



LDBC Graphalytics Benchmark v0.2.12(2) - Draft Release

Coordinator: Alexandru Iosup (Vrije Universiteit Amsterdam (VU) and Delft University of Technology (TUD))

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

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 conduct 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 for 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

EXECUTIVE SUMMARY	3
DOCUMENT INFORMATION	4
1 INTRODUCTION	8
1.1 Motivation for the Benchmark	8
1.2 Relevance to Industry and Academia	8
1.3 General Benchmark Overview	8
1.4 Participation of Industry and Academia	9
2 FORMAL DEFINITION	10
2.1 Requirements	10
2.2 Data	10
2.2.1 Definition	11
2.2.2 Representation	11
2.2.3 Size and Scale	11
2.2.4 Datasets	12
2.3 Algorithms	13
2.3.1 Breadth-First Search (BFS)	14
2.3.2 PageRank (PR)	14
2.3.3 Weakly Connected Components (WCC)	14
2.3.4 Community Detection using Label-Propagation (CDLP)	15
2.3.5 Local Clustering Coefficient (LCC)	15
2.3.6 Single-Source Shortest Paths (SSSP)	15
2.4 Output Validation	16
2.5 Job	17
2.5.1 Description	17
2.5.2 Operations	17
2.5.3 Metrics	18
2.6 System-under-Test	19
2.7 Renewal Process	20
3 BENCHMARK PROCESS	21
3.1 Benchmark	21
3.2 Benchmark Type	21
3.2.1 Test Benchmark	21
3.2.2 Standard Benchmark	22
3.2.3 Full Benchmark	23
3.2.4 Custom Benchmark	24
3.3 Benchmark Execution	24
3.3.1 Execution Flow	24
3.3.2 Run Flow	25
3.3.3 Data Flow	26
3.3.4 Failure Indication	27
3.4 Benchmark Result	27
4 GRAPHALYTICS-BASED COMPETITIONS	29
4.1 Official Graphalytics Competitions	29
4.1.1 The Global LDBC Competition	29

4.1.2	The Global Graphalytics Competition	29
4.2	Competition Method	30
4.2.1	Single value-of-merit Approach	30
4.2.2	Tournament-based approach	31
5	AUDITING RULES	32
6	IMPLEMENTATION INSTRUCTIONS	33
6.1	Graphalytics Softwares and Documentation	33
6.2	Graphalytics-related Softwares	33
6.3	Graphalytics Platform Driver	34
A	PSEUDO-CODE FOR ALGORITHMS	35
A.1	Breadth-First Search (BFS)	35
A.2	PageRank (PR)	35
A.3	Weakly Connected Components (WCC)	36
A.4	Local Clustering Coefficient (LCC)	36
A.5	Community Detection using Label-Propagation (CDLP)	37
A.6	Single-Source Shortest Paths (SSSP)	37
B	DATA FORMAT FOR BENCHMARK RESULTS	38
C	RELATED WORK	40

LIST OF FIGURES

2.1	Example of directed weighted graph in EVLP format.	11
2.2	Example of validation with <i>exact match</i>	16
2.3	Example of validation with <i>equivalence match</i>	16
2.4	Example of validation with <i>epsilon match</i>	17
2.5	A typical graph processing job with the underlying operations.	18
3.1	The composition of a benchmark.	21
3.2	Benchmark execution in the Graphalytics benchmark suite.	25
B.1	Result Format: Overview	38
B.2	Result Format: System under Test	38
B.3	Result Format: Benchmark Configuration	39
B.4	Result Format: Experiment Result	39

LIST OF TABLES

2.1	Mapping of dataset scales (“T-shirt sizes”) in Graphalytics.	12
2.2	Real-world datasets used by Graphalytics.	12
2.3	Synthetic Graph500 datasets used by Graphalytics.	13
2.4	Synthetic Datagen datasets used by Graphalytics.	13
3.1	The benchmark composition of the Test benchmark.	22
3.2	The benchmark composition of the Standard benchmark on target-scale S.	22
3.3	The benchmark composition of the Standard benchmark on target-scale M.	23
3.4	The benchmark composition of the Standard benchmark on target-scale L.	23
3.5	The benchmark composition of the Standard benchmark on target-scale XL.	23
C.1	Overview of related work	40

1 INTRODUCTION

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 [?].

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 C 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 provide 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 requirements of (1) targeting platforms and systems, (2) incorporating 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., S, M, L, XL). The benchmarking process is made future-proof, through a *renewal process*.

Each system under test undergoes a standard benchmark (See Chapter 3) per target-scale, which executes in total 90 graph-processing jobs (six core algorithms, five different datasets, and three repetitions per job). **For full benchmark, additional experiments on scalability and robustness will also be included. 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. (Future in v.2.0)**

1.4 Participation of Industry and Academia

The Linked Data Benchmark Council (ldbouncil.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 February 2017, the list of participants is as follows:

- CENTRUM WISKUNDE & INFORMATICA, the Netherlands (CWI)
- DELFT UNIVERSITY OF TECHNOLOGY, the Netherlands (TUD)
- [VRIJE UNIVERSITEIT AMSTERDAM, the Netherlands \(VU\)](#) (New in v.0.2.12)
- GEORGIA INSTITUTE OF TECHNOLOGY, USA (GT)
- HUAWEI RESEARCH AMERICA, USA (HAUWEI)
- INTEL LABS, USA (INTEL)
- ORACLE LABS, USA (ORACLE)
- POLYTECHNIC UNIVERSITY OF CATALONIA, Spain (UPC)

2 FORMAL DEFINITION

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. (Future in v.2.0)** 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.

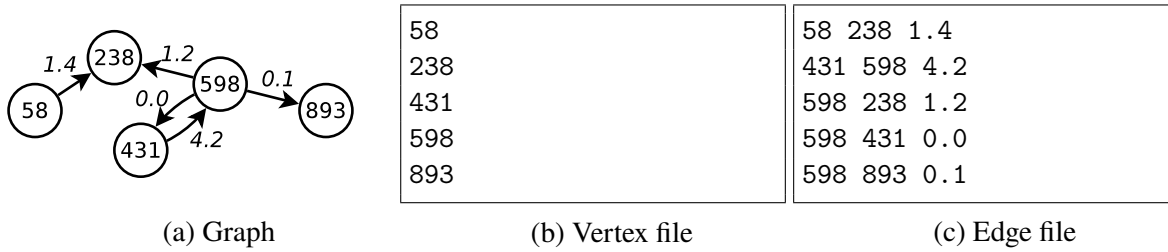


Figure 2.1: Example of directed weighted graph in EVLP format.

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 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, graphalytics 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 Size and Scale

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

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

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:

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

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 Datasets

Graphalytics uses both graphs from real-world applications and synthetic graphs which are generated using data generators, the selection of which spans a variety of sizes and densities.

Real-world Datasets: By including real-world graphs from a variety of domains, Graphalytics covers users from different communities, including graphs from the knowledge, gaming, and social network domains. Table 2.2 lists the real-world datasets used by the Graphalytics benchmark.

Table 2.2: Real-world datasets used by Graphalytics.

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

Renew the graph selection process for real-world graphs, using new graph data archive, for example, with suggestion from Mihai.... (Future in v.2.0)

Synthetic Datasets: Besides real-world datasets, Graphalytics adopts two commonly used generators that generate two types of graphs: power-law graphs from the **Graph500** generator [?, ?] and social network graphs from **LDBC Datagen** [?]. Table 2.3 and 2.4 lists the graph500 datasets and the Datagen datasets.

Table 2.3: Synthetic Graph500 datasets used by Graphalytics.

ID	Name	n	m	Scale
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

Table 2.4: Synthetic Datagen datasets used by Graphalytics.

ID	Name	n	m	Scale
D7.5(S)	Datagen-7.5-fb	0.6M	34.2M	7.5
D7.6(S)	Datagen-7.6-fb	0.8M	42.2M	7.6
D7.7(S)	Datagen-7.7-zf	13.2M	32.8M	7.6
D7.8(S)	Datagen-7.8-zf	16.5M	41.0M	7.7
D7.9(S)	Datagen-7.9-fb	1.4M	85.7M	7.9
D8.0(M)	Datagen-8.0-fb	1.7M	1.1B	8.0
D8.1(M)	Datagen-8.1-fb	2.1M	1.3B	8.1
D8.2(M)	Datagen-8.2-zf	43.7M	1.1B	8.1
D8.3(M)	Datagen-8.3-zf	53.5M	1.3B	8.2
D8.4(M)	Datagen-8.4-fb	3.8M	2.7B	8.4
D8.5(L)	Datagen-8.5-fb	4.6M	3.3B	8.5
D8.6(L)	Datagen-8.6-fb	5.7M	4.2B	8.6
D8.7(L)	Datagen-8.7-zf	1.5B	3.4B	8.6
D8.8(L)	Datagen-8.8-zf	1.7B	4.1B	8.7
D8.9(L)	Datagen-8.9-fb	10.6M	8.5B	8.9
D9.0(XL)	Datagen-9.0-fb	12.9M	10.5B	9.0
D9.1(XL)	Datagen-9.1-fb	16.1M	13.4B	9.1
D9.2(XL)	Datagen-9.2-zf	4.3B	10.4B	9.1
D9.3(XL)	Datagen-9.3-zf	5.6B	13.1B	9.2
D9.4(XL)	Datagen-9.4-fb	29.3M	25.9B	9.4

2.3 Algorithms

The Graphalytics benchmark consists of six algorithms which need to be executed on the different 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, pseudo-code is given in Appendix A. Furthermore, a link to the reference implementation is presented in Section 6.3. 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 $N_{out} = \{u \in V | (v, u) \in E\}$, a set of incoming neighbors $N_{in} = \{u \in V | (u, v) \in E\}$, and a set of all neighbors $N(v) = N_{in}(v) \cup N_{out}(v)$. 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 [?]. 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.

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

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:

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

Where $d \in [0, 1]$ is called the *damping factor* and $D = \{v \in V | |N_{out}(v)| = 0\}$ 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* algorithm 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, i.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. [?]. 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.

$$L_0(v) = v \quad (2.4)$$

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:

$$L_i(v) = \arg \max_l \left[|\{u \in N_{in}(v) \mid L_{i-1}(u) = l\}| + |\{u \in N_{out}(v) \mid L_{i-1}(u) = l\}| \right] \quad (2.5)$$

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:

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

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

$$LCC(v) = \frac{\sum_{u \in N(v)} |N(v) \cap N_{in}(u)|}{|N(v)|(|N(v)|-1)} \quad (2.7)$$

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 Output 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. The reference output are typically generated by a specifically chosen reference platform, the implementation of which is cross-validated with at least two other platforms up till target-scale L. At the moment, the chosen reference platform for Graphalytics is OpenG (See Section 6.3).

The validation output presents numbers either as integers or floating-point numbers, depending on the algorithm definition. Note that these numbers are stored in the file system as decimal values in plain text (ASCII). For integers, any number between $-2^{63} - 1$ and $2^{63} - 2$ represents its actual value, whereas the numbers -9223372036854775808 (-2^{63}) and 9223372036854775807 ($2^{63} - 1$) represents negative and positive infinity. For floating-point numbers, a scientific notation with 15 significant digits (e.g., 2.476533217845853e-08) is presented, whereas negative and positive infinity are represented as by the string “*infinity*” and “*-infinity*”.

The system’s output generated during the benchmark must be stored as decimal values in plain text (ASCII) file(s), grouped into a single file directory. The formatting rules for integers must be exactly the same as the validation output. However, floating-point numbers require less than 15 significant digits, also that positive infinity can be represented by either “*inf*”, “*+inf*”, “*infinity*”, or “*+infinity*”, and negative infinity can be represented by either “*-inf*” or “*-infinity*”.

There are three methods used for validation:

1 3 2 1 3 2 4 0 5 1	1 3 2 1 3 2 4 0 5 1	1 4 2 1 3 2 4 5 5 1
(a) Reference output	(b) Example of correct result	(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	1 81 2 81 3 81 4 32 5 32 6 12	1 31 2 52 3 31 4 31 5 31 6 74
(a) Reference output	(b) Example of correct result	(c) Example of incorrect result

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

- **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.

1 0	1 0	1 0.000001
2 0.3	2 0.30002	2 0.3
3 0.45	3 0.45	3 0.46
4 0.23	4 0.229997	4 0.22
5 +inf	5 +inf	5 1.79769e+308
6 0.001	6 0.001	6 0

(a) Reference output (b) Example of correct result (c) Example of incorrect result

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

- **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 s is within 0.01% of r , i.e., the equation $|r - s| < \epsilon|r|$ must hold where $\epsilon = 0.0001$. The value of ϵ was chosen such that minor errors that result from rounding are not penalized. Figure 2.4 shows an example of validation with epsilon match.

A future validation component of Graphalytics will focus on performance. *Why?* The current approach is correct, but slow. Validating XL graphs is already too time consuming and memory-intensive. *How?* (Future in v.2.0)

2.5 Job

A graph-processing job is the process of executing a graph algorithm (see Section 2.3) on a graph dataset (see Section 2.2.4). This section discusses the description of a graph-processing job, the underlying operations constituting a graph processing job, and the metrics used to measure the job performance.

2.5.1 Description

Each graph-processing job is specified by a list of descriptive information, e.g., the system description, the algorithm and the dataset, and the benchmark configuration. The job description should be uniformly applicable to any graph-processing system.

- **System:** platform type, environment type, and system cost.
- **Algorithm:** algorithm type, and algorithm parameters.
- **Dataset:** vertex size, edge size, and graph size (vertex size + edge size).
- **Benchmark:** deployment mode, allocated resources, and time-out duration.

2.5.2 Operations

Graph processing is data-intensive, sensitive to the data irregularity, and often involves iterative processing. Typically, a graph-processing system facilitates a **Loading** phase to pre-process the data, and follows with one or more **Running** phases to run various graph algorithms on the pre-processed data.

During the **Loading** step, the input graph data can be converted into an optimized system-specific data format, and be pre-loaded into a local/share/distributed storage system. The pre-processed data must be uniformly usable for any graph algorithms.

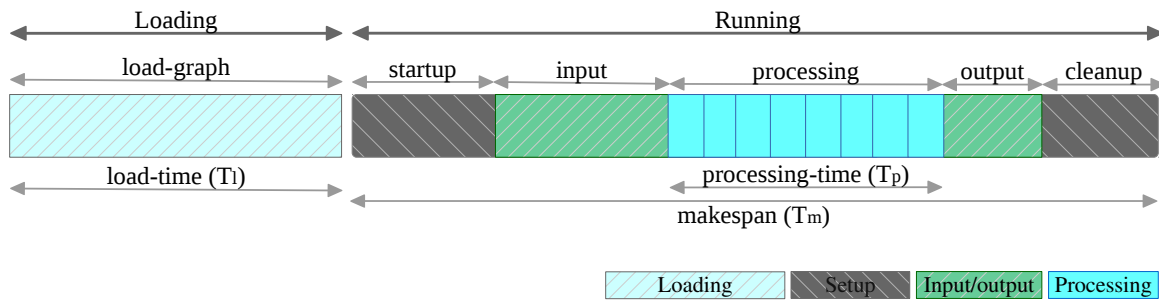


Figure 2.5: A typical graph processing job with the underlying operations.

During the **Running** step, the graph-processing system carries out a series of operations to facilitate efficient algorithm execution on the pre-processed data. These operations are categorized into three types: setup, input/output, and processing operations (See Figure 2.5).

- **Setup operations** reserve computational resources in distributed environments, prepare the system for operation, clean up the environment after the termination of the job.
- **Input/output operations** transfer graph data from storage to the memory space, and convert the data to specific formats before/after data processing, and offload the outputs back to the storage. Some platforms load/unload data from a distributed file system, other distributed platforms from share/local storage in each node.
- **Processing operations** take in-memory data and process it according to an user-defined algorithm and its expression in a programming paradigm. Processing operations typically include iterative “processing” steps.

2.5.3 Metrics

This section describes the metrics used in Graphalytics. 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. Graphalytics reports performance, rate metrics, cost metrics, and ratio metrics, as follows.

The **Performance metrics** report the execution time of various platform operations.

- **Load time (T_l)**, in *seconds*: 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: (T_m)**, in *seconds*: 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. The makespan metric corresponds to the operation of a *cold graph-processing system*, which corresponds to the situation where the system is boot up, processes a single dataset using a single algorithm, and then is shut down.
- **Processing time (T_p)**, in *seconds*: 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. The processing time metric corresponds to the operation of an in-production, *warmed-up graph-processing system*, where especially loading of the graph from the file system and graph partitioning, both of which are typically done only once and are algorithm-independent, are not considered.

The execution time is capped by the time-out duration configured in each benchmark. Once the time-out is reached, the graph-processing job is terminated, and the time-out duration is reported as the performance metrics instead.

The **Throughput metrics** focus on the processing rate of the system under test. They use a notion of workload intensity, expressed in the graph-specific number of processed edges and vertices:

- **Edges Per Second (EPS)**, in *unit/second*: The ratio between the number of edges processed (edge size) and the processing time (T_p) is used by other benchmarks (Graph500 in Table C.1) to quantify the rate of operation of the system under test. This is fine for edge-dominated algorithms, such as the BFS used in the same benchmarks, but does not explain the performance of vertex-dominated algorithms or of algorithms whose performance is a complex function of the structural properties of the dataset.
- **Edges and Vertices per Second (EVPS)**, in *unit/second*: Graphalytics uses the ratio between the sum of the number of edges and the number of vertices (graph size) processed by the system, and the processing time (T_p). EVPS is closely related to the scale of a graph, as defined by Graphalytics (see Section 2.2.3).

Graphalytics also reports **Cost Metrics**:

- **Three-year Total Cost of Ownership (TCO)**, in *dollar*: reported in compliance with the LDBC rules [?], so *not* computed by Graphalytics. In particular, LDBC currently adapts the TPC standard pricing model v2.0.0 [?].
- **Price-per-performance (PPP)**, in *dollar/unit*: as the ratio between TCO and EVPS. This is a metric included for compliance with the LDBC charter [?].
- **Three-year Total Energy Costs (TEC)**, in *watt*: reported in compliance with the SPEC Power, so *not* computed by Graphalytics and *not* relying only on the (poor) metrics provided by the operating system software and hardware performance counters of current processors/systems. (Future in v.2.0)
- **Energy-per-performance (EPP)**, in *watt/dollar*: as the ratio between TEC and EVPS. This is a metric included for compliance with the LDBC charter [?]. (Future in v.2.0)

The **Ratio metrics** reported by Graphalytics 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 performance achieved by a reference graph-processing platform defined by the Graphalytics team until the next revision, here, the public version of Giraph with the reference drivers, in a reference environment defined by the Graphalytics team until the next revision, here, DAS5 in the Netherlands. The renewal process of Graphalytics (see Section 2.7) can change the baseline. (Future in v.2.0)
- ...

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 platform and hardware environment 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

The Graphalytics benchmark suite is developed to facilitate the benchmark process described in this technical specification. This chapter describes the benchmark composition, the benchmark type, the detailed steps of the benchmark execution, and the format of the benchmark report.

3.1 Benchmark

A benchmark is a standardized process to quantify the performance of the system-under-test. Figure 3.1 depicts the benchmark composition: each benchmark contains a set of benchmark experiments, each experiment consists of multiple benchmark jobs, and each job is executed repeatedly in the form of benchmark runs.

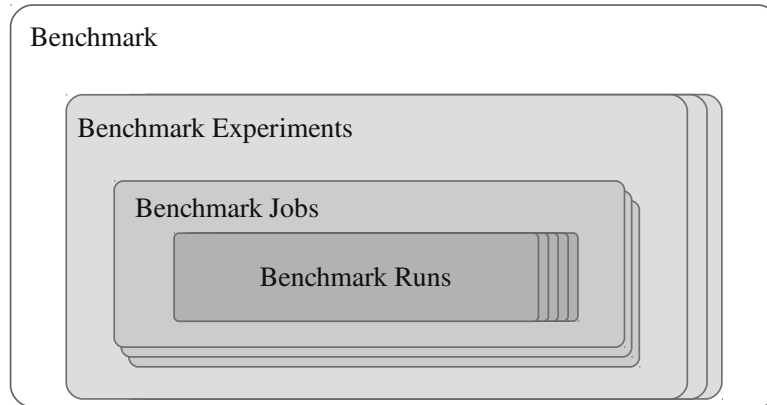


Figure 3.1: The composition of a benchmark.

- A **benchmark experiment** addresses a specific performance characteristic of the system-under-test, e.g., the performance of an algorithm, or the weak scalability of a system. Each experiment gathers benchmark results from multiple benchmark jobs to quantify a specific performance characteristic.
- A **benchmark job** describes, uniformly across all system-under-tests, the exact job specification of a graph-processing job (See Section 2.5). The job specification contains information, e.g., the system-under-test (platform and the environment), the type of algorithm and dataset, and how much resources is used. These information instructs how the system should be configured during the benchmark execution.
- A **benchmark run** is a real-world execution of a benchmark job. To gather statistically reliable benchmark results, each benchmark job is repeated multiple times in the form of a benchmark run to mitigate the performance variability during the benchmark execution.

3.2 Benchmark Type

The Graphalytics benchmark suite supports four types of benchmark: *test*, *standard*, *full*, and *custom*. This section describes the differences and the composition of these four benchmark types.

3.2.1 Test Benchmark

Running the entire benchmark is very time-consuming. To verify that the cluster environment and all underlying software and hardware stacks are configured properly, user can choose to first run the test benchmark

before starting the actual benchmark. In addition, this can be used to debug a platform driver which is under development.

The test benchmark consists of two experiments which run all six core algorithms on two tiny graphs “example-directed” and “example-undirected”, with only one repetition. In total, there are $6 \times 2 \times 1 = 12$ benchmark runs.

Experiments						
directed (<i>d</i>), undirected (<i>u</i>)						
Algorithm	Dataset	Job	Repetition	Runs	Scale	Time-out
6	2	12	1	12	< 0.5	10 min
Job	BFS	WCC	PR	CDLP	LCC	SSSP
XDIR	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>
XUDIR	<i>u</i>	<i>u</i>	<i>u</i>	<i>u</i>	<i>u</i>	<i>u</i>

Table 3.1: The benchmark composition of the Test benchmark.

3.2.2 Standard Benchmark

Participating in the Graphalytics global competition requires benchmark to be executed in a standardized format, which is defined as the standard benchmark. The job composition of the standard benchmark should be representative, covering the variability in algorithm complexity and dataset properties that have several impact on the performance of graph processing systems.

The standard benchmark evaluates the system performance with six core algorithms, BFS, WCC, PR, CDLP, LCC, SSSP (See Section 2.3). For each algorithm, five datasets with various of sizes and densities are selected within a target-scale(See Section 2.2.4). Whenever possible, two real-world datasets, two Datagen datasets, and one Graph500 dataset will be included in the dataset selection. If such selection is not possible, e.g., there is a lack of real-world datasets in certain target-scale, or an algorithm requires datasets to be accompanied with edge properties, then additional Datagen datasets will be used to make sure that at least five datasets are available per target-scale.

A standard benchmark can fall into one of the four target-scales, S, M, L, and XL. Each target-scale focuses on processing graphs within certain range of data size, and therefore a corresponding time-out duration has been imposed.

The standard benchmark (per target-scale) consists of six experiments executing six graph algorithm on five graph graphs, with five repetitions per job. In total, there are $6 \times 5 \times 5 = 150$ benchmark runs.

Experiments						
BFS (<i>bfs</i>), WCC (<i>wcc</i>), PR (<i>pr</i>), CDLP (<i>cdlp</i>), LCC (<i>lcc</i>), SSSP (<i>sssp</i>)						
Algorithm	Dataset	Job	Repetition	Runs	Scale	Time-out
6	6	30	5	150	7.5 – 7.9	15 min
Job	BFS	WCC	PR	CDLP	LCC	SSSP
DOTA	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D7.6						<i>sssp</i>
D7.7	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D7.8	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D7.9	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
G22	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	

Table 3.2: The benchmark composition of the Standard benchmark on target-scale S.

Experiments						
BFS (<i>bfs</i>), WCC (<i>wcc</i>), PR (<i>pr</i>), CDLP (<i>cdlp</i>), LCC (<i>lcc</i>), SSSP (<i>sssp</i>)						
Algorithm	Dataset	Job	Repetition	Runs	Scale	Time-out
6	6	30	5	150	8.0 – 8.4	30 min
Job	BFS	WCC	PR	CDLP	LCC	SSSP
D8.0						<i>sssp</i>
D8.1	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D8.2	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D8.3	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D8.4	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
G24	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	

Table 3.3: The benchmark composition of the Standard benchmark on target-scale M.

Experiments						
BFS (<i>bfs</i>), WCC (<i>wcc</i>), PR (<i>pr</i>), CDLP (<i>cdlp</i>), LCC (<i>lcc</i>), SSSP (<i>sssp</i>)						
Algorithm	Dataset	Job	Repetition	Runs	Scale	Time-out
6	6	30	5	150	8.5 – 8.9	60 min
Job	BFS	WCC	PR	CDLP	LCC	SSSP
D8.5						<i>sssp</i>
D8.6	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D8.7	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D8.8	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D8.9	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
G25	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	

Table 3.4: The benchmark composition of the Standard benchmark on target-scale L.

Experiments						
BFS (<i>bfs</i>), WCC (<i>wcc</i>), PR (<i>pr</i>), CDLP (<i>cdlp</i>), LCC (<i>lcc</i>), SSSP (<i>sssp</i>)						
Algorithm	Dataset	Job	Repetition	Runs	Scale	Time-out
6	8	30	5	150	9.0 – 9.4	120 min
Job	BFS	WCC	PR	CDLP	LCC	SSSP
FSTER	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	
TWIT	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	
D9.0						<i>sssp</i>
D9.1						<i>sssp</i>
D9.2	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
D9.3						<i>sssp</i>
D9.4	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	<i>sssp</i>
G26	<i>bfs</i>	<i>wcc</i>	<i>pr</i>	<i>cdlp</i>	<i>lcc</i>	

Table 3.5: The benchmark composition of the Standard benchmark on target-scale XL.

3.2.3 Full Benchmark

Full benchmark, including new experiment design on quantifying the scalability and robustness of the system-under-test, will be made available in Graphalytics 2.0.

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 standard 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 Dagen 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 dataset 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. (Future in v.2.0)

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 Dagen 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. (Future in v.2.0)

3.2.4 Custom Benchmark

Custom benchmark, which allows users to define the benchmark composition themselves, will be made available in Graphalytics 2.0.

3.3 Benchmark Execution

The benchmark execution of Graphalytics benchmark suite is illustrated in Figure 3.2. This section explains how the benchmark suite executes a benchmark with regard to its execution flow, run flow, data flow, metric collection and failure indication.

3.3.1 Execution Flow

After a benchmark is loaded, the benchmark suite analyzes the exact composition of the benchmark. Each benchmark consists of a number of benchmark runs, which will be grouped by the graph dataset used by that benchmark run. For each graph dataset, the input data of that dataset will be loaded only once, and be reused for all corresponding benchmarks runs, before finally being removed.

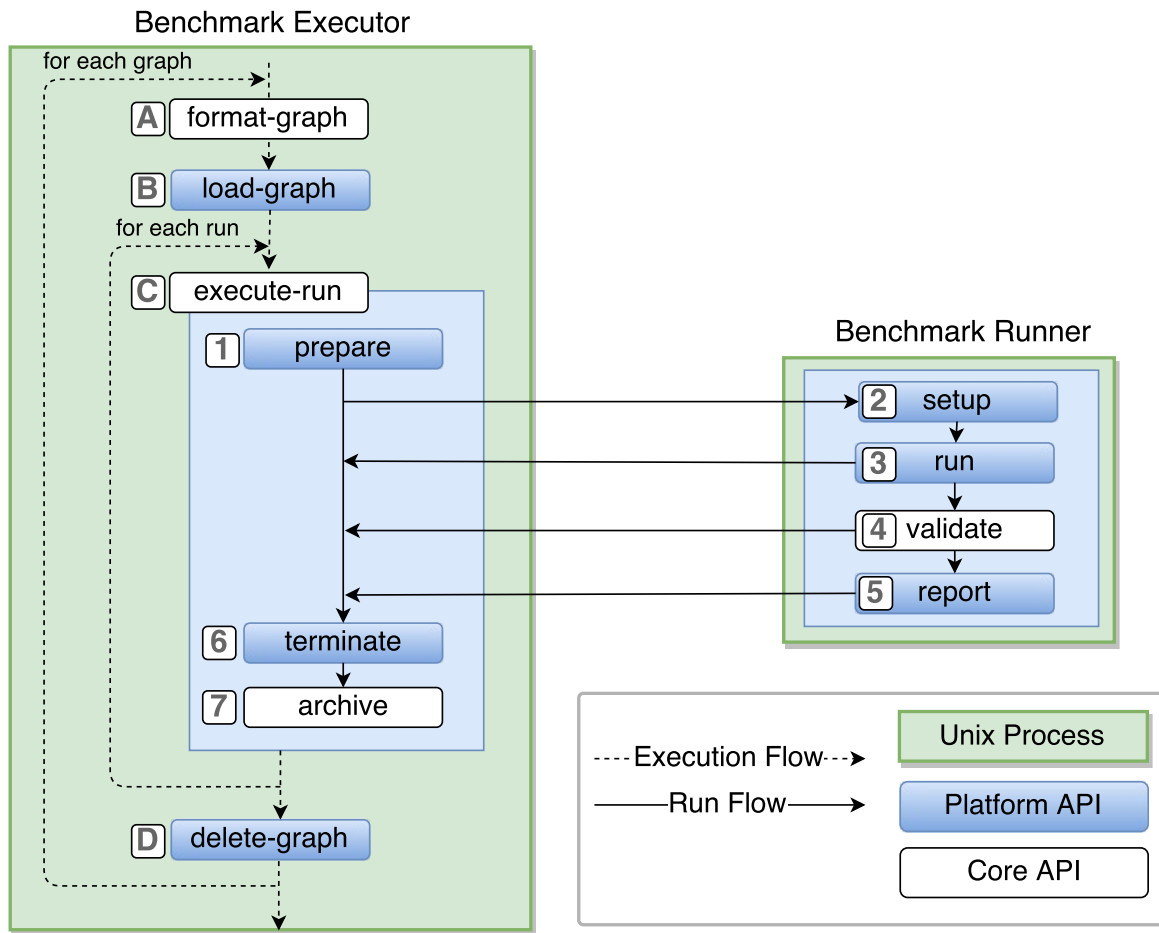


Figure 3.2: Benchmark execution in the Graphalytics benchmark suite.

- **[A] Format-graph:** The benchmark suite minimizes the “input data” into “formatted dataset” (See more in Section 3.3.2) by removing unused vertex and edge properties.
- **[B] Load-graph:** The platform converts the “formatted data” into any platform-specific data format and loads a graph dataset into a storage system, which can be either a local file system, a share file system or a distributed file system. This step corresponds to the “Loading” step of a graph processing job described in 2.5.2.
- **[C] Execute-run:** The platform executes a benchmark run with a specific algorithm and dataset (See more details in Section 3.3.2). All benchmark runs using the same input dataset can use the prepared graph dataset during the “load-graph” step.
- **[D] Delete-graph:** The platform unloads a graph dataset from the storage system, as part of the cleaning up process after all benchmark runs on that graph dataset have been completed.

Note that “load-graph” and “delete-graph” are platform-specific API, which can be implemented in the platform driver via the “Platform” interface, whereas “execute-run” is a uniform step for all platforms.

3.3.2 Run Flow

The execution of each benchmark run consists of seven steps in total, i.e., “prepare”, “setup”, “run”, “cleanup”, “validate”, “terminate”, and “report”. To ensure the stability, the benchmark suite only prepares for the bench-

mark, and terminates the benchmark run. Each benchmark run is partially executed in an isolated operating-system process, such that a timed-out job can be terminated properly.

- **Prepare:** The platform requests computation resources from the cluster environment and makes the background applications ready.
- **Setup:** The platform configures the benchmark run, with regard to real-time cluster deployment information, e.g., input directory, output directory, and log directory.
- **Run:** The platform runs a graph-processing job as defined in the benchmark run. The graph-processing job must complete within the time-out duration, or the benchmark run will fail. This step corresponds to the “Running” step of a graph processing job described in 2.5.2.
- **Report:** The platform gathers the benchmark information, and make the environment ready again for the next benchmark run.
- **Validate:** The benchmark suite validates the platform output with the validation data. The system-under-test must succeed in this step, or the benchmark run will fail.
- **Terminate:** The platform forcibly stops the benchmark job and clean up the environment, given that the time-out has been reached.
- **Archive:** The benchmark suite archives the benchmark results, gathering information regarding performance metrics and failure indications.

Note that “prepare”, “setup”, “run”, “cleanup”, “terminate”, and “report” are platform-specific API, which can be implemented in the platform driver via the “Platform” interface, whereas “archive” is a uniform step for all platforms.

3.3.3 Data Flow

The graph datasets go through a series of execution steps during the benchmark execution, and in the process of which the format, the representation, and the content of the graph dataset change accordingly.

For each graph, the input datasets and the validation datasets are publicly available benchmark resources.

- **Input data:** The “input data” consists of a vertex file and edge file in EVLP format, as defined in Section 2.2.2.
- **Validation data:** The “validation data” consists of correct outputs for all six core algorithm, as defined in Section 2.4.

The input dataset can be converted into the following format during the benchmark.

- **Formatted data:** The “input data” can plausibly contain dozens of vertex and edge properties. During the “load-graph” step, the benchmark suite identifies for each algorithms which properties are needed and which are not, and minimizes the “input data” into the “formatted data”. The “formatted data” is cached in the storage system for future uses.
- **Loaded data:** The minimized “formatted data” is loaded into a storage system during the “load-graph” step, which can either be a local file system, a share file system or a distributed file system.
- **Output data:** The “output data” is the output of a graph-processing job being benchmarked during the “process” step. The “output data” is compared to the “validation data” to ensure the correctness of the benchmark execution.

3.3.4 Failure Indication

Failures can occur during the benchmark for many reasons. The benchmark suite logs the benchmark execution and identifies the type of failures.

- **DAT**: "Data failure" occurs when the "load-graph" step fails to produce "formatted-graph". For example, "Input graph" can be missing or simply be misplaced, or alternatively the conversion from "input-graph" to "formatted-graph" could be prematurely interrupted, which leads to data corruption.
- **INI**: "Initialization failure" occurs when the platform fails to properly make the environment ready for the benchmark during the "prepare" or "setup" step. For example, the deployment system may fail to allocate the cluster resources needed.
- **TIM**: "Time-out failure" occurs when the time-out is reached during the "process" step.
- **EXE**: "Execution failure" occurs when the execution of the benchmark run fails during the "process" step, which leads to non-existing or incomplete output results.
- **VAL**: "Validation failure" occurs when the "output data" is returned by the system, but fails the validation during the "validation step".
- **MET**: "Metric failure" occurs when the compulsory performance metrics are missing during the "archive" step. For example, the log files containing the information can be non-existing or corrupted.

3.4 Benchmark Result

The benchmark result will be updated according to the implementation. (Future in v.1.0) A complete result for the Graphalytics benchmark includes at least the following information:

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 in v.2.0)
7. **If robustness experiments are performed:**
 - (a) **Summary of the results for every benchmark job.**

(Future in v.2.0)

Future versions of the benchmark specification will include a Full Disclosure Report template and a process for submitting official Graphalytics results. A sample data format can be found in Appendix B.

4 GRAPHALYTICS-BASED COMPETITIONS

Add examples for each type of competitions. (Future in v.1.0) .

In this chapter, we describe how to participate in Graphalytics-based competitions. Graphalytics defines several *official competitions* (Section 4.1), which are open globally to everyone who satisfies the rules for participation. Each competition ranks the benchmark submission based on a competition method defined in (Section 4.2).

4.1 Official Graphalytics Competitions

Currently, Graphalytics defines two official competitions: (1) the Global LDBC Competition, and (2) the Global Graphalytics Competition.

4.1.1 The Global LDBC Competition

The Global LDBC Competition is maintained by LDBC, in particular by the Graphalytics Task Forces. By the rules of the LDBC charter [?], the competition method follows the single value-of-merit approach described in Section 4.2.1, and focuses on two primary metrics: “**performance**” and “**cost-performance**”.

The competition reports the following list of metrics:

1. (informative only) Full disclosure of the “system-under-test” (platform + environment).
2. (informative only) *Target-scale* of the benchmark.
3. (informative only) *Date* of the benchmark execution.
4. (flagship value-of-merit) *Performance metric*, as summarized from “EVPS” of all benchmarked jobs.
5. (capability value-of-merit) *Cost-performance metric*, as summarized from “PPP” of all benchmarked jobs.
6. (informative only) Three *performance metrics*, as summarized from T_l , T_m , and T_p respectively.

Maintained by: LDBC, ldbouncil.org.

Audience: The LDBC Competition accepts submissions from a global audience.

Ratio metrics (scalability) and additional energy-related metrics will also be considered. (Future in v.2.0) .

4.1.2 The Global Graphalytics Competition

The Global Graphalytics Competition is maintained by the Graphalyticsteam. The competition method follows the tournament-based approach described in Section 4.2.2, and focuses on two primary scores: “**performance**” and “**cost-performance**”.

The Graphalytics consists of a number of *matches*, where each match represents a type of experiment that focuses on a specific performance characteristic that is common across all systems, for example, the EVPS of the BFS algorithm on a Dagen dataset. Each match consists of a set of instances, with the **tournament score** being for each system the sum of **instance scores** accumulated by the platform across all matches in which it participates. Each *instance* is a head-to-head comparison between two systems, for example, comparing the EVPS of any algorithm-dataset for the pair (Giraph, GraphX): the winner receives 1 point, the loser 0 points, and a draw rewards each platform with 0.5 points each.

1. (informative only) Full disclosure of the “system-under-test” (platform + environment).

2. (informative only) *Target-scale* of the benchmark.
3. (informative only) *Date* of the benchmark execution.
4. (ranking) *Performance score*, by comparing pair-wisely “EVPS” of all benchmarked jobs.
5. (ranking) *Cost-performance score*, by comparing pair-wisely “PPP” of all benchmarked jobs.

Maintained by: Graphalytics, graphalytics.org.

Audience: The Global Graphalytics Competition accepts submissions from a global audience.

Ratio metrics (scalability) and additional energy-related metrics will also be considered. (Future in v.2.0) .

4.2 Competition Method

Different competition methods have been developed to performance comparison in many application domains. Comparing multiple platforms across multiple performance metrics is not trivial. Two major approaches exist for this task: (i) creating a compound metric, typically by weighting the multiple metrics, and comparing multiple platforms using only this single-value-of-merit, and (ii) using a tournament format that allows for multiple participants (platforms) to be compared across multiple criteria.

The former requires metrics to be easy to compare and compose, that is, to be normalized, to be similarly distributed, to have the same meaning of better (e.g., lower values), to be of importance universally recognized across the field of practice so that weights can be easily ascribed. The latter requires a good tournament format, which does not favor any of the participants, and which does not make participation cumbersome through a large set of rules.

4.2.1 Single value-of-merit Approach

Where metrics (See Section 2.5.3) are collected repeatedly e.g., each combination of algorithm and dataset, a single value-of-merit can be summarized following the typical processes of benchmarking HPC systems [?]:

- For Performance metrics, the arithmetic mean across all data.
- For Throughput metrics, because they are rate metrics, in two consecutive steps: (i) let a be the arithmetic mean of the performance metric (e.g., processing time) and w be the (constant, total) workload (e.g., count of edges plus vertices), (ii) report as throughput metric the ratio between w and a . In other words, instead of averaging the rate per sample, that is, $EVPS_i$ for sample i , Graphalytics first averages the performance metric and then reports the rate.
- For Cost metrics, the harmonic mean across all data. This is because the denominator (e.g., EVPS for PPP) gives meaning to the ratio (TCO is constant across experiments with the same System Under Test), which indicates that the arithmetic mean would be misleading [?, S.3.1.1].
- For Speedup and other ratio metrics, the geometric mean across all data.

In many ranking approaches, it is convenient to have each participant compared not against each other, but against a *baseline* (or golden standard, or reference). Among the approaches, we have in particular considered an adapted version of the method of Fractional Difference Comparison [?]. In this method, a baseline constructed analytically, by ascribing it, for each metric, the best value observed among all platforms in the competition; in other words, the baseline is possibly sub-optimal, but could presumably be achievable in practice, by combining the best features of the participating platforms. (Future in v.1.0)

4.2.2 Tournament-based approach

In a tournament-based approach, the system performance is ranked by means of competitive tournaments [?]. Generally, a Round-Robin pair-wise tournament [?] (from hereon, *tournament*) of p participants involves a balanced set of (pair-wise) comparisons between the results of each pair of participants; if there are c criteria to compare the participants, there will be $\frac{1}{2} \times c \times p(p - 1)$ pair-wise comparisons. In a pair-wise comparison, a pre-defined amount of points (often, 1 or 3) is given to the better (*winner*) participant, from the pair. It is also common to give zero points to the worse (*loser*) participant, from the pair, and to split the points between participants with equal performance. Similar tournaments have been used for decades in chess competitions, in professional sports leagues such as (European and American) football, etc.

We do not consider here other pair-wise tournaments, such as replicated tournaments [?] and unbalanced comparisons [?], which have been used especially in settings where comparisons are made by human referees and are typically discretized on 5-point Likert scales, and thus are quantitatively less accurate than the Graphalytics measurements.

5 AUDITING RULES

The auditing rules are currently under discussion in the LDBC consortium. We welcome feedback from all stakeholders, and also from both industry and academia organizations. (Future in v.1.0)

6 IMPLEMENTATION INSTRUCTIONS

Graphalytics provides a set of benchmark software and resources which are open-source and publicly available. This chapter explains how to work with Graphalytics softwares and enumerates the available benchmark resources which are necessary for the benchmark.

6.1 Graphalytics Softwares and Documentation

The Graphalytics team develops and maintains the core Graphalytics softwares, which facilitates the benchmark process and is extendable for benchmarking and analysing the performance of various graph processing platforms.

Graphalytics Core: The Graphalytics Core contains the core implementation for the Graphalytics benchmark, provides an programmable interface for platform drivers.

Link: https://github.com/ldbc/ldbc_graphalytics

Graphalytics Wiki: The Graphalytics wiki provides detailed instructions on how to install and run the Graphalytics benchmark.

Link: https://github.com/ldbc/ldbc_graphalytics/wiki

Graphalytics Website: The Graphalytics website presents the newest updates of the project, provides links to benchmark resources, and organizes periodically global competitions.

Link: (working in progress) (Future in v.1.0)

Reference Implementation: The Graphalytics reference platform driver provides an simple but correct implementation of all core algorithms defined in Section 2.3, which serves as a guideline for correct implementation of more sophisticated graph processing platforms.

Link: https://github.com/ldbc/ldbc_graphalytics_platforms_reference

Driver Prototype: The Graphalytics benchmark suite can be easily extended by developing a platform driver for your own platform, built upon the platform driver template. Follow the detailed instructions in the manual on implementing driver.

Link: https://github.com/ldbc/ldbc_graphalytics/graphalytics-platforms-archetype (Future in v.1.0)

6.2 Graphalytics-related Softwares

To enhance the depth and comprehensiveness of the benchmark process, the following software tools are integrated into Graphalytics. Usage of these tools is optional, but highly recommended.

Datagen The LDBC-SNB Data Generator is the responsible of providing the data sets used by all the LDBC benchmarks. This data generator is designed to produce directed labeled graphs that mimic the characteristics of those graphs of real data.

Link: https://github.com/ldbc/ldbc_snb_datagen

Granula: Granula is a fine-grained performance analysis system consists of four main modules: the modeller, the monitor, the archiver, and the visualizer. By using Granula, enriched performance results can be obtained for

each benchmark run, which helps in facilitating in-depth performance analysis, e.g. failure analysis, performance regression testing.

Link: <https://github.com/atlarge-research/granula>

6.3 Graphalytics Platform Driver

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

- Giraph (no vendor optimization):
<https://github.com/atlarge-research/graphalytics-platforms-giraph>
- Graphmat:
<https://github.com/atlarge-research/graphalytics-platforms-graphmat>
- GraphX (no vendor optimization):
<https://github.com/atlarge-research/graphalytics-platforms-graphx>
- OpenG:
<https://github.com/atlarge-research/graphalytics-platforms-openg>
- PowerGraph (no vendor optimization):
<https://github.com/atlarge-research/graphalytics-platforms-powergraph>

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 $N_{out}(v) = \{u \in V | (v, u) \in E\}$ and a set of incoming neighbors $N_{in}(v) = \{u \in V | (u, v) \in E\}$.

A.1 Breadth-First Search (BFS)

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

```
1: for all  $v \in V$  do
2:    $depth[v] \leftarrow \infty$ 
3: end for
4:  $Q \leftarrow \text{CREATE\_QUEUE}()$ 
5:  $Q.\text{PUSH}(root)$ 
6:  $depth[root] \leftarrow 0$ 
7: while  $Q.\text{SIZE} > 0$  do
8:    $v \leftarrow Q.\text{POP-FRONT}()$ 
9:   for all  $u \in N_{out}(v)$  do
10:    if  $depth[u] = \infty$  then
11:       $depth[u] \leftarrow depth[v] + 1$ 
12:       $Q.\text{PUSH-BACK}(u)$ 
13:    end if
14:  end for
15: end while
```

A.2 PageRank (PR)

input: graph $G = (V, E)$, integer $max_iterations$
output: array $rank$ storing PageRank values

```
1: for all  $v \in V$  do
2:    $rank[v] \leftarrow \frac{1}{|V|}$ 
3: end for
4: for  $i = 1, \dots, max\_iterations$  do
5:    $dangling\_sum \leftarrow 0$ 
6:   for all  $v \in V$  do
7:     if  $|N_{out}(v)| = 0$  then
8:        $dangling\_sum \leftarrow dangling\_sum + rank[v]$ 
9:     end if
10:  end for
11:  for all  $v \in V$  do
12:     $new\_rank[v] \leftarrow (1 - d) \frac{1}{|V|} + d \left( \sum_{u \in N_{in}(v)} \frac{rank[u]}{|N_{out}(u)|} + \frac{dangling\_sum}{|V|} \right)$ 
13:  end for
14:   $rank \leftarrow new\_rank$ 
15: end for
```

A.3 Weakly Connected Components (WCC)

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

```
1: for all  $v \in V$  do
2:    $\text{comp}[v] \leftarrow v$ 
3: end for
4: repeat
5:    $\text{converged} \leftarrow \text{true}$ 
6:   for all  $v \in V$  do
7:     for all  $u \in N_{in}(v) \cup N_{out}(v)$  do
8:       if  $\text{comp}[v] > \text{comp}[u]$  then
9:          $\text{comp}[v] \leftarrow \text{comp}[u]$ 
10:       $\text{converged} \leftarrow \text{false}$ 
11:     end if
12:   end for
13: end for
14: until  $\text{converged}$ 
```

A.4 Local Clustering Coefficient (LCC)

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

```
1: for all  $v \in V$  do
2:    $d \leftarrow |N_{in}(v) \cup N_{out}(v)|$ 
3:   if  $d \geq 2$  then
4:      $t \leftarrow 0$ 
5:     for all  $u \in N_{in}(v) \cup N_{out}(v)$  do
6:       for all  $w \in N_{in}(v) \cup N_{out}(v)$  do
7:         if  $(u, w) \in E$  then ▷ Check if edge  $(u, w)$  exists
8:            $t \leftarrow t + 1$  ▷ Found triangle  $v - u - w$ 
9:         end if
10:      end for
11:    end for
12:     $\text{lcc}[v] \leftarrow \frac{t}{d(d-1)}$ 
13:  else
14:     $\text{lcc}[v] \leftarrow 0$  ▷ No triangles possible
15:  end if
16: end for
```

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

input: graph $G = (V, E)$, integer $max_iterations$
output: array $labels$ storing vertex communities

```
1: for all  $v \in V$  do
2:    $labels[v] \leftarrow v$ 
3: end for
4: for  $i = 1, \dots, max\_iterations$  do
5:   for all  $v \in V$  do
6:      $C \leftarrow \text{CREATE\_HISTOGRAM}()$ 
7:     for all  $u \in N_{in}(v)$  do
8:        $C.ADD(labels[u])$ 
9:     end for
10:    for all  $u \in N_{out}(v)$  do
11:       $C.ADD(labels[u])$ 
12:    end for
13:     $freq \leftarrow C.GET\_MAXIMUM\_FREQUENCY()$  ▷ Find maximum frequency of labels.
14:     $candidates \leftarrow C.GET\_LABELS\_FOR\_FREQUENCY(freq)$  ▷ Find labels with max. frequency.
15:     $new\_labels[v] \leftarrow \text{MIN}(candidates)$  ▷ Select smallest label
16:  end for
17:   $labels \leftarrow new\_labels$ 
18: end for
```

A.6 Single-Source Shortest Paths (SSSP)

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

```
1: for all  $v \in V$  do
2:    $dist[v] \leftarrow \infty$ 
3: end for
4:  $H \leftarrow \text{CREATE\_HEAP}()$ 
5:  $H.INSERT(root, 0)$ 
6:  $dist[root] \leftarrow 0$ 
7: while  $H.SIZE > 0$  do
8:    $v \leftarrow H.DELETE\_MINIMUM()$  ▷ Find vertex  $v$  in  $H$  such that  $dist[v]$  is minimal.
9:   for all  $w \in N_{out}(v)$  do
10:    if  $dist[w] > dist[v] + weight[v, w]$  then
11:       $dist[w] \leftarrow dist[v] + weight[v, w]$ 
12:       $H.INSERT(w, dist[w])$ 
13:    end if
14:  end for
15: end while
```

B DATA FORMAT FOR BENCHMARK RESULTS

The information in this section will be updated according to the implementation. (Future in v.1.0) This appendix shows an example of Graphalytics benchmark results of the reference implementation. Graphalytics benchmark defines a specific data format for the benchmark results. The result is formatted in *json*, and consists of three main component: system under test, benchmark configuration, and experimental results. Figure B.1 depicts the top-level structure of the result format.

Figure B.1: Result Format: Overview

```
{
  "id": "b2940223",
  "system": {...},
  "configuration": {...},
  "result": {...}
}
```

For the system under test, Graphalytics reports the detailed descriptions of the graph analytic platform, the cluster environment, and the benchmark tool.

Figure B.2: Result Format: System under Test

```
"system": {
  "platform": {"name": "reference", "acronym": "ref", "version": "1.4.0",
    "link": "https://github.com/ldbc/ldbc_graphalytics_platforms_reference"},
  "environment": {
    "name": "DAS Supercomputer", "acronym": "das5", "version": "5",
    "link": "http://www.cs.vu.nl/das5/",
    "machines": [
      {
        "quantity": 20,
        "operating-system": "Centos",
        "cpu": {"name": "XEON", "cores": "16"},
        "memory": {"name": "?", "size": "40GB"},
        "network": {"name": "Infiniband", "throughput": "10GB/s"},
        "storage": {"name": "SSD", "volume": "20TB"},
        "accel": {}
      }
    ]
  },
},
"benchmark": {
  "graphalytics-core": {
    "name": "graphalytics-core", "version": "1.1.0",
    "link": "https://github.com/ldbc/ldbc_graphalytics"
  },
  "graphalytics-platforms-reference": {
    "name": "graphalytics-platforms-reference", "version": "2.1.0",
    "link": "https://github.com/ldbc/ldbc_graphalytics_platforms_reference"
  }
}
}
```

For the benchmark configuration, the target-scale and the computation resource usage is reported. For each resource type, the baseline resource usage, and the scalability of that resource type is reported.

Figure B.3: Result Format: Benchmark Configuration

```

"configuration": {
  "target-scale": "L",
  "resources": {
    "cpu-instance": {"name": "cpu-instance", "baseline": 1, "scalability": true},
    "cpu-core": {"name": "cpu-core", "baseline": 32, "scalability": false},
    "memory": {"name": "memory", "baseline": 64, "scalability": true},
    "network": {"name": "network", "baseline": 10, "scalability": false}
  }
}

```

Figure B.4: Result Format: Experiment Result

```

"result":{
  "experiments": {
    "e34252":{"id":"e34252", "type": "baseline-alg-bfs", "jobs":["j34252",
"j75352", "j23552"]},
    "e75352":{"id":"e75352", "type": "baseline-alg-pr", "jobs":["j34252", "j75352",
"j23552"]},
    "e23552":{"id":"e23552", "type": "baseline-alg-cdlp", "jobs":["j34252",
"j75352", "j23552"]}
  },
  "jobs":{
    "j34252":{"id":"j34252", "algorithm":"bfs", "dataset":"D100", "scale":1,
    "repetition": 3, "runs":["r649352", "r124252", "r124252"]},
    "j75352":{"id":"j75352", "algorithm":"pr", "dataset":"D1000", "scale":1,
    "repetition": 3, "runs":["r649352", "r124252", "r124252"]},
    "j23552":{"id":"j23552", "algorithm":"cdlp", "dataset":"G25", "scale":1,
    "repetition": 3, "runs":["r649352", "r124252", "r124252"]}
  },
  "runs":{
    "r649352":{"id":"r649352", "timestamp":1463310828849,
    "success":true, "makespan":23423422, "processing-time":2234},
    "r124252":{"id":"r124252", "timestamp":1463310324849,
    "success":true, "makespan":2343422, "processing-time":234},
    "r643252":{"id":"r124252", "timestamp":1463310324849,
    "success":true, "makespan":2343422, "processing-time":234}
  }
}

```

For the experimental results, the set of experiments, the underlying jobs, and the corresponding runs are reported.

C RELATED WORK

Table C.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 [?], only graph elements	D/MC	PowerGraph	S	Rnd	Exp	—	No	No	No
S	Montresor et al. [?]	D/MC	3 classes	0	Rnd	Exp	—	No	No	No
B	HPC-SGAB [?]	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 [?]	D	—	SE+	Exp	Exp	1B Edges	No	No	No
S	Own prior work [?, ?, ?]	D/MC/GPU	10 classes	S	Exp	1-stage	1B Edges	W/S/V/H	No	No
S	Özsu et al. [?]	D	Pregel	0	Exp,Rnd	Exp	—	W/S/V/H	No	No
B	BigDataBench [?, ?], only graph elements	D/MC	Hadoop	S	Rnd	Rnd	—	S	No	No
S	Satish et al. [?]	D/MC	6 classes	S	Exp,Rnd	Exp	—	W	No	No
S	Lu et al. [?]	D	4 classes	S	Exp,Rnd	Exp	—	S	No	No
B	GraphBIG [?]	P/MC/GPU	System G	S	Exp	Exp	—	No	No	No
S	Cherkasova et al. [?]	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 C.1, which is reproduced from [?], 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 C.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 [?, ?] explicitly refers to Graphalytics as its option for future benchmarking of graph analysis platforms.

Previous studies typically tested the open-source platforms Giraph [?], GraphX [?], and PowerGraph [?], but our contribution here is that vendors (Oracle, Intel, IBM) in our evaluation have themselves tuned and tested their implementations for PGX [?], GraphMat [?] and OpenG [?]. 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 [?, ?], 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 [?], XGDBench [?], and LDBC SNB [?]; efforts focusing on RDF graph processing, such as LUBM [?], the Berlin SPARQL Benchmark [?], SP²Bench [?], and WatDiv [?] (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.