# Benchmarking Graph Databases on the Problem of Community Detection

Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris

Information Technologies Institute, CERTH, 57001, Thermi, Greece
{sotbeis,papadop,ikom}@iti.gr

**Abstract.** Thanks to the proliferation of Online Social Networks (OSNs) and Linked Data, graph data have been constantly increasing, reaching massive scales and complexity. Thus, tools to store and manage such data efficiently are absolutely essential. To address this problem, various technologies have been employed, such as relational, object and graph databases. In this paper we present a benchmark that evaluates graph databases with a set of workloads, inspired from OSN mining use case scenarios. In addition to standard network operations, the paper focuses on the problem of community detection and we propose the adaptation of the Louvain method on top of graph databases. The paper reports a comprehensive comparative evaluation between three popular graph databases, Titan, OrientDB and Neo4j. Our experimental results show that in the current development status Neo4j is the most efficient graph database for most of the employed workloads, while Titan handles better single insertion operations.

## 1 Introduction

Over the past few years there has been vivid research interest in the study of networks (graphs) arising from various social, technological and scientific activities. Typical examples of social networks are graphs constructed with data from Online Social Networks (OSNs), one of the most famous and widespread Web 2.0 application categories. The rapid growth of OSNs contributes to the creation of high-volume and velocity data, which are modeled with the use of graph structures. The increasing demand for massive graph data management and processing systems has been addressed by the researchers proposing new methods and technologies, such as RDBMS, OODBMS, graph databases, etc. Every solution has its pros and cons so benchmarks to evaluate candidate solutions with respect to specific applications are considered necessary.

Relational databases have been widely used for the storage of a variety of data, including social data, and have proven their reliability. On the other hand RDBMS lack operations to efficiently analyze the relationships among the data points. This led to the development of new systems, such as object and graph databases. More specifically, graph databases are designed to store and manage effectively big graph data and constitute a powerful tool for graph-like queries, such as "find the friends of a person".



N. Bassiliades et al. (eds.), *New Trends in Database and Information Systems II*,
Advances in Intelligent Systems and Computing 312, DOI: 10.1007/978-3-319-10518-5_1

In this paper we address the problem of comparing graph databases in terms of performance, focusing on the problem of community detection. We implement a clustering workload, which consists of a well-known community detection algorithm for modularity optimization, the Louvain method [1]. We employ cache techniques to take advantage of both graph database capabilities and in-memory execution speed. The use of the clustering workload is the main contribution of this paper, because to our knowledge other existing benchmark frameworks evaluate graph databases in terms of loading time, node creation/deletion or traversal operations, such as "find the friends of a person" or "find the shortest path between two persons". Furthermore, the benchmark comprises three supplementary workloads that simulate frequently occurring operations in real-world applications, such as the the creation and traversal of the graph. The benchmark implementation is available online as an open-source project[1].

We use the proposed benchmark to evaluate three popular graph databases, *Titan*[2], *OrientDB*[3] and *Neo4j*[4]. For our experiments we used both synthetic and real networks and the comparative evaluation is held with respect to the execution time. Our experimental results show that Neo4j is the most efficient graph database to apply community detection algorithms, in our case the Louvain method. Concerning the supplementary workloads, Neo4j is also the fastest alternative, although Titan performs the incremental creation of the graph faster.

The paper is organized as follows. We begin in Section 2 by providing a survey in the area of benchmarks between database systems oriented to store and manage big graph data. In Section 3 we describe the workloads that compose the benchmark. In Section 4 we list some important aspects of the benchmark. Section 5 presents our experimental study, where we describe the datasets used for the evaluation and report the obtained experimental results. Finally, Section 6 concludes the paper and delineates our future work ideas.

## 2 Related Work

Until now many benchmarks have been proposed, comparing the performance of different databases for graph data. Giatsoglou et al. [2], present a survey of existing solutions to the problem of storing and managing massive graph data. Focusing on the Social Tagging System (STS) use case scenario, they report a comparative study between the Neo4j graph database and two custom storages (H1 and Lucene). Angles et al. [3], considering the category of an OSN as an example of Web 2.0 applications, propose and implement a generator that produces synthetic graphs with OSN characteristics. Using this data and a set of queries that simulate common activities in a social network application, the authors compare two graph databases, one RDF and two relational data management systems. Similarly, in LinkBench [4] a Facebook-like data generator is employed and the performance of a MySQL database system is evaluated. The

[1] `https://github.com/socialsensor/graphdb-benchmarks`
[2] `http://thinkaurelius.github.io/titan/`
[3] `http://www.orientechnologies.com/`
[4] `http://www.neo4j.org/`

authors claim that under certain circumstances any database system could be evaluated with LinkBench.

In a recent effort, Grossniklaus et al. [5] define and classify a workload of nine queries, that together cover a wide variety of graph data use cases. Besides graph databases they include RDBMS and OODBMS in their evaluation. Vicknair et al. [6] also present a benchmark that combines different technologies. They implemented a query workload that simulates typical operations performed in provenance systems and they evaluate a graph (Neo4j) and a relational (MySQL) database. Furthermore, the authors describe some objective measures to compare the database systems, such as security, flexibility, etc.

In contrast with the above works, we argue that the most suitable solution to the problem of massive graph storage and management are graph databases, so our research focuses on them. In this direction Bader et al. [7] describe a benchmark that consists of four kernels (operations): (a) bulk load of the data; (b) retrieval of a set of edges that verify a condition (e.g. weight $> 3$); (c) execution of a $k$-hops operation; and (d) retrieval of the set of nodes with maximum betweenness centrality. Dominguez et al. [8] report the implementation of this benchmark and a comparative evaluation of four graph database systems (Neo4j, HypergraphDB, Jena and DEX).

Ciglan et al. [9] are based on the ideas proposed in [8] and [10], and extend the discussion focusing primarily on graph traversal operations. They compare five graph databases (Neo4j, DEX, OrientDB, NativeSail and SGDB) by executing some demanding queries, such as "find the most connected component". Jouili et al. [11] propose a set of workloads similar to [7] and evaluate Neo4j, Titan, OrientDB and DEX. Unlike, previous works they conduct experiments with multiple concurrent users and emphasize the effects of increasing users. Dayarathna et al. [12] implement traversal operation-based workloads to compare four graph databases (Allegrograph, Neo4j, OrientDB and Fuseki). The key difference with other frameworks is that their interest is focused mostly on graph database server and cloud environments.

## 3 Workload Description

The proposed benchmark is composed of four workloads, Clustering, Massive Insertion, Single Insertion and Query Workload. Every workload has been designed to simulate common operations in graph database systems. Our main contribution is the Clustering workload (CW), however supplementary workloads are employed to achieve a comprehensive comparative evaluation. In this section we describe in more detail the workloads and emphasize their importance by giving some real-world examples.

### 3.1 Clustering Workload

Until now most community detection algorithms used the main memory to store the graph and perform the required computations. Although, keeping data in memory leads to fast executions times, these implementations have a major drawback: they cannot manage big graph data reliably, which nowadays is a

key requirement for big graph processing applications. This motivated this work and more specifically the implementation of the Louvain method on top of three graph databases. We used the Gephi Tookit[5] Java implementation of the algorithm as a starting point and applied all necessary modifications to adapt the algorithm to graph databases.

In a first implementation, all the required values for the computations were read directly from the database. The fact that the access of any database (including graph databases) compared to memory is very slow, soon made us realize that the use of cache techniques is necessary. For this purpose we employed the cache implementation of the Guava project[6]. The Guava Cache is configured to evict entries automatically, in order to constrain its memory footprint. Guava provides three basic types of eviction: size-based eviction, time-based eviction, and reference-based eviction. To precisely control the maximum cache size, we utilize the first type of eviction, size-based, and the evaluation was held both between different systems and among different cache sizes. The measurements concern the required time for the algorithm to be completed.

As the authors of the Louvain method mention[7], the algorithm is a greedy optimization method that attempts to optimize the modularity of a partition of the network. The optimization is performed in two steps. First, the method looks for "small" communities by optimizing modularity locally. Second, it aggregates nodes belonging to the same community and builds a new network whose nodes are the communities. We call those *communities* and *nodeCommunities* respectively. The above steps are repeated in an iterative manner until a maximum of modularity is attained.

We keep the community and nodeCommunity values stored in the graph database as a property of each node. The implementation is based on three functions that retrieve the required information either by accessing the cache or the database directly. We store this information employing the LoadingCache structure from the Guava Project, which is similar to a ConcurrentMap[8]. More specifically we use the following functions and structures:

- *getNodeNeighours*: gets the neighbours of a node and stores them to a LoadingCache structure, where the key is the node id and the value is the set of neighbours.
- *getNodesFromCommunity*: gets the nodes from a specific community and stores them to a LoadingCache structure, where the key is the community id and the value is the the set of nodes that the community contains.
- *getNodesFromNodeCommunity*: gets the nodes from a specific nodeCommunity and stores them to a LoadingCache structure, where the key is the nodeCommunity id and the value is the the set of nodes that the nodeCommunity contains.

[5] `https://gephi.org/toolkit/`
[6] `https://code.google.com/p/guava-libraries/`
[7] `http://perso.uclouvain.be/vincent.blondel/research/louvain.html`
[8] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html`

We use the above information to compute values such as, the degree of a node, the amount of connections a node or a nodeCommunity has with a particular community, the size of a community or a nodeCommunity.

The Clustering Workload is very important due to its numerous applications in OSNs [13]. Some of the most representative examples include topic detection in collaborative tagging systems, such as Flickr or Delicious, tag disambiguation, user profiling, photo clustering, and event detection.

### 3.2 Supplementary Workloads

In addition to CW, we recognize that a reliable and comprehensive benchmark should contain some supplementary workloads. Here, we list and describe the three additional workloads that constitute the proposed benchmark.

- *Massive Insertion Workload (MIW)*: we create the graph database and configure it for massive loading, then we populate it with a particular dataset. We measure the time for the creation of the whole graph.
- *Single Insertion Workload (SIW)*: we create the graph database and load it with a particular dataset. Every object insertion (node or edge) is committed directly and the graph is constructed incrementally. We measure the insertion time per block, which consists of one thousand nodes and the edges that appear during the insertion of these nodes.
- *Query Workload (QW)*: we execute three common queries:
  - FindNeighbours (FN): finds the neighbours of all nodes.
  - FindAdjacentNodes (FA): finds the adjacent nodes of all edges.
  - FindShortestPath (FS): finds the shortest path between the first node and 100 randomly picked nodes.

  Here we measure the execution time of each query.

It is obvious that MIW simulates the case study in which graph data are available and we want to load them in batch mode. On the other hand, SIW models a more real-time scenario in which the graph is created progressively. We could claim that the growth of an OSN follows the steps of SIW, by adding more users (nodes) and relationships (edges) between them.

The QW is very important as it applies in most of the existing OSNs. For example with the FN query we can find the friends or followers of a person in Facebook or Twitter respectively, with the FA query we can find whether two users joined a particular Facebook group and with the FS query we can find at which level two users connect with each other in Linkedin. It is critical for every OSN that these queries can be executed efficiently and in minimal time.

## 4 Benchmark Description

In this section we discuss some important aspects of the benchmark implementation. The graph database systems selected for the evaluation are Titan (v0.4.1), OrientDB (v1.7-RC2) and Neo4j (v2.0.1). The benchmark was implemented in

Java 1.7 using the Java API of each database. In order to configure each database, we used the default configuration and the recommendations found in the documentation of the web sites.

For Titan we implement MIW with the BatchGraph interface that enables batch loading of a large number of edges and vertices, while for OrientDB and Neo4j we employ the OrientGraphNoTx and BatchInserter interface respectively, which drop the support for transactions in favor of insertion speed. For all graph databases we implement SIW without using any special configuration. The operations for the QW and CW for the OrientDB and Titan were implemented using the Gremlin API, whereas for the Neo4j we employed its respective API.

To ensure that a benchmark provides meaningful and trustworthy results, it is necessary to guarantee its fairness and accuracy. There are many aspects that can influence the measurements, such as the system overhead. It is really important that the results do not come from time periods with different system status (e.g. different number of processes in the background), so we execute MIW, SIW and QW sequentially for each database. In addition to this, we execute them in every possible combination for each database, in order to minimize the possibility that the results are affected by the order of execution. We report the mean value of all measurements.

Regarding the CW, in order to eliminate the cold cache effects we execute it twice and keep always the second value. Moreover, as we described in the previous section to get an acceptable execution time, cache techniques are necessary. The cache size is defined as a percentage of total nodes. For our experiments we use six different cache sizes (5%, 10%, 15%, 20%, 25%, 30%) and we report the respective improvements.

## 5 Experimental Study

In this section we present the experimental study. At first we describe the datasets used for the evaluation. We include a table with some important statistics of each dataset. Then we report and discuss the results.

### 5.1 Datasets

The right choice of datasets that will be used for running database benchmarks is important to obtain representative and meaningful results. It is necessary to test the databases on a sufficient number of datasets of different sizes and complexity to get an approximation of the database scaling properties.

For our evaluation we use both synthetic and real data. More specifically, we execute MIW, SIW and QW with real networks derived from the SNAP dataset collection[9]. On the other hand, with the CW we use synthetic data generated with the LFR-Benchmark generator [1] that produces networks with power-law degree distribution and implanted communities within the network. The Table 1 presents the summary statistics of the datasets.

[9] `http://snap.stanford.edu/data/index.html`

**Table 1.** Datasets used in the experiments

| Dataset | Nodes | Edges | max. $\kappa$ | $\langle\kappa\rangle$ | $\langle cc\rangle$ |
|---|---|---|---|---|---|
| Graph500 | 500 | 1,975 | 25 | 7.904 | 0.267 |
| Graph1000 | 1,000 | 7,745 | 50 | 15.490 | 0.296 |
| Graph2000 | 2,000 | 29,720 | 100 | 29.721 | 0.291 |
| Graph3000 | 3,000 | 65,839 | 150 | 43.893 | 0.303 |
| Graph4000 | 4,000 | 120,814 | 200 | 60.407 | 0.301 |
| Graph5000 | 5,000 | 184,711 | 249 | 73.884 | 0.313 |
| Graph10000 | 10,000 | 748,105 | 500 | 149.622 | 0.291 |
| Enron (EN) | 36,692 | 367,662 | 1,383 | 20.041 | 0.497 |
| Amazon (AM) | 334,863 | 925,872 | 168 | 5.530 | 0.398 |
| Youtube (YT) | 1,134,890 | 2,987,624 | 28,576 | 5.265 | 0.081 |
| Livejournal (LJ) | 3,997,962 | 34,681,189 | 14,703 | 17.349 | 0.045 |

### 5.2 Benchmark Results

In this section we report and discuss the performance of Titan, OrientDB and Neo4j employing the proposed benchmark. Table 2 lists the required time for the execution of MIW and QW, while Figure 1 illustrates the experimental results of SIW. Table 3 and Figure 2 depict the measurements of CW. Note that in every table we mark the best performance with bold. All experiments were run on a 2×Intel Xeon 6-core at 2.1Ghz with 128GB of main memory and a 2.8 TB hard disk, the OS being Ubuntu Linux 12.04 (64bit).

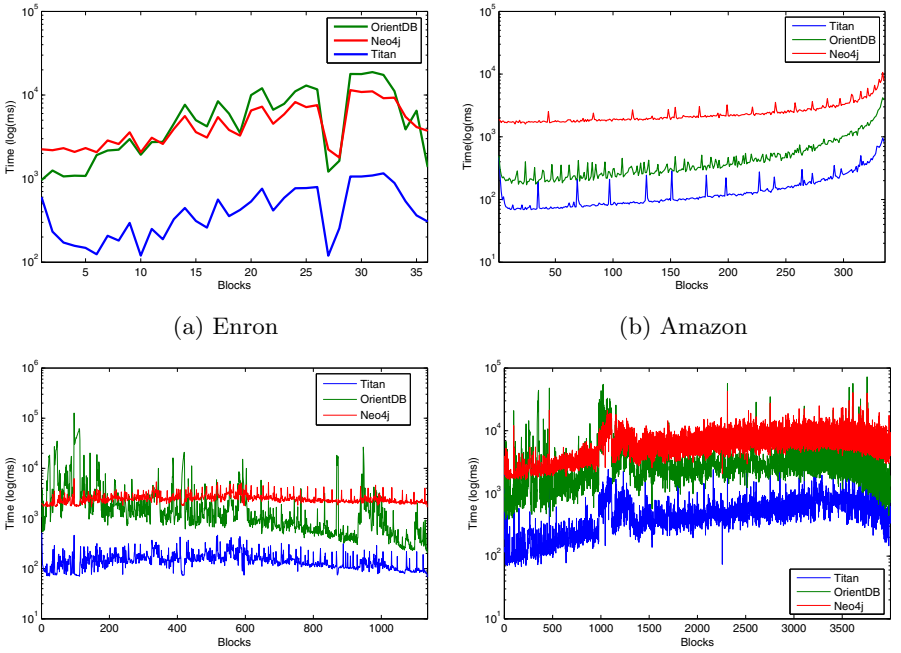
Table 2 summarizes the measurements of the MIW and QW for all the benchmarked graph databases with respect to each real dataset. According to the benchmark results, we observe that Neo4j handles the massive insertion of the data more efficiently from its competitors. Titan is also an effective alternative, while OrientDB could not load the data in a comparable time.

Concerning the QW, Table 2 indicates that Neo4j performs queries more effectively than the other candidates. More specifically, although Neo4j has slightly smaller execution times comparing to OrientDB in the FN query load, Neo4j is considerably faster in the FA and FS query loads. It is worth mentioning that the shortest path search is limited to paths of depth 6, because with larger depth the FS query workload in Titan and OrientDB cannot be executed in a reasonable amount of time.

The results for SIW with each real dataset are illustrated in Figure 1. Each sub-figure includes three diagrams, one for every graph database, that plot the required time for the insertion of a block. As we described in Section 3, a block consists of 1,000 nodes and the edges that appear during the insertion of these nodes. In order to present more readable diagrams for the three technologies we used a logarithmic scale for the time axis. It appears that Titan is the most efficient solution for single insertion of data. Moreover, we observe that the performance of OrientDB and Neo4j is comparable, however OrientDB seems to perform slightly better.

**Table 2.** MIW and QW results (sec)

| Graph | Workload | Titan | OrientDB | Neo4j |
|---|---|---|---|---|
| EN | MIW | 7.020 | 378.192 | **1.091** |
| AM | MIW | 28.997 | 298.560 | **4.635** |
| YT | MIW | 80.064 | 5963.333 | **16.830** |
| LJ | MIW | 720.179 | 2485.529 | **229.465** |
| EN | QW-FN | 4.707 | **0.976** | 1.144 |
| AM | QW-FN | 12.243 | 4.75 | **3.224** |
| YT | QW-FN | 37.912 | 14.804 | **10.532** |
| LJ | QW-FN | 352.968 | 68.176 | **109.494** |
| EN | QW-FA | 6.132 | 3.449 | **0.478** |
| AM | QW-FA | 18.143 | 17.877 | **1.446** |
| YT | QW-FA | 58.921 | 44.013 | **3.486** |
| LJ | QW-FA | 504.909 | 341.306 | **44.510** |
| EN | QW-FS | 10.652 | 16.554 | **0.483** |
| AM | QW-FS | 0.149 | 11.382 | **0.293** |
| YT | QW-FS | 6.598 | 6.927 | **0.223** |
| LJ | QW-FS | 31.01 | 47.183 | **0.479** |




(a) Enron (b) Amazon

(c) Youtube (d) Livejournal

**Fig. 1.** SIW benchmark results

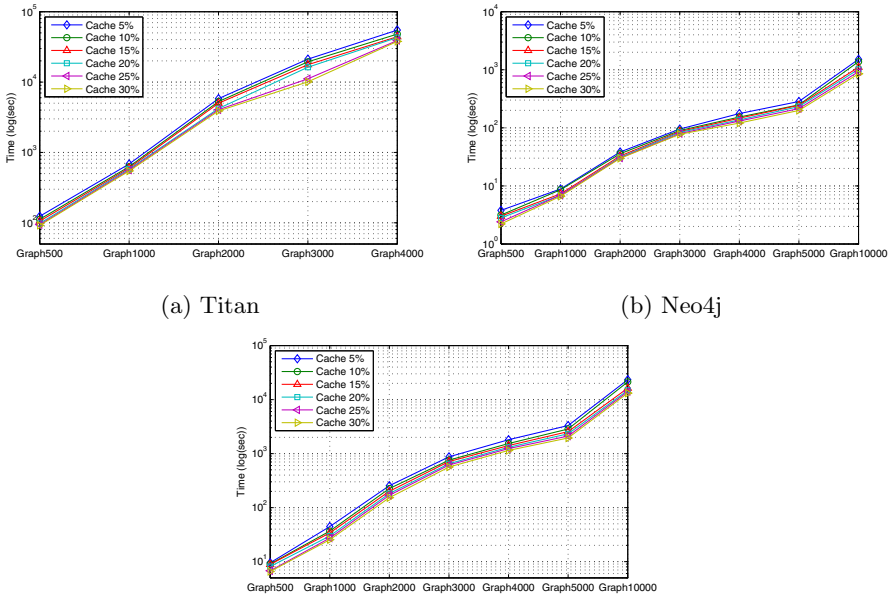
**Table 3.** CW results (sec)

| Graph-Cache | Titan | OrientDB | Neo4j |
|---|---|---|---|
| Graph500-5% | 123.774 | 9.525 | **3.786** |
| Graph500-10% | 111.943 | 9.019 | **3.15** |
| Graph500-15% | 105.425 | 8.879 | **3.01** |
| Graph500-20% | 98.084 | 8.126 | **2.808** |
| Graph500-25% | 94.183 | 6.776 | **2.408** |
| Graph500-30% | 92.021 | 6.566 | **2.212** |
| Graph1000-5% | 683.572 | 44.449 | **8.841** |
| Graph1000-10% | 626.777 | 36.651 | **8.591** |
| Graph1000-15% | 599.873 | 33.896 | **7.312** |
| Graph1000-20% | 579.948 | 29.797 | **6.912** |
| Graph1000-25% | 562.162 | 27.227 | **6.94** |
| Graph1000-30% | 544.43 | 25.448 | **6.629** |
| Graph2000-5% | 5822.678 | 254.05 | **38.29** |
| Graph2000-10% | 5283.674 | 224.65 | **35.824** |
| Graph2000-15% | 5004.3 | 200.074 | **32.747** |
| Graph2000-20% | 4202.316 | 182.04 | **31.111** |
| Graph2000-25% | 4041.574 | 168.995 | **30.317** |
| Graph2000-30% | 3884.714 | 152.193 | **29.833** |
| Graph3000-5% | 21191.741 | 870.295 | **95.355** |
| Graph3000-10% | 19364.449 | 755.737 | **89.931** |
| Graph3000-15% | 17663.532 | 711.586 | **86.228** |
| Graph3000-20% | 16250.283 | 649.357 | **83.474** |
| Graph3000-25% | 1137.774 | 614.796 | **79.753** |
| Graph3000-30% | 1081.654 | 564.096 | **77.595** |
| Graph4000-5% | 54758.397 | 1806.228 | **175.89** |
| Graph4000-10% | 47998.816 | 1533.492 | **153.151** |
| Graph4000-15% | 44158.921 | 1408.739 | **146.413** |
| Graph4000-20% | 42834.153 | 1309.597 | **136.671** |
| Graph4000-25% | 38979.079 | 1237.009 | **130.942** |
| Graph4000-30% | 37809.707 | 1145.699 | **120.792** |
| Graph5000-5% | - | 3308.99 | **284.463** |
| Graph5000-10% | - | 2846.417 | **248.167** |
| Graph5000-15% | - | 2547.742 | **244.285** |
| Graph5000-20% | - | 2327.123 | **231.639** |
| Graph5000-25% | - | 2114.561 | **213.956** |
| Graph5000-30% | - | 1944.151 | **199.226** |
| Graph10000-5% | - | 23351.878 | **1514.992** |
| Graph10000-10% | - | 21250.209 | **1378.643** |
| Graph10000-15% | - | 16292.998 | **1106.668** |
| Graph10000-20% | - | 14866.024 | **1017.766** |
| Graph10000-25% | - | 14042.227 | **927.603** |
| Graph10000-30% | - | 13183.961 | **852.232** |

The experimental results of CW are reported in Table 3. Both the OrientDB and the Neo4j are much faster than the Titan graph database. The serious delay of the Titan did not allow us to complete the experiments with larger graphs (Graph5000 and Graph10000). Furthermore, Table 3 reveals that while OrientDB has comparable execution times with Neo4j for the small graphs, it does not scale as good as Neo4j. Thus, for graphs with >1,000 nodes, Neo4j is much faster. Given that the Louvain method consists of a set queries, such as "get the neighbours of a node", the above results are expected, if we take into consideration the results of the QW.

Additionally, Table 3 points out the positive impact of increasing the cache size. We observe that for all graph databases regardless of the graph size, as the cache size increases the execution time decreases. We wrap up the comparative evaluation, including Figure 2 which depict the scalability of each database when the CW is executed. Every sub-figure contains six diagrams, one for each cache value, that plot the required time for the convergence of the algorithm for the respective synthetic graph. For better representation we used a logarithmic scale for the time axis. We can deduce that since the diagrams with the logarithmic scale increase linearly, the execution time follows an exponential distribution.

In summary, we found that despite the fact that Neo4j falls behind in SIW compared to Titan and OrientDB, for the MIW, QW and CW, is clearly the most efficient solution, especially when we deal with big graph data. On the




(a) Titan

(b) Neo4j

(c) OrientDB

**Fig. 2.** CW benchmark results

other hand, Titan is the fastest alternative for the incremental creation of a graph database (SIW). Titan also has competitive performance in MIW, but does not scale very well compared to its two competitors.

## 6 Conclusions and Future Work

In this paper we proposed a benchmark framework for the comparative evaluation of database systems oriented to store and manage graph data. The benchmark consists of four workloads, Massive Insertion, Single Insertion, Query and Clustering Workload. For the Clustering Workload we implemented a well-known community detection algorithm, the Louvain method, on top of three graph databases. Employing the proposed benchmark we evaluated the selected graph databases, Titan, OrientDB and Neo4j using both synthetic and real networks.

The experimental results demonstrate that in most cases the measurements are comparable when processing small graphs. But when the size of the datasets grows significantly, Neo4j appears to be the most efficient solution for storing and managing graph data. On the other hand, Titan seems to be the best alternative for single insertion operations.

In the future we hope to investigate the performance gain if we parallelize the operations of the graph databases. Moreover, it would be interesting to run the benchmark employing the distributed implementations of Titan and OrientDB in order to examine their horizontal and vertical scalability properties. Also, we intend to improve the performance of the implemented community detection algorithm and test it on graphs of much larger size.

**Acknowledgments.** This work was supported by the SocialSensor FP7 project, partially funded by the EC under grant agreement 287975.

## References

1. Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008)
2. Giatsoglou, M., Papadopoulos, S., Vakali, A.: Massive graph management for the web and web 2.0. In: Vakali, A., Jain, L.C. (eds.) New Directions in Web Data Management 1. SCI, vol. 331, pp. 19–58. Springer, Heidelberg (2011)
3. Angles, R., Prat-Pérez, A., Dominguez-Sal, D., Larriba-Pey, J.L.: Benchmarking database systems for social network applications. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, pp. 15:1–15:7. ACM, New York (2013)
4. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: Linkbench: a database benchmark based on the facebook social graph (2013)
5. Grossniklaus, M., Leone, S., Zäschke, T.: Towards a benchmark for graph data management and processing (2013)

6. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: A data provenance perspective. In: Proceedings of the 48th Annual Southeast Regional Conference, ACM SE 2010, pp. 42:1–42:6. ACM, New York (2010)
7. Bader, D.A., Feo, J., Gilbert, J., Kepner, J., Koester, D., Loh, E., Madduri, K., Mann, B., Meuse, T., Robinson, E.: HPC scalable graph analysis benchmark (2009)
8. Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazán, N., Larriba-Pey, J.L.: Survey of graph database performance on the hpc scalable graph analysis benchmark. In: Shen, H.T., et al. (eds.) WAIM 2010. LNCS, vol. 6185, pp. 37–48. Springer, Heidelberg (2010)
9. Ciglan, M., Averbuch, A., Hluchy, L.: Benchmarking traversal operations over graph databases. In: 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW), pp. 186–189 (April 2012)
10. Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, V., Baleta, P., Larriba-Pey, J.: A discussion on the design of graph database benchmarks. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 25–40. Springer, Heidelberg (2011)
11. Jouili, S., Vansteenberghe, V.: An empirical comparison of graph databases. In: 2013 International Conference on Social Computing (SocialCom), pp. 708–715 (September 2013)
12. Dayarathna, M., Suzumura, T.: Xgdbench: A benchmarking platform for graph stores in exascale clouds. In: 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 363–370 (December 2012)
13. Papadopoulos, S., Kompatsiaris, Y., Vakali, A., Spyridonos, P.: Community detection in social media. Data Mining and Knowledge Discovery 24(3), 515–554 (2012)