*im*Graph: A distributed in-memory graph database

Salim Jouili
Eura Nova R&D
1435 Mont-Saint-Guibert, Belgium
Email: salim.jouili@euranova.eu

Aldemar Reynaga
Université Catholique de Louvain
1348 Louvain-La-Neuve, Belgium
Email: aldemar.reynaga@student.uclouvain.be

***Abstract*—Diverse applications including cyber security, social networks, protein networks, recommendation systems or citation networks work with inherently graph-structured data. The graphs modeling the data of these applications are large by nature so the efficient processing of them becomes challenging.**

In this paper we present *im*Graph, a graph system that addresses the challenge of efficient processing of large graphs by using a distributed in-memory storage. We use this type of storage to obtain fast random data access which is mostly required for graph exploration. *im*Graph uses a native graph data model to ease the implementation of graph algorithms. On top of it, we design and implement a traversal engine that achieves high performance by efficient memory access, distribution of the work load, and optimizations on network communications. We run a set of experiments on real graph datasets of different sizes to asses the performance of *im*Graph in relation to other graph systems. The results show that *im*Graph gets better performance on traversals on large graphs than its counterparts.

I. Introduction

Over the past few years, managing graph-structured data has gained significant attention in both industrial and research communities (see, e.g., a survey [1]). This is due to the emergence of modern applications whose data structure is naturally represented as graphs in areas such as social networks, transportation [2], biological networks and semantic webs. In fact, the graph structure provides a flexible representation for modeling highly-connected and dynamic data. In addition to their structural complexity, the graphs handled in these fields can have a very large size; for instance, the social graph of Facebook contains more than 750 million active users and an average friend count of 130. According to Bizer et al. [4], the Web of Data has 31 billion RDF triples and 466 million RDF links. Another example is Business networks mining [10] which handles large scale networks consisting of several thousand integration related and business process participants with ten or hundred thousand relationships between them as well as business participants relationships to social media networks.

The managing and processing of these large graphs can be very challenging because the huge amount of data causes a high I/O latency. As data access on graphs generally requires random data access (no locality), there is a high number of I/O operations in traversal processing. Although memory caching can be used to reduce the number of I/O operations, the no locality of data access on graphs reduces the efficiency of those caches on large graphs because the cache contents will be frequently modified so the database engine will have to perform several data reads on disk.

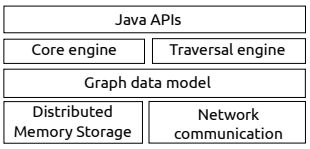

Fig. 1. *im*Graph overview

Having these challenges in mind, we introduce a new graph database system called *im*Graph [1]. We have considered the random access requirement for large graphs as a key factor on deciding the type of storage. Then, we have designed a graph database where all data is stored in memory so the speed of random access is maximized. However, as large graphs can not be completely loaded in the RAM of a single machine, we designed *im*Graph to store graph data in the memory of a group of machines. Furthermore, we implemented on *im*Graph a graph traversal engine that takes advantage of distributed parallel computing and fast in-memory random access to gain performance.

The rest of the paper has the following organization. In Section 2, we outline the design of *im*Graph, section 3 describes its graph data model, section 4 describes its core engine, section 5 describes its traversal engine, section 6 presents a brief description of other graph systems, section 7 presents the results of experiments, and section 8 includes conclusion of our work and a short description of future work directions.

II. An Overview of *im*Graph

Figure 1 illustrates a high level architecture of *im*Graph. On the bottom of the stack we have a Memory storage module, which is a distributed in-memory key/value store that can be accessed from any machine of the cluster. The network communication module provides an efficient infrastructure which allows an efficient synchronous and asynchronous exchange of messages between the machines of the cluster.

On top of the Memory storage module, we have the Graph Data Model that represents graphs in their native form; this bring us the possibility to associate information to vertices and edges in a more natural way. The Graph Data Model is explained in more detail in section 3. The next layer contains the core engine of the database which provides the basic operations on graphs and transactional support. This layer also provides an optimized graph traversal functionality which allows an efficient execution on-line queries; this traversal

[1] *im*Graph will be open sourced soon

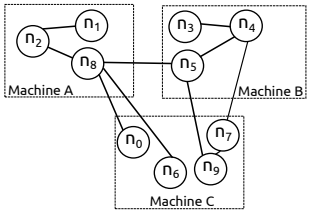

Fig. 2. Distribution of vertices in a cluster of 3 machines

functionality is detailed in section 4. Finally, on top of the stack there is a set of Java APIs which allow external clients to perform queries and data modifications on the graph database.

III. The Graph Data Model

*im*Graph models a graph using the native graph data model, i.e. a set of vertices interconnected by edges. Formally, a graph is denoted by $g = (V, E, \mu, v)$ where V denotes a finite set of vertices, $E \subseteq V \times V$ a set of edges, $\mu : V \to L$ a vertex labeling function assigning each vertex an attribute from L, and $v : E \to L$ and edge labeling function.

The basic data structure in *im*Graph is called *cell* which has a unique global id in the cluster; each cell has a map of key/value pairs to store properties. *im*Graph models vertices, edges and hyperedges as subclasses of a cell. Then, edges are cells containing a pair of cell ids pointing to two vertices, a type (IN, OUT or UNDIRECTED), and a label; vertices are cells that contain an indexed list of edges; hyperedges are cells containing an indexed list of vertex ids. Cells are stored in the key/value memory storage using its cell id as the key, and the cell itself as the value. *im*Graph determines the machine where each cell is stored using the value of a hash function of the cell id. Figure 2 shows how the vertices of a small graph would be distributed in the memory of a cluster of 3 machines.

IV. The core engine

The basic retrieval of vertices and edges is performed using the global unique numeric identifer (id) that each cell (vertex or edge) has. *im*Graph also supports the retrieval of data through indexes of edges and vertices based on their properties. The core module also provides a set of APIs to manipulate the objects representing vertices and edges, *im*Graph enforces that every data modification must be enclosed in a transaction. The *im*Graph transactions prevent dirty and non-repeatable reads by using thread local transaction contexts. However, *im*Graph currently offers no locking mechanism to control concurrent writings on the same data.

V. On-line query processing

On-line queries can be processed on *im*Graph using the basic data manipulation and retrieval Java APIs or the Blueprints interfaces[2]. However, when we are working with large graphs, the execution time of queries implemented with the basic APIs is increased considerably because of the high number of messages that are exchanged among the machines. Therefore, *im*Graph provides a traversal engine which is designed to increase local processing of nodes so that the number of messages exchanged will be reduced.

A. The basic traversal processing

Algorithm 1 Modified breadth first search algorithm

Input: $search\_message$, $search\_criteria$
Output: Path for the searched vertex if any

```
1: input_vertices := search_message.vertices
2: for all start_vertex in input_vertices do
3:     if start_vertex satisfies search_criteria then
4:         RETURN path(start_vertex)
5:     end if
6:     add start_vertex to queue
7: end for
8: cur_depth := input_vertices.depth
9: while queue is not empty do
10:    vertex := pop(queue)
11:    if vertex.depth > cur_depth) then
12:        for all m in cluster_machines do
13:            msg.vertices := remote_queue[m]
14:            send msg to m
15:        end for
16:        cur_depth := vertex.depth
17:    end if
18:    add vertex to explored
19:    for all neighbor of vertex s.t. neighbor ∉ explored∪queue
       do
20:        if neighbor is local then
21:            if neighbor satisfies search_criteria then
22:                RETURN path(neighbor)
23:            end if
24:            add neighbor to queue
25:        else
26:            add neighbor to remote_queue and explored
27:        end if
28:    end for
29: end while
30: for all m in cluster_machines do
31:    msg.vertices := remote_queue[m]
32:    send msg to m
33: end for
```

A traversal executed using basic retrieval APIs of *im*Graph causes high network traffic because the machine executing the traversal has to retrieve vertices stored in other machines. This high network traffic added to the serialization/deserialization costs results in poor performance. Furthermore, the resources of the machine running the traversal could be exhausted while running a traversal on a large graph. Then, we decided to make each machine traverse only the vertices stored in its memory. The machines will exchange messages only to coordinate the portions of the traversal that each machine will perform.

As we distribute the traversal among the machines of the cluster, we need a centralized coordinator process to start and stop the traversal processing, and collect the results of the sub traversals. Then, we added a new type of server process to perform these tasks, the new process is called traversal manager. In an *im*Graph system we can have as many traversal manager processes as we want so they don't represent a single point of failure; traversal managers can be executed on any machine of the cluster or in a dedicated machine. However, it must be noted that the traversal manager process and the *im*Graph cluster machine process runs in different JVMs.

[2]Blueprints is a generic Java API for graph databases developed by Tinkerpop (https://github.com/tinkerpop/blueprints/)

The traversal process starts when a traversal manager receives a query request. Since the traversal manager does not have direct access to graph data, it sends an asynchronous search request message to the machine that stores the start vertex indicating the start vertex id and the query specification. All search request messages have a unique message id that is used later to determine if all messages have been processed, i.e. to determine when the traversal has been completed. In fact, the traversal manager stores a pending searches list containing the search request message ids that have not yet been processed.

When the cluster machine receives the message from the traversal manager, it initiates the traversal by executing a modified breadth first search algorithm (Algorithm 1) starting from the received vertex id; we have chosen the BFS algorithm as a basis because it allows us to better distribute the traversal. The algorithm only explores the nodes that are stored in the local memory, the vertices located in remote vertices are saved for posterior processing in search request queues; the algorithm classifies the remote vertices according to the remote machine where they are stored. The processing of remote vertices consists in sending search request messages to the remote machines so that they can start a new portion of the traversal. To prevent overloading the network, the algorithm does not send a search request message for each remote vertex. Instead, it groups all the remote vertices belonging to a search depth before sending them; then, it sends asynchronously a single search request message containing the query specifications and the ids of the remote vertices to be explored. When a remote machine receives a search request message, it executes a new instance of the modified BFS algorithm based on the query specifications and starting on the vertex ids contained on the message. When the processing of the modified BFS algorithm is completed, an asynchronous search response message is sent to the traversal manager containing the id of the search request message that triggered the algorithm, the paths found and the ids of new search request messages sent during the execution of the algorithm. These new request message ids are sent so that the traversal manager will knows that it must wait for responses to them.

Upon the reception of a search response message, the traversal manager drops the corresponding element of the pending searches list, adds the ids of the new search requests to the pending searches list and stores the paths found. The query processing is completed when one or more paths satisfying the query criteria are found or when the pending searches list is empty. Then, the traversal manager sends back a query response to the client and broadcast a stop search message so the machines of the cluster stop processing search request messages for this query and release resources.

Figure 3 shows the flow of messages among the machines of the cluster when a traversal is processed. The flow starts with a traversal request message sent from the client to the traversal manager. Then, the traversal manager sends and search request message to Machine B which in turn sends search request messages to machines A and C. Then, machines A and C process the received request messages, machine A sends an additional search request message to machine C. Later, all the machines send response message to the traversal manager, one response message is sent for each request message processed. Finally, the traversal manager sends a traversal response to the client.

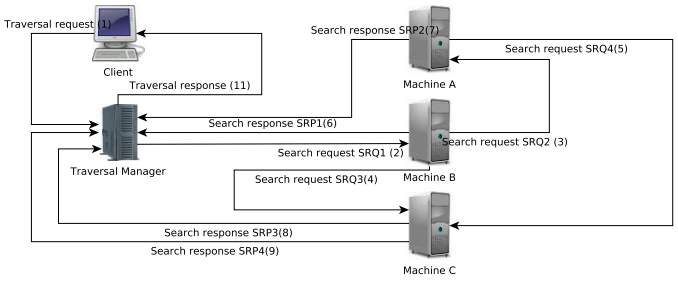

Fig. 3. Flow of messages during a traversal in *im*Graph

B. Distributed Traversal by means of virtual edges

Let's consider a case where a vertex n_1 stored in machine A has a 1-hop neighbor n_5 located in a remote machine B which in turn has a 1-hop neighbor n_8 located in machine A. If we are going to traverse this graph starting from vertex n_1, a search request message will be sent to machine B to explore the vertex n_5; then, the machine B will send again a message to machine A for the vertex n_8. Suppose that we are looking for vertex n_8, in this simple case two messages will have to be sent to found it. It would have been good that the traversal algorithm could recognize that n_8 is located in machine A before sending a search request to machine B so that it can explore n_8 and machine B will not need to send back a message to machine A for that node. Then, the number of sent messages would have been reduced to one with a related improvement on performance.

In order to address this type of situations, we redefine the *im*Graph's data model by introducing the concept of *virtual edge*. Then, we denote a graph by $g = (V, E, W, \mu, v)$ where V denotes a finite set of vertices, $E \subseteq V \times V$ a set of edges, $W \subseteq V \times V$ a set of virtual edges, $\mu : V \rightarrow L$ a vertex labeling function assigning each vertex an attribute from L, and $v : E \rightarrow L$ and edge labeling function. The set W is defined as follows: $W = \{(v_1, v_2) \in V \times V \mid \exists (v_1, v_r), (v_r, v_2) \in E : m(v_1) = m(v_2) \wedge m(v_1) \neq m(v_r)\}$, where $m : V \rightarrow MC$ is a function mapping a vertex to the machine of the cluster where it is stored, and MC denotes a set of machines. Then, for the case described in the previous paragraph, there is a virtual edge between n_1 and n_8; this virtual edge will allow the algorithm to recognize that n_8 can be reached from n_1, so it can be explored, before sending a message to explore n_5.

With the help of virtual edges we improved the performance of the traversal processing by making two adjustments in the modified BFS algorithm described in the previous subsection. The first adjustment consisted in getting the local neighbors of a vertex from the sets of edges and virtual edges so that the algorithm will be able to find more reachable local vertices, i.e. a single instance of the algorithm will be able to explore more vertices. Therefore, the number of running instances of the algorithm required to complete the whole traversal will be reduced, this in turn will reduce the traversal execution time. The second adjustment consisted in including in the search request message the ids of the vertices reached through virtual edges. By doing this, the remote machine that

TABLE I. GRAPH SYSTEMS

	Handle dynamic graph	Native graph model	Distributed storage	Distributed query processing	Memory based exploration	Transaction and index support
Neo4J	Yes	Yes	No	No	No	Yes
Titan	Yes	Yes	Yes	No	No	Yes
GBase	No	No	Yes	Yes	No	No
Trinity	Yes	Yes	Yes	Yes	Yes	No
*im*Graph	Yes	Yes	Yes	Yes	Yes	Yes

receives this message will know that the machine that sent the message is already aware of those vertices so the remote machine will not send a search request to explore them. This adjustment helps to reduce the number of messages that have to be sent to complete the traversal and as a consequence it improves the traversal performance. The implementation of virtual edges on the traversal engine represents a reduction of the processing time of approximately 40 %.

VI. RELATED WORKS

In this section we describe some graph systems currently available. This description will be done considering six factors based on the ones described by Shao et al. [12]. First, query processing on dynamic graph; second, the use of native graph representation that facilitates the implementation of graph algorithms; third, distributed storage; fourth, distributed parallel query processing; fifth, memory based exploration, which allows fast random access; fifth, transaction and index support. Table I displays the recently described factors for *im*Graph and other graph systems we will describe in the following paragraphs.

Neo4J[3] is an open source graph database that uses a graph model for data representation, provides full ACID transaction support, indexing and query language; it uses a custom disk-based native storage engine. Neo4J doesn't provide support for distributed storage and the processing power of Neo4J is limited to the computation power of a single machine.

Titan[4]is an open source graph database that uses a graph model for data representation and supports full ACID transactions, indexing and a query language. The data storage can be distributed across a cluster of machines using the distributed storage backends Apache Cassandra or Apache HBase. Although the storage can be distributed, Titan does not implement parallel processing for queries.

GBase [7] is a platform conceived to handle large static graphs that uses the sparse adjacency matrix format as data model. The graph data is loaded and automatically indexed by storing compressed blocks in a distributed storage such as HDFS of Hadoop or in a RDBMS. There's a query execution engine built on top of Hadoop that takes advantage of the indexing made during the graph loading.

Trinity [12] is a graph system that has an approach similar to *im*Graph. The graph is represented in its native form and is stored in the RAM of machines belonging to a cluster in the form of a key/value store. Trinity supports SPARQL language for queries and provides a parallel computing engine that process queries by sending asynchronous requests recursively to remote machines. Although Trinity can handle dynamic graphs, it does not provide transaction support or indexing.

VII. EXPERIMENTS

We have carried a set of experiments in order to compare the time performance of *im*Graph with Neo4J version 1.8.2; and Titan version 0.3. using an Apache Cassandra cluster version 1.2.5 as backend storage in the remote mode (Titan and the Cassandra cluster running on different JVMs).

A. Setup of experiments

The experiments were performed using four graph datasets: *soc-Epinions*[9] (75K vertices and 508K edges), a directed graph representing the who-trust-whom online social network of a general consumer review site Epinions.com; *youTube*[14] (1.1M vertices and 3M edges), an undirected graph representing the Youtube social network; *ukgraph* [5], [15] (4.5M vertices and 66M edges) a directed graph representing a segment of a snapshot of the web network in UK; and *soc-LiveJournal* [3], [8] (4.8M edges and 69M edges) a directed graph representing LiveJournal, a free on-line community with almost 10 million members with a significant fraction highly active members.

We performed three types of experiments: read tests consisting in 400 sequential retrieval of vertices; write tests consisting in 200 sequential additions of vertices, where each new vertex has two edges to already existing vertices; and graph traversal tests grouped in 4 traversal test configurations. Each traversal consisted in finding a path between two given vertices considering a determined maximum number of hops. The 4 traversal test configurations are: Traversal-3H, 200 traversals between two vertices at maximum 3 hops; Traversal-4H, 200 traversals between two vertices at maximum 4 hops; Traversal-Path-3H, 200 traversals at maximum 3 hops between two vertices having at least one path at 3 hops; and Traversal-Path-4H, 200 traversals at maximum 4 hops between two vertices having at least one path at 4 hops. Some traversals of the configuration Traversal-3H and Traversal-4H could not have a path at a specified maximum number of hops which implies that the traversal will explore all the neighbors of the start vertex at a determined maximum number of hops. Neo4J traversals were executed using its Traversal Framework, Titan traversals were executed using an implementation of the BFS algorithm because Titan has no built-on traversal functionality and *im*Graph traversals were executed using its traversal engine.

All the experiments were performed on Amazon EC2 instances of type m1.large running Ubuntu 13.04. Each m1.large instance has 64-bit processor architecture, 2 vCPU, 4 ECU, 7.5 GB of memory and moderate network performance [5]. Neo4J tests were performed in only one Amazon EC2 instance, Titan and *im*Graph tests were performed in two configurations: one running on a 5 machine cluster and the other running on a 10 machine cluster. For *im*Graph tests, we run the traversal manager on a separate machine and for Titan tests, we run

[3]Neo4j. Home.http://neo4j.org, 2013

[4]http://thinkaurelius.github.io/titan/

[5]Amazon EC2 instance types. http://http://aws.amazon.com/ec2/instance-types/#instance-details, 2013

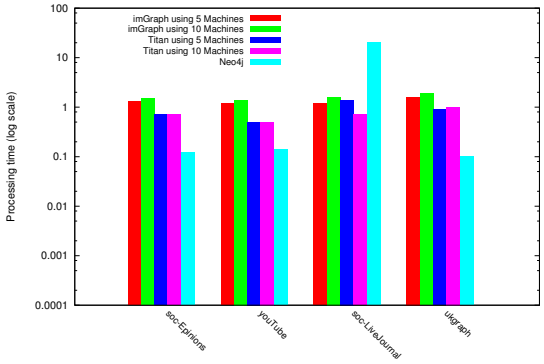

Fig. 4. Read performance results

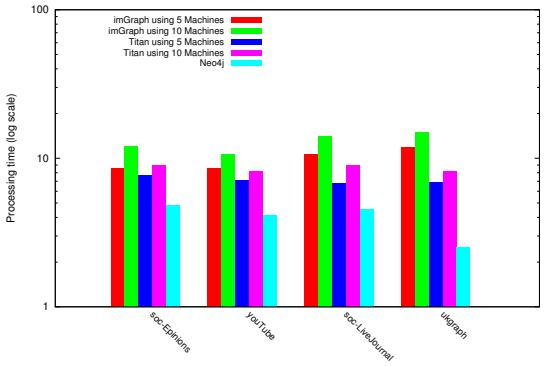

Fig. 5. Write performance results

the Titan system on a machine not belonging to the Cassandra cluster.

B. Results

The figure 4 shows the average execution time of the read tests and figure 5 shows the average execution time of the write tests; both figures show results for the five configurations and the four datasets, the average time is shown in the Y-axis using a logarithmic scale for better displaying. Regarding the read tests, *im*Graph has lower performance than the other databases because of the serialization/deserialization costs of getting remote vertices. We can also see that the write time of *im*Graph is higher than the others due to the calculation of virtual edges of the affected vertices.

Figures 6, 7, 8 and 9 show the results of the traversal tests performed on the four datasets. In all the figures we plot on the Y-axis the average traversal time for the five run configurations described in the previous section, in all the figures the time is represented in logarithmic scale. We couldn't perform the traversal tests at 4 hops (Figures 8 and 10) on Titan using the soc-LiveJournal dataset because we got out-of-memory exceptions; i.e, the amount of memory required by the BFS algorithm to explore this graph at 4 hops exceeded the capacity of the machine running the Titan system.

If we compare the results of *im*Graph and Neo4J we can observe that Neo4J is faster than our implementation only when the smallest dataset (75K vertices) is tested; when larger

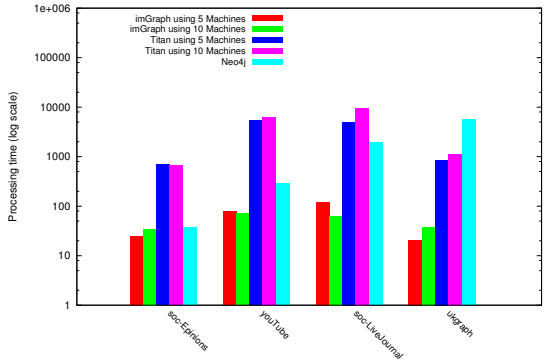

Fig. 6. Performance results of the Traversal-3H configuration

datasets are tested, *im*Graph gets better results than Neo4J. We observe that Neo4J's performance is more negatively affected by the large graphs than *im*Graph's performance. When the graph is not too large, Neo4J is able to load a considerable part of it into the cache so the traversal will be done almost in memory. However, if the graph is larger, Neo4J will be able to load in the cache only a small portion of the graph so it will have to read from the disk more frequently.

Regarding Titan and *im*Graph, both using a distributed storage, we can observe that *im*Graph performs better in all the traversal configurations for all the datasets. In this case, *im*Graph's distributed parallel computing makes the difference. As Titan traversals run on a single machine there is a considerable amount of network communication to obtain graph data; furthermore, the computation power of Titan is limited to the capacity of a single machine. Another factor that makes *im*Graph perform better is the faster random access provided by the *im*Graph's in-memory storage. Although Cassandra provides a caching functionality, this cache cannot store all data from a large graph so many disk reads will be required.

Now, let's analyse the performance results of the two run configurations of *im*Graph, i.e. *im*Graph running on a 5 machine cluster, and *im*Graph running on a 10 machine cluster. The average traversal time for both configurations are very close when the soc-Epinions, youTube and ukgraph datasets are tested. However, the traversal results obtained with the soc-LiveJournal dataset show a remarkable difference in performance between the *im*Graph run configurations; we can see that the configuration of 10 machines performs considerably better than the configuration of 5 machines. This difference suggests that the messaging between the machines of the cluster is not a bottleneck in the traversal, i.e. the higher number of exchanged messages in the 10 machine cluster doesn't affect considerably the time performance. In contrast, the better performance of the 10 machine cluster suggests that the execution of graph searches in each machine is a critical factor of the performance. In fact, the searches belonging to a traversal are distributed among the machines of the cluster according to the storage distribution; then, a machine of the 5 machine configuration performs more graph searches during a traversal because its memory stores more vertices.

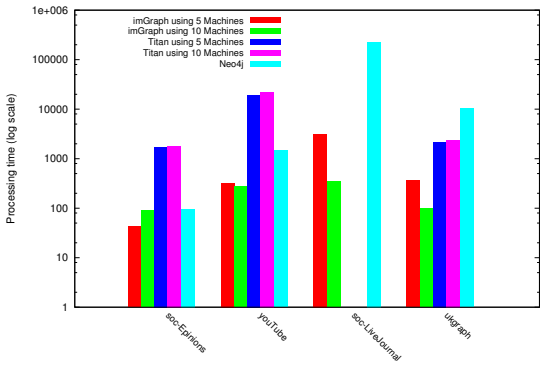

Fig. 7. Performance results of the Traversal-4H configuration

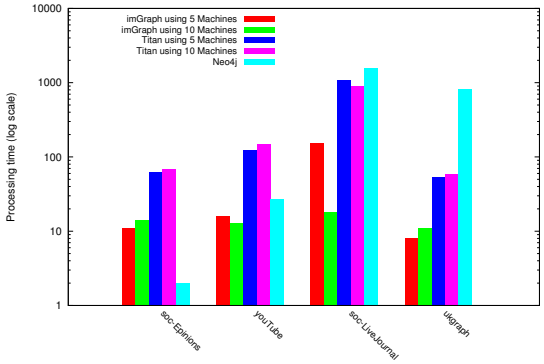

Fig. 8. Performance results of the Traversal-Path-3H configuration

VIII. Conclusion

In this paper we propose a distributed graph database called *im*Graph whose most remarkable characteristic is the storage of all graph data in the memory of machines belonging to a cluster. This characteristic brings the possibility of performance optimizations. Indeed, we have implemented a graph traversal engine that takes advantage of this characteristic to achieve high performance. The experiments we have run shown that the *im*Graph's traversal performs better than the traversals of other graph systems.

Regarding future work directions, *im*Graph could be improved by adding a mechanism for storing data on disk as well as a faster mechanism of serialization/deserialization to improve readings. An analysis of memory consumption and the evaluation of data compression techniques could also be done as well as tests on graphs having more than 1 billion of vertices. Another future work direction is the improvement of the flexibility and usability of the traversal engine by allowing the user to provide rich traversal specifications through query languages [13], [11]. The implementation of vertex-based offline graph analytic functionalities on top of *im*Graph could also be considered.

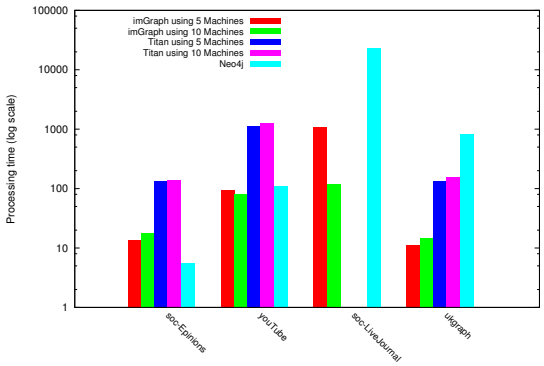

Fig. 9. Performance results of the Traversal-Path-4H configuration

Acknowledgment

The authors would like to thank Eura Nova's staff for their support and encouragement that was very important for the writing of this paper. We also want to mention that this work was done in the context of a master's thesis supervised by Peter Van Roy in the ICTEAM Institute at UCL.

References

[1] R. Angles and C. Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), 2008.

[2] L. Antsfeld and T. Walsh. Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm. In *Intelligent Transport Systems World Congress*, Vienna, October 2012.

[3] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In T. Eliassi-Rad, L. H. Ungar, M. Craven, and D. Gunopulos, editors, *KDD*, pages 44–54. ACM, 2006.

[4] C. Bizer, P. A. Boncz, M. L. Brodie, and O. Erling. The meaningful use of big data: four perspectives - four challenges. *SIGMOD Record*, 40(4):56–60, 2011.

[5] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[6] K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 2012.

[7] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. gbase: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

[8] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[9] M. Richardson, R. Agrawal, and P. Domingos. Trust management for the semantic web. In D. Fensel, K. P. Sycara, and J. Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2003.

[10] D. Ritter. From network mining to large scale business networks. In A. Mille, F. L. Gandon, J. Misselis, M. Rabinovich, and S. Staab, editors, *WWW (Companion Volume)*, pages 989–996. ACM, 2012.

[11] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The graph story of the sap hana database. In V. Markl, G. Saake, K.-U. Sattler, G. Hackenbroich, B. Mitschang, T. Härder, and V. Köppen, editors, *BTW*, LNI, pages 403–420. GI, 2013.

[12] B. Shao, H. Wang, and Y. Xiao. Managing and mining large graphs: systems and implementations. In Candan et al. [6], pages 589–592.

[13] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[14] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In M. J. Zaki, A. Siebes, J. X. Yu, B. Goethals, G. I. Webb, and X. Wu, editors, *ICDM*, pages 745–754. IEEE Computer Society, 2012.

[15] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In Candan et al. [6], pages 517–528.