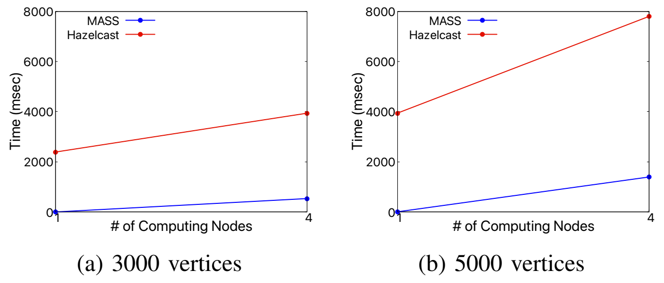


Fig. 9. Vertex retrieval performance with one to four computing nodes

these operations as it has no knowledge of other processes in existence. On the other hand, over four computing nodes, Hazelcast not only identifies the owner of a new vertex but also adds its backup to a different computing node. MASS vertex addition is basically the same as its single-node execution except choosing the computing node that should own a new vertex.

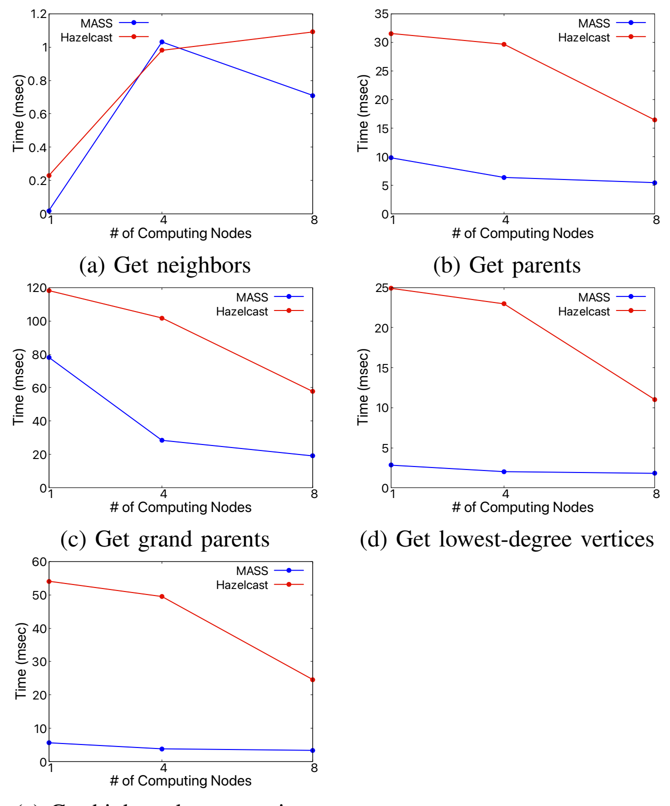
For vertex retrievals, MASS outperforms Hazelcast regardless of the number of computing nodes. This is because MASS simply retrieves a given vertex from memory without invoking an Aeron broadcast while Hazelcast needs to identify the owner of its IMap data.

2) Graph attribute retrievals: consider five attributes: (1) neighbors of each vertex, (2) parents of each vertex, (3) grand parents of each vertex, (4) vertices with the lowest degree, and (5) those with the highest degree, each measured with one to eight computing nodes. Figures 10-(a) through 10-(e) compare MASS and Hazelcast performance for retrieving these attributes respectively, from a graph with 100,000 vertices and 3,999,366 edges.

MASS outperforms Hazelcast in all the five benchmarks. This is because MASS GraphPlaces has the *callAll()* method that makes a simultaneous function call at all VertexPlace objects and retrieves their values through one round-trip communication between *main()* and all remote processes.

3) Graph loading and computing: constructs a graph from a given file and counts the number of triangles in that graph. Three files are prepared, each including a different size of graph, as listed in Table III.

Figures 11-(a) through 11-(c) compare MASS and Hazelcast when reading these three files and constructing the corresponding graphs over one to 20 computing nodes. The results



(a) Get neighbors (b) Get parents (c) Get grand parents (d) Get lowest-degree vertices (e) Get highest-degree vertices

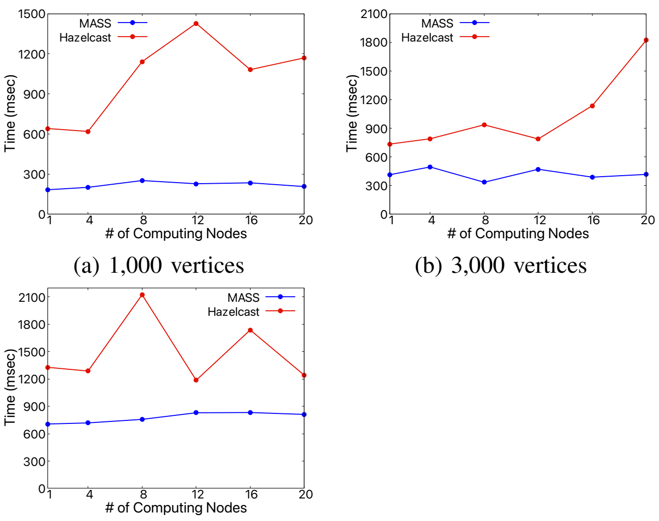
Fig. 10. Graph attribute retrieval performance with one to eight computing nodes

TABLE III
THREE VARIABLE SIZES OF GRAPHS USED FOR MEASURING THEIR CONSTRUCTION AND TRIANGLE-COUNT PERFORMANCE

files	# vertices	# edges	# triangles
file 1	1,000	93,480	165,138
file 2	3,000	293,804	192,146
file 3	10,000	989,990	200,053

demonstrate that no matter what size the graph is and how many computing nodes are used, MASS performs better than Hazelcast. This is due to the difference between their vertex-adding operations. Hazelcast needs to create original and backup copies of vertices, each mapped to a different computing node, whereas MASS processes write back all their vertices to /dev/shm upon their termination.

We picks up triangle count as a graph-computing benchmark. MASS uses the agent-based algorithm as illustrated in Figure 5. On the other hand, our Hazelcast version distributes a triple-nested loop over a cluster system, in support with Hazelcast's aggregator API. As shown in Listing 1, each computing node starts its triangle exploration from its assigned key-value pairs in the most outer loop (line 4). Each time it finds an edge going back to the original pair after two edge traversals (line 7), the computing node increments *triangles*, a distributed implementation of atomic integer (line 8). At the end, the number of triangles must be divided by 6 to exclude



(a) 1,000 vertices (b) 3,000 vertices (c) 10,000 vertices

Fig. 11. Graph construction over one to 20 computing nodes

duplicates (line 11).

Figure 12-(a) through 12-(c) show the parallel performance of MASS and Hazelcast when counting the number of triangles with one to 20 computing nodes.

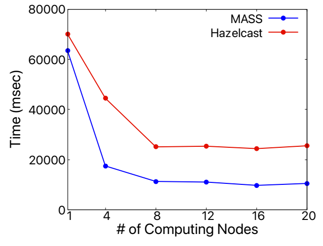
For triangle count on a single machine, MASS runs slightly faster than Hazelcast with a graph with 1,000 vertices since both systems do not consume too much memory space. However, as we increase the graph size, MASS slows down its performance and eventually results in an out-of-memory exception, due to its rapid growth of agent population.

For execution over multiple computing nodes, MASS outperforms Hazelcast as it is optimized to prevent agents from double-counting the same triangles. Needless to say, with cache support such as Eclipse UnifiedMap, Hazelcast will be able to run faster than MASS. From this viewpoint, remote caching rather than agent dispatching should be considered as one of our future plans for improving MASS performance.

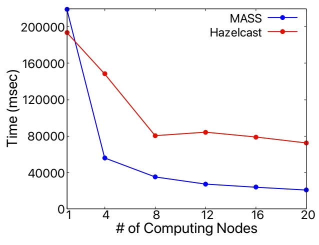
Listing 1. Triangle count with Hazelcast
```
1  int triangle_count( ) {
2    AtomicInteger triangles = new AtomicInteger( 0 );
3    public void accumulate( ) {
4      for v = 1 .. n {
5        for each neighbor u of v {
6          for each neighbor w of u {
7            if wv forms an edge {
8              triangles.incrementAndGet( );
9      } } } }
10   }
11   return triangles.get() / 6.
12 }
```

B. Comparison with Neo4j and ArangoDB in Graph Database Queries

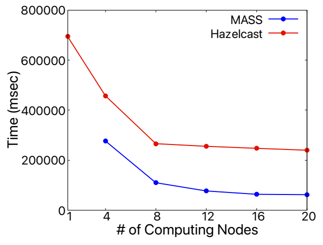
While our Cypher-to-agent translation supports only CREATE, DELETE, and MATCH clauses, thus still in middle of lining up all the keywords, this section intends to check



(a) 1,000 vertices



(b) 3,000 vertices



(c) 10,000 vertices

Fig. 12. Triangle count execution with one to 20 computing nodes

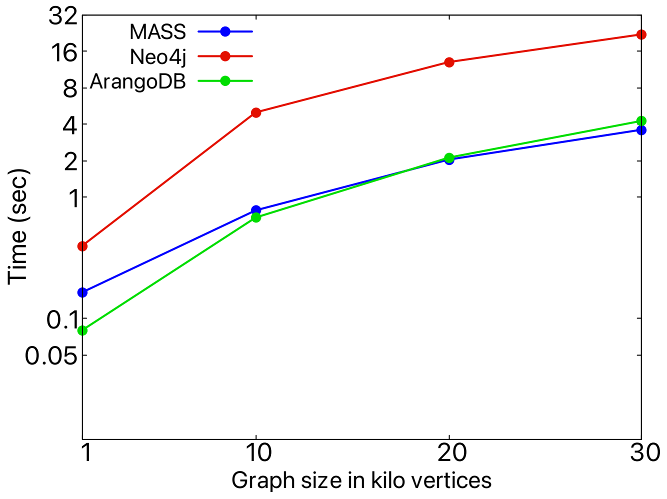


Fig. 13. Graph DB construction performance

MASS-based graph DB system's advantages in in-memory distributed graph construction and parallel deployments of agent-based MATCH queries.

To evaluate the spatial scalability of each graph DB system, we capture the time it takes to create a different size of graph via CSV importing, and the query execution time when traversing the graph from depth 1 to depth 3. We consider two cases of graph traversal queries: one repeats starting a query from a random vertex, whereas the other exhausts an entire graph DB with each query. As Neo4j and ArangoDB's free versions are restricted to single-machine uses, our DB comparison work is conducted on one of our cluster nodes.

We use the Twitch Social Network dataset with 168,114 vertices and 6,797,557 edges [21], and generate four random graphs of sizes 1K vertices and 257 edges, 10K vertices and 24K edges, 20K vertices and 92K edges, and 30K vertices and 204K edges. Our random graph generation is based on [22] that has been developed in biological network motif analysis for preserving the degree of each vertex.

Table IV shows the queries used for graph traversals. The queries are written in Cypher for MASS and Neo4j as well as in AQL for ArangoDB.

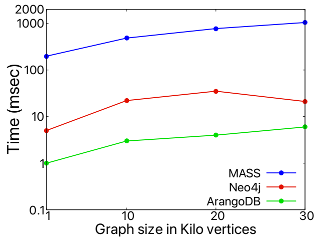
TABLE IV
QUERIES USED FOR GRAPH TRAVERSALS

Depth	Cypher	AQL
1	MATCH (a)→(b) RETURN b	FOR vertex in socialNodes FOR vertices, edges, paths = 1..1 OUTBOUND vertex GRAPH social RETURN vertices
2	MATCH (a)→() →(b) RETURN b	FOR vertex in socialNodes FOR vertices, edges, paths = 2..2 OUTBOUND vertex GRAPH social RETURN vertices
3	MATCH (a)→() → () →(b) RETURN b	FOR vertex in socialNodes FOR vertices, edges, paths = 3..3 OUTBOUND vertex GRAPH social RETURN vertices

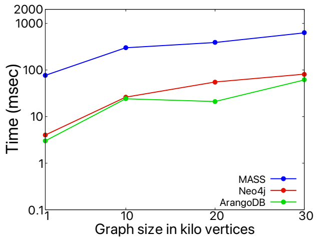
1) DB construction benchmark: measures the time it takes for each graph DB system to create graphs of different sizes. Figure 13 shows that, for larger graphs with 20K and 30K, MASS-based graph DB system outperforms both Neo4j and ArangoDB. Generally, Neo4j is the slowest for graph creation. This is likely due to the way we store vertex and edge data, (i.e, node and relationship in the following discussions whenever referring to Neo4j) 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 DB files [23]. In ArangoDB, data is also stored on the disk but as a JSON object referred to as a document [24]. Documents are organized into collections where vertices and edges are stored separately. MASS-based graph DB system stores vertices as a PropertyVertexPlace object, with two separate hash-maps 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.

2) Random DB-traversal benchmark: identifies the baseline performance for each DB system to traverse graphs of 1K, 10K, 20K, and 30K vertices. By starting from a random vertex, the data should not be available in cache or memory for the disk-based solutions, assuming that many users simultaneously send their independent queries, each irrelevant to others. This helps highlight the differences between disk-based (Neo4j and ArangoDB) and memory-based (MASS-graph DB system) query times.

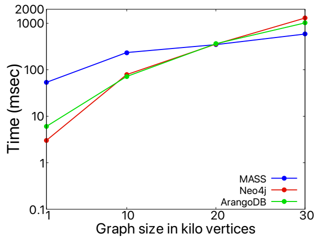
Figure 14 shows that MASS performs the worst for depth 1 and depth 2 traversals. For 20K and 30K graphs, at depth 3 traversal, we see that MASS outperforms both Neo4j and ArangoDB. This is likely due to both Neo4j and ArangoDB needing to access the disk to return data at depth 3 traversal. The Neo4j and ArangoDB execution times for depths 1 and 2 will serve as the single-machine performance goal for MASS as more enhancements and optimizations are made to the



(a) Depth 1

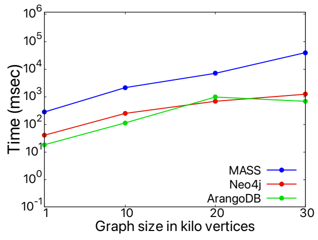


(b) Depth 2

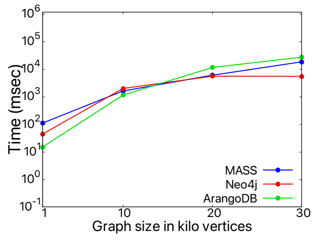


(c) Depth 3

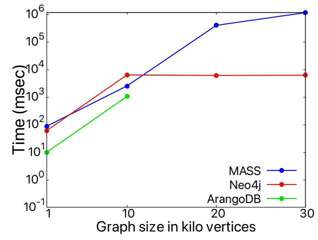
Fig. 14. Graph DB traversal depths 1-3 from random vertices



(a) Depth 1



(b) Depth 2



(c) Depth 3

Fig. 15. Exhaustive graph DB traversal depths 1-3

agents for graph traversal.

3) Entire DB-traversal benchmark: exhaustively checks all potential relationships with depth 1, 2, and 3 over an entire graph. We believe that this is one of the most intensive stress test while it is not realistic in practical DB queries.

Figures 15-(a) to 15-(c) compare the query execution time of MASS, Neo4j, and ArangoDB. As shown in Figure 15-(a), MASS rapidly increases its depth-1 traversal time as the graph size grows. This is because every single vertex needs to propagate an agent to all its neighbors. The figure also shows that ArangoDB is the most performant for depth 1 traversal. Figure 15-(b) however demonstrates that MASS Graph DB system is more performant than ArangoDB for depth-2 traversal. This is because the population of agents drastically gets decreased on their further graph navigation. Again in depth-3 traversals, as shown in Figure 15-(c), MASS outperforms ArangoDB 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.

In contrast to a relational DB where performance is usually dependent on the size of the tables, index-free adjacency 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 DB system's weakness in larger graph traversals is caused by agent propagation from all vertices. We further improves the MASS infrastructure to populate agents only at applicable vertices.

V. Conclusion

Motivated from our former experience in deploying agents to distributed data structures, this project aimed to apply agents to an implementation of in-memory distributed graph DB system. We developed a distributed key-value hash-map as our DB system infrastructure. To support both property-based vertex search and agent-based graph navigation, each computing node maintains a vector of graph IDs in integer. Multiple users share a graph on the /dev/shm directory but cache graph vertices with a write-back and write-update protocol. We confirmed that this approach outperforms Hazelcast. We have developed an ANTLR4-based translator to convert Cypher queries into agent code so that agents navigate over a graph DB to respond to their queries. Our comparison with Neo4j and ArangoDB demonstrated that MASS-based graph DB system performs competitively on random graph traversals with depth 3 over a graph with 30K vertices.

However, our preliminary measurements also indicated two challenges of our agent-based approach. One is that agent deployment could not always outperform Hazelcast if Hazelcast developers would tune up their graph algorithms with remote execution and local caching techniques. The other is that agent-based graph traversals with depth 1 or 2 cannot outperform Neo4j and ArangoDB at all, due to the current agent population and execution control.

Based on these observations, we keep pursuing our agent-based approach to distributed graph DB construction as working on the following five tasks: (1) to complete the translator to convert all Cypher queries to agent code, (2) to line up more graph DB benchmark queries, (3) to extend our performance comparisons to cluster versions of Neo4j, ArangoDB, and more graph DB systems such as AnzoGraph DB, (4) to improve MASS GraphPlaces, VertexPlace, and GraphAgent for their distribution, population, and control over a cluster system, and (5) to develop machine-learning agents in graph computing, including link predictions.

Acknowledgment

We greatly appreciate Mr. Michael Robinson, an undergraduate student at University of Washington Bothell for his development work on the graph-benchmark programs that we used for the performance measurements.

References

[1] Y. Tian, "The World of Graph Databases from An Industry Perspective," *ACM SIGMOD*, vol. 51, no. 4, pp. 60–67, January 2023.

[2] TigerGraph, "Featured Customer Success Stories: Why Our Customers Choose TigerGraph," Web accessed on: August 13, 2024. [Online]. Available: https://www.tigergraph.com/customers/, 2024.

[3] M. Wu, "Graph Database Market Overview," Nebula Graph, Blog accessed on: August 13, 2024. [Online]. Available: https://www.nebula-graph.io/posts/graph-database-market-overview, 2023.

[4] M. Fukuda, C. Gordon, U. Mert, and M. Sell, "Agent-Based Computational Framework for Distributed Analysis," *IEEE Computer*, vol. 53, no. 4, pp. 16–25, 2020.

[5] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," in *7th Int'l Workshop on BigGraphs'20*. IEEE, December 2020, pp. 2957–2966.

[6] S. Duraisamy, "Agent-Based Parallelization of Biological Network Motif Detection, MS Capstone White Paper," University of Washington Bothell, Tech. Rep., June 2020.

[7] Y. Hong and M. Fukuda, "Pipelining Graph Construction and Agent-based Computation over Distributed Memory," in *9th Int'l Workshop on BigGraphs'22*. IEEE, December 2022, pp. 4616–4624.

[8] C. Martella, D. Logothetis, and R. Shaposhnik, *Practical Graph Analytics with Apache Giraph*. Berkeley, CA: Apress, 2015.

[9] M. Junghanns, M. Kießling, N. Teichmann, K. Gómez, A. Petermann, and E. Rahm, "Declarative and distributed graph analytics with GRADOOP," in *Proceedings of the VLDB Endowment, Volume 11, Issue 12*, August 2018, pp. 2006–2009.

[10] S. Gupta, "Morpheus – Cypher for Spark," knóldus, Blog accessed on: August 14, 2024. [Online]. Available: https://blog.knoldus.com/morpheus-cypher-for-spark/, 2023.

[11] Oracle, "Oracle coherence," Web accessed on: August 15, 2024. [Online]. Available: https://www.oracle.com/java/coherence/.

[12] P. Cailliau, T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine, "RedisGraph GraphBLAS Enabled Graph Database," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops*. Rio de Janeiro, Brazil: IEEE, May 2019.

[13] Memcached, "Accessed on: August 15, 2024. [online]. available: https://aws.amazon.com/memcached/."

[14] SoftwareAG, "Ehcache 2.7 documentation," Documentation accessed on: August 14, 2024. [Online]. Available: https://www.ehcache.org/documentation/2.7/index.html.

[15] NebulaGraph, "A review of graph databases," Documentation accessed on: August 14, 2024. [Online]. Available: https://www.nebula-graph.io/posts/review-on-graph-databases, March 2020.

[16] Hazelcast, "Accessed on: August 27, 2024. [online]. available: https://https://hazelcast.com/resources/hazelcast-vs-coherence/."

[17] ArangoDB, "Cluster deployments," Documents accessed on: August 13, 2024. [Online]. Available: https://docs.arangodb.com/stable/deploy/cluster/.

[18] V. Mohan, A. Potturi, and M. Fukuda, "Automated Agent Migration over Distributed Data Structures," in *Proc. of the 15th International Conference on Agents and Artificial Intelligence*, February 2023, pp. 363–371.

[19] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, February 2013.

[20] L. Cao, "An Incremental Enhancement of Agent-Based Graph Database System, MS Capstone White Paper," University of Washington Bothell, Tech. Rep., June 2024.

[21] J. Leskovec, "Twitch social networks," Stanford Network Analysis Project Accessed on September 1, 2024 [Online]. Available: https://snap.stanford.edu/data/twitch-social-networks.html.

[22] Z. Li and W. Kim, "Investigating statistical analysis for network motifs," in *BCB '21: Proceedings of the 12th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*. ACM, August 2021.

[23] G. Sarma, "Neo4j storage internals," Medium, Blog accessed on: August 15, 2024 [Online]. Available: https://gauravsarma1992.medium.com/neo4j-storage-internals-be8d150028db, August 2020.

[24] ArrangoDB, "Datat structure," Documents accessed on: September 1, 2024. [Online]. Available: https://docs.arangodb.com/3.10/concepts/data-structure/.