# LDBC Graphalytics Benchmark v0.2.6 - Draft Release

**Coordinator: Alexandru Iosup (Delft University of Technology (TUD))**

**With contributions from: Tim Hegeman (TUD), Wing Lung Ngai (TUD), Stijn Heldens (TUD), Arnau Prat Pérez (UPC), Thomas Manhardt (Oracle), Siegfried Depner (Oracle), Hassan Chafi (Oracle), Mihai Capotă (Intel), Narayanan Sundaram (Intel), Michael Anderson (Intel), Ilie Gabriel Tănase (IBM), Yinglong Xia (Huawei), Lifeng Nai (Georgia Tech), Peter Boncz (VUA).**

***Abstract***

In this document we describe LDBC Graphalytics, an industrial-grade benchmark for graph analysis platforms. The main goal of Graphalytics is to enable the fair and objective comparison of graph analysis platforms. Due to the diversity of bottlenecks and performance issues such platforms need to address, Graphalytics consists of a set of selected deterministic algorithms for full-graph analysis, standard graph datasets, synthetic dataset generators, and reference output for validation purposes. Its test harness produces deep metrics that quantify multiple kinds of systems scalability, weak and strong, and robustness, such as failures and performance variability. The benchmark also balances comprehensiveness with runtime necessary to obtain the deep metrics. The benchmark comes with open-source software for generating performance data, for validating algorithm results, for monitoring and sharing performance data, and for obtaining the final benchmark result as a standard performance report.

## Executive Summary

Processing graphs, especially at large scale, is an increasingly important activity in a variety of business, engineering, and scientific domains. Tens of very different graph-processing platforms, such as Giraph, GraphLab, and even the generic Hadoop, can already be used for this purpose. For graph-processing platforms to be adopted and to continue their evolution, users must be able to select with ease the best-performing graph-processing platform, and developers and system integrators have to find it easy to quantify the non-functional aspects of the system, from performance to scalability. Compared to traditional benchmarking, benchmarking graph-processing platforms must provide a diverse set of kernels (algorithms), provide recipes for generating diverse yet controlled datasets at large scale, and be portable to diverse and evolving platforms.

LDBC's Graphalytics is a combined industry and academia initiative, formed by principal actors in the field of graph-like data management. The main goal of LDBC Graphalytics is to define a benchmarking framework, and the associated open-source software tools, where different graph-processing platforms and core graph data management technologies can be fairly tested and compared. The key feature of Graphalytics is the understanding of the irregular and deep impact that dataset and algorithm diversity can have on performance, leading to bottlenecks and performance issues. For this feature, Graphalytics proposes a set of selected deterministic algorithms for full-graph analysis, standard graph datasets, synthetic dataset generators, and reference output for validation purposes. Furthermore, the Graphalytics test harness can be used to conducts diverse experiments and produce deep metrics that quantify multiple kinds of systems scalability, weak and strong, and robustness, such as failures and performance variability. The benchmark also balances comprehensiveness with runtime necessary to obtain the deep metrics. Because issues change over time, LDBC Graphalytics also proposes a renewal process.

Overall, Graphalytics aims to drive not only the selection of adequate graph processing platforms, but also help with system tuning by providing data for system-bottleneck identification, and with feature and even system (re-)design by providing quantitative evidence of the presence of sub-optimal designs. To this end, the development of the benchmark follows and extends the guidelines set by previous LDBC benchmarks, such as LDBC SNB.

To increase adoption by industry and research organizations, LDBC Graphalytics provides all the necessary software to run its comprehensive benchmark process. The open-source software contains tools for generating performance data, for validating algorithm results, and or monitoring and sharing performance data. The software is designed to be easy to use and deploy at a small cost. Last, the software is developed using modern software engineering practices, with low technical debt, high quality of code, and deep testing and validation processes prior to release.

This preliminary version of the LDBC Graphalytics specification contains a formal definition of the benchmarking framework and of its components, a description of the benchmarking process, links to the open-source software including a working benchmarking harness and drivers for several vendor- and community-driven graph-processing platforms, pseudo-code for all algorithms included in the benchmark, and a comparison with related tools, products, and concepts in the benchmarking space.

# Table of Contents

# List of Figures

# List of Tables

In this work we introduce the LDBC Graphalytics benchmark, explaining the motivation for creating this benchmark suite for graph analytics platforms, its relevance to industry and academia, the overview of the benchmark process, and the participation of industry and academia in developing this benchmark. A scientific companion to this technical specification has been published in 2016 [25].

## 1.1 Motivation for the Benchmark

Responding to increasingly larger and more diverse graphs, and the need to analyze them, both industry and academia are developing and tuning graph analysis software platforms. Already tens of such platforms exist, but their performance is often difficult to compare. Moreover, the random, skewed, and correlated access patterns of graph analysis, caused by the complex interaction between input datasets and applications processing them, expose new bottlenecks on the hardware level, as hinted at by the large differences between Top500 and Graph500 rankings (See Appendix B for the related work). Therefore, addressing the need for fair, comprehensive, standardized comparison of graph analysis platforms, in this work we propose the LDBC Graphalytics benchmark.

## 1.2 Relevance to Industry and Academia

A standardized, comprehensive benchmark for graph analytics platforms is beneficial to both industry and academia. Graphalytics allows a comprehensive, fair comparison across graph analysis platforms. The benchmark results provids insightful knowledge to users and developer on performance tuning of graph processing, and increases understanding of the advantage and disadvantages of design and implementation, therefore stimulating academic research in graph data storage, indexing, and analysis. By supporting platform variety, it reduces the learning curve of new users to graph processing systems.

## 1.3 General Benchmark Overview

This benchmark suite evaluates the performance of graph analysis platforms that facilitate complex and holistic graph computations. This benchmark must suffice to the requiriments of (1) targeting platforms and systems, (2) incoporating diverse, representative benchmark elements, (3) using a diverse, representative process, (4) including a renewal process, and (5) developed under modern software engineering.

In the benchmark (See Chapter 2 for the formal definition), we carefully motivate the choice of our algorithms and datasets to conduct our benchmark experiments. Graphalytics consists of six core algorithms: breadth-first search, PageRank, weakly connected components, community detection using label propagation, local clustering coefficient, and single-source shortest paths. The workload includes real and synthetic datasets, which are classified into intuitive "T-shirt" sizes (e.g., XS, S, M, L, XL). The benchmarking process is made future-proof, through a *renewal process*.

Each system under test undergoes the benchmark process (See Chapter 3) which consists of a baseline experiment, a scalabiltiy experiment, and a robustness experiment. Our test harness characterizes performance and *scalability* with deep metrics (strong vs. weak scaling), and also characterizes *robustness* by measuring SLA compliance, performance variability, and crash points.

## 1.4 Participation of Industry and Academia

The Linked Data Benchmark Council (`ldbcouncil.org`, LDBC), is an industry council formed to establish standard benchmark specifications, practices and results for *graph data management systems*. The list of institutions that take part in the definition and development of LDBC-Graphalytics is formed by relevant actors from both

the industry and academia in the field of large-scale graph processing. As of September 1, 2016, the list of participants is as follows:

- CWI AMSTERDAM, the Netherlands
- DELFT UNIVERSITY OF TECHNOLOGY, the Netherlands
- GEORGIA TECH, USA
- HUAWEI RESEARCH AMERICA, USA
- INTEL LABS, USA
- ORACLE LABS, USA
- UPC BARCELONA, Spain

## 2.1 Requirements

The Graphalytics benchmark is the result of a number of design choices.

**(R1) Target platforms and systems:** benchmarks must support any graph analysis platform operating on any underlying hardware system. For platforms, we do not distinguish between programming models and support any model. For systems, we target the following environments: multi-core and many-core single-node systems, systems with accelerators (GPUs, FPGAs, ASICs), hybrid systems, and distributed systems that possibly combine several of the previous types of environments. Without R1, a benchmark could not service a diverse industrial following.

**(R2) Diverse, representative benchmark elements:** data model and workload selection must be representative and have a good coverage of real-world practice. In particular, the workload selection must include datasets and algorithms which cover known system bottlenecks and be representative in the current and near-future practice. Without representativeness, a benchmark could bias work on platforms and systems towards goals that are simply not useful for improving current practice. Without coverage, a benchmark could push the community into pursuing cases that are currently interesting for industry, but not address what could become impassable bottlenecks in the near-future.

**(R3) Diverse, representative process:** the set of experiments conducted by the benchmark automatically must be broad, covering the main bottlenecks of the target systems. In particular, the target systems are known to raise various scalability issues, and also, because of deployment in real-world clusters, be prone to various kinds of failures, exhibit performance variability, and overall have various robustness problems. The process must also include validation of results, thus making sure the processing is done correctly. Without R3, a benchmark could test very few of the diverse capabilities of the target platforms and systems, and benchmarking results could not be trusted.

**(R4) Include renewal process:** unlike many other benchmarks, benchmarks in the area of graph processing must include a renewal process, that is, not only a mechanism to scale up or otherwise change the workload to keep up with increasing more powerful systems, but also a process to automatically configure the mechanism, and a way to characterize the reasonable characteristics of the workload for an average platform running on an average system. Without R4, a benchmark could become less relevant for the systems of the future.

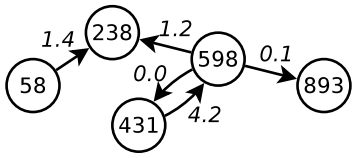
**(R5) Modern software engineering:** benchmarks must include a modern software architecture and run a modern software-engineering process. The Graphalytics benchmark is provided with a extensive benchmarking suite that allows users to easily add new platforms and systems to test. This makes it possible for practitioners to easily access the benchmarks and compare their platforms and systems against those of others. Without R5, a benchmark could easily become unmaintainable or unusable.

## 2.2 Data

The Graphalytics benchmark operates on a single type of dataset: graphs. This section provides the definition of a graph, the definition of the representation used by Graphalytics for its input and output data, and the datasets used for the benchmarks.

### 2.2.1 Definition

Graphalytics does not impose any requirements on the semantics of the graph and the benchmark uses a typical data model for graphs. A graph consists of a collection of *vertices* which are linked by *edges*. Each vertex




(a) Graph

```
58
238
431
598
893
```

(b) Vertex file

```
58 238 1.4
431 598 4.2
598 238 1.2
598 431 0.0
598 893 0.1
```

(c) Edge file

Figure 2.1: Example of directed weighted graph in EVLP format.

is assigned a unique identifier represented as a signed 64-bit integer, i.e., between $-2^{63}$ and $2^{63} - 1$. Vertex identifiers do not necessary start at zero, nor are the identifiers consecutive. Edges are represented as pairs of vertex identifiers. Graphalytics supports both *directed* graphs (i.e., edges are unidirectional) and *undirected* graphs (i.e., edges bidirectional). Every edge is unique and connects two distinct vertices. This implies self-loops (i.e., vertices having edges to them self) and multi-edges (i.e., multiple edges between a pair of vertices) are not allowed. For undirected graphs, every pair of vertices $u$ and $v$, the edges $(u, v)$ and $(v, u)$ are considered to be identical.

Vertices and edges can have properties which can be used to store meta-data such as weights, timestamps, labels, costs, and durations. Currently, graphaltyics supports three types of properties: *integers*, *floating-point numbers*, and *booleans*. All floating-point numbers must be internally stored and handled in 64-bit double precision IEEE-754 format. We explicitly do not allow the single precision format, as this can speedup computation and presents and unfair advantage. Integers can be stored and processed in any format which seems suitable by the system.

### 2.2.2 Representation

In Graphalytics, the file format used to represent the graphs is the "Edge/Vertex-List with Properties" (*EVLP*) format for graphs. Both formats consist of two human-readable files: a vertex-file containing the vertices (with optional properties) and an edge-list containing the edges (with optional properties). Both files are encoded using ASCII and consist of a sequence of lines.

For the vertex files, each line contains exactly one vertex id. The vertex identifiers are sorted in ascending order to facilitate easy conversion to other formats. For the edge files, each line contains two vertex identifiers separated by a space. The edges are sorted in lexicographical order of the two endpoints. For directed graphs, the endpoints correspond to the source and destination vertex, respectively. For undirected graphs, the smallest identifier of the two vertices is listed first and each edge is only listed once in one direction.

Vertices and edges can have optional properties. These values of these properties are listed in the vertex/edge files after each vertex/edge and are separated by spaces. The interpretation of these properties is not provided by the files.

Figure 2.1 shows an example of the EVLP format for a small directed weighted graph consisting of 5 vertices and 5 directed edges.

### 2.2.3 Datasets

Graphalytics includes both graphs from real-world applications and synthetic graphs which are created using graph generators. Graphalytics uses a broad range of graphs with a large variety in domain, size, densities, and characteristics. To facilitate performance comparison across datasets, the *scale* of a graph is derived from the number vertices and number of edges. Formally, the scale of a graph is defined by calculating the sum of the number of vertices ($n$) and number of edges ($m$), taking the logarithm of this value in base 10, and truncating the result to one decimal place. Formally, this can be written as follows:

$$Scale(n, m) = \lfloor 10 \log_{10}(n + m)) \rfloor / 10 \tag{2.1}$$

Table 2.1: Mapping of dataset scales ("T-shirt sizes") in Graphalytics.

| Label | Scales |
|---|---|
| XXS | $6.5 - 6.9$ |
| XS | $7.0 - 7.4$ |
| S | $7.5 - 7.9$ |
| M | $8.0 - 8.4$ |
| L | $8.5 - 8.9$ |
| XL | $9.0 - 9.4$ |
| XXL | $9.5 - 10.0$ |

Table 2.2: Real-world datasets used by Graphalytics.

| ID | Name | $n$ | $m$ | Scale | Domain |
|---|---|---|---|---|---|
| R1(2XS) | wiki-talk [2] | 2.39 M | 5.02 M | 6.9 | Knowledge |
| R2(XS) | kgs [19] | 0.83 M | 17.9 M | 7.3 | Gaming |
| R3(XS) | cit-patents [2] | 3.77 M | 16.5 M | 7.3 | Knowledge |
| R4(S) | dota-league [19] | 0.06 M | 50.9 M | 7.7 | Gaming |
| R5(XL) | com-friendster [2] | 65.6 M | 1.81 B | 9.3 | Social |
| R6(XL) | twitter_mpi [9] | 52.6 M | 1.97 B | 9.3 | Social |

Table 2.3: Synthetic datasets used by Graphalytics generated using the Graph500 generator.

| ID | Name | $n$ | $m$ | Scale |
|---|---|---|---|---|
| G16(3XS) | Graph500-16 | 66k | 1.0M | 6.0 |
| G17(3XS) | Graph500-17 | 0.13M | 2.1M | 6.3 |
| G18(2XS) | Graph500-18 | 0.26M | 4.2M | 6.6 |
| G19(2XS) | Graph500-19 | 0.52M | 8.4M | 6.9 |
| G20(XS) | Graph500-20 | 1.0M | 16.8M | 7.2 |
| G21(S) | Graph500-21 | 2.1M | 33.6M | 7.5 |
| G22(S) | Graph500-22 | 4.2M | 67.1M | 7.8 |
| G23(M) | Graph500-23 | 8.4M | 0.13B | 8.1 |
| G24(M) | Graph500-24 | 16.8M | 0.27B | 8.4 |
| G25(L) | Graph500-25 | 33.6M | 0.54B | 8.7 |
| G26(XL) | Graph500-26 | 67.1M | 1.1B | 9.0 |
| G27(XL) | Graph500-27 | 0.13B | 2.1B | 9.3 |
| G28(2XL) | Graph500-28 | 0.27B | 4.3B | 9.6 |
| G29(2XL) | Graph500-29 | 0.54B | 8.6B | 9.9 |
| G30(3XL) | Graph500-30 | 1.1B | 17.1B | 10.2 |

The scale of a graph gives users and intuition of what the size of graph means in practice. Scales are grouped into classes spanning 0.5 *scale units* and these classes are labeled using familiar system of "T-shirt sizes": small (S), medium (M), and large (L), with extra (X) prepended for extremes scales. The reference point is class L, which is roughly defined by the Graphalytics team as the largest class such that the BFS algorithm completes within an hour on any graph from that scale using a state-of-the-art graph analysis platform and a single commodity machine. Table 2.1 summarizes the classes used in Graphalytics.

### 2.2.4 Real-world Datasets

Table 2.2 lists the real-world datasets used by the Graphalytics benchmark.

Table 2.4: Synthetic datasets used by Graphalytics generated using the LDBC Datagen generator.

| ID | Name | $n$ | $m$ | Scale |
|---|---|---|---|---|
| D100(M) | Datagen-100 | 1.67M | 102M | 8.0 |
| D300(L) | Datagen-300 | 4.35M | 304M | 8.5 |
| D1000(XL) | Datagen-1000 | 12.8M | 1.01B | 9.0 |
| D3000(XL) | Datagen-3000 | 34.8M | 3.01B | 9.4 |

### 2.2.5 Dataset Generators

Besides real-world datasets, Graphalytics adopts two commonly used generators that generate two types of graphs: power-law graphs from the **Graph500** generator [10, 29] and social network graphs from **LDBC Datagen** [14]. Table 2.3 lists the graph500 datasets and Table 2.4 lists the Datagen datasets, together with corresponding Graphalytics scales.

## 2.3 Algorithms

The Graphalyics benchmark consists of six algorithms which need to be executed on the differen datasets: five algorithms for unweighted graphs and one algorithm for weighted graphs. These algorithms have been selected based on the results of multiple surveys and expert advice from LDBC TUC. Below abstract descriptions are provided for the six algorithms

Each workload of graphalytics consists of executing a single algorithm on a single dataset. Graphalytics consists of six algorithms: five for unweighted graphs and one for weighted graphs. These algorithms can be considered to be representative for graph analysis in general. Below, abstract descriptions are provided for the six algorithms and pseudo-code is given in Appendix A, however, Graphalytics does not impose any constraint on the implementation of algorithms. Any implementation is allowed, as long as its correctness can be validated by comparing its output to correct reference output (Section 2.4).

In the following sections, a graph $G$ consists of a set of vertices $V$ and a set edges $E$. For undirected graphs, each edge is bidirectional, so if $(u, v) \in E$ then $(v, u) \in E$. Each vertex $v$ has a set of outgoing neighbors $N_{out} = \{u \in V | (v, u)\}$, a set of incoming neighbors $N_{in} = \{u \in V | (u, v)\}$, and a set of all neighbors $N(v) = N_{in}(v) \cup N_{out}(u)$. Note that for undirected graphs, each edge is bidirectional so we have $N_{in}(v) = N_{out}(v) = N(v)$.

### 2.3.1 Breadth-First Search (BFS)

*Breadth-First Search* (BFS) is a traversal algorithm that labels each vertex of a graph with the length (or *depth*) of the shortest path from a given source vertex (*root*) to this vertex. The root has depth 0, its outgoing neighbors have depth 1, their outgoing neighbors have depth 2, etc. Unreachable vertices should be given the special value of $-1$.

### 2.3.2 PageRank (PR)

*PageRank* (PR) is an iterative algorithm that assigns to each vertex a ranking value. The algorithm was originally used by Google Search to rank websites in their search results [31]. Let $PR_i(v)$ be the PageRank value of vertex $v$ after iteration $i$. Initially, each vertex $v$ is assigned the same value $1/|V|$ such that the sum of all vertex values is 1.

$$PR_0(v) = \frac{1}{|V|} \tag{2.2}$$

After iteration $i$, each vertex pushes its PageRank over its outgoing edges to its neighbors. The PageRank for each vertex is updated according to the following rule:

$$PR_i(v) = \frac{1-d}{|V|} + d \sum_{u \in N_{in}(v)} \frac{PR_{i-1}(u)}{|N_{out}(u)|} + d \sum_{u \in D} \frac{PR_{i-1}(u)}{|V|} \tag{2.3}$$

Where $d \in [0, 1]$ is called the *damping factor* and $D = \{v \in V \mid |N_{out}(v)| = 0\}$ is the set of *dangling vertices*, i.e., vertices having no outgoing edges. Dangling vertices have nowhere to push their PageRank to, so the total sum of the PageRanks for the dangling vertices is evenly distributed over all vertices.

The PageRank algorithm should continue for a fixed number of iterations. The floating-point values must be handled as 64-bit double precision IEEE 754 floating-point numbers.

### 2.3.3 Weakly Connected Components (WCC)

The *Weakly Connected Components* algorithms finds the weakly connected components in a graph and assigns each vertex a unique label that indicates which component it belongs to. Two vertices belong to the same component, and thus have the same label, if there exists a path between these vertices along the edges of the graph. For directed graphs, it is allowed to travel over the reverse direction of an edge, .e., the graph is interpreted as if it is undirected.

### 2.3.4 Community Detection using Label-Propagation (CDLP)

The *community detection* algorithm in Graphalytics uses *label propagation* (CDLP) and is based on the algorithm proposed by Raghavan et al. [33]. The algorithm assigns each vertex a label, indicating its community, and these labels are iteratively updated where each vertex is assigned a new label based on the frequency of the labels of its neighbors. The original algorithm has been adapted to be both deterministic and parallel, thus enabling output validation and parallel execution.

Let $L_i(v)$ be the label of vertex $v$ after iteration $i$. Initially, each vertex $v$ is assign a unique label which matches its identifier.

$$L_0(v) = v \tag{2.4}$$

In iteration $i$, each vertex $v$ determines the frequency of the labels of its incoming and outgoing neighbors and select the label which is most common. If the graph is directed and a neighbor is reachable via both an incoming and outgoing edge, its label will be counted twice. In case there multiple labels with maximum frequency, the smallest label is chosen. In case a vertex has no neighbors, it retains its current label. This rule can be written as follows:

$$L_i(v) = \underset{l}{arg\,max} \Big[ |\{u \in N_{in}(v) \mid L_{i-1}(v) = l\}| + |\{u \in N_{out}(v) \mid L_{i-1}(v) = l\}| \Big] \tag{2.5}$$

Where ties are broken by choosing the lowest value for $l$.

### 2.3.5 Local Clustering Coefficient (LCC)

The *Local Clustering Coefficient* (LCC) algorithm determines the local clustering coefficient for each vertex. This coefficient indicates the ratio between the number of edges between the neighbors of a vertex and the maximum number of possible unique edges between the neighbors of this vertex. If the number of neighbors of a vertex is less than two, its coefficient is defined as zero. The definition of LCC can be written as follows:

$$LCC(v) = \begin{cases} 0 & \text{If } |N(v)| \leq 1 \\ \frac{|\{(u,w) | u,w \in N(v) \wedge (u,w) \in E\}|}{|(u,w) | u,w \in N(v)|} & \text{Otherwise} \end{cases} \tag{2.6}$$

Note that the second case can also be written using the sum over the neighbors of $v$.

$$LCC(v) = \frac{\sum_{u \in N(v)} |N(v) \cup N_{in}(u)|}{|N(v)|\big(|N(v)| - 1\big)} \tag{2.7}$$

```
1 3
2 1
3 2
4 0
5 1
```

(a) Reference output

```
1 3
2 1
3 2
4 0
5 1
```

(b) Example of correct result

```
1 4
2 1
3 2
4 5
5 1
```

(c) Example of incorrect result

Figure 2.2: Example of validation with *exact match*.

```
1 1
2 1
3 1
4 2
5 2
6 3
```

(a) Reference output

```
1 81
2 81
3 81
4 32
5 32
6 12
```

(b) Example of correct result

```
1 31
2 52
3 31
4 31
5 31
6 74
```

(c) Example of incorrect result

Figure 2.3: Example of validation with *equivalence match*.

```
1 0
2 0.3
3 0.45
4 0.23
5 +inf
6 0.001
```

(a) Reference output

```
1 0
2 0.30002
3 0.45
4 0.229997
5 +inf
6 0.001
```

(b) Example of correct result

```
1 0.000001
2 0.3
3 0.46
4 0.22
5 1.79769e+308
6 0
```

(c) Example of incorrect result

Figure 2.4: Example of validation with *epsilon match*.

### 2.3.6 Single-Source Shortest Paths (SSSP)

The *Single-Source Shortest Paths* algorithm (SSSP) marks each vertex with the length of the shortest path from a given *root* vertex to every other vertex in the graph. The length of a path is defined as the sum of the weights on the edges of the path. The edge weights are floating-point numbers which must be handled as 64-bit double precision IEEE-754 floating-point numbers. The weights are never negative, infinity, or invalid (i.e., *NaN*), but are allowed to be zero. Unreachable vertices should be given a value of `Infinity`.

## 2.4 Validation

The output of every execution of an algorithm on a dataset must be validated for the result to be admissible. All algorithms in the Graphalytics benchmark are deterministic and can therefore be validated by comparing to reference output for correctness. Three methods are used for validation:

- **Exact match (applies to BFS, CDLP):** the vertex values of the system's output should be identical to the reference output. Figure 2.2 shows an example of validation with exact match.
- **Equivalence (applies to WCC):** the vertex values of the system's output should be equal to the the reference output *under equivalence*. This means a two-way mapping should exists that maps the system's

output to be identical to reference output and the inverse of this mapping maps the reference output to be identical to the system's output. In other words, the output is considered to be valid if all vertices which have the same label in the system's output also have the same label in the reference output, and vice versa. Figure 2.3 shows an example of validation with equivalence.

- **Epsilon match (applies to PR, LCC, SSSP):** a slight margin of error is allowed for some algorithms due to floating-point rounding errors. Let $r$ be the reference value of vertex $v$ and $s$ the value of the vertex from the system's output. Then these values are considered to match if $r$ is within $0.01\%$ of $s$, i.e., the equation $|r-s| < \epsilon|r|$ must hold where $\epsilon = 0.0001$. The value of $\epsilon$ was chosen such that minor errors that result from rounding are not penalized. Figure 2.4 shows an example of validation with epsilon match.

## 2.5 Metrics

The Graphalytics benchmark includes several metrics to quantify the performance and other characteristics of the system under test. The performance of graph analytics systems is measured by the time spent on several phases in the execution of the benchmark. The following metrics are used:

**Upload time:** The time spent loading a particular graph into the system under test, including any preprocessing to convert the input graph to a format suitable for the system. This phase is executed once per graph before any commands are issued to execute specific algorithms.

**Makespan:** The time between the Graphalytics driver issuing the command to execute an algorithm on a (previously uploaded) graph and the output of the algorithm being made available to the driver. The makespan can be further divided into processing time and overhead.

**Processing time:** Time required to execute an actual algorithm. This does not include platform-specific overhead, such as allocating resources, loading the graph from the file system, or graph partitioning.

Other metrics and the characteristics they measure are:

**Speedup:** The ratio between processing times for scaled and baseline resources. This metric is used to quantify the scalability of the system under test. The baseline is defined as the minimum number of resources used in a particular experiment.

## 2.6 System Under Test

Responding to requirement R1 (see Section 2.1), the LDBC Graphalytics framework defines the System Under Test as the combined software and hardware platform that is able to execute graph-processing algorithms on graph datasets. This is an inclusive definition, and indeed Graphalytics has been executed in the lab of SUTs with software ranging from community-driven prototype systems, to vendor-optimized software; and with hardware ranging from beefy single-node multi-core systems, to single-node CPU and (multi-)GPU hybrid systems, to multi-node clusters with or without GPUs present.

## 2.7 Renewal Process

To ensure the relevance of Graphalytics as a benchmark for future graph analytics systems, a renewal process is included in Graphalytics. This renewal process updates the workload of the benchmark to keep it relevant for increasingly powerful systems and developments in the graph analytics community. This results in a benchmark which is future-proof. Renewing the benchmark means renewing the algorithms as well as the datasets. For every new version of Graphalytics, a two-stage selection process will be used by the LDBC Graph Analytics Task Force. The same selection process was used to derive the workload in the current version of the Graphalytics benchmark.

To achieve both workload representativeness and workload coverage, a two-stage selection process is used. The first stage identifies classes of algorithms and datasets that are representative for real-world usage of graph analytics systems. In the second stage, algorithms and datasets are selected from the most common classes such that the resulting selection is diverse, i.e., the algorithms cover a variety of computation and communication patterns, and the datasets cover a range of sizes and a variety of graph characteristics.

Updated versions of the Graphalytics benchmark will also include renewed definitions of the scale classes defined in Section 2.2.3. The definition of the scale classes is derived from the capabilities of state-of-the-art graph analytics systems and common-off-the-shelf machines, which are expected to be improved over time. Thus, graphs that are considered to be large as of the publication of the first edition of Graphalytics (labeled $L'16$ to indicate the 2016 edition) may be considered medium-sized graphs in the next edition (e.g., $M'18$).

# 3 Benchmark Process

This section describes the benchmark process of Graphalytics, which summarizes the benchmark lifecycle for each system under test. We discuss the overview of the benchmark process, the details of each platform run, the list of experiments, and the format of the benchmark results.

## 3.1 Overview

Each Graphalytics benchmark contains a set of experiments, each experiment evaluates specific performance characteristics of a system under test. A experiment consists of multiple benchmark jobs, each repeated in several identical platform runs to collect statistically reliable results.

Each experiment (See Figure 3.1) has the following setup: first, Graphalytics provisions the graph datasets needed at the proper location and in the proper data format as required by the platform; then, Graphalytics initializes (several) platform runs for each benchmark job defined in the experiment setup; finally Graphalytics aggregates performance results from the platform runs and registers results for each benchmark job.

To ensure that results obtained using Graphalytics represent reasonable use of graph analytics systems, an SLA applies to all platform runs. The Graphalytics SLA requires the system under test to provide valid output, as defined in Section 2.4, with a makespan that does not exceed one hour. If the SLA is not met for a particular platform run, that run is considered to have failed.

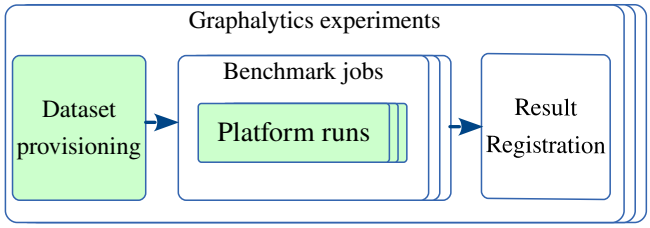



Figure 3.1: One experiment in the Graphalytics benchmark process.

## 3.2 Platform Run

Each platform run undergoes six stages: setup, loading, execution, offloading, cleanup, and validation (See Figure 3.2). Note that these stages may be overlapping or reorganized in alternative orders by different platforms. Each platform must report at least the start and end time for the execution stage to derive the processing time, $T_{proc}$. The makespan is measured directly by the Graphalytics harness and encompasses all stages executed by the system under test or its driver.

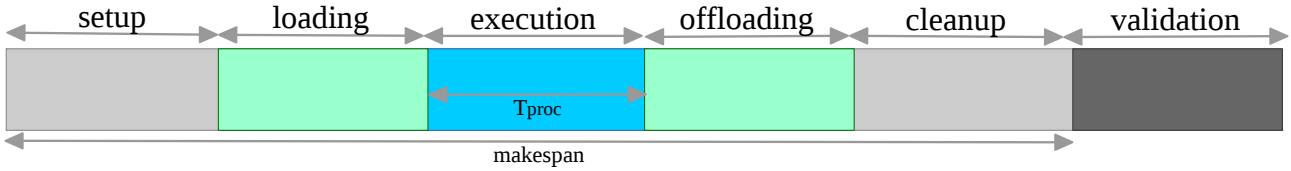



Figure 3.2: A platform run and the underlying suboperations.

### 3.2.1 Setup

During the "Setup" stage, the platform prepares itself to be ready for the algorithm execution. This usually involves provisioning computation resources and using the job parameters to properly setup the workers.

| **Category** | Experiment | **Algorithms** | **Datasets** |
|---|---|---|---|
| Baseline | Dataset variety | BFS, PR | All (-T) |
| | Algorithm variety | All | 2 (-T) |
| Scalability | Strong | BFS, PR | Datagen (T) |
| | Weak | BFS, PR | Graph500 (T+) |
| Robustness | Stress test | BFS | All |
| | Variability | BFS | Datagen (T) |

Table 3.1: Experiments included in Graphalytics. Dataset selection includes datasets of sizes up to and including the target scale (-T), at the target scale (T), or starting from the target scale (T+).

### 3.2.2 Loading

During the "Loading" stage, the platform loads the dataset into the memory space of its processes, and converts the datza and partitions them into an (optimized) in-memory structure.

### 3.2.3 Execution

During the "Execution" stage, the platform executes a graph algorithm on the in-memory graph data in multiple iterations. This is the most important stage of the platform run, and the runtime of which is defined as the processing time ($T_{proc}$) of a platform run.

### 3.2.4 Offloading

During the "Offloading" stage, the platform offloads the output from the memory space of its processes to the storage. The output must be convertible to the Graphalytics format.

### 3.2.5 Cleanup

During the "Cleanup" stage, the platform cleanup the environment, terminates its processes, and signals Graphalytics or other services that the run is completed.

### 3.2.6 Validation

During the "Validation" stage, Graphalytics collects the outputs from the platform, which is validated against the correct output. Only validated jobs are be considered to be successfully terminated.

## 3.3 List of Graphalytics Experiments

Graphalytics consists of several experiments designed to quantify several properties of the system under test. The selection of datasets is parameterized by a target scale (T), which allows the benchmark to be run on a variety of systems. The target scale is a label as defined in Section 2.2.3 and defines an upper limit on the scale of graphs to use for most experiments. The target scale is typically chosen by the benchmark user as the scale up to which they expect the system under test to successfully, e.g., target scale L for a common-off-the-shelf system. The experiments included in Graphalytics are summarized in Table 3.1.

### 3.3.1 Baseline Experiments

To establish the baseline performance of the system under test, Graphalytics includes two experiments for testing performance on a variety of datasets and algorithms. The benchmark jobs that make up the dataset variety experiment consist of running BFS and PR on all real-world datasets no larger than the target scale plus for every synthetic generator the largest dataset that is not larger than the target scale. For example, the datasets to

use for target scale L are R1(2XS), R2(XS), R3(XS), R4(S), G25(L), and D300(L). The benchmark jobs that make up the algorithm variety experiment consist of running all algorithms on the largest Datagen dataset no larger than the target scale and the largest real-world graph with the required properties to run all algorithms (either R2(XS) or R4(S)). For example, the datasets to use for target scale L are R4(S) and D300(L). Each benchmark job must be run three times. For each run the makespan and processing time must be reported. If any of the three runs fails the Graphalytics SLA, the corresponding benchmark job is considered to have failed.

To summarize the performance of the system under test, a single score is computed for each algorithm. For every successfully executed benchmerk job, the EVPS metric must be computed using the median processing time of three runs. The score assigned to an algorithm is the harmonic mean of the EVPS metric for all benchmark jobs for the algorithms. If any benchmark job for a particular algorithm fails, no score is assigned for the algorithm.

### 3.3.2 Scalability Experiments

For scalable systems, Graphalytics defines two experiments to evaluate the scalability of the system under test. Both experiments must be run with increasing amounts of hardware resources. Which resources are scaled depends on the architecture of the system under test and must be chosen and reported by the benchmark user. Typical examples include scaling the number of machines in a distributed system, scaling the number of processor cores in a parallel machine, or adding GPUs in a heterogeneous system. The system under test as used for the baseline experiments is considered the baseline for the scalability experiments. This system is assumed to have N resources for the definition of the scalability experiments.

The strong scalability experiment consists of running BFS and PR on the largest Datagen dataset no larger than the target scale using N, 2N, 4N, 8N, and 16N resources. The weak scalability experiment consists of running BFS and PR on Graph500 datasets of increasing sizes using N, 2N, 4N, 8N, and 16N resources. The largest Graph500 dataset no larger than the target scale must be used for all weak scalability benchmark jobs using N resources, while a Graph500 datasets x times larger is used for a system with xN resources. For example, target scale L requires G25(L) to be run on N resources, G26(XL) on 2N resource, G27(XL) on 4N resources, etc.

For both scalability experiments, all benchmark jobs must be run three times, the makespan and processing time must be reported for each run, and the median processing time must be taken as the processing time of the benchmark job. Per experiment and algorithm the speedup in processing time over using N resources must be reported when using 2N, 4N, 8N, or 16N resources. If any run fails to meet the SLA, no results for the corresponding experiment and algorithm may be reported.

### 3.3.3 Robustness Experiments

The robustness of the system under test is evaluated using two experiments targeted at the system's limits through stress testing and at its variability in performance. The workload of the stress test experiments consists of running BFS on all datasets in increasing order of size. The smallest dataset for which the system under test fails the SLA must be reported. If the system under test successfully completes BFS on two synthetic graphs of consecutive scales for a particular dataset generator, successful execution on all smaller graphs for the same generator may be assumed to reduce the number of jobs required to find the system's limit. The workload of the variability experiment consists of running BFS on the largest Datagen dataset no larger than the target scale, which must be repeated ten times. The makespan and processing time must be reported for every run. The variability score of the system under test is the coefficient of variation of the processing times. If any run fails the SLA, no variability score is assigned.

## 3.4 Benchmark Result

A complete result for the Graphalytics benchmark includes at least the following information (work in progress):

1. Target scale (T).

2. Environment specification, including number and type of CPUs, amount of memory, type of network, etc.
3. Versions of the platform and Graphalytics drivers used in the experiments.
4. Any non-default configuration options for the platform required to reproduce the system under test.
5. For every benchmark job:
   (a) Job specification, i.e., dataset and algorithm.
   (b) For every platform run, report the measured processing time, makespan, and whether the run breached the Graphalytics SLA.
   (c) (optional) `Granula` archives for each platform run, enabling deep inspection, visualization, modeling, and sharing of performance data.
6. If scalability experiments are performed:
   (a) Definition of the hardware resources that were scaled up for each scalability experiment.
   (b) For every benchmark job corresponding to a scalability experiment, include the resource scale (i.e., 1, 2, 4, 8, or 16).

Future versions of the benchmark specification will include a Full Disclosure Report template and a process for submitting official Graphalytics results.

## 3.5 Sample Configuration

LDBC Graphalytics arrives pre-configured, but also allows detailed configuration using configuration files. The configuration files are self-documented, and are part of the regular LDBC Graphalytics distribution available at:
`https://github.com/tudelft-atlarge/graphalytics`

Listing 3.1 describes a sample configuration of the benchmark. To manage complexity, configuration files have a hierarchical definition, with Listing 3.1 linking to configurations that describe which experiments to run (Listing 3.2) and which datasets to use in each experiment (Listing 3.3). Further configuration is possible, for example detailed configuration of each dataset (Listing 3.4).

Listing 3.1: The master-file for a benchmark, "benchmark.properties".

```
# Properties file describing the benchmark

# Include benchmark run properties file, which defines a subset of
# graphs and algorithms to run
include = runs/all.properties

# Include other properties files
include = graphs.properties
```

Listing 3.2: Selection of which experiments to run, "runs/all.properties".

```
# Properties file for selecting a subset of the full benchmark suite to run
# all.properties: selects all graphs and algorithms

# Name of the benchmark run for the results report
benchmark.run.name = Full Benchmark

# Select a subset of the graphs (leave blank to select all)
benchmark.run.graphs =

# Select a subset of the algorithms (leave blank to select all)
```

```
benchmark.run.algorithms =

# Specify if the output should be stored and where
benchmark.run.output-required = true
benchmark.run.output-directory = ./output/

# Specify if the output should be validated and where the reference output
is stored.
benchmark.run.validation-required = false
benchmark.run.validation-directory = ./validate/
```

Listing 3.3: Selection of graph datasets, “graphs.properties”.

```
# Properties file describing the list of available graphs

# Exhaustive list of available graphs to enable property discovery
graphs.names = ldbc-1, ldbc-3, ldbc-10, ldbc-30, ldbc-100, ldbc-300, ldbc-1000,\
        amazon0302, wiki-Talk, cit-Patents, com-friendster, kgs, twitter_mpi, \
        graph500-22, graph500-23, graph500-24, graph500-25, graph500-26, \
        dota-league, \
        datagen-100, datagen-100-fb-cc0_05, datagen-100-fb-cc0_15, datagen-300, \
        datagen-1000, \
        example-directed, example-undirected

# Root directory containing graphs on local filesystem
graphs.root-directory = /data/graphalytics/graphs

# Directory to cache derived datasets in (optional, defaults to
# ${graphs.root-directory}/cache).
# During execution, Graphalytics will create additional graphs from
# source datasets with subsets of properties,
# e.g. given a graph with edge weights, Graphalytics will write an edge list
# to disk without weights to run BFS on.
# Cached datasets are reused by Graphalytics between invocations to speed up
# the benchmarking process.
# graphs.cache-directory = /data/graphalytics/graphs/cache

# Include the properties files describing each individual graph
include = graphs/ldbc-1.properties
include = graphs/ldbc-3.properties
include = graphs/ldbc-10.properties
include = graphs/ldbc-30.properties
include = graphs/ldbc-100.properties
include = graphs/ldbc-300.properties
include = graphs/ldbc-1000.properties
include = graphs/amazon0302.properties
include = graphs/wiki-Talk.properties
include = graphs/cit-Patents.properties
include = graphs/com-friendster.properties
include = graphs/dota-league.properties
include = graphs/kgs.properties
include = graphs/twitter_mpi.properties
include = graphs/datagen-100.properties
include = graphs/datagen-100-fb-cc0_05.properties
include = graphs/datagen-100-fb-cc0_15.properties
include = graphs/datagen-300.properties
include = graphs/datagen-1000.properties
include = graphs/example-directed.properties
```

```
include = graphs/example-undirected.properties
include = graphs/graph500-22.properties
include = graphs/graph500-23.properties
include = graphs/graph500-24.properties
include = graphs/graph500-25.properties
include = graphs/graph500-26.properties
```

Listing 3.4: Detailed configuration for one datasets, “graphs/ldbc-1.properties”.

```
# Properties file describing the LDBC-1 dataset

# Filenames of graph on local filesystem
graph.ldbc-1.vertex-file = ldbc-1.v
graph.ldbc-1.edge-file = ldbc-1.e

# Graph metadata for reporting purposes
graph.ldbc-1.meta.vertices = 10999
graph.ldbc-1.meta.edges = 452622

# Properties describing the graph format
graph.ldbc-1.directed = true

# List of supported algorithms on the graph
graph.ldbc-1.algorithms = bfs, cdlp, lcc, pr, wcc


#
# Per-algorithm properties describing the input parameters to each algorithm
#

# Parameters for BFS
graph.ldbc-1.bfs.source-vertex = 12094627913375

# Parameters for CDLP
graph.ldbc-1.cdlp.max-iterations = 10

# No parameters for LCC

# Parameters for PR
graph.ldbc-1.pr.damping-factor = 0.85
graph.ldbc-1.pr.num-iterations = 10

# No parameters for WCC
```

# 4 Auditing Rules

The auditing rules are currently under discussion in the LDBC consortium. We welcome feeback from all stakeholders, and also from both industry and academia organizations.

# 5 Implementation Instructions

## 5.1 Software Installation

A complete guide on how to install LDBC Graphalytics is available at:
`https://github.com/tudelft-atlarge/graphalytics`

## 5.2 Platform Driver Specification

Instructions for creating a new platform driver for Graphalytics are available on the Graphalytics wiki at:
`https://github.com/tudelft-atlarge/graphalytics/wiki/`

Examples of platform drivers already built into LDBC Graphalytics are available at:

- Giraph (no vendor optimization):
  `https://github.com/tudelft-atlarge/graphalytics-platforms-giraph`
- GraphLab (no vendor optimization):
  `https://github.com/tudelft-atlarge/graphalytics-platforms-graphlab`
- Graphmat:
  `https://github.com/tudelft-atlarge/graphalytics-platforms-graphmat`
- GraphX (no vendor optimization):
  `https://github.com/tudelft-atlarge/graphalytics-platforms-graphx`
- Hadoop/YARN (no vendor optimization):
  `https://github.com/tudelft-atlarge/graphalytics-platforms-mapreducev2`
- OpenG:
  `https://github.com/tudelft-atlarge/graphalytics-platforms-openg`
- PowerGraph (no vendor optimization):
  `https://github.com/tudelft-atlarge/graphalytics-platforms-powergraph`

## 5.3 Benchmark Configuration

A complete guide on how to configure LDBC Graphalytics is available at:
`https://github.com/tudelft-atlarge/graphalytics`

## 5.4 Benchmark Operation

A guide on how to run LDBC Graphalytics is available at:
`https://github.com/tudelft-atlarge/graphalytics`

## 5.5 Results Gathering, Analysis, and Visualization

An overview of the Granula component of LDBC Graphalytics, which is responsible for results gathering, analysis, and visualization, is available at:
`https://github.com/tudelft-atlarge/granula`

## REFERENCES


[1] Apache Giraph. `http://giraph.apache.org`.

[2] SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[3] Günes Aluç et al. Diversified stress testing of RDF data management systems. In *ISWC*, pages 197–212, 2014.

[4] Khaled Ammar and M. Tamer Özsu. WGB: towards a universal graph benchmark. In *WBDB*, pages 58–72, 2013.

[5] Timothy Armstrong et al. LinkBench: a database benchmark based on the Facebook social graph. In *SIGMOD*, pages 1185–1196, 2013.

[6] David Bader and Kamesh Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *HiPC*, pages 465–476, 2005.

[7] Christian Bizer and Andreas Schultz. The Berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

[8] Mihai Capota et al. Graphalytics: A big data benchmark for graph-processing platforms. In *GRADES*, pages 7:1–7:6, 2015.

[9] Meeyoung Cha et al. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM*, page 30, 2010.

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[11] Miyuru Dayarathna and Toyotaro Suzumura. Graph database benchmarking on cloud environments with XGDBench. *Autom. Softw. Eng.*, 21(4):509–533, 2014.

[12] Assaf Eisenman et al. Parallel graph processing: Prejudice and state of the art. In *ICPE*, 2016.

[13] Benedikt Elser and Alberto Montresor. An evaluation study of bigdata frameworks for graph processing. In *Big Data*, pages 60–67, 2013.

[14] Orri Erling et al. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[15] Jing Fan et al. The case against specialized graph analytics engines. In *CIDR*, 2015.

[16] Michael Ferdman et al. Clearing the clouds: a study of emerging scaleout workloads on modern hardware. In *ASPLOS*, pages 37–48, 2012.

[17] Ahmad Ghazal et al. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*, pages 1197–1208, 2013.

[18] Joseph E Gonzalez et al. PowerGraph: Distributed graph parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[19] Yong Guo and Alexandru Iosup. The game trace archive. In *NETGAMES*, page 4. IEEE Press, 2012.

[20] Yong Guo et al. How well do graph-processing platforms perform? In *IPDPS*, pages 395–404, 2014.

[21] Yong Guo et al. An empirical performance evaluation of gpu-enabled graph-processing systems. In *CCGrid*, pages 423–432, 2015.

[22] Yuanbo Guo et al. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[23] Minyang Han et al. An experimental comparison of pregel-like graph processing systems. *PVLDB*, 7(12):1047–1058, 2014.

[24] Sungpack Hong et al. PGX.D: a fast distributed graph processing engine. In *SC*, pages 58:1–58:12, 2015.

[25] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *PVLDB*, 9(13):1–12, 2016.

[26] Alekh Jindal et al. Vertexica: your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014.

[27] Yi Lu et al. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[28] Zijian Ming et al. BDGS: A scalable big data generator suite in big data benchmarking. In *WBDB*, pages 138–154, 2013.

[29] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray UserâĂŹs Group (CUG)*, 2010.

[30] Lifeng Nai et al. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC*, pages 69:1–69:12, 2015.

[31] Lawrence Page et al. The pagerank citation ranking: bringing order to the web. 1999.

[32] Tilmann Rabl et al. The vision of BigBench 2.0. In *DanaC*, pages 3:1–3:4, 2015.

[33] Usha Raghavan et al. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[34] Nadathur Satish et al. Navigating the maze of graph analytics frameworks using massive datasets. In *SIGMOD*, pages 979–990, 2014.

[35] Michael Schmidt et al. Spˆ2 bench: a SPARQL performance benchmark. In *ICDE*, pages 222–233, 2009.

[36] Narayanan Sundaram et al. Graphmat: High performance graph analytics made productive. *PVLDB*, 8(11):1214–1225, 2015.

[37] Lei Wang et al. BigDataBench: a big data benchmark suite from internet services. In *HPCA*, pages 488–499, 2014.

[38] Reynold Xin et al. GraphX: A resilient distr. graph system on Spark. In *GRADES*, page 2, 2013.

# A Pseudo-code for Algorithms

This chapter contains pseudo-code for the algorithms described in Section 2.3. In the following sections, a graph $G$ consists of a set of vertices $V$ and a set of edges $E$. For undirected graphs, each edge is bidirectional, so if $(u, v) \in E$ then $(v, u) \in E$. Each vertex has a set of outgoing neighbors $N_{out}(v) = \{u\ inV | (v, u)\ inE\}$ and a set of incoming neighbors $N_{in}(v) = \{u\ inV | (u, v)\ inE\}$.

## A.1 Breadth-First Search (BFS)

**input:** graph $G = (V, E)$, vertex $root$
**output:** array $depth$ storing vertex depths

**for all** $v \in V$ **do**
    $depth[v] \leftarrow \infty$
**end for**
Q $\leftarrow$ CREATE_QUEUE()
Q.PUSH($root$)
$depth[root] \leftarrow 0$
**while** Q.SIZE > 0 **do**
    $v \leftarrow$ Q.POP-FRONT( )
    **for all** $u \in N_{out}(v)$ **do**
        **if** $depth[u] = \infty$ **then**
            $depth[u] \leftarrow depth[v] + 1$
            Q.PUSH-BACK($u$)
        **end if**
    **end for**
**end while**

## A.2 PageRank (PR)

**input:** graph $G = (V, E)$, integer $max\_iterations$
**output:** array $rank$ storing PageRank values

**for all** $v \in V$ **do**
    $rank[v] \leftarrow \frac{1}{|V|}$
**end for**
**for** $i = 1, \ldots, max\_iterations$ **do**
    $dangling\_sum \leftarrow 0$
    **for all** $v \in V$ **do**
        **if** $|N_{out}(v)| = 0$ **then**
            $dangling\_sum \leftarrow dangling\_sum + rank[v]$
        **end if**
    **end for**
    **for all** $v \in V$ **do**
        $new\_rank[v] \leftarrow (1 - d)\frac{1}{|V|} + d\Big(\sum_{u \in N_{in}(v)} \frac{rank[u]}{|N_{out}(u)|} + \frac{dangling\_sum}{|V|}\Big)$
    **end for**
    $rank \leftarrow new\_rank$
**end for**

## A.3 Weakly Connected Components (WCC)

**input:** graph $G = (V, E)$
**output:** array $comp$ storing component labels

**for all** $v \in V$ **do**
    comp$[v] \leftarrow v$
**end for**
**repeat**
    $converged \leftarrow$ true
    **for all** $v \in V$ **do**
        **for all** $u \in N_{in}(v) \cup N_{out}(v)$ **do**
            **if** comp$[v] >$ comp$[u]$ **then**
                comp$[v] \leftarrow$ comp$[u]$
                $converged \leftarrow$ false
            **end if**
        **end for**
    **end for**
**until** $converged$

## A.4 Local Clustering Coefficient (LCC)

**input:** graph $G = (V, E)$
**output:** array $lcc$ storing LCC values

**for all** $v \in V$ **do**
    $d \leftarrow |N_{in}(v) \cup N_{out}(v)|$
    **if** $d \geq 2$ **then**
        $t \leftarrow 0$
        **for all** $u \in N_{in}(v) \cup N_{out}(v)$ **do**
            **for all** $w \in N_{in}(v) \cup N_{out}(v)$ **do**
                **if** $(u, w) \in E$ **then** ▷ Check if edge $(u, w)$ exists
                    $t \leftarrow t + 1$ ▷ Found triangle $v - u - w$
                **end if**
            **end for**
        **end for**
        lcc$[v] \leftarrow \frac{t}{d(d-1)}$
    **else**
        lcc$[v] \leftarrow 0$ ▷ No triangles possible
    **end if**
**end for**

## A.5 Community Detection using Label-Propagation (CDLP)

**input:** graph $G = (V, E)$, integer $max\_iterations$
**output:** array $labels$ storing vertex communities

**for all** $v \in V$ **do**
  $labels[v] \leftarrow v$
**end for**
**for** $i = 1, \ldots, max\_iterations$ **do**
  **for all** $v \in V$ **do**
    C $\leftarrow$ CREATE_HISTOGRAM()
    **for all** $u \in N_{in}(v)$ **do**
      C.ADD($(labels[u])$)
    **end for**
    **for all** $u \in N_{out}(v)$ **do**
      C.ADD($(labels[u])$)
    **end for**
    $freq \leftarrow$ C.GET_MAXIMUM_FREQUENCY( ) ▷ Find maximum frequency of labels.
    $candidates \leftarrow$ C.GET_LABELS_FOR_FREQUENCY($freq$) ▷ Find labels with max. frequency.
    $new\_labels[v] \leftarrow$ MIN($candidates$) ▷ Select smallest label
  **end for**
  $labels \leftarrow new\_labels$
**end for**

## A.6 Single-Source Shortest Paths (SSSP)

**input:** graph $G = (V, E)$, vertex $root$, edge weights $weight$.
**output:** array $dist$ storing distances

**for all** $v \in V$ **do**
  $dist[v] \leftarrow \infty$
**end for**
H $\leftarrow$ CREATE_HEAP()
H.INSERT(root, 0)
$dist[root] \leftarrow 0$
**while** H.SIZE $> 0$ **do**
  $v \leftarrow$ H.DELETE_MINIMUM( ) ▷ Find vertex $v$ in H such that $dist[v]$ is minimal.
  **for all** $w \in N_{out}(v)$ **do**
    **if** $dist[w] > dist[v] + weight[v, w]$ **then**
      $dist[w] \leftarrow dist[v] + weight[v, w]$
      H.INSERT($w$, $dist[w]$)
    **end if**
  **end for**
**end while**

Table B.1: Overview of related work. (Acronyms: *Reference type*: **S**, study, **B**, benchmark. *Target system, structure*: **D**, distributed system; **P**, parallel system; **MC**, single-node multi-core system; **GPU**, using GPUs. *Input*: **0**, no parameters; **S**, parameters define scale; **E**, parameters define edge properties; **+**, parameters define other graph properties, e.g., clustering coefficient. *Datasets/Algorithms*: **Rnd**, reason for selection not explained; **Exp**, selection guided by expertise; **1-stage**, data-driven selection; **2-stage**, 2-stage data- and expertise-driven process. *Scalability tests*: **W**, weak, **S**, strong, **V**, vertical, **H**, horizontal.)

| Reference (chronological order) | | Target System (R1) | | Design (R2) | | | | Tests (R3) | | (R4) |
|---|---|---|---|---|---|---|---|---|---|---|
| | Name [Publication] | Structure | Programming | Input | Datasets | Algo. | Scalable? | Scalability | Robustness | Renewal |
| B | CloudSuite [16], only graph elements | D/MC | PowerGraph | S | Rnd | Exp | — | No | No | No |
| S | Montresor et al. [13] | D/MC | 3 classes | 0 | Rnd | Exp | — | No | No | No |
| B | HPC-SGAB [6] | P | — | S | Exp | Exp | — | No | No | No |
| B | Graph500 | P/MC/GPU | — | S | Exp | Exp | — | No | No | No |
| B | GreenGraph500 | P/MC/GPU | — | S | Exp | Exp | — | No | No | No |
| B | WGB [4] | D | — | SE+ | Exp | Exp | 1B Edges | No | No | No |
| S | **Own prior work** [20, 21, 8] | **D/MC/GPU** | **10 classes** | **S** | **Exp** | **1-stage** | **1B Edges** | **W/S/V/H** | **No** | **No** |
| S | Özsu et al. [23] | D | Pregel | 0 | Exp,Rnd | Exp | — | W/S/V/H | No | No |
| B | BigDataBench [28, 37], only graph elements | D/MC | Hadoop | S | Rnd | Rnd | — | S | No | No |
| S | Satish et al. [34] | D/MC | 6 classes | S | Exp,Rnd | Exp | — | W | No | No |
| S | Lu et al. [27] | D | 4 classes | S | Exp,Rnd | Exp | — | S | No | No |
| B | GraphBIG [30] | P/MC/GPU | System G | S | Exp | Exp | — | No | No | No |
| S | Cherkasova et al. [12] | MC | Galois | 0 | Rnd | Exp | — | No | No | No |
| **B** | **LDBC Graphalytics** (this work) | **D/MC/GPU** | **10+ classes** | **SE+** | **2-stage** | **2-stage** | **Process** | **W/S/V/H** | **Yes** | **Yes** |

Table B.1, which is reproduced from [25], summarizes and compares Graphalytics with previous studies and benchmarks for graph analysis systems. R1–R5 are the requirements formulated in Section 2.1. As Table B.1 indicates, there is no alternative to Graphalytics in covering requirements R1–R4. We also could not find evidence of requirement R5 being covered by other systems than LDBC. While there have been a few related benchmark proposals (marked "B"), these either do not *focus* on graph analysis, or are much narrower in scope (e.g., only BFS for Graph500). There have been comparable studies (marked "S") but these have not attempted to define—let alone maintain—a benchmark, its specification, software, testing tools and practices, or results. Graphalytics is not only industry-backed but also has industrial strength, through its detailed execution process, its metrics that characterize robustness in addition to scalability, and a renewal process that promises longevity. Graphalytics is being proposed to SPEC as well, and BigBench [17, 32] explicitly refers to Graphalytics as its option for future benchmarking of graph analysis platforms.

Previous studies typically tested the open-source platforms Giraph [1], GraphX [38], and PowerGraph [18], but our contribution here is that vendors (Oracle, Intel, IBM) in our evaluation have themselves tuned and tested their implementations for PGX [24], GraphMat [36] and OpenG [30]. We are aware that the database community has started to realize that with some enhancements, RDBMS technology could also be a contender in this area [15, 26], and we hope that such systems will soon get tested with Graphalytics.

Graphalytics complements the many existing efforts focusing on graph databases, such as LinkedBench [5], XGDBench [11], and LDBC SNB [14]; efforts focusing on RDF graph processing, such as LUBM [22], the Berlin SPARQL Benchmark [7], SP$^2$Bench [35], and WatDiv [3] (targeting also graph databases); and community efforts such as the TPC benchmarks. Whereas all these prior efforts are interactive database query benchmarks, Graphalytics focuses on algorithmic graph analysis and on different platforms which are not necessarily database systems, whose distributed and highly parallel aspects lead to different design trade-offs.