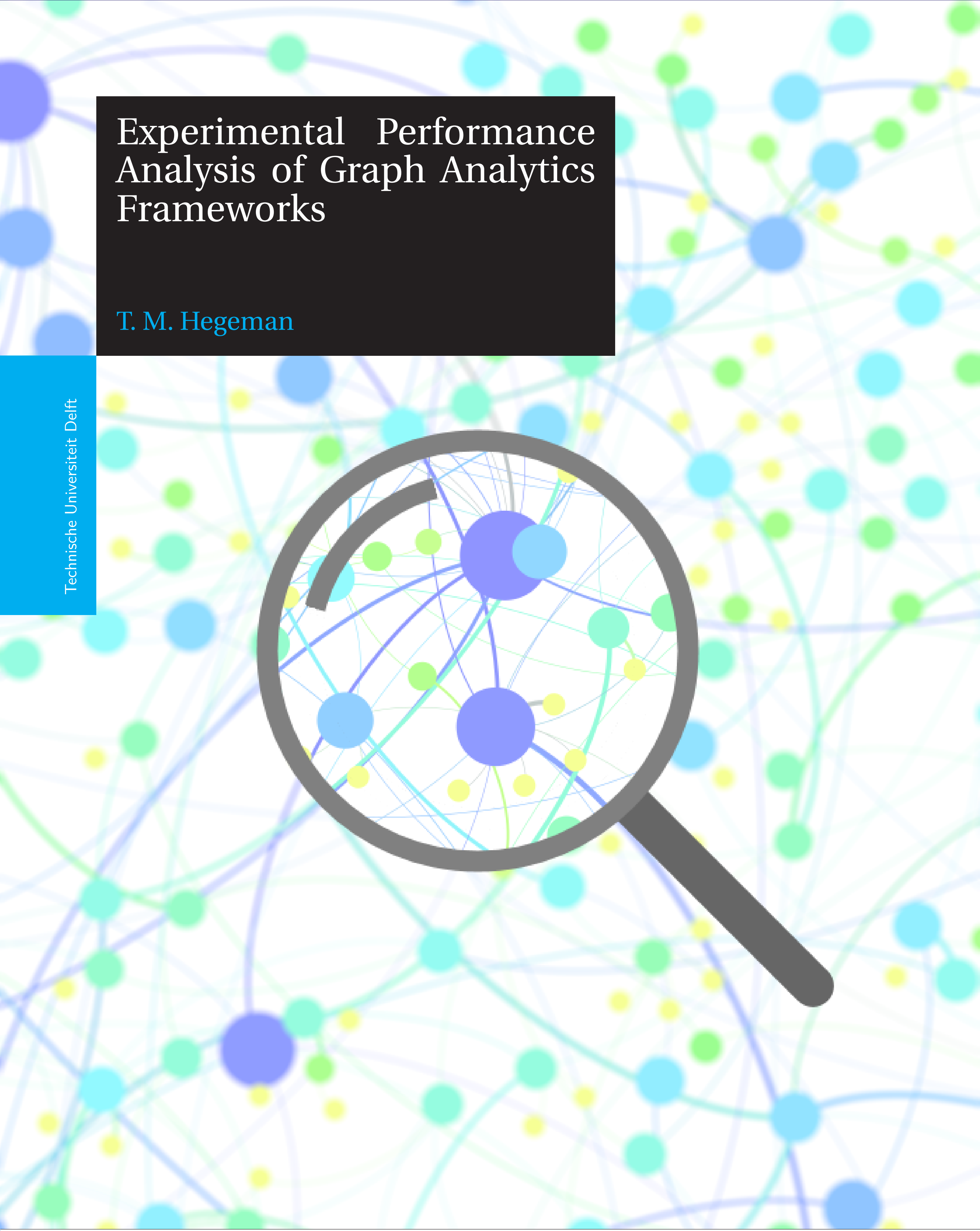
Experimental Performance Analysis of Graph Analytics Frameworks

T. M. Hegeman

Technische Universiteit Delft



Delft University of Technology

Challenge the future

Experimental Performance Analysis of Graph Analytics Frameworks

by

T. M. Hegeman

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Tuesday July 17, 2018 at 10:00 AM.

Supervisor:	Prof. dr. ir. A. Iosup	
Thesis committee:	Prof. dr. ir. A. Iosup,	TU Delft/VU Amsterdam
	Dr. J. S. Rellermeyer,	TU Delft
	Dr. A. E. Zaidman,	TU Delft

Abstract

Big data, the large-scale collection and analysis of data, has become ubiquitous in the modern, digital society. Within the big data landscape, graphs are widely used to study collections of entities and the complex relationships that connect them. The analysis of graphs has applications in social networks, logistics, finance, bioinformatics, and many other domains. With the rapidly increasing amounts of data being collected, analyzing large-scale graphs has become a necessity. To address this need, many dedicated graph analysis frameworks have been developed in recent years. However, their performance is poorly understood.

In this thesis, our goal is to improve insight into the performance of graph analysis frameworks. We design the Graphalytics ecosystem, a set of complementary systems for understanding the performance of graph analysis frameworks, with a focus on two key components. First, we design, implement, and evaluate Graphalytics, a comprehensive benchmark for graph analysis frameworks that facilitates the comparison of performance between these frameworks. Second, we design, implement, and evaluate Grade10, a system for automated, in-depth performance analysis of graph analysis frameworks. Through experimental evaluation of the Graphalytics ecosystem, we gain insight into the performance of six modern graph analysis frameworks.

Preface

This thesis is the culmination of many years of education and research at Delft University of Technology. It was a long, but gratifying, journey and I am happy to have completed it. Throughout my time in Delft, and in particular during my work on this thesis, I have had the pleasure of meeting and working with many people, some of whom deserve a special thank you.

First, I want to thank Alexandru Iosup, my supervisor, for inspiring me to get the most out of myself and any project we have worked on together. During the course you taught in the first year of my Bachelor's degree, you motivated me to go beyond what the program required of me, and to pursue what *I* wanted to get out of my education. Several months later, we started our first collaboration on the topic of big data, and I have enjoyed working with you since that day. Thank you, again, for pushing me to be the best version of myself.

Second, I want to thank some of my many collaborators over the years. Bogdan and Mihai, you mentored me through the early stages of learning how to do scientific research. I want to thank you for your guidance during our collaboration and afterward. Ahmed, Alex Uta, Stijn, and Wing, you have made my time working on the Graphalytics project a great experience through countless discussions, both on- and off-topic. Many thanks also to all members of the @Large Research team for your suggestions and feedback, and for all of our insightful discussions.

Third, I want to thank my friends, Jesse, Stefan, Otto, Pim and Pim, and Elvan, for being supportive throughout my education and for keeping me sane. Jesse and Stefan, thank you for the countless hours we spent together working on courses, and for always being there when I needed feedback or needed to bounce ideas off someone. Thank you for never failing to put a smile on my face, and for the times we spent playing games or watching a movie when anyone of us needed a distraction. You have both impacted my experience at TU Delft in many different ways, and I can not imagine what it would have been like without either of you.

Finally, I want to thank my family for their copious amounts of love, support, and patience. You have always been there for me to help me on this journey.

Tim Matthijs Hegeman
Delft, July 2018

CONTENTS

1

Introduction

The problem and approach proposed in this thesis appear in the context of *big data*. Big data, the large-scale collection and analysis of data, has rapidly become ubiquitous in the modern, digital society. It helps us find information through search engines, lets us connect and communicate via social media, and recommends to us products or movies we may be interested in. Big data increasingly also powers business processes, from fraud detection and market prediction, to understanding and optimizing user experiences. Advances in big data analysis systems are also enabling data-driven scientific discoveries throughout many disciplines, commonly referred to as the *fourth paradigm* of science [1]. The diversity and number of big data applications has also led to significant investments; in 2016 alone, startups in big data received $14.6 billion in venture capital [2], and IDC forecasts double-digit annual growth in big data and businesses analytics revenue, up to $210 billion in 2020 [3].

This thesis focuses on the important data-class of *graph data*, i.e., information about collections of entities and the complex and often arbitrary relationships that connect them. The theory of graphs has been studied since Euler formulated the famous Seven Bridges of Königsberg problem in 1736. However, the widespread application of graph analysis for scientific and commercial purposes is a more recent phenomenon driven by the increased availability of both data and computational resources. Graph analysis techniques can provide insight into the web of relationships, e.g., by identifying tight-knit communities of entities, by ranking the importance of entities in a graph, or by determining how pairs of entities are (indirectly) related. The analysis of graphs can be applied in many applications, including social network analysis [4–6], computer network security [7], bioinformatics [8, 9], and power grid operations [10]. Our own survey in 2018 [11] covers tens of applications in 11 application domains. Recent developments include successful startups in graph data management (Neo4j [12]), and the inclusion of graph analysis techniques in data processing stacks, both community-driven (GraphX on Spark [13, 14]) and industrial (Oracle Big Data Spatial and Graph [15] and Cray Graph Engine [16]). Serving as further evidence for the timeliness of graph analysis research, we observe in the academic community an increase in venues for graph data management and graph analysis. Several dedicated workshops have been organized, e.g., GRADES since 2013 [17], NDA since 2016 [18], and HPGDMP in 2016 [19]. Graph research is also prevalent in major data management conferences; at SIGMOD 2017 [20] and VLDB 2017 [21] there were more sessions on graphs than on any other topic.

We focus in particular on the problem of understanding the performance of graph analysis systems. Due to the rapidly increasing amounts of data being collected, there is a growing demand for analyzing *large* graphs. To support this need, graph analysis algorithms have been designed to run in parallel and distributed environments. Many techniques have been developed for structuring and optimizing parallel and distributed data processing applications. These techniques exploit a variety of properties typical to such applications, including highly regular computation, predictable communication patterns and strong locality properties. In contrast, graph analysis applications typically are data-driven and unstructured, have poor data locality, have irregular computation patterns[1], and are predominantly bound by data access instead of computation [22]. To address these properties, tens of dedicated graph analysis systems have been developed since 2010 [23], many inspired by Google's Pregel [24]. Although many developers of such systems claim that they outperform state-of-the-art, performance comparisons between systems are not standardized, are rarely comprehensive [25], and, as several studies have shown [25–27], are potentially biased.

[1] Research on graph analysis is frequently published in the Workshop on Irregular Applications: Architectures and Algorithms (IA^3).

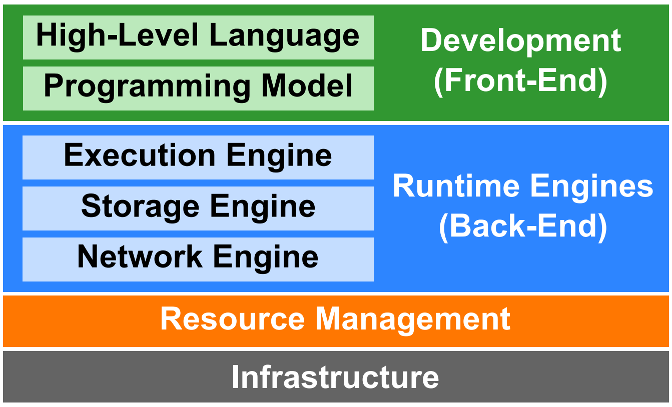


Figure 1.1: Architecture of a typical large-scale graph analysis platform.

1.1. A Primer on Graph Analysis Systems

A modern large-scale graph analysis system comprises a stack of inter-operating components. Figure 1.1 depicts a typical architecture for such a system. The architecture includes a front-end that allows users to program their graph analysis applications, runtime engines that execute the user-defined program, resource management that shares resources between applications in a cluster, and the underlying (hardware) infrastructure. Although the depicted architecture is representative for many graph analysis systems, not all components in the front-end and back-end are present in each system (e.g., some systems do not interface with a high-level language [28, 29]). Furthermore, some components are conceptually distinct, but are implemented together (e.g., the runtime of Giraph [30] includes both an execution and a network engine).

Key to understanding the execution of a graph analysis application, and therefore to understanding the performance of a graph analysis system, are the programming model and the execution engine. The program model of a system dictates how users can interface with the system and express their application. Most parallel and distributed graph analysis systems use a high-level programming abstraction [23], that is, they hide data partitioning and communication mechanisms from the user [31]. Frequently used programming models [31] include vertex-centric processing [24], Scatter-Gather [32], Gatter-Apply-Scatter [28], and subgraph-centric processing [33]. A user builds their graph analysis application by implementing a small set of functions (hooks) for their specific algorithm (e.g., in the Scatter-Gather model, users implement a *scatter* and a *gather* function).

An execution engine takes a user program written for a supported programming model and executes it. Responsibilities of the execution engine include loading and partitioning the input data, communication and synchronization between machines, ensuring fault-tolerant execution, and calling functions provided by the user when appropriate (as defined by the programming model). This has important implications for how users understand performance; much of a job's execution takes places in the execution engine, outside of the user's control and. Also, most execution engines use an iterative, synchronous execution model [31], which potentially introduces a variety of performance issues, including synchronization overhead and imbalanced execution.

1.2. Problem Statement

Our goal is to improve insight into the performance of graph analysis systems, for both users and developers. We identify two distinct but related problems in state-of-the-art practices for studying performance. First, there is an increasing need for systematically comparing the performance of graph analysis platforms, to help users select a platform that will efficiently perform the types of analysis specific to their needs. As new graph analysis platforms are being designed [23], their authors typically evaluate a platform's performance through experimental evaluation and comparison against other state-of-the-art platforms [25]. However, experiments performed by different authors are rarely comparable, due to differences in the datasets they use, the graph algorithms they run, their environment, etc. Dangerously for the community, the de facto standard at the start of this work, Graph500 [34], focuses on a single algorithm and dataset type, which threatens to bias the community efforts on problems that have limited coverage of real-world applications. A standardized process for evaluating and comparing graph analysis platforms is needed.

Second, there is an increasing need to find good configuration parameters (tuning), and to identify performance issues that systems designers and engineers can address (re-designing). As graph analysis platforms evolve and their complexity increases, their performance becomes more difficult to understand. Many existing performance analysis tools for parallel computation [35–37] focus on low-level analysis and are tailored to traditional high-performance computing (HPC) applications. These applications are typically built on top of a limited set of communication and synchronization primitives (e.g., those provided by MPI or OpenMP). In contrast, many graph analysis platforms provide a powerful API that hides the internal complexity [31], so users will be unable to interpret low-level performance analysis. Further, this type of analysis may be prohibitively expensive in the amount of data collected for large-scale distributed systems. A novel approach is needed to provide insight into the performance of graph analysis systems that can be used by both users and developers.

1.3. Main Research Questions

To facilitate the comparison and analysis of the performance of graph analysis platforms, we address in this work the following research questions:

1. **RQ1: How to systematically yet efficiently compare the performance of graph analytics frameworks?** The performance of a graph analytics framework is an important factor in selecting the right framework for a given use case. Thus, performance comparison is of particular importance to potential users of graph analytics frameworks. A method for systematic comparison is also important for developers of frameworks to provide fair and insightful comparison against state-of-the-art. We explicitly include the requirement for an approach that is both systematic, i.e., it produces representative results covering a wide range of possible applications, and efficient, i.e., it aims to minimize the time and effort required to compare a set of frameworks.

2. **RQ2: How to automatically analyze performance in the execution of graph analytics workloads?** Analyzing the performance of graph analytics workloads is critical to identifying the limitations of a platform, understanding the impact of design choices or optimizations on performance, and ultimately improving performance. This affects both developers and users of graph analytics frameworks. Developers may want to identify components and processes in their framework that could be optimized to achieve large performance gains for common workloads. Users may want to learn how they should change their algorithms or tune their setup to get faster response times for their use cases. An automated approach is needed to help both developers and users analyze the often overwhelming volume of performance data that can be collected for large-scale systems.

3. **RQ3: How to validate the designed approaches through prototypes?** To understand how well our approaches for understanding the performance of graph analysis platforms work, we use real-world experimentation through prototypes. Answering this question requires the design of experiments that validate the chosen approaches in practice and, where possible, validate individual aspects of an approach. It also requires (sometimes painstaking) engineering effort in deploying and using different graph analysis systems, many of which have not been designed and engineered for production use. In this situation, science and engineering meet as part of computer science methodology.

1.4. Approach

Guided by the first two research questions, we identify two main dimensions in understanding the performance of graph analysis platforms: *breadth* and *depth*. The breadth dimension characterizes the variety and number of workloads and systems for which performance is inspected or compared. The depth dimension characterizes the depth of the performance data that is collected and analyzed. Within the breadth-depth spectrum, we have designed a set of complementary systems which we collectively refer to as the *Graphalytics ecosystem*. Figure 1.2 depicts the four main components of the Graphalytics ecosystem and their relative placement along the breadth and depth dimensions. This thesis makes key contributions to two of these components, Graphalytics and Grade10.

Addressing RQ1, we design Graphalytics, a benchmark for graph analytics frameworks with a focus on breadth. Included in the design of Graphalytics are its architecture, the specification of benchmark elements (i.e., the workloads), the benchmark process (i.e., the metrics), and a renewal process.

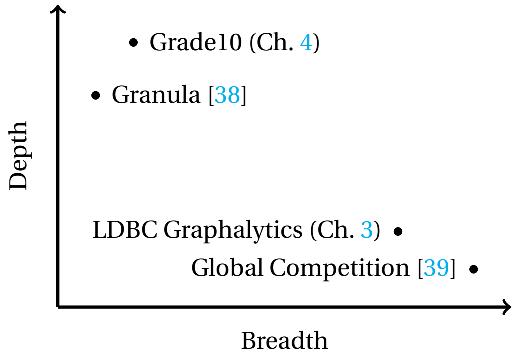


Figure 1.2: The Graphalytics ecosystem defined along two dimensions of understanding performance: *breadth* and *depth*. We present in this thesis Graphalytics (Chapter 3) and Grade10 (Chapter 4).

Next, to address RQ2 we design Grade10, a system for automated, in-depth performance analysis of graph analytics workloads. We present novel approaches for modeling the execution of a graph analysis job, resource attribution, bottleneck detection, and performance issue identification.

Finally, we implement Graphalytics and Grade10 and evaluate both through real-world experiments to address RQ3. We use the Graphalytics benchmark to compare the performance of six state-of-the-art graph analytics frameworks, including three community-driven and three industry-driven frameworks. We evaluate Grade10 by assessing for two of its main components how a variety of mechanisms and possibilities for tuning impact the outcome of Grade10's analysis. We also use Grade10 to analyze the performance of two widely used graph analysis frameworks and a subset of the Graphalytics workload. All experiments are performed on the DAS-5 [40], a distributed supercomputer that is representative of a typical environment for big data analytics.

1.5. MAIN CONTRIBUTIONS

The main contributions of this thesis are:

1. (Conceptual) The first contribution is the design of Graphalytics, a benchmark for graph analytics frameworks. We propose for Graphalytics a new process for selecting relevant workloads (algorithms and datasets) and a renewal process to systematically update the benchmark as real-world usage of graph analysis evolves. We further introduce a comprehensive set of metrics and corresponding experiments for comparing the performance of graph analysis platforms. This contribution has resulted in two publications: **M. Capotă, T. Hegeman, et al., *Graphalytics: A big data benchmark for graph-processing platforms*, in GRADES (2015)** and **A. Iosup, T. Hegeman, et al., *LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms*, in PVLDB 9(13) (2016)**.

2. (Conceptual) The second contribution is the design of Grade10, a system for automatically identifying performance bottlenecks and performance issues in graph analysis jobs. We propose for Grade10 novel processes and mechanisms for modeling the execution of a graph analysis job, resource attribution, bottleneck detection, and performance issue identification.

3. (Experimental) The third contribution is the comparison of six state-of-the-art graph analysis platforms using Graphalytics, including three community-driven and three industry-driven systems. We analyze their performance using a variety of metrics and experiments focusing on baseline performance, scalability, and robustness.

4. (Experimental) The fourth contribution is the experimental validation of the processes and mechanisms included in Grade10. Furthermore, we use Grade10 to analyze the performance of two widely used graph analysis platforms, for a variety of workloads.

5. (Software Artifact) We have developed the Graphalytics benchmark as free open source software, available on GitHub [41]. The implementations of Graphalytics drivers for the platforms we tested are also

available as FOSS in the @Large Research organization on GitHub [42], except where prohibited by licenses.

6. (Software Artifact) We have released the source code of Grade10 as a research prototype on GitHub [43].

7. (Standardization) LDBC Graphalytics is now the de facto standard benchmark for industry and academia working in the databases and middleware communities. In contrast, Graph500, a subset of Graphalytics, remains the de facto standard for industry and academia working in the HPC community.

1.6. Guidelines for Reading

The remainder of this thesis is structure as follows. In Chapter 2 we discuss background information and related work on performance comparison for graph analysis systems, and on performance analysis for graph analysis systems and, more generally, for parallel and distributed systems. We present the design of LDBC Graphalytics in Chapter 3, and the design of Grade10 in Chapter 4. In Chapter 5 we evaluate experimentally LDBC Graphalytics and Grade10. Finally, in Chapter 6 we conclude and present directions for future work.

To readers who are interested in learning about performance benchmarking using LDBC Graphalytics, we recommend reading Section 2.1, Chapter 3, Chapter 5 (except Sections 5.2.4 and 5.5) and Chapter 6. To readers who are interested in learning about in-depth performance analysis using Grade10, we recommend reading Section 2.2, Chapter 4, Chapter 5 (except Sections 5.2.3 and 5.4) and Chapter 6. Finally, to readers who are interested in learning more about the Graphalytics ecosystem, we recommend visiting `https://graphalytics.org/docs` for a list of our publications and presentations.

2

Background

In this chapter we discuss related work on understanding the performance of graph analysis systems. In Section 2.1 we survey existing benchmarks and experimental performance studies for graph analysis systems. In Section 2.2 we discuss existing approaches for performance analysis of parallel and distributed systems.

2.1. Benchmarks for Graph Analysis Systems

To understand the requirements of a comprehensive benchmark for graph analysis systems, we compare in this section existing benchmark proposals and experimental performance studies, and we identify their shortcomings.

Table 2.1 summarizes and compares Graphalytics (presented in Chapter 3) with previous studies and benchmarks for graph analysis systems. As Table 2.1 indicates, Graphalytics covers a wider range of graph analysis systems than existing approaches. Furthermore, it is the first to include a two-stage data- and expertise-driven workload selection process. In contrast, the state-of-the-art selection process is guided by personal experience or not explained, and the workloads used by many studies consist of a narrow selection of (classes of) datasets and algorithms. Graphalytics further includes two novel contributions in its renewal process and evaluation of the robustness of graph analysis systems.

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.

Previous studies typically tested the open-source platforms Giraph [30], GraphX [14], and PowerGraph [28], but our contribution here is that vendors (Oracle, Intel, IBM) in our evaluation have themselves tuned and tested their implementations for PGX [55], GraphMat [29] and OpenG [53]. 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 [56, 57], 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 [58], XGDBench [59], and LDBC Social Network Benchmark [60]; efforts focusing on RDF graph processing, such as LUBM [61], the Berlin SPARQL Benchmark [62], SP^{2}Bench [63], and WatDiv [64] (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.

2.2. Performance Analysis of Parallel and Distributed Systems

Performance analysis has become increasingly challenging due to the complexity of modern hardware and software. The introduction of mechanisms such as parallelism, caching, and asynchronous execution has led to significant performance improvements, but has also made performance unpredictable and hard to understand. Performance of large-scale parallel and distributed systems is especially difficult to analyze;

Table 2.1: Summary 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		Target System		Design					Includes tests for...		Other Distinguishing
Type	Name [Publication]	Structure	Programming	Input	Data.	Algo.	Scalable?	Renewal	Scalability	Robustness	Features
B	CloudSuite [44], latest graph elements	D/MC	1 class (PowerGraph)	S	Rnd(3)	Exp(1)	—	No	No	No	Micro-architecture
S	Montresor et al. [45]	D/MC	3 classes	0	Rnd(7)	Exp(1)	—	No	No	No	Distribution model
B	HPC-SGAB [46]	P	—	S	Exp(RMAT)	Exp(1+)	—	No	No	No	HPC architecture
B	Graph500 [34]	P/MC/GPU	—	S	Exp(RMAT)	Exp(1)	—	No	No	No	HPC architecture, industry support
B	WGB [47]	D	—	SE+	Exp(RTG)	Exp(several)	1B Edges	No	No	No	Query type
S	**Own work, prior to Graphalytics** [25, 48, 49]	D/MC/GPU	10 classes	S	Exp(6,RMAT)	1-stage(5 classes)	—	No	W/S/V/H	No	Distributed and other platforms, Scalability
S	Han et al. [26]	D	1 class(Pregel)	0	Exp(5)+Rnd(1)	Exp(4)	—	No	W/S/V/H	No	Pregel-like features
B	BigDataBench [50, 51], only graph elements	D/MC	1 class (Hadoop)	S	Rnd(7)	Rnd(2)	—	No	Strong	No	—
S	Satish et al. [52]	D/MC	6 classes	S	Exp(6)+Rnd(1)	Exp(4)	—	No	Weak	No	Optimized code, classes of distributed platforms
S	Lu et al. [27]	D	4 classes	S	Exp(5c/7)+Rnd(1)	Exp(5c/7)	—	No	Strong	No	Effects of individual techniques
B	GraphBIG [53]	P/MC/GPU	1 class (System G)	S	Exp(5)	Exp(10)	—	No	No	No	Memory sub-system, System G-like features
S	Cherkasova et al. [54]	MC	1 class(Galois)	0	Rnd(2)	Exp(5)	—	No	No	No	Memory sub-system, Galois-like features
B	**LDBC Graphalytics (this work)**	D/MC/GPU	10+ classes	SE+	2-stage(10,RMAT,LDBC)	2-stage(6 classes)	1B Edges	Yes	W/S/V/H	Yes	Distributed and other platforms, industry support
Legend: S - Study, B - Benchmark; D - distributed system, P - parallel system, MC - single-node multi-core system, GPU - using GPUs.											
Input: **0** - no parameters; **S** - parameters to define scale; **E** - parameters to define edge properties; **+** - parameters to define other graph properties, e.g., clustering coefficient.											
Data[sets]/Algo[rithms]: **Rnd** - reason for selection not explained; **Exp** - selection seems guided by personal expertise; **1-stage** - data-driven selection; **2-stage** - 2-stage data-driven and expertise-driven process.											
Can test scalability? Weak/Strong/Vertical/Horizontal.											
"—" - feature not available or unclear from publication(s).											

communication between independent machines creates new challenges in understanding the order of events in the system, correlating performance characteristics across machines, identifying bottlenecks, etc.

In response to these new challenges, performance engineering has introduced tens of new techniques since the seminal work by Connie Smith [65] and Raj Jain [66]. Approaches have been proposed for analyzing the performance of a variety of distributed systems, although none have specifically focused on graph analysis systems. We discuss in this section several prominent (classes of) approaches and their applicability to understanding the performance of graph analysis systems.

2.2.1. Performance Modeling and Prediction

Performance models are used to estimate or predict the performance of an application for a given workload and setup, without having to run the application. A performance model defines how the expected or peak performance of an application depends on a set of input parameters describing the execution environment (e.g., CPU speed, network topology) and the workload or application (e.g., input size, computation intensity, communication patterns). Performance predictions may be derived from a model and its parameters through analytical or stochastic techniques, or though simulation. The outcome may be used to determine the bottlenecks of an application for different workloads, evaluate which configuration of a system is likely to yield the best performance, etc.

Models are commonly used in the high performance computing (HPC) community to study parallel computations, e.g., MPI-based applications. Several abstract models of parallel computing, such as LogP [67] and various derivatives [68–70], have been developed to reason about and predict the performance of message-passing applications. These models are specifically tailored to the message-passing model and the highly regular execution of most HPC applications. The roofline model [71] provides insight into the performance of floating-point programs by computing the theoretical peak performance given the memory and compute bandwidth of the system and the operational intensity of the program. However, the roofline model does not consider network communication, which is an integral component in the performance of graph analysis systems.

More recently, several performance models have been developed by the big data community, in particular for MapReduce systems [72–74]. However, unlike most HPC performance models, the models for MapReduce systems require either a lot of parameters or a set of highly specific parameters to provide accurate predictions. As stated by Culler et al. [67], "[...] a model must strike a balance between detail and simplicity in order to reveal important bottlenecks without making analysis of interesting problems intractable." Performance models for big data systems have not yet struck such a balance. Likewise, we expect that any accurate performance model for graph analysis systems requires many parameters to capture the irregularity of both graph data and algorithms (e.g., the distribution of edges, communication patterns that vary over time). Thus, such models would be difficult to define and tune, yet provide only limited insight.

2.2.2. Empirical Performance Analysis

A widely used approach for understanding the performance of an application is to observe how it performs in practice. This approach of empirical performance analysis consists of collecting a variety of metrics at runtime that describe the performance of an application. Typical metrics include application-level metrics, such as throughput or latency, and system-level metrics, such as the time spent per function, number and volume of messages sent, or time spent on synchronization. In this section we present several (classes of) empirical performance analysis techniques.

2.2.2.1. Sampling Profilers and Tracing

Two techniques for collecting performance metrics are sampling and tracing. Although sampling and tracing are distinct approaches, they are often provided by the same tool, e.g., TAU [35], Vampir [36], Paraver [37], and Dimemas [75]. Sampling is the act of periodically capturing the state of an application's execution, e.g., the call stack of each thread, and the values of various hardware and software counters. The collected samples are used to build a statistical profile of an application, i.e., an estimation of how much time is spent in each function. Sampling can be performed with little overhead, but the generated profiles lack chronological data and can not distinguish outliers (e.g., individual function calls that take exceptionally long). Thus, aggregate profiles would provide limited insight into the irregularity of graph analysis workloads.

In contrast, a trace captures individual performance events and does not aggregate at runtime. Captured events may included the start and end of each function and system call, the size and type of every message sent, etc. A trace allows for accurate reconstruction of how an application performed throughout its execution. However, the overhead of tracing is significant due to the amount of data that is collected and stored. Although we envision use cases for tracing in understanding the performance of graph analysis systems, low-overhead approaches are desired.

Common to both profiling and tracing is their strong ties to the source code of the application being analyzed; performance metrics are typically captured per function or per instruction. This approach is tailored towards HPC applications in which a user explicitly invokes functions from MPI or other libraries. Many big data applications are built on high-level abstractions and are structured differently; the big data runtime dictates the control flow and invokes user code as needed. For example, a user of Pregel only provides a *vertex program* whereas the runtime loads the graph, exchanges message between machines, ensures fault-tolerant execution, and calls the user's program where appropriate. Consequently, a sizable fraction of the execution time of a graph analysis job is spent in parts of the runtime that the user has not explicitly invoked. For users this means that when a profiler or tracing tool points out bottlenecks in the runtime, they will likely need extensive knowledge about the system's internals to understand why the bottleneck exists or how to solve it.

2.2.2.2. Dedicated Techniques for Big Data Systems

In response to the many differences between modern big data systems and traditional HPC applications, the performance engineering and big data communities have developed new approaches for the empirical analysis of big data systems.

Closest to our work are approaches that broadly address the identification of bottlenecks in big data systems. Ousterhout et al. [76] propose *blocked time analysis* as an approach to estimating the impact of disk and network usage on task durations in Apache Spark. Their estimations are based on task speedups that would be attained with an infinitely fast disk or network and a simulation of the Spark scheduler with the reduced task runtimes to compute the overall makespan. Otus [77] uses resource attribution to map utilization data captured per process to high-level entities in a MapReduce job (map tasks, reduce tasks, data node, etc.). This allows resource utilization in a shared MapReduce cluster to be traced back to individual jobs or users. Unlike the big data systems addressed by these works, most graph analysis systems do not use a task-based execution model, so we can not apply existing techniques directly. We do, however, incorporate some aspects in our approach.

Many other approaches apply statistical or machine learning methods to identify performance anomalies. BDTune [78] combines a correlation-based approach to detect anomalies and a threshold-based approach to identify straggler tasks and workload imbalance in task-based big data systems. Similarly, Sonata [79] uses correlation-based performance analysis for MapReduce. It correlates (straggler) tasks with resource utilization in a MapReduce job to identify bottlenecks. Marcu et al. [80] correlate the execution plans of Spark and Flink with resource usage to understand performance differences between the platforms. Qi et al. [81] build a decision tree to identify straggler tasks in MapReduce using a large set of metrics, including stage durations, data skew, and resource utilization. Common to these approaches is the goal of identifying outliers and identifying their bottlenecks. In contrast, we focus in our approach on understanding the bottlenecks of *all* components in the system, because we want to provide insight into the performance of graph analysis systems at any point in its execution, not just when outliers are detected.

Other approaches focus on guiding the manual analysis of performance data. For example, HiTune [82] models big data applications as dataflow graphs with computation vertices and communication edges. By depicting task durations and system utilization over time, HiTune helps users manually diagnose a variety of performance issues in dataflow systems. Dias et al. [83] manually inspect the performance of Spark jobs with respect to four dimensions (data layout, task placement, parallelism, load balancing) to understand how various aspects of a job can non-trivially impact performance. As stated by RQ2 (Section 1.3), an automated approach for analyzing the performance of graph analysis is desired.

2.2.2.3. Performance Anomaly Detection and Bottleneck Identification

Another class of performance analysis techniques focuses on *performance anomaly detection* and *bottleneck identification* (PADBI). We base our study of PADBI systems on a recent survey [84] on the challenges of and approaches for PADBI. The goal of PADBI systems is to detect unexpected or unwanted changes in performance (*anomalies*) and their causes (*bottlenecks*). The field of PADBI focuses on analyzing performance as expressed by *key performance indicators* (KPI), i.e., metrics represent the performance of a system over time. Commonly used KPIs include response time and throughput. The PADBI community studies primarily the performance of systems that continuously perform *the same or similar tasks*. Examples of such systems include web applications responding to a stream of user requests, databases processing queries, and file servers transferring data to and from clients.

Graph analysis systems are fundamentally different from the systems studied by the PADBI community; their workloads are highly irregular. Approaches for anomaly detection are based on the notion of "expected behavior" [84], that is, the expected evolution of the key performance indicator over time. In this context, anomalies are defined as deviations from the expected behavior. For graph analysis systems, deviations in performance *are* the expected behavior due to the irregularity inherent in graph data and algorithms.

Based on this difference, we argue that using PADBI to study the performance of a graph analysis system across many jobs is not insightful. As shown in our experimental evaluation of graph analysis systems using Graphalytics (Section 5.4), changes in the workload (algorithms and datasets) of a graph analysis system result in significant differences in execution time, that is, there is no relation between the durations of independent jobs. Thus, analyzing the throughput or response time of graph analysis jobs over time for anomalies yields little to no insight into the performance of the system executing those jobs.

Similarly, applying PADBI techniques within an individual graph analysis job is not likely to be insightful. A prerequisite to apply anomaly detection is the definition of a KPI and its expected behavior over time. Currently, there is no known metric that accurately describes the performance of a graph analysis job over time. Typical metrics based on data processed over time are insufficient, because they do not account for the irregular computation and communication patterns of graph analysis. Furthermore, a typical job consist of multiple distinct subtasks (e.g., loading, processing, and writing results) with potentially incomparable performance metrics, so anomaly detection would be restricted to individual subtasks of a job.

3

LDBC Graphalytics: A Benchmark for Large-Scale Graph Analytics Platforms

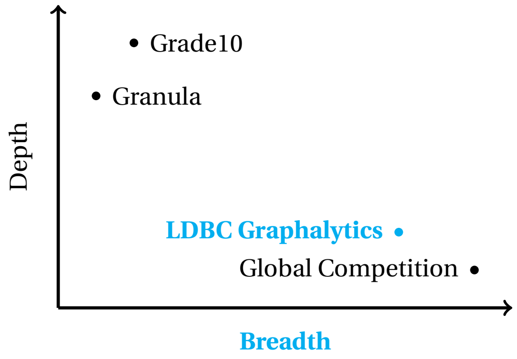


Figure 3.1: The position of LDBC Graphalytics within the Graphalytics ecosystem.

We address in this chapter our first research question: How to systematically yet efficiently compare the performance of graph analysis frameworks? Our approach is a benchmark for graph analysis, *LDBC Graphalytics*, which provides a broad understanding of the performance of graph analysis workloads. Graphalytics tests a graph analysis framework, consisting of a software platform and underlying hardware system. Graphalytics models holistic graph analysis workloads, such as computing global statistics and clustering, which run on the entire dataset on behalf of a single user. The position of the Graphalytics benchmark within the Graphalytics ecosystem is depicted in Figure 3.1.

The work presented in this chapter has been published [49, 85] and is the result of a collaborative effort with partners from both academia and industry, organized through LDBC [86]. My conceptual contributions span many aspects of Graphalytics, most notably in architecture design, workload selection, and experiment design. Furthermore, I was the technical lead from the inception of the Graphalytics project until the publication of [85]. Absent in this chapter are the integration of Granula [38] and Datagen [60] into Graphalytics, for which my contributions were minor.

The remainder of this chapter is structured as follows. The requirements are laid out in Section 3.1. We introduce the architecture of Graphalytics in Section 3.2. The benchmark specification is presented in Section 3.3, which defines the benchmark elements, and Section 3.4, which describes the benchmark process. Finally, we present a process for periodically renewing the Graphalytics benchmark in 3.5.

3.1. Requirements

A benchmark is always the result of a number of design choices, responding to a set of requirements. In this section we discuss the main requirements addressed by LDBC Graphalytics:

(R1) Target platforms and systems: benchmarks must support any graph analysis platform operating on any hardware system. For platforms, we do not distinguish between programming models and support different models, including vertex-centric, gather-apply-scatter, and sparse matrix operations. For systems, we target the following environments: distributed systems, multi-core single-node systems, many-core GPU systems, hybrid CPU-GPU systems, and distributed hybrid systems. Without R1, a benchmark could not service the diverse industrial following of LDBC.

(R2) Diverse, representative benchmark elements: data model and workload selection must be representative and have good coverage of real-world practice. In particular, the workload selection must not only include datasets or algorithms because experts believe they cover known system bottlenecks (e.g., they can stress real-world systems), but also because they can be shown to be representative of 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 LDBC community into pursuing cases that are currently interesting for the 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 possibility to validate the algorithm output, 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 a renewal process: unlike many other benchmarks, benchmarks in this area must include a renewal process, that is, not only a mechanism to scale up or otherwise change the workload to keep up with increasingly more powerful systems (e.g., the scale parameters of Graph500), 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. They must make it possible to support R1, provide easy ways to add new platforms and systems to test, and allow practitioners to easily access the benchmark and compare their platforms and systems against those of others. Without R5, a benchmark could easily become unmaintainable or unusable.

3.2. Architecture

The Graphalytics architecture, depicted in Figure 3.2, consists of a number of components, including the system under test and the testing system.

As input for the benchmark, the Graphalytics team provides a benchmark description (1). This description includes definitions of the algorithms, the datasets, and the algorithm parameters for each graph (e.g., the root for BFS or number of iterations for PR). In addition, the system customer, developer, or operator can configure the benchmark (2). The benchmark user may select a subset of the Graphalytics workload to run, or they may tune components of the system under test for a particular execution of the benchmark.

The workload of Graphalytics is executed on a specific graph analysis platform (3), as provided by the user. This platform is deployed on user-provided infrastructure, e.g., on machines in a self-owned cluster or on virtual machines leased from IaaS clouds. The graph analysis platform and the infrastructure it runs on form the system under test (4). The graph analysis platform may optionally include policies to automatically tune the system under test for different parts of the benchmark workload.

At the core of the testing system are the Graphalytics harness services (5). The harness processes the benchmark description and configuration, and orchestrates the benchmarking process. Two components of the workload, datasets (6) and algorithm implementations (i.e., driver code (10)), must be provided by the benchmark user. Datasets can be obtained through public workload archives, or generated using a workload

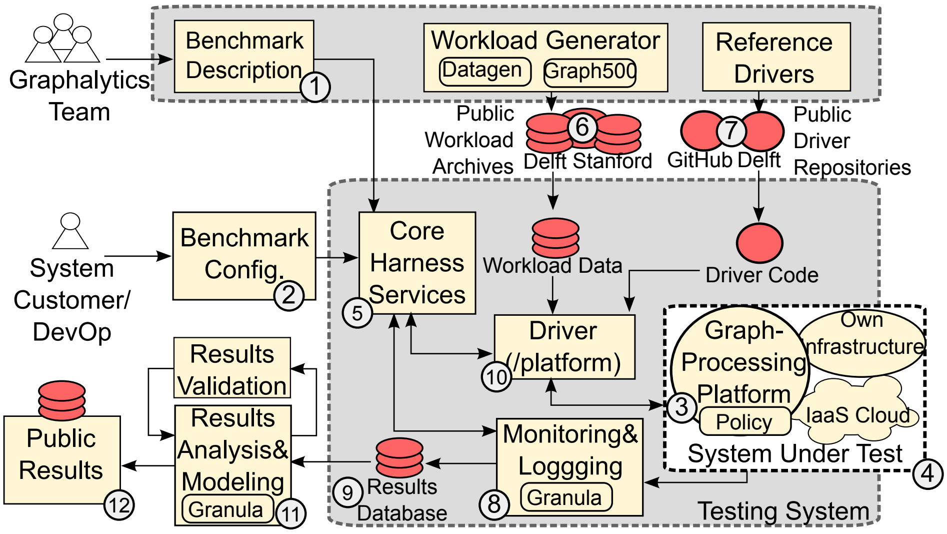


Figure 3.2: Graphalytics architecture, overview.

generator, such as LDBC Datagen. Reference drivers can be provided by platform vendors or obtained from public repository. The Graphalytics team also offers the drivers for a number of platforms (7).

A platform can be integrated with the Graphalytics harness through a platform-specific driver (10). The driver must implement a well-defined API consisting of several operations, including uploading a graph to the system under test (this may include pre-processing to transform the provided dataset into a format compatible with the target platform), executing an algorithm with a specific set of parameters on an uploaded graph, and returning the output of an algorithm to the harness for validation. Additionally, the driver may provide a detailed performance model of the platform to enable detailed performance analysis using Granula [38].

The final component of the testing system is responsible for monitoring and logging (8) the system under test, and storing the obtained information in a results database (9). Optionally, raw monitoring information is gathered using Granula, and can be analyzed after each run or offline to extract rich information about the performance of the system under test (11). Finally, the results are validated, the SLA is checked and the results are stored in a repository to track benchmark results across platforms.

To address the requirement for modern software engineering practices (R5), all components of the Graphalytics architecture provided by the Graphalytics team are developed online as open source software. To maintain the quality of the Graphalytics software, continuous integration is used and contributions are peer-reviewed by the Graphalytics maintainers. Through its development process, Graphalytics also invites collaboration with platform vendors, as evidenced by the contributions already made to Graphalytics drivers.

3.3. SPECIFICATION OF BENCHMARK ELEMENTS

Addressing requirement R2, the key benchmarking elements in Graphalytics are the data model, the workload selection process, and the resulting algorithms and datasets.

3.3.1. DATA MODEL

The Graphalytics benchmark uses a typical data model for graphs; a graph consists of a collection of vertices, each identified by a unique integer, and a collection of edges, each consisting of a pair of vertex identifiers. Graphalytics supports both *directed* and *undirected* graphs. Edges in directed graphs are identified by an ordered pair (i.e., the source and destination of the edge). Edges in undirected graphs consist of unordered pairs. Every edge must be unique and connect two distinct vertices. Optionally, vertices and edges have properties, such as timestamps, labels, or weights.

To accommodate requirement R2, Graphalytics does not impose any requirement on the semantics of graphs. That is, any dataset that can be represented as a graph can be used in the Graphalytics benchmark if it is representative of real-world graph-analysis workloads.

Table 3.1: Results of surveys of graph algorithms.

Graph	Class (selected candidates)	#	%
Unweighted	Statistics (PR, LCC)	24	17.0%
	Traversal (BFS)	69	48.9%
	Components (WCC, CDLP)	20	14.2%
	Graph Evolution	6	4.2%
	Other	22	15.6%
Weighted	Distances/Paths (SSSP)	17	34%
	Clustering	7	14%
	Partitioning	5	10%
	Routing	5	10%
	Other	16	32%

Table 3.2: Mapping of dataset scale ranges to labels ("T-shirt sizes") in Graphalytics.

Scale	< 7	$[7, 7.5)$	$[7.5, 8)$	$[8, 8.5)$	$[8.5, 9)$	$[9, 9.5)$	≥ 9.5
Label	2XS	XS	S	M	L	XL	2XL

3.3.2. Two-Stage Workload Selection Process

To achieve both workload representativeness and workload coverage, we used a *two-stage selection process* to select the workload for Graphalytics. The first stage identifies classes of algorithms and datasets that are representative for real-world usage of graph analysis platforms. 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.

3.3.3. Selected Algorithms

Addressing R1, according to which Graphalytics should allow different platforms to compete, the definition of the algorithms of Graphalytics is abstract. For each algorithm, we define its processing task and provide a reference implementation and reference output. Correctness of a platform implementation is defined as output equivalence to the provided reference implementation.

To select algorithms which cover real-world workloads for graph analysis platform, we have conducted two comprehensive surveys of graph analysis articles published in ten representative conferences on databases, high-performance computing, and distributed systems (e.g., VLDB, SIGMOD, SC, PPoPP). The first survey (conducted for our previous paper [25]) focused only on unweighted graphs and resulted in 124 articles. The second survey (conducted for this paper) focused only on weighted graphs and resulted in 44 articles. Table 3.1 summarizes the results from these surveys. Because one article may contain multiple algorithms, the number of algorithms exceeds the number of articles. In general, we found that a large variety of graph analysis algorithms are used in practice. We have categorized these algorithms into several classes, based on their functionality, and quantified their presence in literature.

Based on the results of these surveys, with expert advice from LDBC TUC we have selected the following five core algorithm for unweighted graphs, and a single core algorithm for weighted graphs, which we consider to be representative for graph analysis in general:

Breadth-first search (BFS): For every vertex, determines the minimum number of hops required to reach the vertex from a given source vertex.

PageRank (PR) [87]: Measures the rank ("popularity") of each vertex by propagating influence between vertices using edges.

Weakly connected components (WCC): Determines the weakly connected component each vertex belongs to.

Community detection using label propagation (CDLP): Finds "communities" in the graph, i.e., non-overlapping densely connected clusters that are weakly connected to each other. We select for community detection the label propagation algorithm [88], modified slightly to be both parallel and deterministic.

Table 3.3: Real-world datasets used by Graphalytics.

ID	**Name**	$\lvert V\rvert$	$\lvert E\rvert$	**Scale**	**Domain**
R1(2XS)	wiki-talk [89]	2.39 M	5.02 M	6.9	Knowledge
R2(XS)	kgs [90]	0.83 M	17.9 M	7.3	Gaming
R3(XS)	cit-patents [89]	3.77 M	16.5 M	7.3	Knowledge
R4(S)	dota-league [90]	0.06 M	50.9 M	7.7	Gaming
R5(XL)	com-friendster [89]	65.6 M	1.81 B	9.3	Social
R6(XL)	twitter_mpi [91]	52.6 M	1.97 B	9.3	Social

Table 3.4: Synthetic datasets used by Graphalytics.

ID	**Name**	$\lvert V\rvert$	$\lvert E\rvert$	**Scale**
D100(M)	datagen-100	1.67 M	102 M	8.0
D100’(M)	datagen-100-cc0.05	1.67 M	103 M	8.0
D100”(M)	datagen-100-cc0.15	1.67 M	103 M	8.0
D300(L)	datagen-300	4.35 M	304 M	8.5
D1000(XL)	datagen-1000	12.8 M	1.01 B	9.0
G22(S)	graph500-22	2.40 M	64.2 M	7.8
G23(M)	graph500-23	4.61 M	129 M	8.1
G24(M)	graph500-24	8.87 M	260 M	8.4
G25(L)	graph500-25	17.1 M	524 M	8.7
G26(XL)	graph500-26	32.8 M	1.05 B	9.0

Local clustering coefficient (LCC): Computes the degree of clustering for each vertex, i.e., the ratio between the number of triangles a vertex closes with its neighbors to the maximum number of triangles it could close.

Single-source shortest paths (SSSP): Determines the length of the shortest paths from a given source vertex to all other vertices in graphs with double-precision floating-point weights.

Appendix A lists the definition of each algorithm in the Graphalytics workload.

3.3.4. Selected Datasets

Graphalytics uses both graphs from real-world applications and synthetic graphs which are generated using data generators. Table 3.3 summarizes the six selected real-world graphs. By including real-world graphs from a variety of domains, Graphalytics covers users from different communities. Our two-stage selection process led to the inclusion of graphs from the knowledge, gaming, and social network domains. Within the selected domains, graphs were chosen for their variety in sizes, densities, and characteristics.

The real-world graphs in Graphalytics are complemented by two synthetic dataset generators, to enable performance comparison between different graph scales. The synthetic dataset generators are selected to cover two commonly used graphs: power-law graphs generated by *Graph500*, and social network graphs generated using *LDBC Datagen*. The graphs generated for the experiments are listed in Table 3.4.

To facilitate performance comparisons across datasets, we define the *scale* of a graph in Graphalytics as a function of the number of vertices ($\lvert V\rvert$) *and* the number of edges ($\lvert E\rvert$) in a graph: $s(V,E) = \log_{10}(\lvert V\rvert + \lvert E\rvert)$, rounded to one decimal place. To give its users an intuition of what the scale of a graph means in practice, Graphalytics groups dataset scales into *classes*. We group scales in classes spanning 0.5 *scale units*, e.g., graphs in scale from 7.0 to 7.5 belong to the same class. The classes are labelled according to the familiar system of “T-shirt sizes”: small (S), medium (M), and large (L), with extra (X) prepended to indicate smaller and larger classes to make extremes such as 2XS and 3XL possible.

The reference point is class L, which is intuitively defined by the Graphalytics team to be the largest class such that the BFS algorithm completes within an hour on any graph from that class in the Graphalytics benchmark using a state-of-the-art graph analysis platform on a single common-off-the-shelf machine. The resulting classes used by Graphalytics are summarized in Table 3.2.

3.4. Process

Addressing R3, the goal of the Graphalytics benchmark is to objectively compare different graph analysis platforms, facilitating the process of finding their strengths and weaknesses, and understanding how the performance of a platform is affected by aspects such as dataset, algorithm, and environment. To achieve this, the benchmark consists of a number of different *experiments*. In this section, we introduce these experiments, which we detail and conduct in Chapter 5.

The *baseline* experiments measure how well a platform performs for different workloads on a single machine. The core metric for measuring the performance of platforms is run-time. Graphalytics breaks down the total run-time into several components:

- **Upload time**: Time required to preprocess and convert the graph into a suitable format for a platform.
- **Makespan**: Time required to execute an algorithm, from the start of a job until termination.
- **Processing time** (T_{proc}): Time required to execute an actual algorithm as reported by the Graphalytics performance monitoring tool, Granula, or self-reported by the system under test. This does not include platform-specific overhead, such as allocating resources, loading the graph from the file system, or graph partitioning.

In our experiments we focus on T_{proc} as a primary indication of the performance of a platform. We complement this metric with two user-level throughput metrics:

- **Edges per second (EPS)**: Number of edges in a graph divided by T_{proc} in seconds. EPS is used in other benchmarks, such as Graph500.
- **Edges and vertices per second (EVPS)**: Number of edges plus number of vertices (i.e., 10^{scale}, see Section 3.3.4), divided by T_{proc} in seconds. EVPS is closely related to the scale of a graph, as defined by Graphalytics.

To investigate how well a platform performs when scaling the amount of available resources, the size of the input, or both, Graphalytics includes *scalability* experiments. We distinguish between two orthogonal types of scalability: strong vs. weak scalability, and horizontal vs. vertical scalability. The first category determines whether the size of the dataset is increased when increasing the amount of resources. For *strong scaling*, the dataset is kept constant, whereas for *weak scaling*, the dataset is scaled. The second category determine *how* the amount of resources is increased. For *horizontal scaling*, resources are added as additional computing machines, whereas for *vertical scaling* the added resources are cores within a single machine. Graphalytics expresses scalability using a single metric:

- **Speedup (S)**: The ratio between T_{proc} for scaled and baseline resources. We define the baseline for each platform and workload as the minimum amount of resources needed by the platform to successfully complete the workload.

Finally, Graphalytics assesses the *robustness* of graph analysis platforms using two metrics:

- **Stress-test limit**: The scale and label of the smallest dataset defined by Graphalytics that the system cannot process.
- **Performance variability**: The coefficient of variation (CV) of the processing time, i.e., the ratio between the standard deviation and the mean of the repeatedly measured performance. The main advantage of this metric is its independence of the scale of the results.

For all experiments, Graphalytics defines a service-level agreement (SLA): generate the output for a given algorithm and dataset with a makespan of up to 1 hour. A job breaks this SLA, and thus does not complete successfully, if its makespan exceeds 1 hour or if it crashes (e.g., due to insufficient resources).

After each job, its output is validated by comparing it against the reference output. The output does not have to be exactly identical, but is must be equivalent under an algorithm-specific comparison rule (for example, for PageRank we allow a 0.01% error).

3.5. Renewal Process

Addressing requirement R4, we include in Graphalytics a renewal process which leads to a new version of the benchmark every two years. This renewal process updates the workload of the benchmark to keep it relevant for increasingly powerful systems and developments in the graph analysis 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, we follow the same two-stage workload selection process as presented in Section 3.3.2.

The algorithms of Graphalytics have been selected based on their representativeness. However, over time, graph algorithms might lose or gain popularity in the community. For example, community detection is an active field of graph research nowadays, even though our label propagation algorithm [88] was only introduced less than a decade ago. To ensure that algorithms stay relevant, for every version of the benchmark, we will select a new set of core algorithms using the same process as presented in Section 3.3.3. We will perform a new comprehensive survey on graph analysis in practice to determine new algorithm classes and select new algorithms from these classes using expert advice from LDBC TUC. If a new algorithm is found to be relevant which was not part of the set of core algorithms, it will be added. If an older core algorithm is found to be no longer relevant, it is marked as obsolete and will be removed from the specification in the next version.

The datasets of Graphalytics have been selected based on their variety in size, domain, and characteristics [25, 89]. Using the same process as described for algorithms, the Graphalytics team will introduce additional real-world datasets and synthetic dataset generators as they become relevant to the community. This may include graphs from new application domains if they are not yet represented by similar graphs from other domains. In addition, with every new version of the specification the notion of a "large" graph is reevaluated. In particular, class L is redefined as the largest class of graphs such that at a state-of-the-art platform can complete the BFS algorithm within one hour on all graphs in class L using a single common-off-the-shelf machine. The selection of platforms used to determine class L is limited to platforms implementing Graphalytics that are available to the Graphalytics team when the new specification is formalized.

4

Grade10: Automated Bottleneck and Performance Issue Identification

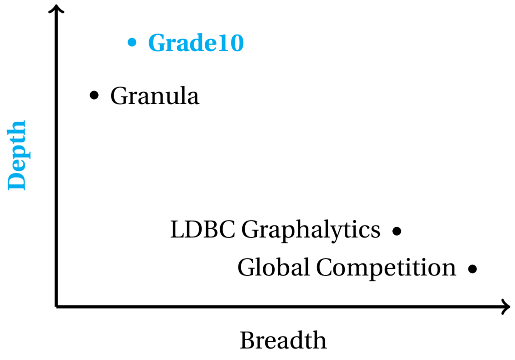


Figure 4.1: The position of Grade10 within the Graphalytics ecosystem.

We address in this chapter our second research question: How to automatically analyze performance in the execution of graph analytics workloads? Our approach is the design of *Grade10*, a system for analyzing in-depth the performance of individual graph analysis jobs. As its key conceptual contribution, Grade10 combines a model of a graph analysis job's execution with resource usage metrics to automatically identify bottlenecks and a variety of performance issues. The position of Grade10 within the Graphalytics ecosystem is depicted in Figure 4.1.

The remainder of this chapter is structured as follows. The requirements are laid out in Section 4.1. The architecture of Grade10 is described in Section 4.2. The data and information sources available to Grade10 are presented in Section 4.3. Finally, Section 4.4 describes Grade10's performance analysis process.

4.1. Requirements

In this section, we discuss the main requirements addressed by Grade10.

(R1) Identify performance bottlenecks: the system must be able to identify bottlenecks on a variety of physical and virtual resources. A performance bottleneck refers to a period of time during the execution of a system in which a particular resource (either hardware or software) is used to its capacity. The presence of a bottleneck implies that the performance of the system during that period of time is limited by the corresponding resource. Thus, identifying bottlenecks is key to understanding the performance of a system. Furthermore, bottlenecks should be determined for individual phases in the execution of a job (as defined by an expert), because bottlenecks are expected to vary throughout a job (e.g., a loading step may be disk-bound and processing may be CPU-bound). Without R1, Grade10 could not provide insight into the components and resources that limit a graph processing system's performance.

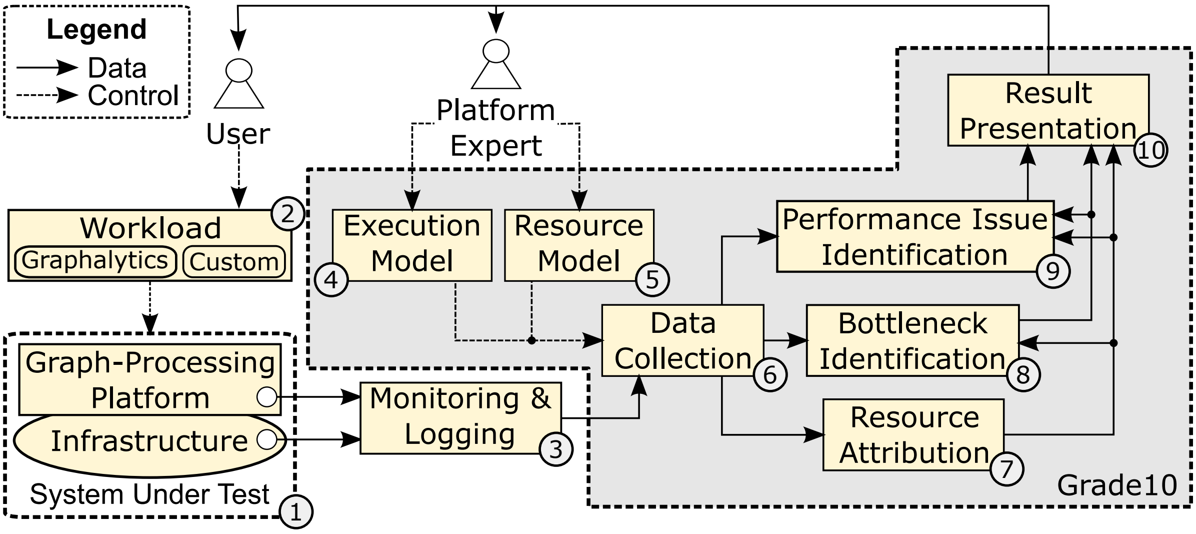


Figure 4.2: The Grade10 architecture. Grade10 takes as input performance metrics and system logs, interprets the input through an expert-defined execution and resource models, analyzes the results, and presents a set of identified performance bottlenecks and performance issues back to the user.

(R2) Identify performance issues: the system must be able to identify several classes of performance issues that typically occur in graph-processing systems. Performance bottlenecks are not a necessary, nor sufficient, condition for *performance issues* to occur. For example, in a well-optimized HPC application a high CPU load is an expected bottleneck and not necessarily indicative of a performance issue. However, in the same HPC application the lack of a bottleneck in a given process may indicate that the process is waiting for external input (e.g., waiting to receive data from other processes), which can be considered a performance issue. Other common performance issues may not be caused by bottlenecks at all (e.g., workload imbalance). Without R2, Grade10 could not provide insight into many factors that are known to cause poor performance in graph analysis systems.

(R3) Work platform- and workload-independently: the bottleneck and performance issue identification process should not be tailored to any specific graph analysis platform or workload. However, the focus of this work is on parallel and distributed graph analysis platforms. Without R3, Grade10 could not be used to study the performance of the large variety of available graph analysis systems.

(R4) Present results at variable levels of abstraction, for both experts and common users: the system should provide insight for both expert and non-expert users by presenting performance results at variable levels of abstraction. That is, non-expert users should be able to extract high-level observations on the system's performance without knowledge of the system's internals. Conversely, expert users should be able to analyze in-depth individual stages of execution, bottlenecks, etc. Without R4, Grade10 could not give both system developers and average users of graph analysis systems insight into performance.

(R5) Ensure a good trade-off between the depth of results and the time to get meaningful results, through an incremental process: the performance analysis process should facilitate an incremental workflow. A user should be able to derive high-level bottlenecks and performance issues with minimal effort and data collected. To obtain more in-depth results, an expert user should be able to extend the analysis process to focus on particular platform components of interest. Without R5, Grade10 could not be used without significant up-front investment in tuning the analysis process.

4.2. ARCHITECTURE

To address the requirements, our conceptual contribution is the design of an architecture for deep analysis of graph analysis platforms. Figure 4.2 depicts our Grade10 architecture, including the system under test and the Grade10 system. Table 4.1 summarizes the components of the Grade10 architecture, and lists the contribution of each component to the fulfillment of Grade10's requirements. In the remainder of this section, we present each component in more detail and discuss their individual contributions.

Table 4.1: Overview of the components in the Grade10 architecture (Figure 4.2) and their contribution to the fulfillment of Grade10's requirements (Section 4.1). Legend: Key, component is necessary for Grade10 to address a requirement; Helper, component is designed to contribute toward fulfilling a requirement, but is not considered (one of) the most important component(s); No, component does not specifically address a requirement.

Component	**R1**	**R2**	**R3**	**R4**	**R5**
(1) System Under Test	No	No	Key	No	No
(2) Workload	No	No	Key	No	No
(3) Monitoring & Logging	No	No	Key	No	No
(4) Execution Model	Key	Key	Key	Key	Key
(5) Resource Model	Key	Helper	Key	No	Key
(6) Data Collection	Helper	Helper	Helper	Helper	No
(7) Resource Attribution	Key	Helper	Helper	Key	No
(8) Bottleneck Identification	Key	Key	Helper	Key	No
(9) Performance Issue Identification	No	Key	Helper	Key	No
(10) Result Presentation	No	No	No	Key	Helper

As input to Grade10's performance analysis process, the user provides the system under test (SUT) and the workload it needs to process. The system under test (component 1 in Figure 4.2) encompasses all components in the typical architecture for graph analysis systems (Section 1.1), including graph-processing framework and the infrastructure it runs on. The workload (2) can either be provided by Graphalytics for a comprehensive analysis of typical graph-processing workloads, or, in general, be defined by the user to analyze a particular workload of interest. During workload execution, the SUT records performance data through monitoring and logging (3), pre-configured to deliver a good trade-off between sampling accuracy and volume of data collection. This requires instrumentation of both the graph-processing platform and the infrastructure, which is typically provided by the platform vendor. With the combined use of components $1-3$, Grade10 can obtain performance data for, and analyze the performance of, a wide range of workloads and platforms, satisfying R3.

In addition to monitoring and logging data, Grade10's performance analysis process requires two models of the system under test: an execution model and a resource model. The execution model (4) breaks down the execution of a workload into a set of execution phases, as described in Section 4.3.1. The resource model (5) describes the set of hardware and software resources that are monitored as the system under test is processing its workload. Section 4.3.2 provides a detailed definition of the resource model. Because both the execution and resource models embed much expert knowledge, Grade10 requires an expert of the graph-processing platform under test to develop or at least consult in the design process leading to these models. The explicit definition of execution and resource models is key to Grade10's ability to analyze performance at variable levels of granularity (R4), and to Grade10's ability to support an incremental workflow for performance analysis (R5). Furthermore, Grade10 can analyze the performance of many different classes of graph analysis platforms (R3), because, unlike existing performance analysis tools, it is not limited to supporting one (implicit) execution model.

The monitoring and logging data, the execution model, and the resource model are combined in Grade10's data collection component (6). As described in Section 4.4.1, this component combines performance data from the various inputs to create a structured representation for further analysis. This component is necessary in any approach that collects and analyzes data from multiple sources, e.g., in any performance analysis tool for distributed systems. In our approach, this component is challenging because it combines an abstract view of a job's execution (i.e., the execution and resource models) with concrete data collected during the job's execution (e.g., resource usage data).

Grade10 processes the collected performance data in three stages. First, the resource attribution component (7) *attributes* resource utilization data to execution phases as described in Section 4.4.2. The challenging resource attribution process is key to Grade10's ability to analyze the performance of individual execution phases at variable levels of granularity (R4). Existing approaches instead either collect resource utilization data per component of the SUT, which requires additional instrumentation and produces large volumes of data; use resource correlation instead of attribution, which can not reveal which components caused the observed resource utilization; or avoid the problem altogether by not providing performance analysis for individual low-level components. Next, the bottleneck identification component (8) analyzes every execution phase to determine which resources it bottlenecked on (R1), as described in Section 4.4.3. Finally, the performance issue identification component (9) is used to identify possible performance issues based on bottlenecks and other performance data (R2), as described in Section 4.4.4. While many existing performance analysis tools are designed to identify a mix of bottlenecks and performance issues, Grade10 proposes a conceptual distinction between the two types of performance characteristics. To this end, Grade10 proposes two systematic approaches, based on Grade10's hierarchical execution model, that can be tuned to identify many different bottlenecks and performance issues.

The bottlenecks and performance issues identified by Grade10 are fed into the final component for result presentation (10). This component is responsible for selecting a limited set of results, e.g., the most significant issues, as described in Section 4.4.5. The final results are presented to the user to inform them of any performance bottlenecks or issues that may have occurred during the execution of their workload. Additionally, the results may be used by a platform expert to refine the execution and resource models.

4.3. Input

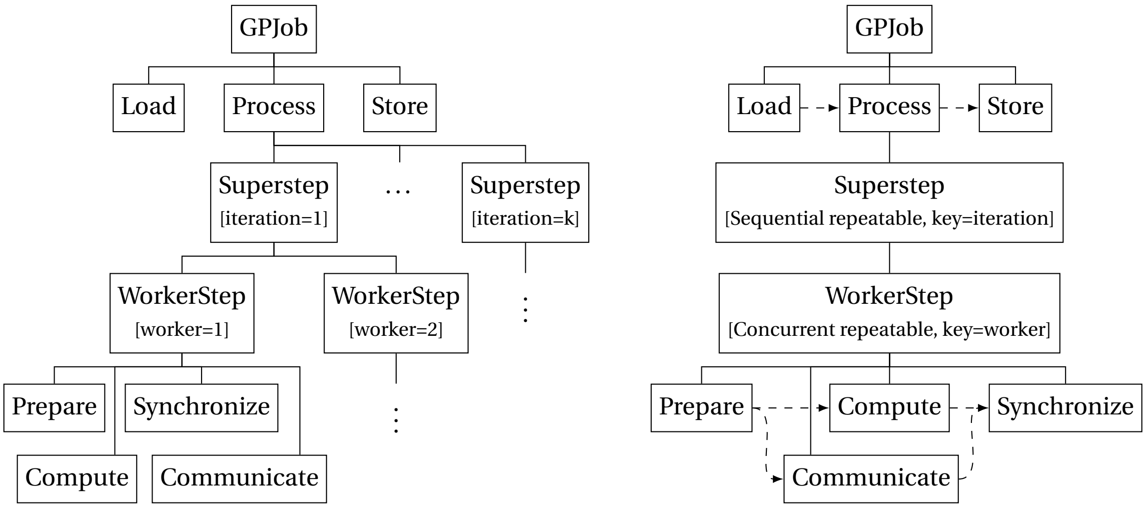
In this section, we describe the input to Grade10's performance analysis pipeline. The input includes an execution model describing the phases of a graph processing job (Section 4.3.1), a resource model describing the physical and virtual resources available to and used by the job (Section 4.3.2), and monitoring and logging data collected during the execution of the job (Section 4.3.3).

4.3.1. Execution Model

A key component of the Grade10 approach to bottleneck and performance issue identification is an *execution model* describing a job running on the graph-processing platform being analyzed. The purpose of the model is to provide a detailed description of the different phases of execution that constitute a job running on the platform. Without knowledge of the internals of a platform, performance analysis requires a black-box approach, i.e., observing performance metrics for the complete duration of a particular workload execution, and deriving possible bottlenecks from these observations. However, this does not take into consideration that different bottlenecks may occur at different levels — staggered, or worse, simultaneously — throughout a single execution, e.g., I/O can become the bottleneck when loading a graph and network can become the bottleneck when distributing messages in a distributed graph analysis system. By defining the phases of execution in a model, Grade10 is able to identify bottlenecks and performance issues in individual phases (satisfying R4). Furthermore, the use of a user-defined execution model allows Grade10 to analyze jobs for any platform with a well-defined model (R3).

A phase-based execution model matches well the typical design of a modern graph analysis system (Section 1.1). Many graph analysis systems use well-defined programming and execution models that naturally divide the execution of a job into phases, e.g., loading the input graph, and executing a user-defined algorithm iteratively (in distinct steps). Furthermore, most systems use a synchronous execution model which imposes hard barriers between different phases of execution.

Our approach is inspired by the Granula performance model [38], to whose development we have contributed. In Granula, a performance model consists of a hierarchy of operations and describes the execution of a specific job. Every operation is characterized by an actor performing a given mission, e.g., the actor "worker-1" performs "superstep-1". An operation can have multiple sub-operations to provide a more fine-grained definition of the process of achieving the parent operation's mission. Additionally, an operation has an arbitrary set of information associated with it to describe the operation's performance characteristics, typically including the start and end time of the operation. Grade10 extends this previous work in three non-trivial and impactful ways. First, Grade10 facilitates comparison across jobs by explicitly distinguishing between an execution model describing *a specific job* (Section 4.3.1.1) and an execution model specification describing *any job* (Section 4.3.1.2). This distinction also allows Grade10 to differentiate between phases of the same and of different types in its performance analysis. Second, Grade10 explicitly models *repeatability*, sequential or concurrent, to support automated analysis of performance issues concerning parallelism. For example, imbalanced execution times of parallel phases may be indicative of poor workload distribution across machines in a distributed graph analysis system, whereas imbalance across sequential iterations is typically a result of algorithm design (e.g., the amount of computation required per iteration is irregular, in particular for traversal algorithms). Third, Grade10 explicitly models precedence constraints, which enables detailed analysis of the combined impact of performance issues caused by different phases.



(a) Execution model for one job run on a generic graph-processing platform.

(b) Corresponding execution model specification for the platform.

Figure 4.3: Example of an execution model for an example job (a) and the corresponding execution model specification for graph-processing platform (b). Nodes represent phase types or phases in subfigure (a) or (b), respectively, with a name and additional properties in square brackets. Vertical connections represent a parent-child relation between the top and bottom phase (type). Horizontal dashed connections represent dependencies between two phase types and point from the dependency to the dependent phase type.

4.3.1.1. DEFINITION OF A GRADE10 EXECUTION MODEL

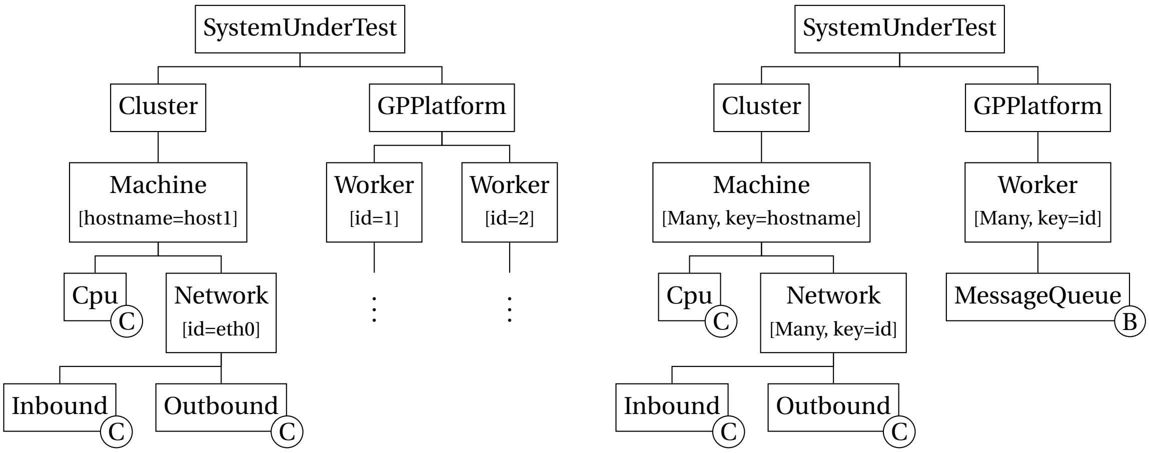
A Grade10 execution model comprises a hierarchy of execution *phases*. The root of the hierarchy is a single phase representing the entire execution of a job. To increase the granularity of Grade10's performance analysis, the root phase may be recursively split into *subphases*, through parent-child relationships. Each (sub)phase must have a unique name and is annotated with the start and end time of the phase's execution. The Grade10 execution model distinguishes between phases without children (*leaf phases*) and phases with at least one child (*composite phases*).

A core design principle for Grade10 execution models is the equivalence of a composite phase and its subphases. Each phase in a Grade10 execution model represents an individual, often conceptual, task in the completion of a job. In the design of a Grade10 execution model, a parent-child relationship indicates that the task represented by the parent phase can be decomposed into a set of subtasks, each represented by a subphase (child). Thus, the completion of all subphases with the same parent is equivalent to the completion of their parent phase. This equivalence ensures that an execution model can be extended at any time by splitting a phase into an equivalent set of subphases (R5), and it ensures, combined with further steps in the Grade10's performance analysis process, that subsequent analysis of the performance of the individual subphases results in meaningful conclusions about the performance of the parent phase (R4).

Figure 4.3a depicts an example of a Grade10 execution model for a generic graph-processing job. The *GPJob* phase is the root of the model and represents a single graph-processing job. The task of executing a graph-processing job can be split into three (conceptual) subtasks: (1) *loading* the input graph, (2) *processing* the graph using a user-defined algorithm, and (3) *storing* the result. Similarly, the *GPJob* phase in Figure 4.3a is split into three subphases: *Load*, *Process*, and *Store*. The *Process* phase may be the most interesting phase to a user trying to understand the performance of their graph analysis job, so we further split that phases into multiple *Superstep* subphases, one for each superstep (iteration) in the job. Any phase may be further split into subphases until the appropriate level of depth is reached (i.e., a trade-off is made between the effort of modeling and instrumenting the SUT, and the extra insight gained from higher-granularity performance analysis).

4.3.1.2. DEFINITION OF A GRADE10 EXECUTION MODEL SPECIFICATION

The workload of a graph analysis system is typically comprises multiple, seemingly independent, jobs. To facilitate analysis for the execution of a single job and comparison across multiple jobs, Grade10 uses an *execution model specification* of a platform to define the *types of phases* present in any job executed by the platform. Mirroring the organization of phases in a Grade10 execution model, phase types in an execution model specification are organized hierarchically. Conceptually, the execution model specification can be



(a) Resource model for one job run on a generic graph-processing platform.

(b) Corresponding resource model specification for the platform.

Figure 4.4: Example of a resource model for an example job (a) and the corresponding resource model specification for a generic graph-processing platform (b). Nodes represent resources in (a) and resource types in (b). A resource (type) with an associated metric (type) is marked with a “B” or “C” for a blocking or consumable metric, respectively.

seen as a blueprint for execution models; it specifies which (sub)phases can be present in an execution model, how many instances of the same type of phase can occur, in what order phases are executed, etc.

An execution model specification includes several elements that help expert users define this “blueprint”. Precedence constraints may be added between sibling phase types to indicate the execution order of phases with the same parent. Additionally, phase types (except the root) may be annotated to indicate that they are repeatable, i.e., they can be executed multiple times during one execution of their parent. These executions may be sequential with implicit precedence constraints between phases, or concurrent without any constraints.

Figure 4.3b depicts an example specification of an execution model, for some generic graph-processing platform. The specification dictates that any job on the modeled platform consists of a *GPJob* root phase. This root phase must have three subphases: *Load*, *Process*, and *Store*, which are executed in that order, as indicated by the precedence constraints (dashed connections). The *Process* phase may consist of any number of *Superstep* subphases executed in sequence, and each *Superstep* may consist of any number of *WorkerStep* executed concurrently. Finally, each *WorkerStep* consists of four subphases executed in three stages, as dictated by the precedence constraints: (1) *Prepare*, (2) *Compute* and *Communicate* concurrently, and (3) *Synchronize*. The execution model depicted in Figure 4.3a matches our example specification, so that execution model is considered to be a valid *instance* of the specification depicted in Figure 4.3b.

It is rarely the case that all the possible uses of a model can be anticipated. Some models need refinement to achieve better accuracy. Others are pruned to prevent too large volumes of data acquired to calibrate the model. Yet other models are developed concurrently, often with small variations, because none of the individual models can be generalized to encompass the others. Such problems may be alleviated with the features provided by Grade10. In particular, the specification of an execution model may be refined by an expert over time, by adding subphase types to model a particular phase type of interest in more detail, thus leading to an incremental process (R5).

4.3.2. Resource Model

To organize resource usage data and attribute it to execution phases, Grade10 proposes a resource model and corresponding specification that describe the set of monitored resources and their associated metrics. In particular, Grade10 defines the *resource model specification* as a tree of *resource types*. Each resource type may have one associated *metric type*, which describes how the usage of a resource of the given type is measured (e.g., the unit of measurement, the resource class as defined in Section 4.3.2.1, a description to be displayed to users). Similarly to an execution model specification, a resource model specification can easily be extended to include new metric or resource types to deepen the analysis (R5).

An example of a resource model specification is depicted in Figure 4.4b. The root of the resource model specification represents the system-under-test, including both hardware and software components. Child nodes represent resource types that are a logical subset of their parent. In the example, the system-under-test includes hardware (a *Cluster*) and software (a *GPPlatform*). The *Cluster* consists of zero or more *Machines*, each equipped with a *CPU* and zero or more *Network* resources (i.e., network interfaces). The *GPPlatform* consists of zero or more *Workers*, each with access to a software-defined *MessageQueue*. The *resources* and *metrics* monitored for a single job are represented in a *resource model*, as depicted in Figure 4.4a.

The resource model introduced by Grade10 is flexible, in that it can support many different resource classes and types. We now detail the resource classes (Section 4.3.2.1) and resource types (Section 4.3.2.2) currently considered in Grade10, in turn.

4.3.2.1. Resource Classes

Grade10 supports two classes of resources and associated metrics. A *consumable resource* is a resource that can be utilized over time to perform a task. For example, the usage of a network interface (resource) may be measured by the volume of incoming traffic per second (metric). Conversely, a *blocking resource* prevents a process from performing its task if the resource is not available. For example, the usage of a lock (resource) may be measured by the time it is held or the time processes spend waiting for the lock to become available (metric).

Consumable metrics[1] are characterized by measuring units (e.g., cycles, bytes) that become available over time and can be *consumed* to perform a task. They have a limited capacity during a unit of time (i.e., a maximum rate of consumption), but the total capacity over time scales linearly in the amount of time, without limitations. For example, the cycles of a CPU are consumable; CPU cycles can be used at a limited rate, the CPU's clock rate, to perform computation. Other examples of consumable metrics include inbound/outbound network bandwidth or disk reads/writes expressed in bytes per second. An assumption made by Grade10 is that consumable metrics are linear, i.e., doubling the rate of consumption for performing a task should halve the duration of the task, provided the task does not require other resources. Similarly, if two concurrent processes each have a non-zero rate of consumption, their total rate of consumption should equal the sum of their individual rates.

Blocking metrics[1] describe resources that a process is either waiting for or not at any given moment. While waiting for a blocking resource, a process does not consume any consumable resources and thus cannot complete its task. For example, while a process is waiting to acquire a lock, it cannot continue. The corresponding metric is a binary value indicating whether a resource is blocked or in use during a given period of time.

The restriction to consumable and blocking resources facilitates the identification of performance bottlenecks. Anytime a consumable resource is utilized to its capacity, the process utilizing this resource is bottlenecked; the process can only be sped up if the capacity of the resource is increased (equal consumption at a higher rate) or if the demand for the resource is increased (lower consumption at an equal rate). Likewise, if a process is spending all of its time waiting for a blocking resource, this resource is a bottleneck; the process can only be sped up by reducing the amount of time spent waiting.

4.3.2.2. Example Resource Types

Although Grade10's performance analysis focuses on two classes of resources, consumable and blocking, many resources can be modeled to fit this restriction and to enable Grade10 to identify them as bottlenecks in the system-under-test. Examples of common resources that can be either directly represented or (partially) modeled as a consumable or blocking resource include:

CPU, disk, network utilization: These resources can be directly represented as consumable resources.

Memory utilization: Unlike other hardware resources that are typically monitored by performance engineers (e.g., CPU, network), memory is not considered a consumable resource. Memory utilization does not represent a rate of consumption, but rather how much memory is in use at a given moment. However, there are multiple sources of bottlenecks in memory management which can still be analyzed by Grade10. For example, high memory utilization may manifest in a process blocking on garbage collection activity, on frequent swapping events, or on slow memory allocations. The waiting time for such resources can be represented as a blocking resource, which may be identified as the bottleneck of a process.

[1] The terms "consumable metric" and "blocking metric" refer to metrics describing the usage of a consumable or blocking resource, respectively. They are not meant to imply that the metrics themselves are consumable or blocking.

Locks: Locks translate naturally to a blocking resource: a process is either waiting for a lock or not. However, this simplification prevents the detection of some performance issues caused by locks that can be identified with specialized tools. For example, Grade10 does not consider which process has acquired a lock to identify the source of starvation. Despite the lack of specialized analysis, Grade10's approach to analyzing blocking resources may still be valuable to users, because (the absence of) a significant bottleneck on a lock may guide the decision to invest time in more detailed analysis.

Queues: Queues can be represented as a pair of blocking resources: one for consumers that are waiting for an empty queue to start filling up, and one for producers that are waiting for a full queue to clear. Similar to locks, this simplification results in a loss of information, such as the depth or throughput of the queue, or the time individual elements spend in the queue. However, by using Grade10 to analyze bottlenecks caused by waiting on a full or empty queue, the user can decide if further analysis of the queue's usage might result in a significant performance improvement.

Grade10 already supports many of the important resources and metrics that have appeared in our multi-year discussion with the community about the performance of graph analysis systems. We have not yet investigated how Grade10 can be used with low-level hardware performance counters to study the impact of cache misses, branch mispredictions, etc. These counters are primarily used to analyze the performance of systems that are already well-optimized.

4.3.3. Monitoring and Logging

To analyze the performance of a graph-processing platform executing a workload, information must be gathered from various sources during and about the execution. In particular, for every metric in the job's resource model (Section 4.3.2) measurements of the resource's usage over time must be collected, and for every execution phase in the job's execution model (Section 4.3.1) the start and end time must be collected. Although Grade10 does not impose any restrictions on the methods used for collecting this data, a typical collection process combines monitoring, e.g., periodic measurement of the usage of a resource, and logging, e.g., storing the start and end time of an execution phase as it completes.

4.4. The Grade10 Process and Key Algorithms

In this section, we describe Grade10's internal performance analysis processes: data collection (Section 4.4.1), resource attribution (Section 4.4.2), bottleneck identification (Section 4.4.3), performance issue identification (Section 4.4.4), and result presentation (Section 4.4.5).

4.4.1. Data Collection and Representation

The first step in Grade10's performance analysis pipeline is data collection. Grade10 collects information about a job's execution from various sources and combines this information to form a unified, comprehensive dataset detailing the execution of a job. Relevant information includes the execution and resource model specifications for the platform on which the job ran, logs listing the start and end time of each execution phase, and usage data for each metric.

Many aspects of the data collection process are technical or have platform-specific solutions, e.g., how to parse logging and monitoring data?, how to derive a hierarchical execution model from "flat" logs? Although we necessarily address such challenges to make Grade10 a useful instrument, we focus in the remainder of this section on conceptual challenges. We address two conceptual challenges: how to combine timestamped data with different sources or precision, and how to unify the execution and resource model?

4.4.1.1. Combining Timestamped Data

Data describing the execution of a job is frequently timestamped. Timestamps mark the start or end of a phase, the moment the usage of a resource was measured, etc. To simplify analysis of timestamped performance data, Grade10 assumes that time is discrete, that is, the execution time of a job is divided into a number of *time slices* of fixed duration. The state of the SUT is assumed to be static during a time slice; for the duration of a given time slice each execution phase is either active or not, and resource usage is constant. Under this assumption, the system's state may transition, e.g., a phase may start or end, or resource usage may change, only between two time slices. Tweaking the duration of a time slice enables users to make a trade-off between the granularity of Grade10's analysis and the CPU and memory requirements of Grade10.

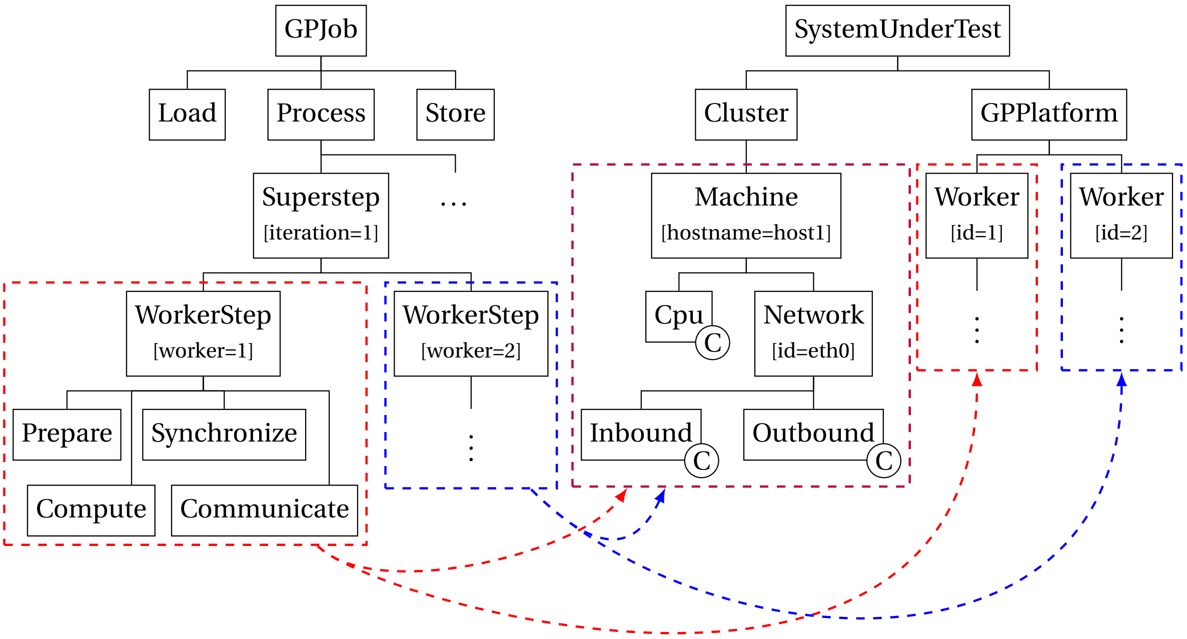


Figure 4.5: An example mapping from the execution model depicted in Figure 4.3a to the resource model depicted in Figure 4.4a.

Further, to facilitate analysis of performance data from multiple sources, Grade10 assumes that all timestamps attached to performance data are derived from a shared clock. In practice, this assumption rarely holds for distributed (graph analysis) systems. Each machine in such a system has its own clock and these clocks are typically synchronized to within a standardized error bound. However, the impact of this assumption is limited to analysis steps that directly compare timestamped data across machine boundaries. For example, when attributing resources that are shared between machines (does not occur in our experiments) or when aggregating bottlenecks identified on different machines (does occur in our experiments). Overall, we believe the impact of this assumption to be minor; new approaches at Google and in high frequency trading [92, 93] emphasize that the accuracy of this synchronization can improve beyond a thousandth (1/1,000) of a second, and well toward a millionth (1/1,000,000) of a second.

4.4.1.2. MAPPING EXECUTION PHASES TO RESOURCES

In a distributed system, not all processes have access to the same resources. For example, a process running on one machine can not utilize the CPU of another machine. In Grade10's execution and resource models, this translates to certain execution phases (e.g., phases representing individual processes) having access to a subset of the resources described in the resource model (e.g., the physical resources available on one machine). To represent this restriction, Grade10 creates a mapping from the execution model to the resource model to indicate for every phase in the execution model which resources it can use.

To perform this step, Grade10 requires a user of the system to select a function which assigns each execution phase to the one or more resources it had access to during its execution. Similar to execution and resource model specifications, the mapping function is typically written once by an expert user (i.e., a developer of the SUT) and provided to users. Mapping an execution phase to a set of resources indicates that for the execution of that phase no resources were used outside the specified set. The restriction of an execution phase to a set of resources also applies to all subphases; any phase has access to a subset of the resources its parent has access to. An example mapping from execution phases to resources is depicted in Figure 4.5. The left and right tree in the figure represent an execution model and resource model, respectively. Two *WorkerStep* phases are each mapped to two groups of resources: the resource for the physical machine on which the corresponding worker was executed, the resource representing the worker itself, and the subresources of both.

4.4.2. Resource Attribution

To determine bottlenecks in individual execution phases, Grade10 requires knowledge of the resources being utilized by each phase at any given point in time during a job's execution. This is trivial to determine if only one phase is active at a time; in this situation, the active phase must be responsible for all resource usage observed at that time. However, in the presence of concurrent phases, which is common practice for graph analysis systems, it may not be apparent how much of the total usage of a resource was caused by each phase. Similarly, if resource usage data is captured at low granularity, multiple phases may be active during a single measurement interval, making it unclear how much each phase contributed to the total observed resource usage.

To address this complex problem, Grade10 uses a process of *resource attribution* to determine the contribution of each phase to the usage of each resource. The resource attribution process uses as inputs a job's execution model, resource model, and phase-resource mapping. It is guided by attribution rules selected by the user. The process consists of two steps to attribute blocking and consumable resources, respectively.

Throughout the process, Grade10 assumes that only leaf phases use resources. The resource usage of a composite phase is defined as the sum of the resource usage of each of its subphases. This matches the design of a Grade10 execution model; a composite phases is split into a set of subphases that, when combined, represent the same task in the execution of a graph analysis job. If the resource attribution process included both composite and leaf phases, all resource usage observed while a leaf phase was executing would be attributed to the leaf phase *and* its parent, thus attributing usage multiple times for the execution of the same task.

4.4.2.1. Attribution Rules

Grade10 uses attribution rules, which are based on simple logical filters and operators, to allow users to configure the attribution process. By default, Grade10 assumes every phase with access to some resource, as defined by the phase-resource mapping (see Section 4.4.1.2), uses an equal share of the resource. In practice, this assumption does not hold; if a compute-intensive phase and a communication-intensive phase occur simultaneously, it is likely that the former contributes significantly more to any observed CPU usage. Similarly, the compute-intensive phase may not use any network or storage resources, even though it has access to them. Such platform-specific knowledge of the (expected) resource usage of each phase is encoded in Grade10 using attribution rules. An attribution rule must be provided for every combination of a phase type and a resource type.

Grade10 uses different sets of rules for blocking resources vs. consumable resources. For *blocking* resources Grade10 currently supports two kinds of attribution rules:

- **None:** The None rule indicates that a phase does not block on a resource type, even if it has access to resources of the given type.
- **Full (default):** The Full rule indicates that a phase blocks on any resource of the given type it has access to.

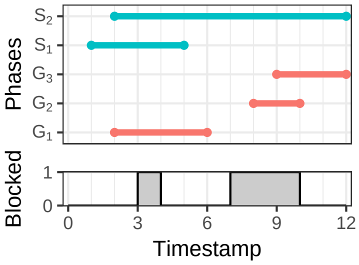
Grade10 also currently supports three kinds of attribution rules for *consumable* resources:

- **None:** The None rule assigns none of the usage of the given resource to the given execution phase. This rule may be used when the phase is known to not use the given type of resource.
- **Greedy:** The Greedy rule is used for phases that are expected to use a resource intensively. The Greedy rule has one parameter, the *cap*, that sets a limit on how much of a resource the phase can use. Grade10 divides the observed usage of a resource at any given moment over eligible phases with the Greedy rule proportional to their cap.
- **Sink (default):** The Sink rule is used for phases that use a given resource but do not fit the Greedy rule. If the usage of a resource can not be fully attributed to Greedy phases, the remainder is divided equally over all eligible phases with the Sink rule.

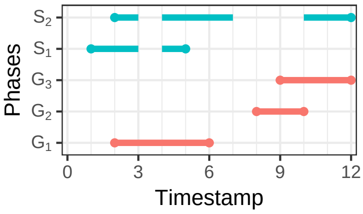
4.4.2.2. Step 1: Attributing Blocking Resources

Blocking resources, such as locks and queues, are frequently causes of transient bottlenecks. They are also frequently residing in software, and thus not easy to observe through typical resource monitoring tools. Thus, they pose important challenges to understanding system bottlenecks.

The first step of the resource attribution process is attributing the usage of blocking resources to individual execution phases. The usage of a blocking resource is defined by a binary metric; at any point in time the resource is either blocked or available. Similarly, a graph analysis phase is either waiting for a blocking resource, and in this case the blocking resource acts as a completely blocked bottleneck, or it is not, and in this case the blocking resource does not act as a bottleneck.



Phase	Rule
S_1	Full
S_2	Full
G_1	None
G_2	None
G_3	None

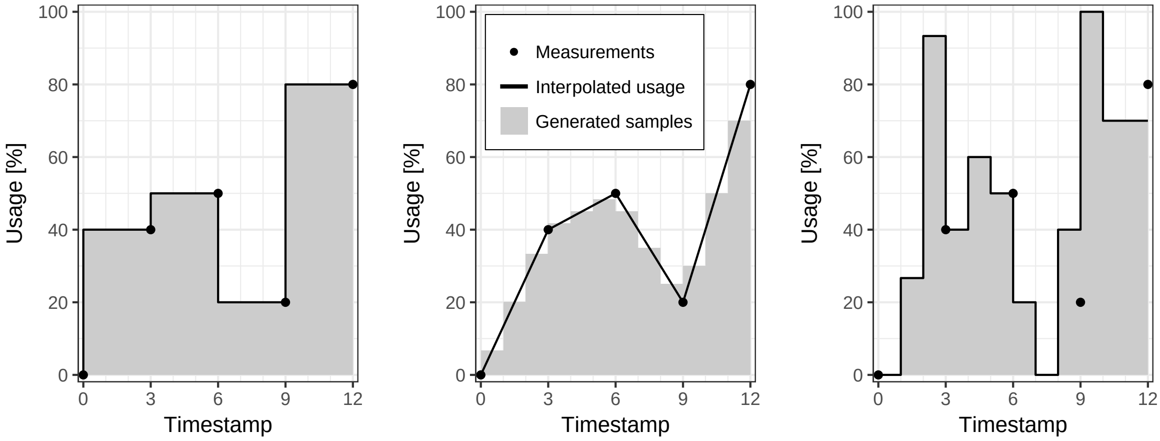


(a) Input: (top) phase lifetimes and (bottom) blocking resource, as binary data

(b) Configuration: attribution rules, as phase-rule pairs.

(c) Output: active phase periods.

Figure 4.6: Example application of two key steps in the Grade10 resource attribution process: the blocking resource attribution and the active phase detection.



(a) Traditional sampling method: constant interpolation using a left-continuous step function.

(b) Traditional sampling method: linear interpolation.

(c) Grade10 sampling method: phase-aware sampling, with the process depicted in Figure 4.8.

Figure 4.7: Three sampling methods applied to a constructed example of resource usage. The input data consists of five measurements taken every third time slice (i.e., each three units of time apart from the previous). The dots indicate the input data. The continuous curves provide a bound for the values considered by the sampling method. The shaded areas indicates for every time slice the values actually generated by the sampling method.

Grade10 uses a straightforward algorithm for the attribution of blocking resource usage to phases: for every time slice during a phase's execution, the phase is waiting whenever a triplet of conditions are met: (1) a blocking resource is blocked, (2) the phase has access to the given resource, and (3) the Full attribution rule is selected for the given phase-resource combination.

4.4.2.3. STEP 2.1: IDENTIFYING ACTIVE EXECUTION PHASES

The first step of the resource attribution process for consumable resources is identifying the active execution-phases. That is, the first step determines, for every time slice during a job's execution, which phases were active and could be considered as consuming resources. Grade10 uses two criteria to determine when a phase is active: (1) a phase can only use resources after it starts and before it ends, and (2) a phase can only use consumable resources when it is not waiting on a blocking resource.

The set of active phases during a given time slice is used in the second and third steps of the resource attribution process, to determine which phases could be responsible for the observed usage of a resource. Figure 4.6 depicts an example of the combined blocking resource attribution and active phase detection steps. Figure 4.6a depicts at the top the lifetimes of each of the 5 phases included in the example, some of which are concurrent. At the bottom, the figure depicts the usage data of a blocking resource. The configuration of the Grade10 resource attribution process, that is, the attribution rule selected for each phase, is presented in Figure 4.6b. Finally, Figure 4.6c depicts the result of attributing the blocking resource to the phases in the

Algorithm 1 Phase-aware upsampling method for resource usage metrics.

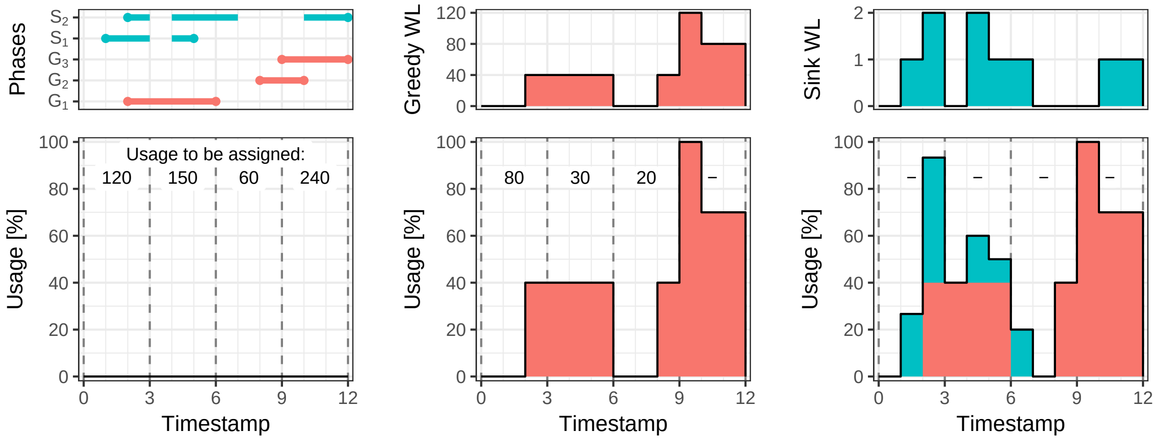
```
Input:
    $T$                ▷ Set of time slices in measurement period
    $u$                ▷ Measured average resource usage
    $C$                ▷ Capacity of resource
    $L_G[t]$           ▷ Total capacity of active Greedy phases during time slice $t$
    $L_S[t]$           ▷ Total number of active Sink phases during time slice $t$
Output:
    $S[t]$             ▷ Sampled resource utilization for time slice $t$

function UPSAMPLE($T$, $u$, $C$, $L_G$, $L_S$)
    $R \leftarrow u \cdot |T|$
    $C_G[t] \leftarrow \min\{L_G[t], C\}$ for all $t \in T$
    $S_G$, $R \leftarrow$ GENERATESAMPLES$(T, R, L_G, C_G)$          ▷ Assign first to Greedy phases
    $C_S[t] \leftarrow C - S_G[t]$ for all $t \in T$
    $S_S$, $R \leftarrow$ GENERATESAMPLES$(T, R, L_S, C_S)$          ▷ Assign remainder to Sink phases
    $C_B[t] \leftarrow C - S_G[t] - S_S[t]$ for all $t \in T$
    $S_B$, $R \leftarrow$ GENERATESAMPLES$(T, R, \mathbf{1}, C_B)$   ▷ Assign remainder to background noise
    return $S_G + S_S + S_B$
end function

function GENERATESAMPLES($T$, $R$, $L$, $C$)
    $R' \leftarrow R$
    $L_R \leftarrow \sum_{t \in T, C[t]>0} L[t]$          ▷ $L_R$: Workload of remaining time slices with non-zero capacity
    $S[t] \leftarrow 0$ for all $t \in T$
    for $t \in \{t' \in T | L[t] > 0, C[t] > 0\}$ sorted descending by $\frac{L[t]}{C[t]}$ do
        $S[t] \leftarrow \min\left\{R' \cdot \frac{L[t]}{L_R}, C[t]\right\}$
        $R' \leftarrow R' - S[t]$
        $L_R \leftarrow L_R - L[t]$
    end for
    return $S$, $R'$
end function
```

example. We note that phases G$_1$-G$_3$, for which the None rule was selected, are marked in the output as active for their respective lifetimes, that is, the phases are never blocked on the resource in this example. Phase S$_1$ and S$_2$ are active during their respective lifetimes, except during the time periods between $3-4$ and between $7-10$, because the Full rule was selected for them. We finally note that the presence of concurrent phases has no bearing on the attribution of a blocking resource; at $t = 3-4$, two phases are using the same resource while it is blocked, so both phases are waiting for the resource to become available, regardless of how many other phases are using the resource.

4.4.2.4. Step 2.2: Sampling Consumable Resource Usage Metrics

Grade10 requires the total resource usage per time slice as input to the final step in its attribution process. The duration of a time slice may be much shorter than the time between two measurements of a resource's usage (e.g., 1 millisecond vs. 50 milliseconds in our prototype). To convert the input metrics of variable frequencies to the fixed frequency of one value per time slice, Grade10 uses sampling. We focus in particular on *upsampling*, i.e., sampling input metrics with measurement intervals longer than one time slice.

Upsampling is a process used primarily in digital signal processing, to increase the sampling rate of a signal. Increasing the sampling rate is typically done through interpolation, i.e., generating new samples between existing samples based on the values of surrounding samples. Simple interpolation methods include constant interpolation (i.e., every new sample is equal to the next/previous/nearest existing sample) and linear interpolation (i.e., every sample lies on the line segment between the next and previous existing sample in a time-value graph). An example of applying these methods is depicted in Figure 4.7. In practice, digital



(a) Input to the phase-aware sampling method.

(b) Generation of initial samples based on Greedy workload.

(c) Generation of final samples based on Sink workload.

Figure 4.8: Step-by-step application of the Grade10 phase-aware sampling method on the resource usage data presented in Figure 4.7. (a), Top: Execution phases (rows) and their active periods (line segments). Gaps signify periods of inactivity (i.e., a phase is blocked). Colors distinguish between Greedy (G_1-G_3) and Sink (S_1, S_2) phases. (b), Top: Greedy workload over time, equal to capacity in %. The capacities of G_1-G_3 are 40, 40, and 80, respectively. (c), Top right: Sink workload over time, equal to number of active sink phases. (a)-(c), Bottom: State of samples and remaining usage to be assigned, in % times number of time slices, per measurement period of three time slices. In the end, plot (c), Bottom depicts a situation where all the resource usage has been attributed, and thus explained by Grade10.

signal processing applications use a variety of more sophisticated interpolation methods. Grade10 supports constant interpolation for upsampling, but also includes a novel sampling method: *phase-aware sampling* (detailed in the next section).

4.4.2.5. PHASE-AWARE SAMPLING METHOD

Grade10's resource attribution process operates under the premise that observed resource usage is caused by specific phases of a running job. (Grade10 does not consider background or interfering or otherwise external resource usage.) The phase-aware sampling method of Grade10 uses knowledge of the active phases at any given point in a job's execution to upsample data about resource usage. In particular, the method estimates the job workload based on the Greedy and Sink phases active during each time slice. The total observed resource consumption during an observation period (i.e., the measured average usage times the duration of the period) is then divided over each time slice, proportional to estimated the workload.

The *phase-aware sampling algorithm* (Algorithm 1) is applied independently to every measurement period of every resource. Included in the algorithm's input are the Greedy and Sink workload during each time slice, defined as the total capacity of active Greedy phases and the number of active Sink phases, respectively. Intuitively, the algorithm assigns resource usage first proportional to the Greedy workload, and second, if any resource usage remains, equally to Sink phases. This definition derives naturally from the definition of Greedy and Sink phases (Section 4.4.2.1).

We present Algorithm 1 step-by-step: The algorithm first (lines 2-4) divides the total measured resource usage over individual time slices proportional to the Greedy workload. Next (lines 5-6), any remaining usage is divided proportional to the Sink workload. Finally (lines 7-8), the last remaining usage is divided equally over all time slices, based on the assumption that a constant background workload is present. This ensures that the total of the generated samples is equal to the observed resource usage. Each of these steps uses a greedy algorithm (lines 11-21) which processes time slices in descending order of load to capacity ratio (line 15). This ensures that all time slices are assigned an amount of usage proportional to the workload, unless this exceeds the capacity for that time slice.

The runtime complexity of Algorithm 1 is dominated by the sorting operation in GENERATESAMPLES (line 15) which takes $O(T_M \times \log(T_M))$ time, where T_M is the number of time slices per measurement period. For the analysis of a job, this algorithm is repeated for every resource and every measurement period. Thus, the total runtime is $O(R \times M \times T_M \times \log(T_M))$, where R is the number of resources and M is the number of measurement periods. This can be further simplified to $O(R \times T_J \times \log(T_M))$, where T_J is the number of time slices in the job's duration. In comparison, the traditional sampling methods presented in Figure 4.7 run in

Algorithm 2 Resource attribution algorithm applied per metric and time slice.

```
Input:
    $s$                         ▷ Sampled resource usage
    $c$                         ▷ Capacity of resource
    $P_G$                       ▷ Set of active Greedy phases
    $P_S$                       ▷ Set of active Sink phases
    $G_p$                       ▷ Cap of Greedy rule for $p \in P_G$
Output:
    $A_p$                       ▷ Attributed usage of $p \in P_G \cup P_S$
    $M_p$                       ▷ Available capacity (max. usage) of $p \in P_G \cup P_S$

function AttributeSample($s, c, P_G, P_S, G_p$)
    $G \leftarrow \Sigma_{p \in P_G} G_p$
    for $p \in P_G$ do
        $A_p \leftarrow \min\{sG_p/G, G_p\}$
        $M_p \leftarrow \min\{cG_p/G, G_p\}$
    end for
    $s' \leftarrow s - \Sigma_{p \in P_G} A_p$
    $c' \leftarrow s - \Sigma_{p \in P_G} M_p$
    $S \leftarrow |P_S|$
    for $p \in P_S$ do
        $A_p \leftarrow s'/S$
        $M_p \leftarrow c'/S$
    end for
    return $A_p, M_p$ for all $p \in P_G \cup P_S$
end function
```

$O(R \times T_J)$ time. We analyze the accuracy of each method, which provides the other side of the trade-off that users of Grade10 must understand, in Section 5.5.1.3 (in particular, Table 5.11).

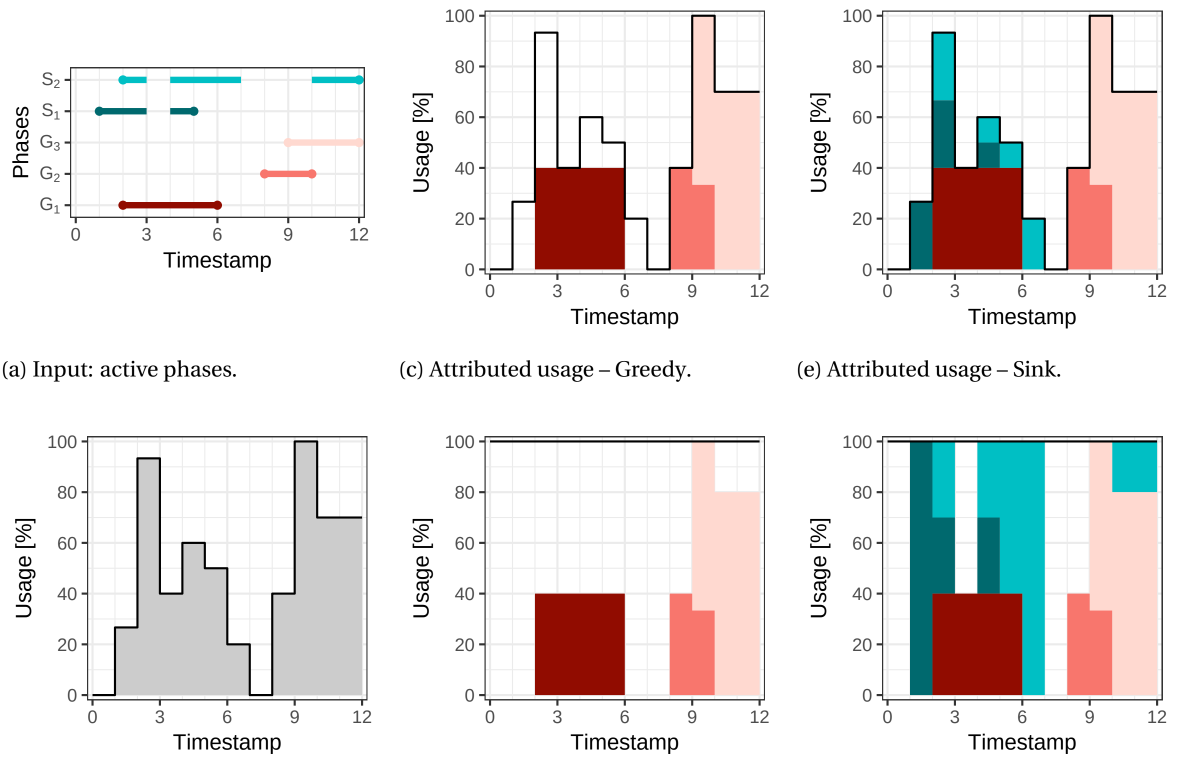
We present an exemplary application of the phase-aware sampling method in Figure 4.8. First, Figure 4.8a depicts the initial state of the algorithm, including the initial values of R and S for each measurement period (see Algorithm 1). Next, Figure 4.8b depicts the Greedy workload (L_G), the corresponding samples (S_G), and the remaining usage (R). During the first three measurement periods, all generated samples equal the capacity of active Greedy phases. During the last measurement period, the observed usage (240%, depicted in Figure 4.8a, Bottom) is less than the capacity of Greedy phases ($280\% = 120\% + 80\% + 80\%$, the sum of the workload during the last measurement period, depicted in Figure 4.8b, Top), so the samples have smaller values. The first sample of the last period is further limited by the capacity of the resource to 100%. Finally, Figure 4.8c depicts the Sink workload (L_S), the corresponding samples (S_S), and no remaining usage. Within each measurement period the produced samples are proportional to the workload, but the sampled usage per phase is different across periods, e.g., 26.7% per phase in the first period, and 10% per phase in the second period. Zero-valued samples are produced for time slices during which the Sink phases were blocked (e.g., at timestamp $3-4$) or the remaining usage was zero (e.g., at timestamp $10-12$). Because all remaining samples are equal to zero, the background samples are also equal to zero and their computation is skipped. The sums of the Greedy and Sink samples are returned as a single sample per time slice, as depicted in Figure 4.7c.

By design, the phase-aware sampling method has three desirable properties.

1. It is consistent with the measured usage. The average of the samples produced for a measurement periods is equal to the measured average usage.

2. It is consistent with the platform workload. The produced samples are proportional to Grade10's estimation of the workload, except where limited by the capacity of the resource.

3. It is consistent with the resource attribution rules. Samples are produced for Greedy phases first and for Sink phases only if the measured usage exceeds the total capacity of Greedy phases during a measurement period.



(a) Input: active phases.

(c) Attributed usage – Greedy.

(e) Attributed usage – Sink.

(b) Input: resource usage samples.

(d) Available capacity – Greedy.

(f) Available capacity – Sink.

Figure 4.9: Step-by-step application of the Grade10 resource attribution step to the active phases derived in Figure 4.6 and with the resource-usage samples derived in Figure 4.8.

4.4.2.6. Step 2.3: Attributing Consumable Resources

The final step of Grade10's resource attribution process is attributing sampled resource-usage data to active phases. For each resource and time slice, Grade10 divides the sampled usage over the active Greedy and Sink phases, using Algorithm 2. First (lines 2-4), all Greedy phases are assigned a fraction of the sample proportional and up to their capacity. The (lines 7-11), Sink phases are each assigned an equal fraction of the remaining sample. Through lines 5 and 12, Grade10 also attributes to each phase an *available capacity* for the resource to indicate how much of the resource a phase *could have used*. The available capacity is computed by attributing resource usage samples equal to the capacity of the resource.

The runtime of Algorithm 2 is $O(P)$, where P is the maximum number of phases using a given resource at any point in time. The total runtime of the final resource attribution step for a job is $O(R \times T_J \times P)$, where R is the number of resources and T_J is job's duration in time slices.

Figure 4.9 demonstrates the final resource attribution step for the active phases and sampled resource usage presented in Figures 4.6 and 4.8, respectively. The example includes several combinations of phases, attribution rules, and sampled usage that highlight properties of the attribution process. First, Figures 4.9c and 4.9d depict the attributed usage and available capacity of Greedy phases, respectively. Timestamps 2 – 6 and 8 – 9 present a typical situation: the attributed usage and available capacity are equal to the capacity of each phase. At timestamp 9 – 10, the attributed usage and available capacity are equal, but both are smaller than the capacity of each phase as defined by their respective Greedy rules. That is, phase G_2 can use up to 40% of the resource, but due to competing demands of phase G_3 only 33.3% is available to (and used by) G_2. Conversely, at timestamp 10 – 12, the attributed usage (70%) is limited by the sampled usage instead of the capacity (80%). Next, Figures 4.9e and 4.9f depict the final output after processing Sink phases. The total available capacity is always equal to the resource's capacity (100%) when at least one Sink phase is active, because the capacity of Sink phases is only limited by the capacity of the resource. Whenever both S_1 and S_2 are active simultaneously, they have the same attributed usage and available capacity. As a result, the usage and capacity of Sink phases can fluctuate significantly over time, as the number of active Greedy and Sink phases varies. This matches our expert view on the real-world performance of graph analytics systems.

Table 4.2: Overview of functions computed by Grade10's process for bottleneck identification.

Sec.	Eq.	Function	Description
4.4.3	4.1	$RTB(p,R,t)$	*Resource-Type Bottleneck*. This function is the output of the bottleneck identification process. It determines whether a given phase p is bottlenecked on resource type R during time slice t.
4.4.3.1	4.2	$LRB(lp,r,t)$	*Leaf-phase Resource Bottleneck*. This function determines whether a given leaf phase lp is bottlenecked on resource r during time slice t.
	4.3	$BB(lp,r,t)$	*Blocking-resource Bottleneck*. This function determines whether a given leaf phase lp is bottlenecked on blocking resource r during time slice t.
	4.4	$CB(lp,r,t)$	*Consumable-resource Bottleneck*. This function determines whether a given leaf phase lp is bottlenecked on consumable resource r during time slice t.
4.4.3.2	4.5	$LRTB(lp,R,t)$	*Leaf-phase Resource-Type Bottleneck*. This function determines whether a given leaf phase lp is bottlenecked on resource type R during time slice t.
4.4.3.3	4.6	$CRTB(cp,R,t)$	*Composite-phase Resource-Type Bottleneck*. This function determines whether a given composite phase cp is bottlenecked on resource type R during time slice t.
	4.7	$SRTB(cp,R,t)$	*Subphases Resource-Type Bottlenecks*. This function determines for every subphase of a given phase cp whether the subphase is bottlenecked on resource type R during time slice t.

A final observation from Figure 4.9e is that the sums of the Greedy and Sink phases match the partial samples produced by the phase-aware sampling method, as depicted in Figure 4.8c. This is a direct result of the design of the phase-aware sampling method; both the sampling and attribution steps first assign resource usage to Greedy phases up to their capacity, and then assign the remainder to Sink phases. A key difference between the two steps is that the sampling step divides resource usage across time slices, whereas the resource attribution step divides usage across phases within a single time slice. Furthermore, Grade10's resource attribution process can be configured to use other sampling methods, as demonstrated in our experiments (see Section 5.5.1.3).

4.4.3. Bottleneck Identification

Grade10 uses the outcome of its resource attribution process to identify *bottlenecks* (R1), that is, periods of resource usage equal or close to the resource's capacity. The bottleneck identification process determines for every phase, type of resource, and time slice whether a particular phase was bottlenecked on the given resource type during the given time slice. The output of this process is, for each phase in a job, a list of identified bottlenecks over time. This data may be visualized to give users insight into when a phase is bottlenecked and on which resource(s) (Section 4.4.5), or it may be further analyzed by Grade10, e.g., to identify the most impactful bottlenecks in a job (Section 4.4.4.2).

Grade10's process for bottleneck identification is defined through a set of functions, which we present in this section. Each function determines for a specific combination of inputs whether a bottleneck is detection (e.g., for a given combination of a phase, a resource, and a time slice). Functions computed for bottleneck identification are named with acronyms that reflect whether they compute per resource (R) or resource type (RT), whether they are specific to leaf phases (L) or to composite phases (C), etc. Table 4.2 presents an overview of the functions introduced in this section, including their acronyms and short descriptions.

The outcome of the bottleneck identification process is defined by the function RTB (Resource-Type Bottleneck, Equation 4.1). Because this function is defined for all phases in an execution model, it provides both high-level and low-level insight into the bottlenecks of a job (R4). We explain in the next sections the main components of function RTB, in turn. To compute $LRTB$ (Section 4.4.3.2), Grade10 first computes a helper function, LRB, which characterizes the bottlenecks for leaf-phases, per resource (Section 4.4.3.1). The function computing the bottlenecks for combined-phases, $CRTB$ (Section 4.4.3.3), does not require a separate step.

$$RTB(p,R,t) = \begin{cases} LRTB(p,R,t) & \text{if } p \text{ is a leaf phase, as defined in Equation 4.5} \\ CRTB(p,R,t) & \text{if } p \text{ is a composite phase, as defined in Equation 4.6} \end{cases} \tag{4.1}$$

4.4.3.1. Step 1: Identify Leaf Phase Bottlenecks Per Resource

The first step of Grade10's bottleneck identification process determines for every leaf phase, for every resource, and for every time slice whether a phase was bottlenecked on a resource during a time slice. For blocking resources, we define a bottleneck as a period during which a phase was waiting for the resource to become available. For consumable resources, bottlenecks occur when a resource's usage is at least some fraction δ of its capacity or when the usage attributed to a phase is at least some fraction δ of the phase's available capacity. The intuitive definition of a bottleneck is obtained for $\delta = 1$; whenever a resource's usage is equal to its capacity, any phase using that resource can only be sped up by reducing the resource's usage or increasing its capacity. Setting a lower threshold leads to a broader definition of bottlenecks (e.g., periods of at least 90% utilization for $\delta = 0.9$), but also reduces the sensitivity to small measurement or attribution errors. Although Grade10 allows different thresholds to be set per resource and per resource-phase pair, we use a fixed value of $\delta = 0.95$ in our prototype.

A comprehensive definition of when a leaf phase is bottlenecked is given by Equation 4.2 where $LRB(lp,r,t)$ indicates whether leaf phase lp is bottlenecked on resource r during time slice t. Function LRB is undefined if lp was not alive during time slice t or if lp cannot use resource r (i.e., lp is not mapped to r or the None attribution rule is configured for lp and r). Functions BB and CB define the conditions for being bottlenecked on a blocking or consumable resource, respectively. The latter function depends on U, which is either the total sampled usage of resource r or the usage attributed to phase lp, on C, which is either resource r's capacity or the capacity available to phase lp, and on δ as previously defined.

$$LRB(lp,r,t) = \begin{cases} \text{UNDEFINED} & \text{if } lp \text{ starts after } t \text{ or ends before } t \\ \text{UNDEFINED} & \text{if } lp \text{ cannot use } r \\ BB(lp,r,t) & \text{if } r \text{ is a blocking resource, with } BB \text{ defined below} \\ CB(lp,r,t) & \text{if } r \text{ is a consumable resource, with } CB \text{ defined below} \end{cases} \tag{4.2}$$

$$BB(lp,r,t) = \begin{cases} \text{TRUE} & \text{if } r \text{ is blocked during time slice } t \\ \text{FALSE} & \text{otherwise} \end{cases} \tag{4.3}$$

$$CB(lp,r,t) = \begin{cases} \text{FALSE} & \text{if } lp \text{ is waiting for any blocking resource during time slice } t \\ \text{TRUE} & \text{if } U(lp,r,t) > 0 \text{ and } U(lp,r,t) \geq \delta(lp,r)C(lp,r,t) \\ \text{TRUE} & \text{if } U(r,t) \geq \delta(r)C(r) \\ \text{FALSE} & \text{otherwise} \end{cases} \tag{4.4}$$

4.4.3.2. Step 2: Identify Leaf Phase Bottlenecks Per Resource Type

A typical distributed graph-processing platforms targeted by Grade10 contains several types of resources and multiple instances of most types. For example, in the resource model presented in Figure 4.4 there are numerous *CPU* and *Network* resources across the different machines in the distributed system. For high-level analysis of bottlenecks, e.g., to identify performance issues (see Section 4.4.4.2), it is often more important to determine which types of resources a phase is bottlenecked on, instead of which specific instances of those types. Thus, the second step in Grade10's bottleneck identification process summarizes the per-resource bottlenecks to identify bottlenecks per resource type. We consider a phase to be bottlenecked on a resource type whenever it is bottlenecked on at least one resource of that type, as expressed by Equation 4.5.

$$LRTB(lp,R,t) = \begin{cases} \text{UNDEFINED} & \text{if } LRB(lp,r,t) \text{ is undefined for all resources } r \text{ of type } R \\ \text{TRUE} & \text{if } LRB(lp,r,t) \text{ for any resource } r \text{ of type } R \\ \text{FALSE} & \text{otherwise} \end{cases} \tag{4.5}$$

4.4.3.3. STEP 3: IDENTIFY COMPOSITE PHASE BOTTLENECKS

The final step in Grade10's bottleneck identification process aims to identify bottlenecks in composite phases. Because resource usage is only attributed to leaf phases, bottlenecks in a composite phase cannot be identified directly. Instead, Grade10 uses a recursive approach that defines bottlenecks in a composite phase as a function of bottlenecks in its subphases. Equation 4.6 defines whether a composite phase cp is bottlenecked on resource type R during time slice t. The user-specified function P_*, a *bottleneck predicate*, combines subphase-bottleneck pairs to determine if the composite phase should be marked as bottlenecked.

$$CRTB(cp,R,t) = \begin{cases} \textsc{Undefined} & \text{if } cp \text{ starts after } t \text{ or ends before } t \\ \textsc{Undefined} & \text{if no leaf phase under } cp \text{ can use a resource of type } R \\ P_*(SRTB(cp,R,t)) & \text{if } SRTB(cp,R,t) \neq \emptyset\text{, with } SRTB \text{ defined below} \\ \textsc{False} & \text{if } SRTB(cp,R,t) = \emptyset \end{cases} \tag{4.6}$$

$$SRTB(cp,R,t) = \Big\{\big(p, RTB(p,R,t)\big) : p \text{ is a subphase of } cp,\ RTB(p,R,t) \text{ is defined}\Big\} \tag{4.7}$$

To demonstrate the functionality of Grade10's bottleneck identification process, we provide several predefined bottleneck predicates, as summarized in Table 4.3. Grade10 includes three basic bottleneck predicates: ANY, ALL, and MAJORITY. These predicates hold (i.e., indicate that a composite phase is bottlenecked) if any, all, or the majority of subphases are bottlenecked, respectively. The selected bottleneck predicate greatly impacts the number, duration, and interpretation of identified bottlenecks. For example, the ANY predicate will propagate a bottleneck in any leaf phase to the top of the execution model hierarchy and present the bottleneck as a job-wide bottleneck. This may significantly overstate the importance of the bottleneck. Conversely, the ALL predicate fails if a single leaf phase is not bottlenecked and may thus fail to present important bottlenecks.

More comprehensive bottleneck predicates may use phase- or resource-specific rules or otherwise exploit domain knowledge to determine when bottlenecks identified in subphases should be considered bottlenecks in a composite phase. Grade10 includes one such compound predicate: MAJORITY+ANY. First, this predicate considers separately bottlenecks in different subphase types; a bottleneck in a subphase is only relevant if it occurs in the majority of subphases of the same type (using the MAJORITY predicate). Second, the composite phase is considered to be bottlenecked if at least one type of subphase is bottlenecked (using the ANY predicate). The impact of all four bottleneck predicates on the bottlenecks identified in a real-world job is studied through experimentation in Section 5.5.2.

4.4.4. PERFORMANCE ISSUE IDENTIFICATION

Grade10's final process for analyzing graph-processing jobs identifies a variety of performance issues (R2). Grade10 proposes a hierarchical, rule-based approach for systematically analyzing the phases of a job and attributing performance issues to specific phases at any level in the execution model (R4).

Grade10 first analyzes leaf phases for *potential* performance issues. Next, the performance issues identified for child phases are aggregated at the level of each parent phase, to ultimately determine if and how these performance issues impact the root phase. Finally, a post-processing step extracts identified performance issues and estimates their impact on a phase's makespan. That is, Grade10 first identifies *potential* performance issues in a graph analysis job, and later keeps only the *actual* performance issues. This approach is motivated by the fact that some performance issues are only revealed by aggregating the results of many phases (e.g., an individual phase can not be imbalanced, but at higher levels of the hierarchy it may be revealed that there is an imbalance in duration across many phases of the same type). User-defined analysis

Table 4.3: Predefined bottleneck predicates included in Grade10 for identifying bottlenecks in composite phases.

Name	Description
ANY	A composite phase is bottlenecked when at least one of its subphases is bottlenecked.
ALL	A composite phase is bottlenecked when all of its subphases are bottlenecked.
MAJORITY	A composite phase is bottlenecked when at least half of its subphases are bottlenecked.
MAJ+ANY	A composite phase is bottlenecked when at least half of all subphases of the same type are bottlenecked, for at least of type of subphase.

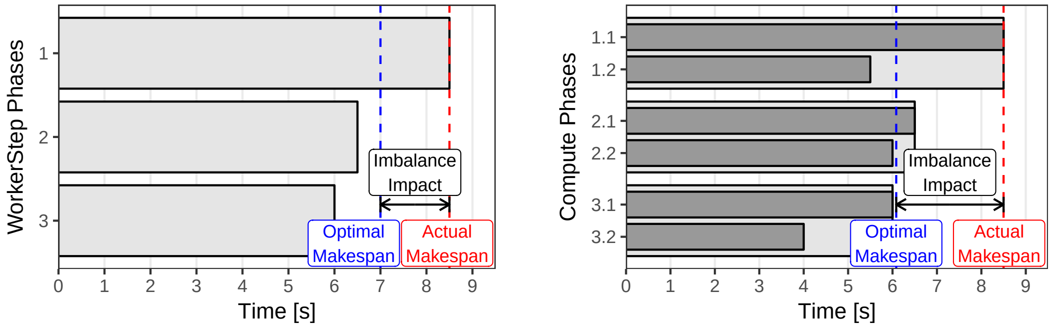
Algorithm 3 Framework for hierarchical performance issue identification. Functions AnalyzeLeaf, AnalyzeComposite, and ExtractIssues are specific to identifying a particular kind of performance issue and are used to analyze leaf phases, analyze composite phases, and extract performance issues from analysis results, respectively.

```
Input:
    $p$                                                        ▷ Root of job's execution model
Output:
    $I$                                                        ▷ Identified performance issues

 1: function IdentifyPerformanceIssues($p$)
 2:     $I \leftarrow []$
 3:     for $(p', r) \in$ AnalyzePhaseHierarchy($p$) do
 4:         append all issues returned by ExtractIssues($p', r$) to $I$
 5:     end for
 6:     return $I$
 7: end function

 8: function AnalyzePhaseHierarchy($p$)
 9:     if $p$ is a leaf phase then
10:         return $\{(p, \text{AnalyzeLeaf}(p))\}$
11:     else
12:         $R \leftarrow \bigcup_{s \in p.subphases}$ AnalyzePhaseHierarchy($s$)      ▷ Recurse over subphases
13:         $sR \leftarrow \{(p', r) \in R \mid p' \in p.subphases\}$               ▷ Extract records of subphases
14:         $r \leftarrow (p, \text{AnalyzeComposite}(p, sR))$     ▷ Analyze composite phase using subphase results
15:         return $R \cup \{r\}$
16:     end if
17: end function
```

rules specify for a particular kind of performance issue how it can be identified for a given leaf or composite phase using the execution model, resource model, resource attribution output, and/or identified bottlenecks as inputs.

Algorithm 3 formalizes the Grade10 approach for performance issue identification. The algorithm consists of two main steps and uses three user-defined functions (i.e., an analysis rule, provided by the user, that can identify a particular kind of performance issue). First (line 3, 8-17), the algorithm recursively analyzes the hierarchy of phases (i.e., the execution model) that represents a job, to extract potential performance issues (referred to as *records*). The base case of the recursion (lines 9-10) is the analysis of a leaf phase; a leaf phase has no further subphases, so the algorithm invokes the user-defined AnalyzeLeaf function to extract potential performance issues for a given leaf phase. The recursive case (lines 12-15), for a given composite phase, first recursively analyzes the (indirect) subphases of the composite phase, then extracts all records pertaining to (direct) subphases of the composite phase, and finally passes those records to a user-defined function (AnalyzeComposite) to identify potential performance issues in the composite phase, based on issues identified for subphases. The second step of Algorithm 3 (lines 4-6) extracts and outputs actual performance issues, from the records produced for all execution phases, using the user-defined ExtractIssues function.

Grade10's prototype includes two analysis rules (i.e., implementations of the AnalyzeLeaf, AnalyzeComposite, and ExtractIssues functions) for identifying performance issues related to phase imbalance and resource bottlenecks. We detail each of these rules, and algorithms that enforce them, in Section 4.4.4.1 for phase imbalance and Section 4.4.4.2 for resource bottlenecks. Additional rules can be added by expert users by implementing AnalyzeLeaf, AnalyzeComposite, and ExtractIssues functions that identify a specific kind of performance issue. We expect many kinds of performance issues that could be identified by Grade10 to be generic (i.e., not specific to one particular graph analysis platform), so their implementations could be shared within the community and reused across systems (satisfying R3).



(a) Estimated impact of *WorkerStep* imbalance on a *Superstep*.

(b) Estimated impact of *Compute* imbalance across *WorkerSteps* on a *Superstep*.

Figure 4.10: Identification of phase imbalance for an example phase, *Superstep*, and its subphases. The *Superstep* consists of three concurrent *WorkerStep* subphases. Each *WorkerStep* consists of two concurrent *Compute* subphases.

4.4.4.1. Analysis Rule for Identifying Phase Imbalance Issues

Workload imbalance is a common source of performance issues in data-processing systems with synchronous execution models. A single process taking much longer than others to complete its task, e.g., because its workload is much higher, will cause all other processes to wait at the next synchronization barrier. Thus, the makespan of the job increases while most system resources are idling. Graph analysis platforms are particularly vulnerable to imbalance, because many (distributed) graph analysis platforms use a synchronous, iterative execution model [23], and because real-world graphs and graph algorithms are highly irregular.

Grade10 is capable of identifying performance issues caused by imbalance through analysis of synchronous phases and of the distribution of their durations. Our approach consists of computing the actual and the optimal makespan for the execution of a set of phases of the same type (e.g., all *Compute* phases in a job or in a single *Superstep*). We define the makespan of a set of phases to be the time between the start of the first phase and the end of the last, executing each phase in sequence or concurrently as defined by the execution model specification. The difference between the actual, observed makespan and the optimal, computed makespan is considered to be the *estimated impact* of imbalance for a set of phases, i.e., by how much the makespan could have been reduced if there were no imbalance. This estimation is optimistic; it disregards the presence of other types of phases in the system, the impact of other performance issues, etc.

Algorithm 4 implements the computation of actual and optimal makespan, using Grade10's hierarchical performance analysis framework, and identifies for any phase (the *target phase*) which types of subphases are imbalanced. The actual makespan of a set of phases is computed as the maximum duration if the phases ran concurrently, and the sum of durations if the phases ran sequentially. The optimal makespan is conservatively computed as the mean duration of concurrent phases, and the sum of sequential phases. These definitions are based on the assumption that the total workload of a set of concurrent phases is fixed for a given job, so the optimal makespan is achieved by dividing the workload evenly across phases. Imbalance in sequential phases is not considered; redistributing workload for sequential phases does not typically reduce the makespan of a job.

The complexity of Algorithm 4 is $O(P \times P^{\star})$ where P is the number of phases in a job and $P^{\star}$ is the number of phase types. This upper bound can be obtained by determining the maximum number of records that can be produced (complexity $O(P \times P^{\star})$) and the amount of work performed per issue (complexity $O(1)$). The output of Algorithm 4 includes for every phase one record per (indirect) subphase type, so $O(P \times P^{\star})$ records overall. For a leaf phase (lines 1-3), one record is produced in constant time, so $O(1)$ work per record. For a composite phase (lines 4-25), the amount of work depends on the number of records produced by its subphases. The nested for loops (lines 6-9) effectively partition the list of input records sR into one list R' per subphase type, which takes linear time in the number of input records. The inner loop (lines 10-21) takes $O(|R'|)$ time, so the total duration is linear in the number of input records and constant per record. Thus, the work performed by AnalyzeComposite is $O(1)$ per record. The final step, ExtractIssues, also takes constant time per record.

Figure 4.10 depicts an example application of Grade10's approach to detecting phase imbalance. The example in Figure 4.10a consists of a single *Superstep* phase comprising three *WorkerStep* phases. The *WorkerSteps* take 8.5, 6.5, and 6 seconds, respectively. These phases are executed concurrently, so the actual duration

Algorithm 4 Algorithm for identifying performance issues caused by phase duration imbalance using Grade10's hierarchical performance analysis framework (see Algorithm 3). For every phase p and every (indirect) subphase type rt (and the type of p), the algorithm produces a record containing the actual and optimal makespan of all phases of type rt under p. Records with an actual makespan exceeding the optimal makespan are presented to the user to indicate phase imbalance issues.

```
function AnalyzeLeaf($p$)
    return $[\{type = p.type, actMakespan = p.duration, optMakespan = p.duration\}]$
end function

function AnalyzeComposite($p$, $sR$)
    $ROut \leftarrow [\{type = p.type, actMakespan = p.duration, optMakespan = p.duration\}]$
    for $st \in p.subphaseTypes$ do                                         ▷ For each subphase type
        $R \leftarrow$ concatenate lists of records $[R \mid (s, R) \in sR, s.type = st]$
        for $rt \in \{r.type \mid r \in R\}$ do       ▷ For each phase type with $st$ as ancestor (and $st$ itself)
            $R' \leftarrow [r \in R \mid r.type = rt]$          ▷ Extract imbalance record(s) for phases of type $rt$
            if $|R'| = 1$ then                          ▷ If one record is found, imbalance does not change
                append $R'[1]$ to $ROut$
            else                                 ▷ Otherwise, determine imbalance across phases of type $rt$
                if $st$ is sequentially repeatable then
                    $am \leftarrow \sum_{r \in R'} r.actMakespan$
                    $om \leftarrow \sum_{r \in R'} r.optMakespan$
                else
                    $am \leftarrow \max_{r \in R'} r.actMakespan$
                    $om \leftarrow \sum_{r \in R'} r.optMakespan / |R'|$
                end if
                append record $\{type = rt, actMakespan = am, optMakespan = om\}$ to $ROut$
            end if
        end for
    end for
    return $ROut$
end function

function ExtractIssues($p$, $R$)
    $I \leftarrow []$
    for $r \in R, r.actMakespan > r.optMakespan$ do
        $\Delta \leftarrow r.actMakespan - r.optMakespan$
        append issue $\{targetPhase = p, imbalancedPhaseType = r.type, estimatedImpact = \Delta\}$ to $I$
    end for
    return $I$
end function
```

of the *Superstep* is 8.5 seconds, compared to 7 seconds if the *WorkerSteps* were balanced. The computed impact of phase imbalance is 1.5 seconds, or ≈ 18% (1.5/8.5). Figure 4.10b extends the example by analyzing the imbalance of the *Compute* phases that make up each *WorkerStep*. The actual duration of the *Superstep* remains 8.5 seconds, but the optimal duration would be ≈ 6.1 seconds. Thus, the observed impact of imbalance is ≈ 28%. Because each *WorkerStep* takes as long as its longest subphase, some of the effects of workload imbalance are "hidden". By computing imbalance recursively, Grade10 is able to reveal imbalance in all layers of an execution model.

4.4.4.2. Analysis Rule for Identifying Resource Bottleneck Issues

The second kind of performance issue currently identified by Grade10 is the presence of bottlenecks on hardware and software resources. Similar to the approach used to identify phase imbalance issues, Grade10 estimates the time spent bottlenecked on a given resource for each phase, based on the bottlenecks identified in subphases. Algorithm 5 formalizes our approach using the hierarchical performance analysis framework.

Algorithm 5 Algorithm for identifying performance issues caused by bottlenecks using Grade10's hierarchical performance analysis framework (see Algorithm 3). For every phase p and every resource type rt used by p, the algorithm produces a record containing the estimated time p (or its subphases) spent bottlenecked on any resource of type rt. Records with a non-zero time spent bottlenecked on a resource are presented to the user as performance issues.

```
function AnalyzeLeaf($p$)
    $ROut \leftarrow []$
    for each resource type $rt$ used by $p$ do
        append record $\{source = rt, impact = (\text{time } p \text{ is bottlenecked on } rt)\}$ to $ROut$
    end for
    return $ROut$
end function

function AnalyzeComposite($p$, $sR$)
    $stR \leftarrow \emptyset$
    for $st \in p.subphaseTypes$ do
        $R \leftarrow$ concatenate lists of records $[R \mid (s, R) \in sR, s.type = st]$
        for $src \in \{r.source \mid r \in R\}$ do
            $R' \leftarrow [r \in R \mid r.source = src]$
            if $|R'| = 1$ then
                $i \leftarrow R'[1].impact$
            else if $st$ is sequentially repeatable then
                $i \leftarrow \sum_{r \in R'} r.impact$
            else
                $i \leftarrow \sum_{r \in R'} r.impact / |R'|$
            end if
            add record $\{subphaseType = st, source = src, impact = i\}$ to $stR$
        end for
    end for
    $ROut \leftarrow []$
    for $src \in \{r.source \mid r \in stR\}$ do
        $R \leftarrow [r \in stR \mid r.source = src]$
        $G \leftarrow$ ConstructDAG($p, R$)
        $i \leftarrow$ length of longest path in $G$ by sum of node weights ("critical path" for possible optimization)
        append record $\{source = src, impact = i\}$ to $ROut$
    end for
    return $ROut$
end function

function ExtractIssues($p$, $R$)
    $I \leftarrow []$
    for $r \in R, r.impact > 0$ do
        append issue $\{targetPhase = p, source = r.source, estimatedImpact = r.impact\}$ to $I$
    end for
    return $I$
end function

function ConstructDAG($p$, $R$)
    $V \leftarrow$ all subphase types of $p$
    $w(v) \leftarrow 0$ for all $v \in V$                                  ▷ Default vertex weight of 0
    for $v \in V$ with a record $r \in R$ such that $r.subphaseType = v$ do
        $w(v) \leftarrow r.impact$
    end for
    $E \leftarrow \{(x, y) \mid x, y \in V, x$ precedes $y\}$          ▷ Add edges to each subphase type from all preceding types
end function
```

First (lines 10-23), for each composite phase, bottleneck durations are combined per subphase type by summing the durations for sequential phases of the same type and averaging concurrent phases. Next (lines 25-32), the durations of subphase types are combined by identifying the sequence of subphase types (i.e., subphase types that must execute sequentially due to precedence constraints) with the longest total bottleneck duration. The estimated bottleneck duration represents an upper bound on the reduction in makespan that may be achieved by eliminating all bottlenecks on a given resource. Furthermore, it represents optimistic estimation of the fraction of a phase's total duration that can be attributed to the usage of a specific resource.

The complexity of Algorithm 5 is $O(P \times (P^\star)^2 \times R^\star)$ where P is the number of phases in a job, $P^\star$ is the number of phase types, and $R^\star$ is the number of resource types. This upper bound can be obtained by determining the maximum number of records that can be produced (complexity $O(P \times R^\star)$) and the amount of work performed per issue (complexity $O((P^\star)^2)$). The output of Algorithm 5 includes for every phase one record per resource type, so $O(P \times R^\star)$ records overall. For a leaf phase (lines 1-7), each record is produced in $O(1)$ time, assuming bottlenecks have already been identified. For a composite phase (lines 8-34), there are two steps to consider. The first step is a nested loop (lines 9-23) which partitions the list of input records per subphase type and resource type. Its runtime is $O(1)$ per input record. The second step loops over resource types and produces $O(R^\star)$ records. Per record the algorithm creates a DAG of phase types with $O(P^\star)$ nodes and $O((P^\star)^2)$ edges. The creation of the DAG and the identification of the longest path take $O((P^\star)^2)$ time. Thus, Algorithm 5 requires $O((P^\star)^2)$ work per record.

Grade10's approach for identifying resource bottleneck issues is inspired by the blocked time analysis methodology by Ousterhout et al. [76], which has been applied to understand the performance of Apache Spark [13, 94], a big data analytics platform. Blocked time analysis first computes for each task how much faster it could complete if it never blocked on network or disk. The "optimized" task runtimes are then used to compute the job's reduced runtime by simulating how Spark would have scheduled the shorter tasks.

There are several key differences between Grade10 and previous work on blocked time analysis. First, Ousterhout et al. instrument Spark to log when and how long each task blocks on network or disk, whereas Grade10 derives this information automatically using its bottleneck identification process, and without the code intrusions of the method proposed by Ousterhout et al. Grade10's approach also analyzes other resources that may be present in a job's resource model, in particular, the software resources that are likely to block in Spark and in all other Java-based environments (e.g., due to the Java Garbage Collector starting its process, outside the control of the user). Second, in contrast to the implicit execution model of one job with multiple tasks used in [76], Grade10 supports arbitrarily complex, hierarchical, explicit models. Similarly to the blocked time analysis method, Grade10 aims to estimate the impact of removing bottlenecks on subphases (tasks) on the duration of the parent phase (job). However, Grade10's estimations are expected to be less accurate, primarily due to the added complexity of Grade10 execution models (e.g., precedence constraints) and phenomena that are not currently modeled (e.g., speeding up computation on only one machine may lead to longer wait times and no overall improvement in the job completion time). More accurate estimation of the impact of bottlenecks on composite phases may improve Grade10's analysis of related performance issues.

4.4.5. Result Presentation

The final step in Grade10's performance analysis pipeline is presenting the outcome of the analysis to users. Grade10's result presentation is phase-centric. Users may select any phase in a job to retrieve performance data specific to that phase. This allows users of varying levels of expertise to obtain either a high-level overview of a job's performance (for phases near the root of the hierarchy) or analyze in-depth the performance of deeply-nested phases (R4). Performance data presented at each phase includes performance issues ranked by estimated impact, a timeline of bottlenecks during the phase's execution, a breakdown of bottlenecks per resource and per subphase, and the resource usage and capacity attributed to the phase (for leaf phases only).

Grade10 currently includes a novel visualization method for bottlenecks. Figure 4.11 depicts an example report of the bottlenecks Grade10 has identified for one phase in a graph analysis job, the *target phase*. The report consists of multiple plots, each depicting identified bottlenecks over time. The plots are grouped into three sections: an overview of the bottlenecks identified for the target phase, a breakdown of the identified bottlenecks by resource, and an overview of the bottlenecks identified for each subphase of the target phase.

Each section in a bottleneck identification report can, with minor experience in reading such a report, quickly reveal various details of the bottlenecks identified for a phase. From the top section, users can learn how much time a phase is bottlenecked on any resource (non-white shaded area) vs. not bottlenecked at all (white shaded area). They can also learn which bottlenecks often occur simultaneously (e.g., in Figure 4.11, the message queue, garbage collector, and CPU resources are frequently bottlenecked simultaneously around $t = 16000$ ms). From the middle section, users can learn for a particular resource when it is a bottleneck. This information is somewhat obscured in the top plot, due to the fluctuations in vertical space assigned to a resource. From the bottom section, users can learn which subphases contributed to a particular bottleneck, therefore guiding the user's analysis to the lower-level components that contributed to a bottleneck.

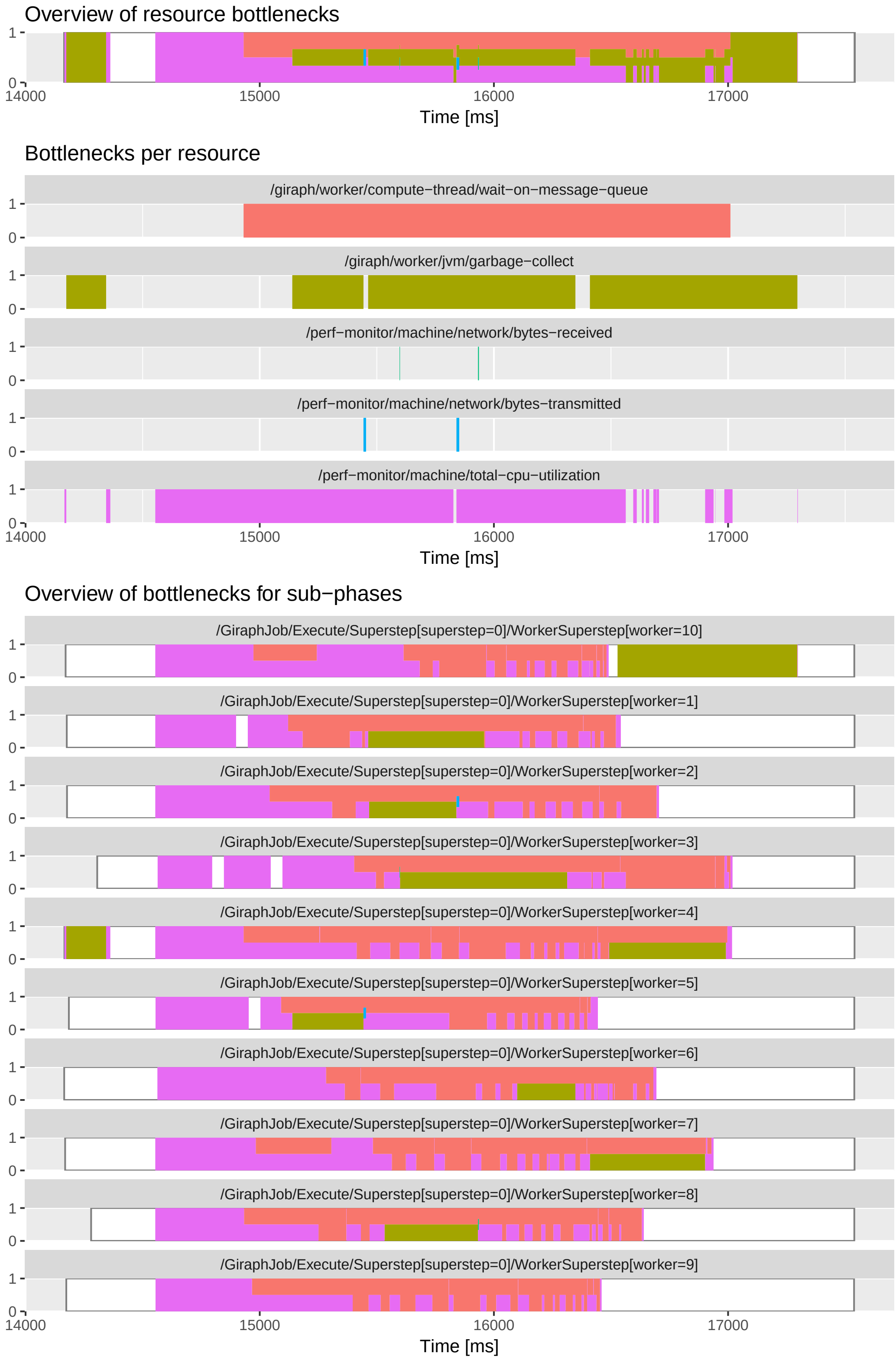


Figure 4.11: Example report produced by Grade10's bottleneck identification process for one phase in a graph analysis job. The visualization consists of three groups of plots. The horizontal axes represent time and are aligned for all plots. (Top) Overview of bottlenecks in the phase for which this report was generated. (Middle) Breakdown of the identified bottlenecks by resource. Functions as a legend for the resources and their corresponding colors. (Bottom) Overview of bottlenecks for each subphase of the phase depicted in the top plot. (Top & Bottom) The white shaded areas indicate a lack of bottlenecks. Shaded areas of other colors indicate the presence of a bottleneck on the resource corresponding to that color. The dark gray rectangle delineates the start and end of a phase.

5

Experimental Results

We address in this chapter our third research question: How to validate the designed approaches through prototypes? Our approach is the implementation of Graphalytics and Grade10 as prototypes, and the design and analysis of real-world experiments to validate our approaches in practice. We compare six modern graph analysis platforms using the Graphalytics benchmark to demonstrate the insight it provides into the performance of those platforms. We use Grade10 to further analyze the performance of two platforms in-depth, and we manually validate Grade10's internal processes with real-world data.

The remainder of this chapter is structured as follows. In Section 5.1, we discuss the prototype implementations. We present the experimental setup and design in Section 5.2. We discuss how we ensure the validity of our results in Section 5.3. Finally, we present and discuss the results of our experiments for Graphalytics in Section 5.4, and for Grade10 in Section 5.5.

5.1. Prototype Implementation

For our experiments, we have implemented prototypes of Graphalytics and Grade10. Graphalytics has been developed using modern software development practices, including open-source development through GitHub [41], continuous integration through Jenkins, and source code quality analysis through SonarQube. We have implemented Graphalytics using the Java programming language to ensure portability across the many environments in which the Graphalytics benchmark may be used. For the implementation of each platform driver, we have used Java to interface with the Graphalytics benchmark harness, and the native language of the given graph analysis platform to implement the algorithms included in the benchmark. Since the publication of our work on Graphalytics[85], our prototype has been deployed in multiple environments and several users have reportedly implemented drivers for their own systems to benchmark them using Graphalytics. In particular, the PGX product team of Oracle and the Graphmat product team of Intel are using Graphalytics in their product development processes.

Grade10 has been developed internally as a research prototype. We have implemented all features presented in Chapter 4 in the Kotlin programming language. Additionally, we have designed prototype execution models for two graph processing platforms, Giraph and PowerGraph (introduced in Section 5.2), and instrumented the platforms, using their respective built-in logging facilities, to enable analysis of their performance using Grade10. To collect resource usage data during our experiments, we have implemented a lightweight custom monitoring tool in C on top of Linux's *procfs*.

5.2. Experimental Setup and Design

In this section, we describe the experimental setup and design for our Graphalytics and Grade10 experiments. The goals of the experiments with Graphalytics and Grade10 are different, alongside the dimensions each of these instruments favors (see Section 1.4). For Graphalytics, we focus on broad benchmarking experiments, where comparing systems is the key goal of the experiment. For Grade10, we focus on two kinds of experiments: validation experiments, where validating the internal processes of Grade10 is the key goal of the experiment, and on deep custom-benchmarking experiments, where exploring the performance issues of a single system is the key goal of the experiment. For each experiment, the experimental setup includes:

a selection of graph analysis platforms, and the design of the experiment itself (goals, metrics, workload as algorithms and datasets, and deployment environment).

5.2.1. Selected Graph-Processing Platforms

A major contribution of this work is the evaluation and comparison of graph analysis platforms whose development is *industry-driven*, and of other, *community-driven*, platforms. This is a contribution, because it is rare to be able to compare industry-driven efforts — typically, companies refuse to contribute to the evaluation process, and in many cases the leading companies refuse through clauses in the terms of use the right to publish performance results from commercial products. We evaluate and compare in this work six different graph analysis platforms, three *community-driven* (C) and three *industry-driven* (I), see Table 5.1. These platforms are based on six different programming models, spanning an important design space for real-world graph analysis.

The platforms can be categorized into two classes: *distributed* (D) and *non-distributed* (S) platforms. Distributed platforms use when analyzing graphs multiple machines connected using a network, whereas non-distributed platforms can only use a single machine. Distributed systems suffer from a performance penalty because of network communication, but can scale to handle graphs that do not fit into the memory of a single machine. Non-distributed systems cannot scale as well because of the limited amount of resources of a single machine.

- **Apache Giraph** [30] uses an iterative vertex-centric programming model similarly to Google's Pregel. Giraph is open source and built on top of Apache Hadoop's MapReduce.
- **Apache GraphX** [14] is an extension of Apache Spark, a general platform for big data processing. GraphX extends Spark with graphs based on Spark's Resilient Distributed Datasets (RDDs).
- **PowerGraph** [28], developed by Carnegie Mellon University, is designed for real-world graphs which have a skewed power-law degree distribution. PowerGraph uses a programming model known as Gather-Apply-Scatter (GAS).
- **GraphMat** [29, 95], developed by Intel, maps Pregel-like vertex programs to high-performance sparse matrix operations, a well-developed area of HPC. GraphMat supports two different backends which need to be selected manually: a single-machine shared-memory backend [29] and a distributed MPI-based backend [95].
- **OpenG** [53] consists of handwritten implementations for many graph algorithms. OpenG is used by *GraphBIG*, a benchmarking effort initiated by Georgia Tech and inspired by IBM System G.
- **PGX** [96], developed by Oracle, is designed to analyze large scale graphs on modern hardware systems. PGX has two different runtimes: a single-machine shared memory runtime implemented in Java and a distributed runtime [55] implemented in C++. Both runtimes share the same user facing API and are part of the Oracle Big Data Spatial and Graph product [15].

5.2.2. Environment

We perform our experiments on the DAS-5 [40] (Distributed ASCII Supercomputer), a supercomputer consisting of 6 clusters with over 200 dual 8-core compute nodes. DAS-5 is funded by a number of organizations and universities from the Netherlands and is actively used as a tool for computer science research in the Netherlands. We use individual clusters for our experiments and we test all platforms on the same hardware. The hardware specifications of the machines in DAS-5 are listed in Table 5.2.

The environment we use in our experiments is representative of the environments used in practice for big data analytics; we use common-off-the-shelf hardware, which matches the setups reported by industry, e.g.,

Table 5.1: Selected graph-processing platforms. Acronyms: C, community-driven; I, industry-driven; D, distributed; S, non-distributed.

Type	Name	Vendor	Language	Model	Version
C, D	Giraph [30]	Apache	Java	Pregel	1.1.0 (Graphalytics) / 1.2.0 (Grade10)
C, D	GraphX [14]	Apache	Scala	Spark	1.6.0
C, D	PowerGraph [28]	CMU	C++	GAS	2.2
I, S/D	GraphMat [29, 95]	Intel	C++	SpMV	May '16
I, S	OpenG [53]	G.Tech	C++	Native code	May '16
I, S/D	PGX [55, 96]	Oracle	Java/C++	Push-pull	May '16

Table 5.2: Hardware specifications.

Component	**Specification**
CPU	2 × Intel Xeon E5-2630 @ 2.40 GHz
Cores	16 (32 threads with Hyper-Threading)
Memory	64 GiB
Disk	2 × 4 TB
Network	1 Gbit/s Ethernet, FDR InfiniBand

Table 5.3: Experiments used for Graphalytics benchmarks.

Category	**Sec.**	**Experiment**	**Algorithms**	**Datasets**	**#nodes**	**#threads**	**Metric**
Baseline	5.4.1	Dataset variety	BFS, PR	All, up to L	1	-	T_{proc}, E(V)PS
	5.4.2	Algorithm variety	All	R4(S), D300(L)	1	-	T_{proc}
Scalability	5.4.3	Vertical	BFS, PR	D300(L)	1	1-32	T_{proc}, S
	5.4.4	Strong/Horizontal	BFS, PR	D1000(XL)	1-16	-	T_{proc}
	5.4.5	Weak/Horizontal	BFS, PR	G22(S)-26(XL)	1-16	-	T_{proc}
Robustness	5.4.6	Stress test	BFS	All	1	-	SLA
	5.4.7	Variability	BFS	D300(L), D1000(XL)	1, 16	-	CV

Table 5.4: Design of Grade10 experiments. Acronyms: *Platform:* G, Giraph; P, PowerGraph. *Jobs:* 1, PageRank on Datagen-300; All, all combinations of algorithm (BFS, CDLP, PR, WCC) and dataset (Datagen-1000, Graph500-26). *Nodes:* W, workers, O, other.

Category	**Sec.**	**Experiment**	**Platforms**	**#nodes**	**Jobs**
Resource Attribution	5.5.1.1	Attribution Rules	G	10 W + 3 O	1
	5.5.1.2	Blocking Events	G	10 W + 3 O	1
	5.5.1.3	Sampling Method	G/P	10 W + 3/1 O	1
Bottleneck Identification	5.5.2.1	Predicates	G	10 W + 3 O	1
Performance Issue Identification	5.5.3.1	Bottlenecks	G/P	10 W + 3/1 O	All
	5.5.3.2	Phase Imbalance	G/P	10 W + 3/1 O	All

by Google [24] and Facebook [4]. Furthermore, our environment is comparable to the setups used in many recent performance comparisons in the community [14, 27, 28, 52]

5.2.3. DESIGN OF GRAPHALYTICS EXPERIMENTS

Graphalytics conducts automatically the complex set of experiments summarized in Table 5.3. The experiments are divided into three categories: *baseline*, *scalability*, and *robustness* (all introduced in Section 3.4). Each category consists of a number of experiments, for which Table 5.3 lists the parameters used for the benchmarks (algorithm, dataset, number of machines, and number of threads) and the metrics used to quantify the results. We run the Graphalytics benchmark for all six platforms described in Section 5.2.1.

For GraphMat and PGX, we report single-machine results using the single-machine backend, and horizontal scalability results using the distributed backend. For the single-machine horizontal scalability experiments we report results for both backends. Processing times reported for PGX shared memory exclude its integrated warm up procedure. Cold processing times are additionally reported in our technical report [97].

5.2.4. DESIGN OF GRADE10 EXPERIMENTS

The experiments we use to assess Grade10 are listed in Table 5.4. We design the experiments to demonstrate Grade10's performance analysis components and to evaluate the impact of tuning Grade10 on the accuracy and/or correctness of each component. As described in the table, we present results for Giraph for all experiments and for PowerGraph wherever adding a second platform provides additional insight in Grade10's operation. All resource attribution and bottleneck identification experiments use results from the same job(s): running PageRank on Datagen-300 with the platform(s) listed for the experiment. This job was arbitrarily chosen, but is small enough to allow manual inspection of Grade10's intermediate results (e.g., resource attribution data) to validate the outcome. Applying Grade10 to a broad range of platforms and workloads to study their performance in-depth is outside the scope of this work, but we present in Section 5.5.3 preliminary results of using the full Grade10 analysis pipeline to identify performance issues in Giraph and PowerGraph for a variety of jobs. The execution and resource model specifications used to analyze Giraph and PowerGraph are presented in Appendix B.

All experiments use the same hardware setup: 10 machines as workers in the DAS-5 (described in Section 5.2.2). For Giraph, we use 3 additional machines for (1) Hadoop master processes, Zookeeper, and the Graphalytics benchmark harness; (2) the MapReduce application master; and (3) the Giraph application master. For PowerGraph, we use 1 additional machine for the Graphalytics benchmark harness. We use a custom light-weight monitoring tool to capture CPU and network utilization directly from Linux's *procfs* at high frequency (20 Hz).

5.3. Validation of Results

Reproducibility of experiments in distributed systems is an open challenge [98]. To ensure validity of our results, we used a combination of approaches: (1) Where possible, we use manual verification, e.g., to confirm through small example inputs that the output of an algorithm implementation is correct. (2) We frequently engage with the Graphalytics team (through bi-weekly meetings) and the graph processing community (e.g., at GRADES workshops and meet-ups organized by LDBC) to present preliminary results and ensure that our results match the expectations of the community. (3) The developers of the industry-driven platforms listed in Section 5.2.1 are directly involved in tuning, running, and validating experiments involving their systems. (4) Last, Graphalytics validates the outcomes of the algorithms using a trusted code-base for workloads that are infeasible to verify manually.

5.4. Graphalytics Experiments

In this section, we use the Graphalytics benchmark to compare the performance of six graph analysis platforms. The experiments used for this comparison are summarized in Table 5.3. Key findings:

1. The performance of graph analysis systems is highly variable across datasets, algorithms, and platforms.

2. The workload (i.e., algorithm and dataset) impacts the performance of graph analysis platforms non-trivially; a platform performing well compared to others on one dataset and algorithm does not guarantee that it will perform well on other datasets or algorithms.

3. None of the tested platforms achieve (near) optimal vertical or horizontal scalability.

4. Performance variability is low across all platforms.

5.4.1. Dataset Variety

For this experiment, Graphalytics reports the processing time of all platforms executing BFS and PageRank on a variety of datasets using a single node. Key findings:

- GraphMat and PGX significantly outperform their competitors in most cases.
- PowerGraph and OpenG are roughly an order of magnitude slower than the fastest platforms.
- Giraph and GraphX are consistently two orders of magnitude slower than the fastest platforms.
- Across datasets, all platforms show significant variability in performance normalized by input size.

The workload consists of two selected algorithms and all datasets up to class L. We present the processing time (T_{proc}) in Figure 5.1, and the processed edges per second (EPS) and processed edges plus vertices per second (EVPS) in Figure 5.2. The vertical axis in both figures lists datasets, ordered by scale (results for missing datasets are available in our technical report [97]).

Figure 5.1 depicts the processing time of BFS and PageRank for all platforms on a variety of datasets. For both algorithms, GraphMat and PGX are consistently fast, although PGX has significantly better performance on BFS. Giraph and GraphX are the slowest platforms and both are two orders of magnitude slower than GraphMat and PGX for most datasets. Finally, OpenG and PowerGraph are generally slower than both PGX and GraphMat, but still significantly faster than Giraph and GraphX. A notable exception is OpenG's performance for BFS on dataset R3(XS). The BFS on this graph covers approximately 10% of the vertices in the

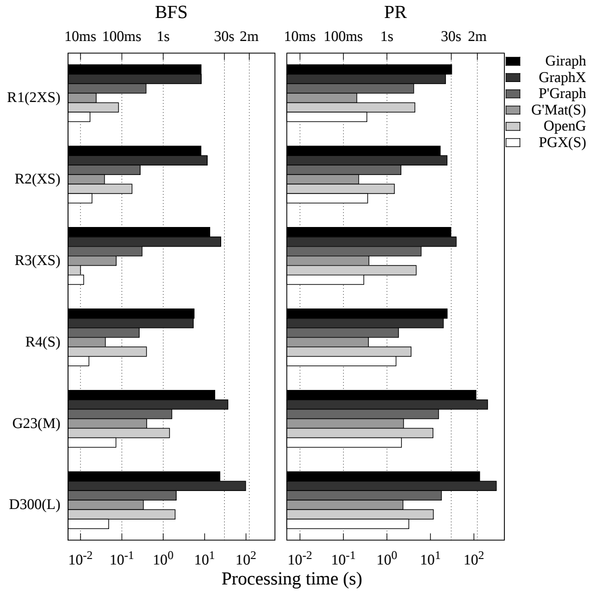


Figure 5.1: Dataset variety: T_{proc} for BFS and PR.

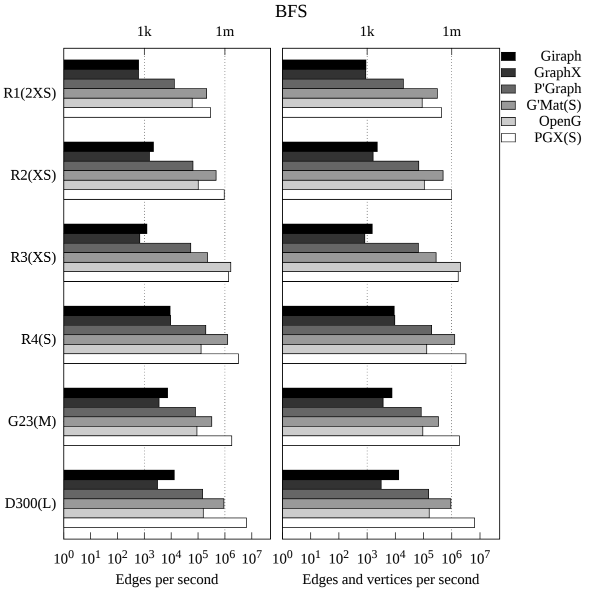


Figure 5.2: Dataset variety: EPS and EVPS for BFS.

graph, so OpenG's queue-based BFS implementation results in a large performance gain over platforms that process all vertices using an iterative algorithm.

To better understand the sensitivity of the tested platforms to the datasets, we present normalized processing times for the BFS algorithm in Figure 5.2. The left and right subfigures depict EPS and EVPS, respectively. Ideally, a platform's performance should be proportional to graph size, thus the normalized performance should be constant. As evident from the figure, all platforms show signs of dataset sensitivity, as EPS and EVPS vary between datasets.

Besides T_{proc}, it is also interesting to look at the makespan (i.e., time spent on the complete job for one algorithm). This includes platform-specific overhead such as resource allocation and graph loading. Table 5.5 lists the makespan, T_{proc}, and their ratio for BFS on D300(L). The percentages show that the overhead varies widely for the different platforms and ranges from 66% to over 99% of the makespan. However, we note that the platforms have not been tuned to minimize this overhead and in many cases it could be significantly reduced by optimizing the configuration. In addition, we observe that the majority of the runtime for all platforms is spent in loading the input graph, indicating that algorithms could be executed in succession with little overhead.

5.4.2. ALGORITHM VARIETY

The second set of baseline experiments focuses on the algorithm variety in the Graphalytics benchmark, and on how the performance gap between platforms varies between workloads. Key findings:

- Relative performance between platforms is similar for BFS, WCC, PR, and SSSP.
- LCC is significantly more demanding than the other algorithms, Giraph and GraphX are unable to complete it without breaking the SLA.
- GraphX is unable to complete CDLP. The performance gap for the remaining platforms for CDLP is smaller than for the other algorithms.

Figure 5.3 depicts T_{proc} for the core algorithms in Graphalytics on two graphs with edge weights: R4(S), the largest real-world graph in Graphalytics with edge-weights; and D300(L). BFS, WCC, PR, and SSSP all

Table 5.5: T_{proc} and makespan for BFS on D300(L).

Time	Giraph	GraphX	P'Graph	G'Mat(S)	OpenG	PGX(S)
Makespan	277.9 s	278.4 s	216.5 s	23.3 s	5.7 s	14.3 s
T_{proc}	23.4 s	97.9 s	2.1 s	0.3 s	1.9 s	0.05 s
Ratio	8.4%	35.2%	1.0%	1.3%	33.3%	0.3%

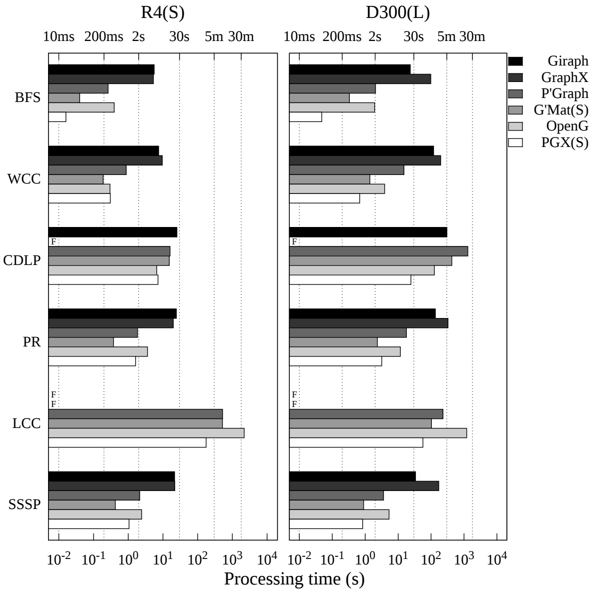


Figure 5.3: Algorithm variety: T_{proc}.

show similar results. PGX and GraphMat are the fastest platforms. Giraph, GraphX and PowerGraph are much slower, with GraphX showing the worst performance, especially on D300(L). OpenG's performance is close to that of PGX and GraphMat on WCC, and up to an order of magnitude worse for BFS, PR, and SSSP. CDLP requires more complex computation which results in longer processing times for all platforms, reducing the performance impact of the chosen platform, especially on smaller graphs like R4(S). GraphX is unable to complete CDLP on both graphs. LCC is also very demanding; Giraph and GraphX break the SLA for both graphs. The complexity of the LCC algorithm depends on the degrees of vertices, so longer processing times are expected on dense graphs. Because of the high density of R4(S), processing times are larger on this graph than on D300(L), despite it being an order of magnitude smaller.

5.4.3. VERTICAL SCALABILITY

To analyze the effect of adding additional resources in a single machine, we use Graphalytics to run the BFS and PageRank algorithms on D300(L) with 1 up to 32 threads on a single machine. Key findings:

- All platforms benefit from using additional cores, but only PowerGraph exceeds a speedup of 10 on 16 cores.
- Most platforms experience minor or no performance gains from Hyper-Threading.

Figure 5.4 depicts the processing time for this experiment. The majority of tested platforms show increasing performance as threads are added up to 16, the number of cores. Adding additional threads up to 32, the number of threads with hardware support through Hyper-Threading, does not appear to improve the performance of GraphX, GraphMat, or PGX. PowerGraph benefits most from the additional 16 threads; it achieves an additional 1.5x speedup on BFS. The maximum speedup obtained by each platform is summarized in Table 5.6. Overall, PowerGraph scales best with a maximum speedup of 12.5.

5.4.4. STRONG HORIZONTAL SCALABILITY

We use Graphalytics to run BFS and PR for all distributed platforms on D1000(XL) while increasing the number of machines from 1 to 16 in powers of 2 to measure strong scalability. Key findings:

Table 5.6: Vertical scalability: speedup on D300(L) for 1–32 threads on 1 machine.)

Alg.	Giraph	GraphX	P'Graph	G'Mat(S)	OpenG	PGX(S)
BFS	5.6	4.6	12.5	7.2	6.9	9.5
PR	8.5	3.1	10.5	11.2	6.3	7.7

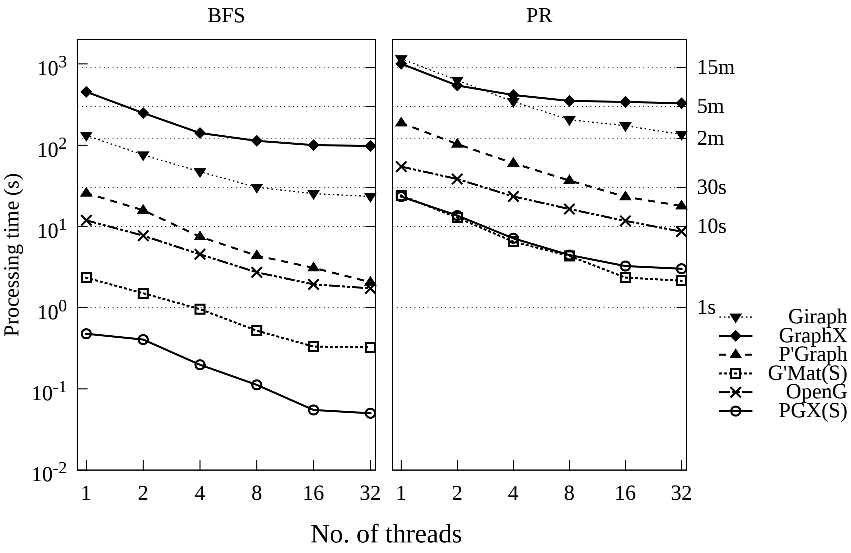


Figure 5.4: Vertical scalability: T_{proc} vs. #threads.

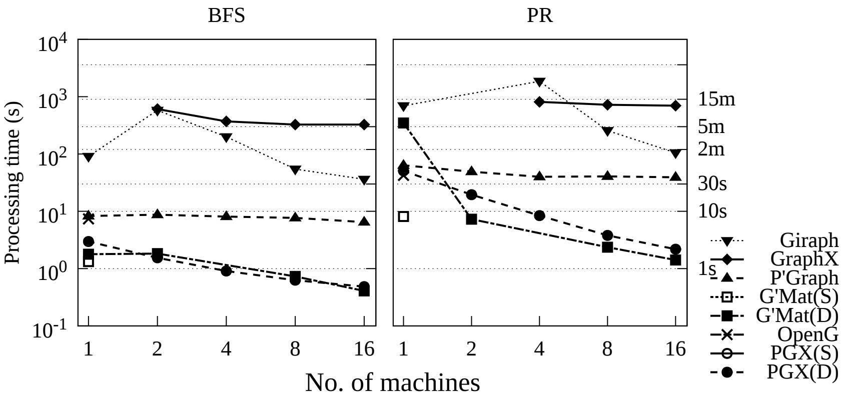


Figure 5.5: Strong scalability: T_{proc} vs. #machines.

- PGX and GraphMat show a reasonable speedup.
- Giraph's performance degrades significantly when switching from 1 machine to 2 machines, but improves significantly with additional resources.
- PowerGraph and GraphX scale poorly; GraphX shows no performance increase past 4 machines.

The processing times for this experiment are depicted in Figure 5.5. Ideally, T_{proc} halves when the amount of resources (i.e., the number of machines) is doubled given a constant workload. Giraph suffers a large performance hit when switching from 1 machine to a distributed setup with 2 machines. For PR, this results in an SLA failure on 2 machines, even though it succeeds on 1 machine. At least 4–8 machines are required for Giraph to improve in performance over the single-machine setup. GraphX also scales poorly for the given workload. It requires 2 machines to complete BFS, and 4 machines to complete PR. GraphX achieves a speedup of 1.9 using 8 times as many resources on BFS, and a speedup of 1.2 with 4 times as many resources on PR. PowerGraph is able to process the D1000(XL) graph on any number of nodes, but scales poorly for both BFS and PR. Both PGX and GraphMat show significant speedup. However, for PR both platforms show super-linear speedup when using 2 machines, possibly due to resource limitations on a single machine. In our environment, GraphMat crashed on 4 machines due to an unresolved issue in the used MPI implementation.

For comparison, results for the single-machine backends are included. Distributed GraphMat on two machines performs on-par with the single-machine backend. PGX shared memory was unable to complete either algorithm due to memory limitations.

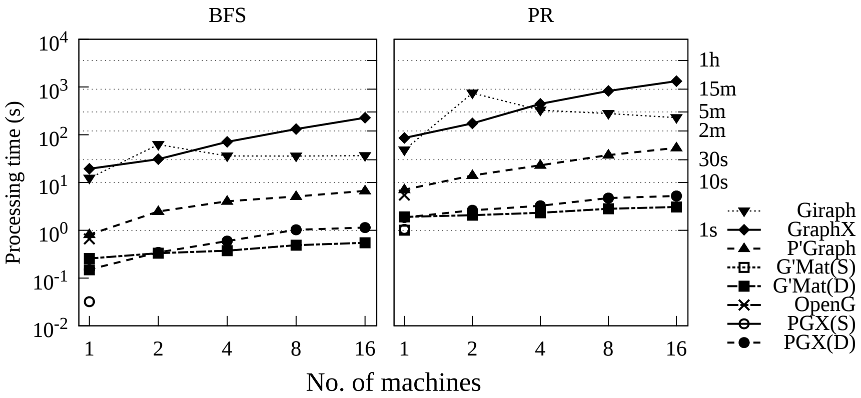


Figure 5.6: Weak scalability: T_{proc} vs. #machines.

5.4.5. Weak Horizontal Scalability

To measure weak scalability, Graphalytics runs BFS and PR for all distributed platforms on Graph500 G22(S) through G26(XL) while increasing the number of machines from 1 to 16 in powers of 2. The amount of work per machine is approximately constant, as each graph in the series generated using Graph500 is twice as large as the previous graph. As the workload per machine is constant, T_{proc} is ideally constant. Key findings:

- None of the tested platforms achieve optimal weak scalability.
- Giraph's performance degrades significantly on 2 machines, but scales well from 4 to 16 machines.
- GraphX and PowerGraph scale poorly, whereas GraphMat scales best.

In Figure 5.6, GraphX and PowerGraph show increasing processing times as the number of machines increases, peaking at a maximum slowdown (i.e., inverse of speedup) of 15.5 and 8.2, respectively. Similar to the strong scalability experiments, Giraph's performance is worst with two machines and shows a slowdown of 15.5 on PR. Performance improves slightly as more machines are added, for a slowdown of 4.7 with 16 machines on PR. GraphMat shows the best scalability with a maximum slowdown of 2.1. Although PGX outperforms GraphMat on a single machine, GraphMat's better scalability allows it outperform PGX when using 2 or more machines.

5.4.6. Stress Test

To test the maximum processing capacity of each platform, we use Graphalytics to run the BFS algorithm on all datasets, and report the scale of the smallest dataset that breaks the SLA (Section 3.4) on a single machine. Key findings:

- GraphX and PGX fail to process the largest class L graph on a single machine.
- Most platforms fail on a Graph500 graph, but succeed on a Datagen graph of comparable scale. This indicates sensitivity to graph characteristics other than graph size.
- PowerGraph and OpenG can process the largest graphs on a single machine, up to scale 9.0.

Table 5.7 lists the smallest graph, by scale, for which each platform fails to complete. The results show that both GraphX and PGX are unable to complete the BFS algorithm on Graph500 scale 25, a class L graph. PGX is specifically optimized for machines with large amount of cores and memory, and thus exceeds the memory capacity of our machines. Like GraphX and PGX, Giraph and GraphMat fail on a Graph500 graph. Both platforms successfully process D1000 with scale 9.0, but fail on G26 of the same scale. This suggests that characteristics of the graphs affect the performance of graph analysis platforms, an issue not revealed by the Graph500 benchmark. Finally, PowerGraph and OpenG fail to complete BFS on the Friendster graph, a scale 9.3 graph and among the largest graphs currently used by Graphalytics.

Table 5.7: Stress Test: the smallest dataset that failed to complete BFS successfully on one machine.

Platform	Giraph	GraphX	P'graph	G'Mat(S)	OpenG	PGX(S)
Dataset	G26(XL)	G25(L)	R5(XL)	G26(XL)	R5(XL)	G25(L)
Scale	9.0	8.7	9.3	9.0	9.3	8.7

Table 5.8: Variablity: T_{proc} mean and coefficient of variation. BFS on 1 (S) and 16 (D) nodes, $n = 10$.

		Giraph	GraphX	P'graph	G'Mat	OpenG	PGX
S	Mean	22.3s	101.5s	2.1s	0.4s	2.0s	49ms
	CV	5.0%	2.6%	1.5%	13.6%	5.8%	8.6%
D	Mean	36.4s	326.9s	6.6s	0.4s	-	0.6s
	CV	8.0%	4.3%	3.3%	4.2%	-	28.5%

5.4.7. Variability

To test the variability in performance of each platform, Graphalytics runs BFS 10 times on D300(L) with 1 machine for all platforms, and on D1000(XL) with 16 machines for the distributed platforms.

- Most platforms have a CV of at most 10%, i.e., their standard deviation is at most 10% of the mean T_{proc}.
- GraphMat (S) and PGX show higher than average variability. However, due to their much smaller mean, the absolute variability is small.

The mean and CV for T_{proc} are reported in Table 5.8. In both S and D configurations, PowerGraph shows the least variability in performance. GraphX has similarly low variability, but due to its significantly longer mean processing time it can deviate by tens of seconds between runs. Conversely, GraphMat and PGX show much larger variability between runs, but in absolute values their deviation is limited to tens of milliseconds.

5.5. Grade10 Real-World Experiments

In this section, we evaluate individually Grade10's three performance analysis steps: resource attribution, bottleneck identification, and performance issue identification.

Key findings from our real-world experiments:

1. Overall, the automated resource attribution process of Grade10, when equipped with all its features and tuned incrementally by an expert to a reasonable execution model, can match closely manual expert-level analysis of resource utilization in distributed graph-processing platforms. Grade10 automates this complex and tedious task.

2. The Grade10 mechanism, based on predicates, to identify and combine bottlenecks at different levels of the system offers fine-grained control to the expert analyst. By defining custom predicates, users can control when bottlenecks in low-level system components are flagged as a problem by Grade10, and gain new insights.

3. Grade10's performance issue identification process is able to identify a diverse set of performance issues, which differ across the different graph analysis systems we have tested.

5.5.1. Resource Attribution

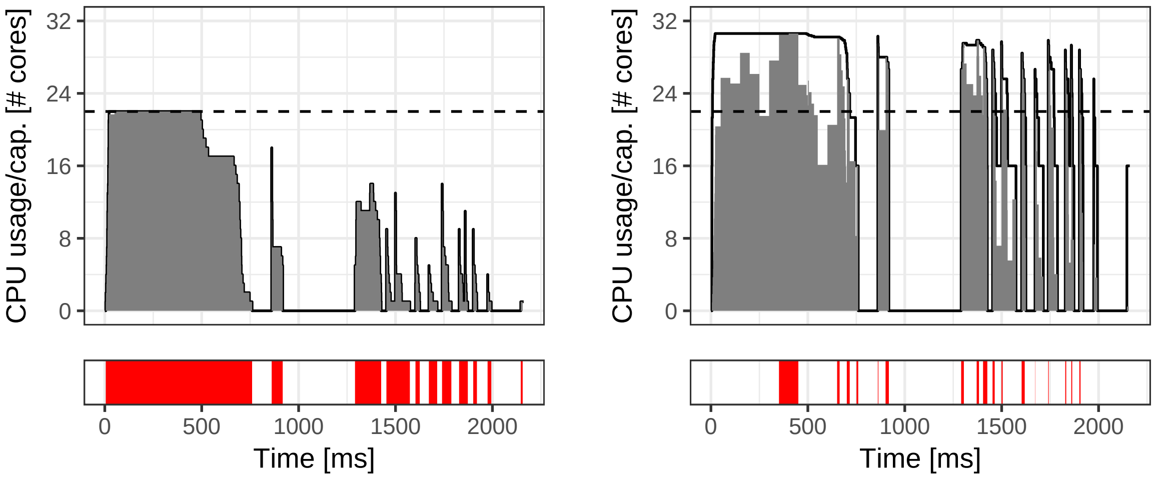
The accuracy of Grade10's resource attribution depends on the configuration of the resource attribution process, and the quality of the execution and resource models. In this section, we vary three inputs to Grade10's resource attribution algorithm to demonstrate and quantify their impact on accuracy: the use of attribution rules, the inclusion of blocking resources in the resource model, and the resource usage sampling technique.

Key findings from our real-world experiments with the Grade10 resource attribution process:

1. All three rules used by Grade10's resource attribution process are contributing to increase the accuracy of the process. This suggests a reason why non-expert users have such a difficult time understanding the reasons that lead to performance bottlenecks.

2. By modeling blocking resources, Grade10 is able to discover previously ignored performance bottlenecks.

3. By using its phase-aware sampling method, Grade10 can better approximate the real utilization of resources, but it is more sensitive to inaccuracies in the execution model (manually created by experts).



(a) Configuration G/S/N: using the full set of Greedy, Sink, and None attribution rules.

(b) Configuration S/N: using only the Sink and None attribution rules.

Figure 5.7: Outcome of Grade10's resource attribution process for an exemplary *Compute* phase. Top subplots: CPU usage (shaded area) and available CPU capacity (curve), over time. Dashed horizontal line indicates the total number of compute threads. Bottom subplots: presence of bottlenecks (shaded if present), over time. The top and bottom sub-plots use the same horizontal axis, that is, the time axes are aligned.

5.5.1.1. IMPACT OF ATTRIBUTION RULES ON ACCURACY

We compare the outcome of Grade10's resource attribution process when using all resource attribution rules (G/S/N, defined in Section 4.4.2) and when excluding the Greedy rule (S/N). To this end, we first present selected results from one *Compute* phase of a Giraph job running PageRank on Datagen-300. Figure 5.7 depicts the attributed CPU usage, the available CPU capacity, and the derived CPU bottlenecks for this phase. The Giraph job is configured to use 22 threads, which leads to a limit of 22 cores being used at any time by a *Compute* phase as indicated by a dashed line in the figure. Throughout the phase, compute threads fill up a message queue and frequently have to wait when the message queue is full. As the number of messages in the queue drops below some threshold, all waiting compute threads resume their tasks until they fill up the message queue again or complete. This results in the bursty CPU usage observed in the figure.

We first show that Grade10 can achieve high accuracy when using all of its attribution rules (that is, G, S, and N). If all resource attribution rules are used (see Figure 5.7a), the CPU capacity available for computation is correctly limited to 22 cores. The available capacity frequently drops below 22 cores to match the logical limit of 1 core per active thread. The CPU usage attributed to all active threads is close or equal to the available capacity for most of the *Compute* phase's duration: for 99.6% of the time during which at least one thread was active, a CPU bottleneck was detected (i.e., attributed usage was at least 95% of the available capacity).

In Figure 5.7b, we analyze the same *Compute* phase using only the Sink and None rules by replacing all Greedy rules with Sinks. Because the Sink rule does not impose a limit on the resource capacity available to a phase, the assigned capacity and attributed usage frequently exceed the logical limit of 22 cores. In addition, the available capacity consistently exceeds the limit of 1 core per active thread by a factor of 1.4 – 16. As a result, CPU bottlenecks are detected during a smaller portion of the *Compute* phase: only 20.2% of the time during which at least one thread was active.

Using any other subset of attribution rules (not depicted here) results in similar inaccuracies. For example, we observe that excluding the None rule results in a non-zero fraction of network utilization being attributed to *Compute* phases. This is reflected in an overestimation of network utilization for *Compute* phases and an underestimation for *Communicate* phases. When excluding the Sink rule, we observe either overestimation (when replacing Sink rules with Greedy rules) or underestimation (when replacing with None rules). We conclude that Grade10 achieves the highest accuracy when using all of its attribution rules.

To quantify the inaccuracy caused by excluding the Greedy rule, we systematically compare the attributed CPU usage for all *Compute* phases in the same Giraph job as summarized in Table 5.9. For this class of distributed systems, it remains an open research challenge to find the ground truth (baseline). We use in this work the G/S/N configuration as a baseline, that is, we consider for every sample that the usage and available capacity computed though the combined G/S/N rules represent the correct values. When comparing with another method, any value different from the baseline is analyzed as either underestimation or overestimation. We compare the CPU usage attributed by the S/N configuration with the baseline (Table 5.9a). S/N overestimates CPU usage for 60.6% of samples. When overestimating, S/N attributes an additional 9.40 core-ms on average, a significant increase compared to the average value attributed by the baseline, which is only 6.04 core-ms. Underestimations occur for only 3.9% of samples, resulting in a net overestimation consistent with the results presented in Figure 5.7. In total, S/N attributes 1526 core-seconds of CPU usage to *Compute* phases, a 94% increase over the 787 core-seconds attributed by the baseline.

Additionally, we define over- and underestimation events as contiguous sequences of over- and underestimated samples, respectively, and summarize them by duration (Table 5.9b) and area (duration times average magnitude, Table 5.9c). The frequency and magnitude of over- and underestimation events we have observed require further investigation to fully understand. This investigation falls outside the scope of this work.

5.5.1.2. IMPACT OF MODELING BLOCKING EVENTS ON ACCURACY

We compare the impact on the accuracy of Grade10's resource attribution process of modeling blocking resources (Grade10) vs. not modeling such resources (other performance tools in this space). We first present results from the *Compute* phase presented in Section 5.5.1.1 and the *Communicate* phase executed in parallel on the same machine. Figure 5.8 depicts the outcome of attributing CPU usage to both phases, for two configurations: a Giraph model including garbage collection and message queues as blocking resources (labeled *B* and presented in Figures 5.8a and 5.8c, for the *Compute* and *Communicate* phases, respectively), and a Giraph model without blocking resources (*NB*/No Blocking, Figures 5.8b and 5.8d, for the *Compute* and *Communicate* phases, respectively).

We first show that Grade10 achieves higher accuracy by modeling blocking phases. When blocking resources are modeled, the CPU capacity available for computation frequently drops below the limit of 22 as threads are blocked on full message queues or a garbage collection event. Without blocking resources, the available CPU capacity remains at 22 cores until the first threads finish. This results in frequent overestimation of CPU usage in configuration NB as the attributed usage exceeds the logical limit of 1 core per active thread. The presence or absence of blocking resources for one phase can also impact concurrent phases. When compute threads block on full message queues, the CPU capacity that is not used for computation becomes available for the *Communication* phase, a phase that is modeled as a Sink. Conversely, when garbage collection events occur, both computation and communication block and neither phase has access to the CPU. When blocking resources are not modeled, communication is limited to 10 cores (i.e., 32 cores present on the host minus 22 cores reserved for computation) until the first compute threads finish. CPU usage attributed to communication is significantly lower for configuration NB, due to the overestimation of CPU usage for computation: 8.3 vs. 2.2 cores for Giraph models with/without blocking resources, respectively.

We also show that not modeling blocking resources results in both false positives and false negatives in Grade10's subsequent bottleneck identification step. For example, in Figure 5.8b around 950 – 1250 ms, no compute threads are active due to a garbage collection event, but Grade10 falsely attributes a CPU bottleneck

Table 5.9: Analysis of over- and underestimation of CPU usage across *Compute* phase instances when using the Sink and None resource attribution rules. Total samples = 130,289, total CPU usage in baseline = 787,216.7 core-ms, time per sample = 1 ms.

(a) Magnitude of over- and underestimated resource usage samples in core-ms.

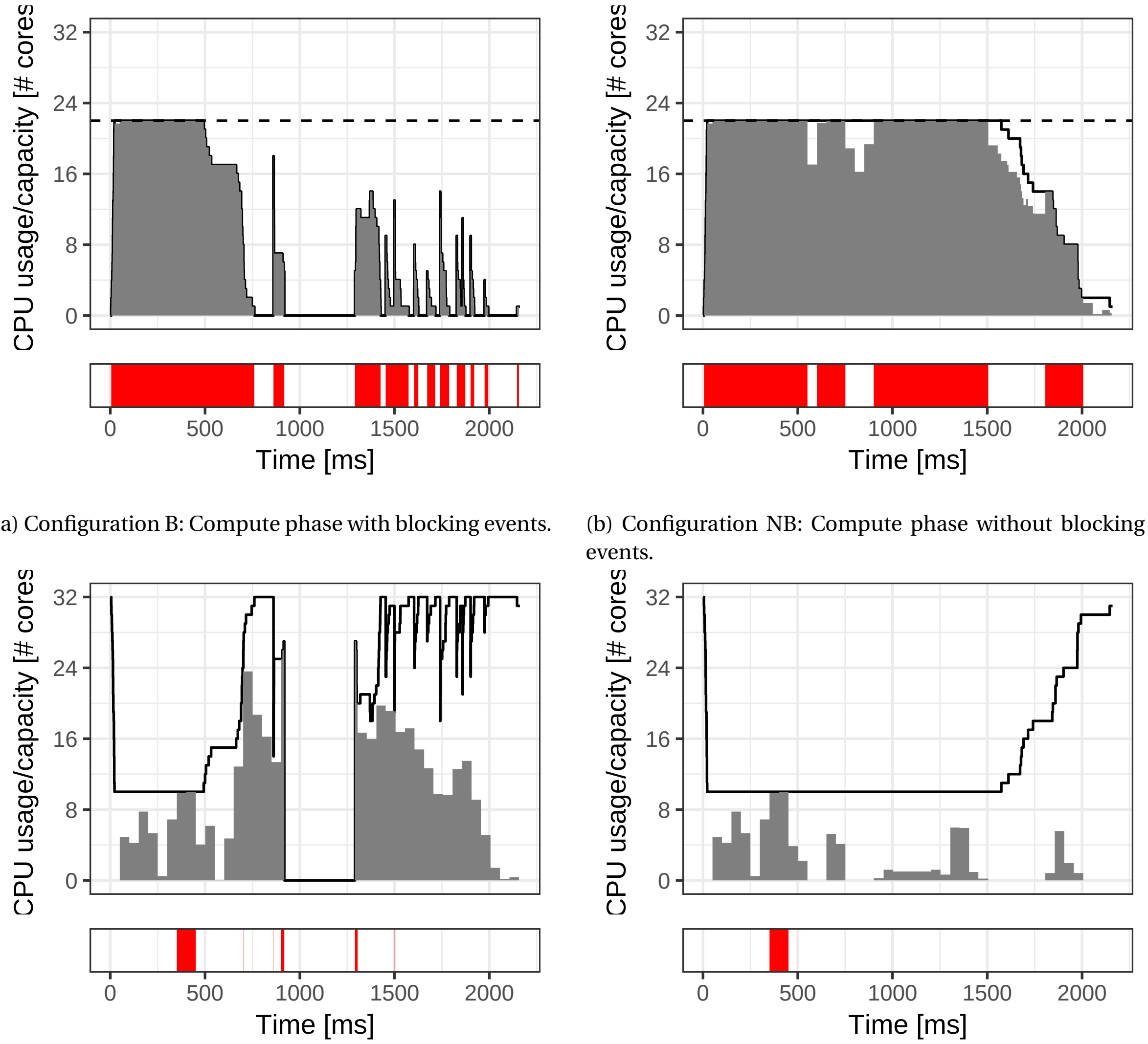
	N	Sum	Mean	SD	CV	Min	Q_1	Med.	Q_3	Max
Over	78,999	742,937.03	9.40	5.76	0.61	0.00	4.81	8.61	13.38	21.67
Under	5,089	4,181.00	0.82	0.88	1.07	0.00	0.32	0.64	0.96	8.54

(b) Duration of over- and underestimated resource usage events.

	N	Sum	Mean	SD	CV	Min	Q_1	Med.	Q_3	Max
Over	1,053	78,999	75.02	114.94	1.53	1	17.00	36.00	68.00	852
Under	112	5,089	45.44	37.61	0.83	1	15.00	48.50	58.25	211

(c) Area of over- and underestimated resource usage events.

	N	Sum	Mean	SD	CV	Min	Q_1	Med.	Q_3	Max
Over	1,053	742,937.03	705.54	1,061.55	1.50	0.32	132.85	345.15	694.46	6,937.11
Under	112	4,181.00	37.33	28.83	0.77	0.22	13.06	36.34	53.88	110.99



(a) Configuration B: Compute phase with blocking events.

(b) Configuration NB: Compute phase without blocking events.

(c) Configuration B: Communicate phase with blocking events.

(d) Configuration NB: Communicate phase without blocking events.

Figure 5.8: Outcome of Grade10's resource attribution process for an exemplary *Compute* phase and a concurrent *Communicate* phase. Top subplots: CPU usage (shaded area) and available CPU capacity (curve). Bottom subplots: presence of bottlenecks (shaded if present).

to the *Compute* phase when the garbage collector is not modeled. A false negative occurs in Figure 5.8b around 900 ms, when a small number of threads is active, but incorrectly not labeled as bottlenecked on the CPU. Configuration NB attributes 16 – 20 cores of CPU usage, so Grade10 does not identify a bottleneck even though only 7 threads were active.

To quantify the inaccuracy caused by not modeling blocking resources, we apply the analysis introduced in Experiment 1; we compare the attributed CPU usage for all *Compute* and *Communicate* phases using configuration B as a baseline. Table 5.10 summarizes the over- and underestimation of samples by configuration NB. We observe that in this configuration, Grade10 overestimates CPU usage by *Compute* phases for the majority of samples (81.7%). In total, it attributed 176% higher CPU usage when not modeling blocking resources. For *Communicate* phases, configuration NB underestimates 32.4% of samples overestimates 8.0%. However, it underestimates by a larger magnitude on average as a result of the significant overestimation for *Compute* phases. The overestimated samples are primarily the result of a non-zero attribution of CPU usage during garbage collection events. We conclude that the inaccuracies observed in Figure 5.8 are validated by the systematic analysis across many phases.

Table 5.10: Analysis of over- and underestimation of CPU usage across *Compute* and *Communicate* phase instances when not modeling blocking events. Values in the table depict the number and magnitude in core-ms of over- and underestimated resource usage samples.

(a) Summary of *Compute* phase instances. Total samples = 130,289, total CPU usage in baseline = 787,216.7 core-ms, time per sample = 1 ms.

	N	Sum	Mean	SD	CV	Min	Q_1	Med.	Q_3	Max
Over	106,476	1,383,180.53	12.99	5.91	0.45	0.01	8.52	13.00	17.42	22.00
Under	653	1,422.54	2.18	2.66	1.22	0.00	0.54	1.07	2.75	19.00

(b) Summary of *Communicate* phase instances. Total samples = 285,899, total CPU usage in baseline = 1,271,327 core-ms, time per sample = 1 ms.

	N	Sum	Mean	SD	CV	Min	Q_1	Med.	Q_3	Max
Over	22,748	97,530.58	4.29	4.99	1.16	0.02	1.00	1.23	6.05	22.17
Under	92,621	992,065.76	10.71	5.12	0.48	0.02	7.16	11.15	14.46	32.00

5.5.1.3. Impact of Sampling Method on Accuracy

We compare the impact of the resource sampling step on the accuracy of Grade10's resource attribution process. In particular, we compare two sampling methods: constant interpolation using a left-continuous step (referred to as *traditional*) and *phase-aware* as defined in Section 4.4.2. These sampling methods produce high-frequency samples (e.g., one per millisecond in our prototype) from low-frequency input metrics (e.g., one measurement per 50 milliseconds for our experiments). As such, a ground truth per sample is not available. To compare the sampling methods, we use instead the average utilization during a measurement as the baseline, and compare the average of all samples produced by either method with the baseline for each measurement interval. Next, we apply both sampling methods to downsampled, lower-frequency metrics and compare the results against the original, higher-frequency baseline to assess how closely each sampling method approximates the baseline. We downsample the stream of measurements corresponding to a metric by computing the average of k consecutive measurements, to derive a new stream of measurements with k times lower frequency.

We first present selected results from the *Compute* and *Communicate* phases presented in the experiments of Sections 5.5.1.1 and 5.5.1.2. Figure 5.9 depicts the CPU usage samples produced by both sampling methods from a baseline input metric measured approximately once per 50ms. The average magnitude of samples during a single measurement interval is equal to the magnitude of the corresponding measurement for both methods, so they achieve the same 100% accuracy compared to the baseline. When using the *traditional* method, the combined CPU usage of the *Compute* and *Communicate* phases remains constant for extended periods of time because this method outputs samples of equal magnitude throughout a measurement interval. As the CPU load (and attributed usage) for computation decreases, the CPU usage for communication increases, and vice versa. The *phase-aware sampling method* produces samples that result in the same varying CPU usage for computation, but a constant CPU usage for communication throughout a measurement interval. As a result, the combined CPU usage can fluctuate significantly within a measurement

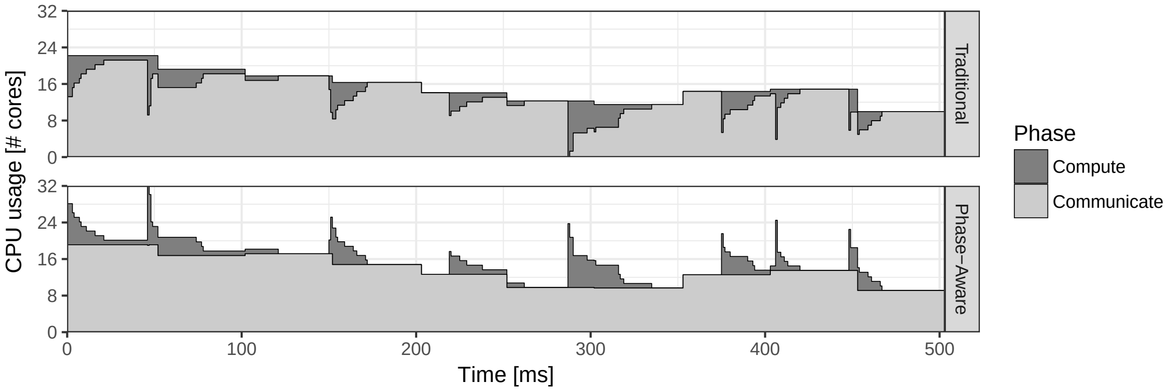


Figure 5.9: CPU usage attributed to a *Compute* and a *Communicate* phase using traditional and phase-aware sampling methods. Sample frequency: 1ms per sample. Measurement frequency: approximately 50ms per measurement.

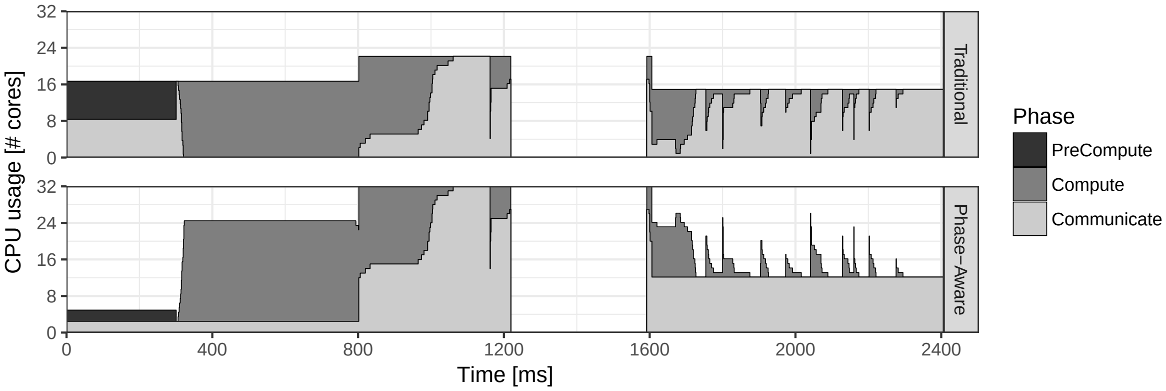


Figure 5.10: CPU usage attributed to a *PreCompute*, a *Compute*, and a *Communicate* phase using traditional and phase-aware sampling methods. Sample frequency: 1ms per sample. Measurement frequency: approximately 800ms per measurement, downsampled by 16x from the original measurement at 50ms intervals.

interval. Without additional information about the internals of the *Communication* phase, we assume, as experts do, that it is unlikely for the CPU usage for communication to increase whenever the CPU usage for computation decreases proportionally. We conclude that both sampling methods achieve the same accuracy when averaging samples during a measurement interval, but that the phase-aware sampling likely results in higher accuracy for individual samples.

In Figure 5.10, we present the CPU usage samples produced by both sampling methods from a derived metric for three measurements, approximately one per 800ms. The first measurement interval captures the end of a *PreCompute* phase and the start of a *Compute* phase. When using the traditional sampling method, Grade10 initially attributes more CPU usage to both the *PreCompute* phase and *Communicate* phase compared to the phase-aware method. It also attributed less CPU usage to the *Compute* phase. As a result, the CPU usage for computation is 16.7 cores at all times during the interval, despite 22 CPU-intensive compute threads being active throughout most of the interval. The phase-aware method produces samples of larger magnitude when the *Compute* phase is active and attributes the majority of CPU usage during the first measurement interval to that phase. The second measurement interval includes a garbage collection (GC) event. The traditional method produces samples of equal magnitude throughout this interval, but the samples covering the GC event are not attributed to any phase and are thus not depicted. The phase-aware method assumes there is no CPU load during the GC event, because there are no active phases. Thus, it distributes the total CPU usage during the interval over a small period and produces samples of larger magnitude before and after the GC event. In reality, garbage collection not only blocks ongoing phases, but also uses CPU resources to complete its task. This is not modeled in the prototype execution model for Giraph, so the phase-aware method makes incorrect assumptions about the CPU load and consequently produces (likely) incorrect samples. The third measurement interval mirrors the behavior identified in Figure 5.9. We conclude that: (1) Grade10 produces more accurate results using the phase-aware sampling method, but (2) the phase-aware sampling method is sensitive to incomplete execution models. Concerning point (2), for this use-case, we ex-

Table 5.11: Average relative sampling error produced by two sampling methods for a variety of platform configurations and input metrics.

		Measurement Interval						
Platform & Model	**Sampling Method**	**50 ms**	**100 ms**	**200 ms**	**400 ms**	**800 ms**	**1600 ms**	**3200 ms**
Giraph	Traditional	0.00%	12.03%	20.43%	31.57%	47.32%	68.52%	82.97%
(not tuned)	Phase-Aware	0.03%	15.24%	26.89%	41.26%	58.30%	80.78%	91.02%
Giraph (GC)	Traditional	0.00%	12.03%	20.43%	31.57%	47.32%	68.52%	82.97%
(tuned model)	Phase-Aware	0.03%	7.37%	11.91%	18.83%	29.75%	46.37%	56.71%
PowerGraph	Traditional	0.00%	32.82%	58.97%	81.95%	94.94%	97.96%	98.71%
(tuned model)	Phase-Aware	0.12%	7.36%	11.27%	11.62%	11.87%	13.23%	15.28%

pect all experts in graph analysis platforms built using Java technology to be aware of the operation of the Garbage Collector, and tune the Grade10 model of their system accordingly.

To analyze the accuracy of both sampling methods systematically, we present in Table 5.11 results for a job running PageRank on Datagen-300 on two platforms (Giraph and PowerGraph) using two sampling methods and various measurement intervals. For Giraph we present results using two execution models: the prototype model (Giraph), and an extension, matching the expected behavior of an expert modeler, that models garbage collection as a CPU-intensive phase to address the inaccuracies identified in Figure 5.10 (Giraph (GC)). For each configuration, CPU metric, and measurement period, we compute the sampling error as absolute difference between the average sample and the baseline. In the table, we present per configuration the average relative sampling error, i.e., the average error per sample as a percentage of the average sample in the baseline.

We note that for a measurement interval of 50ms (i.e., the baseline metrics) the traditional sampling method produces no error by definition; each sample is equal to the baseline measurement. For the same interval, the phase-aware sampling method produces a small error during the first and last measurement period if measurements do not start and end precisely when the job starts and ends. For larger measurement intervals, the traditional sampling method becomes increasingly more inaccurate up to an average error of 83-99% on inputs with a measurement interval of 3,200ms. The phase-aware sampling method performs worse than the traditional method on Giraph using the prototype execution model, but it performs better with the tuned execution model with an error of up to 57%. The remaining error can likely be further reduced by extending the execution model to model large components of Giraph in more detail, e.g., the *Communicate* and *LoadGraph* phases. Investigating this further is outside the scope of this work. For PowerGraph, the phase-aware sampling method is significantly more accurate: its error is up to 8 times lower than the traditional method. This result can be attributed to PowerGraph's more comprehensive Grade10 model (and the exclusion of stages that are not modeled at all, such as graph loading). We conclude that with a comprehensive execution model, the phase-aware sampling method significantly outperforms the traditional method.

5.5.2. Bottleneck Identification

Grade10's process for bottleneck identification, as described in Section 4.4.3, consists of two steps: identifying bottlenecks in leaf phases based on resource utilization, and identifying bottlenecks in composite phases by deriving them from bottlenecks identified in subphases. The first step uses a simple threshold-based approach and is not evaluated experimentally in this work. The second step uses a set of predicates selected by the user to specify how bottlenecks identified in subphases are combined. In this section, we analyze the impact of the selected predicate on identified bottlenecks.

Key findings from our real-world experiments with the Grade10 bottleneck identification process:

1. Grade10 provides the controls for analysts to define useful predicates. This enables users to tune when low-level bottlenecks should be flagged by Grade10 as bottlenecks in higher-level phases.

2. By combining different predicates, e.g., Maj and Any, the Grade10 bottleneck identification process can reveal nuanced analysis, automatically and in tractable execution time and memory footprint. This new capability could enable experts to conduct more in-depth analysis of graph processing systems. We are discussing with the production teams at Oracle and Intel about possible use in their practice.

5.5.2.1. Impact of Predicates on Identified Bottlenecks

Grade10's process for bottleneck identification uses the bottlenecks identified in leaf phases to find bottlenecks in composite phases. To this end, Grade10 allows users to select predicates that combine bottlenecks identified in subphases to determine whether their parent phase is also bottlenecked. The choice of predicate significantly impacts the bottlenecks that are identified by Grade10 in higher levels of an execution model. To analyze the impact of predicates on identified bottlenecks and to demonstrate the control Grade10 provide to users, we apply the predicates defined in Section 4.4.3 to an exemplary Giraph job.

We present in Figure 5.11 the result of applying Grade10's predefined bottleneck predicates[1] to a list of bottlenecks obtained from a Giraph job. We observe large differences between the bottlenecks identified by each predicate. When using the Any predicate, Grade10 identifies bottlenecks on at least one resource for 87% of the output phase's duration, compared to 62% for Majority and 34% for All. Because few garbage

[1] We exclude the combined Majority+Any predicate from this comparison, because it behaves like Majority for the given input in this experiment.

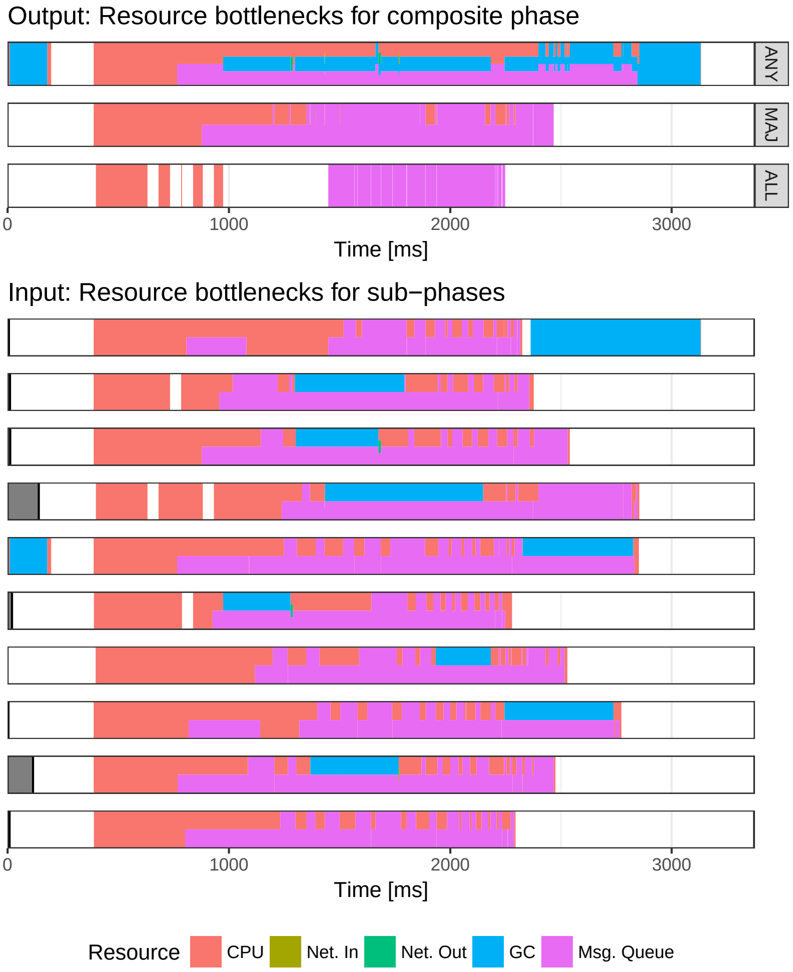


Figure 5.11: Exemplary bottleneck identification result obtained by applying three predicates (output plots) to bottlenecks identified in a collection of subphases (input plots). The horizontal axis represents time since the start of the output phase. For outputs and inputs, any area left blank (white) indicates a lack of bottlenecks during a period of time. For inputs, gray shaded areas at the left or right of the plot indicate time before the start or after the end of the given subphase, respectively. Areas shaded with a different color (i.e., not white and not gray) indicate the presence of a bottleneck on the resource corresponding to that color at a specific point in time. The vertical axis *does not* indicate any property of the bottlenecks identified at a given point in time; when multiple bottlenecks occur simultaneously, they are depicted by their respective colors and each given an equal share of vertical space in the plot. For example, at time = 2,000 ms in the top output plot, the output phase is bottlenecked simultaneously on `CPU`, `GC`, and `Msg. Queue`.

collection events overlap in the depicted subphases, the Majority and All predicates do not identify any GC bottlenecks. In contrast, when using the Any predicate, Grade10 indicates the composite (output) phase is bottlenecked on GC bottlenecks during the majority of its duration. GC events can cause significant slowdown in distributed systems if not coordinated[99, 100]. Thus, using the Any predicate, Grade10 can help detect problematic GC events, especially in the common situation when the GC events are spread out over time due to a lack of coordination between the distributed machines; this type of event is not under the control of the developer of the graph analysis system, so it can happen often in practice without a mechanism to prevent it. Alternatively, users could define a custom predicate that identifies when bottlenecks occur in several subphases, e.g., using a custom threshold, to be less sensitive to outliers than Any, yet identify more bottlenecks than Majority or All can.

In Table 5.12, we summarize the result of equipping Grade10 with one of the four different predicates, and then using Grade10 to analyze bottlenecks in a Giraph job. We present for several phases the total duration of the bottlenecks identified by Grade10, as a percentage of the total duration of those phases. Overall, the All predicate does not identify any bottleneck throughout the Giraph job, whereas the Any predicate identifies bottlenecks on at least one resource for 43.75% of the job duration. The Any predicate reveals that CPU, garbage collection, and message queues each affect at least one component in the platform for 24.91-35.48%

of time. However, the Majority predicate indicates that only message queue bottlenecks occur simultaneously across a large number of components. The main differences between the presented predicates become apparent across the selected phases.

When combining *ComputeThread* results to identify bottlenecks in *Compute* phases, the All predicate identifies few message queue bottlenecks while the other predicates closely match each other. For CPU bottlenecks, the Any predicate is the outlier; it detects over twice as many bottlenecks as the other predicates. Results for garbage collection bottlenecks are identical across all predicates for phases up to *WorkerSuperstep*, because garbage collection events block all phases on the same machine simultaneously. The All, Majority, and Majority+Any predicates no longer detect any GC bottlenecks when combining single-machine *WorkerSupersteps* to form a distributed *Superstep*. In contrast, the Any predicate detects 5.4 times as many GC bottlenecks by duration, which is consistent with the observations in Figure 5.11. The difference between Majority and Majority+Any is most apparent for the *WorkerSuperstep* phase, which consists primarily of the *Compute* and *Communicate* phases. The Majority predicate identifies a bottleneck if and only if it occurs in the majority of subphases, which means for this example a bottleneck occurring in both the *Compute* and *Communicate* phases. Consequently, the Majority predicate detects no network bottlenecks and few CPU bottlenecks in *WorkerSupersteps*. The Majority+Any predicate behaves like Any when combining results from different phases (as opposed to instances of the same phase), so it propagates both the CPU bottlenecks of the *Compute* phase and the network bottlenecks of the *Communicate* phase.

5.5.3. Performance Issue Identification

Grade10's process for the identification of performance issues aims to provide high-level insight into the performance of a single graph-processing job. This scenario is common in practice, for example when the product team has to satisfy a specific client, or when the development team tries to focus on a particular limit-case to improve the system. In this section, we use Grade10 to identify the performance issues with the highest performance impact as experienced by the user, for a variety of jobs across platforms (Giraph and PowerGraph), algorithms (BFS, CDLP, PR, and WCC, as defined in the Graphalytics benchmark in Section 3.3.3), and graphs (Datagen-1000 and Graph500-26). Using our Grade10 prototype, we detect two types of perfor-

Table 5.12: Average percentage of time, from the total duration of the phase, during which Grade10 identified a bottleneck on the given resource, by predicate used to identify the bottleneck. For the ComputeThread and Communicate phases, the star symbol (*) denotes that the same bottlenecks are identified by each predicate.

		Resource					
Phase	**Predicate**	**CPU**	**Net. In**	**Net. Out**	**GC**	**Msg. Queue**	**None**
GiraphJob	Any	29.58%	0.54%	0.54%	35.48%	24.91%	56.25%
	All	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
	Maj	0.07%	0.00%	0.00%	0.01%	18.00%	81.92%
	Maj+Any	4.45%	0.00%	0.00%	0.01%	18.00%	77.53%
Superstep	Any	49.00%	0.45%	0.29%	45.57%	46.66%	40.37%
	All	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
	Maj	0.13%	0.00%	0.00%	0.00%	33.71%	66.16%
	Maj+Any	8.34%	0.00%	0.00%	0.00%	33.71%	57.94%
WorkerSuperstep	Any	29.44%	0.04%	0.03%	8.44%	34.97%	53.47%
	All	1.60%	0.00%	0.00%	8.44%	10.35%	81.25%
	Maj	2.66%	0.00%	0.00%	8.44%	31.17%	62.45%
	Maj+Any	11.52%	0.04%	0.03%	8.44%	31.17%	53.96%
Compute	Any	64.37%	0.00%	0.00%	16.04%	76.73%	0.48%
	All	17.47%	0.00%	0.00%	16.04%	22.70%	47.38%
	Maj	24.29%	0.00%	0.00%	16.04%	68.39%	1.65%
	Maj+Any	24.29%	0.00%	0.00%	16.04%	68.39%	1.65%
ComputeThread	*	30.61%	0.00%	0.00%	16.48%	61.87%	0.33%
Communicate	*	3.07%	0.04%	0.03%	8.44%	0.00%	88.49%

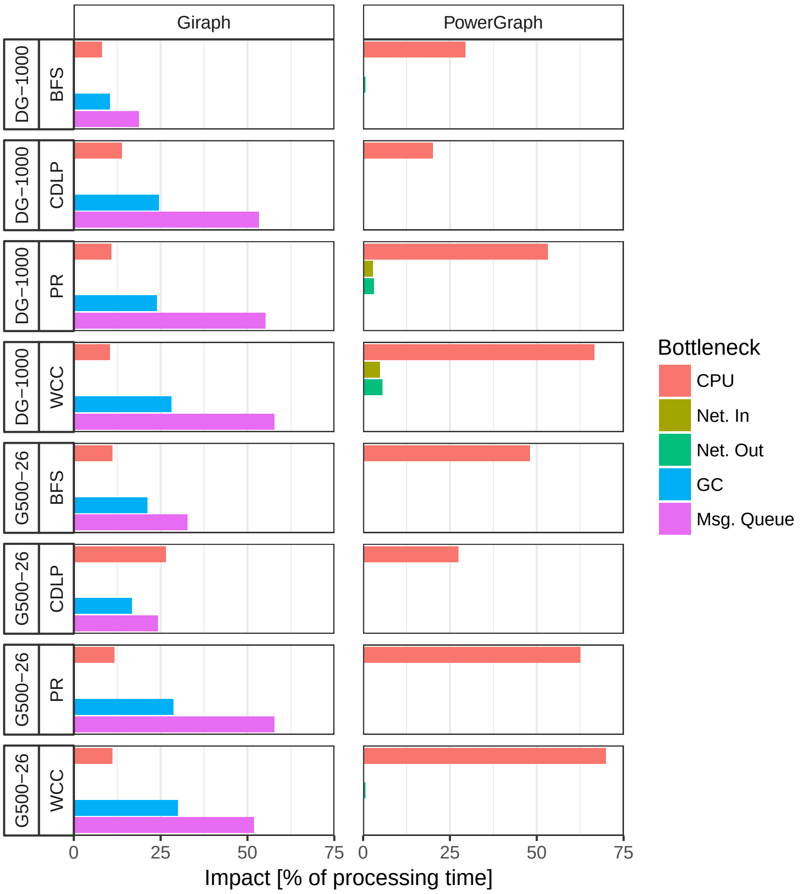


Figure 5.12: Optimistic estimation of the impact of various bottlenecks on the processing time of graph analysis jobs spanning 2 platforms, 2 datasets (`datagen-1000` and `graph600-26` as defined by Graphalytics), and 4 algorithms. The garbage collection (`GC`) and message queue (`Msg. Queue`) bottlenecks are only defined for Giraph in our experiments, but would apply to many other graph analysis systems.

mance issues: resource bottlenecks and phase imbalance. A more comprehensive study, e.g., running the full Graphalytics benchmark and identifying more types of performance issues, may provide additional insight into Giraph's and PowerGraph's performance, but is outside the scope of this work's experimental evaluation of Grade10.

Key findings from our real-world experiments with the Grade10 process for identifying performance issues:

1. Using Grade10, we found a previously undiscovered synchronization bug in PowerGraph.

2. For most Giraph jobs, waiting for full message queues to be cleared is the largest bottleneck, taking up to half of its processing time.

3. In contrast to Giraph, PowerGraph's processing time is dominated by CPU bottlenecks, but the impact varies significantly between algorithms and graphs.

4. Not only do JVM-based systems perform on average significantly worse than their C++-based commercial and community-driven counterparts (we refer back to the Graphalytics experiments, Section 5.4), but with Grade10 we find that GC causes important performance bottlenecks. We also quantify the contribution of GC to the degradation of performance.

5. The CDLP algorithm causes the largest phase imbalance on both platforms.

6. Phase imbalance in Giraph seems to be largely insensitive to graph and algorithm, with CDLP on Graph500-26 as a notable outlier.

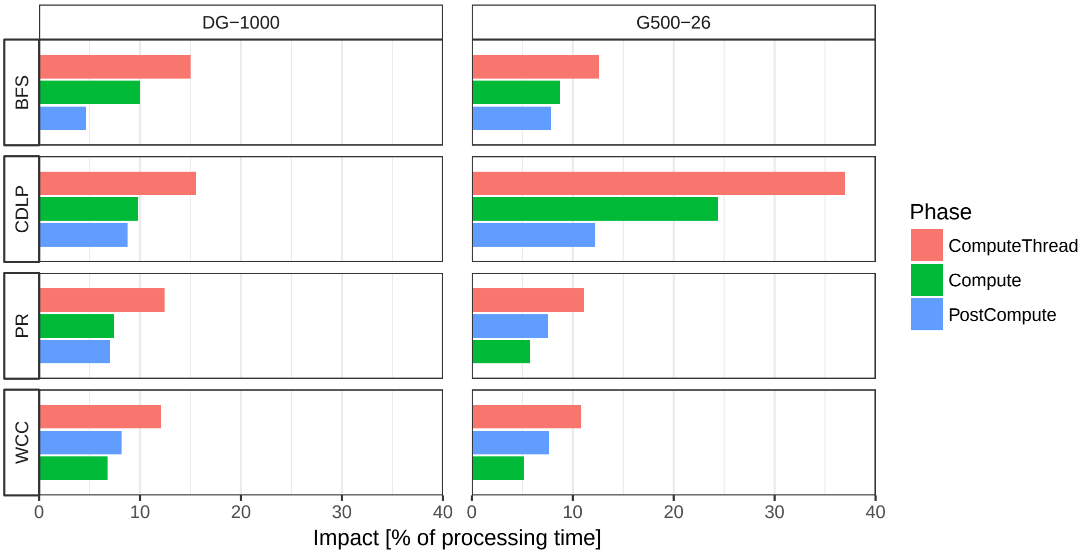


Figure 5.13: Optimistic estimation of the impact of imbalance on the processing time of Giraph jobs. The graph depicts only the top 3 phases, by performance impact, for each job.

7. PowerGraph's phase imbalance is comparable across graphs, but varies significantly between algorithms. In particular, the stages in PowerGraph's execution that are most imbalanced are different between algorithms.

5.5.3.1. Optimistic Analysis of Impact of Bottlenecks on Processing Time

We analyze the impact of resource bottlenecks on the processing time of Giraph and PowerGraph jobs. We compare the identified bottlenecks to gain insight into the specific impact of the platform, the algorithm, and the dataset on the bottlenecks that occur during a job.

Figure 5.12 depicts the estimated impact of bottlenecks on processing time (i.e., the *Execute* phase). The presented estimates are optimistic; they represent by how much the processing time could be reduced if a bottleneck is eliminated, assuming this would not introduce new bottlenecks on other resources and disregarding other existing bottlenecks. For Giraph, message queues becoming or staying full are the most impactful bottleneck for 7 out of 8 jobs, taking an unexpectedly high fraction of the processing time (19-58%). The next largest bottleneck is garbage collection at 10-30%, followed by CPU bottlenecks at 8-14% (except for an outlier at 27%). The impact of network-related bottlenecks is below 0.1% for all jobs. The lack of network-related bottlenecks confirms recent findings by Satish et al. [52], who found that Giraph achieved poor network utilization in each of their experiments.

Among the Giraph jobs included in this experiment, CDLP on Graph500-26 is a clear outlier. Its CPU bottleneck is almost twice as big as the second-most CPU-bottlenecked job and it is more impactful than the garbage collector and message queues. A possible cause for this behavior is that Giraph's CDLP implementation is more CPU-intensive and is more sensitive to changes in graph topology than other algorithms.

For PowerGraph, which is C++-based so does not include a Garbage Collector the same way as Java-based systems do, and for which the engineers have tuned the use of communication sockets, as expected CPU bottlenecks are the only major bottlenecks detected by Grade10. Their impact varies significantly across jobs between 20% and 70%. For both graphs, the order of algorithms from least to most impacted by CPU bottlenecks is consistent: CDLP, BFS, PR, and WCC. However, we have not found a clear pattern in the magnitude of the impact of CPU bottlenecks across graphs and algorithms. Unlike Giraph, PowerGraph experiences a noticeable bottleneck on network for some jobs, up to 5% and up to 6% for inbound and outbound traffic, respectively. Garbage collection and message queues are not modeled for, and/or are not used by, PowerGraph, so no bottlenecks were identified for those resources.

5.5.3.2. Optimistic Analysis of Impact of Phase Imbalance on Processing Time

We analyze the impact of phase imbalance on the processing time of Giraph and PowerGraph jobs. The outcome of Grade10's phase imbalance analysis is not directly comparable between platforms if their execution models are different. Thus, we present the results for each platform individually.

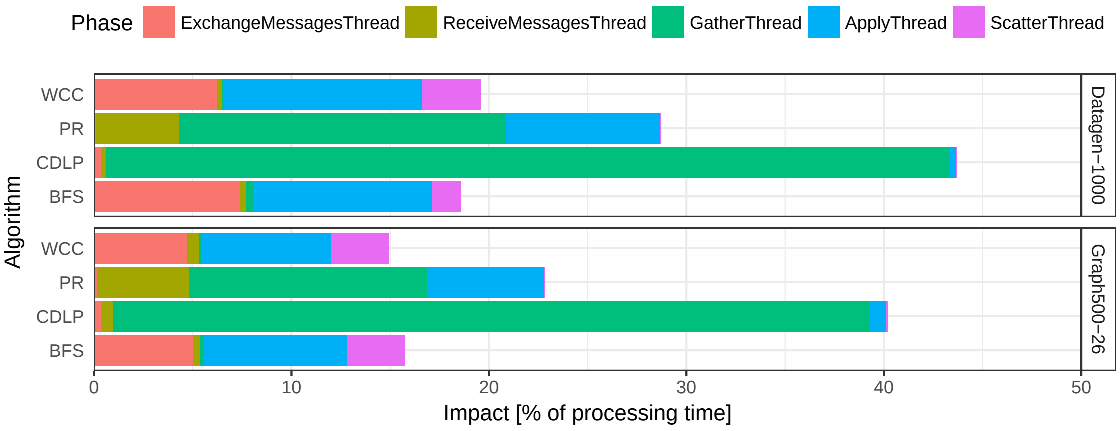


Figure 5.14: Optimistic estimation of the impact of imbalance on the processing time of PowerGraph jobs, detailed by phase.

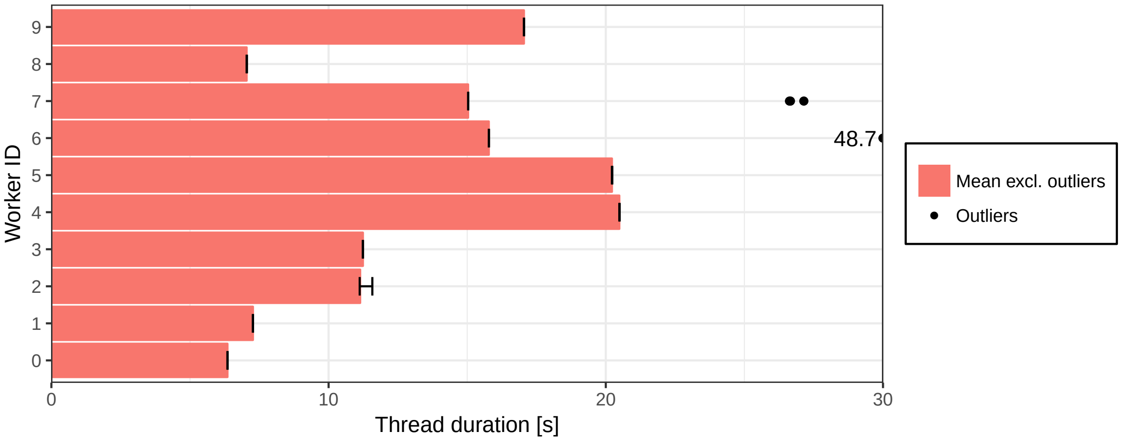


Figure 5.15: Duration of *GatherThreads* in the first superstep of PowerGraph running CDLP on Graph500-26. Outliers, defined as threads taking at least 10% longer than the median thread duration of a worker, are depicted separately. The min, mean, and max duration of all threads not labeled as outlier are depicted by the bars and error bounds.

We first present the results for Giraph, depicted in Figure 5.13. For each job, we ordered all subphases of *Execute* by impact of their imbalance, and present only the top three most impactful subphases. Imbalance across *ComputeThreads* is the most impactful for all jobs, taking 11-16% of processing time for most jobs. The combination of CDLP and Graph500-26 is again an outlier, with an impact of 37% for *ComputeThread* imbalance. *Compute* and *PostCompute* are the second and third-most imbalanced phases for all jobs, although their order is different across jobs. We note that imbalance is lower for *Compute* phases than for their constituent *ComputeThread* phases. This effect is explained through a simplified example in Section 4.4.4.1 and Figure 4.10.

Next, we present the results for PowerGraph. PowerGraph's supersteps consist of five stages, each executed synchronously across machines and threads. Figure 5.14 depicts the imbalance in each stage, represented by their corresponding *-Thread* phase. We observe a clear correlation between the used algorithm and the impact of phase imbalance on performance. Overall, phase imbalance in PowerGraph is the largest for the CDLP algorithm, with an impact of over 40% on the total processing time. BFS and WCC cause the least imbalance, with 15-20%. Furthermore, we observe that the imbalance caused by individual stages differs significantly between algorithms, but is consistent across graphs. For example, for both CDLP jobs, phase imbalance is predominantly observed in the *Gather* stage. BFS and WCC are similar in phase imbalance: the *ExchangeMessages* and *Apply* stages are responsible for the majority of imbalance.

We further investigate the PowerGraph job running CDLP on Graph500-26 to discover the root cause of the high imbalance indicated in Figure 5.14. Using Grade10 to analyze the imbalance of individual supersteps, we identify the first two supersteps as highly imbalanced (44.7% and 37.5% estimated impact, respectively). We present in Figure 5.15 the duration of *GatherThreads*, i.e., the imbalanced stage identified in Figure 5.14, during the first superstep. Overall, the duration of the Gather stage is dominated by one straggler, on Worker 6. Through manual code inspection, we learned that all Gather threads on one worker iterate over and process vertices in the local graph partition. Each thread checks periodically if there are any incoming messages in a shared message queue and processes these messages until the queue is empty. After processing all local vertices, threads synchronize using a local barrier (this marks the end of the local computation and is considered the end of the *GatherThread* in our model), then process any remaining messages in the queue, and finally synchronize across workers using a global barrier. We observed that stragglers can occur if the majority of threads (all but one, in our example) have already reached the local barrier when the last active thread(s) check the message queue. The active thread(s) will then proceed to process all incoming messages for the remainder of the superstep, while the majority of threads (and CPU cores) idle. This synchronization issue has not been reported or fixed previously, despite the significant engineering effort invested in the PowerGraph platform.

6

Conclusion and Future Work

Big data is ubiquitous in the modern, digital society. In this thesis, we have focused on graph data, an important class of big data. To address society's growing needs for analyzing large-scale graphs, academia and industry have developed a variety of graph analysis systems. To compare these systems, but also to learn how tune and re-design them, and to understand why graph analysis systems perform the way they do, we have addressed in this thesis two distinct but related challenges in understanding the performance of graph analysis. Our approach is the design and implementation of the Graphalytics ecosystem, a set of complementary systems for studying the performance of graph analysis systems, with a focus on both the breadth and the depth dimensions of understanding performance.

In this chapter, we conclude this thesis by presenting our approaches toward answering the research questions we posed in Chapter 1, and we discuss directions for future work in extending the Graphalytics ecosystem and the components presented in this thesis.

6.1. Conclusion

In this thesis, we have answered three main research questions through the combined contributions of Graphalytics, Grade10, and their experimental evaluation:

RQ1: How to systematically yet efficiently compare the performance of graph analytics frameworks?
For potentials users of graph analysis, the performance of graph analysis frameworks is an important factor in selecting the right framework for their needs. A method for systematic comparison is also important for developers of frameworks to provide fair and insightful comparison against state-of-the-art. Previous best practices included performance comparisons that were not standardized, rarely comprehensive, and potentially biased. We have designed and implemented LDBC Graphalytics, an industrial-grade benchmark for large-scale graph analysis (Chapter 3).

The Graphalytics benchmark includes multiple novel contributions. Its workload was designed using a two-stage selection process that ensures both diversity and representativeness of real-world usage of graph analysis platforms. Beyond traditional performance metrics, Graphalytics measures scalability and robustness to achieve a more comprehensive comparison of platforms than existing benchmarks. To ensure continued relevance in the evolving field of graph analysis, Graphalytics defines a process for periodically renewing the benchmark's workload.

RQ2: How to automatically analyze performance in the execution of graph analytics workloads?
Analyzing the performance of graph analytics workloads is critical to identifying the limitations of a platform, understanding the impact of design choices or optimizations on performance, and ultimately improving performance. For users, insight into a platform's performance may guide tuning efforts to speed up their application. For system designers and engineers, the identification of key performance issues will help them redesign or optimize their system to improve performance for many users or customers. To address the needs of both groups, we have designed and implemented Grade10, a system for automated bottleneck detection and performance issues identification (Chapter 4). Part of the Grade10 system, we have equipped the key mechanisms with the ability to execute user-defined policies, and have designed various algorithms that can

act as policies in the Grade10 system.

Central to Grade10 is its hierarchical approach to performance analysis. This approach includes a novel formalism for defining an execution model, a hierarchical representation of the execution phases of a graph analysis job. Similarly, Grade10 uses a resource model to capture the usage of resources, both hardware and software, for a graph analysis job. Grade10's resource attribution process attributes resource usage, as measured by a traditional cluster monitoring, to individual phases of a job. This process is guided by attribution rules, selected by an expert user, that define how phases use a given resource. Based on the attributed resource usage, Grade10 identifies bottlenecks for each execution phase. In a final step, Grade10 identifies a variety of performance issues, including phase imbalance and impactful bottlenecks.

RQ3: How to validate the designed approaches through prototypes?

We have implemented both Graphalytics and Grade10 as prototypes (Chapter 5). We have designed for both systems a set of real-world experiments to validate our approaches in practice, and, for Grade10, to validate internal processes. We have used six graph analysis platforms for our experiments, including three community-driven and three industry-driven.

We have validated Graphalytics by using our benchmark to compare six graph analysis platforms. The vendor of each industry-driven platform has implemented and tuned the algorithms in the Graphalytics benchmark for their platform. Using the Graphalytics instrument as a key element of our experimental study, we have found that the performance of graph analysis platforms depends non-trivially of the workload, thereby validating Graphalytics's workload selection that covers many algorithms and datasets.

We have instrumented two graph analysis platforms, Giraph and PowerGraph, to enable analysis of their performance using Grade10, and to collect real-world performance data to validate Grade10's internal processes. We have found that Grade10's resource attribution process, when reasonably tuned by an expert user, can match closely manual expert-level analysis. Also, we have found that Grade10's predicate-based bottleneck identification process can identify bottlenecks at any level in an execution model's hierarchy, and offers expert users flexibility to define which bottlenecks in low-level components are relevant. Finally, we have used Grade10's performance issue identification process to analyze two graph analysis platforms, for a variety of workloads. We find that Grade10 is able to identify a diverse set of issues affecting common, modern graph analysis systems, including a previously undiscovered synchronization bug.

6.2. Future Work

The LDBC Graphalytics benchmark, Grade10, and the surrounding Graphalytics ecosystem are part of the ongoing Graphalytics project. We propose in this section future research directions for the Graphalytics project, first for LDBC Graphalytics and Grade10 individually, and finally for the Graphalytics ecosystem at large.

6.2.1. LDBC Graphalytics

We propose for LDBC Graphalytics three directions for future work:

1) **Extending the Graphalytics workload to include workflows.**

   A workload in LDBC Graphalytics consists of executing one algorithm on one dataset. In practice, users of graph analysis typically use more than one algorithm to analyze the same dataset [11]. Moreover, the output of one algorithm (e.g., WCC to select the largest connect component) may be used as the input of another algorithm. By including workflows to represent such usage patterns of graph analysis systems, Graphalytics may provide more comprehensive comparisons between systems.

2) **New metrics for scalability and elasticity.**

   Although Graphalytics includes metrics to quantify the scalability of systems, these do not yet consider the complexity of some modern architectures. For example, in a heterogeneous system (e.g., using both CPUs and GPUs), it may be possible to independently scale the various resources. Similarly, in a traditional distributed system, using twice as many machines in the same cluster may not result in twice as much network throughput, and thus may skew scalability results. Finally, we note that current scalability metrics do not consider elasticity, i.e., graph analysis platforms that obtain or release resources at runtime as needed.

3) **Applying the renewal process.**

As new applications of graph analysis arise and new platforms are developed, there is a reasonable expectation that the workloads seen in practice change over time. To ensure the continued relevance of Graphalytics, the algorithms and datasets included in the benchmark should be updated through the renewal process.

6.2.2. Grade10

We propose for Grade10 the following directions for future work:

1) **Extending the execution model formalism to include common non-trivial interactions between phases.**

The process of modeling two graph analysis systems has revealed several types of interactions and dependencies between phases that can not be explicitly expressed in Grade10. For example, a phase waiting for a lock to be released is considered by Grade10 to be bottlenecked on that lock. In this scenario, there is an implicit dependency from the phase requesting the lock on the phase holding the lock. However, Grade10 does not have a notion of which other phase is holding the lock, so it cannot express or analyze this relationship. A second example is the existence of (conceptual) phases whose duration is defined by another phase. For example, the communication phase of a job lasts at least until the end of a corresponding compute phase, because the former cannot complete until no new messages can be sent by the latter. A comprehensive study of different graph analysis systems may reveal more patterns that can not currently be expressed as a Grade10 execution model.

2) **Generalizing the resource attribution rules.**

The Greedy and Sink attribution rules allow for a two-tier division of phases; those that are expected to use much of a resource, and those that may use any amount. A generalization of this approach could assign to each (type of) phase a priority and attribute resource usage in order of decreasing priority. Although this allows for strictly more control over the attribution process, it also risks the creation of complex rule sets that may not match reality.

3) **Selecting the resource attribution rules automatically.**

As demonstrated in the experimental evaluation of Grade10 (see Section 5.5.1.3), the accuracy of a given configuration of the resource attribution process can be determined experimentally. This process could be applied to a variety of jobs to automatically select the resource attribution rules and corresponding weights that lead to the highest accuracy.

4) **Identifying causes of bottlenecks.**

Grade10 can be extended to distinguish between resource bottlenecks with different causes. For example, a phase may be bottlenecked on the CPU because it is performs CPU-intensive operations (resource saturation), or because of the presence of many concurrent phases (resource contention). Another scenario may find all phases in the system are bottlenecked on the CPU at the same. This may be the result of all CPUs in the system being fully utilized (resource provisioning), or of imbalance in the system (resource allocation). Distinguishing between these causes may help identify the root cause of a performance bottleneck.

5) **Extending bottleneck predicates to enable more complex rules.**

A bottleneck predicate in Grade10 determines whether a composite phase is bottlenecked on a resource at a given point in time. Its input consists of the bottleneck status (a boolean value) of each of its subphases at the same point in time. This approach could be extended to allow for more complex predicates. For example, converting the bottleneck status from a binary value to a fraction allows for more nuanced predicates. Furthermore, allowing predicates to use historical data would allow for dithering techniques or smoothing functions.

6) **Improving the impact estimation techniques used for performance issue identification.**

Grade10's predefined performance issue identification rules compute an optimistic estimate of the impact of an identified issue. Alternative methods for estimating impact may improve accuracy, e.g., by simulating how long a job might have taken if the issue were resolved (an approach used in blocked time analysis [76] for analyzing the performance of Spark jobs).

7) **Facilitating the comparison of performance between two jobs with a similar execution model.**

Currently, Grade10 only analyzes individual jobs and does not compare performance across jobs. When the execution model of two jobs is similar (e.g., because they were run on the same platform), a direct comparison of identified bottlenecks and performance issues may provide insight into the causes of any observed performance differences. In our experiments, we conducted such a comparison manually, but an automated approach would require additional investigation (e.g., how to treat minor differences, such as different numbers of workers or iterations between jobs).

6.2.3. Graphalytics Ecosystem

We propose several directions for future work that combine multiple components in the Graphalytics ecosystem:

1) **Integration into the development cycle of a graph analysis platform.**

We envision developers of graph analysis platforms using a combination of the Graphalytics benchmark and Grade10 in performance regression testing. By design, Graphalytics provides a workload that is representative for a wide range of graph analysis applications. Periodically running the benchmark and comparing against earlier versions of the same platform may reveal performance regressions. Furthermore, Grade10 can be used to compare performance issues to help isolate the cause of a regression.

2) **Studying how algorithms and datasets impact performance.**

We have shown using Graphalytics that both the algorithm and dataset have significant impact on the performance of a graph analysis job. Using Grade10, we can further study how the different algorithms and datasets included in Graphalytics affect different execution phases in a graph analysis platform. This approach may reveal new bottlenecks and inspire future research into improving the performance of graph analysis platforms.

A

Definition of Algorithms in LDBC Graphalytics

This chapter contains pseudo-code for the algorithms included in LDBC Graphalytics, as described in Section 3.3.3. 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 ← 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
        $new\_rank[v] \leftarrow (1-d)\frac{1}{|V|} + d\left(\sum_{u \in N_{in}(v)} \frac{rank[u]}{|N_{out}(u)|} + \frac{dangling\_sum}{|V|}\right)$
    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
    $\mathrm{comp}[v] \leftarrow v$
end for
repeat
    $converged \leftarrow \mathrm{true}$
    for all $v \in V$ do
        for all $u \in N_{in}(v) \cup N_{out}(v)$ do
            if $\mathrm{comp}[v] > \mathrm{comp}[u]$ then
                $\mathrm{comp}[v] \leftarrow \mathrm{comp}[u]$
                $converged \leftarrow \mathrm{false}$
            end if
        end for
    end for
until $converged$
```

A.4. 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.5. 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.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 ← 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
```

B

GRADE10 MODELS

B.1. SPECIFICATION OF A GIRAPH EXECUTION MODEL

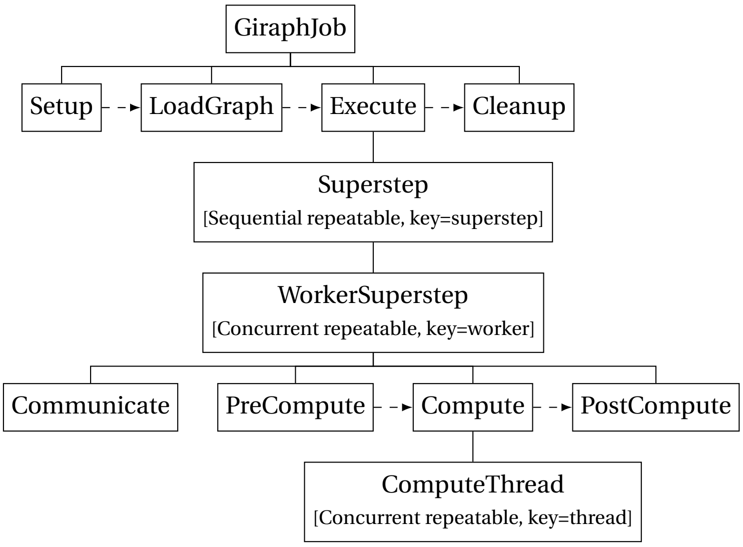


Figure B.1: Execution model specification for Giraph as used in the experimental evaluation of Grade10 (Section 5.5).

B.2. Specification of a Giraph Resource Model

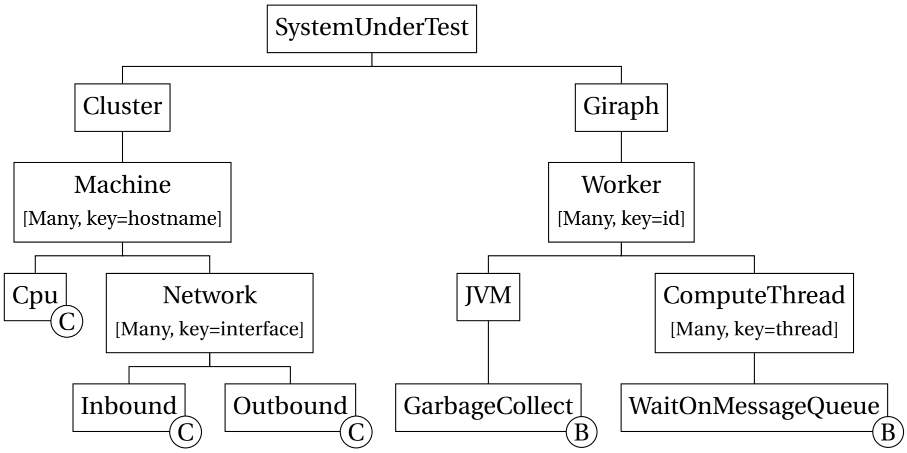


Figure B.2: Resource model specification for Giraph as used in the experimental evaluation of Grade10 (Section 5.5).

B.3. Specification of a PowerGraph Execution Model

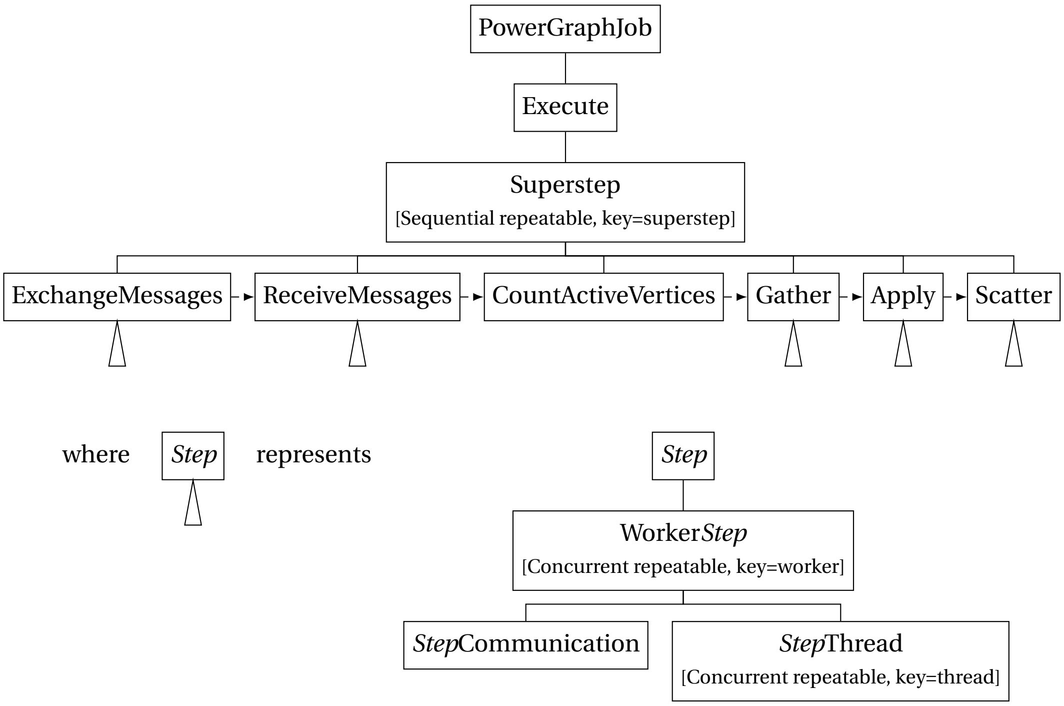


Figure B.3: Execution model specification for PowerGraph as used in the experimental evaluation of Grade10 (Section 5.5). Each Superstep consists of six consecutive steps, five of which are further split into WorkerSteps (where "Step" is replaced with the name of the step, e.g., Gather is split into WorkerGathers). Each WorkerStep is further split into one StepCommunication and multiple StepThread phases (e.g., WorkerGather into GatherCommunication and GatherThreads). Loading the input graph and writing the result of an algorithm are not modeled in the Grade10 prototype model for PowerGraph.

B.4. SPECIFICATION OF A POWERGRAPH RESOURCE MODEL

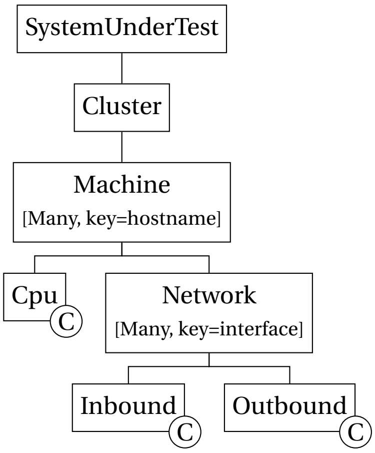


Figure B.4: Resource model specification for PowerGraph as used in the experimental evaluation of Grade10 (Section 5.5).

Bibliography

[1] T. Hey, S. Tansley, and K. Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery* (Microsoft Research, 2009).

[2] Matt Turck, *Firing on All Cylinders: The 2017 Big Data Landscape,* http://mattturck.com/bigdata2017/ (2017), accessed on 2018-06-30.

[3] International Data Corporation, *Big Data and Business Analytics Revenues Forecast to Reach $150.8 Billion This Year, Led by Banking and Manufacturing Investments, According to IDC,* https://www.idc.com/getdoc.jsp?containerId=prUS42371417 (2017), accessed on 2018-06-30.

[4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, *One Trillion Edges: Graph Processing at Facebook-Scale,* PVLDB **8**, 1804 (2015).

[5] X. Wang, F. Wei, X. Liu, M. Zhou, and M. Zhang, *Topic sentiment analysis in Twitter: a graph-based hashtag sentiment classification approach,* in *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011* (2011) pp. 1031–1040.

[6] L. Tang and H. Liu, *Graph Mining Applications to Social Network Analysis,* in *Managing and Mining Graph Data* (Springer US, 2010) pp. 487–513.

[7] P. Ammann, D. Wijesekera, and S. Kaushik, *Scalable, graph-based network vulnerability analysis,* in *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002* (2002) pp. 217–224.

[8] O. Mason and M. Verwoerd, *Graph theory and networks in biology,* IET systems biology **1**, 89 (2007).

[9] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, *Using graph theory to analyze biological networks,* BioData Mining **4**, 10 (2011).

[10] Z. Wang, A. Scaglione, and R. J. Thomas, *Electrical centrality measures for electric power grid vulnerability analysis,* in *Proceedings of the 49th IEEE Conference on Decision and Control, CDC 2010, December 15-17, 2010, Atlanta, Georgia, USA* (2010) pp. 5792–5797.

[11] T. Hegeman and A. Iosup, *Survey of Graph Analysis Applications,* CoRR **abs/1807.00382** (2018), arXiv:1807.00382 .

[12] *The Neo4j Graph Platform,* https://neo4j.com (no date), accessed on 2018-06-30.

[13] *Apache Spark,* http://spark.apache.org (no date), accessed on 2018-06-30.

[14] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, *GraphX: a resilient distributed graph system on Spark,* in *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013* (2013) p. 2.

[15] *Oracle Big Data Spatial and Graph,* http://oracle.com/database/big-data-spatial-and-graph (no date), accessed on 2018-06-30.

[16] *Cray Graph Engine (CGE): Graph Analytics for Big Data,* https://www.cray.com/products/analytics/cray-graph-engine (no date), accessed on 2018-06-30.

[17] P. A. Boncz and T. Neumann, eds., *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013* (CWI/ACM, 2013).

[18] A. Arora, S. Roy, and S. Mehta, eds., *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016* (ACM, 2016).

[19] *2016 High Performance Graph Data Management and Processing Workshop, HPGDMP@SC 2016, Salt Lake City, UT, USA, November 13, 2016* (IEEE, 2016).

[20] S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, eds., *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (ACM, 2017).

[21] P. Boncz and K. Salem, eds., *Proceedings of the 43rd International Conference on Very Large Data Bases, Munich, Germany*, Vol. 10.1-10.13 (2016/2017).

[22] A. Lumsdaine, D. P. Gregor, B. Hendrickson, and J. W. Berry, *Challenges in Parallel Graph Processing,* Parallel Processing Letters **17**, 5 (2007).

[23] N. Doekemeijer and A. L. Varbanescu, *A Survey of Parallel Graph Processing Frameworks*, Tech. Rep. PDS-2014-003 (Delft University of Technology, 2014) http://www.ds.ewi.tudelft.nl/research-publications/technical-reports/2014/.

[24] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: a system for large-scale graph processing,* in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010* (2010) pp. 135–146.

[25] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, *How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis,* in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014* (2014) pp. 395–404.

[26] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, *An Experimental Comparison of Pregel-like Graph Processing Systems,* PVLDB **7**, 1047 (2014).

[27] Y. Lu, J. Cheng, D. Yan, and H. Wu, *Large-Scale Distributed Graph Computing Systems: An Experimental Evaluation,* PVLDB **8**, 281 (2014).

[28] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs,* in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012) pp. 17–30.

[29] N. Sundaram, N. Satish, M. M. A. Patwary, S. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, *GraphMat: High performance graph analytics made productive,* PVLDB **8**, 1214 (2015).

[30] *Apache Giraph,* http://giraph.apache.org (no date), accessed on 2018-06-30.

[31] V. Kalavri, V. Vlassov, and S. Haridi, *High-level programming abstractions for distributed graph processing,* IEEE Trans. Knowl. Data Eng. **30**, 305 (2018).

[32] P. Stutz, A. Bernstein, and W. W. Cohen, *Signal/Collect: Graph Algorithms for the (Semantic) Web,* in *The Semantic Web - ISWC 2010 - 9th International Semantic Web Conference, ISWC 2010, Shanghai, China, November 7-11, 2010, Revised Selected Papers, Part I* (2010) pp. 764–780.

[33] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, *From "Think Like a Vertex" to "Think Like a Graph",* PVLDB **7**, 193 (2013).

[34] *Graph500,* http://www.graph500.org (no date), accessed on 2018-06-30.

[35] S. Shende and A. D. Malony, *The TAU Parallel Performance System,* IJHPCA **20**, 287 (2006).

[36] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, *The Vampir Performance Analysis Tool-Set,* in *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart* (2008) pp. 139–155.

[37] V. Pillet, J. Labarta, T. Cortes, and S. Girona, *Paraver: A tool to visualize and analyze parallel code,* in *Proceedings of WoTUG-18: Transputer and Occam Developments*, Vol. 44 (IOS Press, 1995) pp. 17–31.

[38] W. L. Ngai, T. Hegeman, S. Heldens, and A. Iosup, *Granula: Toward Fine-grained Performance Analysis of Large-scale Graph Processing Platforms,* in *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017* (2017) pp. 8:1–8:6.

[39] *Graphalytics Global Competition,* `https://graphalytics.org/competition` (no date), accessed on 2018-06-30.

[40] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff, *A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term,* IEEE Computer **49**, 54 (2016).

[41] *GitHub: LDBC Graphalytics,* `https://github.com/ldbc/ldbc_graphalytics` (no date), accessed on 2018-06-30.

[42] *GitHub: @Large Research,* `https://github.com/atlarge-research` (no date), accessed on 2018-06-30.

[43] *GitHub: Grade10,* `https://github.com/thegeman/grade10` (no date), accessed on 2018-07-05.

[44] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, *Clearing the clouds: a study of emerging scale-out workloads on modern hardware,* in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012* (2012) pp. 37–48.

[45] B. Elser and A. Montresor, *An evaluation study of BigData frameworks for graph processing,* in *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA* (2013) pp. 60–67.

[46] D. A. Bader and K. Madduri, *Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors,* in *High Performance Computing - HiPC 2005, 12th International Conference, Goa, India, December 18-21, 2005, Proceedings* (2005) pp. 465–476.

[47] K. Ammar and M. T. Özsu, *WGB: Towards a Universal Graph Benchmark,* in *Advancing Big Data Benchmarks - Proceedings of the 2013 Workshop Series on Big Data Benchmarking, WBDB.cn, Xi'an, China, July 16-17, 2013 and WBDB.us, San José, CA, USA, October 9-10, 2013 Revised Selected Papers* (2013) pp. 58–72.

[48] Y. Guo, A. L. Varbanescu, A. Iosup, and D. H. J. Epema, *An Empirical Performance Evaluation of GPU-Enabled Graph-Processing Systems,* in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015* (2015) pp. 423–432.

[49] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. A. Boncz, *Graphalytics: A Big Data Benchmark for Graph-Processing Platforms,* in *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015* (2015) pp. 7:1–7:6.

[50] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan, *BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking,* in *Advancing Big Data Benchmarks - Proceedings of the 2013 Workshop Series on Big Data Benchmarking, WBDB.cn, Xi'an, China, July 16-17, 2013 and WBDB.us, San José, CA, USA, October 9-10, 2013 Revised Selected Papers* (2013) pp. 138–154.

[51] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, *BigDataBench: A big data benchmark suite from internet services,* in *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014* (2014) pp. 488–499.

[52] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, *Navigating the maze of graph analytics frameworks using massive graph datasets,* in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014* (2014) pp. 979–990.

[53] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C. Lin, *GraphBIG: understanding graph computing in the context of industrial solutions,* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015* (2015) pp. 69:1–69:12.

[54] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, P. Faraboschi, and S. Katti, *Parallel Graph Processing: Prejudice and State of the Art,* in *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016* (2016) pp. 85–90.

[55] S. Hong, S. Depner, T. Manhardt, J. V. D. Lugt, M. Verstraaten, and H. Chafi, *PGX.D: a fast distributed graph processing engine,* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015* (2015) pp. 58:1–58:12.

[56] J. Fan, A. G. S. Raj, and J. M. Patel, *The Case Against Specialized Graph Analytics Engines,* in *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings* (2015).

[57] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker, *VERTEXICA: Your Relational Friend for Graph Analytics!* PVLDB **7**, 1669 (2014).

[58] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, *LinkBench: a database benchmark based on the Facebook social graph,* in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (2013) pp. 1185–1196.

[59] M. Dayarathna and T. Suzumura, *Graph database benchmarking on cloud environments with XGDBench,* Autom. Softw. Eng. **21**, 509 (2014).

[60] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz, *The LDBC Social Network Benchmark: Interactive Workload,* in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015) pp. 619–630.

[61] Y. Guo, Z. Pan, and J. Heflin, *LUBM: A benchmark for OWL knowledge base systems,* J. Web Sem. **3**, 158 (2005).

[62] C. Bizer and A. Schultz, *The Berlin SPARQL benchmark,* Int. J. Semantic Web Inf. Syst. **5**, 1 (2009).

[63] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, *SP2Bench: A SPARQL Performance Benchmark,* in *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China* (2009) pp. 222–233.

[64] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, *Diversified Stress Testing of RDF Data Management Systems,* in *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I* (2014) pp. 197–212.

[65] C. U. Smith, *Performance Engineering of Software Systems,* 1st ed. (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990).

[66] R. Jain, *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling,* Wiley professional computing (Wiley, 1991).

[67] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a Realistic Model of Parallel Computation,* in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993* (1993) pp. 1–12.

[68] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. J. Scheiman, *LogGP: Incorporating Long Messages into the LogP Model - One Step Closer Towards a Realistic Model for Parallel Computation,* in *SPAA* (1995) pp. 95–105.

[69] F. Ino, N. Fujimoto, and K. Hagihara, *LogGPS: a parallel computational model for synchronization analysis,* in *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01), Snowbird, Utah, USA, June 18-20, 2001* (2001) pp. 133–142.

[70] C. A. Moritz and M. I. Frank, *LoGPC: Modeling Network Contention in Message-Passing Programs,* in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems, SIGMETRICS '98 / PERFORMANCE '98, Madison, Wisconsin, USA, June 22-26, 1998* (1998) pp. 254–263.

[71] S. Williams, A. Waterman, and D. A. Patterson, *Roofline: an insightful visual performance model for multicore architectures,* Commun. ACM **52**, 65 (2009).

[72] E. Vianna, G. Comarela, T. Pontes, J. M. Almeida, V. A. F. Almeida, K. Wilkinson, H. A. Kuno, and U. Dayal, *Analytical Performance Models for MapReduce Workloads,* International Journal of Parallel Programming **41**, 495 (2013).

[73] D. Glushkova, P. Jovanovic, and A. Abelló, *MapReduce Performance Models for Hadoop 2.x,* in *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017.* (2017).

[74] H. Herodotou, *Hadoop Performance Models,* CoRR **abs/1106.0940** (2011), arXiv:1106.0940 .

[75] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, *DiP: A Parallel Program Development Environment,* in *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II* (1996) pp. 665–674.

[76] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, *Making Sense of Performance in Data Analytics Frameworks,* in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015* (2015) pp. 293–307.

[77] K. Ren, J. López, and G. Gibson, *Otus: Resource Attribution in Data-Intensive Clusters,* in *Proceedings of the second international workshop on MapReduce and its applications* (ACM, 2011) pp. 1–8.

[78] R. Ren, Z. Jia, L. Wang, J. Zhan, and T. Yi, *BDTUne: Hierarchical correlation-based performance analysis and rule-based diagnosis for big data systems,* in *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016* (2016) pp. 555–562.

[79] Q. Guo, Y. Li, T. Liu, K. Wang, G. Chen, X. Bao, and W. Tang, *Correlation-based performance analysis for full-system MapReduce optimization,* in *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA* (2013) pp. 753–761.

[80] O. Marcu, A. Costan, G. Antoniu, and M. S. Pérez-Hernández, *Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks,* in *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016* (2016) pp. 433–442.

[81] W. Qi, Y. Li, H. Zhou, W. Li, and H. Yang, *Data Mining Based Root-Cause Analysis of Performance Bottleneck for Big Data Workload,* in *19th IEEE International Conference on High Performance Computing and Communications; 15th IEEE International Conference on Smart City; 3rd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2017, Bangkok, Thailand, December 18-20, 2017* (2017) pp. 254–261.

[82] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu, *HiTune: Dataflow-Based Performance Analysis for Big Data Cloud,* in *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011* (2011).

[83] V. V. dos Santos Dias, R. Moreira, W. M. Jr., and D. O. Guedes, *Diagnosing Performance Bottlenecks in Massive Data Parallel Programs,* in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016* (2016) pp. 273–276.

[84] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, *Performance Anomaly Detection and Bottleneck Identification,* ACM Comput. Surv. **48**, 4:1 (2015).

[85] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. J. Anderson, I. G. Tanase, Y. Xia, L. Nai, and P. A. Boncz, *LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms,* PVLDB **9**, 1317 (2016).

[86] *Linked Data Benchmark Council,* http://ldbcouncil.org (no date), accessed on 2018-06-30.

[87] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank Citation Ranking: Bringing Order to the Web.*, Technical Report 1999-66 (Stanford InfoLab, 1999).

[88] U. N. Raghavan, R. Albert, and S. Kumara, *Near linear time algorithm to detect community structures in large-scale networks,* Phys. Rev. E **76**, 036106 (2007).

[89] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection,* http://snap.stanford.edu/data (2014), accessed on 2018-06-30.

[90] Y. Guo and A. Iosup, *The Game Trace Archive,* in *11th Annual Workshop on Network and Systems Support for Games, NetGames 2012, Venice, Italy, November 22-23, 2012* (2012) pp. 1–6.

[91] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, *Measuring User Influence in Twitter: The Million Follower Fallacy,* in *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010* (2010).

[92] J. Markoff, *Time Split to the Nanosecond Is Precisely What Wall Street Wants,* The New York Times (2018).

[93] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, *Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization,* in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (2018) pp. 81–94.

[94] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, *Spark: Cluster Computing with Working Sets,* in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010* (2010).

[95] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, *GraphPad: Optimized Graph Primitives for Parallel and Distributed Platforms,* in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016* (2016) pp. 313–322.

[96] *Oracle Labs PGX: Parallel Graph Analytics Overview,* http://oracle.com/technetwork/oracle-labs/parallel-graph-analytics (no date), accessed on 2018-06-30.

[97] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. P. Pérez, T. Manhardt, H. Chafi, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz, *LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms,* Tech. Rep. DS-2016-001 (Delft University of Technology, 2016) http://www.ds.ewi.tudelft.nl/research-publications/technical-reports/2016/.

[98] T. Hoefler and R. Belli, *Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results,* in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015* (2015) pp. 73:1–73:12.

[99] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz, *Trash Day: Coordinating Garbage Collection in Distributed Systems,* in *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015* (2015).

[100] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, *Broom: Sweeping Out Garbage Collection from Big Data Systems,* in *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015* (2015).