Graphalytics: A Big Data Benchmark for Graph-Processing Platforms

Mihai Capotă
Delft University of Technology
m.capota@tudelft.nl

Tim Hegeman
Delft University of Technology
t.m.hegeman@tudelft.nl

Alexandru Iosup
Delft University of Technology
a.iosup@tudelft.nl

Arnau Prat-Pérez
Universitat Politècnica de Catalunya
aprat@ac.upc.edu

Orri Erling
OpenLink Software, U.K.
oerling@openlinksw.com

Peter Boncz
CWI, Amsterdam, The Netherlands
p.boncz@cwi.nl

ABSTRACT

Graphs are increasingly used in industry, governance, and science. This has stimulated the appearance of many and diverse graph-processing platforms. Although platform diversity is beneficial, it also makes it very challenging to select the best platform for an application domain or one of its important applications, and to design new and tune existing platforms. Continuing a long tradition of using benchmarking to address such challenges, in this work we present our vision for Graphalytics, a big data benchmark for graph-processing platforms. We have already benchmarked with Graphalytics a variety of popular platforms, such as Giraph, GraphX, and Neo4j.

1. INTRODUCTION

Graph data is increasingly used in industry, governance, and science. Generic big data processing platforms, such as Hadoop, can process graphs, but are generally slow for challenging graph-processing algorithms [3, 4] or graph datasets [4, 7]. Consequently, many competing graph-processing platforms, such as Giraph and GraphX, have recently emerged. Selecting the right platform for a particular application is a difficult process, because performance depends not only on the processing platform, but also on the workload, that is, the algorithm being executed and the graph data itself. Benchmarking graph-processing platforms is thus an important and timely topic. Several studies have compared the performance of graph processing platforms [3, 4, 7] using multiple algorithms and/or datasets, but the *de facto* benchmarking standard is currently Graph500, which is limited to a single algorithm applied to a synthetic graph model. In this work, we present our vision for Graphalytics, a big data benchmark for graph-processing platforms.

We envisage that Graphalytics will be used to benchmark mostly distributed processing platforms, like Giraph, because they are best suited for running data-intensive algorithms on large datasets (we also refer to them as "graph programming frameworks"). However, we also support in Graphalytics traditional graph databases and include an implementation for one such platform, Neo4j. Furthermore, we plan to support databases for RDF semantic web data and are working on implementing support for OpenLink Virtuoso, a popular RDF database.

Benchmarking big-data graph-processing platforms is challenging both methodologically and practically [6]. Defining the workload is a methodological challenge: the algorithms should be meaningful for real-world processing, but also stress the *choke points* of the systems under test; and the datasets should be representative of real-world graphs, but also be suitable for processing on systems of different scales. The algorithms used so far in graph-processing benchmarks are either simplistic, such as the BFS traversal algorithm used in Graph500, or detailing operations that are specific to (distributed) graph databases [1, 16]. They do not match the diverse operations and algorithms seen in distributed graph-processing platforms [4]. Similarly, an analysis of choke points, which exists for graph database operations [16], is still lacking for distributed graph-processing platforms. The datasets used in Graph500 are generated with a synthetic structure, R-MAT, which requires extensions to represent well the detailed interconnections and attributes present in the real graphs [10, 17].

Benchmarking also raises numerous practical challenges: designing a benchmarking system that can accommodate new graph-processing platforms. Academic studies generally rely on custom experimental setups that are not portable to new platforms. A benchmarking harness that supports a variety of graph-processing platforms could reduce significantly the engineering effort needed for benchmarking new platforms or in new environments.

Our vision is to address these and other [6, 9] methodological and practical challenges in Graphalytics, a benchmarking effort that assembles expertise from the Linked Data Benchmark Council (LDBC) and the Standard Performance Evaluation Corporation (SPEC) communities. Toward creating an universal benchmark for graph-processing platforms, in this work our contribution is two-fold:

1. We present the Graphalytics benchmark design (Section 2). We focus on three important steps forward: toward a choke point analysis for selecting algorithms,

toward a scalable graph generator whose output mimics the characteristics of real-world graphs, and toward a *benchmarking harness* with an API that enables developers of graph-processing platform to easily integrate their platforms into the benchmark.

2. We present real-world results obtained with the current tools of the Graphalytics benchmark (Section 3). The results include a performance analysis of the data generator, results from benchmarking three generic and graph-processing platforms, and statistics about the evolution of code quality.

2. GRAPHALYTICS: OUR VISION

Our partners in LDBC are helping us develop Graphalytics using a choke-point based design. We analyze processing platforms to identify low-level technical challenges encountered during graph processing and include in Graphalytics the workloads that stress the choke points. In addition to representative real-world datasets [4], we include in Graphalytics synthetic datasets generated using the LDBC Social Network Benchmark (SNB) data generator (Datagen) [18]. While developing Graphalytics, we are also enhancing Datagen with the capability of mimicking the structure of arbitrary real-world datasets. Additionally, Graphalytics features an advanced benchmarking harness that offers a unified execution environment for all graph processing platforms, and consistent reporting that facilitates comparisons between all possible combinations of platforms, datasets, and algorithms.

2.1 Choke Points

In order to make a benchmark *representative*, one should base its design on a survey of real-life scenarios and datasets. Such an approach transplants real-life workloads into a synthetic, isolated, benchmark setting. However, relying purely on the analysis of existing workloads one may fall victim to a kind of tunnel vision: it may lead to benchmarking only those tasks that already are being performed using existing graph data management systems, thereby excluding tasks where currently technology currently is not yet useful.

Besides allowing the systematic comparison of existing systems, benchmark design can also have the goal of advancing the state of the art, and stimulating the emergence of new kinds of systems. Therefore, a "choke-point" based methodology for benchmark design was devised in the LDBC project. The main idea is to involve system architect experts in database design, typically people who were involved in designing existing systems, to identify the crucial technological challenges that they are struggling with. These challenges are called "choke points". When devising the benchmark workload with scenarios based on real-world usage, the technical experts again assess in how far these scenarios cover the identified choke points. This may lead to introducing additional complexities in the scenarios. In case of Graphalytics some of the choke points we have identified are:

Excesive network utilization. Graph database systems and programming frameworks can either use a single-server or distributed approach, where the latter have an ability to scale the system resources with the complexity of the task. This is an attractive property, however it comes at the price of having to distribute graph computations over multiple machines and communicate between them. Consequently, network messages need to be sent and received, and if the communication needs of all nodes and their CPU exceed the available network capacity, the system becomes network work bound and ceases to scale. As such, graph workloads call for methods that may reduce the network communication in distributed algorithms. Examples of possible directions are replication schemes, data compression, and advanced (e.g., min-cut) graph partitioning methods.

Large graph memory footprint. Graph database systems often prefer to work in main memory, the main reason being that the complex structure of graphs makes it hard to use sequential or block-based access methods; hence graph algorithms tend to prefer random access memory (RAM). With reference to the previous point, single-server systems by design have a limitation in the amount of RAM such that for them the compact representation of graphs directly affects their scalability. Hence, there is a drive for new and compact graph storage and compression and summarization algorithms that allow to store more data in less RAM.

Poor access locality. The lack of temporal and spatial locality in many graph algorithms, even when only considering RAM based systems provides system level challenges. Modern computers are known not to perform well on intensive random-access workloads, because RAM latencies are high in comparison with CPU clock speeds and therefore the ability of on-chip caches to reduce the amount of memory access is quite important. Further, the emergence of flash storage (and in the further future persistent memory as well) provides opportunities to scale beyond RAM, but such platforms in fact even more strongly punish workloads without locality. As such, we foresee a tendency to optimize graph processing methods by looking at the fine-grained access patterns, and making them more local in terms of temporal locality on the small block (cache line) level.

Skewed execution intensity. Graph algorithms in distributed systems typically work in iterative fashion, often with synchronization barriers in between. Therefore, it important to assure that the diverse computing nodes at each iteration have exactly the same amount of work, in order to get full resource usage and achieve linear scalability. Often, when handling complex graph processing tasks, on real-world graphs, which are highly interconnected by also highly correlated in nature, one observes a *skew* in the workload, however. Additionally, iterative algorithms often have a varying workload in the diverse iterations, e.g., those that compute a converging metric (e.g., PageRank or clustering) in the later iterations typically perform less work (e.g., less vertices in the graph remain active). Depending on what the stopping condition of the algorithm is, there can sometimes be many of such final iterations with little work. In such situations, the network latency and synchronization very easily becomes dominant over CPU cost, and decreases the overall efficiency of the system. Given this choke point, possible techniques that may arise are: adaptive graph repartitioning or replication to achieve better work balance, or the use of asynchronous distributed query processing, and/or adaptive switching of distributed computation to central computation to handle iterations with little work.

These choke points are just illustrations and are not final. The idea of LDBC is to design the Graphalytics workload such that all these issues arise at some point, thereby rewarding innovative systems that in the future will address these challenges.

Dataset	Nodes	Edges	Gl. CC	Avg. CC	Asrt.
Amazon	0.3M	1.2M	0.2361	0.4198	0.0027
Youtube	1.1M	3.0M	0.0062	0.0808	-0.0369
LiveJournal	4M	35M	0.1253	0.2843	0.0452
Patents	3.8M	16.5M	0.0671	0.0757	0.1332
Wikipedia	2.4M	5.0M	0.0022	0.0526	-0.0853

Table 1: Characteristics of real graphs.

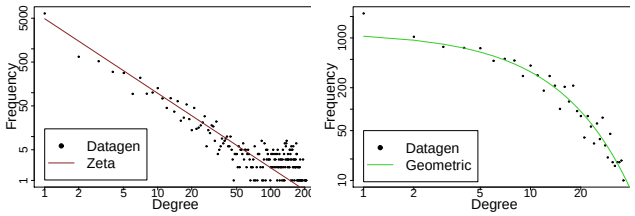

Figure 1: Node degree of Datagen graphs compared to Zeta and Geometric models.

2.2 Data Generation

The choke points described depend not only on the algorithms but also on the data. Besides the size of the dataset, a comprehensive benchmark must also consider other features observed in real data, such as the node degree distribution or structural properties like the clustering coefficient or the degree of assortativity, as these can severely affect the performance of the systems under test. For example, a graph with a large clustering coefficient (which indicates the presence of a community structure) opens interesting possibilities such as to be laid out in memory to have better cache locality [18].

By observing real data, we see that real graphs are diverse with regard to these characteristics. In Table 1, we show the characteristics of a set of real datasets[1], including their size, their global and average clustering coefficient and their degree of assortativity. We note that there is not a particular dominant configuration, but the configuration space is heterogeneous. We also analyzed the degree distributions of these graphs, by fitting them with several existing models: Zeta, Geometric, Weibull and Poisson. We observed that, depending on the graph, the best fitting model changed, being sometimes very different from the shape of the observed degree distribution as in the case of the Amazon graph.

Ideally, for a graph analytics benchmark one would desire a comprehensive battery of real graphs with different characteristics, mainly for two reasons: the first is that using real graphs reinforces the credibility of the benchmark, specially if these come from different domains, as a domain based classification of the benchmarked systems is later possible. Second, it allows to properly characterize the behavior of the systems when these characteristics are scaled individually, allowing to enclose the scenarios where the tested systems perform well (both in terms of the data and the algorithms executed). However, as shown in Table 1, real graphs are very diverse, and finding a set of them covering a rich enough configuration space is not feasible in practice, thus in Graphalytics we propose to complement the usage of real graphs with synthetic graphs that follow the characteristics observed in real data. For this reason, we propose using Datagen [16], the data generator used in the LDBC Social Network Benchmark. Datagen is an evolution of the S3G2 Data Generator [15], which simulates the activity of a social network realistically, where nodes are structurally correlated based on their attributes. Furthermore, it is built on top of Hadoop, thus being capable of generating large datasets using commodity clusters. Datagen fulfills many of the requirements of a data generator for Graphalytics: it generates a social network, which is easy to understand for the users of the benchmark; it can produce very large datasets, which is necessary as our target systems are typically used for large scale data analysis; and it is deterministic, guaranteeing reproducible results and fair comparisons.

However, in order to match the needs of Graphalytics, we need to extend Datagen's functionality in the following way:

Multiple degree distributions: In its current version, Datagen supports only a single distribution following that observed by the engineers of Facebook [19]. To support the ability to generate graphs of different characteristics, we have extended Datagen with the capability to dynamically reproduce different distributions by means of plugins. We have already implemented those for the Zeta and Geometric distribution models, but more will be added in the future as more real graphs are analysed. Figure 1 shows the actual degree distribution of two graphs generated following the Zeta ($\alpha = 1.7$) and Geometric ($p = 0.12$) distributions respectively, compared to the expected result. We see that Datagen can reliably reproduce these two distributions. Furthermore, for those graphs whose distributions cannot be theoretically modeled, we have implemented a plugin to feed Datagen with empirical data to be reproduced, in a similar way Datagen already does for the Facebook distribution.

Different structural characteristics: The current output of Datagen has an average clustering coefficient of about 0.1 with a negative degree assortativity. These are a consequence of the correlated edge generation process implemented in Datagen and cannot be determined a priori. In Table 1 we see that, in real graphs, the values of average clustering coefficient range from 0.05 to 0.63, and we observe graphs either with positive or negative assortativity. As explained above, these structural characteristics highly influence some of the choke points, thus being able to configure them is of high importance. Therefore, for Graphalytics we plan is to extend the current windowed based edge generation process of Datagen, to allow the generation of graphs with a target average clustering coefficient, but also to decide whether the assortativity is positive or negative, while preserving the degree distribution of the graph. We envision this process as a post processing step where the graph is iteratively rewired until the desired values are achieved, in a hill climbing fashion. For similar techniques on this topic please refer to [8] and [20].

2.3 Advanced Benchmarking Harness

In addition to the choke-point based design and integration with Datagen, the main advantage of Graphalytics over previous graph processing performance evaluation tools is its advance benchmarking harness which facilitates the addition of new datasets, algorithms, and platforms, as well as the actual performance evaluation process, including result collection and reporting.

Figure 2 presents an overview of the Graphalytics architecture. The *Benchmark Core* module implements the benchmark harness that binds together Graphalytics. There is a *Platform-specific algorithm implementation* module for

[1]Downloaded from SNAP `http://snap.stanford.edu`

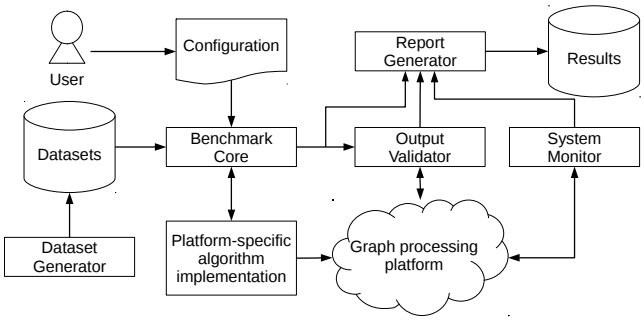

Figure 2: Graphalytics architecture.

each supported platform. The *System Monitor* is responsible for gathering resource utilization statistics from the SUT. The *Output Validator* checks the outcome of the benchmark to ensure correctness. The *Report Generator* produces the main outcome of Graphalytics, a detailed report on the performance of the SUT during the benchmark, which includes all relevant configuration information. Additionally, Graphalytics has a database for *Datasets*, which includes preconfigured graphs ready to be used with Graphalytics. Furthermore, users can generate using the Datagen *Data Generator* new synthetic datasets to suit the requirements of their applications. The design also includes a database for *Results* that is hosted by us online and accepts results submissions from Graphalytics users.

We plan to stabilize the development of Graphalytics and offer an API that will enable third party developers to port our benchmark to their graph processing platforms. From a high-level perspective, adding a new platform to Graphalytics consists of implementing the algorithms, adding a dataset loading method, providing a workload processing interface, and logging the information required for results reporting.

For users, benchmarking with Graphalytics involves four steps. *Add graphs.* We provide a set of graphs for download through the Graphalytics website. We also provide configuration files associated with these graphs. Alternatively, users can generate synthetic graphs using Datagen. In this case, users must write their own configuration files. *Configure the platform.* Users must setup the platforms and configure Graphalytics according to this. For example, for platforms running on top of Hadoop, the `HADOOP_HOME` path must be set. *Choose the workload.* By default, Graphalytics runs all the algorithms implemented on all configured graphs. If users want to run a subset of the algorithms, they must define a *run* that includes only the algorithms and graphs of interest. *Run the benchmark.* Graphalytics includes a Unix shell script that triggers the execution of the benchmark. After the execution completes, the benchmark report is available in the local file system.

3. GRAPHALYTICS AFTER 9 MONTHS: EXPERIMENTAL RESULTS

Although Graphalytics is still in an early phase of development, it has already enabled us to enrich our previous graph benchmarking results with new datasets and platforms. Moreover, it enables us to benchmark platforms on different clusters and collect the results without minimal configuration and no source code modifications.

3.1 Datagen scalability

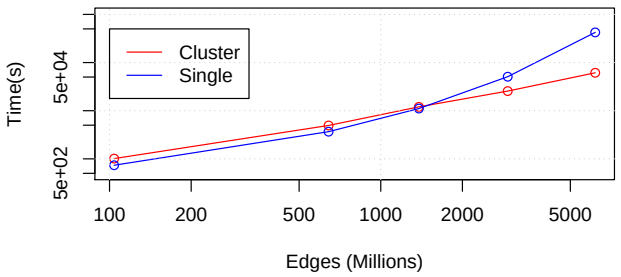

Figure 3: Scalability of Datagen.

For Graphalytics we are only interested in a subset of the social network data generated by Datagen, namely the person-knows-person graph, so we disabled the generation of other data. Figure 3 shows the times to generate increasingly larger graphs, on two different systems. The first system consists of a small cluster with 4 nodes, each of them with an Intel Xeon E5530 2.40 GHz 4-core CPU, 32 GiB RAM and 2 TB HDD each. The second is a single node, more modern machine with 2 Intel Xeon E5-2630 v3 2.40 GHz 8-core CPUs, 256 GiB RAM and a 2 TB HDD. In the first case, executions were performed using 8 cores, while in the second where done using 16 cores, as these configurations provided the best performance. On the one hand, the single node machine is faster than the cluster for smaller graphs, were computation is mostly CPU bound. It can generate a 1.3B edge graph in about 3 hours. This means thats with a relatively affordable machine, one can generate graphs of a considerable size. On the other hand, even though the cluster is slower for smaller graphs, it provides better scalability when data size grows as the computations become I/O bound, thanks to the greater disk bandwidth provided by the four disks. Therefore, one can increase the size of a cluster horizontally with relatively cheap hardware to generate large datasets.

3.2 Supported Algorithms and Platforms

We have included so far in Graphalytics five algorithms that are representative for real-world usage and stress the choke points of platforms. The **general statistics** (STATS) algorithm counts the numbers of vertices and edges in the graph and computes the mean local clustering coefficient. The **breadth-first search** (BFS) algorithm traverses the graph starting from a seed vertex, visiting first all the neighbors of a vertex before moving to the neighbors of the neighbors. The **connected components** (CONN) algorithm determines for each vertex the connected component it belongs to. The **community detection** (CD) algorithm detects groups of nodes that are connected to each other stronger than they are connected to the rest of the graph [12]. The **graph evolution** (EVO) algorithm predicts the evolution of the graph according to the "forest fire" model [11].

We provide in Graphalytics implementations for the following four graph-processing platforms. **Hadoop MapReduce** is an Apache open-source project implementing the MapReduce programming model introduced by Google [2]. Specifically, we use Hadoop MapReduce version 2, which runs on top of the Hadoop YARN resource manager. **Giraph** is an Apache open-source project implementing the Pregel programming model introduced by Google [14]. In Pregel, a type of bulk synchronous parallel processing (BSP), computation is vertex-centric and progresses in steps sepa-

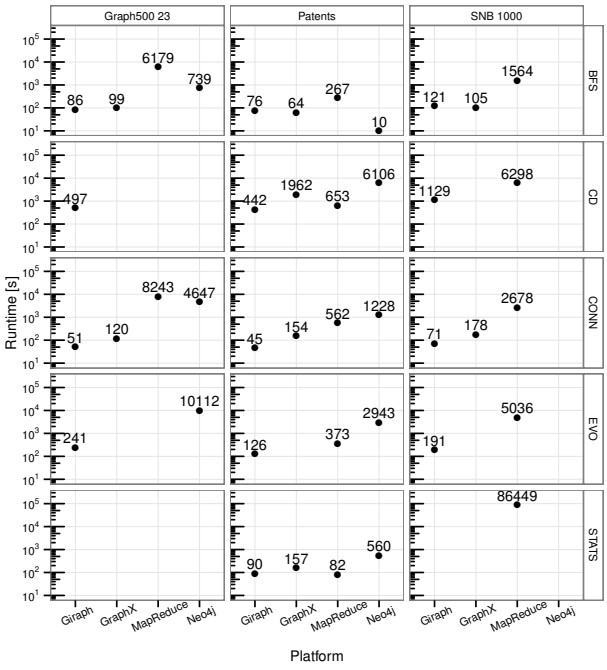

Figure 4: Runtimes for all implementations of all algorithms running on Graph500 23, Patents, and SNB 1000 graphs. Missing values indicate failures. Logarithmic vertical axis.

rated by synchronization barriers. All vertices execute the same function in parallel during a computation step, using as input messages received from other vertices. **GraphX** [21] is a graph-processing library built on top of the generic Apache Spark distributed processing platform. GraphX represents graphs as Spark resilient distributed datasets (RDDs) and provides built-in operation such as retrieving the number and degree of vertices. Additionally, GraphX supports iterative algorithms implemented according to the Pregel programming model. **Neo4j** is an open-source non-distributed graph database. We include it in Graphalytics to provide perspective on the performance and scalability of the distributed platforms we benchmark. Neo4j is not able to process graphs larger than the memory of a single machine, but its performance is generally the best due to its non-distributed nature.

3.3 Results for the Supported Platforms

In Figure 4, we summarize the runtimes of the currently supported Graphalytics platforms on several graphs. The runtime measures the complete execution of an algorithm, from job submission to result availability, but does not include ETL. Comparing ETL times of different platforms is left as future work. For MapReduce, Giraph, and GraphX we use 11 machines with 24 GiB RAM and dual Xeon E5620 CPUs, of which 10 are used for HDFS and computation, and 1 is used as a master node for Hadoop, ZooKeeper, and our benchmarking system. Neo4j is benchmarked on a machine with 192 GiB RAM and dual Xeon E5-2450 v2 CPUs.

Analyzing the results, we note that GraphX is significantly slower that Giraph for the CONN algorithm ($\sim 3\times$), although our implementation uses the built-in GraphX connected components algorithm. At the same time, GraphX is unable to process some of the workloads that Giraph can

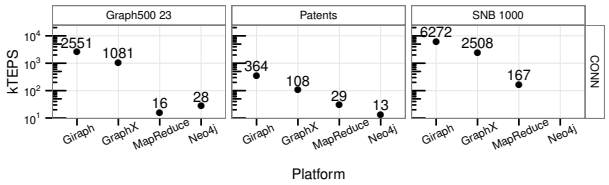

Figure 5: Thousands of traversed edges per second (kTEPS) for all implementations of CONN algorithm running on Graph500 23, Patents, and SNB 1000 graphs. Missing values indicate failures. Logarithmic vertical axis.

process, indicated by missing values in the figure. This is surprising considering they both use the Java virtual machine and should be using similar amounts of memory. At the same time, MapReduce can process all the workloads, if given enough time. In our experiments, MapReduce can be two orders of magnitude slower than Giraph and GraphX (e.g., respectively 6179 s, 86 s, and 99 s for BFS on Graph500). However, MapReduce does not need to keep graph data in memory during processing and thus does not crash even when processing the largest workload. Due to time constraints, MapReduce was not able to complete some algorithms on Graph500.

In Figure 5, we show the CONN performance of all platforms using the traversed edges per second (TEPS) metric. The size of the processed graph is included in this metric, which reveals the influence of the graph characteristics on performance. For example, we note that Giraph is more than an order of magnitude faster computing the connected components in the SNB 1000 graph than in the Patents graph (6272 kTEPS vs. 364 kTEPS).

3.4 BFS on a DBMS

We use the OpenLink Virtuoso column store to experiment with performance dynamics of BFS graph traversal in a DBMS. Virtuoso features column-wise compression, vectored execution, and intra-query parallelism with optimized partitioned aggregation. Virtuoso supports SQL and SPARQL and offers an SQL extension for transitive traversal, which we use in this experiment. The operation counts the reachable vertices starting from a single point (vertex ID 420). The query is:

```
select count (*) from (select spe_to from
    (select transitive t_in (1) t_out (2) t_distinct
     spe_from, spe_to from sp_edge) derived_table_1
   where spe_from = 420) derived_table_2;
```

The transitive modifier of derived_table_1 causes each value of the output column spe_to to be recycled as a binding for spe_from, the input column for deriving the next set of reachable vertices. The state of the computation is kept in a partitioned hash table, with one thread reading/writing each partition, with an exchange operator between the lookup of outbound edges and the recording of the new border, as the source and target of any edge most often fall in a different partition.

The dataset is SNB 1000. The query execution profile shows 2.28×10^6 random lookups (getting the outbound edges of a vertex), with 2.89×10^8 edges end points visited. The query takes 7 s on a 12-core, 24-thread dual Xeon E5-2630 (2.3 GHz). This gives a rate of 41.3 MTEPS. The

CPU utilization is 1930% (out of 2400% max). The CPU profile indicates 33% of cycles on the hash table containing the border, 10% for the exchange operator (get partition hash of a vector, split into per partition vectors by hash) and the remaining 57% for column store random access and decompression.

3.5 Code Quality

Traditional benchmarks are a set of specifications (process, benchmarking kernels, input data, etc.) accompanied by reference implementations for the supported platforms. Unfortunately, while the standards of the specifications have constantly improved, the standards for the reference implementations have not. We believe that modern software engineering practices should be used: in Graphalytics, the code for the reference implementations is accompanied by code quality reports, such as code complexity, bugs discovered through static analysis, etc.

All significant modifications to Graphalytics are peer reviewed by developers and are automatically tested by our Jenkins continuous integration server. Furthermore, all code commits are statically analyzed by SonarQube, which automatically signals regressions, such as an increase in the number of potential bugs.

4. RELATED WORK

We have already compared, throughout this work, the Graphalytics benchmark with other benchmarks proposed for graph-processing [7, 13, 22]. In summary, Graphalytics is much more comprehensive and ambitious than previous work: it supports more diverse and realistic datasets [4, 16], more diverse and realistic algorithms [4], and reference implementations for more platforms (preliminary results obtained for 10 platforms [4, 5]). Moreover, Graphalytics includes in its vision a fundamental understanding of choke points, extensions to the dataset generation, and an advanced benchmarking harness that will evolve into a public databased of useful results.

5. CONCLUSION

Benchmarking graph-processing platforms enables system comparison, tuning, and (re-)design for increasingly more domains. Responding to a dearth of comprehensive benchmarking approaches for graph-processing platforms, in this work we have proposed our vision: Graphalytics.

Conceptually, Graphalytics focuses on diverse datasets and algorithms, and methodologically it greatly extends the shortcomings of related work. Novel from previous work, including our own, Graphalytics focuses on a fundamental understanding of choke points, extensions to the dataset generation, and an advanced benchmarking harness that will evolve into a public database of useful results.

The Graphalytics vision is also pragmatic. The reference Graphalytics implementation covers currently 4 popular platforms, and will soon include 6 more platforms for which we already have shown proof-of-concept implementations [4, 5]. All reference implementations follow an advanced software engineering process, in which software quality metrics are monitored through continuous integration. Graphalytics aims to become an accepted benchmarking standard by both the LDBC and the SPEC Research Group communities, and attract further implementations from the creators of graph-processing platforms themselves. Graphalytics is hosted at `http://graphalytics.ewi.tudelft.nl`.

6. REFERENCES

[1] M. Dayarathna and T. Suzumura. Graph database benchmarking on cloud environments with xgdbench. *Autom. Softw. Eng.*, 21(4):509–533, 2014.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[3] B. Elser and A. Montresor. An evaluation study of BigData frameworks for graph processing. In *IEEE International Conference on Big Data*, pages 60–67. IEEE, Oct. 2013.

[4] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 395–404. IEEE, May 2014.

[5] Y. Guo, A. L. Varbanescu, A. Iosup, and D. H. J. Epema. An empirical performance evaluation of gpu-enabled graph-processing systems. In *CCGRID*, pages 927–932, 2015. (in print, available online: `http://www.pds.ewi.tudelft.nl/~iosup/perf-eval-gpu-graph-processing15ccgrid.pdf`).

[6] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Benchmarking Graph-Processing Platforms: A Vision. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 289–292. ACM Press, 2014.

[7] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An Experimental Comparison of Pregel-like Graph Processing Systems. In *VLDB*, 2014.

[8] C. Herrera and P. J. Zufiria. Generating scale-free networks with adjustable clustering coefficient via random walks. *arXiv preprint arXiv:1105.3347*, 2011.

[9] A. Iosup, A. L. Varbanescu, M. Capotă, T. Hegeman, Y. Guo, W. L. Ngai, and M. Verstraaten. Towards Benchmarking IaaS and PaaS Clouds for Graph Analytics. In *Workshop on Big Data Benchmarking (WBDB)*, Potsdam, Germany, 2014.

[10] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *J Mach Learn Res*, 11:985–1042, 2010.

[11] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *ACM SIGKDD*, 2005.

[12] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft. Towards real-time community detection in large networks. *Phys. Rev. E*, 79:066107, Jun 2009.

[13] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation. In *VLDB*, 2014.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *ACM International Conference on management of data (SIGMOD)*, page 135. ACM Press, 2010.

[15] M. Pham, P. A. Boncz, and O. Erling. S3G2: A scalable structure-correlated social graph generator. In *TPCTC*, 2012.

[16] A. Prat-Pérez and A. Averbuch. Benchmark design for navigational pattern matching benchmarking. Deliverable 3.3.34, LDBC, October 2014. [Online] Available: `http://ldbc.eu/sites/default/files/LDBC_D3.3.34.pdf`.

[17] A. Prat-Pérez and D. Dominguez-Sal. How community-like is the structure of synthetically generated graphs? In *GRADES*, pages 7:1–7:9. ACM, 2014.

[18] A. Prat-Pérez, D. Dominguez-Sal, and J. Larriba-Pey. Social based layouts for the increase of locality in graph operations. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2011.

[19] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[20] E. Volz. Random networks with tunable degree distribution and clustering. *Physical Review E*, 70(5):056115, 2004.

[21] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *GRADES*, pages 1–6. ACM Press, 2013.

[22] Y. Zhao, K. Yoshigoe, M. Xie, and S. Zhou. Evaluation and Analysis of Distributed Graph-Parallel Processing Frameworks. *Journal of Cyber Security and Mobility*, 3:289–316, 2014.