TÉCNICO LISBOA

UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Graph Processing: Distributed Frameworks, Approximations and Compact Data Structures

Miguel Carvalho Valente Esaguy Coimbra

Supervisor:	Doctor Luís Manuel Antunes Veiga
Co-Supervisor:	Doctor Alexandre Paulo Lourenço Francisco

Thesis approved in public session to obtain the PhD Degree in
Computer Science and Engineering

Jury final classification: Pass with Distinction

2021

UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Graph Processing: Distributed Frameworks, Approximations and Compact Data Structures

Miguel Carvalho Valente Esaguy Coimbra

Supervisor: **Doctor Luís Manuel Antunes Veiga**
Co-Supervisor: **Doctor Alexandre Paulo Lourenço Francisco**

Thesis approved in public session to obtain the PhD Degree in
Computer Science and Engineering

Jury final classification: Pass with Distinction

Funding Institutions
Universidade de Lisboa
European Commission
Fundação para a Ciência e Tecnologia

2021

Acknowledgements

Gratitude is an important secret to life. As it was when I began this journey of higher studies, I start with my supervisors, Professor Luís Veiga and Professor Alexandre Francisco, whose role I am extremely grateful for. Their guidance and constant availability to discuss new ideas, insights and to give me feedback was a cornerstone of my evolution as an engineer, researcher and student. They provided me with countless opportunities to engage in new environments and challenges and their expertise and availability were impeccable and constant as we progressed through both new ideas and their many iterations. I am thankful for their acceptance to supervise me when I came knocking on their doors, as well as their openness and encouragement to partake in internship programs and to explore new ideas. Their high standards as researchers were a strong factor of growth, which always motivated me to learn and keep improving the quality of this work.

During this journey, I had the pleasure of collaborating with Mennan Selimi and Sérgio Esteves on different initiatives, for which I am thankful. Of special importance was Professor Alexandre and Professor Gonzalo Navarro's invitation to intern at the University of Chile, which led to collaborations with him, Professor Travis Gagie and my colleague Joana Hrotko. I am also thankful to Professor José Borbinha for inviting me to participate in lectures and additional projects which complemented my skill set and led me to work on new topics. A special thanks to Professor Alberto Silva for his availability to coordinate my internship and admission into the Portuguese Order of Engineers in the context of my research work. During this journey I was a teaching assistant on different courses, and I am thankful to the senior colleagues with whom I learned more about the art of teaching: Professor João Barreto, Professor Miguel Pardal, Professor Ricardo Chaves and Professor Luís Veiga. Equally important, I thank the members of the Distributed Systems Group and of the Information and Decision Support Systems Lab at INESC-ID.

In different stages of this journey, I had the pleasure of working alongside remarkable colleagues, with whom I exchanged ideas and support: Amin Khan, Ana Correia, Daniel Porto, David Matos, Diogo Barradas, Hugo Rodrigues, João Loff, Jorge Oliveira, Maria Casimiro, Nuno Diegues, Naércio Magaia, Pedro Mota, Sérgio Esteves, Shady Issa, Sofia Teixeira and Tiago Brito. I am thankful for having met them, as well as others who made this process so much more rich. I also deeply appreciate the close friends who were present in this journey and with whom I shared many moments. Lastly, I thank my parents for the privilege of their unconditional support which I have always had since I could remember. My father who has always been available to advise me. He was crucial to the footsteps I took until the point I began my higher studies. Had it not been for his contagious interest in engineering and computers, I may have not delved into this wonderful field. My mother who always had the patience of a saint towards everyone even in times of hardship. Her unwavering resolve to help everyone around her is a model unlike any other.

Lisbon, September 29, 2021
Miguel Carvalho Valente Esaguy Coimbra

The author was primarily supported by a Doctoral Degree Scholarship from the University of Lisbon and a PhD Studentship from Fundação para a Ciência e Tecnologia. The individual scientific research initiatives were partly supported by the Portuguese government through FCT – Fundação para a Ciência e Tecnologia (FCT) [grants UID/CEC/500021/2013, CMUP-ERI/TIC/0046/2014, PTDC/EEI-SCR/6945/2014, PTDC/CCI-BIO/29676/2017, PTDC/CPO-CPO/28495/2017, PTDC/EEI-COM/30644/2017, UID/CEC/50021/2019, UIDB/50021/2020], by the ERDF through COMPETE 2020 Programme, within project POCI-01-0145-FEDER-016883, by the European H2020 project LightKone (H2020-732505), by the Spanish government under contract TIN2016-77836-C2-2-R., by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie [grant agreement No 690941] (while the author was visiting the University of Chile), by MICINN-AEI (PGE and ERDF) [grants TIN2016-77158-C4-3-R,RTC-2017-5908-7,PID2019-105221RB-C41], by Xunta de Galicia (co-founded by ERDF) [grant ED431C 2017/58] and by ANID - Millennium Science Initiative Program - Code ICN17_002.

"Just because nobody complains doesn't mean all parachutes are perfect."
- Benny Hill

Resumo

Este trabalho foi desenvolvido num contexto de *big data*, grafos dinâmicos e as diferentes arquiteturas e abordagens usadas para processar e armazenar grafos. Apresenta-se uma visão de topo do estado-da-arte do processamento de grafos, resultante de um estudo das técnicas e sistemas usados para representar grafos computacionalmente, as diferentes arquiteturas de sistemas que os processam e de bases de dados. Estes aspetos estão interligados entre si e relacionam-se com questões que ganham forma em função de novas necessidades de processamento de grafos.

Dada a crescente dimensão dos dados representados como grafos em *big data*, que equilíbrio será possível entre cálculo exato e aproximado? Até que ponto se poderá ganhar desempenho em troca de qualidade dos resultados no processamento distribuído de grafos? Empregando sistemas distribuídos, como se poderá obter escalabilidade horizontal em métodos de *clustering* (como exemplo de um relevante algoritmo em grafos) para processar grafos de grande dimensão, aproveitando o poder computacional de infraestruturas distribuídas? Quão fácil será adotar soluções incrementais para processamento de grafos dinâmicos? Será possível atualizar resultados do processamento, tendo como base somente o próprio resultado das mudanças do grafo?

Estas questões são abordadas em VEILGRAPH, um estudo e sistema de processamento aproximado de grafos usando uma estrutura de sumarização para processamento em *stream*. Avaliam-se diferentes parâmetros de aproximação executando a lógica do sistema em ambiente *cloud*. Foi também estudado o potencial de melhorar sistemas distribuídos (*community network micro-clouds*) ao definir grupos computacionais e eleição de líderes como problema de grafos. Os resultados são apresentados em GELLY-SCHEDULING, estudando as propriedades das *community network micro-clouds*.

Um aspeto relevante no processamento de grafos é a representação computacional dos mesmos. Como se poderão usar representações compactas de grafos para manipular grafos dinâmicos grandes? Esta questão está alinhada com as outras, pois a escolha de representação do grafo tem influência no desempenho. É explorado este aspeto com a implementação dinâmica da estrutura de dados k^2-tree, desenvolvida no decorrer deste trabalho. É uma estrutura compacta para representar grafos, para a qual se apresenta um estudo comparativo de diferentes implementações.

Amplamente, o processamento de grafos tornou-se uma interseção de múltiplas perspetivas. Esta tese realça estas tendências e as suas relações através do estudo e produção científica desenvolvidas na mesma.

Abstract

This work was pursued in context of big data, dynamic graphs and the different architectures and approaches used to process and store graphs. Herein an overview is provided of the state-of-the-art of the graph processing landscape as result of a survey on the techniques and systems to computationally represent graphs, different architectures of systems that process them and graph databases. These aspects of the landscape are interconnected and relate to questions that emerge as different needs for graph processing take form.

Given the increasing dimension of data sets represented as graphs in the advent of big data, how far can one relax from exact to approximate computing? And to what extent can performance be gained with a trade-off in accuracy in distributed system graph-processing scenarios? Employing distributed systems, how may one scale-out methods of graph clustering (as an example of a relevant graph algorithm) to process large graphs while harnessing the computational power of distributed computing infrastructures? How easy is it to adopt incremental solutions for the processing of dynamic graphs? Could existing graph processing results be updated based solely on previous results and a given change, or set of changes, in the graph?

These questions are approached with the VEILGRAPH contribution, a study and system for approximate graph processing using a summary structure applied to the case of stream processing. With it, different approximation parameters in a cloud computational environment are evaluated. As part of this work, the focus was also cast on the potential to improve distributed systems (community network micro-clouds) by defining computational groups and leader election as a graph processing problem. Results inherent to this were obtained with the GELLY-SCHEDULING contribution, where the properties of community network micro-clouds are studied.

A relevant aspect underlying these questions is the computational representation of graphs. In what way may compact graph representations be used to manipulate big dynamic graphs? The answer to this question is aligned with the others, as the choice of graph representation has direct influence on performance. This aspect is explored with a dynamic implementation of the k^2-tree data structure to represent graphs, developed throughout the research effort. It is a dynamic compact graph data structure, of which a comparative performance study for different implementations is presented.

Overall, graph processing has become an intersection of multiple perspectives, all interconnected. This thesis highlights these tendencies and their relations through the developed research and contributions.

Palavras Chave
Keywords

Palavras Chave

Processamento de Grafos
Sistemas Distribuídos
Processamento em Stream
Computação Aproximada
Estruturas de Dados

Keywords

Graph Processing
Distributed Systems
Stream Processing
Approximate Computing
Data Structures

Index

List of Tables

1 Introduction

Graphs are, and have been for many decades, a powerful mental artifice to model many real-world problems. The inception of graph theory as a field of study was due to Leonhard Euler and his work on the *Seven Bridges of Königsberg* problem of mathematics (Euler 1956), performed in 1736. He is regarded in graph theory to be the one that formally pioneered the field (Jones and Pevzner 2004). The problem he solved is illustrated in Figure 1.1. As another example, consider the important algebra operation of inverting a matrix: fifty years ago a proposal was made for an algorithm (Nathan and Even 1967) which represents a matrix as a graph, in order to rapidly perform this operation.

Data that benefits from graph-representations is found almost everywhere, with examples emerging across decades such as bio-informatics data representation via *de Bruijn* graphs (de Bruijn 1946) in metagenomics (Zerbino and Birney 2008; Li, Zhu, Ruan, Qian, Fang, Shi, Li, Li, Shan, Kristiansen, Li, Yang, Wang, and Wang 2010; Pell, Hintze, Canino-Koning, Howe, Tiedje, and Brown 2012; Muggli, Bowe, Noyes, Morley, Belk, Raymond, Gagie, Puglisi, and Boucher 2017), atoms, covalent relationships and other applications in chemistry (Balaban 1985; Ivanciuc 2013; Lim, Ryu, Park, Choe, Ham, and Kim 2019), analysing the structure of the World Wide Web (Brin and Page 1998; Boldi and Vigna 2004b; Chung 2010; Meusel, Vigna, Lehmberg, and Bizer 2015), massive parallel learning of tree ensembles (Panda, Herbach, Basu, and Bayardo 2009; Krawczyk, Minku, Gama, Stefanowski, and Woźniak 2017), the structure of distributed computation itself (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, and Czajkowski 2010; Murray, McSherry, Isaacs, Isard, Barham, and Abadi 2013; Schelter, Palumbo, Quinn, Marthi, and Musselman 2016) and parallel topic models (Smola and Narayanamurthy 2010; Zhao, Zhou, Li, and Huang 2018; Jain, Arora, and Agrawal 2020). Academic research centres in collaboration with industry players like Facebook, Microsoft and Google have rolled out their own graph processing systems, contributing to the development of several open-source frameworks (Ching 2013; Xin, Gonzalez, Franklin, and Stoica 2013; Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015; Microsoft 2017; Mariappan and Vora 2019). They need to deal with huge graphs, such as the case of the Facebook graph with billions of vertices and hundreds of billions of edges (Gordon Donnelly 2020). Graph theory is thus, in itself, both a means and an end in many scenarios.

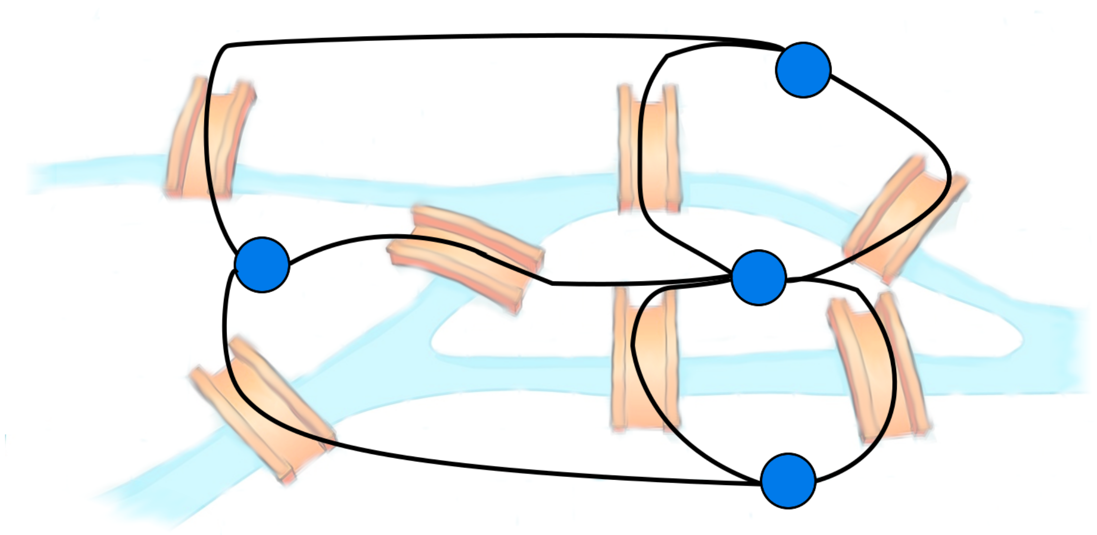


Figure 1.1: Euler's graph-based perspective on the Seven Bridges of Königsberg problem. It consisted in devising a walk through the city that would cross each of these bridges exactly once.

1.1 Domains and Motivation

We list some of the domains of human activity that are best described by relations between elements - graphs:

- **Social networks.** They make up a large portion of social interactions in the Internet. We name some of the best-known ones: Facebook (2.50 billion monthly active users as of December 2019 (Facebook 2020)), Twitter (330 million monthly active users in Q1'19 (Twitter, Inc. 2020)) and LinkedIn (330 million monthly active users as of December 2019 (LinkedIn Corporation 2020)). In these networks, the vertices represent users and edges are used to represent friendship or follower relationships. Furthermore, they allow the users to send messages to each other. This messaging functionality can be represented with graphs with associated time properties. Other examples of social networks are WhatsApp (1.00 billion monthly active users as of early 2016 (WhatsApp Inc. 2016)) and Telegram (300 million monthly active users (Securities and Exchange Commission 2019)).

- **World Wide Web.** Estimates point to the existence of over 1.7 billion websites as of October 2019 (InternetLiveStats.com 2020), with the first one becoming live in 1991, hosted at CERN. Commercial, educational and recreational activities are just some of the many facets of daily life that gave shape to the Internet we know today. With the advent of business models built over the reachability and reputation of websites (e.g. Google, Yahoo and Bing as search engines), the application of graph theory as a tool to study the web structure has matured during the last two decades with techniques to enable the analysis of these massive networks (Boldi and Vigna 2004a; Boldi and Vigna 2004b).

- **Telecommunications.** These networks have been used for decades to enable distant communication between people and their structural properties have been studied using graph-based approaches (Baritchi, Cook, and Holder 2000; Balasundaram and Butenko 2006). Though some of its activity may have transferred to the applications identified above as social networks, they are still relevant. The vertices in these networks represent user phones, whose study is relevant for telecommunications companies wishing to assess closeness relationships between subscribers, calculate churn rates, enact more efficient marketing strategies (Al-Molhem, Rahal, and Dakkak 2019) and also to support foreign signals intelligence (SIGINT) activities (Pfluke 2019).

- **Recommendation systems.** Graph-based approaches to recommendation systems have been heavily explored in the last decades (Grujić 2008; Gu, Zhou, and Ding 2010; Silva, Tsang, Cavalcanti, and Tsang 2010). Companies such as Amazon and eBay provide suggestions to users based on user profile similarity in order to increase conversion rates from targeted advertising. The structures underlying this analysis are graph-based (Beyene, Faloutsos, Chau, and Faloutsos 2008; Zhao, Yao, Li, Song, and Lee 2017; Yang and Toni 2018).

- **Transports, smart cities and IoT.** Graphs have been used to represent the layout and flow of information in transport networks comprised of people circulating in roads, trains and other means of transport (Euler 1956; Unsalan and Sirmacek 2012; Rathore, Ahmad, Paul, and Thikshaja 2016). The Internet-of-Things (IoT) will continue to grow as more devices come into play and 5G proliferates. The way IoT devices engage for collaborative purposes and implement security frameworks can be represented as graphs (George and Thampi 2018).

- **Epidemiology.** The analysis of disease propagation and models of transition between states of health, infection, recovery and death are very important for public health and for ensuring standards of practices between countries to protect travellers and countries' populations (Colizza, Barrat, Barthélemy, and Vespignani 2007; Bajardi, Poletto, Ramasco, Tizzoni, Colizza, and Vespignani 2011; Brockmann and Helbing 2013; Chinazzi, Davis, Ajelli, Gioannini, Litvinova, Merler, y Piontti, Mu, Rossi, Sun, et al. 2020). These are represented as graphs, which can also be applied to localized health-related topics like reproductive health, sexual networks and the transmission of infections (Liljeros, Edling, and Amaral 2003; Bearman, Moody, and Stovel 2004). They have even been used to model epidemics in massively multiplayer online games such as World of Warcraft (Lofgren and Fefferman 2007). Real-life epidemics are perhaps at the forefront of examples of this application of graph theory for health preservation, with the most recent example as COVID-19 (Surveillances 2020).

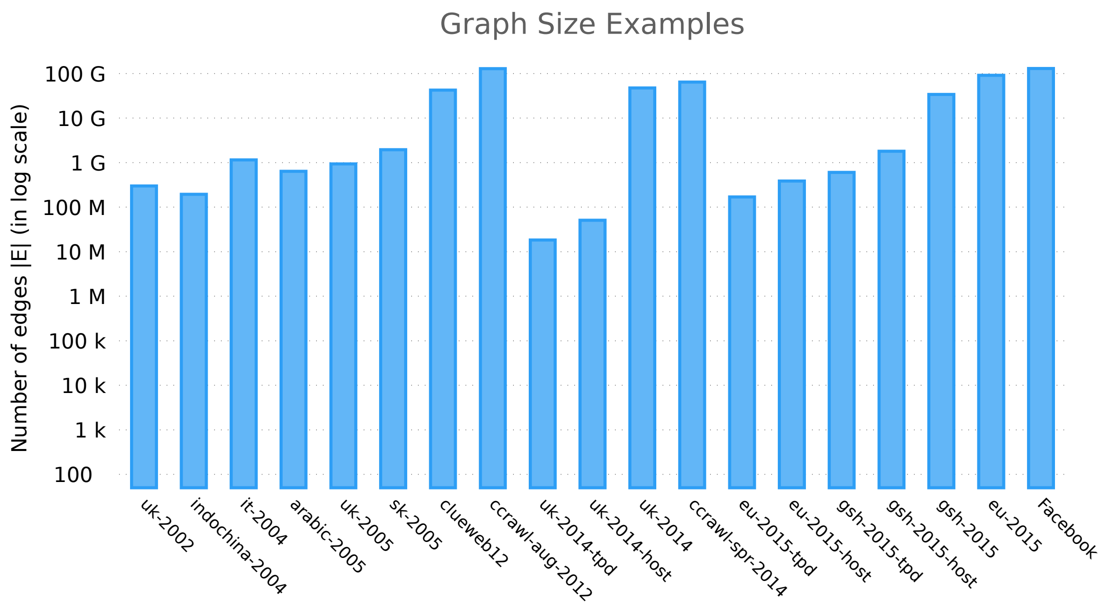


Figure 1.2: Web graph edge counts for domain crawls since the year 2000 plus the size of the Facebook graph (in log scale).

Other types of data represented as graphs can be found (Sedgewick and Wayne 2011). To illustrate the growing magnitude of graphs, we focus on web graph sizes of different web domains in Figure 1.2, where we show the number of edges for web crawl graph datasets made available by the `Laboratory of Web Algorithmics` (Laboratory for Web Algorithmics 2020) and by `Web Data Commons` (Meusel, Vigna, Lehmberg, and Bizer 2015) and also the size of the Facebook graph. If one were to retrieve insights on the structure of some of the largest graphs (for example at the magnitude of trillions of edges), it would become immediately clear that a combination of computer resources and specific software are necessary in order to process them. Such has already occurred with different computational approaches across the years (Ching 2013; Ching, Edunov, Kabiljo, Logothetis, and Muthukrishnan 2015; Maass, Min, Kashyap, Kang, Kumar, and Kim 2017b), and the trend remains regarding the growth of graph-based datasets.

1.2 Opening Questions

Our objective is to study the relations between these requirements and contribute to state-of-the-art techniques. Before delving into the research vector of this thesis, we present a set of research questions. While most of them have been studied in the literature, we consider they have tentative answers so far and present them as examples. We attribute this to the fact that any attempt at optimally answering them will be impacted by how many of the questions are considered at the same time.

Given the increasing dimension of data sets represented as graphs in the advent of big data, how far can one relax from exact to approximate computing? Naturally, if all the time of the world is available to us, it suffices to mine the whole graph (no matter how big it is) using one of the existing computational batch approaches (e.g., using a single machine with a lot of memory versus a cluster of distributed workers). If temporal requirements are relaxed, one could also trigger a new computation over the whole graph as soon as the previous one finishes and after integrating graph updates (such as edge removal/addition operations from a data stream) for offline analysis.

For more common scenarios, time is critical. To reduce time in the face of increasing data volumes, other approaches must be taken. Approximate results may open the door for attaining desired speedups, but how may we reliably define and ensure controlled error bounds given the volatility of graphs such as those representing social networks? We envision two practical approaches to this question. One could start by applying techniques to discard non-essential processing tasks. This is one of many approaches undertaken in distributed systems – consider the possibility of discarding older straggling tasks that have not yet completed in a distributed execution and consciously accumulating error (Goiri, Bianchini, Nagarakatte, and Nguyen 2015). A complete execution may be necessary at a given point in time to restore result quality. Another approach could be aware of the data itself (graph structure and how it evolves) in order to trim unnecessary computation (Vora, Gupta, and Xu 2017). Elements of the graph may contribute to results in a non-significant way and thus be ignored.

Establishing a basis for solving these challenges would pave the way for greater performance and resource-efficiency in the analysis of many graph-based big data scenarios. There exists a plethora of commercial and open-source distributed systems (some are general-purpose with graph processing modules, others are dedicated to graph processing) which may be used as a platform for conducting further studies. This work aims to respond to two challenges: *1)* to gain an understanding of the most desirable system(s) for graph processing to use for further work; *2)* exploring important graph problems in the literature while considering the dimensions of incremental, approximate and streaming computing as well as efficient graph representations.

1.3 Research Objectives and Contributions

This section enumerates our research objectives and our contributions towards them in light of our research activity. Overall, they were developed under three conceptual areas: graph processing, stream processing and distributed systems. Figure 1.3 showcases their overlap. While there are pairwise relations between them, our ultimate objectives and effort lie in the overlap of the three. We aimed to produce novel solutions to improve the processing of big evolving graphs in distributed systems.

The research efforts herein detailed, while they focus on what appear to be separate topics in the broader scope of graph processing challenges, have important links

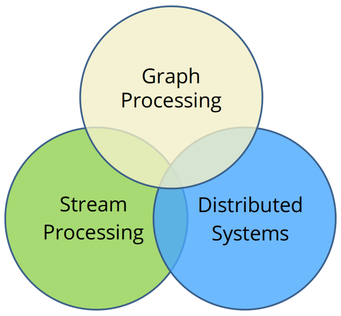


Figure 1.3: Areas of multidisciplinary research effort.

between them, warranting additional context on their sequencing.

We have carried out and published a survey on the landscape of graph processing, covering many different approaches to processing, computational representation and storage. While this survey branches out into different components of the landscape, it is of the utmost importance to highlight that such components have links between them in context of each of our research initiatives, taking the form of different related works.

In the scope of our work GELLY-SCHEDULING, focusing on distribution of computation for the definition of communities and leader election, we performed an initial analysis of graph processing frameworks, having considered, from among others, `Apache Flink` and `Apache Spark` as relevant candidates on which to execute our experiments. Among other factors, `Flink`'s highly-active development community led to choosing it. With GELLY-SCHEDULING, the target of community network micro-clouds (CNMCs) and our novel approach to it was validated. However, additional concerns were inherent to this work if the problem space was to be expanded. With an increasing number of devices and dissemination of the Internet to the further reaches of the world, the ability to maintain updated views over communities and leadership assignment would become a key-element of the network's management. Hence the approach based on Gelly/Flink that leverages its horizontal scalability.

However, not all changes in these bigger settings represented with graphs would have the same importance. Based on the fact that there exist different applications and computational classes of connected nodes, changes taking place in bigger networks (potentially greater in order of magnitude) would have different relevance with respect to management.

We thus focused on this following problem of the need to be able to process incoming information, while ensuring the most relevant and potentially impactful graph changes are taken into account when updating metrics, and also exploring the possibilities of sacrificing result accuracy for faster execution times. As a consequence, we explored this problem in the scope of stream processing with our work VEILGRAPH. Due to the reason for expanding the ideas of GELLY-SCHEDULING (though with the potential to apply VEILGRAPH not only to its problem) in this stream processing context, we opted to continue building over `Apache Flink`.

`Apache Flink` has become a very refined dataflow programming framework, offering many libraries and the ability to harness distributed and parallel execution of jobs. However, we faced some limitations (detailed in Chapter 7) when needing to reuse the same data for separate `Flink` dataflow jobs executed in sequence. In the case of `Flink` (and also `Spark`), such a reuse incurs growing costs associated with continuously reading and writing the graph data to secondary storage.

But what if it was possible to lessen the impact of this I/O-heavy behaviour by representing the graph data in a more compact form, perhaps allowing for redundancy between cluster nodes, with each one able to represent the full dataset? Before exploring the idea of incorporating such a structure into `Flink` (even though a preliminary analysis was performed of its architecture for that purpose), we posited that the first step in exploring such an idea would be to analyse different computational representations in the literature. Particularly, we focused on the k^2-tree data structure, a recent idea of which there was already a compact representation (thus supporting changes to the graph, as an addition to enabling a space-efficient representation). To this end, we contacted the authors of different k^2-tree representations and also implemented our own based on a technique different from existing implementations.

Orthogonally to this series of questions and research efforts, we continuously analysed alternative architectures, representations and systems relevant to each of our work. We then structured all these observations and the many connected aspects of the landscape of graph processing and included this structured review in our published survey.

Ultimately, procuring these solutions led to these interrelated research initiatives:

- **A survey on the field of graph processing and existing techniques and systems.** Assessing the most relevant marks in the literature pertaining the landscape of graph processing. As part of the state-of-the-art presented in this document, we provide an overview of existing computational graph representations, usage of compression techniques, architectures of graph processing systems and databases. This entails a detailed analysis of the intrinsic aspects of each of these topics as well as their relationships. We present from simple to complex graph representations used in the literature, spanning linked lists, adjacency matrices, sparse vectors as well as more elaborate schemes which involve efficient compression. From the conceptual perspective of expressing the computation and processing of graphs, we present approaches which use different computational units for processing graphs, such as for example their vertices, edges or subgraphs. All these innovations encompassed in the literature debuted as key implementations and designs of systems and databases. Some produced changes in paradigm, while others improved specific cases for existing ones. We go over these dynamics and important milestones by presenting an exhaustive list of systems, respective features and their comparisons in Chapter 3, providing value to both experienced researchers and developers as well as individuals grasping how to approach graph-based data. This was done in the scope of our published survey (Coimbra, Francisco, and Veiga 2021)

- **Studying trade-offs between result accuracy and execution speed when employing summarization techniques on graph stream processing scenarios.** Exploring new approaches to processing evolving graphs by using approximate processing methods. When a graph evolves due to a stream of updates, the impact will vary depending on the type of data mined from the graph. For example, vertex rank (such as PageRank) fluctuations will be barely noticeable if peripheral network vertices are cut off. However, if high-degree vertices (topic of interest in *power-law* graphs (Fortunato, Flammini, and Menczer 2006)) have their connections changed in a way that influences network topology, then the variation in ranks may be noted much more.

  There have been proposals of structures such as spanners, sparsifiers and subgraphs to represent properties of the graph in the context of update streams (Ahn, Guha, and McGregor 2012). To efficiently update existing results of a graph algorithm over an evolving graph, some structures and representations may be more suitable than others. To research the benefits of innovative representations, it became an objective to explore proper architectures to express graph processing under these concerns. With the analysis of existing state-of-the-art contributions in the literature, we elected the most suitable tools for distributed graph processing in mind. These tools had a primary focus on distributing execution and working with data streams. While they provide abstractions to manipulate graphs, they treat as second-class citizens (or completely disregard) the graph-centric concerns that are necessary when exploiting distributed systems and streams of data.

  Chapter 4 presents our submission VEILGRAPH (Coimbra, Esteves, Francisco, and Veiga 2021), which is the contribution of a model for approximate graph processing which is evaluated using a summarization technique over PageRank. We aimed to provide an API through which the model may easily be configured. One could be interested in obtaining approximate results at a given time instead of performing a full computation. We envision this work to branch out into different experiments. First, it would be desirable to perform further validation of the model and API by testing with other graph algorithms. Second, it would be interesting to implement an automated computation strategy through statistical learning techniques to arm users with what would be an auto-pilot strategy.

- **Exploring and analysing the application of graph processing techniques to cloud community micro-clouds.** The initiative was relevant because it served as a motivator and a way to gain experience on graph processing techniques and challenges when implementing solutions. It was an innovative application of graph processing for the task of leader election in community network micro-clouds. Chapter 5 details our publication GELLY-SCHEDULING (Coimbra, Selimi, Francisco, Freitag, and Veiga 2018) on the application of graph processing techniques to study service placement in community network micro-clouds. This work could potentially be further developed with additional application scenarios to validate our approach. An example of such a development would be execution in a production network environment.

- **Analysing and improving compact graph representations.** Studying smart graph representations and their applications. Smart graphs are compact graph representations, typically memory-efficient (taking up less space than adjacency matrices, lists and sparse vectors) but allowing access and manipulation of the graph, without requiring a full transformation of the smart graph into a non-compact representation. As a first step in this direction, we have studied the k^2-tree compact graph data structure and compared different implementation variants including our own to assess their efficiency, having published the results (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020) (Chapter 6).

1.4 List of Publications

In the course of our research, we produced different submissions which we list:

- **An analysis of the graph processing landscape**. Submission to Springer Open - Journal of Big Data of a survey analysing the graph processing landscape (Coimbra, Francisco, and Veiga 2021), mainly-detailed in Chapter 3 (*SCImago Journal Q1*)).

- **VeilGraph: Streaming Graph Approximations**. Submission to Springer Open - Journal of Big Data of the VEILGRAPH framework (Coimbra, Esteves, Francisco, and Veiga 2021) using summarization to explore performance and accuracy trade-offs with approximations over streams, detailed in Chapter 4 (*SCImago Journal Q1*)).

- **Gelly-Scheduling: Distributed Graph Processing for Service Placement in Community Networks**. Publication to ACM Symposium on Applied Computing 2018 of the GELLY-SCHEDULING system (Coimbra, Selimi, Francisco, Freitag, and Veiga 2018) which applies graph processing techniques to community network micro-clouds, fully-addressed in Chapter 5 (*CORE2018/2020 Rank: B*).

- **On dynamic succinct graph representations**. Publication to the Data Compression Conference 2020 of an implementation and comparative study k^2-tree data structure implementations (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020), described in Chapter 6 (*CORE2020 Rank: A**).

- **A practical succinct dynamic graph representation**. Submission to the Elsevier Information and Computation Journal special issue of a library-based implementation of the k^2-tree data structure, the implementation of popular graph algorithms over it and a comparative analysis against other state-of-the-art implementations (Coimbra, Hrotkó, Francisco, Russo, de Bernardo, Ladra, and Navarro 2021) fully-addressed in Chapter 6 (*SCImago Journal Q2*)).

1.5 Document Structure and Style

Chapter 2 details important graph problems in the literature. Chapter 3 describes the multidisciplinary topics in the related work of this study. This pertains to the dimension of relevant graph algorithms (along with their distinctive aspects) and the frameworks and tools with graph processing capabilities. Chapter 4 contains our results and experiments of approximate graph processing with VEILGRAPH. Chapter 5 presents our approach to community micro-clouds with GELLY-SCHEDULING. Chapter 6 shows our comparative study of k^2-tree implementations. Lastly, Chapter 7 summarizes our contributions and proposes future vectors of research.

2 Background

This chapter begins in Section 2.1 with a brief introduction to the common notation and concepts inherent to the representation of graphs, both computational and in manuscripts. This chapter then presents an overview of important problems in the literature of graph theory in Section 2.2. They partake in the formal definition of algorithms in the literature which were created over the years to provide answers to problems in graph theory.

2.1 An Introduction to Graphs

Here we detail terms and concepts which are known in graph theory. We include preliminary notions that serve as a basis to familiarize the reader with the language used in scientific documents on graph applications, processing systems and novel techniques. In the literature (Newman 2010), a graph G is written as $G = (V, E)$ – it is usually defined by a set of vertices V of size $n = |V|$ and a set of edges E of size $m = |E|$. Vertices are sometimes referred to as *nodes* and edges as *links*. Along this document we will use the terms *vertex/vertices* and *edge/edges* when referring to elements of the graph. An undirected graph G is a pair (V, E) of sets such that $E \subseteq V \times V$ is a set of unordered pairs. If E is a set of ordered pairs, then we say that G is a directed graph. Between the same two vertices there is usually at most one edge; if there are more, then the graph is called a *multigraph* (note: an ordered graph in which a pair of vertices share two edges in opposite direction is not necessarily a multigraph). Multigraphs are more common when looking at the applications and use-cases for graph databases such as `Neo4j` (Miller 2013), where one may model more than one relation type between the same vertices.

Additionally, given a graph $G = (V, E)$, the set of vertices of G may also be written as $V(G)$ and the set of edges as $E(G)$. In the literature of the graph processing landscape, we find V and E to denote the vertex and edge sets respectively. The concept of a vertex's *surrounding* is important for specifying traversals (relevant when considering graph query languages (Holzschuher and Peinl 2013)) and also defining scopes and units of computation in graph processing (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, and Czajkowski 2010; Roy, Mihailovic, and Zwaenepoel 2013; Tian, Balmin, Corsten, Tatikonda, and McPherson 2013). Two vertices $u, v \in V$ are considered adjacent or neighbours if (u, v) is an edge, that is, $(u, v) \in E$. Given a vertex $v \in V(G)$, its set of neighbours is denoted by $N_G(v)$ or succinctly by $N(v)$ when clearing G from the context. The set of edges with v as a source or target is written as $E(v) = \{(u, u') \in E(G) \mid v = u \text{ or } v = u'\}$. More generally, for two vertex sets $X, Y \subseteq V$, we denote $E(X)$ as the set of edges with at least one end in X: $E(X) = \{(u, v) \in E \mid u \in X \text{ or } v \in X\}$ and denote as $E(X, Y)$ the set of edges whose sources are in X and targets are in Y or vice-versa: $E(X, Y) = \{(u, v) \in G \mid \text{either } u \in X \text{ and } v \in Y, \text{ or } v \in X \text{ and } u \in Y\}$. Furthermore, the degree of a vertex is used to indicate its number of neighbours. For a graph $G = (V, E)$ and a vertex $v \in V$, the degree of v may simply be commonly written as $d_G(v)$ or $d(v)$, with $d(v) = |N(v)|$. If G is a directed graph, we find in the literature to be written as $d_{in}(v)$ and $d_{out}(v)$ the number of incoming and outgoing neighbours of v respectively.

These notations are the basic building blocks for graph theory and a mandatory learning topic for those who come from backgrounds such as Mathematics, Computer Science and many other fields. We note that across decades, the field of graph theory has matured and refined the notations and systematization of how we research and approach problems represented as graphs. These advancements have been accompanied with an even faster rhythm of technological progress. In tune, so has evolved the way these notations and representations are translated to concrete actionable computational tasks and data. The last decades have seen the introduction of query languages and models to represent graphs from the perspective of managing computational resources and solving graph-oriented problems.

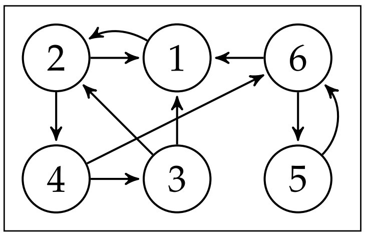
There are different ways by which to computationally represent the edge set E of a graph. They may be represented as an adjacency list, or perhaps an adjacency matrix. For the later case, the following matrix A is an example:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{2.1}$$

In this example, the first row of A represents the outgoing edges of vertex 1, which is connected to vertices 2 and 5. It is common in the literature (Cormen, Leiserson, Rivest, and Stein 2009, Chapter 22) to use the subscript notation $A_{i,j}$ to refer to the presence of a specific edge in matrix A (the notation is relevant for theoretical purposes even if using another type of representation) linking vertex i to vertex j:

$$A_{i,j} = \begin{cases} 1 \text{ if there is an edge from } i \text{ to } j, \\ 0 \text{ otherwise.} \end{cases} \tag{2.2}$$

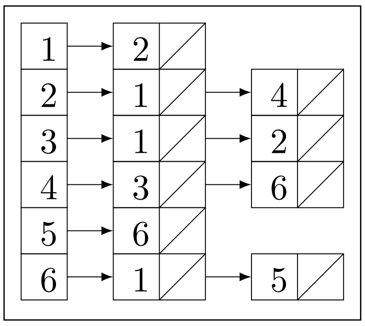
We can thus write $A_{1,5} = 1$ to state there is an edge from vertex 1 to vertex 5. Matrix A also takes on a particular configuration depending on the graph being directed or undirected. In the later case, there is no explicit sense of source or target of an edge, leading to symmetry in matrix A. There are pros and cons to using these representations. Traditionally, the edge list would be more appropriate to represent a low-density



(a) Sample graph G

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(b) Adjacency matrix A



(c) Adjacency list

$$[1 \,|\, 0 \quad 3 \,|\, 0 \quad 1 \,|\, 2 \quad 5 \,|\, 5 \,|\, 0 \quad 4]\ [0 \quad 1 \quad 3 \quad 5 \quad 7 \quad 8 \quad 10]$$

(d) Compressed sparse row

Figure 2.1: Simple computational representations of sample directed graph G shown in (a).

(sparse) graph ($m << n$) where most vertices are not connected. In the context of a sparse graph, the absence of an edge in the graph is represented by its omission from the edge list (or a value of zero in the corresponding position of an adjacency matrix A in case that is the chosen representation). It would be wasteful of storage space to explicitly represent information indicating the non-existence of elements. However, if a graph is dense ($m \sim \frac{n(n-1)}{2}$), then most vertices are connected and it is more efficient to represent E with an adjacency matrix A. This is not merely a trade-off in space. Matrices and lists, as data structures, have different computational complexities pertaining the time for certain operations (such as accessing a vertex – it is done by indexes in a matrix in $O(1)$ but it may take longer on a list due to traversals). This means that it is not mandatory to use one specific type of representation depending exclusively on the density of a graph.

Additionally, there are more complex data structures to represent E if the use-case calls for it. An instance of such a structure is the compressed sparse row (CSR) or column (CSC) representation, where a matrix A may be represented as a sequence of sequences. Each of the sequences contains only the non-zero elements, with each element stored as a pair – the actual value of the element and the index in the row of the matrix in the CSR representation (or column, in the case of CSC).

2.1.1 Computational Representations

We show in Figure 2.1 an example of different representations for the same graph. Figure 2.1a shows a sample graph G, for which the adjacency matrix is shown in Figure 2.1b and the corresponding adjacency list in Figure 2.1c. The first row of the adjacency matrix A represents the outgoing edges of vertex 1, which is connected to vertex

2. Matrix A also takes on a particular configuration depending on the graph being directed or undirected. In the later case, there is no explicit sense of source or target of an edge, leading to symmetry in matrix A. Implementations of graph processing systems often represent undirected graphs as directed graphs such that the undirected edge between a pair of vertices is represented by two directed edges in opposite directions between the vertices of the pair.

There are more space-efficient ways to represent a graph, and they become a necessity when exploring the realm of big graphs. The choice between an adjacency list or matrix is bound to the density of the graph. But to justify other representation types, factors such as graph size, storage limitations and performance requirements need special focus. The compressed sparse row (CSR), also known as the *Yale format*, is able to represent a matrix using three arrays: one containing non-zero values; the second containing the extents of rows; the third storing column indices. Figure 2.1d shows a representation in this format (we omit the array containing the non-zero values as they are all one in this case).

Let us consider that indices are zero-based. The array $[1 \mid 0 \;\; 3 \mid 0 \;\; 1 \mid 2 \;\; 5 \mid 5 \mid 0 \;\; 4]$ on the left side is the column index array, where the pipe character $\mid$ separates groups which belong to each row of A. The second row (with index 1) of matrix A has elements at position indexes 0 and 3 in A. Therefore, the second group of the column index array has elements $[0, 3]$. The array $[0 \;\; 1 \;\; 3 \;\; 5 \;\; 7 \;\; 8 \;\; 10]$ on the right is the row index array which has one element per row in matrix A and an element which is the count of non-zero elements ($|E|$) of A at the end of the array (there are variations without this count). For a given row i, it encodes the start index of the row group in the column index array (on the left in Figure 2.1d). This way, for example, the second row of matrix A (Figure 2.1b) has row index 1 in A. Then, looking at the row index array (the one on the right), as the second row of matrix A has row index 1, we access the elements with indices $[1, 2]$ in the row index array, which returns the pair $(1, 3)$, indicating that the second row (index one) of A is represented in the column index array starting (inclusive) at index 1 and ending at index 3 (exclusive). If we look at the column index array and check the elements from index 1 (inclusive) to 3 (exclusive), we get the set of values $\{0, 3\}$. And if we look at the second row in A, column index 0 and column index 3 are exactly the positions of the edges in A for that row. Generally, for a matrix M's row index i, we access indices $[i, i+1]$ in the row index array, and the returned pair dictates the starting (inclusive) and ending (exclusive) index interval in the column index array. The set of elements in that interval in the column index array contains the indices of the columns with value 1 for row index i in M. We point the reader to (Buluç, Fineman, Frigo, Gilbert, and Leiserson 2009) for details on its representation and construction. There is also the compressed sparse column (CSC), which is similar but focused on the columns, as the name suggests.

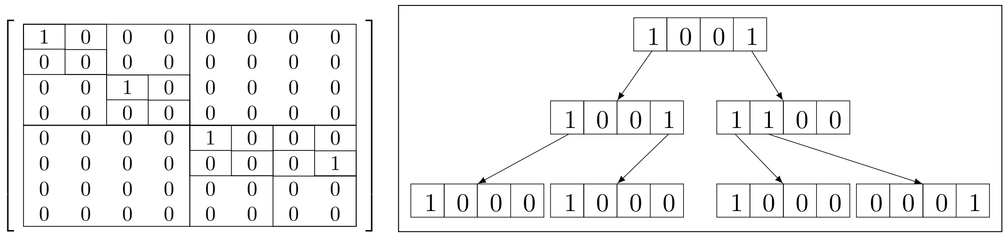
2.1.2 Advanced Compression Techniques

Other approaches take advantage of domain-specific properties of graphs. Such is the case of `WebGraph` (Boldi and Vigna 2004b), which exploits certain properties of web graphs to represent them with increased compression. An important property they exploit is *locality*, as many links stay within the same domain, that is, if the web graph is lexicographically ordered, most links point close by. Another property is *similarity*: pages that are close by in the lexicographical order are likely to have sets of neighbours that are similar. The study performed with `WebGraph` also highlighted, among other facts, the following: similarity was found to be much more concentrated than previously thought; consecutivity is common regarding web graphs. The properties of ordering (and different techniques to produce orderings) have also been exploited by the same authors to obtain compression with social networks. `WebGraph` was used in an extensive analysis of many different data sets, which were made available online by the `Laboratory for Web Algorithmics` (Boldi, Codenotti, Santini, and Vigna 2004; Boldi and Vigna 2004b; Boldi, Rosa, Santini, and Vigna 2011; Boldi, Marino, Santini, and Vigna 2014; Laboratory for Web Algorithmics 2020).

The k^2-tree is another data structure employed to represent and efficiently store graphs (Brisaboa, Ladra, and Navarro 2014). It may be used to represent static graphs and binary relations in general, such as web graphs, social networks and `RDF` data sets by internally using compressed bit vectors. It is a data structure that also efficiently matches the properties of sparseness and clustering of web graphs. A dynamic version of the k^2-tree structure was proposed for this purpose (Brisaboa, Cerdeira-Pena, de Bernardo, and Navarro 2017). Using compact representations of dynamic bit vectors to implement this data structure, the k^2-tree was used to provide a compact representation for dynamic graphs. However, this representation with dynamic compact bit vectors suffers from a known bottleneck in compressed dynamic indexing (Navarro 2016). It suffers a logarithmic slowdown from adopting dynamic bit vectors. A recent comparative study on the graph operations supported by different k^2-tree implementations has also been performed as part of this research effort (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020). This work also presented an innovative take on implementing dynamic graphs by employing the k^2-tree data structure with a technique that confers dynamic properties to a document collection (Munro, Nekrich, and Vitter 2015), avoiding the bottleneck in compressed dynamic indexing.

To construct the k^2-tree data structure, conceptually, we recursively subdivide each block of a graph's adjacency matrix A until we reach the level of individual cells of the matrix. The idea is to divide (following an `MX-Quadtree` strategy (Samet 2006, Section 1.4.2.1)) the matrix in blocks and then assign 0 to the block if it only contains zeros (no edges) or 1 if it contains at least an edge. We show in Figure 2.2 a sample adjacency matrix on the left and the corresponding k^2-tree representation of the decomposition. This representation of the adjacency matrix is actually a k^2-tree of height $h = \lceil \log_k n \rceil$, where $n = |V|$ and each node contains a single bit of data. It is 1 for internal nodes and 0 for leaves, except for the last level, in which all nodes are leaves representing values from the adjacency matrix.

Another proposal, `Log(Graph)` (Besta, Stanojevic, Zivic, Singh, Hoerold, and Hoefler 2018) is a graph representation that combines high compression ratios with low



(a) Decomposition ($k = 2$) (b) k^2-tree ($h = \lceil \log_k n \rceil = \lceil \log_2 8 \rceil = 3$)

Figure 2.2: A sample adjacency matrix ($n = |V| = 8$) and corresponding k^2-tree representation.

overhead to enable competitive processing performance while making use of compression. It achieved compression ratios similar to `WebGraph` while reaching speedups of more than 2x. The authors achieve results by applying logarithm-based approaches to different graph elements. They describe its application on *fine elements* of the adjacency array (the basis of `Log(Graph)`: vertex IDs, offsets and edge weights. From information theory, the authors note that a simple storage lower bound can be the number of possible instances of an entity, meaning the number of bits required to distinguish them. Using this type of awareness on the different elements that represent an adjacency array and by incorporating bit vectors, the authors present a `C++` library for the development, analysis and comparison of graph representations composed of the many schemes described in their work.

Relevant techniques for graph compression have been proposed in the literature across decades (Feder and Motwani 1995; Buehrer and Chellapilla 2008; Apostolico and Drovandi 2009; Kang and Faloutsos 2011; Fan, Li, Wang, and Wu 2012; Lim, Kang, and Faloutsos 2014), with the `WebGraph` framework (Boldi and Vigna 2004a; Boldi and Vigna 2004b) as one of the most well-known, and more recently the k^2-tree structure (Hernández and Navarro 2014; Brisaboa, Ladra, and Navarro 2014; Brisaboa, de Bernardo, Gutiérrez, Ladra, Penabad, and Troncoso 2015; Gagie, González-Nova, Ladra, Navarro, and Seco 2015). Only in the more recent years was the focus cast on being able to represent big graphs with compression while allowing for updates. Furthermore, if we add the possibility of dynamism of the data (the graph is no longer a static object that one wishes to analyse) to the factors guiding representation choice, then it makes sense to think about how to represent a big graph in common hardware not only for storage purposes but also for efficient access with mutability.

2.1.3 Classifying Graphs

We may classify graphs according to many characteristics. A graph is considered dense if its number of edges is close to the maximum possible number. As an example, for simple graphs, we may write their density D as:

Undirected graph:

$$D = \frac{2|E|}{|V|(|V|-1)} \tag{2.3}$$

Directed graph:

$$D = \frac{|E|}{|V|(|V|-1)} \tag{2.4}$$

Graphs also have other properties which vary depending on the topology and configuration of the network:

- Diameter, which is the greatest length out of all the pair-wise shortest paths in a graph.
- Girth, the length of the shortest cycle which exists in the graph.
- Betweenness centrality, a measure of centrality which is based on shortest paths. This measure, for each vertex, is the number of shortest paths passing through it.
- Modularity, a measure of the decomposition of the graph into groups.
- Clustering coefficient, a measure of the way vertices tend to form clusters.

Delving deeper into *finding groups within graphs*, we may further detail the concept of modularity and the relation between it and clustering. In a network represented with a graph, one may define a community as a dense group of interconnected vertices where the group is only connected sparsely to the rest of the network. Each community may have different values for the previously-mentioned properties compared to the average network. Community detection is an example of finding groups in a graph, focusing on high average degree subgraphs. This is an instance of clustering, which by itself encompasses a broader range of methods (further detailed in Section 2.2.3). Both modularity and clustering coefficient are formulations of the measure of clustering in the graph. Modularity considers edge densities in clusters compared against edge densities between the clusters, while the clustering coefficient is the fraction of paths of length two in the network that are closed (Newman 2010).

2.2 Known Graph Problems

In this section, an overview of known graph problems is provided. For some of the problems, different algorithms or approaches are shown. The graph problems herein described are well-known and studied in the literature (Cormen, Leiserson, Rivest, and Stein 2009). Section 2.2.2 highlights the importance of establishing an order among the vertices of a graph by means of ranking. Measures of centrality are covered and then the focus is cast on the famous case of PageRank (Page, Brin, Motwani, and Winograd 1999), which we used as a use-case algorithm for our research on approximate graph processing, presented in Chapter 4. Section 2.2.3 analyses the problem of community detection and graph clustering. We applied one technique for community detection to the task of leader placement in cloud network micro-clouds. This application is detailed in Chapter 5 and highlights the potential made possible by considering existing problems in graph-oriented perspectives. Instances of these classic problem's algorithms were used to evaluate our proposal of the dynamic k^2-tree graph representation, which is described in Chapter 6.

2.2.1 Research Context

Notations and computational representations for graphs are essential to describe the logic operating on them. However, pertaining to graph problems and algorithms themselves, additional motivational context is warranted.

Vertex centrality, whose more well-known algorithms we explore in Section 2.2.2, is relevant to the techniques proposed in GELLY-SCHEDULING. In it, we use different heuristics of network nodes for leader election within communities through a graph-oriented approach. Multiple nodes compete for the position of leader, and ultimately we compute an ordering of the nodes (their importance) from these heuristics. It is a concept that drew inspiration from known vertex centrality algorithms such as PageRank, which itself builds on more primitive forms of centrality measurement.

Exploring approximate graph processing techniques, and due to the synergy and applicability that the topic has with GELLY-SCHEDULING, we focused on a new technique to explore performance and accuracy trade-offs for vertex centrality algorithms in VEILGRAPH. As part of this work, and delving into lower-level implementation details, an analysis of vertex centrality algorithms and how they function becomes even more important. Due to these reasons, and for completeness, as these algorithms provide insight into how we developed GELLY-SCHEDULING and VEILGRAPH, we expand on them in Section 2.2.2.

We also detail clustering and community detection algorithms in Section 2.2.3. These are relevant as we use such algorithms in the community definition phase of GELLY-SCHEDULING as well as specific evaluation algorithms to assess the performance of our k^2-tree compact graph representation. Furthermore, as part of our state-of-the-art analysis on techniques to maintain graph representations in stream processing, we found path-related contributions to be relevant (Kalavri, Simas, and Logothetis 2016).

The third type of problem we detail is that of pathfinding. We employ different pathfinding algorithms in the evaluation of our k^2-tree compact graph representation. This research contribution included a diverse set of algorithms for the benchmarking of the graph algorithm library we developed, and due to this reason we detail them in Section 2.2.4.

2.2.2 Vertex Ranking

Establishing relative importance between vertices in a graph is a problem that has drawn much attention. There are applications of graph theory to real-world scenarios where it is desired to know the most important vertices. But the definition of importance will vary depending on the context. For example, in social network analysis, there may be interest in knowing which users are able to propagate information with maximum visibility (so the most important users are potentially those with the bigger audiences). Depending on the problem and data exploration needs, the criteria may vary, but the desire remains to place the users (vertices) on an axis, typically from most to least important.

This implies the attribution of an individual score to each vertex, according to some reasoning of criteria. Such a task has been explored in the past, taking the form of vertex centrality algorithms, or ranking of vertices. Scoring schemes known as centrality measures exist with different levels of refinement. One may directly attribute the *score* of a vertex to its degree (degree centrality). More refined methods of measuring the score of a vertex account for its number of incoming edges and the weights of their respective sources. Eigen centrality (Newman 2010) and Katz centrality (Katz 1953) are important approaches. In the literature, one may find these and other centrality measures (Koschützki, Lehmann, Peeters, Richter, Tenfelde-Podehl, and Zlotowski 2005; Qi, Fuller, Wu, Wu, and Zhang 2012), with variations in both their definitions and targeted applications such as electrical power grid analysis (Wang, Scaglione, and Thomas 2010) and influence in social networks (Grando, Noble, and Lamb 2016), as well as the algorithms employed to compute them. Perhaps the most well-known measure is PageRank (Newman 2010), of which there are many applications and refinements (Berkhin 2005).

The original PageRank algorithm (Page, Brin, Motwani, and Winograd 1999) initializes all vertices with the same value. In this document, we focus on a vertex-centric (an approach to define computation over graphs and detailed in Chapter 3) implementation of PageRank, having used it as a case-study, where for each iteration, each vertex u sends its value (divided by its outgoing degree) to each outgoing edge. A vertex v defines its score as the sum of values received from its incoming edges, multiplied by a constant factor β and then summed with a constant value $(1 - \beta)$ with $0 \leq \beta \leq 1$. PageRank, based on the random surfer model (Blum, Chan, and Rwebangira 2006), uses β as a dampening factor. If the PageRank of a vertex represents the probability that a web surfer would visit the page, then the β factor represents the chances of the surfer switching to another random page. This process terminates when a maximum number of iterations has been reached, or when the values have converged within a predefined limit (David and Jon 2010). Chapter 4 analyses details related to how PageRank may be implemented and how we evaluated it under our approximate processing model in VEILGRAPH.

2.2.3 Clustering and Community Detection

Graph clustering is a problem which arises in several domains, with the goal of finding groups that are homogeneous (in the sense that the vertices contained in a graph cluster probably share common properties) and under certain separation criteria. Algorithms for this may try to optimize for a specific set of parameters or try to apply knowledge about the underlying data. Clustering may vary depending on whether *soft* or *hard* clustering is considered. This is usually formalized by defining a quality measure which may differ with the problem domain. When analysing graph community structure, an attempt is made to evaluate the separation of sparsely connected dense subgraphs from each other. Formally, let $G = (V, E)$ be an undirected graph. A non-overlapping clustering or partition P of G is a collection of sets $\{V_1, ..., V_k\}$, with $k \in \mathbb{N}$, such that $V_i \neq \emptyset$ for $1 \leq i \leq k$, $V_i \cap V_j = \emptyset$ for $1 \leq i < j \leq k$, and $\bigcup_{1 \leq i \leq k} V_i = V$.

Ideally, a system would be able to compute the changes impacting the graph with *enough speed*, that is, so that the human end-users of the system do not suffer a negative experience (typically manifested by their awareness of the computational delay). Likewise, the clients of these computations (performed in *Software-as-a-Service* infrastructures for example) could be automated software components which must have their data queries solved under strict time constraints. Furthermore, if the graph represents a social network, such as Facebook with its vast user base and colossal amount of edges and graph properties (Ching, Edunov, Kabiljo, Logothetis, and Muthukrishnan 2015), it is highly unrealistic to assume that graph operations occur sequentially, which means that an operation will have consequences in terms of both the amount of data affected and the way different agents interact in the graph. The degree to which hardware resources can be employed for parallel processing in graph clustering algorithms depends heavily on the nature of the computation performed (and is further discussed in Chapter 3). This section presents some of the most relevant graph clustering methods in the literature:

- **Louvain Method**. A greedy optimization method, able to identify communities (again, a subset of clustering variants) in large networks. For an undirected graph $G = (V, E)$, this technique has an apparent [1] computational complexity of $O(|V|\ log\ |V|)$; it requires effort *almost* linear in the $|V|$ number of vertices of the graph. It is not unusual to encounter the term *Louvain Modularity* in the literature. That is the name of the metric which the Louvain Method attempts to optimize (Blondel, loup Guillaume, Lambiotte, and Lefebvre 2008). It has been used to explore the problem of partitioning social networks onto different machines and to identify dynamic communities in dynamic social networks (inherent in mobile networks). The modularity of a partition is defined as a scalar value in the interval $[-1, 1]$, measuring the amount of edges contained inside communities versus the edges across communities. Modularity in fact measures the fraction of edges in the network that connect within-community edges minus the expected value of the same quantity in a graph network with the same community divisions, but random vertex connections. If the number of edges connecting same-community vertices is much lower than that of the random network, modularity can become negative. However, most methods that calculate modularity typically do not produce values lower than those obtained when each vertex is its own community (and that scenario would yield a negative value). Values are usually between the value of that scenario and the upper bound

[1] No formal proof exists in the literature, this is based on experimental results by researchers.

of 1. In the literature one may find a definition such as the following:

$$Q = \frac{1}{2m} \sum_{i,j} [A_{ij} - \frac{k_i k_j}{2m}] \delta(c_i, c_j) \tag{2.5}$$

In it, m and k_i are defined as:

$$m = \frac{1}{2} \sum_{ij} A_{ij} \qquad k_i = \sum_{j} A_{ij} \tag{2.6}$$

Where $\mathbf{A}$ is a weight matrix with element A_{ij} representing the weight of an edge between a vertex i and a vertex j, m is half of the sum of all elements of A and k_i is the sum of the weights of the edges attached to vertex i (the sum of row i of matrix $\mathbf{A}$). δ is the Dirac delta function. Modularity Q has been used as both a quality assessment of partitioning methods and also as an optimization problem in itself. The method works by iterating until no further increases in modularity occur. Initially, each vertex is its own community. For every vertex i, the potential gain in the modularity from moving a vertex out of its own community (thus acting as an isolated vertex) and into each of its neighbours' communities is calculated through a similar formula. In the literature, this main method of operation is said to be using a heuristic called *local moving heuristic* (Waltman and van Eck 2013). Essentially, if vertex i is moved into [2] a community C, the change in modularity may be obtained with the following expression:

$$\Delta\, Q = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m} \right)^2 - \left(\frac{k_i}{2m} \right)^2 \right] \tag{2.7}$$

For this expression, $\sum_{in}$ is the sum of the weights of the edges in community C and $\sum_{tot}$ is the sum of the weights of the edges incident to vertices in C and $k_{i,in}$ is the sum of the weights of edges between vertex i and vertices of community C. Other symbols retain the previous meaning. Phase two consists in building a new network where vertices are new communities found during the previous phase. Weights of edges between new vertices (which are the communities established during phase one) are the sum of weight of edges between vertices in the two corresponding communities.

As shown in Figure 2.3, a visual notion of hierarchy emerges as the algorithm progresses. This algorithm is unsupervised and it is considered fast; albeit proof of linearity is missing, simulations on large ad hoc modular networks suggest its complexity is linear on typical and sparse data (Blondel, loup Guillaume, Lambiotte, and Lefebvre 2008).

The aforementioned method, as a modularity optimization technique, incurs resolution limitations regarding the size of communities, as noted in (Fortunato and

[2]The removal of a vertex i from a community C also has a similar formula.

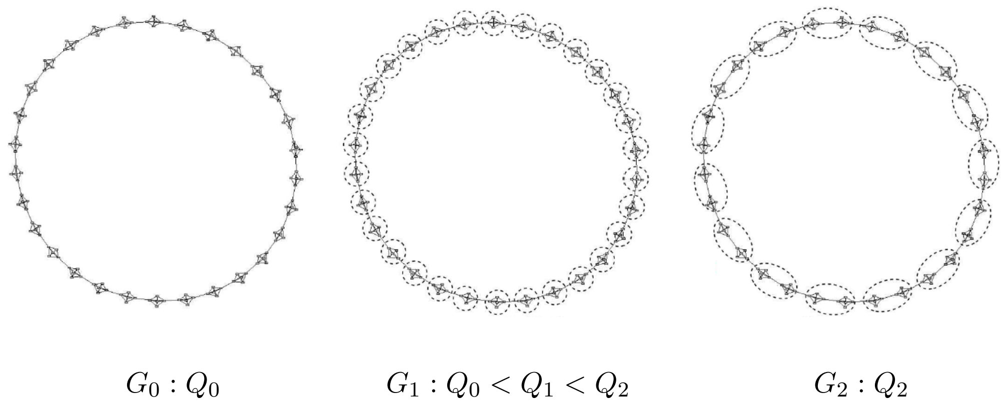


Figure 2.3: Repetition of the Louvain Method for the 30-clique example shown in Blondel, loup Guillaume, Lambiotte, and Lefebvre (2008). After the first passage of the two phases of the algorithm, there is graph G_1 with thirty communities. Executing one more phase leads to even *bigger vertices* in graph G_2. Visually, iteration of the method occurs from left to right, with ever-increasing modularity: $Q_0 < Q_1 < Q_2$.

Barthélemy 2007; Fortunato and Hric 2016). However, it is important to retain that this method yields different community resolution levels as iterations progress. As such, and as its authors suggest, the sequence of iterations produces heterogeneous community configurations, depending on the execution step. This could allow end-users to *zoom in* the network and observe it with a specific resolution, depending on how much the two phases are iterated.

- **Label Propagation**. Label propagation algorithms do not on their own involve modularity optimization. Their application for community detection is therefore scale-independent and not affected by the previously mentioned community size resolution limit. Label propagation typically starts with each vertex being associated to a unique label. In every iteration, the label of each vertex is updated by choosing the label that most of its neighbours have (maximal label). Tie-breaking between multiple maximal labels can be done with random label picking. An example of a basic label propagation algorithm proposal can be found in (Leung, Hui, Liò, and Crowcroft 2009).

  In more elaborate variants of label propagation, score measures may be employed to account for the weights of the edges (Raghavan, Albert, and Kumara 2007). Scoring was incorporated to deal with *epidemic labels* (covering for example over 50% of the vertex count), which override smaller communities lacking strong-enough edges.

  A variant of label propagation is the Layered Label Propagation (LLP) method (Boldi, Rosa, Santini, and Vigna 2011), devised to tackle the problem of resolution limits. It verified the hypothesis posed by the authors of the Louvain Method – they suggested the generation of different community resolution hierarchy levels, depending on the execution step. LLP is a parameter-free scalable clustering algorithm that can reorder large graphs, with vertex count $|V|$ in the order of billions. Originally, it was devised whilst exploring ways to improve graph compression (this work was a ramification of prior research on web graph compression) by harnessing graph ordering. A partition induces an ordering, and the appropriateness of the partition may be measured by comparing an original order with the possible refinements brought about by the induced ordering. For a given graph $G = (V, E)$ and a partitioning $\mathscr{U}$, the entropy as a function of the partition (please refer to (Boldi, Rosa, Santini, and Vigna 2011) for additional in-depth detail on mathematical clarity) is defined as:

$$\mathbb{H}(\mathscr{U}) = -\sum_{i=0}^{R} \frac{|\mathscr{U}_i|}{|V|} log\left(\frac{|\mathscr{U}_i|}{|V|}\right) \tag{2.8}$$

Each sub-component of the partition is identified by $\mathscr{U}_i$ and R is the number of subcomponents. The contextual basis of Layered Label Propagation was the web graph, in which vertices represent web pages (URLs). Considering that, it was found in previous work (Boldi and Vigna 2004b) that lexicographical ordering of vertices (based on the URLs) yielded better compression capabilities. In that regard, and to assess the power of Layered Label Propagation (and other methods which produce orderings), the authors make use of another statistical artifice, the variation of information between a given intrinsic partition $\mathscr{H}$ and a respective refinement based on an ordering π (which translates into a partition represented as $\mathscr{H}_{|\pi}$). After some manipulations, they arrive at the following formula:

$$\mathbb{VI}(\mathscr{H}, \mathscr{H}_{|\pi}) = \mathbb{H}(\mathscr{H}_{|\pi}) - \mathbb{H}(\mathscr{H}) \tag{2.9}$$

The core principle of this heuristic is that a lower variation of information implies that the ordering is more *desirable* or *of better quality*. The major fact pertaining this method is that initially it was motivated by the study of web graphs, but it was also applied to social network data sets. Making use of the previous formulations, and based on generic label propagation algorithms in the literature, the Layered Label Propagation algorithm was implemented. It is an iterative algorithm that produces a sequence of vertex orderings. For each iteration, the Absolute Potts Model (APM) algorithm (Ronhovde and Nussinov 2010) is executed. APM has a resolution parameter γ which describes a specific community resolution of the graph. Intuitively, as observed by the authors, optimality as a notion does not apply to γ: different values for it simply produce descriptions of the graph at various resolution levels. Lower values of γ showcase a coarser graph structure with sparser and bigger clusters, and increasing the value produces smaller and denser clusters, uncovering a finer structure. The algorithm is such that it produces an order of the graph keeping vertices with the same label close to one another in the ordering. For K iterations, each iteration i will be subjected to a chosen γ_i. In line with the previous note on the (lacklustre) merit of optimizing γ, rather than trying to find an optimal value for each iteration, γ_i is instead uniformly drafted at random from the set $\{0\} \cup \{2^{-i}, i = 0, ..., K\}$. This conclusion was reached through experimental results; optimizing γ on each iteration was not computationally efficient.

- **K-means clustering**. Another variant of clustering, rooted in signal processing, which has the objective of partitioning n observations into k clusters, such that each observation belongs to the graph cluster with the nearest mean. The k-means is a point-assignment type of family of clustering algorithms (a heuristic-based variant is Lloyd's algorithm). It works under the assumption that the number of clusters k is known $a\ priori$ and execution occurs in the context of an Euclidean space. Despite this, methods exist which entail a trial-and-error approach to determining k. A high-level conceptual description of the algorithm is presented.

Basic k-means pseudo-code
INPUT: Graph $G = (V, E)$
OUTPUT: Collection of sets $\{P_1, P_2, ..., P_k\} : P = P_1 \cup P_2 \cup ... \cup P_k = V$, **as per Sec. 2.2.3**

```
1: $L \longleftarrow$ CHOOSE($V, k$)
2: $P \longleftarrow$ DEFINE-CLUSTERS($L, V$)
3: for each point $w \in V \setminus L$:
4:     $index \longleftarrow$ FIND-CLOSEST-CENTROID($P, L, w$)
5:     $P_{index} \longleftarrow P_{index} \cup w$                    (Expectation)
6:     UPDATE-CENTROID($P_{index}$)                           (Maximization)
```

It is suggested (Chapter 7 of (Leskovec, Rajaraman, and Ullman 2014)) that k-means should be executed with increasing values of $k = 1, 2, 4, 8, ...$ The formulation behind the suggestion is that eventually, a value v will be found such that, for variations of $k \in [v, 2v]$, there will be little change in the measure of graph cluster cohesion used. Lines five and six of the presented pseudo-code should receive particular attention: in reality, the k-means algorithm iteration cycle can be described as having an expectation step (line five) followed by a maximization step (line six). This constitutes an incarnation of the expectation-maximization algorithm (EM for short). In (Murphy 2012), the expectation-maximization introduction defines the following equation, which is the expected complete data log likelihood:

$$Q(\theta, \theta^{t-1}) = \mathbb{E}[l_c(\theta)|\mathcal{D}, \theta^{t-1}] \tag{2.10}$$

In this equation, θ represents the unknown state of nature, $\mathcal{D}$ the observed data, t is the current iteration number and Q is called the auxiliary function. [3] It makes use of the complete data log likelihood l_c, defined as:

$$l_c(\theta) \triangleq \sum_{i=1}^{N} \log p(x_i, z_i|\theta) \tag{2.11}$$

[3]Concretely, what is calculated in the expectation step are the fixed data-dependent parameters of the function Q.

The missing data is z_i, in this case representing the assignment of vertices to communities. Here z_i represents $\{q\}$, the model's parameters $\{\pi\}$ are implicit and there is a direct mapping to the amount of clusters $\{k\}$. K-nearest neighbours is a supervised learning method for classification.

- **Nearest neighbours clustering**. Delving further into clustering, it may be interpreted as a general baseline algorithm for minimizing arbitrary clustering objective functions. It is in fact a single-link clustering technique: an agglomerative (bottom-up hierarchical clustering) method. For each step, the closest pair of clusters (pair of elements in the data set in the first iteration) are chosen, with the clusters being merged. Naive versions of this algorithm have a computational complexity ranging from $O(|V|^2)$ to $O(|V|^3)$ which, coupled with its greedy (without making use of backtracking mechanisms) nature, constitute relative drawbacks. The algorithm begins with every vertex v as its own graph cluster. After $|V|-1$ iterations have ensued, a single graph cluster (binary tree) remains. It should then be *cut* at a parametrized depth to produce a specific partitioning (the result of clustering may be seen using a representation such as a dendrogram). A visual example of such a distance-specific hierarchy is displayed in Figure 2.4. A major configuration of this algorithm resides in the choice of distance function between clusters. [4] This graph cluster distance may be mathematically written as:

$$D(A, B) = \min_{a \in A, b \in B} d(a, b) \tag{2.12}$$

Where A and B are any two clusters and $d(a, b)$ represents the distance between two elements a and b of each graph cluster. The description provided so far is a naive version, sharing many similarities with Kruskal's (Kruskal 1956) minimum spanning tree algorithm. Improved methods exist: in particular, it has been pointed out in the literature that better results may be achieved by running Prim's algorithm (Prim 1957) for spanning trees before Kruskal's. The idea is to use Prim's algorithm without priority queues, using $O(|V|)$ space and $O(|V|^2)$ execution time. This results in a minimum spanning tree of the elements and distances (this tree is actually a sparse graph). Afterwards, Kruskal's algorithm may be put to use over the edges of the minimum spanning tree, finally producing the clustering; additional space requirements of $O(|V|)$ and $O(|V| \log |V|)$ ensue. Ultimately, this variant has a temporal complexity improvement, ensuring its bound is $O(|V|^2)$.

2.2.4 Pathfinding

The task of finding a path is of utmost importance, whether it serves the purpose of generating an efficient path when we wish to travel, by car or foot, from a starting point to a target destination or to find paths of minimum length between vertices. We list here some of the most well-known variants.

[4] Variants exist depending on the definition of graph cluster distance: single-link or minimum distance, complete-link or maximum distance, average distance and mean distance.

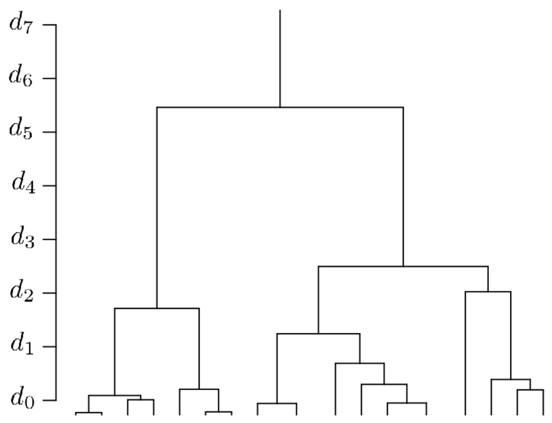


Figure 2.4: A dendrogram distance-based *cut* example which was handed a graph cluster at iteration number $|V| - 1$. As the *cutting* distance d is changed, one operates with different degrees of clustering resolution.

- **Breadth-first Search (BFS)**. A type of graph search algorithm which, starting from a specific source vertex s in a graph G, will systematically explore edges to find vertices that are accessible from s. This algorithm calculates the smallest distance from s to each vertex reachable from it. Its process of reaching vertices is based on finding all vertices within a perimeter k of s before exploring perimeter $k+1$. The worst-case running time complexity for BFS is $O(|V| + |E|)$ and space complexity is $O(|V|)$.

- **Depth-first Search (DFS)**. Similar to the previous one, but instead of exploring all vertices for a specific perimeter size k before increasing the perimeter, depth is explored as far as possible (the priority is depth rather than breadth). That is, with the process starting from a source vertex s, edges of the most recently reached vertex v are explored first. When the algorithm has explored all of the vertices of v, the search moves to the parent of v to process the parent's unexplored edges. This process continues until all of the vertices have been discovered. The worst-case running time complexity for DFS is $O(|V| + |E|)$ and space complexity is $O(|V|)$. Time and space complexities are the same as BFS.

- **Dijkstra's algorithm**. An algorithm to solve the single-source shortest-paths problem on weighted, directed graphs (with non-negative weights). It maintains a set S of vertices for which the weights of the shortest paths from the source s have been determined already. This algorithm repeatedly selects the vertex $v \in V - S$ which has the minimum shortest-path estimate and then relaxes its edges (updating estimates of the out-neighbours of v).

3 State-of-the-Art: Survey

This chapter incorporates our findings, detailed in our submission (Coimbra, Francisco, and Veiga 2021), from studying the landscape of graph processing. We firstly reference contributions providing insights on the profile of users and developers interested in graph processing. This is followed by an analysis of different approaches to defining computation over graphs - should operations be conceptually applied to edges, vertices or parts of the graph? How should systems be designed to enable parallel and distributed processing of graphs? We detail approaches on the nature of workloads found in graph processing and definitions of dynamism - what does it mean to process a dynamic graph, or what is implied by having to keep the result of computations up-to-date in the advent of the graph changing due to topological updates? Herein we explore approaches in the literature to such questions and then provide an exhaustive list of graph processing and storage architectures, with works from academia and industry grouped by architecture types.

Our survey on the landscape of graph processing was written to provide specific insight for experienced researchers and developers facing a more complex use-case, as well as awareness of important aspects of graph processing for novices. Most of these systems were analysed as part of related work of our publications and contributions.

3.1 Motivation

The recent years have seen a positive tendency in the field of all things related to graph processing. As its aspects are further explored and optimized, with new paradigms proposed, there has been a proliferation of multiple surveys (Malicevic, Roy, and Zwaenepoel 2014; Han, Daudjee, Ammar, Özsu, Wang, and Jin 2014; Kalavri, Ewen, Tzoumas, Vlassov, Markl, and Haridi 2014; Kalavri, Vlassov, and Haridi 2017; Heidari, Simmhan, Calheiros, and Buyya 2018; Sahu, Mhedhbi, Salihoglu, Lin, and Özsu 2017; Soudani, Fatemi, and Nematbakhsh 2019). They have made great contributions in systematizing the field of graph processing, by working towards a consensus of terminology and offering discussion on how to present or establish hierarchies of concepts inherent to the field. Effectively, we have seen vast contributions capturing the maturity of different challenges of graph processing and the corresponding responses developed by academia and industry.

This review of the literature highlights recent contributions in computational graph representations and systems addressing graph processing, targeting algorithms such as community detection and vertex centrality. The study was conducted as part of a larger context of big data, dynamic graphs and the emerging roles of approximate and incremental computing.

Herein we perform an exhaustive analysis of existing solutions for graph processing and storage. To provide context, we detail in Section 3.2 an analysis of the profiles of individuals interested in graph-oriented data solutions and their use-cases (Sahu, Mhedhbi, Salihoglu, Lin, and Özsu 2017). In Section 3.3 we go over known conceptual models and query languages to represent graphs from the perspective of performing queries over the semantics of the graph (e.g. *querying the graph of a social network to find all users who live in England and have three degrees of separation from the football coach José Mourinho but no less than five from Her Majesty*).

In Section 3.4 we present the different levels of granularity used in graph processing solutions to represent elements of computation and in Section 3.5 we delve into the problem of graph partitioning and its connection to enabling parallel and distributed computing. Section 3.6 gathers and explains different definitions associated to the idea of dynamism in graphs and in Section 3.7 we show two relevant workload types regarding graph processing. Afterwards we present and compare existing solutions for graph processing grouped by architectures such as single-machine solutions (Section 3.8), high-performance computing (Section 3.9) and distributed systems (Section 3.10), as well as graph databases (Section 3.11). Lastly we provide remarks on recent trends in the graph processing landscape (Section 3.12).

3.2 Profile of Developers and Researchers

There has been a previous study on the profile of practitioners in both academia and industry (Sahu, Mhedhbi, Salihoglu, Lin, and Özsu 2017). To our knowledge, it has been the first one to identify the profile of users and the types of computational tasks they need to process over graphs. Specifically, this is the result of an online survey aiming to establish: 1) the types of graph data used; 2) the computations users run on their graphs; 3) the softwares used to perform computations; 4) the major challenges faced by users when processing the graph data.

The authors share five main findings as a result of their online survey, namely that: there is **variety in the data** represented with graphs; very large graphs are present from **small to large enterprises**; the **ability to process very large graphs efficiently** is still the biggest limitation of existing software; the second most desired ability is **visualization**, related to challenges in graph query languages; relational database management systems **(RDBMSes) are still relevant** in managing and processing graphs.

This survey (Sahu, Mhedhbi, Salihoglu, Lin, and Özsu 2017) was important as it highlighted that graphs falling in the realm of big data are still the top priority for innovative solutions, thus further motivating the importance of concisely defining the landscape of graph processing and different ways in which the need to process big graphs manifests. The processing of a graph can take place from different perspectives such as vertices, edges or parts of the graph. These perspectives can more naturally

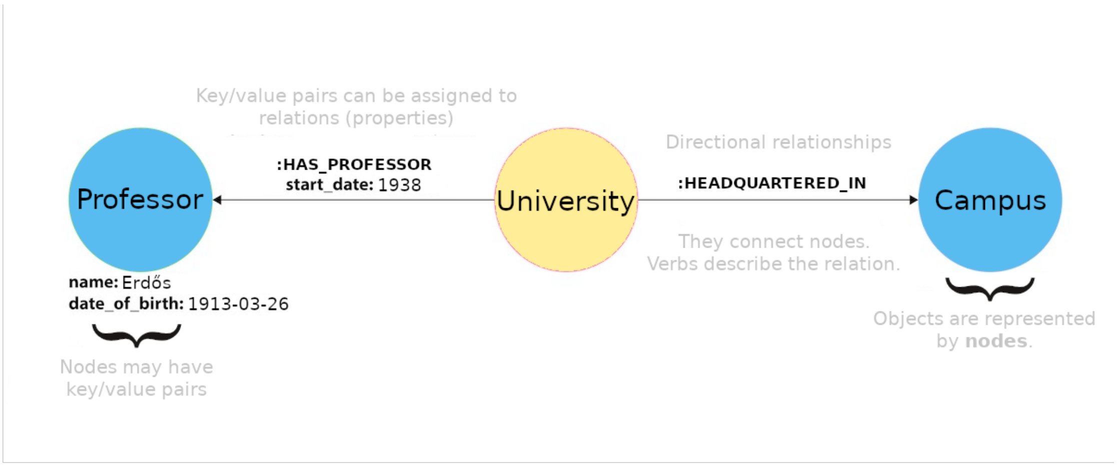

Key/value pairs can be assigned to relations (properties)
Directional relationships
Professor
:HAS_PROFESSOR
start_date: 1938
University
:HEADQUARTERED_IN
Campus
They connect nodes. Verbs describe the relation.
name: Erdős
date_of_birth: 1913-03-26
Objects are represented by nodes.
Nodes may have key/value pairs

Figure 3.1: Illustration of the property graph model, inspired by the memory of Professor Paul Erdős.

represent one type of graph computation than other.

3.3 Property Model and Query Languages

In this section we detail languages, models and subsequent standards for querying graph-based data. The property graph model defines the organization of interrelated data as nodes (vertices), relationships (edges) and the properties of both types of elements (Neo4j 2020b). Instances of this model are usually found coupled to the design of graph databases and at the level of graph query languages (Rodriguez 2015a; van Rest, Hong, Kim, Meng, and Chafi 2016; Angles 2018; Green, Junghanns, Kießling, Lindaaker, Plantikow, and Selmer 2018). Figure 3.1 shows a depiction of the properties that may be associated to vertices and edges in the context of the property graph model.

The ISO SQL Committee has accepted on September 2019 the Graph Query Language project proposal (JCC Consulting 2020), to enable SQL users to use property graph style queries on top of SQL tables. This will promote the interoperability between SQL databases and graph databases and constitutes an important mark in the approximation between graph-structured data and the databases that support it, an area that has been studied for decades (Sheng, Ozsoyoglu, and Ozsoyoglu 1999; He and Singh 2008). The language standards engineering team of `Neo4j` identify the following cornerstones regarding the growing popularity of the property graph model (Plantikow 2019):

1. An intuitive model geared for application developers.
2. Ability of rapid prototyping with schema-optionality.
3. Availability of native graph databases.

4. Declarative query languages focusing on ease of use with graph pattern matching and visual syntax, top-down linear statement order and language composability.

3.3.1 Graph Query Languages

The access to elements offered by graph databases is performed by means of specific query languages. There are even projects focused on analytics which offer the ability to explore datasets using graph query languages without actually running a graph database, such as `GRADOOP` (Junghanns, Kießling, Teichmann, Gómez, Petermann, and Rahm 2018) and `GraphFrames` (Mishra and Raman 2019). Languages such as `Cypher` even support queries on graphs which produce new graphs (for example to represent specific entities and relationships of the original graph) upon which to run further queries. Here we list some of the most relevant graph query languages (and proposals in the scope of graph query languages), projects that structure the access to graph-based data as well as open-source constructs that form their basis.

- `Cypher`. An evolving graph query language (Francis, Green, Guagliardo, Libkin, Lindaaker, Marsault, Plantikow, Rydberg, Selmer, and Taylor 2018) which debuted with `Neo4j`'s entrance in the field of graph databases. There have been efforts to adopt and develop the language in an open-source approach (Green, Junghanns, Kießling, Lindaaker, Plantikow, and Selmer 2018). `Cypher` has been an open and evolving language as part of the `openCypher` project (Horn and Rydberg 2020). The project has members involved in aspects such as synergy of engineering efforts with `Apache Spark` (Zaharia, Chowdhury, Franklin, Shenker, and Stoica 2010), a language group and even interoperability features for systems that use `Gremlin`. This graph query language heavily influenced the ISO project for creating a standard graph query language (Neo4j 2018) and has a syntax familiar to developers with knowledge of `SQL`. `Cypher` queries may be run on the following databases: `Neo4j`, `Graphflow` (Kankanamge, Sahu, Mhedbhi, Chen, and Salihoglu 2017), `RedisGraph` (Cailliau, Davis, Gadepally, Kepner, Lipman, Lovitz, and Ouaknine 2019), `SAP Hana Graph` (Rudolf, Paradies, Bornhövd, and Lehner 2013) and all databases where `Gremlin` is supported (Neo4j 2019). This language is also used to express computation in `GRADOOP` (Junghanns, Kießling, Averbuch, Petermann, and Rahm 2017) and the `Python Ruruki` (Optiver 2016) lightweight in-memory graph database.

- `Gremlin`. A graph traversal machine and language, developed in scope of the `Apache TinkerPop` project (Foundation 2019b). This project's development and growth was promoted by the now-defunct `Titan` graph database (Aurelius 2015) which was forked into the open-source `JanusGraph` database (JanusGraph Authors 2017) and the commercial `DataStax Enterprise Graph` solution. The traversal machine of `Gremlin` is defined as a set of three components (Rodriguez 2015a): the data represented by a graph G, a traversal Ψ (instructions) which consists of a tree of functions called *steps*; a set of traversers T (read/write heads). With this composition, `Gremlin` and its traversal machine enable the exploration of multi-dimensional structures that *model a heterogeneous set of "things" related to each other in a heterogeneous set of ways* as detailed in (Rodriguez 2015a; Rodriguez 2015b). A traversal evaluated against a graph may generate billions of traversers, even on small graphs, due to the exponential growth of the number of paths that exist with each step the traversers take. Databases supporting `Gremlin` include `OrientDB` (Tesoriero 2013), `Neo4j` (Webber 2012), `DataStax Enterprise` (DataStax, Inc. 2016), `InfiniteGraph` (Objectivity 2016), `JanusGraph` (JanusGraph Authors 2017), `Azure Cosmos DB` (Paz 2018) and `Amazon Neptune` (Bebee, Choi, Gupta, Gutmans, Khandelwal, Kiran, Mallidi, McGaughy, Personick, Rajan, et al. 2018), while the graph processing systems that allow its use are `Apache Giraph` (Ching 2013) and `Apache Spark` (Armbrust, Xin, Lian, Huai, Liu, Bradley, Meng, Kaftan, Franklin, Ghodsi, and Zaharia 2015).

- `SPARQL`. The standard query language for `RDF` data (triplets), also known as the query language for the semantic web. We mention it due to its graph pattern matching capability (Pérez, Arenas, and Gutierrez 2009) and scalability potential of querying large `RDF` graphs (Huang, Abadi, and Ren 2011). As `RDF` is a direct labelled graph data format, `SPARQL` becomes a language for graph-matching. Its queries have three components: a *pattern matching part* allowing for pattern unions, nesting, filtering values of matchings and choosing the data source to match by a pattern; a *solution modifier* to allow modifications to the computed output of the pattern, such as applying operators as projections, orderings, limits and distinct; the *output*, which can be binary answers, selections of values for variables that matched the patterns, construction of new `RDF` data from the values or descriptions of resources. While graph databases may not necessarily be triplet stores, the graph query languages they support may allow for example that the `RDF`-specific semantics of a `SPARQL` query may be translated to `Cypher`, `Gremlin` or another language. `SPARQL` is also supported (analytics) over `GraphX` (Schätzle, Przyjaciel-Zablocki, Berberich, and Lausen 2015) and the higher-level graph analytics tool `GraphFrames` (Bahrami, Gulati, and Abulaish 2017). Among the graph databases that support this language we have `Amazon Neptune` (Bebee, Choi, Gupta, Gutmans, Khandelwal, Kiran, Mallidi, McGaughy, Personick, Rajan, et al. 2018) and `AllegroGraph` (Inc. 1984) (the later two more oriented to the purpose of `RDF`).

- `GraphQL`. A framework developed and internally used at Facebook for years before its reference implementation was released as open-source (Facebook, Inc. 2016). It introduced a new type of web-based interface for data access. As a framework, one of its core components is a query language for expressing data retrieval requests sent to web servers that are `GraphQL`-aware. The queries are syntactically similar to `JavaScript Object Notation (JSON)`. The `GraphQL` specification implicitly assumes a logical data model implemented as a virtual, graph-based view over an underlying database management system (Hartig and Pérez 2017). It has been studied with the semantics of its queries formalized as a labelled-graph data model and the total size of a GraphQL response shown to be computable in polynomial time (Hartig and Pérez 2018). GraphQL is more than a query language - it defines a contract between the back-end and front-end over an agreed-upon type system, forming an application data model as a graph. It is useful as it proposes a decoupling between the back-end and front-end, allowing each component to be changed independently of the other. For example, to serve queries in the graph, the data in the back-end could come from databases (e.g. Neo4j as the back-end serving GraphQL queries received at a web endpoint (Neo4j 2020a)), in-memory representations or other APIs.

- PGQL. The Property Graph Query Language, based on the paradigm of graph pattern matching (van Rest, Hong, Kim, Meng, and Chafi 2016). It closely follows the syntactic structures of SQL, providing regular path queries with conditions on labels and properties to enable reachability and path finding queries. The data types it defines are the intrinsic *vertex*, *edge*, *path* and also an intrinsic *graph* type, allowing for graph construction and query composition. It was motivated by the fact that SPARQL is the RDF standard query language, thus imposing that graphs be represented as a set of triples (or edges), and by Cypher's lack of support for regular path queries and graph construction as fundamental graph querying functionalities. PGQL also provides tabular output, allowing its queries to be naturally nested inside SQL queries, allowing for easy integration into existing database technology. It is used in Oracle's products.

- G-Log. A declarative query language on graphs which was designed to combine the expressiveness of logic, the modelling of complex objects with identity and the representations enabled by graphs (Paredaens, Peelman, and Tanca 1995). The authors describe it as a *deductive language for complex objects with identity*, with a data model that captures the modelling capabilities of object-oriented languages, although lacking their typical data abstraction features which are related to system dynamism. They claim G-Log may be seen as a graph-based symbolism for first-order logic and they prove that all sentences of first-order logic may be written in G-Log. Secondly, they define the semantics of the language for database query evaluation. Lastly, the authors state that due to being *a very powerful language*, its computation could be unnecessarily inefficient in the most general case. We mention G-Log as it is historically relevant due to highlighting the importance and expressiveness of graph-based data models in manipulating the relationships between data.

A study on these aspects inherent to graph-structured data has been performed using different data models (RDF, property graph and relational model) and a sample of systems (RDF: Virtuoso (Erling 2012), TripleRush (Stutz, Verman, Fischer, and Bernstein 2013); graph databases: Neo4j, Sparksee (Martinez-Bazan, Gomez-Villamor, and Escale-Claveras 2011); relational database: Virtuoso) that offer the models (Gubichev and Then 2014). These systems were compared using LUBM, a well-known and used synthetic RDF benchmark (Guo, Pan, and Heflin 2005).

The authors observe there is greater verbosity when implementing queries in Sparksee due to its design of delegating the execution plan implementation to the developer, as opposed to using declarative languages like Cypher. They also reiterate the importance of cost-based query optimization within query engines. Query patterns used in the literature fall in the following categories (van Rest, Hong, Kim, Meng, and Chafi 2016; Gubichev and Then 2014): single triple patterns satisfying a given condition; matches on the vertices that are adjacent and edges that are incident to a given vertex; triangles, which look for three vertices adjacent to each other; paths of fixed or variable length.

Query languages for graphs are not a recent concept. This section presents an overview of the most relevant ones found in the literature to illustrate the existing desire for high-level tools to describe the manipulation of graphs. In our literature review, we go over many systems for the processing and storage of graphs. We note that many such systems offer developers methods to operate on graphs through APIs in popular programming languages.

However, we point out that there are systems which have added support for graph query languages in some form or another (such as the aforementioned GRADOOP and GraphFrames) to facilitate their use by more analysts and developers. From low-level implementations, to using APIs all the way to high-level query languages, there is progress to be made in enabling interoperability between these levels, whether to enable an existing system to be used by those who only know a graph query language, or to build a common API over low-level functions to allow developers to more easily switch components in a system.

3.4 Graph Processing: Computational Units and Models

Here we detail the most relevant paradigms and computational units used to express computation in graph processing systems. Programming models for graph processing have been studied and documented in the literature (Kalavri, Vlassov, and Haridi 2017; Heidari, Simmhan, Calheiros, and Buyya 2018). They define properties such as the granularity of the unit of computation, how to distribute it across the cluster and how communication is performed to synchronize computational state across machines.

3.4.1 Unit: Vertex-Centric (TLAV)

The vertex-centric paradigm, also known as *think-like-a-vertex* (TLAV), debuted with Google's Pregel system (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, and

Czajkowski 2010). An open-source implementation of this model known as `Apache Giraph` (Ching 2013) was then offered to the public. Other example systems that were created using that model are `GraphLab` (Low, Gonzalez, Kyrola, Bickson, Guestrin, and Hellerstein 2014b), `PowerGraph` (Gonzalez, Low, Gu, Bickson, and Guestrin 2012), `PowerLyra` (Chen, Shi, Chen, and Chen 2015). As the unit of computation is the vertex itself, the user algorithm logic is expressed from the perspective of vertices. The idea is that a vertex-local function will receive information from the vertex's incoming neighbours, perform some computation, potentially update the vertex state and then send messages through the outgoing edges of the vertex. A vertex is the unit of parallelization and a vertex program receives a directed graph and a vertex function as input. It was then extended to the concept of vertex scope, which includes the adjacent edges of the vertex. The order of these steps will vary depending on the type of vertex-centric model used (scatter-gather, gather-apply-scatter).

3.4.2 Model: Superstep Paradigm

In a *superstep* S, a user-supplied function is executed for each vertex v (this can be done in parallel) that has a status of active. When S terminates, all vertices may send messages which can be processed by user-defined functions at step $S+1$.

3.4.3 Model: Scatter-Gather

Scatter-gather shares the same idea behind vertex-centric but separates message sending from message collecting and update application (Stutz, Bernstein, and Cohen 2010). In the scatter phase, vertices execute a user-defined function that sends messages along outgoing edges. In the gather phase, each vertex collects received messages and applies a user-defined function to update vertex state.

3.4.4 Model: Gather-Apply-Scatter

Gather-Sum-Apply-Scatter (GAS) was introduced by `PowerGraph` (Gonzalez, Low, Gu, Bickson, and Guestrin 2012) and was aimed at solving the limitations encountered by vertex-centric or scatter-gather when operating on power-law graphs. The discrepancy between the ratios of high-degree and low-degree vertices leads to imbalanced computational loads during a superstep, with high-degree vertices being more computationally-heavy and becoming stragglers. GAS consists of decomposing the vertex program in several phases, such that computation is more evenly distributed across the cluster. This is achieved by parallelizing the computation over the edges of the graph. In the gather phase, a user-defined function is applied to each of the adjacent edges of each vertex in parallel.

3.4.5 Unit: Edge-Centric (TLEV)

The edge-centric approach, also referred as *think-like-an-edge* (TLEV), was popularized by systems like `X-Stream` (Roy, Mihailovic, and Zwaenepoel 2013) and

`Chaos` (Roy, Bindschaedler, Malicevic, and Zwaenepoel 2015) which specify the computation from the point-of-view of edges. These systems made of use of this paradigm to optimize the usage of secondary storage and network communication with cloud-based machines to process large graphs.

3.4.6 Unit: Sub-Graph-Centric (TLAG)

The previous models are subjected to higher communication overheads due to being fine-grained. It is possible to use sub-graph structure to reduce these overheads (also known as *component-centric* (Heidari, Simmhan, Calheiros, and Buyya 2018)). In this category, the work of (Kalavri, Vlassov, and Haridi 2017) denotes two sub-graph-centric approaches: *partition-centric* and *neighbourhood-centric*. *Partition-centric* instead of focusing on a collection of non-associated vertices, considers sub-graphs of the original graph. Information from any vertex can be freely propagated within its physical partition, as opposed to the vertex-centric approach where a vertex only accesses the information of its most immediate neighbours. This allows for reduction in communication overheads. Ultimately, the partition becomes the unit of parallel execution, with each sub-graph being exposed to a user function. This sub-graph-centric approach is also known as *think-like-a-graph* (Tian, Balmin, Corsten, Tatikonda, and McPherson 2013) (TLAG). *Neighbourhood-centric*, on the other hand, allows for a physical partition to contain more than one sub-graph. Shared state updates exchange information between sub-graphs of the same partition, with replicas and messages for sharing between sub-graphs that aren't in the same partition. For completion, we refer the reader to an analysis of distributed algorithms on sub-graph centric graph platforms (Kakwani and Simmhan 2019).

3.4.7 Model: MEGA

The MEGA model was introduced by `Tux2` (Xiao, Xue, Miao, Li, Chen, Wu, Li, and Zhou 2017), a system designed for graph computations in machine learning. The model is composed of four functions defined by the user: an *exchange* function which is applied to each edge and can change the value of the edge and adjacent vertices; an *apply* function to synchronize the value of vertices with their replicas; a global *sync* function to perform shared computations and update values shared among partitions; a *mini-batch* function to indicate the execution sequence of other functions in each round.

There are graph processing systems that offer more than one type of model. To achieve parallelism and harness multiple machines in clusters, it is necessary to define how to break down the graph - we provide a high-level overview of methods employed in most well-known graph processing solutions

3.4.8 Units and Models: Notes

This section presented different models for graph processing and the granular computational units that enable them. *Think-like-a-vertex* (TLAV), known as the vertex-centric paradigm, emerged as far as we know as part of the first proposed model for distributed graph processing with `Pregel`. This paradigm was then extended with variations on the order of data processing and state updating with works such as `PowerGraph`.

The other type of granular element of the graph, the edge, has also been explored as a computational unit, manifesting as the edge-centric *think-like-an-edge* (TLEV) approach with `X-Stream` and `Chaos`. Reducing the level of granularity, one may find coarser units such as the variants of the sub-graph-centric model like *partition-centric* and *neighbourhood-centric*.

An awareness of the multiple models, even if not of the systems that presented them, is relevant and helpful for developers. It enables the implementation of graph processing logic in a more performance-oriented way. If using existing systems, knowing these models and sub-models is important as more than one may be offered by a given framework. If building a completely new system, knowing existing approaches will contribute to more informed design decisions.

3.5 Dimension: Partitioning

Graph partitioning is an important problem in graph processing, and this importance manifests in two formats. The first, is out of a user's domain application with the goal of splitting the graph in parts which provide a relevant view of the data. The second, is when partitioning may be considered as a *hyper-algorithm*, that is, it is employed to divide the parts of the graph across a computational infrastructure, typically within the distributed systems' coordination layer, or across processing units or cores within machines. Machine loads in distributed graph processing systems depend on the way computational units are distributed across workers. The communication between them then depends on the number of units that are replicated. We observe that partitioning has a cyclical nature to itself in the scope of distributed processing: one may wish to execute graph partitioning over a distributed system as part of a domain-specific problem; however, before that graph algorithm can execute, the graph data also incurs partitioning followed by distribution in the underlying (distributed) computational infrastructure. While the study of graph partitioning is not recent, it gained additional depth in the last decade as the number of factors guiding optimization of partitioning increased with the complexity of graph processing systems. We explore partitioning as a relevant dimension to classify these systems as they must approach it in order to enable parallel computation over graphs. The way it is approached becomes a distinctive feature between the systems.

Graph partitioning aims to divide the vertices of the graph into mutually-exclusive groups and to minimize the edges between partitions. This is effectively a grouping of the vertex set of the graph, which can represent a minimization of communication between partitions, with each partition for example assigned to a specific worker in a distributed system. Partitioning is a task that produces groups of vertices, but grouping vertices is not only achieved with partitioning. We note that other terms exist in the literature such as clustering and community detection. They are not interchangeable, for if a clustering algorithm breaks down the graph into three clusters, it does not necessarily hold true that each cluster represents its own community. As an example, executing a clustering algorithm over a social network graph will result in a number of clusters. If each cluster represents for example a different continent, that does not necessarily mean each cluster represents one single community. Community detection algorithms, on the other hand, consider properties such as the density and interconnections within communities. While clustering and community detection aim to identify similarities between vertices, their underlying assumptions of the graph are not equal, even though proposals have been made to map between these two tasks (Guidotti and Coscia 2017). Graph clustering shares similarities with graph partitioning in the sense that both produce groups of vertices. However, the objective functions they use are defined differently and subject to different constraints. Graph partitioning, on which we focus, for example, requires that the number of groups (partitions) is known beforehand and is typically subject to more constraints.

An earlier work on balanced graph partitioning (Andreev and Racke 2006) defines the problem as (k, v)-balanced partitioning: to divide the vertices of the graph into k components of almost equal size, with each of size less than $c \cdot \frac{n}{k}$ for a given constant $c > 1$. It is a balanced k-way partitioning problem which has been studied in the literature (Buluç, Meyerhenke, Safro, Sanders, and Schulz 2016). To consider the partitioning problem as a challenge to enabling distributed processing, it is necessary to ask if the goal is to distribute the vertices (edge cut model - EC) or the edges (vertex cut model - VC) of the graph across machines in order perform it. We provide detail into these problem formulations with an example of vertex-cut and edge-cut in Figure 3.2. Furthermore, different combinations between computational unit and cut model are possible: vertex-cut can be used to process in a vertex-centric (Mofrad, Melhem, and Hammoud 2018) or edge-centric (Gonzalez, Low, Gu, Bickson, and Guestrin 2012) way, and the same is possible using edge-cut used to partition a graph where computation is vertex-specific (Bao and Suzumura 2013; Martella, Logothetis, Loukas, and Siganos 2017) or edge-specific.

3.5.1 Edge-Cut (EC)

Balanced k-way partitioning may be defined for edge-cut partitioning, which is associated to vertex-centric (TLAV) systems, the most common computational model in graph processing systems (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, and Czajkowski 2010; Gonzalez, Xin, Dave, Crankshaw, Franklin, and Stoica 2014; Sakr, Orakzai, Abdelaziz, and Khayyat 2016). We reproduce the definition of (Soudani, Fatemi, and Nematbakhsh 2019, Section 2) for this case, where for a given graph $G = (V, E)$, we wish to find a set of partitions $P = \{P_1, P_2, \ldots, P_k\}$. These partitions must be pairwise disjoint and their union is equal to V while following these conditions (Soudani, Fatemi, and Nematbakhsh 2019):

$$\min_P |\{e | e = (v_i, v_j) \in E, v_i \in P_x, v_j \in P_y, x \neq y\}| \tag{3.1}$$

$$s.t.\frac{\max_i |P_i|}{\frac{1}{k}\sum_{i=1}^{k}|P_i|} \leq \epsilon \tag{3.2}$$

Depending on the application objective for which this partitioning type will be performed, Equation 3.2 should be adapted. For example, in the case of machines having different characteristics, it should be considered that the load of any machine will be less than the maximum computing power. Or if the graph structure is stored in secondary memory, the interest is on having balanced size partitions with high speed sequential storage access and decreasing the number of cut edges is no longer a focus.

3.5.2 Vertex-Cut (VC)

In the vertex-cut model, the goal is to distribute edges across partitions. They are placed in different partitions, with vertices being copied in partitions which have their adjacent edges. Care must be taken to balance the number of edges per partition (its measure of size) and to minimize the number of vertex copies. This objective may be formulated as such (Soudani, Fatemi, and Nematbakhsh 2019):

$$\min_P \frac{1}{|V|}\sum_{v\in V}|P(v)| \tag{3.3}$$

$$s.t. \max_{p_i} |\{e \in E | P(e) = p_i\}| \leq \epsilon\frac{|E|}{k} \tag{3.4}$$

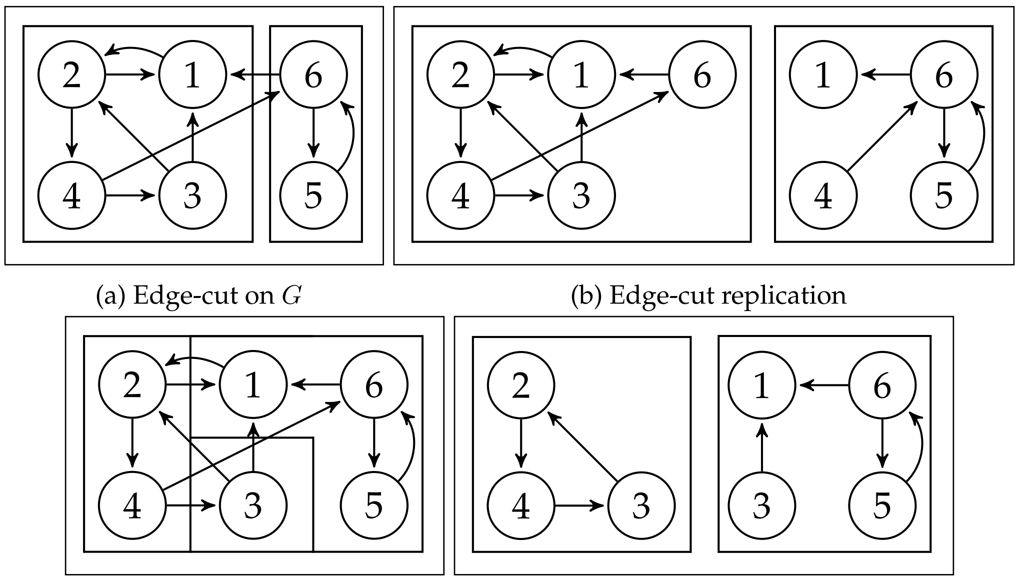
Vertex-cut achieves better performance than edge-cut for natural graphs such as those representing web structure and social networks (Soudani, Fatemi, and Nematbakhsh 2019).

3.5.3 Hybrid-Cut (HC)

Hybrid strategies can be employed to perform the cuts. They can for example be guided with heuristics such as vertex degree in order to decide what to do with them. The `PowerLyra` (Chen, Shi, Chen, and Chen 2015) system for example allocates the incoming edges of vertices with low degree in a worker. It uses edge-cut for vertices of low-degree and vertex-cut for high-degree vertices.

3.5.4 Stream-based partitioning

In these methods of partitioning, vertices or edges in the graph are analysed in succession in a stream. Placement decisions are made *online*, that is, when the vertices or edges appear in the stream, and the decisions are based on the location of previous elements. This is done under the assumption that there will be no information on the edges or vertices that will arrive in the flow of the stream. This type of method can rely on edge-cut partitioning (e.g. `Random heuristic` and the `Linear Deterministic Greedy` (Stanton and Kliot 2012), `Gemini` which uses chunk-based assuming adjacency list model (Zhu, Chen, Zheng, and Ma 2016), `Fennel` (Tsourakakis, Gkantsidis,



(a) Edge-cut on G (b) Edge-cut replication

(c) Vertex-cut on G (d) Vertex-cut replication

Figure 3.2: Depiction of vertex-cut and edge-cut over the sample graph G.

Radunovic, and Vojnovic 2014)), vertex-cut partitioning (e.g. `Grid` heuristic (Jain, Liao, and Willke 2013), `PowerGraph` greedy heuristic (Gonzalez, Low, Gu, Bickson, and Guestrin 2012), `Graphbuilder` (Jain, Liao, and Willke 2013) placing the edge in the smallest partition, `HDRF` (Petroni, Querzoni, Daudjee, Kamali, and Iacoboni 2015) method which takes into consideration vertex degrees) and there are aspects of these methods that will have different approaches regarding how this is achieved with parallel and distributed execution. Stream-based partitioning is also used as a good choice for loading the graph as it does not have to be fully loaded in memory for partitioning.

3.5.5 Distributed Partitioning

Many distributed partitioning algorithms are based on label propagation algorithms (Zhu and Ghahramani 2002; Gregory 2010; Liu and Murata 2010; Boldi, Rosa, Santini, and Vigna 2011), with variations such as how the specific labelling of a vertex should be influenced by its neighbours, if it should also be influenced by the label's global representation in the graph and also constraints on the minimum and maximum sizes required for partitions. For example, `Revolver`, which performs vertex-centric graph partitioning with reinforcement learning, assigns an agent to each vertex, with agents assigning vertices to partitions based on their probability distribution (these are then refined based on feedbacks (Mofrad, Melhem, and Hammoud 2018)). The authors of (Soudani, Fatemi, and Nematbakhsh 2019) note that other approaches consider the partitioning problem as a multi-objective and multi-constraint problem, achieving better results compared to one-phase methods (Slota, Madduri, and Rajamanickam 2014).

Distributed partitioning systems are good for when partitioning is performed once and then calculations are repeatedly performed.

3.5.6 Dynamic Graph Partitioning

When the graph is no longer static, vertex and edges may be added or removed as time passes - this is especially true in social networks. This implies that for graphs from which we need to perform computations as they evolve, the original partitioning may become inefficient. With predictable algorithm runtime characteristics, it becomes feasible to keep close the vertices which will be used together in the same supersteps, using for example graph traversal algorithms. But when this is not the case, systems can be designed for example to monitor the load and communication of the machines and migrate vertices as appropriate, with different techniques having been proposed for that purpose (among others, `xDGP` (Vaquero, Cuadrado, Logothetis, and Martella 2013) to repartition massive graphs to adapt to structural changes, `GPS` (Salihoglu and Widom 2013b) which reassigns vertices based on communication patterns, `X-Pregel` (Bao and Suzumura 2013) with reduction of message exchanges and dynamic repartitioning). Dynamic partitioning methods have the advantage of outputting very good load balancing and communication cost reductions due to considering heterogeneous hardware and runtime characteristics.

3.5.7 Partitioning: Summary

Employed graph partitioning strategies vary, with different systems offering different solutions. Among performance-impacting factors (Soudani, Fatemi, and Nematbakhsh 2019), we have the number of active vertices and edges influencing machine load. At the same time, communication will be more expensive depending on how replication of edges and vertices is performed. Partitioning must balance communication and machine loads. The partitioning challenge in vertex-centric systems is relevant due to how widespread this model is. The authors of (Soudani, Fatemi, and Nematbakhsh 2019) note three major approaches for big graph partitioning: *a)* partitioning the graph serially in a single pass and permanently assigning the partition on the first time an edge or vertex is assigned (stream-based); *b)*; methods that partition in a distributed way; *c)* dynamic methods that adapt the partitions based on monitoring the load and communication of machines during algorithm execution. For an in-depth analysis of partitioning methods, vertex cut models and their relation to the dynamic nature of data, we invite the reader to read (Soudani, Fatemi, and Nematbakhsh 2019).

Being able to decompose the graph is a cornerstone for efficient and distributed computation of graphs. An equally-important aspect that determines how we must approach the computation is the possible dynamism of the graph. A static graph over which we want to perform analytics is a scenario different from maintaining a large graph available for separate queries and susceptible to updates.

3.6 Dimension: Dynamism

We include and consider *dynamism* a relevant dimension of graph processing due to there existing different meanings associated to it in the literature and for which different systems can be attributed. While one may consider static graphs to be completely unrelated to dynamism, there is in fact a relation to it due to what is known as *stream processing*. For example, a graph processing system may ingest an unbounded stream of edges and update statistics over the stream (e.g., keeping a triangle count updated (Ahmed, Duffield, Willke, and Rossi 2017)), but stream processing may also take place in static graph processing. This is the case with approaches that process a static graph but process its elements from a stream perspective (e.g., `Chaos` (Roy, Bindschaedler, Malicevic, and Zwaenepoel 2015) and `X-Stream` (Roy, Mihailovic, and Zwaenepoel 2013) with their edge-centric approach). Considering if a system targets graphs that change or are immutable (static) is an obvious way to separate graph processing systems when classifying them. However, this dimension is actually a spectrum between the immutable (e.g. stream-based perspectives to process static graphs) and the changing - for example, is the whole graph structure kept in memory (or secondary storage) in a single machine (or across cluster computing nodes), or is it discarded by proxy of some criteria (and thus one simply updates mathematical properties of the graph using only recent information from the stream)? For this spectrum, the authors of (Besta, Fischer, Kalavri, Kapralov, and Hoefler 2019) cover definitions found in the literature:

- Temporal graphs. These are, in essence, static graphs which have annotated temporal information which allows for recreating the domain represented by the graph at any given point in time. It is not structurally-changed while doing so; it means that for a given time range or event, only the elements with valid timestamps under required constraints are considered for computation. The work of (Kostakos 2009) introduces the temporal graph as a representation encoding temporal data into the graph while retaining the temporal information of the original data. They present metrics that can be used to study temporal graphs and use the representation to explore dynamic temporal properties of data using graph algorithms without requiring data-driven simulations. `ImmortalGraph` (Miao, Han, Li, Wu, Yang, Zhou, Prabhakaran, Chen, and Chen 2015) is a storage and execution engine designed with temporal graphs in mind, having achieved greater efficiency than database solutions for graph queries. `ImmortalGraph` schedules common bulk operations in a way to maximize the benefit of in-memory data locality. It explores the relation between locality, parallelism and incremental computation while enabling mining tasks on temporal graphs. For more information and reach on the topic of temporal graphs, we direct the reader to (Michail 2016).

- Streaming graph algorithms (Feigenbaum, Kannan, McGregor, Suri, and Zhang 2005). With these, the common scenario starts from an empty graph without edges (and a fixed set of vertices). For each algorithm step, a new edge is inserted into the graph or an edge is removed. It is desired that these algorithms are developed to minimize parameters such as graph data structure storage, the time to process an edge or the time to recover the final solution. There exist several systems which process streaming graph computations - we note also for the reader a recent framework for comparing the systems aimed at this type of dynamism (Erb, Meißner, Kargl, Steer, Cuadrado, Margan, and Pietzuch 2018). The `STINGER` data structure has been used for streaming graphs as well (Ediger, McColl, Riedy, and Bader 2012).

- Sketching and dynamic graph streams. Sketching techniques (Ahn, Guha, and McGregor 2012) may be applied to the edge incidence matrix of the input graph to approximate cut structure and connectivity. The idea is to consume a stream of events in order to generate a probabilistic data structure representing properties of the graph.

- Multi-pass streaming graph algorithms. In this type of algorithm, all updates are streamed more than once in order to approximate the computation quality of the solution. Additional complexity can emerge on how the streaming model behaves - it can for example allow for the stream to be manipulated across passes (Aggarwal, Datar, Rajagopalan, and Ruhl 2004) or to stream sorting passes (Demetrescu, Finocchi, and Ribichini 2009).

- Dynamic graph algorithms. For these types, the focus is cast on being able to approximate combinatorial properties of the graph (Besta, Fischer, Kalavri, Kapralov, and Hoefler 2019) (e.g., connectivity, shortest path distance, cuts, spectral properties) while processing insertions and deletions. The objective with this type of algorithm is to quickly integrate graph updates. `Ringo` (Perez, Sosič, Banerjee, Puttagunta, Raison, Shah, and Leskovec 2015) is a single-machine analytics system that supports dynamic graphs.

While partitioning and dynamism are relevant aspects, the scope of graph processing solutions in both industry and academia was shaped by the type of executed workloads.

3.7 Dimension: Workload

The type of workload performed by a graph processing system also plays an important role in classifying them. The type of task performed by graph databases is different from the systems that run global algorithms over them. The concept of analysing a graph takes on different contexts depending on user needs. We note that when a graph is to be *processed*, the scope of its data analysis usually falls in these two categories:

(a) To retrieve instances of domain-specific relations in the graph (e.g. pattern matching, multi-hop queries). These are usually found in graph databases, with an emphasis on optimization of data query and storage for online transaction processing scenarios. This is often accompanied with the use of graph query languages

(GQLs) to execute queries that return a view on the graph and also potentially producing effects on it.

(b) To execute an algorithm over the whole graph (e.g. PageRank, connected components, detecting communities, finding shortest paths). The solutions for this task, performance-wise, aim to achieve high-performance computational throughput, whether using distributed systems or a single-machine setup. It is a focus leaning on the data analytics aspect.

The former (a) is a common scenario in graph databases such as `Neo4j` (Webber 2012) and `JanusGraph` (JanusGraph Authors 2017), among others. These databases offer graph query languages (usually even allowing interchangeability between languages) such as `Cypher` or `Gremlin` (Holzschuher and Peinl 2013). They are built to store the graph, some with sharding (horizontal scaling) to distribute the graph across the storage/computational infrastructure (some outsource the storage medium to database technologies such as `HBase` (George 2011) or `Cassandra` (Lakshman and Malik 2010)), others in a centralized server (but allowing cluster computing nodes for the specific purpose of redundancy). They employ schemes to store the graph efficiently while offering transaction mechanisms to operate over the graph and to perform queries.

The latter type (b) is seen in big (graph) data processing systems like `Spark (GraphX library)` (Xin, Gonzalez, Franklin, and Stoica 2013) and `Flink (Gelly library)` (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015). The mentioned names are all distributed processing frameworks that can take advantage of multi-core machines and clusters. These systems and their libraries allow for expressive computation over graphs in few lines of code. Many of the systems come with their sets of graph algorithms, allowing for the composition of workflows while abstracting away many details from the programmer (regarding distributed computation orchestration and the internal implementation of the graph algorithms).

It is important to consider two definitions regarding the nature of computational tasks: online analytical processing (`OLAP`) and online transaction processing (`OLTP`). `OLAP` is an approach to enable answering multi-dimensional analytical queries quickly. Among its instances we may find tasks such as business reporting for sales, management reporting, business process management (Benisis 2010), financial reporting and others. `OLTP`, on the other hand, refers to systems that enable and manage transaction-oriented applications, with *transaction* meaning in a computational context the atomic state changes that take place in database systems. `OLTP` examples include retails sales and financial transaction systems, and applications of this type tend to be high-throughput and update/insertion-intensive in order to provide availability, speed, recoverability and concurrency (Corporation 1999).

The earlier type $a)$ of graph processing task may be associated to `OLTP` systems as the goal is to store representations of graphs by quickly ingesting new information, efficiently representing it regarding space consumption and access speed, and being able to execute updates under ACID properties (or a subset of those). For this type of task $a)$, one may find numerous graph databases to match the description, such as those for designed for semantic (RDF) representations (Inc. 1984; Stutz, Verman, Fischer, and Bernstein 2013; Tesoriero 2013; Ontotext 2020), or for property graph models (Martinez-Bazan, Gomez-Villamor, and Escale-Claveras 2011; Miller 2013; Rudolf, Paradies, Bornhövd, and Lehner 2013; Aurelius 2015; DataStax, Inc. 2016; Dubey, Hill, Escriva, and Sirer 2016; JanusGraph Authors 2017; Kankanamge, Sahu, Mhedbhi, Chen, and Salihoglu 2017; Microsoft 2017; Paz 2018; Hwang 2018; Cailliau, Davis, Gadepally, Kepner, Lipman, Lovitz, and Ouaknine 2019; Thomas Schmidts 2020), both (Bebee, Choi, Gupta, Gutmans, Khandelwal, Kiran, Mallidi, McGaughy, Personick, Rajan, et al. 2018) and also other specific purposes (Google 2017). The latter type of task $b)$ may be associated to OLAP, where there is a focus on extracting value from the data and the nature of the task is typically read-only. We include graph processing systems (not databases) in this group of OLAP-type tasks, even the systems which support mutability in graphs due to supporting dynamism in any form.

There is a considerable overlap between OLTP-type tasks and graph databases, and there is also an overlap between OLAP-type tasks and graph processing systems. While the distinction between OLAP and OLTP task types is not a dimension that perfectly divides systems in the graph processing landscape, we note that such a distinction holds value in guiding future taxonomies of the graph processing system landscape, and for that reason we include it as a dimension.

The way these three dimensions are accounted for influence the design of graph processing systems. Many different architectures exist, for which we share an exhaustive list of specific solutions, from single-machine systems to parallel processing in clusters and storage in tailor-made graph databases.

In Table 3.1 we summarize distinguishing features and licenses for the graph processing systems detailed in this section. The last reference in front of every system name is its open-source code repository, when available. The second group from the top (PBGL, CombBLAS and HavoqGT) contains systems which use multiple machines for computation but not in the typical cluster scenario. Instead, they are characterized by using specific machines for high-performance computing.

System	*Multi-core*	*GPU*	*Cluster*	*Languages*	*License*	*Notes*
GraphLab (2010, 2014a)	●		●	C++	AL 2.0	*N/A*
GRACE (2013)	●		●	C++	*Unavailable*	*N/A*
Ligra (2013, 2020)	●			C++	MIT	*N/A*
Ringo (2015, 2016)	●			C++, Python	BSD	*N/A*
Polymer (2015, 2018)	●			C++	AL 2.0	*N/A*
GraphMat (2015, 2017)	●			C++	*Custom*	*N/A*
Mosaic (2017b, 2017a)	●			C++	MIT	*Fast storage*
PBGL (2005, 2018)	●			C++	*Custom*	*Hardware*
CombBLAS (2011b, 2011a)	●			C++	*Custom*	*Hardware*
HavoqGT (2013, 2019)	●			C++	GNU LGPL 2.1	*Hardware*
Apache Giraph (2013, 2019a)	●		●	Java	AL 2.0	*N/A*
Naiad (2013, 2018)	●			C#	AL 2.0	*N/A*
Apache Flink (2015, 2020a)	●		●	Java, Python, Scala	AL 2.0	*N/A*
Apache Spark (2010, 2020b)	●		●	Java, Python, Scala	AL 2.0	*N/A*
GraphTau (2016)	●		●	Java, Scala	*Unavailable*	*N/A*
Tink (2018, 2019)	●		●	Java, Scala	AL 2.0	*N/A*
X-Stream (2013, 2015)	●			C++	AL 2.0	*N/A*
Chaos (2015, 2016)	●		●	C++	AL 2.0	*N/A*
PowerLyra (2015, 2018)	●		●	C++	AL 2.0	*N/A*
Kineograph (2012, 2020)	●		●	*Unknown*	*Unavailable*	*N/A*
Tornado (2016)	●		●	*Unknown*	*Unavailable*	*N/A*
KickStarter (2017)	●		●	C++	MIT	*N/A*
Pixie (2018)	●		●	*Unknown*	*Unavailable*	*N/A*
FlowGraph (2019)	●		●	*Unknown*	*Unavailable*	*N/A*
GPS (2013b, 2013a)	●		●	Java	BSD	*N/A*
GoFFish (2014, 2017)	●		●	Java	*Unknown*	*Copyright*
FBSGraph (2017)	●		●	Unknown	*Unavailable*	*N/A*
GrapH (2018, 2016)	●		●	Java	*Unknown*	*Copyright*
Julienne (2017)	●			C++	*Unavailable*	*N/A*
GraphD (2017)	●		●	Unknown	*Unavailable*	*N/A*
TurboGraph++ (2018)	●		●	Unknown	*Unavailable*	*N/A*
GraphIn (2016)	●			C++	*Unavailable*	*N/A*
MapGraph (2014, 2016)		●		C++	AL 2.0	*Discontinued*
CuSha (2014, 2015)		●		C++	MIT	*N/A*
Gunrock (2016, 2017, 2020)		●		C	AL 2.0	*N/A*
Lux (2017, 2018)	●	●	●	C++	AL 2.0	*N/A*
Frog (2017, 2018)		●		C	GPL 2.0	*N/A*
Gluon (2018, 2020b)	●	●		C++	3C BSD	*N/A*
GraphCage (2019)		●		Unknown	*Unavailable*	*N/A*
FlashGraph (2015, 2014)				C++	AL 2.0	*SSDs*
GraphSSD (2019)				Unknown	*Unavailable*	*SSDs*

Table 3.1: Summary of graph system distinctive features. Circle ● on the *Multi-core*, *GPU* and *Cluster* columns indicate that option is supported. *Languages* lists the programming languages the systems were written in. *License* lists the licenses of the open-source project or of the free edition of a commercial product: AL 2.0 is Apache License 2.0, CC 1.0 is Commons Clause 1.0, (GPL) v3 is GNU General Public License (GPL) v3. *Notes* covers additional information, with *Copyright* meaning that it may be illegal to reuse the source code.

3.8 Single-Machine and Shared-Memory Parallel Approaches

- GraphLab (Low, Gonzalez, Kyrola, Bickson, Guestrin, and Hellerstein 2010) was published as a framework (implemented in C++) for parallel machine learning and later extended to support distributed settings while retaining strong data consistency guarantees (Low, Gonzalez, Kyrola, Bickson, Guestrin, and Hellerstein 2014b). The authors evaluate it on Amazon EC2, outperforming equivalent MapReduce implementations by over 20X and match the performance of specifically-crafted MPI implementations. GraphLab requires the whole graph and program state to reside in RAM. It uses a chromatic engine so that no adjacent vertices have the same colour and to enable the efficient use of network bandwidth and processor time. The authors evaluate it for applications such as Netflix movie recommendation, video co-segmentation and named entity recognition. It is open-source (Low, Gonzalez, Kyrola, Bickson, Guestrin, and Hellerstein 2014a) under the Apache License 2.0.

- GRACE (Wang, Xie, Demers, and Gehrke 2013) is a synchronous iterative graph programming model, with separation of application logic and execution policies. Its design includes the implementation (C++) of a parallel execution engine for both synchronous and user-specified asynchronous execution policies. GRACE stores directed graphs, and in its model and the computation is expressed and performed in a way similar to Pregel. It provides additional flexibility, by allowing the user to relax synchronization of computation. This is achieved with user-defined functions which allow updating the scheduling priority of vertices that receive messages (the vertex order in which computation will take place within an iteration). GRACE's design targets both shared-memory and distributed system scenarios, but the initial prototype focuses on shared-memory. We did not find the source code available.

- Ligra (Shun and Blelloch 2013) is a C++ lightweight graph processing framework targeting shared-memory parallel/multi-core machines, easing the writing of graph traversal algorithms. This framework offers two map primitives to operate a given logic on vertices (VertexMap) and edges (EdgeMap) and supports two data types: the traditional graph $G = (V, E)$ as we described in an earlier section, and another one to represent subsets of vertices. The interface is designed to enable the processing of edges in different orders depending on the situation (as opposed to Pregel or Giraph). The code of Ligra represents in-edges and out-edges as arrays, with in-edges for all vertices being partitioned by their target vertex and storing the source vertices, and the out-edges are in an array partitioned by source vertices and storing the target vertices. While the authors claim to have achieved good performance results, they mention Ligra does not support algorithms based on modifying the input graph. It is available (Shun and Blelloch 2020) under the `MIT License`.

- `Ringo` (Perez, Sosič, Banerjee, Puttagunta, Raison, Shah, and Leskovec 2015) is an approach for multi-core single-machine big-memory setups. It is a high-performance interactive analytics system using a `Python` front-end on a scalable parallel `C++` back-end, representing the graph as a hash table of vertices. It supports fast execution times with exploratory and interactive use, offering graph algorithms in a high-level language and rich support for transformations of input data into graphs. `Ringo` is open-source and available (Perez, Sosič, Banerjee, Puttagunta, Raison, Shah, and Leskovec 2016) under the `BSD License`.

- `Polymer` (Zhang, Chen, and Chen 2015) is a NUMA-aware graph analytics system on multi-core machines that is open-source (Zhang, Chen, and Chen 2018) under the `Apache License 2.0` and implemented in `C++`. It innovated by differentially allocating and placing topology data, application-defined data and mutable run-time graph system states according to access patterns to minimize remote accesses. `Polymer` also deals with random accesses by converting the random ones into sequential remote accesses using lightweight vertex replication across the computational NUMA nodes. It was built with a hierarchical barrier for increased parallelism and locality. The design also includes edge-oriented balanced partitioning for skewed graphs and adaptive data structures in function of the fraction of active vertices. It was compared to `Ligra`, `X-Stream` and `Galois` on an 80-core Intel machine (no hyper-threading) and on a 64-core AMD machine. For different algorithms across several data sets, `Polymer` consistently almost always achieved the lowest execution time.

- `GraphMat` (Sundaram, Satish, Patwary, Dulloor, Vadlamudi, Das, and Dubey 2015) is a framework written in `C++` aimed at bridging the user-friendly graph analytics and native hand-optimized code. It presents itself as a vertex-centric framework without sacrificing performance, as it takes vertex programs and maps them to exclusively use sparse matrix high-performance back-end operations. `GraphMat` takes graph algorithms expressed as vertex programs and performs generalized sparse matrix vector multiplication on them. It achieved greater performance than other frameworks such as 5-7X faster than `GraphLab`, `Galois` and `ComBLAS`. It also achieved multi-core scalability, being over 10X faster than single-threaded implementation on a 24-core machine. It is open-source and available (Sundaram, Satish, Patwary, Dulloor, Vadlamudi, Das, and Dubey 2017) under specific conditions by Intel.

- `Mosaic` (Maass, Min, Kashyap, Kang, Kumar, and Kim 2017b) is a system for single heterogeneous machines with fast storage media (e.g., NVMe and SSDs) and massively-parallel co-processors (e.g., Xeon Phi) developed to enable the processing of trillion-edge graphs. The system is designed explicitly separating graph processing engine components into scale-up and scale-out goals. It is written in

C++ uses a compact representation of the graph using Hilbert-ordered tiles for locality, load balancing and compression and uses a hybrid computation model that uses both vertex-centric operations (on host processors) and edge-centric operations (on co-processors). Mosaic is open-source (Maass, Min, Kashyap, Kang, Kumar, and Kim 2017a) under the MIT License.

3.9 High-Performance Computing

These systems are hallmarks of high-performance computing solutions applied to graph processing. Their merits encompass algebraic decomposition of the major graph operations, implementing them and translating them across different homogeneous layers of parallelism (across cores, across CPUs). Here we mention what are, to the best of our knowledge, the most relevant works:

- Parallel Boost Graph Library (PBGL) (Gregor and Lumsdaine 2005) . It is an extension (C++) of Boost's graph library. It is a distributed graph computation library, also offering abstractions over the communication medium (e.g. MPI). The graph is represented as an adjacency list that is distributed across multiple processors. In PBGL, vertices are divided among the processors, and each vertex's outgoing edges are stored on the processor storing that vertex. PBGL was evaluated on a system composed of 128 compute nodes connected via Infiniband. It is available (Gregor and Lumsdaine 2018) under a custom Boost Software License 1.0.

- CombBLAS (Buluç and Gilbert 2011b). A parallel graph distributed-memory library in C++ offering linear algebra primitives based on sparse arrays for graph analytics. This system considers the adjacency matrix of the graph as a sparse matrix data structure. CombBLAS is edge-based in the sense that each element of the matrix represents an edge and the computation is defined over it. It decouples the parallel logic from the sequential parts of the computation and makes use of MPI. However, its MPI implementation does not take advantage of flexible shared-memory operations. Its authors targeted hierarchical parallelism of supercomputers for future work. It is available (Buluç and Gilbert 2011a) under a custom license.

- HavoqGT (Pearce, Gokhale, and Amato 2013) is a C++ system with techniques for processing scale-free graphs using distributed memory. To handle the scale-free properties of the graph, it uses edge list partitioning to deal with high-degree vertices (hubs) and dummy vertices to represent them to reduce communication hot spots. HavoqGT allows algorithm designers to define vertex-centric procedures in a distributed asynchronous visitor queue. This queue is part of an asynchronous visitor pattern designed to tackle load imbalance and memory latencies. HavoqGT targets supercomputers and clusters with local NVRAM. It is available (Pearce, Gokhale, and Amato 2019) under the GNU Lesser General Public License 2.1.

3.10 Distributed Graph Processing Systems

While the previous systems we detailed performed analytics and enabled the execution of graph algorithms, they did so with a focus on specific hardware and distributed memory. We list here some of the most relevant state-of-the-art systems used for graph processing in the scope of analytics (OLAP). Their use of different architectures (from using local commodity clusters to cloud-based execution) and greater flexibility of deployment scenarios differentiate them from those of the previous section.

The following systems are relevant names in the literature, with Giraph being the first open-source implementation of the Pregel approach to graph processing, and Spark and Flink being open-source general distributed processing systems with graph processing APIs:

- Apache Giraph (Ching 2013) is an open-source Java implementation of Pregel (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, and Czajkowski 2010), tailor-made for graph algorithms, supporting the GAS model and licensed (Foundation 2019a) under the Apache License 2.0. It was created as an efficient and scalable fault-tolerant implementation on clusters with thousands of commodity hardware, hiding implementation details underneath abstractions. Work has been done to extend Giraph from the *think-like-a-vertex* (TLAV) model to *think-like-a-graph* (TLAG) (Tian, Balmin, Corsten, Tatikonda, and McPherson 2013). It uses Hadoop's MapReduce implementation to process graphs. Giraph (Foundation 2019a) allows for master computation, sharded aggregators (relevant when computing a final result comprised of intermediate data from computational nodes), has edge-oriented input, and also uses out-of-core computation – limited partitions in memory. Partitions are stored in local disks, and for cluster computing settings, the out-of-core partitions are spread out across all disks. Giraph attempts to keep vertices and edges in memory and uses only the network for the transfer of messages. Improving Giraph's performance by optimizing its messaging overhead has also been studied (Liu, Zhou, Gao, and Fan 2016). It is interesting to note that single-machine large-memory systems such as Ringo highlight the message overhead as one of the major reasons to avoid a distributed processing scheme.

- Naiad is an open-source (Research 2018) (Apache License 2.0) dataflow processing system (Murray, McSherry, Isaacs, Isard, Barham, and Abadi 2013) offering different levels of complexity and abstractions to programmers. It allows programmers to implement graph algorithms such as weakly connected components, approximate shortest paths and others while achieving better performance than other systems. Naiad is implemented in C# and allows programmers to use common high-level APIs to express algorithm logic and also offers a low-level API for performance. Its concepts are important and other systems could benefit from offering tiered programming abstraction levels as in Naiad. Its low-level primitives allow for the combination of dataflow primitives (similar to those VEILGRAPH uses from Flink) with finer-grained control on iterative computations. An extension to Flink's architecture to offer this detailed control would enrich the abilities that our framework is able to offer to users.

- Apache Flink (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015), formerly known as Stratosphere (Alexandrov, Bergmann, Ewen, Freytag, Hueske, Heise, Kao, Leich, Leser, Markl, Naumann, Peters, Rheinländer, Sax, Schelter, Höger, Tzoumas, and Warneke 2014), it is a framework which supports built-in iterations (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015) (and delta iterations) to efficiently aid in graph processing and machine learning algorithms. It is licensed (Foundation 2020a) under the Apache License 2.0 and has a graph processing API called Gelly, which comes packaged with algorithms such as PageRank, single-source shortest paths and community detection, among others. Flink supports Java, Python and Scala. It explicitly supports three vertex-based programming models: *think-like-a-vertex* (TLAV) described as the most generic model, supporting arbitrary computation and messaging for each vertex; Scatter-Gather, which separates the logic of message production from the logic of updating vertex values, which may typically make these programs have lower memory requirements (concurrent access to the inbox and outbox of a vertex is not required) while at the same time potentially leading to non-intuitive computation patterns; Gather-Sum-Apply-Scatter (GAS), which is similar to Scatter-Gather but the Gather phase parallelizes the computation over the edges, the messaging phase distributes the computation over the vertices and vertices work exclusively on neighbourhood, where in the previous two models a vertex can send a message to any vertex provided it knows its identification. It supports all Hadoop file systems as well as Amazon S3 and Google Cloud storage, among others. Delta iterations are also possible with Flink, which is quite relevant as they take advantage of computational dependencies to improve performance. It also has flexible windowing mechanisms to operate on incoming data (the windowing mechanism can also be based on user-specific logic). Researchers have also looked into extending its DataStream constructs and its streaming engine to deal with applications where the incoming flow of data is graph-based (Kalavri, Carbone, Bali, and Abbas 2019).

- Apache Spark (Zaharia, Chowdhury, Franklin, Shenker, and Stoica 2010) and its GraphX (Xin, Gonzalez, Franklin, and Stoica 2013) graph processing library, licensed (Foundation 2020b) under the Apache License 2.0. It is a graph processing framework built on top of Spark (a framework supporting Java, Python and Scala), enabling low-cost fault-tolerance. The authors target graph processing by expressing graph-specific optimizations as distributed join optimizations and graph views' maintenance. In GraphX, the property graph is reduced to a pair of collections. This way, the authors are able to compose graphs with other collections in a distributed dataflow framework. Operations such as adding additional vertex properties are then naturally expressed as joins against the collection of vertex properties. Graph computations and comparisons are thus an exercise in analysing and joining collections.

- `GraphTau` (Iyer, Li, Das, and Stoica 2016) is a time-evolving graph processing framework on top of `Spark` (`Java`, `Scala`). It represents computations on time evolving graphs as a stream of consistent and resilient graph snapshots and a small set of operators that manipulate such streams. `GraphTau` builds fault-tolerant graph snapshots as each small batch of new data arrives. It is also able to periodically load data from graph databases and reuses many operators from `GraphX` and `Spark Streaming`. For algorithms (based on label propagation) that are not resilient to graph changes, `GraphTau` introduced an online rectification model, where errors caused by underlying graph modifications are corrected in online fashion with minimal state. Its API frees programmers from having to implement graph snapshot generation, windowing operators and differential computation mechanisms. We did not find its source code available.

- `Tink` (Lightenberg, Pei, Fletcher, and Pechenizkiy 2018) is a library for distributed temporal graph analytics. It is built on `Flink` (`Java`, `Scala`) and focuses on *interval* graphs, where each edge has an associated starting time and ending time. The author created different graphs with information provided by Facebook and Wikipedia in order to evaluate the framework. `Tink` defines a temporal property graph model. It is available online (Lightenberg, Pei, Fletcher, and Pechenizkiy 2019), although we did not find information pertaining licensing.

To the best of our knowledge, currently both `Flink` and `Spark` are the most widely-known distributed processing frameworks (we note `GraphTau`, although its code is not available, is built over `Spark`) based on dataflow programming. While the use of dataflows grants flexibility to program implementation and execution by decoupling the program logic from how it is translated to the workers of a cluster, the graph libraries of these systems do not allow in an efficient way for a graph to be updated using stream-processing semantics while also maintaining the graph structure during computation. It is possible to update graphs using these systems, but they make use of batch processing APIs for which the dataflow graphs must not become excessively big (or else dataflow plan optimizers may be *locked* in the phase of exploring the optimization space of the execution plan) and the graphs must be periodically written to secondary storage (as a solution to avoid having progressively bigger execution plans).

`Flink`'s `Gelly` library has been used in `GRADOOP`, which is an open-source (Kevin Gómez 2020) (`Apache License 2.0`) distributed graph analytics research framework (Junghanns, Kießling, Teichmann, Gómez, Petermann, and Rahm 2018) under active development and providing higher-level operations. `GRADOOP` extends `Gelly` with additional specialized operators such as a graph pattern matching operator (which abstracts a cost-based query engine) and a graph grouping operator (implemented as a composition of map, filter, group and join transformations on Flink's DataSet). GRADOOP also adopts the Cypher query language (typically found in graph databases like Neo4j) to express logic that is translated to the relational algebra that underlies Flink's DataSet (Junghanns, Kießling, Averbuch, Petermann, and Rahm 2017).

In a similar way, Spark has its graph processing library GraphX which was built over the system's batch processing API, like the case of Flink's Gelly and also suffering from the same previously mentioned limitations. A higher-level API was designed to extend the functionalities of GraphX while harnessing Spark's DataFrame API. For this, the GraphFrames open-source (UC Berkeley, MIT, and Databricks 2020) (Apache License 2.0) library was created (Dave, Jindal, Li, Xin, Gonzalez, and Zaharia 2016). A look at its implementation reveals that it has less high-level operations than Gelly. Effectively, without simulating some of Gelly's API, equivalent programs in GraphX lend themselves to more conceptual verbosity due to the lack of syntactic sugar.

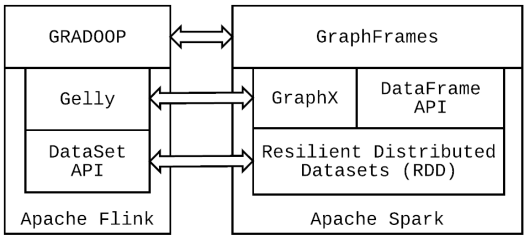


Figure 3.3: Contrast of the Flink and Spark distributed dataflow ecosystems for graph processing.

We display in Figure 3.3 parallels between Flink, Spark and the graph processing ecosystems built on top of them. Gelly's equivalent in Spark is GraphX, implemented in Scala. Vertices and edges are manipulated by using Spark's Resilient Distributed Datasets (RDDs), which can be viewed as a conceptual precursor to Flink's DataSet. Spark also offers the DataFrame API to enable tabular manipulation of data. GraphFrames is another graph processing library for Spark. While it has interoperability and a certain overlap with the functionality offered in GraphX, it integrates the tabular perspective supported by Spark's DataFrame API and also supports performing traversal-like queries of the graph via SparkSQL. In this way, GraphFrames provides graph analytics capabilities in Spark much the same way GRADOOP does in Flink.

The next two examples, X-Stream and Chaos are grouped together as they brought relevance to the edge-centric (TLAE) model and employed it to explore novel ways to balance network latencies and use of SSDs to increase performance:

- X-Stream (Roy, Mihailovic, and Zwaenepoel 2013). A system that provided an alternative view to the traditional *vertex-centric* approach. It is based on considering computation from the perspective of edges instead of vertices and experiments optimized the use of storage I/O both locally and on the cloud. `X-Stream` is an open-source system written in `C++` which introduced the concept of *edge-centric* graph processing via streaming partitions. `X-Stream` exposes an edge-centric scatter-gather programming model that was motivated by the lack of access locality when traversing edges, which makes it difficult to obtain good performance. State is maintained in vertices. This tool uses the streaming partition, which works well with RAM and secondary (SSD and Magnetic Disk) storage types. It does not provide any way by which to iterate over the edges or updates of a vertex. A sequential access to vertices leads to random access of edges which decreases performance. `X-Stream` is innovative in the sense that it enforces sequential processing of edges (edge-centric) in order to improve performance. It is available (Roy, Mihailovic, and Zwaenepoel 2015) under the `Apache License 2.0`.

- `Chaos` (Roy, Bindschaedler, Malicevic, and Zwaenepoel 2015). A system written in `C++` which had its foundations on `XStream`. On top of the secondary storage studies performed in the past, graph processing in `Chaos` achieves scalability with multiple machines in a cluster computing system. It is based on different functionalities: load balancing, randomized work stealing, sequential access to storage and an adaptation of `X-Stream`'s streaming partitions to enable parallel execution. `Chaos` is composed of a storage sub-system and a computation sub-system. The former exists concretely as a storage engine in each machine. Its concern is that of providing edges, vertices and updates to the computation sub-system. Previous work on `X-Stream` highlighted that the primary resource bottleneck is the storage device bandwidth. In `Chaos`, the storage and computation engines' communication is designed in a way that storage devices are busy all the time – thus optimizing for the bandwidth bottleneck. It was released (Roy, Bindschaedler, Malicevic, and Zwaenepoel 2016) under the `Apache License 2.0`.

The following graph processing systems were grouped together because each of the improvements they proposed are important concerns to be aware of in designing graph processing systems.

- `PowerLyra` (Chen, Shi, Chen, and Chen 2015) is a graph computation engine written in `C++` which adopts different partitioning and computing strategies depending on vertex types. The authors note that most systems use a *one-size-fits-all* approach. They note that `Pregel` and `GraphLab` focus in hiding latency by evenly distributing vertices to machines, making resources locally accessible. This may result in imbalanced computation and communication for vertices with higher degrees (frequent in scale-free graphs). Another provided design example is that of `PowerGraph` and `GraphX` which focus on evenly parallelizing the computation by partitioning edges among machines, incurring communication costs on vertices, even those with low degrees. PowerLyra was released under the Apache License 2.0 (Chen, Shi, Chen, and Chen 2018).

- Kineograph (Cheng, Hong, Kyrola, Miao, Weng, Wu, Yang, Zhou, Zhao, and Chen 2012) is a system which combines snapshots allowing full processing in the background and explicit alternative/custom functions that, besides assessing updates' impact, also apply them incrementally, propagating their outcome across the graph. It is a distributed system to capture the relations in incoming data feeds, built to maintain timely updates against a continuous flow of new data. Its architecture uses two types of computational node, *ingest* nodes to register graph update operations as identifiable transactions, which are then distributed to *graph* nodes. Nodes of the latter type form a distributed in-memory key/value store. Kineograph performs computation on static snapshots, simplifying algorithm design. We did not find its source code online.

- Tornado (Shi, Cui, Shao, and Tong 2016) is a system for real-time iterative analysis over evolving data. It was implemented over Apache Storm and provides an asynchronous bounded iteration model, offering fine-grained updates while ensuring correctness. It is based on the observations that: *1)* loops starting from *good enough* guesses usually converge quickly; *2)* for many iterative methods, the running time is closely related to the approximation error. From this, an execution model was built where a main loop continuously gathers incoming data and instantly approximates the results. Whenever a result request is received, the model creates a branch loop from the main loop. This branch loop uses the most recent approximations as a guess for the algorithm. We did not find its source code online.

- KickStarter (Vora, Gupta, and Xu 2017) is a system that debuted a runtime technique for trimming approximation values for subsets of vertices impacted by edge deletions. The removal of edges may invalidate the convergence of approximate values pertaining monotonic algorithms. KickStarter deals with this by identifying values impacted by edge deletions and adapting the network impacts before the following computation, achieving good results on real-world use-cases. Despite this, by focusing on monotonic graph algorithms, its scope is narrowed to selection-based algorithms. For this class, updating a vertex value implies choosing a neighbour under some criteria. KickStarter is now known as GraphBolt, a recent work (Mariappan and Vora 2019) licensed under the MIT License (Mariappan and Vora 2020) which offers a generalized incremental programming model enabling development of incremental versions of complex aggregations. Both were written in C++.

- Pixie (Eksombatchai, Jindal, Liu, Liu, Sharma, Sugnet, Ulrich, and Leskovec 2018) is a graph-based scalable real-time recommendation system used at Pinterest. Using a set of user-specific *pins* (in Pinterest, users have boards in which they store pins, where each pin is a combination of image and text), Pixie chooses in real-time the pins that are most related to the query, out of billions of candidates. With this system, Pinterest was able to execute a custom (Pixie Random Walk) algorithm on an object graph of 3 billion vertices and 17 billion edges. On a single server, they were able to serve around 1200 recommendation requests per seconds with 60 millisecond latency. The authors note that the deployment of `Pixie` benefited from large RAM machines, using a cluster of Amazon AWS r3.8xlarge machines with 244GB RAM. They fitted the pruned Pinterest graph (3 billion vertices, 17 billion edges) in around 120GB of RAM, in a setup that yielded the following advantages: random walk not forced to cross machines, which increases performance; multiple walks can be executed on the graph in parallel; the system can be parallelized and scaled by adding more machines to the cluster. This system is a relevant case study (of applying graph theory to recommendation systems at scale) as a scalable system for processing on large graphs a biased random walk algorithm (with user-specific preferences) while using graph pruning techniques to disregard large boards that are too diverse and diffuse the random walk (the non-pruned graph has 7 billion vertices and 100 billion edges). We did not find the source code available online.

- `FlowGraph` (Chaudhry 2019) is a system that proposes a syntax for a language to detect temporal patterns in large-scale graphs and introduces a novel data structure to efficiently store results of graph computations. This system is a unification of graph data with stream processing considering the changes of the graph as a stream to be processed and offering an API to satisfy temporal patterns. We did not find its source code available.

- `GPS` (Salihoglu and Widom 2013b) is an open-source (`BSD License`) scalable graph processing system written in `Java` and allowing fault-tolerant and easy-to-program algorithm execution on very large graphs. It adopts `Pregel`'s vertex-centric API and extends it with: features to make global computations easier to express and more efficient; dynamic repartitioning scheme to reassign vertices to different workers during computation based on messaging patterns; distribution of high-degree vertex adjacency lists across all computer nodes to improve performance (something that `PowerGraph` and `PowerLyra` later adopted). It was designed to run on a cluster of machines such as Amazon EC2, over which the authors tested their work. `GPS`'s initial version was run on up to 100 Amazon EC2 large instances and on graphs of up to 250 million vertices and 10 billion edges. It is open-source and available under the `BSD License` (Salihoglu and Widom 2013a).

- `GoFFish` (Simmhan, Kumbhare, Wickramaarachchi, Nagarkar, Ravi, Raghavendra, and Prasanna 2014) is a sub-graph centric programming abstraction and framework co-designed with a distributed persistent graph storage for large scale graph analytics on commodity clusters, aiming to combine the scalability of the vertex-centric (TLAV) approach with flexibility of shared-memory sub-graph computation (TLAG). It is written in `Java`. `GoFFish` states that two sub-graphs many not share the same vertex, but they can have remote edges that connect their vertices, provided that the sub-graphs are on different partitions. If two sub-graphs in the same partition share an edge, by definition they are merged into a single-sub-graph. It was evaluated with a cluster of 12 nodes each with 8-core Intel Xeon CPUs, 16 GB RAM and 1 TB SATA HDD. The authors compare the execution of `GoFFish` (one worker per computational node) with `Giraph` (default two workers per computational node), achieving faster execution times for algorithms such as PageRank, connected components and single-source shortest-paths. Its source code is available though we did not find any information pertaining licensing. While its source code is available (Simmhan, Kumbhare, Wickramaarachchi, Nagarkar, Ravi, Raghavendra, and Prasanna 2017), we did not find information regarding licensing.

- `FBSGraph` (Zhang, Liao, Jin, Gu, and Zhou 2017) presents a forward and backward sweeping execution method to accelerate state propagation for asynchronous graph processing. In asynchronous graph processing, each vertex maintains a state which can be asynchronously updated in an iterative fashion. The method presented in `FBSGraph` relies on the observation that state can be propagated faster by processing vertices sequentially along the graph path in each round. They achieve greater execution speed when analysing several graph algorithms across a set of datasets, comparing against systems such as `PowerGraph` and `GraphLab`. We did not find its source available.

- `GrapH` (Mayer, Tariq, Mayer, and Rothermel 2018) is a graph processing system written in `Java` that uses vertex-cut graph partitioning that takes into consideration the diversity of vertex traffic and the heterogeneous costs of the network. It relies on a strategy of adaptive edge migration to reduce the frequency of communication across expensive network links. For this work, the authors focused on vertex-cut as it has better partitioning properties for real-world graphs that have power-law degree distributions. `GrapH` has two partitioning techniques, *H-load* which is used for the initial partitioning of the graph so that each cluster worker can load it into local memory, and *H-adapt*, a distributed edge migration algorithm to address the dynamic heterogeneity-aware partitioning problem. In evaluation, `GrapH` outperformed `PowerGraph`'s vertex-cut partitioning algorithm with respect to communication costs. While its source code is available (Mayer, Tariq, Mayer, and Rothermel 2016), we found no information on licensing.

- `Julienne` (Dhulipala, Blelloch, and Shun 2017) is built over `Ligra` (`C++`) and provides an interface to maintain a collection of buckets under vertex insertions and bucket deletions. They evaluated under bucketing algorithms such as weighted breadth-first search, k-core and approximate set cover. The authors describe as *bucketing-based* algorithms those that maintain vertices in a set of unordered buckets - and in each round, the algorithm extracts the vertices contained in the lowest (or highest) bucket to perform computation on them. Then, it can update the buckets containing the extracted vertices or their neighbours. For example, weighted breadth-first search processes vertices level by level, where level i contains all vertices at distance i from the source vertex. The system was tested in a multi-core machine with 72 cores (4 CPUs at 2.4GHz) and 1TB of main memory, achieving performance improvements on several data sets when compared to systems such as base `Ligra` and `Galois`. We did not find its source code available.

- `GraphD` (Yan, Huang, Liu, Chen, Cheng, Wu, and Zhang 2017) is an out-of-core system inspired by `Pregel` and targeting efficient big graph processing using a small cluster of commodity machines connected by Gigabit Ethernet, contrasting with other out-of-core works that focus on specialized hardware. The authors focus on a setting that sees vertex-centric programs being data-intensive, as the CPU cost of computing a message is small when compared to the network transmission cost. `GraphD` masks disk I/O overhead with message transmission though parallelism of computation and communication. It eliminates the need for (expensive) external-memory join or group-by operations, which are required in other systems such as `Chaos`. It was evaluated on PageRank, single-source shortest-paths and connected components. `GraphD` was evaluated against distributed out-of-core systems `Pregelix`, `HaLoop` and `Chaos`, against single-machine systems `GraphChi` and `X-Stream` and representative in-memory systems `Pregel` and `Giraph`, achieving better performance in some scenarios. We did not find its source available.

- `TurboGraph++` (Ko and Han 2018) is a graph analytics system that exploits external memory for scale-up without compromising efficiency. It introduced an abstraction called nested windowed streaming to achieve scalability and increase efficiency in processing neighbourhood-centric analytics (in which the total size of neighbourhoods around vertices can exceed the available memory budget). This streaming model regards a sequence of vertex values and an adjacency list stream. The goal is to efficiently support the k-walk neighbourhood query (a class of graph queries defined by the authors, where walks are enumerated and then computation is done for each one) with fixed size memory. In the model, during user computation, they define an update stream as the sequence of updates generated to the ending vertex of each walk, with each update represented as a pair of target vertex ID and update value. `TurboGraph++` has the goal of balancing the workloads across machines, which requires balancing the number of edges and the number of high-degree and low-degree vertices among machines. It also focuses on balancing the number of vertices on each machine so that each one requires the same memory budget. We did not find its source code available online.

- `GraphIn` (Sengupta, Sundaram, Zhu, Willke, Young, Wolf, and Schwan 2016) is a dynamic graph analytics framework proposed to handle the scale and evolution of real-world graphs. It aimed to improve over approaches to processing dynamic graphs which repeatedly run static graph analytics on stored snapshots. GraphIn proposes an adaptation of gather-apply-scatter (GAS) called I-GAS which enables the implementation of incremental graph processing algorithms across multiple CPU cores. It also introduces an optimization heuristic to choose between static or dynamic execution based on built-in and user-defined graph properties. Native and benchmarking code were implemented in C++ and for experimental evaluation it was compared to GraphMat and STINGER. The heuristic-base computation made GraphIn faster than systems using fixed strategies. We did not find its source code available.

The following works focus on specific techniques such as using specific hardware such as SSDs or GPUs. We first list frameworks and systems that were proposed in the last years to use the single-instruction multiple-data (SIMD) capabilities of GPUs for graph processing:

- MapGraph (Fu, Personick, and Thompson 2014) is a high-performance parallel graph programming framework, able to achieve up to 3 billion traversed edges per second using a GPU. It represents the graph with a compressed sparse row (CSR) data structure and chooses different scheduling strategies depending on the size of the *frontier* (the set of vertices that are active in a given iteration). It encapsulates the complexity of the GPU architecture while enabling dynamic runtime decisions among several optimization strategies. Users need only to write sequential C++ code to use the framework. The early MapGraph work was first available as an open-source project (Fu, Personick, and Thompson 2016) licensed under the Apache License 2.0, but it has been integrated in the former line of products of Blazegraph, also available online (Systap 2020).

- CuSha (Khorasani, Vora, Gupta, and Bhuyan 2014) is a CUDA-based graph processing framework written in C++ which was motivated by the negative impact that irregular memory accesses have on the compressed sparse row graph (CSR) representation. CuSha overcomes this by: *1)* organizing the graph into autonomous sets of ordered edges called *shards* (a representation they call *G-Shards*) unto which GPU hardware resources are mapped for fully coalesced memory accesses; *2)* accounting for input graph properties such as sparsity (the sparser the graph, the smaller the computation windows) to avoid GPU under-utilization (*Concatenated Windows*, or *CW*). This framework allows users to define vertex-centric algorithms to process large graphs on GPU. It is open-source (Khorasani, Vora, Gupta, and Bhuyan 2015) and available under the MIT License.

- Gunrock (Wang, Davidson, Pan, Wu, Riffel, and Owens 2016; Wang, Pan, Davidson, Wu, Yang, Wang, Osama, Yuan, Liu, Riffel, et al. 2017) is an open-source (Wang, Davidson, Pan, Wu, Riffel, and Owens 2020) (Apache License 2.0) CUDA library for graph processing targeting the GPU and written in C. It implements a data-centric abstraction focused on operations on a vertex or edge frontier. For different graph algorithms, it achieved at least an order of magnitude speedup over PowerGraph and better performance than any other high-level GPU graph library at the time. Its operations are bulk-synchronous and manipulate a frontier, which is a subset of the edges or vertices within the graph that is relevant at a given moment in the computation. Gunrock couples high-performance GPU computing primitives and optimization strategies with a high-level programming model to quickly develop new graph primitives. It was evaluated using breadth-first search, depth-first search, single-source shortest paths, connected components and PageRank.

- Lux (Jia, Kwon, Shipman, McCormick, Erez, and Aiken 2017) is a distributed multi-GPU system written in C++ for fast graph processing by exploiting aggregate memory bandwidth of multiple GPUs and the locality of the memory hierarchy of multi-GPU clusters. It proposes a dynamic graph repartitioning strategy to enable well-balanced distribution of workload with minimal overhead (improving performance by up to 50%), as well as a performance model providing insight on how to choose the optimal number of computational nodes and GPUs to optimize performance. Lux is aimed at graph programs that can be written with iterative computations. Vertex properties are read-only in each iteration, with updates becoming visible at the end of an iteration. It offers two execution models: *pull* which optimizes run-time performance of GPUs (enables optimizations like caching and locally aggregating updates in GPU shared memory); and *push*, which optimizes algorithmic efficiency (maintains a frontier queue and only performs computation over the out-edges of vertices in the frontier). Its source code is available (Jia, Kwon, Shipman, McCormick, Erez, and Aiken 2018) under Apache License 2.0.

- Frog (Shi, Luo, Liang, Zhao, Di, He, and Jin 2017) is a light-weight asynchronous processing framework written in C. The authors note that common colouring algorithms may suffer from low parallelism due to a large number of colours being needed to process large graphs with billions of vertices. Frog separates vertex processing based on colour distribution. They propose an efficient hybrid graph colouring algorithm, relying on a relaxed pre-partition method to solve vertex classification with a lower number of colours, without forcing all adjacent vertices to be assigned different colours. The execution engine of Frog scans the graph by colour, and all vertices under the same colour are updated in parallel in the GPU. For large graphs, when processing each partition, the data transfers are overlapped with GPU kernel function executions, minimizing PCIe data transfer overhead. It is open-source (Shi, Luo, Liang, Zhao, Di, He, and Jin 2018) and licensed under the GNU General Public License 2.0.

- Aspen (Dhulipala, Blelloch, and Shun 2019) is a graph-streaming extension of the Ligra interface, supporting graph updates. To support this, the authors developed and presented the C-tree data structure which achieves good cache locality, lowers space use and has operations which are efficient from a theoretical perspective. It applies a chunking scheme over the tree, storing multiple elements in a tree-node. The scheme takes the ordered set of elements that are represented. More relevant elements are stored in tree nodes, while the remaining ones are associated in tails of the tree nodes. It employs compression and supports parallelism. The authors evaluate it with the largest publicly-available graph, which has more than two hundred billion edges on a multi-core server with 1 TB memory. Source code is available online (Dhulipala, Blelloch, and Shun 2020) albeit no license information was provided.

- `Gluon` (Dathathri, Gill, Hoang, Dang, Brooks, Dryden, Snir, and Pingali 2018) was introduced as a new approach to create distributed-memory graph analytics systems able to use heterogeneity in partitioning policies, processor types (GPU and CPU) and programming models. To use `Gluon`, programmers implement applications in shared-memory programming systems of their choice and then interface the applications with `Gluon` to enable execution on heterogeneous clusters. `Gluon` optimizes communication by taking advantage of temporal and structural invariants of graph partitioning policies. It runs on shared-memory NUMA platforms and NVIDIA GPUs. Its programming model offers a small number of programming patterns implemented in `C++`, its library offers concurrent data structures, schedulers and memory allocators and the runtime executes programs in parallel, using parallelization strategies as optimistic and round-based execution. `Gluon` is available (Dathathri, Gill, Hoang, Dang, Brooks, Dryden, Snir, and Pingali 2020b) under the `3-Clause BSD License`.

- `Hornet` (Busato, Green, Bombieri, and Bader 2018) is a data structure for efficient computation of dynamic sparse graphs and matrices using GPUs. It is platform-independent and implements its own memory allocation operation instead of standard function calls. The implementation uses an internal data manager which makes use of block arrays to store adjacency lists, a bit tree for finding and reclaiming empty memory blocks and B^+ trees to manage them. It was evaluated using an NVIDIA Tesla GPU and experiments targeted the update rates it supports, algorithms such as breadth-first search (BFS) and sparse matrix-vector multiplication. `Hornet` is available (Busato, Green, Bombieri, and Bader 2020) under the `3-Clause BSD License`.

- `faimGraph` (Winter, Mlakar, Zayer, Seidel, and Steinberger 2018) introduced a fully-dynamic graph data structure performing autonomous memory management on the GPU. It enables complete reuse of memory and reduces memory requirements and fragmentation. The implementation has a vertex-centric update scheme that allows for edge updating in a lock-free way. It reuses free vertex indices to achieve efficient vertex insertion and deletion, and does not require restarting as a result of a large number of edge updates. `faimGraph` was benchmarked against `Hornet` on an NVIDIA GeForce GTX Titan Xp GPU using algorithms such as PageRank and triangle counting. Source code is available online (Dathathri, Gill, Hoang, Dang, Brooks, Dryden, Snir, and Pingali 2020a) without a specified license.

- `GraphCage` (Chen 2019) is a cache-centric optimization framework to enable highly efficient graph processing on GPUs. It was motivated by the random memory accesses which are generated by sparse graph data structures, which increase memory access latency. The authors note that conventional cache-blocking suffers from repeated accesses when processing large graphs on GPUs, and propose a throughput-oriented cache blocking scheme (*TOCAB*). `GraphCage` applies the scheme to both push and pull directions and coordinates with load balancing strategies by considering sparsity of sub-graphs. This technique is applied to traversal-based algorithms by considering the benefit and overhead in different iterations with working sets of different sizes. In its evaluation, `GraphCage` achieved in average lower execution times for one PageRank iteration compared to both `Gunrock` and `CuSha`. We did not find its source code available.

For more information on GPU use cases for graph processing approaches, we point the readers to (Shi, Zheng, Zhou, Jin, He, Liu, and Hua 2018).

- `FlashGraph` (Zheng, Mhembere, Burns, Vogelstein, Priebe, and Szalay 2015) is a graph processing engine implemented in `C++` over a user-space SSD file system designed for high IOS and very high levels of parallelism. Vertex state is stored in memory while edge lists are on SSDs. Latency is hidden by overlapping computation with I/O, a concept similar to `X-Stream` and `Chaos`, and edges lists are only accessed if requested by applications from SSDs. `FlashGraph` has a vertex-centric (TLAV) interface, its designed to reduce CPU overhead and increase throughput by conservatively merging I/O requests, and the authors demonstrate that `FlashGraph` in semi-external memory executes many algorithms with a performance of up to 80% of the in-memory implementation and It also outperformed `PowerGraph`. It is open-source (Zheng, Da and Mhembere, Disa and Burns, Randal and Vogelstein, Joshua and Priebe, Carey E and Szalay, Alexander S 2014) under the `Apache License 2.0`.

- `GraphSSD` (Matam, Koo, Zha, Tseng, and Annavaram 2019) is a semantic-aware SSD framework and full system solution to store, access and execute graph analytics. Instead of considering storage as a set of blocks, it accounts for graph structure while choosing graph layout, access and update mechanisms. `GraphSSD` innovates by considering a vertex-to-page mapping scheme and uses advanced knowledge of flash properties to reduce page accesses. It offers a simple API to ease development of applications accessing graphs as native data and its evaluation showcased average performance gains for basic graph data fetch functions on breadth-first search, connected components, random-walk, maximal independent set and PageRank. We did not find its source available.

3.10.1 Distributed Graph Processing: Notes

This section presents many distributed graph processing systems, divided by type of system architecture and including high-level extensions when available.

The first group presented in this section, that of vertex-centric systems, follows a common theme of further expanding on this model's concepts. It includes `Giraph`, which is the first open-source implementation of the `Pregel` model. Other mentioned systems are `Naiad`, which provided dataflow programming APIs that were used to implement graph algorithms, followed by `Flink` and `Spark`, more recent frameworks offering dataflow-based distributed graph processing through specific libraries (`Gelly` in `Flink` and `GraphX` in `Spark`). These systems were used to build time-oriented logic, with `Tink` built over `Flink` and `GraphTau` over `Spark`. Enhanced graph analytics were also implemented, with `GRADOOP` on `Flink` and `GraphFrames` on `Spark`. Overall, `Flink` and `Spark` are two actively-developed frameworks with supporting communities and are being used to enable graph processing use-cases as well as platforms to research new functionalities.

Edge-centric systems were placed in their own group, as we found the offering of this model not as widespread as vertex-centric, thus warranting its own focus. `X-Stream` showcased a competitive implementation of this model, with its extension `Chaos` highlighting how can it be applied to improve the use of I/O-bound resources.

Relevant graph processing design concerns are highlighted with `PowerLyra` which casts focus on the impact of vertex degree (especially regarding scale-free properties), `Kineograph` which explores incremental processing capabilities, `Tornado` which produces approximate results, `KickStarter` which explores trimming approximation values and incremental processing as well and `Pixie`, a scalable graph processing system used for real-time recommendations.

To this group of relevant graph processing design concerns we add other names such as `FlowGraph` which presents a language syntax to detect temporal patterns, `GPS` which adopts the `Pregel` model with expressivity of global computations and awareness of high-degree vertices, `GoFFish` which explores both vertex-centric (scalability) and sub-graph computation (flexibility), `FBSGraph` which explores ordered vertex processing to hasten state propagation, `GrapH` which explores partitioning techniques while considering heterogeneous network costs and vertex traffic, `Julienne` which focuses on bucketing algorithms, `GraphD` which explores out-of-core processing on clusters of commodity machines, `TurboGraph++` which achieves scale-up with external memory and `GraphIn` which adapts gather-apply-scatter to enable incremental graph processing algorithms across CPU cores.

The ability to harness GPUs and SIMD capabilities is relevant, and among such systems we note `MapGraph`, `CuSha`, `Gunrock`, `Lux`, `Frog`, `Aspen`, `Gluon`, `Hornet`, `faimGraph`, `GraphCage`. These systems were included for the different approaches they had to the use this type of hardware. It is not a trivial task to design systems that are both generic as well as capable of making use of special-purpose hardware such as GPUs. They were thus grouped for this use of a hardware type that is essential to machine learning, blockchain technologies and other activities. We believe it would also be interesting to see frameworks such as Flink and Spark that offer many high-level functionalities incorporate GPU-oriented processing capabilities.

The role of I/O plays a relevant role in other systems, but it is further brought to the forefront with FlashGraph which stores vertex state in main memory and edge lists on SSDs, and GraphSSD which considers a vertex-to-page mapping scheme and special use of flash properties to reduce page accesses.

3.11 Graph Databases

Here we list and describe different graph databases which we group together according to their type of supported graph model (e.g., property graph model, RDF, hybrid). Upon developing this list, we observe regarding graph query languages that they either implement their own custom language (neglecting interoperability even though there may be automatic converters developed by third parties), or they use those that have been improved, standardized and adopted by many projects (e.g., Cypher, Gremlin, SPARQL). The list includes both open-source and commercial products, as well as database systems implemented to explore novel techniques as part of research activities.

We showcase these different graph database solutions in Table 3.2, in which we present relevant properties for assessing them: the circle • on the *ACID*, *Visual* and *Scale* (updating a graph that is distributed) columns indicate they are present. *Models* lists the supported data models: property graph model (PG), RDF, multi-model (*MM*) or others (*O*). *Languages* lists the programming languages that interface with the database, either directly or by being supported by the underlying interfaces (e.g., TinkerPop drivers). *GQLs* lists only directly supported graph query languages (and shown in italic if they are not standard) - those supported using compatibility tools as seen previously are not included. *License* lists the licenses of the open-source project or of the free edition of a commercial product: AL 2.0 is Apache License 2.0, CC 1.0 is Commons Clause 1.0, aGPL v3 is the Affero General Public License 3, (GPL) v3 is GNU General Public License (GPL) v3. *Limits* covers free version limitations in commercial products (*N/A* if it does not apply). If a database has *Custom* license and limits are not *N/A*, it means it has a non-standard free license.

We consider these properties to be outstanding features that are important for developers and users to assess their suitability for their use-cases. They provide information on guarantees of graph data storage, parallelism of computation and supported graph sizes, visualization capabilities and the ability to use them freely (regarding both open-source as well as free versions of commercial products):

- Is it ACID compliant (*ACID*)?
- Does the specific graph database ecosystem offer analytics capabilities (e.g., first-party, third-party, none) to complement the OLTP focus of the graph database itself (*Visual*)?

- Does it offer scale-out/horizontal scalability (e.g., Neo4j is very well-known but it does not support sharding, only replication when using multiple machines) to improve performance and support bigger graphs (*Scale*)?
- What graph models does it support, is it for example `RDF`, property graph or others (*Models*)?
- What languages can be used to program functionality that uses the database product, for example `Go`, `Java`, `.NET`, `Python` or others (*Languages*)?
- What graph query languages (e.g., `Cypher`, `Gremlin`, `SPARQL`, custom) does it support (*GQLs*)?
- What license (or types of licenses) is it subjected to - for example, is it a common license type or other (*License*)?
- For database products that have both free and commercial/proprietary editions, what are the features missing from the free versions - the focus here is not on marketing-speech such as five-nines availability, but features that are easy to implement but are clearly missing from the free version to stimulate purchases (*Limits*)?

Database	*ACID*	*Visual*	*Scale*	*Models*	*Languages*	*GQLs*	*License*	*Limits*
Alibaba GDB (2020)	●		●	PG	Go, Java, .NET, Node.js, Python	Gremlin	*Paid*	*N/A*
ChronoGraph(2017)(2020)	●			PG	Java	Gremlin	aGPL v3	*N/A*
DSE (2016)	●	●	●	PG	Java	Gremlin	*Paid*	*N/A*
Dgraph (2020a, 2020b)	●		●	PG	Java, JavaScript, Python	*GraphQL*	AL 2.0	*Admin*
Graphflow (2017)(2019)				PG	Java	*Cypher*	AL 2.0	*N/A*
JanusGraph (2017, 2020)	●		●	PG	Elixir, Go, Java, .NET, PHP, Python, Ruby, Scala	Gremlin	AL 2.0	*N/A*
Nebula Graph (2020)	●	●	●	PG	Go, Java, Python	*nGQL*	AL 2.0 CC 1.0	*N/A*
Neo4j (2012, 2020)	●	●		PG	C/C++, Clojure, Go, Haskell, Java, JavaScript, .NET, Perl, PHP, Python, R, Ruby	Cypher	GPL v3	*Size*
RedisGraph (2019, 2020)				PG	Elixir, Go, Java, JavaScript, PHP, Python, Ruby, Rust	*Cypher*	*Custom*	*N/A*
Hana (2013, 2018, 2018)	●	●	●	PG	*N/A*	Cypher	*Paid*	*N/A*
Sparksee (2011, 2015)	●			PG	C++, .NET, Java, Objective-C, Python	*N/A*	*Custom*	*Size*
TigerGraph (2019, 2020)	●		●	PG	*N/A*	*GSQL*	*Custom*	*Scale*
Weaver (2016, 2016)	●		●	PG	C++, Python	*N/A*	*Custom*	*N/A*
AllegroGraph (1984)	●		●	RDF	C#, C, Common Lisp, Clojure, Java, Perl, Python	SPARQL	EPL v1	*Size*
BlazeGraph (2020)	●			PG, RDF, *O*	.NET, Python	Gremlin, SPARQL	GPL v2	*N/A*
BrightstarDB (2015)	●			RDF	.NET	SPARQL	MIT	*N/A*
Cray Graph Engine (2018)	●		●	RDF	Java, Python	SPARQL	*Paid*	*N/A*
Ontotext GraphDB (2020)	●	●	●	RDF	Java, JavaScript	SPARQL	*Custom*	*Scale*
Neptune (2018, 2020)	●		●	PG,RDF	*Managed*	Gremlin, SPARQL	*Paid*	*N/A*
AnzoGraph DB (2020)	●	●	●	PG,RDF	Java,C++	Gremlin, SPARQL	*Custom*	*Scale*
ArangoDB (2020)	●	●	●	PG	Go, Java, JavaScript, PHP	*AQL*	AL 2.0	*Admin*
IBM System G (2015)	●	●	●	PG, RDF	*Managed*	Gremlin	*Paid*	*N/A*
OrientDB (2013, 2020)	●	●	●	PG, *O*	Clojure, Go, Java, JavaScript,.NET, Node.js, PHP, Python, R, Ruby, Scala	6	AL 2.0	*Visual*
OSG (2020)	●	●	●	PG, RDF	Java	PGQL, SPARQL	*Paid*	*N/A*
Stardog (2020)	●	●	●	RDF	Clojure, Groovy, Java, JavaScript, .NET	GraphQL, Gremlin, SPARQL	*Paid*	*Trial*
Virtuoso (2012, 2020)	●		●	PG, *O*	C/C++, C#, Java, JavaScript, .NET, PHP, Python, Ruby, Visual Basic	SPARQL	GPL v2	*Admin*
Cosmos DB (2018, 2019)	●		●	*MM*	Gremlin	*Managed*	*Paid*	*N/A*
FaunaDB (2020)	●		●	*MM*	Android, C#, Go, Java, JavaScript, Python, Ruby, Scala, Swift	*FQL*	*Custom*	*Quota*
Cayley (2017, 2010)	●	●	●	RDF	*N/A*	Gizmo, GraphQL	AL 2.0	*N/A*
HyperGraphDB (2010)				*HG*	Java	*N/A*	AL 2.0	*N/A*
Objectivity/DB (2016)	●		●	*O*	C++, C#, Java, Python	*N/A*	*Paid*	*N/A*

Table 3.2: Summary of graph database distinctive features.

For further information on the evolution of graph databases, we point the reader to (Angles and Gutierrez 2008; Pokorný 2015), which covers, among other aspects, data models and query languages. The following graph databases we list are focused on the property graph model. Their information describes the use of this model; we found no graph query languages for `RDF` in their features, even if these databases are described as multi-model:

- `Alibaba Graph Database (GD)` (Cloud 2020) is a cloud-oriented (thus supporting horizontal scalability) graph database service supporting ACID transactions and the `TinkerPop` stack. It supports the property graph model, the `Gremlin` query language and there are programming interfaces for `Go`, `Java`, `.NET`, `Node.js` and `Python`. We did not find references to visualization capabilities (visual feedback to query construction and execution).

- `ChronoGraph` (Haeusler, Trojer, Kessler, Farwick, Nowakowski, and Breu 2017) is a `TinkerPop`-compliant (offering `Gremlin` to query the data in property graph model) graph database supporting ACID transactions, system-time content versioning and analysis. It is implemented as a key-value store enhanced with temporal information, using a B-tree data structure. The project is available online (Haeusler, Trojer, Kessler, Farwick, Nowakowski, and Breu 2020) under the `aGPL v3` (open-source and academic purposes) and commercial licenses are available on demand. We did not find visualization functionalities accompanying it.

- `DataStax Enterprise Graph (DSE)` (DataStax, Inc. 2016) is a proprietary fork of `Titan`, licensed under the `Apache License 2.0` and supporting `Java`. It integrates with the `Cassandra` (Lakshman and Malik 2010) distributed database (over which it provides graph data models) and supports `TinkerPop` (property graph with `Gremlin`). It is complemented by the `DataStax Studio`, which allows for interactive querying and visualization of graph data similarly to `Neo4j` and `SAP Hana Graph`.

- `Dgraph` (Dgraph Labs, Inc. 2020a; Dgraph Labs, Inc. 2020b) was written in `Go` and it is a distributed graph database offering horizontal scaling and ACID properties. It is built to reduce disk seeks and minimize network usage footprint in cluster scenarios. `Dgraph` is licensed under two licenses: the `Apache License 2.0` and a `Dgraph Community License`. It automatically moves data to rebalance cluster shards. It uses a simplified version of the `GraphQL` query language. Support for `Gremlin` or `Cypher` has been mentioned for the future but will depend on community efforts. `Dgraph` has a scalability advantage over `Neo4j` as the latter may have multiple servers but they are merely replicas, while the former can grow horizontally (vertical scaling is expensive). There is a proprietary enterprise version (conditions specified under a custom `Dgraph Community License`) with advanced features for backups and encryption. Official clients include `Java`, `JavaScript` and `Python`. To the best of our knowledge `Dgraph` does not offer visualization and global analytics functionalities.

- `Graphflow` (Kankanamge, Sahu, Mhedbhi, Chen, and Salihoglu 2017; Mhedbhi and Salihoglu 2019; Mhedbhi, Gupta, Khaliq, and Salihoglu 2020) was released as a prototype active graph database. It is an in-memory graph store supporting the property graph model and supports one-time as well as continuous sub-graph queries. `Graphflow` supports this using a one-time query processor called *Generic Join* and a *Delta Join* which enables the continuous sub-graph queries. It extends the `openCypher` language with triggers to perform actions upon certain conditions. We did not find information regarding its direct use beyond academic purposes nor about supporting ACID transactions. Its code is available online (Kankanamge, Chathura and Sahu, Siddhartha and Mhedbhi, Amine and Chen, Jeremy and Salihoglu, Semih 2017) under the `Apache License 2.0`.

- `JanusGraph` (JanusGraph Authors 2017) is an open-source project licensed (Authors 2020) under the `Apache License 2.0`. A database optimized for storing (in adjacency list format) and querying large graphs with (billions of) edges and vertices distributed across a multi-machine cluster with ACID transactions. `JanusGraph`, which debuted in 2017, is based on the source `Java` code base of the `Titan` graph database project and is supported by the likes of Google, IBM and the Linux Foundation, to name a few. Like `Titan`, it supports `Cassandra`, `HBase` and `BerkeleyDB`. It was designed with a focus on scalability and it is in fact a transactional database aimed at handling many concurrent users, complex traversals and analytic queries. `JanusGraph` can integrate platforms such as `Spark`, `Giraph` and `Hadoop`. It also natively integrates with the `TinkerPop` graph stack, supporting `Gremlin` applications, the query language and its graph server, with graphs in the property graph model. Due to supporting `TinkerPop`, one may use one of its drivers to use `Gremlin` from `Elixir`, `Go`, `Java`, `.NET`, `PHP`, `Python`, `Ruby` and `Scala`. It supports global analytics using `Spark` integration as well.

- `Nebula Graph` is an open-source graph database (available online (VESoft Inc. 2020)) licensed under `Apache License 2.0`, provides a custom `Nebula Graph Query Language (nGQL)` with syntax close to `SQL` and `Cypher` support is planned. It supports the property graph model, ACID transactions and is implemented with a separation of storage and computation, being able to scale horizontally. It supports multiple storage engines like `HBase` (George 2011) (implementing the graph logic over these key-value stores) and `RocksDB` (Facebook Database Engineering Team 2012) and has clients in `Go`, `Java` and `Python`. It also has the complementing `Nebula Graph Studio` for interactive visual querying and analytics.

- `Neo4j` (Webber 2012) is a graph database with multiple editions (Neo4j Inc. 2020): a community edition licensed under the free `GNU General Public`

License (GPL) v3, a commercial one and also an advanced edition licensed under AGPLv3. It supports different programming languages C/C++, Clojure, Go, Haskell, Java, JavaScript, .NET, Perl, PHP, Python, R and Ruby. Neo4j is optimized for highly-connected data. It relies on methods of data access for graphs without considering data locality. Neo4j's graph processing consists of mostly random data access. For large graphs which require out-of-memory processing, the major performance bottleneck becomes the random access to secondary storage. The authors created a system which supports ACID transactions, high availability, with operations that modify data occurring within transactions to guarantee consistency. It uses the query language Cypher and data is stored on disk as fixed-size records in linked lists. Neo4j has a library offering many different graph algorithms. As far as we know, Neo4j's scale-out capabilities are only true for read operations. All writes are directed to the Neo4j cluster master, an architecture which has its limitations. Among other uses, Neo4j has also been employed for building applications using the GRANDstack framework (Lyon 2020). Neo4j also has an interactive graph explorer to query and update specific elements of the graph.

- RedisGraph (Cailliau, Davis, Gadepally, Kepner, Lipman, Lovitz, and Ouaknine 2019) is a property graph database which uses sparse matrices to represent a graph's adjacency matrix and uses linear algebra for graph queries. It uses custom memory-efficient data structures stored in RAM, having on-disk persistence and tabular result sets. Queries may be written in a subset of Cypher and are internally translated into linear algebra expressions. It has a custom license and client libraries for RedisGraph have been developed in Elixir, Go, Java, JavaScript, PHP, Python, Ruby and Rust, complementing existing accesses that Redis already supports. As far as we know, it only works in single-server mode (bounded by the machine's RAM) and it does not support ACID properties. It has a Community Edition under a custom license. source code available online (Ltd. 2020) under a custom license.

- SAP Hana Graph (Rudolf, Paradies, Bornhövd, and Lehner 2013; Hwang 2018; SAP SE 2018) is a column-oriented, in-memory relational database management system. It performs different type of data analysis, among which graph data processing with the property graph model and ACID transactions. This graph functionality includes interpretation of Cypher and a visual graph manipulation tool. Its graph processing capabilities have served use cases like fraud detection and route planning. We did not find source code available online.

- Sparksee (Martinez-Bazan, Gomez-Villamor, and Escale-Claveras 2011; Sparsity Technologies 2015) (formerly DEX) is a property graph database offering ACID transactions and representing the graph using bitmap data structures with high compression rates (with each bitmap partitioned into chunks that fit disk pages). The graphs in Sparksee are labelled multigraphs and it has multiple licenses depending on the purpose, with free licenses for evaluation, research and development. It offers APIs in `C++`, `.NET`, `Java`, `Objective-C`, `Python` and mobile devices. We found no capabilities for data visualization in `Sparksee`, though it is able to export data to formats supported by third-party software.

- `TigerGraph` (Deutsch, Xu, Wu, and Lee 2019; TigerGraph 2020) is a commercial graph database (formerly `GraphSQL`) implemented in `C++` and comes in three versions: developer edition (supporting only single-machine, no distribution and is only for non-production, research or educational purposes), cloud edition (as a managed service) and enterprise edition (allowing for horizontal scalability - distributed graphs). It supports ACID consistency, access through a `REST` API, has a custom `SQL`-like query language (`GSQL`) and features a graphical user interface named `GraphStudio` to perform interactive graph data analytics. The `TigerGraph` model was designed to support graph vertices, edges and their attributes to support an engine that performs massively-parallel processing to compute queries and analytics. Each vertex and edge acts as both a unit of storage and computation, integrating and extending both TLAV and TLEV paradigms. It supports the property graph model plus extensions to enable the massively-parallel processing. We did not find available source code.

- `Weaver` (Dubey, Hill, Escriva, and Sirer 2016) is an open-source (Dubey 2016) graph database (custom permissive license) for efficient, transactional graph analytics. It introduced the concept of refinable timestamps. It is a mechanism to obtain a rough ordering of distributed operations if that is sufficient, but also fine-grained orderings when they become necessary. It is capable of distributing a graph across multiple shards while supporting concurrency. Refinable timestamps allow for the existence of a multi-version graph: write operations use their timestamps as a mark for vertices and edges. This allows for the existence of consistent versions of the graph so that long-running analysis queries can operate on a consistent version of the graph, as well as historical queries. `Weaver` is written in `C++`, offering binding options for `Python`. We did not find any support for popular graph query languages.

The following graph databases we list are focused on the `RDF` data model and variations (including support for the property graph model by representing them as `RDF` (Hartig 2014)):

- `AllegroGraph` (Inc. 1984) is a proprietary commercial graph database with clients under `Eclipse Public License v1 (EPL v1)` which supports several programming languages (`C#`, `C`, `Common Lisp`, `Clojure`, `Java`, `Perl`, `Python`, `Scala`) that was purpose-built for `RDF` (triple-store). It supports an array of mechanisms to access the information it stores, namely reasoning with ontology (`RDFS++ Reasoning`), materialized reasoning (generating new triples based on inference rules - `OWL2 RL Materialized Reasoner`, `SPARQL` queries, `Prolog` and also low-level APIs.

- BlazeGraph (Systap 2020) (formerly Bigdata) is an RDF database able to support up to billions of edges in a single machine and available under the GNU General Public License v2.0 supporting SPARQL and the TinkerPop Blueprints API. It has .NET and Python clients. One of its associated internal projects is blazegraph-gremlin, which allows the storage of property graphs internally in RDF format, which can then be queried with SPARQL. It essentially has an alternative approach to RDF reification, giving labelled property graph capabilities to RDF graphs, with the ability to query the graphs in Gremlin as well.

- BrightstarDB is an open-source (BrightstarDB 2015) multi-threaded multi-platform (including mobile) .NET RDF store, supporting SPARQL and binding of RDF resources to .NET dynamic objects (it has tools to use .NET interfaces and generate concrete classes to persist their data in BrightstarDB). It is licensed under the MIT License and there is also an Enterprise version. BrightstarDB supports single-threaded writes and multi-threaded reads, with ACID transactions. It does not support horizontal scaling.

- Cray Graph Engine (CGE) (Rickett, Haus, Maltby, and Maschhoff 2018) is an RDF triple database offering the SPARQL query language. As a commercial product, it was designed while considering different architectures of proprietary systems (containing the Cray Aries interconnect (Alverson, Froese, Kaplan, and Roweth 2012)) of the company behind CGE. It offers APIs for Java, Python and Spark, having a number of pre-built graph algorithms. It is not open-source and its back-end relies on internal queries written in C++ to work with a global address space using multiple processes on multiple compute nodes to share data and synchronize operations. This product being both proprietary and reliant on custom hardware has the consequence of not being so widespread. However, its results and special-purpose architecture make it a competitive platform which harnessed innovation in design as a graph database.

- Ontotext GraphDB (Ontotext 2020) is a graph database focused on RDF data and offering ACID transaction properties. It comes in three editions: free which is used for smaller projects and for testing and is only able to execute at most two concurrent queries; standard which can load and query statements at scale; enterprise edition which offers horizontal scalability and other features. It supports SPARQL and offers a Java programming API. We did not find its source code available.

The following support at least both the property graph model and/or RDF explicitly plus other data models (e.g., at least any of: document collections, relational model, object model):

- Amazon Neptune (Bebee, Choi, Gupta, Gutmans, Khandelwal, Kiran, Mallidi, McGaughy, Personick, Rajan, et al. 2018) is a managed proprietary service (freeing the user from having to focus on management tasks, provisioning, patching, etc.) that is ACID-compliant and focused on highly-connected datasets and among its use cases are recommendation engines, fraud detection and drug discovery, among others. Its implementation language and internal graph representation have not been disclosed and it supports both the property graph and RDF models, offering Gremlin and SPARQL to query them. As a full-fledged commercial product, it also has many features related to backup, replicas, security and management tasks, using product features such as Amazon's S3, EC2 and CloudWatch to offer scalability. Usage samples available online (Amazon 2020).

- AnzoGraph DB (Cambridge Semantics 2020) (previously SPARQLVerse) is a proprietary database built to enable RDF with SPARQL and the property graph with Cypher queries to analyse big graphs (trillions of relationships) and it has Java and C++ APIs to create functions, aggregates and services. It supports ACID transactions and also supports an RDF+ inference engine following W3C standards and uses compressed in-memory and on disk storage of data. This database is described as beyond a transaction-oriented database and as a Graph Online Analytics Processing (GOLAP) database, enabling interactive view, analysis and update of graph data, in a way similar to the interactive capabilities of the Neo4j database. It comes as a single-machine (and memory usage limitations) free edition and an enterprise edition which supports unlimited cluster size, both supporting commercial and non-commercial use. It supports third-party visualization tools. We did not find details on its internal data structures nor source code online.

- ArangoDB is an open-source (Thomas Schmidts 2020) multi-threaded database with support for graph storage (as well as key/value pairs and documents) that is available in both a proprietary license and the Apache License 2.0. It is written in JavaScript from the browser to the back-end and all data is stored as JSON documents. ArangoDB provides a storage engine for mostly in-memory operations and an alternative storage engine based on RocksDB, enabling datasets that are much bigger than RAM. It guarantees ACID transactions for multi-document and multi-collection queries in a single instance and for single-document operations in cluster mode. Replication and sharding are offered, allowing users to set up the database in a master-slave configuration or to spread bigger datasets across multiple servers. It exposes a Pregel-like API to express graph algorithms (implying access to the stored data in the database), has a custom SQL-like query language called AQL (ArangoDB Query Language) and includes a built-in graph explorer.

- IBM System G (IBM System G Team 2015) more than a proprietary graph database, is a complete suite of functionalities, able to support the property graph (with Gremlin) as well as RDF (though we did not find comments on RDF-specific query languages). It is comprised of proprietary components as well as open-source and comes with visual query capabilities, providing visual feedback into query building and result analysis to ease the debugging process. ACID transactions are supported and the graph is represented in its native store with a data structure similar to compressed sparse vectors, using offsets to delimit, for each graph element, the latest and earliest temporal information of the element.

- OrientDB (Tesoriero 2013) is a distributed (property model) graph database that supports TinkerPop and functions both as a graph database and NoSQL document database as well, with a Community Edition licensed (OrientDB LTD 2020) under Apache License 2.0 and a commercial edition. There are drivers supporting OrientDB at least in the following languages: Clojure, Go, Java, JavaScript, .NET, Node.js, PHP, Python, R, Ruby and Scala. It supports sharding (horizontal scaling), has ACID support and offers an adapted SQL for querying.

- Oracle Spatial and Graph (OSG) (Oracle 2020) is a long-standing commercial product which has spatial and graph capabilities, among which the property graph model using PGQL and the RDF model with SPARQL. It also supports a feature-rich studio with notebook interpreters, shell user-interface and graph visualization. There are different Java APIs, one for the Oracle Spatial and Graph Property Graph, another for TinkerPop Blueprints and Database Property Graph.

- Stardog (Stardog 2020) is a proprietary (with a limited-time free trial version and a paid Enterprise license) knowledge graph database with a graph model based on RDF and extensions to support the property graph model. It is horizontally-scalable, supports ACID operations, GraphQL, Gremlin and SPARQL for querying and introspection and may be programmed in Clojure, Groovy, Java, JavaScript, .NET and Spring. For exploration, it features Stardog Studio.

- Virtuoso (Erling 2012) is a multi-model database management system that supports relational as well as property graphs and has an open-source (OpenLink Software 2020) edition under the GPLv2 license as well as a proprietary enterprise edition. At least the following programming languages are supported: C/C++, C#, Java, JavaScript, .NET, PHP, Python, Ruby and Visual Basic. It supports horizontal scaling, has functionalities for interactive data exploration and supports SPARQL.

We lastly note the following databases that have been used to represent graphs, though not having an explicit description of supporting the property graph model or RDF:

- Azure Cosmos DB (Paz 2018) is a commercial database solution that is multi-model, globally-distributed, schema-agnostic, horizontally-scalable and fully supports ACID. It is classified as a NoSQL database, but the multi-model API is a relevant offering, for it can expose stored data for example as table rows (Cassandra), collections (MongoDB) and most importantly as graphs (Gremlin). It has connectors for Java, .NET, Python and Xamarin. It is also a fully-managed service with scalability, freeing developer resources from topics such as data centre deployments, software upgrades and other operations. An online repository of source code information is available (Corporation 2019).

- FaunaDB (Fauna 2020) is a distributed database platform targeting the modern cloud and container-centric environments. It has a custom Fauna Query Language (FQL) which operates on schema types such as documents, collections, indices, sets and databases. This language can be accessed through drivers in languages such as Android, C#, Go, Java, JavaScript, Python, Ruby, Scala and Swift. FaunaDB supports concurrency, ACID transactions and offers a RESTful HTTP API.

- Google Cayley is an open-source (Google 2017) database behind Google's Knowledge Graph, having been tested at least since 2014 (and it is the spiritual successor to graphd (Meyer, Degener, Giannandrea, and Michener 2010)). It is a community-driven database written in Go, including a REPL, a RESTful API, a query editor and visualizer. It supports Gizmo (query language inspired by Gremlin) and GraphQL. Cayley supports multiple storage back-end such as LevelDB, Bolt, PostgreSQL, MongoDB (distributed stores) and also an ephemeral in-memory storage. The ability to support ACID transactions is delegated to the underlying storage back-end. Cayley being distributed depends on the underlying storage being distributed as well. Also in active development as of 2019.

- HyperGraphDB (Iordanov 2010) is as an open-source general purpose data storage mechanism. It is used to store hypergraphs (a graph generalization where an edge can join any number of vertices). The low-level storage is based on BerkeleyDB and is implemented in Java. Despite the description on the website mentioning distribution, the support for either distributed sharding or distributed replication is not supported in its current implementation (Iordanov 2020) and we did not find mentions of ACID transaction support.

- Objectivity/DB (Objectivity 2016) is the database technology powering the massively scalable graph software platform ThingSpan (formerly known as InfiniteGraph). It is a fully-distributed database (able to scale horizontally) offering APIs in C++, C#, Java and Python. Objectivity/DB is described as a distributed object database, supporting many data models (among which highly complex and inter-related data). We did not find any information about it supporting graph-specific query languages, and licensing is defined on a case-by-case basis.

Other notable mentions include the OQGRAPH (MariaDB 2016) graph storage engine of MariaDB, developed to handle hierarchies and vertices with many connections and intended for retrieving hierarchical information such as graphs, routes and social relationships in SQL. We note the following resources for further deepening of the aspects involved in the evolution of the study of graph databases (Jin, Bhowmick, Xiao, Cheng, and Choi 2010; Buerli and Obispo 2012; Shimpi and Chaudhari 2012; Robinson, Webber, and Eifrem 2013; Kolomičenko, Svoboda, and Mlýnková 2013; Vaikuntam and Perumal 2014; De Virgilio, Maccioni, and Torlone 2014; Henderson 2014; Robinson, Webber, and Eifrem 2015), covering aspects such as the internal graph representations, experimental comparisons, principles for querying and extracting information from the graph and designing graph databases to make use of distributed infrastructures.

3.12 Analysis and Discussion

This survey initiative explores different aspects of the graph processing landscape and highlights vectors of research. We cover dimensions that enable the classification of graph processing systems according to the mutability of data (dynamism (Besta, Fischer, Kalavri, Kapralov, and Hoefler 2019) and its modalities), the nature of the tasks (workloads where the focus may be efficient storage (Corporation 1999) or swift computation (Chen, Yan, Zhu, Han, and Yu 2008) over transient data) and how the data is associated to different computing agents (e.g., distributed via partitioning (Soudani, Fatemi, and Nematbakhsh 2019) to threads in a CPU, CPUs in a machine, machines in a cluster). Each of these dimensions constitutes a different branch of the study of graph processing, and herein we group their recent literature surveys and draw on their relationships.

On drawing a line between graph processing systems and those that also focus on the storage, the graph databases, we found most commercial graph solutions to fall on the category of graph database. Graph databases, along the last decade, have continued to refine their efficiency in executing traversals and global graph algorithms over the graph representation stored in the database. We consider that a novel approach to extracting value from graph-based data will include the use of graph-aware data compression techniques on scalable distributed systems, potentially breaking the abstraction that these systems establish between the high-level graph data representations and the lower-level data distribution and transmission. We observe that the architecture of systems targeting graphs depend on how generic the graph processing is desired to be. Generic dataflow processing systems offer abstractions over their basic computational primitives in order to represent and process graphs, but in exchange abdicate from fine-tuning and graph-aware optimizations.

As part of our exhaustive analysis of existing contributions of different domains in the state-of-the-art of graph processing and storage, we provide direct links to source code repositories such as GitHub whenever they were available. Should the reader wish to delve into the implementation of a given contribution, a link to the contribution's source code repository is to be found as part of the bibliography. We provide these so that other researchers and developers may look into them without need to engage in error-prone searches looking for up-to-date documentation and source-code.

This systematic analysis fosters some additional comments regarding data processing. Data is abundant, big and evolving, and paradigms such as edge computing and the evolution of the Internet-of-Things come together to reshape our relationship with data. With an increase in *smart* devices and computational capabilities becoming more ubiquitous for example in daily objects such as vehicles and smart homes, new graphs of data mapping interaction and purpose become available. This implies a continuous trend in the increasing size of data. At the same time, the dimension of dynamism (spread across the types we enumerate in this document) gains renewed importance as we move to a faster and ever-connected world. With the advent of 5G technologies and the alternative possibilities of *space internet* (among the private initiatives we count SpaceX's Starlink, Jeff Bezos' Blue Origin and the late Steve Jobs' vision for an always-connected smartphone) becoming a closer reality, the temporal aspect will become even more granular.

One would not be wrong to speculate that we will have more devices which will generate data more frequently. In such a world, the graph processing dimensions we enumerate in this document will play a relevant role in building systems to handle these changing scenarios.

3.12.1 Challenges in the Field

Our survey initiative enumerated many different systems, architectures, hardware specializations and topics of graph processing. This effort was undertaken across the development of our research contributions, gaining from each work's respective related state-of-the-art and providing a more complete background on their challenges.

The following chapters introduce certain challenges and our contributions in their scope. Among them we explored:

- Approximate graph processing techniques in the scope of graphs receiving updates from a stream. We focus on a specific summarization technique that we implemented over `Apache Flink` as our work VEILGRAPH in Chapter 4. With a graph summarization model and proposed parameters, we performed experiments coordinating the relation between performance and result accuracy for the vertex centrality algorithm PageRank. The topic of approximation as a tool to increase speedups has remained relevant across the years, with approaches in the literature ranging from incurring approximation by trimming certain computations, to fine-grained estimates of topological graph change and subsequent algorithm value updates. Some of the systems detailed in the present chapter (such as `KickStarter`) present these approaches.

- An application of a graph-based community detection approach as part of a technique to perform leader election for service placement in community networks, presented as GELLY-SCHEDULING in Chapter 5. There exist approaches in the literature to the problem of leader election which consist for example of a series of nested loops to search the solution space for the most efficient combination of parameters. This problem remains relevant today, not only for community networks but also data centers and other forms of network. In the case of cloud community micro-clouds, we aimed to capture the relevance of network topology as well as the nodes' properties.

- Compressed graph formats exist and enable their analysis on off-the-shelf harware to some degree. However, their scope is limited in the sense that they typically manage a static representation of the graph (such as `WebGraph`), thus not supporting the many use-cases of graph processing with respect to updating the graphs in some way. With compact (smart) graph data structures, changes to the graph are possible without having to convert the compressed representation into an uncompressed one (which would be prohibitive for many datasets unless using extremely powerful hardware). We explore one particular compact representation, the k^2-tree, for which we present an implementation and benchmark against other implementations in Chapter 6.

4 VeilGraph: Streaming Graph Approximations

Herein we present our submission (Coimbra, Esteves, Francisco, and Veiga 2021) VEILGRAPH, an API implementing an innovative model for approximate graph processing. This work highlights the contributions and achievements of VEILGRAPH in the context of the PageRank *power method* algorithm. Our experiments show VEILGRAPH can reduce computational time while achieving result quality above 95% when compared to results of the traditional version of PageRank without any summarization or approximation techniques.

In a context of maintaining an evolving graph (integrating updates from a stream) and processing it, we research the trade-offs between result accuracy and the speedup of approximate computation techniques. The relationships between the frequency of graph algorithm execution, the update rate and the type of update play an important role in applying these techniques. We showcase an innovative model for approximate graph processing implemented in `Apache Flink`. We analyse our model and evaluate it with the case study of the PageRank algorithm (Page, Brin, Motwani, and Winograd 1999), the most famous measure of vertex centrality used to rank websites in search engine results. Our experiments show that VEILGRAPH can improve performance up to 10X speedups, while achieving result quality above 95% when compared to results of the traditional version of PageRank without any summarization or approximation techniques.

4.1 Introduction

We introduce VEILGRAPH, a novel execution model that enables approximate computations on general directed graph applications. Our model uses a summarized graph representation which includes only the vertices most relevant to computation using a set of heuristics over the topological changes in the graph. With this abstraction, we build a representative graph summarization that solely comprises the subset of vertices estimated as yielding a relevant impact to the accuracy of a given graph algorithm. This way, VEILGRAPH is capable of delivering lower latencies in a resource-efficient manner, while maintaining query result accuracy within acceptable limits. We integrated VEILGRAPH with `Apache Flink` (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015; Katsifodimos and Schelter 2016), a modern distributed dataflow processing framework. Experimental results indicate that our approximate computing model can achieve half the latency of the base (exact) computing model, while not degrading result accuracy by more than 5% (this was observed by comparing the complete and the approximate models with the same number of workers in the cluster). Furthermore, our approximate (summarized) computation model is scalable, achieving speedups in the range of 10x-15x while using only 16 workers in our Google Cloud Dataproc cluster experiments (we evaluated with 1, 2, 4, 8 and 16 worker counts in the cluster).

We focus on a vertex-centric implementation of PageRank (Langville and Meyer 2011), where for each iteration, each vertex u sends its value (divided by its outgoing degree) through each of its outgoing edges. A vertex v defines its score as the sum of values received from its incoming edges, multiplied by a constant factor β and then summed with a constant value $(1-\beta)$ with $0 \leq \beta \leq 1$. PageRank, based on the random surfer model, uses β as a dampening factor. For our work, this means that whether one considers one-time offline processing or online processing over a stream of graph updates, the underlying computation of PageRank is an approximate numerical version well known in the literature. This distinction is important, for when we state VEILGRAPH enables approximate computing, we are considering a potential for applicability to a scope of graph algorithms, such as algorithms for computing centrality (Katz 1953; Freeman 1977; Newman 2010), heat kernel (Vassilevich 2003) and optimization algorithms for finding communities (Boldi, Rosa, Santini, and Vigna 2011; Chung and Simpson 2015). Whether the specific graph algorithm itself incurs numerical approximations (such as the *power method*) or not, that is orthogonal to our model and may only enable its benefits further.

This chapter is organized as follows. Section 4.2 describes our summarization model and how it is built. An overview of the VEILGRAPH architecture is provided in Section 4.3. In Section 4.4 we present the experimental evaluation, followed by an analysis of improvements. Section 4.5 notes related systems and techniques. We summarize our contribution and future research in Section 4.6.

4.2 Model: Big Vertex

When a window of graph updates (a batch of edge additions and deletions) is incorporated into the graph, vertices change in different ways. The importance of a vertex with five thousand (5000) neighbours will not change much if five (5) new vertices connect to it. But if the vertex only had five (5) neighbours, it now has ten (10), it is a 100% increase and so the magnitude of its individual topological change is greater.

Our model considers a set K of *hot* vertices upon which computation of an algorithm (e.g. PageRank) will be performed. The model resorts to a synthesis, unifying techniques such as defining and determining a confidence threshold for error in the calculation (Agarwal, Milner, Kleiner, Talwalkar, Jordan, Madden, Mozafari, and Stoica 2014; Goiri, Bianchini, Nagarakatte, and Nguyen 2015), graph sampling (Babcock, Datar, and Motwani 2002; Hu and Lau 2013; Ahmed, Duffield, Willke, and Rossi 2017) and sketching (Ahn, Guha, and McGregor 2012). The aim of this set is to reduce the number of processed vertices as close as possible to $O(|K|)$. For a given graph $G = (V, E)$, we build set K such that the vertices outside K assume the values they had (ranks) on the previous computation and are not updated in the current one. New vertices which were just obtained from the update stream are immediately added to K as they have no algorithm-specific (e.g. PageRank) value yet.

Performing updates to only a subset K of the vertices implies that less data is propagated across the graph. In this model there is an aggregating vertex $\mathcal{B}$. We refer to $\mathcal{B}$ as the *big vertex* – a single vertex representing all the vertices not contained in K (in this model, the values are not updated for vertices represented by $\mathcal{B}$). For the original graph $G = (V, E)$, upon the arrival of edge additions/deletions, we build vertex set K and then define a summary graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = K \cup \{\mathcal{B}\}$. We define $\mathcal{E} = E_K \cup E_{\mathcal{B}}$, where $E_K = \{(u, v) \in E : u, v \in K\}$, which is the set of edges with both source and target vertices contained in K and $E_{\mathcal{B}} = \{(w, z) \in E : w \notin K, z \in K\}$ as the set of edges with sources contained *inside* $\mathcal{B}$ and target in K.

Conceptually, this consists in replacing all vertices of G which are not in K by a single big vertex $\mathcal{B}$ and representing the edges whose targets are *hot vertices* and whose sources are now in $\mathcal{B}$. The summary graph $\mathcal{G}$ does not contain vertices outside of K (again, those are represented by $\mathcal{B}$). By definition, $\mathcal{B}$ represents all vertices whose impact is not expected to change significantly. The contribution of each vertex $v \notin K$ (and therefore represented by $\mathcal{B}$) is constant between iterations (e.g., random walks), so it can be registered initially before updating algorithm values (e.g., rankings) and used afterwards during algorithm execution. As a consequence, the summary graph $\mathcal{G}$ does not contain edges targeting vertices represented by $\mathcal{B}$. However, their existence must be recorded: even if the edges coming out of K and into $\mathcal{B}$ are irrelevant for the computation, they still matter for the vertex degree, which influences the emitted scores of the vertices in K. Despite the fact those edges targeting $\mathcal{B}$ are being discarded when building the summary graph $\mathcal{G}$, the summarized computation must occur as if vertex degrees remained the same. To ensure correctness, for each edge $(u, v) \in E_K$, we store and represent the weight of the edge as $w(u, v) = 1/d_{out}(u)$ with $d_{out}(u)$ as the out-degree of u before discarding the outgoing edges of u targeting vertices in $\mathcal{B}$.

It is also necessary to record the contribution of all the vertices fused together in $\mathcal{B}$. For each edge (u, v) whose source u is inside $\mathcal{B}$ and whose target v is in K, we store the contribution that would originally be sent from u as $w(u, v) = u_s/d_{out}(u)$ where u_s is the stored value of u (resulting from the computed graph algorithm) and the out-degree of u is defined as $d_{out}(u)$. The contribution of $\mathcal{B}$ as a single vertex in $\mathcal{G}$ is then represented as $\mathcal{B}_s$ and defined as:

$$\mathcal{B}_s = \sum_u w(u, v), (u, v) \in E_{\mathcal{B}} \tag{4.1}$$

The fusion of vertices into $\mathcal{B}$ is performed while preserving the global influence that vertices placed inside $\mathcal{B}$ have on vertices in K. Our model intuition is that vertices receiving more updates have a greater probability of having their measured impact change in between execution points. Their neighbouring vertices are also likely to incur changes, but as we consider vertices further and further away from K, contributions are likely to remain the same (Chien, Dwork, Kumar, Simon, and Sivakumar 2003; Babcock, Datar, Motwani, et al. 2003).

To build the hot vertex set K after integrating a window of updates, we use three parameters whose purpose we describe below:

1. **Parameter: update ratio threshold *r*.** This parameter defines the minimal amount of change in vertex degree in order for it to be included in K. A vertex whose in-degree changes r% or more is included in K immediately (e.g., a vertex that changed in-degree from one to two changed by 50%).

   We adopt the notation where the set of neighbours of vertex u in a directed graph at measurement instance t is written as $N_t\left(u\right) = \{v \in V : \left(u, v\right) \in E_t\}$. We further write the degree of vertex u in measurement instance t as $d_t\left(u\right) = \left|N_t\left(u\right)\right|$. The function $d\left(u, v\right)$ represents the length (number of hops) of the minimum path between vertices u and v and $d_t\left(u, v\right)$ represents the same concept at measurement instance t. It is not required to maintain shortest paths between vertices (that would be a whole different problem (Kalavri, Simas, and Logothetis 2016)). This model is based on a vertex-centric breadth-first neighbourhood expansion. Let us define as K_r the set of vertices which satisfy parameter r, where $d_t\left(u\right)$ is the degree of vertex u, t represents the current measurement point and $t-1$ is the previous measurement point:

   $$K_r = \{u : \left|\frac{d_t\left(u\right)}{d_{t-1}\left(u\right)} - 1\right| > r\} \tag{4.2}$$

   New vertices are always included in K_r. The subtraction in the formula registers the degree change ratio with respect to the previous value $d_{t-1(u)}$. This definition allows us to mathematically express conditions such as *keeping all vertices whose degree changed at least 20%*.

2. **Parameter: neighbourhood diameter *n*.** Let $K_{r,n}$ be the set of vertices obtained from applying parameter n to K_r. We expand around the n-hop neighbourhoods of the vertices added to K_r in step 1. Parameter n aims to capture the locality in graph updates: those vertices neighbouring the ones beyond the threshold, and as such still likely to suffer relevant modifications when the *hot vertices'* values are updated (attenuating as distance increases). On measurement point t, for each vertex $u \in K_r$, we will expand a neighbourhood of diameter n (number of hops), starting from u and including every additional vertex $v \in V_t \setminus K_r$ found in the neighbourhood diameter expansion. The expansion is then defined as:

   $$K_{r,n} = K_r \cup \{v : d_t\left(u, v\right) \leq n, u \in K_r, v \in V_t \setminus K_r\} \tag{4.3}$$

   V_t is the set of vertices of the graph at measurement point t. Parameter $n = 0$ may be set to promote performance, while a greater value of n is expected to focus on accuracy at the expense of performance.

3. **Parameter: result-specific neighbourhood extension Δ.** Let $K_{r,n,\Delta}$ be the set of vertices obtained from expanding set $K_{r,n}$ using parameter Δ. The final set K of

hot vertices is defined as $K = K_{r,n,\Delta}$. Between computations, the values change. A vertex u whose algorithm value (rank) changed at least Δ between computations may have a greater influence on its neighbours until the value of vertex u becomes negligible after a given number of hops. We select these influenced neighbours based on the formula:

$$K = K_{r,n,\Delta} = K_{r,n} \cup \{v : d(u,v) \leq f_{\Delta}(v), u \in K_{r,n}, v \in V_t \setminus K_{r,n}\} \tag{4.4}$$

where $f_{\Delta}(v)$ is the Δ-expansion function:

$$f_{\Delta}(v) = \frac{1}{\log \overline{d}} \log \left(\frac{\overline{d}\, v_s}{\Delta\, d_t(v)} \right) \tag{4.5}$$

The symbols of Equation 4.5 are as follows: $\overline{d}$ is the average degree in the graph, v_s is the existing result on vertex v and $d(v)$ is the out-degree of v. The intuition underlying this parameter is the following: for a vertex u that will be expanded via Δ from step 3, its impact will diminish as we hop further away from it (from u to immediate neighbours, then to its neighbours' neighbours and so on). The impact of a vertex u on its i-hop neighbourhood will dilute as we further hop away from u as $i \to \infty$. Consider we perform a number i of hops away from u until we reach a given vertex v. Δ is used with this formula to assign a value to the number of hops. With this, we account for vertices which change in an impacting way so that their neighbourhoods are included in the computation as well.

The vertices outside K are aggregated into a *big vertex* $\mathcal{B}$. To ensure correctness, for each edge $(u,v) \in E_K$, we store and represent the weight of the edge as $w(u,v) = 1/d(u)$ with $d(u)$ as the out-degree of u before discarding the outgoing edges of u targeting vertices in $\mathcal{B}$. This is so the correct degree of the vertices in K is used in the computation of scores, even though their outgoing edges (with targets in $\mathcal{B}$) are not used in the summarization. Thus, the vertices represented in $\mathcal{B}$ are assumed to not change their value during the computation (which may lead to error accumulation – we analyse and deal with it in Section 4.4). What is captured and preserved is their global contribution to the vertices in K. We then have a summary graph written as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = K \cup \{\mathcal{B}\}$.

Equation 4.5 is based on the diminishing returns incurred by continuously expanding the neighbourhood when building K_{Δ}. When the result of a vertex v (e.g., its PageRank) is propagated to a 1st-order neighbour u of v, the score gets diluted by the amount of out-edges of v. So if v has a result of v_s, each of its immediate neighbouring vertices will receive $v_s/d_t(v)$ at measurement point t. If we were to further propagate to 2nd-hop neighbours of v, we would expand from vertex u into a farther vertex w. The value received by w would be $v_s/(d_t(v)d_t(u))$. Expanding into an ith-hop neighbour would accumulate an out-degree division for each vertex expanded. The impact of v's result

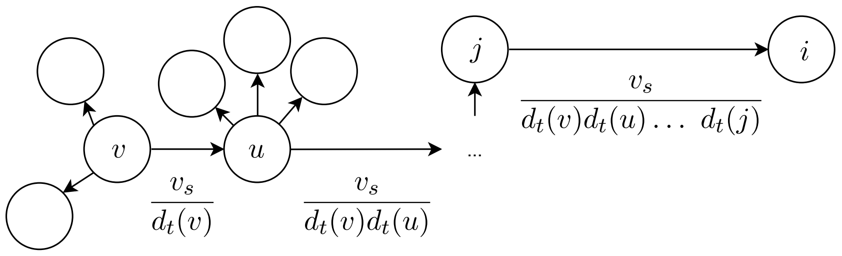

v
u
j
i
...
$\frac{v_s}{d_t(v)}$
$\frac{v_s}{d_t(v)d_t(u)}$
$\frac{v_s}{d_t(v)d_t(u)\dots\ d_t(j)}$

Figure 4.1: The contribution of vertex v diminishes as we expand further away from it (Equation 4.7). Parameter Δ dictates how far to expand around vertices until the accumulated fraction drops below Δ. v_s is the score of vertex v; $d_t(v)$ is the out-degree of vertex v at measurement point t.

on a given i-th-hop vertex i would be defined as (x is the last expanded vertex):

$$\frac{v_s}{d_t(v)d_t(u)\dots\ d_t(x)} \tag{4.6}$$

To avoid fetching the out-degree of each vertex for each expansion (which would incur additional overhead in a dataflow processing system such as `Flink`), we approximate the degree of any vertex beyond v with $\overline{d}$, which is the average degree of the accumulated vertices with respect to the stream. We then have the approximation:

$$\frac{v_s}{d_t(v)d_t(u)\dots\ d_t(x)} \approx \frac{v_s}{d_t(v)(\overline{d})^{i-1}} \tag{4.7}$$

To ensure that we only continue expanding until we drop below a given Δ threshold of result impact (until the ith-hop vertex), we set it as a target of Equation 4.7:

$$\begin{aligned} \frac{v_s}{d_t(v)(\overline{d})^{i-1}} &= \frac{\overline{d}v_s}{d_t(v)(\overline{d})^{i}} = \Delta \\ \frac{\overline{d}v_s}{\Delta\ d_t(v)} &= (\overline{d})^i \\ \log_{\overline{d}}\left(\frac{\overline{d}v_s}{\Delta\ d_t(v)}\right) &= i \\ \frac{1}{\log \overline{d}} \log\left(\frac{\overline{d}v_s}{\Delta\ d_t(v)}\right) &= f_\Delta(v) = i \end{aligned}$$

Figure 4.1 provides a visual example of the dilution of the score of vertex v due to the degree of the vertex at the end of each hop (u, then intermediate vertices represented as '$\dots$', followed by j until i is reached).

We then have a set of *hot vertices* $K = K_r \cup K_n \cup K_\Delta$ which is used as part of a graph summary model (deriving from techniques in iterative aggregation (Langville and Meyer 2004)), written as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

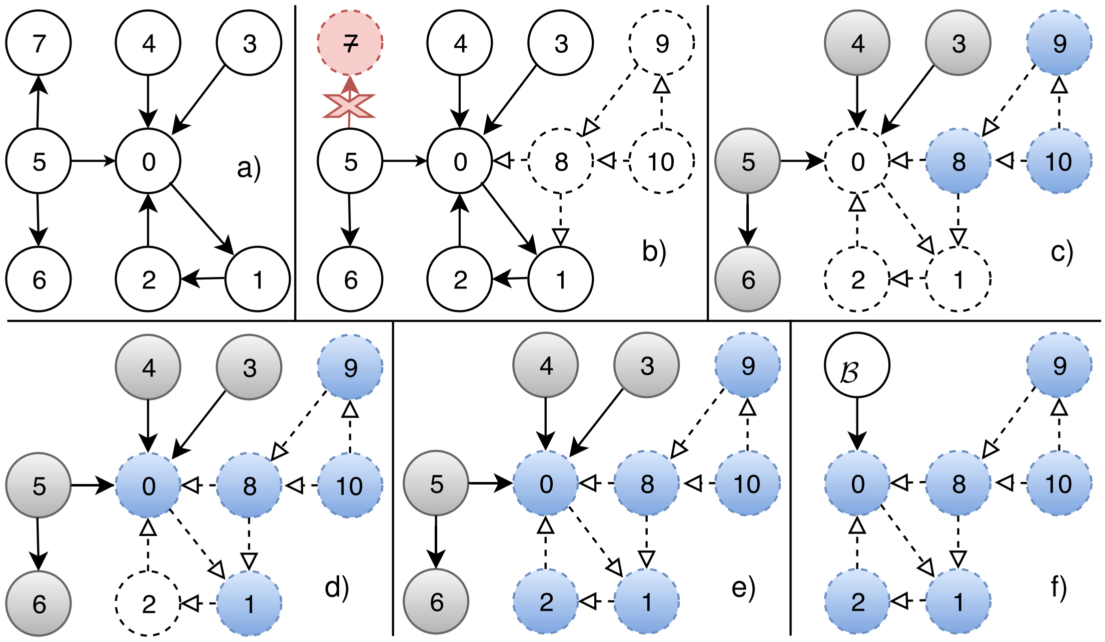


Figure 4.2: Building the big vertex, step-by-step.

An example of applying the updates and building the K set and the big vertex $\mathcal{B}$ is shown in Figure 4.2 with $r = 0.20, n = 1, \Delta = 0.10$: a) is the initial graph; b) shows five new edges (dashed) and one edge deletion (red cross); c) new vertices are automatically included in K for computation (blue); d) vertices 0 and 1 are also included in K (blue) because their in-degree changed at least 20% ($r = 0.20$); e) the neighbourhood diameter expansion of size $n = 1$ around current vertices in K includes vertex 2; f) vertices (3, 4, 5, 6) outside K are collapsed into the *big vertex* $\mathcal{B}$. Impact of Δ not depicted.

4.3 Architecture

The general distributed architecture and workflow of VeilGraph is illustrated in Figure 4.3. The architecture was designed to take into account the relevant stages in the graph processing as updates from a stream arrive. The VeilGraph module is primarily responsible for continuously monitoring one or more streams of data and tracking the updates to be applied to the graph. We use the term *query* to designate a request for up-to-date graph algorithm results. The process started by a query follows the following steps:

1. The VeilGraph module receives a query request.

2. The VeilGraph module submits a `Flink` job to integrate in the graph new information (edge additions/deletions) that arrived from the stream of updates.

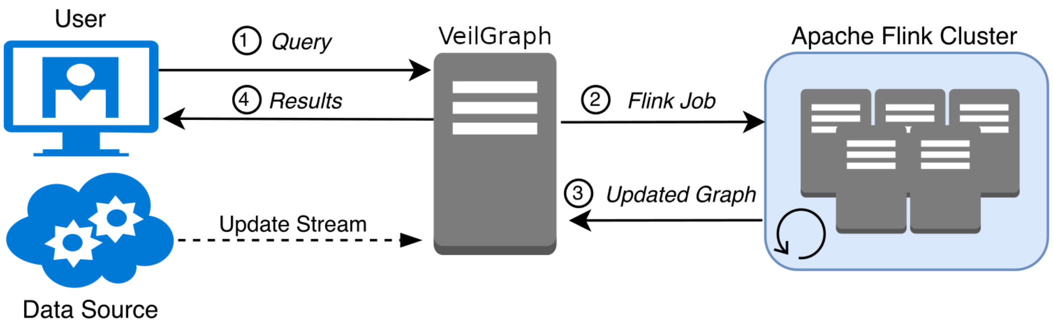


Figure 4.3: Diagram of VEILGRAPH workflow on `Flink`.

3. `Flink` executes the jobs, thus updating the graph by: integrating updates; building a summary graph representation if approximate processing is chosen; executing either the approximate version of the algorithm over the summary graph, or the complete version over the updated graph.

4. VEILGRAPH then returns the update graph data to as a response to the request.

We designed VEILGRAPH to control the flow of data and submit jobs to a `Flink` cluster when appropriate. The cluster may be running in locally-owned hardware, on any cloud provider's offerings (we evaluated with Google Cloud Dataproc) or even with its components within the same large-memory machine. In our model, the stream of updates consists of reading from a graph dataset file in adjacency list format. The update windows consist of batches of edges read from the input file, with a fixed window size set a priori. The windows taken from the stream are non-overlapping and contiguous. We further detail this in Section 4.4. From our analysis of related works, we consider the term *incremental* with respect to processing as meaning that new algorithm results may be computed as a function of previous results and a change of the underlying data. With *streaming*, we refer to the fact information is coming as continuous flow of data from any given source (even though we test VEILGRAPH in a controlled scenario with fixed-size streams).

Among the many distributed systems detailed in Chapter 3, we found at times the lack of a strong development community to be the strongest motivation to disqualify candidates to implement VEILGRAPH. While it is true that there exist systems which achieved high levels of performance, the pitfalls of exploring undocumented code bases with potentially prohibitive and unforeseen flaws are risks that overwhelm the hypothetical use one may make of them. With these criteria, we established `Flink` and `Spark` as the most desirable candidates to implement VEILGRAPH.

Ultimately, two factors were crucial that led us to choosing `Flink` over `Spark`: we found `Flink`'s `Gelly` graph processing library to be in more active development than `Spark`'s, and `Flink`'s community of developers was also more active. The ability to quickly iterate through problems with knowledge of those who directly develop the tools is priceless, and impractical with less active projects. Dataflow programming systems offer the advantage of the declarative specification of logic which is decoupled from the executing infrastructure. As we wished to offer VEILGRAPH as big of an audience as possible, systems such as `Flink` allow less experienced developers to compute over different infrastructures with abstractions.

VEILGRAPH was designed to allow programmers to define fine-grained logic for the approximate computation when necessary. This is achieved through user-defined functions defined in a specific base `Java` class `GraphStreamHandler`. The API of VEILGRAPH uses them to define the execution logic that will guide the processing strategy. They are key points in execution where important decisions should take place (e.g., how to apply updates, how to perform monitoring tasks). To employ our module, the user can express the algorithm using `Flink` dataflow programming primitives.

Additional behaviour control is possible by customizing the model by implementing user-defined functions (left as `abstract` methods of the class implementing the architecture logic). Overall, this approach has the advantage of abstracting away the API's complexity, while still empowering users who wish to create fine-tuned policies. VEILGRAPH's architecture creates a separation between the graph model, the way the graph processing is expressed (e.g. such as vertex-centric) and the function logic to apply on vertices. We employ `Flink's` mechanism for efficient dataflow iterations (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015) with intermediate result usage, expressing computations over its `Gelly` graph library.[1]

VEILGRAPH will monitor the data stream and track the changes made to the graph. When a graph algorithm is to be run (henceforth we call this occurrence a *query*), VEILGRAPH will execute the request by submitting a job to a `Flink` cluster. In our experiments, we trigger the incorporation of updates into the graph whenever a client query arrives.[2] To simplify implementation design, the client queries are also sent in the stream of graph updates. Conceptually, these are the major elements involved in the functioning of VEILGRAPH and are compatible with most graph processing frameworks:

- **Initial graph *G***. The original graph upon which updates and queries will be performed.
- **Stream of updates *S***. Our model of updates could be the removal e_- or addition e_+ of edges and the same for vertices (v_-, v_+). We make as little assumptions as possible regarding S: the data supplied needs not respect any defined order. In our experiments we used both edge additions and removals, with vertices being added or removed as part of the edges they belong to.
- **Result *R***. Information produced by the system as an answer to the queries received in S (e.g. vertex rankings). It is reused in the following computation when the next

[1] `https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/iterations.html#delta-iterate-operator`

[2] While outside the scope of this work, a live scenario would have a more elaborate ingestion scheme, possibly using dedicated ingestion nodes like in `KineoGraph` (Cheng, Hong, Kyrola, Miao, Weng, Wu, Yang, Zhou, Zhao, and Chen 2012).

Algorithm 1 VEILGRAPH Execution Skeleton: graph G, stream S

```
1: OnStart(G, S)   /* Initializations. */
2: graphUpdates ← ∅
3: updateStatistics ← ∅
4: repeat
5:    msg ← TakeMessage(S)
6:    if msg is Add then RegisterAddEdge(msg, graphUpdates, updateStatistics)
7:    else if msg is Remove then RegisterRemoveEdge(msg, graphUpdates, updateStatistics)
8:    else if msg is Query then
9:       needToApplyUpdates? ← CheckUpdateState(graphUpdates, updateStatistics)
10:      if needToApplyUpdates? then
11:         G ← ApplyUpdates(graphUpdates, updateStatistics)
12:      end if
13:      strategy ← DefineQueryStrategy(msg, G, updates, updateStatistics)
14:      if strategy = Repeat-last-answer then
15:         newResults ← previousResults
16:      else if strategy = Compute-approximate then
17:         newResults ← ComputeApproximate(G, previousResults)
18:      else if strategy = Compute-exact then
19:         newResults ← ComputeExact(G)
20:      end if
21:      OutputResults(newResults)
22:      OnQueryResult(msg, G, newResults, jobStatistics)  /* Extrapolate and store job statistics. */
23:   end if
24: until stopped
25: OnStop( )  /* Tear-down procedure. */
```

window of updates and query are received.

Figure 4.4 illustrates different examples of update streams. In our stream scenario, these are non-overlapping counting windows. In the top stream, we see that each window of updates adds five edges and removes one. The greater the amount of elements in the stream window, the more entropy each update would add to the graph. The bottom-right shows a bigger window and the bottom-left shows the extreme case where the element count in the window is one. The smaller the window size, the more resources are consumed (by recalculating ranks completely more often), and the greater the computational benefit, increased speedup and reduced latency of applying our model, as there are practically no changes to the graph.

We present in Algorithm 1 these different UDFs and their coordination. The functions are as follows: For simple rules, these functions do not need to be programmed, as we supply the implementation with parameters for the simplest rules such as threshold comparisons, fixed values, intervals and change ratios.

1. ONSTART. A preparatory function for setting up resources such as files, database accesses or other initial tasks.

2. CHECKUPDATESTATE. Executed after a query q is received, but before graph updates are applied. Its purpose is to enable programmers to choose how and when

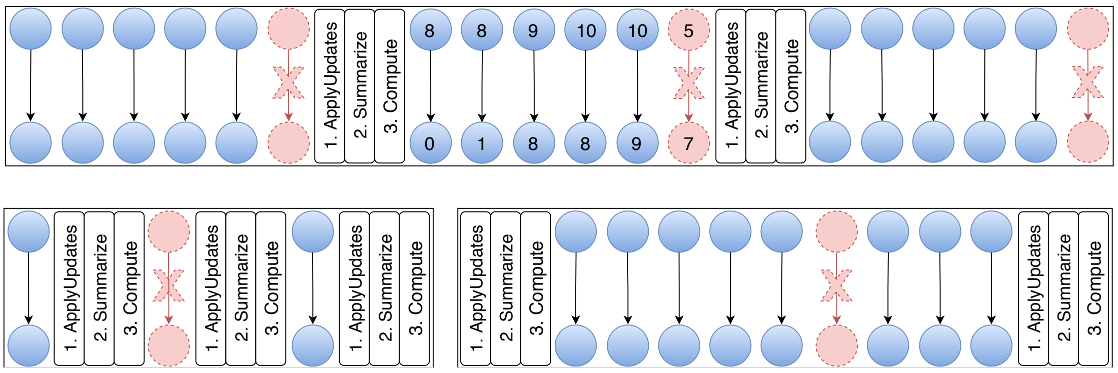


Figure 4.4: Example of different granularities edge addition/removal in streams.

to process the graph updates as a function of the magnitude of their impact. It exposes the sequence of graph operations which were pending since the last query and statistics such as the number of changed vertices and the total amount of vertices and edges in the graph. Its returned value on line 9 will dictate if updates should be integrated into the graph according to some criteria, with line 11 invoking the actual logic to update the graph with topological changes that arrived from the stream.

3. DEFINEQUERYSTRATEGY. Called every time a query q arrives. The query is served after any processing that may have taken place in BEFOREUPDATES. This function is defined in the API to return an action indicator dictating the query strategy to use. It could be done be any of: *a)* by returning the last calculated result; *b)* performing an approximation of the result and returning it; *c)* providing an exact answer after a complete recalculation of the result.[3] Line 13 is responsible for this call to define the strategy that will be used. The chosen strategy is respective to the specific graph algorithm in use. In our evaluation scenario, the algorithm whose results need to be updated is PageRank.

4. ONQUERYRESULT. Invoked after q's response has been processed. This UDF is aware of the action indicator returned by DEFINEQUERYSTRATEGY. It has access to the response's results, execution statistics (such as total execution time, physical space, network traffic, among others) and details specific to the approximation technique used.

5. ONSTOP. Symmetrical to ONSTART, it is responsible for (if necessary) proper resource clearing and post-processing.

It is relevant to note that

4.3.1 Implementation

VEILGRAPH was implemented on `Apache`, a framework built for distributed

[3]In our scenario we always used (*b*)) approximation

stream processing.[4] It has many different libraries, among which Gelly, its official graph processing library. It features algorithms such as PageRank, Single-Source Shortest Paths and Community Detection, among others. Overall, it empowers researchers and engineers to express graph algorithms in familiar ways such as the *gather-sum-apply* or the *vertex-centric* approach of Pregel (Malewicz, Austern, Bik, Dehnert, Horn, Leiser, and Czajkowski 2010), while providing a powerful abstraction with respect to the underlying scheme of distributed computation. We employ Flink's mechanism for efficient dataflow iterations (Kalavri, Ewen, Tzoumas, Vlassov, Markl, and Haridi 2014) with intermediate result usage. To employ our module, the user can express the algorithm using Flink dataflow programming primitives. They will be fed the updated graph and the processing infrastructure of Flink.

VEILGRAPH is implemented as a Java program which triggers Flink jobs to ingest graph updates, build summary graphs and execute algorithm updates as needed. It does not modify Flink's internals and aims to assess the scalability of the graph summarization technique as more worker nodes are present in the Flink cluster, while offering an API to define how update ingestion and approximations may be performed. The stream of updates is implemented through a Python program (streamer) which reads a graph input file and sends batches of updates to VEILGRAPH through a socket. The streamer interleaves the batches of updates with a specific instruction to trigger the processing in VEILGRAPH.

The source of VEILGRAPH is available online for the graph processing community.[5] We provide an API allowing programmers to implement their logic succinctly. VEILGRAPH was evaluated with the PageRank power method algorithm (Page, Brin, Motwani, and Winograd 1999). The PageRank logic is succinctly implemented as a function as follows:

```java
public static class PageRankFunction implements Function<MessageIterator<Double>,
    Double>, Serializable {

      private final Double dampening;

      public PageRankFunction(Double dampening) {
         this.dampening = dampening;
      }

      @Override
      public Double apply(final MessageIterator<Double> inMessages) {

         double rankSum = 0.0;
         for (double msg : inMessages) {
            rankSum += msg;
         }
         return (this.dampening * rankSum) + (1 - this.dampening);
      }
   }
```

This is then passed on to the underlying graph processing paradigm, as such:

[4] https://flink.apache.org/

[5] Access date: 2020-Feb-12: https://fenix.tecnico.ulisboa.pt/homepage/ist162460/veilgraph

```
PageRankFunction prf = new PageRankFunction
(dampeningFactor);

GraphAlgorithm<Long, Double, Double, DataSet<Tuple2<Long, Double>>> algo = new
    VertexCentricAlgorithm
(iterations, prf);

DataSet<Tuple2<Long, Double>> ranks = summaryGraph.run(algo);
```

While we focus our evaluation on PageRank, we note that other random walk based algorithms can be expressed easily. In our PageRank implementation, all vertices are initialized with the same value at the beginning.

4.4 Evaluation

Experimental setup. To realistically evaluate the effect that cluster execution has on speedup, we evaluated our datasets in Google Cloud Dataproc[6] clusters with different worker counts (2, 4, 8, 16). Each machine was created with the `custom-4-26368` flag of the `gcloud` shell utility and runs an image based on `Debian 4.9.168-1+deb9u5`. We used `Flink 1.9.1` configured to use a parallelism of one within each worker. We set the following configuration values for the `Flink` cluster components running on Google's `Dataproc`. This is similar to the actual `Flink` configuration file, though with a `Python` call to retrieve the number of `TaskManager` instances from `Dataproc` (it is set to be equal to the number of cluster workers). This logic is set in an adapted `Python` script from its official GitHub repository [7] with the values below:

```
jobmanager.rpc.address: ${master_hostname}

jobmanager.rpc.port: 6123

rest.address: ${master_hostname}

rest.port: 8081

rest.idleness-timeout: 6000000

jobmanager.heap.size: 4096m

taskmanager.heap.size: 8192m

taskmanager.numberOfTaskSlots: 1

parallelism.default: $(python2 -c "print ${num_taskmanagers} *
${flink_taskmanager_slots}")

taskmanager.network.numberOfBuffers: 2048

fs.hdfs.hadoopconf: /etc/hadoop/conf

env.log.dir: /usr/lib/flink/log

env.hadoop.conf.dir: /etc/hadoop/conf
```

[6]Access date: 2020-Feb-12: https://cloud.google.com/dataproc
[7]https://github.com/GoogleCloudDataproc/initialization-actions

```
env.ssh.opts: -oStrictHostKeyChecking=no

query.server.ports: 30000-35000

query.proxy.ports: 35001-40000

taskmanager.rpc.port: 45001-50000

taskmanager.data.port: 50001

blob.server.port: 55001-60000

blob.client.socket.timeout: 6000000

akka.transport.heartbeat.pause: 6000s

akka.tcp.timeout: 60000s

akka.ask.timeout: 60000s

web.timeout: 600000
```

As this contribution researches the potential of the summary graph model and its offering through an API, we did not implement and benchmark it against other graph processing systems such as `GraphTau`, `Kineograph` and others. In our scenario, PageRank is initially computed over the complete graph G and then we process a stream S of windows of incoming edge updates. For each window received from the stream we: *1)* integrate the edge updates into the graph; *2)* compute the summarized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ as described in Section 4.2 and execute PageRank over $\mathcal{G}$. Thus, we *process* a query when PageRank (summarized or complete) is executed after integrating a window of updates.

Recall that smaller windows (towards continuous complete vertex rank updates even if without relevant changes) would further amplify VEILGRAPH's benefits.

4.4.1 Stream S size and configuration

We do not use a window of updates with size of 1, as that would favor our model's summary graph $\mathcal{G}$ when compared against the repetitive complete execution over the whole graph G. To reduce variability, the stream S of edge updates was set up so the number Q of queries for each dataset and parameter combination is always fixed: fifty ($Q = 50$). Additionally, for each dataset, streams were generated by uniformly sampling from the edges in the original dataset file. A stream size of $|S| = 40000$ was used, implying $|S|/Q = 800$ edges are added before executing every query.

The number of stream queries Q and number of edge additions per query update were chosen to favour (non-approximate) `Flink` for comparability in a sensible scenario against summarized executions. For $|S| = 40000$, if we added 8 edges instead of 800 before executing each query, we would have $Q = 5000$. This is a much longer sequence of queries, where the graph barely changes between them, with VEILGRAPH having near-zero execution times in most, where `Flink` would be 100-fold slower processing the complete graph. To avoid that, we empirically chose the value of 800 edges before each query (resulting in $Q = 50$).

We test both edge additions and deletions. Every time we add edges, we remove an amount equal to 20% of the number of edges added. The edges to remove are chosen at random with equal probability. When additions and removals are applied before an execution, the removal only targets remaining edges which already existed in the original graph or that were added in an older update that preceded a previous execution: in any update window, we do not add and remove new edges, as that would have no effect.

For each dataset and stream S, each combination of parameters r, n, Δ is tested against a replay of the same stream. Essentially, each execution (representing a unique combination of parameters) will begin with a complete PageRank execution followed by $Q = 50$ summarized PageRank executions. This initial computation represents the real-world situation where the results have already been calculated for the whole graph previously. In such a situation, one is focused on the new incoming updates. For each dataset and stream S, we also execute a scenario which does not use the parameters: it starts likewise with a complete execution of PageRank, but the complete PageRank is always executed for all Q queries. This is required to obtain ground-truth results to measure accuracy and performance of the summary model.

Many datasets such as web graphs are usually provided in an incidence model (Boldi and Vigna 2004b; Boldi, Rosa, Santini, and Vigna 2011). In this model, the out-edges of a vertex are provided together sequentially. This may lead to an unrealistically favourable scenario, as it is a property that will not necessarily hold in online graphs and which may benefit performance measurements. To account for this fact, we previously shuffle the stream S. A single shuffle was performed a priori for all datasets so that the randomized stream is the same for different parameter r, n, Δ combinations that were tested. This increases the entropy and allows us to validate our model under fewer assumptions, and assess it to hold in general scenarios.

A relevant note must be made regarding terminology. We observed the term *"streaming"* graph in the literature to refer to the processing of graph-based data (e.g. a stream of edges) arriving into a system with the goal of maintaining graph statistics updated (such as triangle counts) without actually maintaining a working representation of the full graph. With *"dynamic graphs"*, we refer to graphs whose structure is maintained and updated - in VEILGRAPH, we integrate updates that arrive from a stream into a maintained graph representation (which is then used to build the summary graph and so on).

Our simplified stream scenario was designed in such a way as to minimize the amount of variability faced by VEILGRAPH, with the aim of first validating the accuracy and scalability of the summarization model. Deepening VEILGRAPH in the future, it would be relevant to evaluate with public graph datasets with timestamps such as the Stanford SNAP repository and the Open Graph Benchmark.

4.4.2 Datasets

The datasets' vertex and edge counts are shown in Table 4.1. We evaluate results over two types of graphs: web graphs and social networks. They were obtained from

Dataset	$\|V\|$	$\|E\|$	*#Pairs*	$\overline{d}$ $max(d_{in})$ $max(d_{out})$	*%Dang.*	C_1	l_G
`eu-2005`[1]	0.862M	19M	87.01% (± 0.789)	22.297 68 922 6 985	8.31%	752 725 (87.26%)	10.18 (± 0.037)
`eu-2015-host`[1]	11M	386M	57.60% (± 0.119)	34.350 174 433 398 600	21.60%	6.512M (57.82%)	5.83 (± 0.002)
`amazon-2008`[2]	0.735M	5.158M	84.40% (± 0.695)	7.015 1 076 10	12.04%	627 646 (85.36%)	12.06 (± 0.021)
`hollywood-2011`[2]	2.181M	229M	76.56% (± 0.757)	105.003 13 107 13 107	8.96%	1.917M (87.91%)	3.926 (± 0.005)

Table 4.1: Datasets: `Laboratory for Web Algorithmics`. Web graphs are indicated with [1] and social networks with [2].

the Laboratory for Web Algorithmics (Boldi and Vigna 2004b). These datasets were used to evaluate the model against different types of real-world networks.

4.4.3 Assessment Metrics

We measure the results of our approach in terms of the ability to delay computation in light of result accuracy; obtained execution speedup with increasing number of workers; reduction in number of processed edges. Accuracy in our case takes on special importance and requires additional attention to detail. The PageRank score itself is a measure of importance and we wish to compare rankings obtained on a summarized execution against rankings obtained on the non-summarized graph. As such, what is desired is a method to compare rankings.

Rank comparison can incur different pitfalls. If we order the list of PageRank results in decreasing order, only a set of top-vertices is relevant. After a given index in the ranking, the centrality of the vertices is so low that they are not worth considering for comparative purposes. But where to define the truncation? The decision to truncate at a specific position of the rank is arbitrary and leads to the list being incomplete. Furthermore, the contention between ranking positions is not constant. Competition is much more intense between the first and second-ranked vertices than between the two-hundredth and two-hundredth and first.

We employed Rank-Biased-Overlap (RBO) (Webber, Moffat, and Zobel 2010) as a meaningful evaluation metric (representing relative accuracy) developed to deal with these inherent issues of rank comparison. RBO has useful properties such as weighting higher ranks more heavily than lower ranks, which is a natural match for PageRank as a vertex centrality measure. It can also handle rankings of different lengths. This is in tune with the output of a centrality algorithm such as PageRank. The RBO value obtained from two rank lists is a scalar in the interval $[0, 1]$. It is zero if the lists are completely disjoint and one if they are completely equal. While more recent comparison metrics have been proposed (Moffat 2018), they go beyond the scope of what is required in our comparisons.

Performance-wise, we test values of r associated to different levels of sensitivity to vertex degree change (the higher the number, the less expected objects to process per query). With $n = 0$, we minimize the expansion around K and consider just the vertices that passed the degree change of $r\%$. For $n = 1$, we are taking a more conservative approach regarding result accuracy. An overall tendency to expect is that the higher the value of n is, the higher the RBO. The Δ values were chosen to evaluate individual different weight schemes applied to vertex score changes. The relation between parameters r and n has a greater impact in performance and accuracy than the relation of any of these parameters with Δ. We tested with two sets of parameter combinations:

- RBO-oriented $(r = 0.05, n = 2, \Delta = 1.0)$, $(r = 0.05, n = 2, \Delta = 0.5)$, $(r = 0.05, n = 6, \Delta = 1.0)$, $(r = 0.05, n = 6, \Delta = 0.5)$. This has a very low threshold of sensitivity to the ratio of vertex degree change $(r = 0.05)$.
- Performance-oriented $(r = 0.20, n = 0, \Delta = 0.5)$, $(r = 0.20, n = 0, \Delta = 1.0)$, $(r = 0.20, n = 1, \Delta = 0.5)$, $(r = 0.20, n = 1, \Delta = 1.0)$, $(r = 0.20, n = 4, \Delta = 1.0)$. With $r = 0.20$, the goal is to be less sensitive pertaining degree change ratio.

For both of these combinations, we test with low and high values of n to examine how expanding the neighbourhood of vertices complements the initial degree change ratio filter. Using a higher number of ranks for the RBO evaluation favours a comparison of calculated ranks which has greater resolution, as more vertices are being compared. In our evaluation, the RBO of each execution is calculated using 10% of the complete graph's vertex count as the number of top ranks to compare. We address the aforementioned issue of truncation by making the number of truncated ranks specific to each dataset by defining it as a percentage of the graph's vertices. Considering the nature of the rankings, we focus on comparing the top 10% of vertex ranks of the complete and summarized execution scenarios using RBO, as it is in this top that the most relevant ones are concentrated. Furthermore, every 10 executions, we calculate RBO using all of the vertices of the graph to periodically ensure that no artefacts are masked in the lower rank values.

4.4.4 Results

Parameters (r, n, Δ) producing the best accuracy result are not necessarily the same ones producing the best speedups. **The horizontal axis represents the same (except for the candle bar and regular bar plots) for all plots**: it is the sequence of queries from

1 up to $Q = 50$. Due to how the dynamics of parameter combinations and the structure of the data sets behave, some parameter combinations produced extremely similar values, leading to almost overlapping plots. One needs to take into account this is a challenging assessment context for VEILGRAPH. In fact, between each consecutive pair of the 50 queries (i.e., on every of the 800 edge/vertex updates we are ingesting between them), if the user prompted a query execution, VEILGRAPH could offer near-instant results against the previously summarized graph, contrary to a full graph execution (thus yielding several 100-fold speedups each time), and still provide results with very high RBO (in line with those from the preceding and successor of the pair of queries where the update lied between). Speedup candle bar values were obtained by calculating the average and standard deviation values of the computational time across the $Q = 50$ executions.

`eu-2005`: We show the best three and worst three RBO parameter combinations in Figure 4.5 (top-left). Parameters $r = 0.20, n = 0$ captured the highest RBO values. Investing purely on increasing n is not synonymous with achieving higher accuracy. The bottom-left of Figure 4.5 shows that a value of $r = 0.05$ yielded a summary graph $\mathcal{G}$ with a number of edges around 75% of the original graph G (a higher n also contributes to increasing this value). The parameters with more conservative values (bigger n and lower r) led to the biggest summary graphs. Figure 4.6 shows the results of isolating parameters $r = 0.05, n = 2, \Delta = 0.50$ (a balanced combination leaning towards better accuracy) to assess scalability with different worker counts. Speedups of around 10 were achieved with 16 workers (see top-left), with (mainly) computation, update integration and inherent I/O time benefiting from the increase in worker counts (bottom-left).

`amazon-2008`: Figure 4.5 (top-right) shows accuracy (RBO) maximized with parameters $r = 0.20, n = 0$ and different values of Δ, with a combination of $r = 0.05, n = 6$ achieving slightly lower accuracy with a more accentuated error accumulation. We attribute the observed behaviour of lower RBO with a much higher n to the impact of the edge removal on the topology of the `amazon-2008` dataset. The number of summary graph edges as a fraction of the complete graph's edges is in line with previous results: greater values of n led to a bigger summary graph – see Figure 4.5 (bottom-right). Still, there was a pattern with $n = 6$ where there was a tendency for RBO to decrease (green diamond marker) coupled with a tendency for the summary graph edge fraction to decrease (brown $+$ marker) too. Figure 4.6 (top-right) shows that the scalability was lower than that of the `eu-2005` dataset as the number of workers is increased. This difference can be explained by the fact that `amazon-2008` has almost one fourth of the edges present in `eu-2005` (the former has more edges to process and hence to benefit from our model). The way time is distributed between I/O, integration and computation (bottom-right) is similar to `eu-2005`.

`hollywood-2011`: This social network has about 45x more edges than `amazon-2008` (around 229M edges). We evaluated this bigger dataset to focus on the scalability of our model. Figure 4.7 shows (top-left) that the speedup of the computational time was close to linear. Like before, the obtained computational speedups do

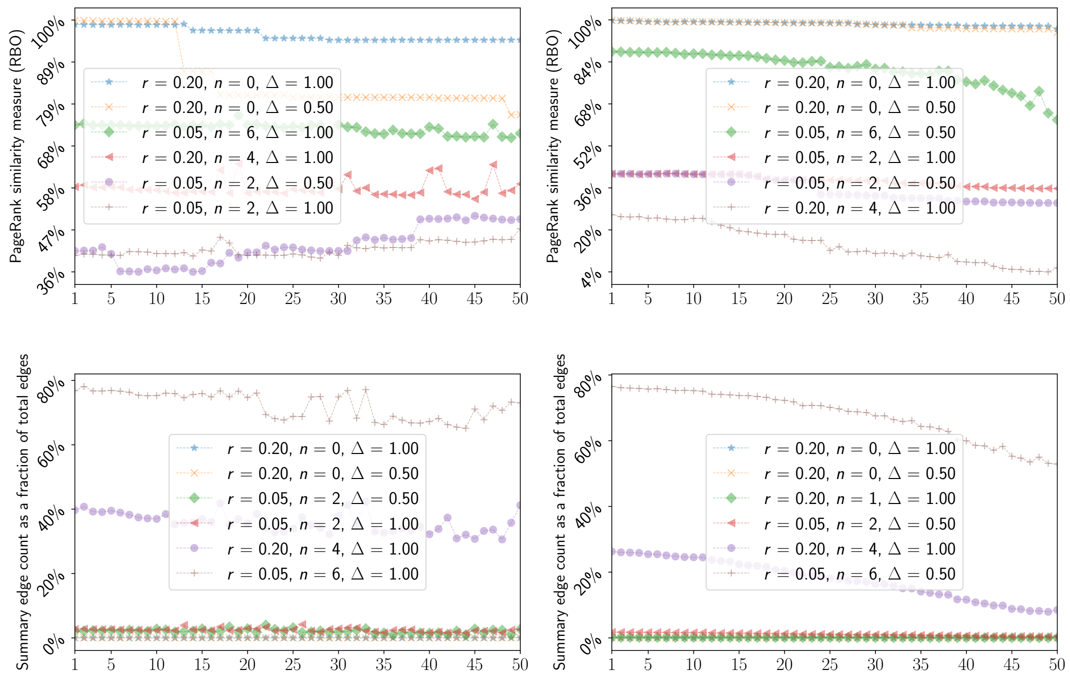
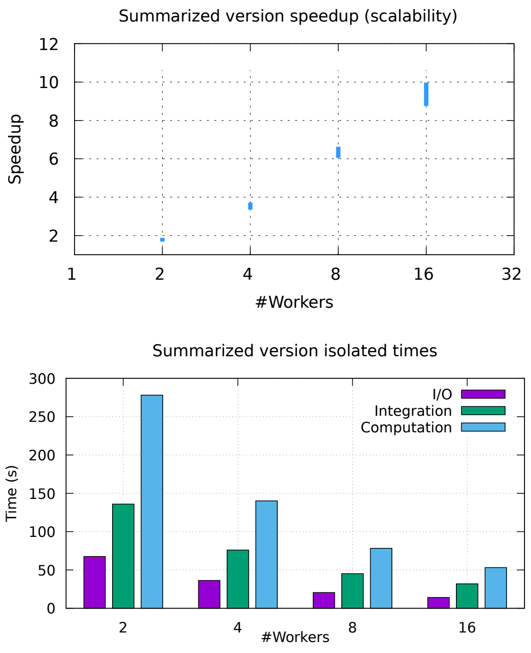


Figure 4.5: Left-side `eu-2005`, right-side `amazon-2008`. First row shows the RBO values, second row shows the number of edges $|\mathcal{E}|$ used by the summary graph as a percentage of the number of edges $|E|$ of the original graph.



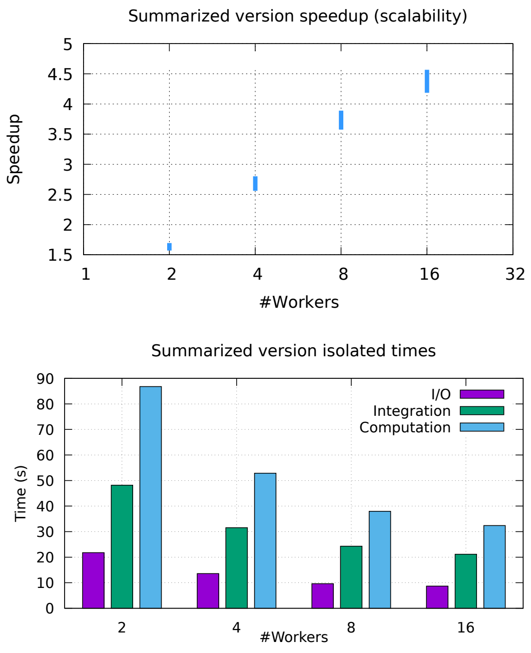


Figure 4.6: Left-side `eu−2005`, right-side `amazon−2008`. First row shows the speedup for different numbers of workers, second row shows how time was distributed between computation and integration of the updates.

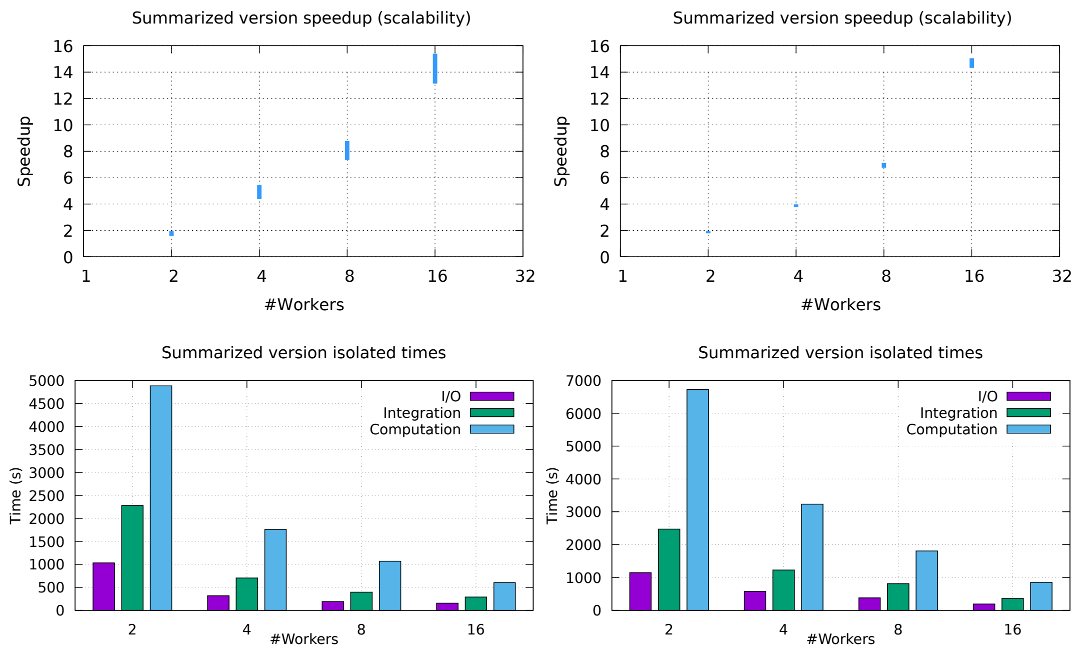


Figure 4.7: Left-side `hollywood-2011`, right-side `eu-2015-host`. First row shows the speedup for different numbers of workers, second row shows how time was distributed between computation and integration of the updates.

not include I/O and integration time as we are focused on highlighting that the scalability of the model itself - the bigger the graph, the greater the benefit that our summary graph model will have from more workers.

`eu-2015-host`: This web graph has around 386M edges and the obtained speedups over the computation are show in Figure 4.7 (top-right). Similarly to `hollywood-2011`, this dataset achieved high speedups, reinforcing the link between the model's increased efficiency and the greater-sized graphs.

Discussion. The speedups are indicative that as we test with larger datasets, executing a graph algorithm over just VEILGRAPH's K set instead of the complete graph is beneficial to performance – the bigger the graph, the greater the benefits of our model. VEILGRAPH is able to achieve faster execution while maintaining very competitive levels of accuracy. such as the case of parameters ($r = 0.20, n = 0, \Delta = 0.50$), achieving over 50% faster computational time with an RBO above 90%. The evolution of RBO for the larger datasets `hollywood-2011` and `eu-2015-host` followed similar tendencies as those illustrated in Figure 4.5, therefore here we focus solely on the performance aspect of these datasets to highlight the impact of executing on real clusters with greater sizes.

We note that it would be interesting to further evaluate `hollywood-2011` and `eu-2015-host` with even bigger cluster sizes to analyse potentially-diminishing returns in light of theoretical models for assessing scalability (Gustafson 2011). Despite this, our method has achieved a very good trade-off between result accuracy and reduction in total computation (be it in number of processed graph elements or direct time comparison), bearing in mind the deliberate lower-bound performance context we are assessing (recall: we are executing and comparing with the full processing of the graph just on 50 instants, against doing it on every vertex/edge update).

Randomized edge removals were used to assess the robustness of the model. Removals have an impact on graph topology which does not necessarily manifest as one would expect. Extending the computational scope by using conservative parameters in VEILGRAPH overall promoted a greater accuracy in our experiments, but special consideration must be given to the cases where removals may lead to a cascading effect, where extending the model to bound error propagation is an interesting challenge.

The visualization of separate computational times as shown in Figure 4.7 is revealing of a bi-dimensional cost that is often overlooked in these distributed computational platforms. This cost manifests in the overheads of communication between cluster elements and the writing/reading of data from distributed storage. Its second dimension (also a trait in `Apache Spark`) is located in the runtime of `Flink` itself. The user-logic (in our case written in `Java`) for the most part is able to abstract away that the underlying execution will take place in a distributed system. However, unoptimized, this incurs relevant communication costs which are not always obvious and whose measurement requires breaking down this abstraction. This may be achieved by directly measuring the metrics of operators executed as part of the optimized execution plan produced by the `Flink` compiler and is something we plan to explore in future work. Another interesting observation is that with more than 8 workers, for the smaller `eu-2005, amazon-2008`, the speedups obtained by horizontally-scaling the cluster diminished faster in the summary model than with using the complete computations. This is due to the summary version processing much fewer elements and thus not leveraging extra workers so much. As we tested with the bigger datasets `hollywood-2011` and `eu-2015-host`, the benefits of horizontal scaling on our model increased dramatically.

4.5 Related Work

This multidisciplinary work encompasses paradigms to express graph computations, stream processing and approximation techniques. We reiterate and comment on related state-of-the-art contributions shown in Chapter 3 that address these dimensions and comment on how VEILGRAPH also addresses them with respect to each model. While there may be some redundancy between the following information and the entries of Chapter 3, we present it as such to comment directly on the specific features of these systems:

- `Kineograph` is a distributed system designed to capture the relations in incoming data feeds (Cheng, Hong, Kyrola, Miao, Weng, Wu, Yang, Zhou, Zhao, and Chen 2012), built to maintain timely updates against a continuous flux of data.

The architecture of `Kineograph` considers two types of working distributed system nodes: *ingest* nodes which are responsible for registering graph update operations as identifiable transactions, to be distributed to *graph* nodes. This latter type of node forms a key/value store that is distributed in memory. `Kineograph` performs computation on static snapshots, simplifying the design of algorithms. The design of VEILGRAPH goes beyond this design by extending its concept to give users the flexibility to design algorithms for either the complete graph or summarized versions, with little difference. Users can incorporate the awareness that the graph has changed, or opt to design an algorithm that considers the current graph as a static version, like `Kineograph`. VEILGRAPH is implemented over `Flink`, a generic dataflow programming framework with a graph processing library, `Gelly`. `Kineograph` has an architecture which attributes importance to the task of registering new graph information and the task of storing it. Our architecture's design provides more flexibility to programmers as it is implemented over a generic programming model compared to `Kineograph`. However, as VEILGRAPH uses `Flink` which does not consider the explicit storage of the graph and its ingestion in design of the workers in a cluster of machines, it could benefit from implementing these concerns directly as a `Flink` module.

- `Naiad` is a dataflow processing system (Murray, McSherry, Isaacs, Isard, Barham, and Abadi 2013) offering different levels of complexity and abstractions to programmers. `Naiad` allows programmers to use common high-level APIs to express algorithm logic and also offers a low-level API for performance. Its concepts are important and other systems could benefit from offering tiered programming abstraction levels as in `Naiad`. Its low-level primitives allow for the combination of dataflow primitives (similar to those VEILGRAPH uses from `Flink`) with finer-grained control on iterative computations. An extension to `Flink`'s architecture to offer this detailed control would enrich the abilities that our framework is able to offer to users.

- `Tornado` is a system with an asynchronous bounded iteration model, offering fine-grained updates while ensuring correctness (Shi, Cui, Shao, and Tong 2016). It is based on the observations that: *1)* loops starting from *good enough* guesses usually converge quickly; *2)* for many iterative methods, the running time is closely relative to the approximation error. Whenever a result request is received, the model creates a branch loop from the main loop. This branch loop uses the most recent approximations as a guess for the algorithm. It is a technique that could benefit from applying VEILGRAPH's summarization model, as `Tornado` 's main loop could produce approximations faster, making the computation result guesses readily available as queries arrive.

- `KickStarter` showcased a technique for trimming approximate values of vertex subsets which were impacted by edge deletion (Vora, Gupta, and Xu 2017). `KickStarter` deals with edge deletions by identifying values impacted by the deletions and adapting the network impacts before the following computation, achieving good results on real-world use-cases. By focusing on monotonic graph algorithms, its scope is narrowed to selection-based algorithms. We decouple in VEILGRAPH the approximation technique, the summarization model and the algorithm type. Thus, we are able to offer the big vertex model and provide a structured sequence of steps to integrate another model or approximation technique (e.g. `KickStarter`'s own technique could be a candidate). Rather than the hard-coded logic for bounding the approximation values specifically to monotonic algorithms of `KickStarter`, we offer in VEILGRAPH a middleware layer which offers additional customization capabilities to programmers who with, for example to implement a different logic to bound approximations or to even implement automated strategies based on statistical analyses and machine learning.

- `GraphBolt` (Mariappan and Vora 2019) is a recent work building on `KickStarter`, offering a generalized incremental programming model enabling the development of incremental versions of complex aggregations. They evaluate different algorithms while defining different aggregation functions for each in order to support the computation approximations. This system is relevant as their work focuses on crafting functions for specific use-cases, though with a loss of control for the programmer as there is no way to provide custom-defined logic for how dependency tracking and error tolerance is defined.

- `FlowGraph` (Chaudhry 2019) is a system that proposes a syntax for a language to detect temporal patterns in large-scale graphs and introduces a novel data structure to efficiently store results of graph computations. This system is a unification of graph data with stream processing considering the changes of the graph as a stream to be processed and offering an API to satisfy temporal patterns. `FlowGraph`'s model for addressing the temporal evolution of the graph and subsequent execution was innovative by formalizing the pattern detection with its own language approach. Integrating this approach in the post-update pre-execution stage of VEILGRAPH could provide a richer model for the programmer.

4.6 Remarks

We designed VEILGRAPH which provides a well-defined structure to incorporate custom approximate processing strategies and to enable choosing between built-in behaviours. Our experiments in the context of random walk problems lead us to conclude that the VEILGRAPH model, even when tested in a deliberately challenging context for comparability, is a viable basis to enable faster, more efficient and configurable graph processing on this type of problem in many real-world scenarios. The results we obtain were produced under a stream scenario where graph updates are big enough so that the changes to the graph do not explicitly benefit the summarization model we evaluated.

In our current design, we do not interleave updates with queries because we first aimed to assess the scalability and potential of the summarization approach to the case of vertex centrality as an initial validation of our idea. Extensions of this work should take into account interleaving as an additional means of improving system performance.

4.6.1 Bridging the Gap for OLAP systems

We have presented members of a rich graph processing landscape. Many of the systems we discussed pioneered new approaches which were adopted by more popular ones. We see that `Giraph` as an instance of `Pregel` has some of its concepts present in `Flink` and `Spark`. Systems such as `KickStarter`, `Kineograph` and `Tornado`, although their source is not available as far as we know, validated approaches to dimensions such as stream processing and expressing approximations over graphs. Notwithstanding the importance of other graph processing systems (from a group one may classify as `OLAP` as per Section 3.7), these are particularly interesting for potential developments due to their booming and active communities as well as higher-level libraries they received (e.g., `GRADOOP` and `GraphFrames`).

`Flink` was designed from the start as a stream-processing engine with rich semantics. It supports batch and stream processing respectively through its `DataSet` and `DataStream` APIs. The internal implementation of these APIs is separate. In the case of batch processing, it has libraries for machine learning and also for graph processing (`Gelly`). `Gelly` is implemented in `Java`[8] and expresses building-blocks of graph computation in dataflow programming over `Flink`'s `DataSet` API.[9] However, as `Gelly` does not use the stream API, all the window semantics of the `DataStream` API become unusable. While `Gelly` has seen a recent freeze regarding new features (most algorithms have been last changed one/two years ago, with the most recent updates dating six months and touching on performance improvements), it has been used in `GRADOOP`, which is an open-source distributed graph analytics research framework (Junghanns, Kießling, Teichmann, Gómez, Petermann, and Rahm 2018) under active development. `GRADOOP` provides an even higher-level of expressing graph manipulation.

`Spark`, was designed as the next-generation big data processing system of its time. As stream processing gained additional attention, it was implemented as the `Spark Streaming` library. Stream and batch processing have different APIs which were implemented differently. It has its graph processing library `GraphX` which was built over the system's batch processing API, like the case of `Flink`'s `Gelly` and also suffering from the same previously mentioned limitations. A higher-level API was designed to extend the functionalities of `GraphX` while harnessing `Spark`'s `DataFrame` API. For this, the `GraphFrames` library was created. It is relevant to mention that in `GraphX`, there is a method to cache results of a dataflow computation for it to be reused. While not a solution, it avoids the recalculation of results and allows the flow of execution to skip intermediate I/O overheads, to a limited degree.

Overall, `Spark` and `Flink` are general-purpose dataflow processing systems with graph libraries. Due to the size of their communities and their activity as open-source

[8] https://github.com/apache/flink/tree/master/flink-libraries/flink-gelly/src/main/java/org/apache/flink/graph

[9] https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/

projects, they become very interesting target candidates to incorporate the overlapping concerns of stream semantics, graph evolution and maintenance across cluster memory. These concerns would have to be implemented across different components of the systems, from the way dataflow jobs are broken down into tasks, to task-scheduling itself, all the way to job execution status (`Spark` and `Flink` have state machines to control the evolution of a job). Orthogonally to this, a new logic would be needed to avoid triggering massive communication overheads when the topology of the maintained graph (in the cluster) changes due to the windows of arriving updates. When it becomes apparent that a vertex in the graph is exhibiting scale-free properties (by virtue of its degree growing), special strategies must be taken to ensure stability of the graph maintained in the cluster. The results achieved in `PowerLyra` (Chen, Shi, Chen, and Chen 2015) provide some preliminary insights on how to approach this challenge.

4.6.2 Bridging the Gap for OLTP systems

The decade of 2010 has seen the launch of many open-source and commercial graph database technologies. They focus overall on aspects of efficient low-level graph representation, with some offering only vertical scaling (`Neo4j`) while others provide horizontal scaling (sharding with `JanusGraph`, `Dgraph`, `ArangoDB` and `Cayley`). Some of these database technologies delegate the ability of sharding to an underlying interchangeable storage technology: for example those derived from `Titan` can switch between `HBase` and `Cassandra`.

The graph dynamism in these technologies is inherent to the fact transactions may be applied to update the graph. However, as far as we know, the way the graph can evolve and algorithm results may be updated does not consider stream semantics or time-bounded conditions in any of the `OLTP` systems herein detailed. We envision the possibility to enrich graph databases with such semantics (e.g. taking inspiration from systems such as `BlinkDB` (Agarwal, Mozafari, Panda, Milner, Madden, and Stoica 2013)).

To this end, under the `OLTP` category, we focus on systems with a high number of algorithms (`Neo4j`), the ability to plug in different distributed back-ends to support the graph database semantics (`JanusGraph`) and inherent sharding capabilities (e.g. the open-source `Dgraph`). `Neo4j`'s biggest limitation is the lack of graph sharding. For `Cayley`, we did not find any list of algorithms implemented over it. Still, it supports sharding like `JanusGraph` by delegating that responsibility over the underlying storage module. The `ArangoDB` website claims[10] it automatically performs sharding of the data. Together with its number of algorithms, it is also a valid candidate to develop new approaches to what is discussed in this document. We do not focus on `Weaver` due to it not being active. `AllegroGraph` is unsuitable as it is neither distributed nor open-source. `Dgraph` is open-source, supports sharding and although it does not have many graph algorithms, has an active open-source community.

Regarding `Neo4j`, while its most relevant capabilities are only available in the En-

[10] https://docs.arangodb.com/3.4/Manual/Scaling/

terprise Edition (such as storing more than 34 billion vertices in a graph), its active community and broadening scope of use cases make it interesting. If Neo4j's development is invested in scale-out capabilities for sharding a graph across multiple cluster nodes (increasing write performance compared to read operations), it will be closer to the possibility of offering a graph that is in fact maintained in a cluster. By using smart caching mechanisms, even if some storage I/O is necessary, it could become a strong candidate to bridge the best features of OLAP and OLTP systems. Access latency to secondary storage in this scenario would be further mitigated due to the fact that the Neo4j process in each worker node could be responsible for both distributed graph processing and mediating I/O access.

Moving towards a system that supports executing over an evolving graph that is quickly-accessible on OLTP-type systems would require some key-points: incorporating stream processing windowing semantics in the API; exploiting low-level graph representation optimizations (as it happens on Neo4j); researching optimal strategies of graph sharding coupled with caching mechanisms (JanusGraph, ArangoDB and Dgraph already have sharding). Of the latter, JanusGraph and Dgraph already have an active community, making them relevant targets for future research.

4.6.3 Summary

With the proposal, implementation and experimental evaluation of VEILGRAPH, we analysed the impact of result approximation by use of graph summarization. We observe it is productive to explore the relationship between result accuracy and computational performance. It is important to note that the analysis of result integrity which relies on directly comparing summarized and non-summarized execution outputs is a way to integrate awareness of domain-specific aspects of the results. We believe this should be coupled with theoretical models for real-time analysis of error propagation.

Future directions for this work would encompass other graph algorithms like online community detection and to further evaluate on bigger datasets. There is a challenge in the disruptive aspects of edge deletions over graph topology and how that may speed up or slow down the propagation of approximation errors. To this end, different approximation strategies based on statistical records, from manually implemented policies to automated decisions on parameters would be interesting to explore. The statistical basis proposed by GraphBolt (Mariappan and Vora 2019) is another relevant approach to define these decisions.

It must be mentioned that the summary graph model of VEILGRAPH had its evaluation focused on the number of edges in the summary as it targets vertex centrality (manifested with PageRank). For the future, and coupled with other summarization techniques, other aspects such as number of vertices may become relevant, depending on use-case. Extending the VEILGRAPH model, applying variants to other types of graph algorithms and precisely evaluating performance and accuracy would make our contribution a more robust approach to exploiting the relationship between these two properties. The ideas explored in VEILGRAPH have synergy with the two phases of our community network micro-cloud technique presented in Chapter 5, GELLY-

SCHEDULING. That exploratory work has a community detection phase followed by a leader election phase (both implemented as graph algorithms). Researching the applicability of GELLY-SCHEDULING to other types of network that are dynamic and greater in size could benefit from the techniques used by VEILGRAPH. The summarization and approximate processing approach of VEILGRAPH have the potential to improve the execution speed of both phases of GELLY-SCHEDULING without negatively impacting the accuracy of elected leaders and therefore Quality-of-Service. This would be relevant to explore even if through a different implementation not based on `Flink`.

VEILGRAPH may also benefit from the use of smart graph data structures. Compact graph data structures, offering both compression of the graph and the ability to change it (without converting the representation to an uncompressed format) could improve resource efficiency of an underlying distributed system such as `Flink`. A relevant exploratory work we envision is the application of a compact data structure such as our contribution on k^2-trees (Chapter 6) to the internal representation in `Flink`. Sufficiently high compressions ratios could enable redundancy by fully representing the graph within each node of the cluster, potentially enabling additional performance gains.

5 Gelly-Scheduling: Service Placement

In this chapter, we present our initiative GELLY-SCHEDULING described in (Coimbra, Selimi, Francisco, Freitag, and Veiga 2018) and developed in collaboration with the University of Cambridge and the Polytechnic University of Catalonia. Here we: *i)* highlight the applicability of leader election heuristics which are important for service placement in community networks and scheduler-dependent scenarios; *ii)* present a parallel and distributed solution designed as a scalable alternative for the problem of service placement, which has mostly seen computational approaches based on centralization and sequential execution.

5.1 Introduction

Community networks (CNs) are owned and managed by volunteers and offer various services to their members. Seamless computing and service sharing in CNs have gained momentum due to the emerging technology of CN micro-clouds. CNs have seen an increase in the last fifteen years. Their members contact nodes which operate Internet proxies, web servers, user file storage and video streaming services, to name a few. Detecting communities of nodes with properties (such as co-location) and assessing node eligibility for service placement is thus a key-factor in optimizing the experience of users. We present a novel solution for the problem of service placement as a two-phase approach, based on: *1)* community finding using a scalable graph label propagation technique and *2)* a decentralized election procedure to address the multi-objective challenge of optimizing service placement in CNs. One such network is guifi.net, located in the Catalonia region of Spain. It is a successful example of this paradigm. Guifi.net is defined as an open, free and neutral CN built by its members pooling resources. Guifi.net was born in 2004, and until today, has grown into a network of more than 34,000 operational nodes. Previous work on guifi.net classified services into network-oriented and user-oriented. For these two types in the Catalonia region, the three most prevalent occurrences were (Selimi, Khan, Dimogerontakis, Freitag, and Centelles 2015): *a) network-oriented services (558 in this region)* – network graph-servers (39.24%), DNS servers (35,48%) and NTP servers (17.20%); *b) user-oriented services (514 in this region)* – proxy servers for Internet access (53.50%), web pages (11.08%) and communication applications such as VoIP, audio, video and instant messaging (9.33%).

Nodes in guifi.net are exclusive to specific geographical zones (there are no overlays) such as what is depicted in Figure 5.1. There are special-purpose nodes called graph-servers, which are responsible for performing network measurements between

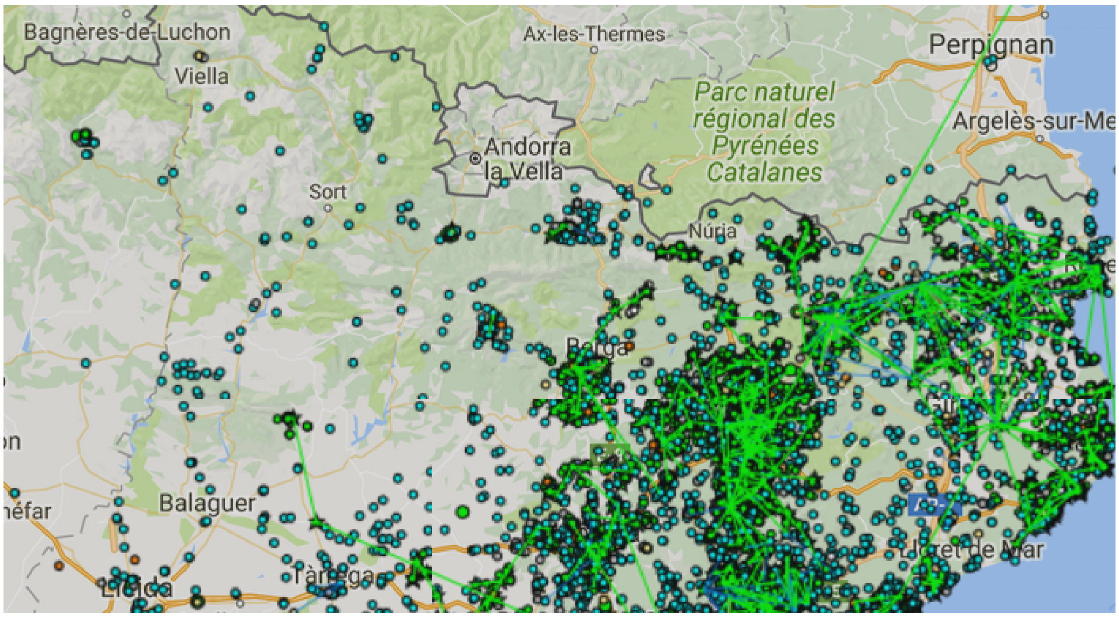


Figure 5.1: Depiction of guifi.net's Osona region.

nodes and have an API for querying node states (Selimi, Freitag, Cerdà-Alabern, and Veiga 2016). These graph-servers comprise a distributed hierarchical monitoring system which records the network's link data traffic properties. Guifi.net is thus a relevant test bed for developing and validating techniques to enhance service placement and system scheduling by exploring their requirement of leader election. In turn, these may be extrapolated to more complex scenarios, such as placement in P2P networks (typically irregular), industrial contexts and IoT scenarios. A simple web proxy would most likely have node latency as its most relevant parameter. On the other hand, a mission-critical quality-of-service proxy could place the focus on node availability. Heuristics may encompass network features such as topology, as well as domain attributes (such as availability and quality of specific resources). While one may intuitively define one heuristic as absolute, this could produce scenarios which are locally optimal but globally undesirable. What if the node with the highest availability happens to be on the outer rims of the network? Aspects of network topology are as relevant for system efficiency as the service-level heuristics which traditionally guide leader election for placements.

Our objective is to devise an efficient, scalable solution which is easy to fine-tune regarding domain-specific attributes, and that provides seamless scalability for increasing network size and number of services. For this, we propose a platform that enables incremental processing in a scenario where information continuously arrives: changes in network, node and service quality are continuously monitored. Our solution is a two-phased approach which optimizes the definition of communities (**Phase One**) and election of leaders (**Phase Two**) in a community communications network.

The concept of scheduling influenced the name GELLY-SCHEDULING. The community definition and leader election procedure aims to produce the most appropriate network configuration such that different Quality-of-Service constraints are optimized for offered services. In particular, each elected leader would be in charge of deciding resource allocation and job/task scheduling within each community. Our contribution in this regard makes use of graph processing to provide an alternative approach to defining this infrastructure where services are assigned to the most desirable nodes.

The chapter is organized as follows. Section 5.2 explains the two main phases of our algorithm. Section 5.3 details our evaluation methodology and obtained results. Section 5.4 highlights relevant studies on community networks and service placement. Section 5.5 summarizes our contribution's highlights.

5.2 Gelly-Scheduling Service Placement

The challenges inherent to service placement for large scale geo-distributed networks (such as community networks) are usually addressed in the literature with a batch-oriented non-scalable approach. The typical approach consists of performing a search (exhaustive or via heuristics) in a centralized computing unit. All the information about network links and nodes is centrally and sequentially processed, in order to determine the best network configurations as far as service placement is concerned. While the unit responsible for this search may benefit from hardware improvements, they are merely a form of vertical scaling (which is limited). This approach does not prioritize reaction to changes in the network and its nodes, in order to make service placement more dynamic in a context of continuous monitoring. It also doesn't scale in the context of larger networks. We present a novel method capable of both achieving scale-out processing for optimizing community network topology as well as electing service placement targets within communities in a decentralized approach. We employ community detection as a parallel technique which enables the partitioning of the problem space to optimize node placement in communities. This allows for an efficient leader election to execute concurrently (each community being responsible for its leader) and in parallel within each community. This work aims to improve service placement for networks in a way that users and processing tasks are balanced regarding bandwidth restrictions and data sources.

Phase One: Community Finding. We use two definitions of community: *default* – the zone-based node distributions, provided in the dataset as-is (insignificant preprocessing is performed in this case); *custom* – a state-of-the-art label propagation technique (Raghavan, Albert, and Kumara 2007) applied for detecting communities. We build an undirected graph $G = (V, E)$ by defining a set of n nodes V and a set of m edges E such that an edge $e \in E$ will be created if and only if there is a corresponding link element between two working devices (each belonging to a working node) in the dataset. Single-leaf nodes were discarded as part of preprocessing. The goal of this phase is to rapidly partition the problem space into a configuration that promotes scalability of computation and efficient resource usage. We provide the pseudo-code for the most relevant actions of Phase One in Algorithm 2, where C is an upper bound on the number of iterations to execute (a default limit of $C = 10$ iterations is common in the literature for convergence (Boldi, Rosa, Santini, and Vigna 2011)). Phase One

Algorithm 2 Phase One: Community Finding

```
1: INPUT: $G = (V, E), C = 10$
2: OUTPUT: $Z$                                  ▷ Set of graphs representing communities
3: for all $v \in V$ do
4:     $v$.generateUniqueLabel()
5: end for
6: $G' \longleftarrow G$.setUndirectedEdges()
7: $i \longleftarrow 1$
8: for $i < C$ do
9:     for all $v \in V$ do
10:        $M \longleftarrow v$.getInboundMessages()
11:        $L \longleftarrow M$.getMostFrequentLabels()
12:        $v$.updateLabel($L$.filterHighestLabel())
13:    end for
14:    $i \longleftarrow i + 1$
15:    if not $G'$.labelsChanged() then
16:        break
17:    end if
18: end for
19: return $Z \longleftarrow G'$.groupByLabels()
```

thus becomes an important instrument in efficiently defining groups of network nodes by employing a state-of-the-art technique in community detection. These groups aid the optimization process of service placement, effectively serving as a useful blueprint for Phase Two of our algorithm. The two phases form a technique to harness current platforms and infrastructures to tackle service placement. Conceptually, there is a top-tier master node which is responsible for: *1)* querying the graph-servers for all of the network's node information; *2)* executing Phase One of our algorithm to obtain a definition of communities; *3)* informing each node of its community's composition. This is depicted in Figure 5.2, where in the middle there is a centralized entity consisting of one or (potentially many) more computational workers. It initially queries graph-servers (or whatever network visibility mechanisms are in place) to obtain a snapshot of the network's nodes. Then it executes Phase One of our algorithm, decomposing the network into communities. A major computational advantage of Phase One is that this master can be a single machine or a set of workers in a cluster, effectively scaling with the computational capability available to this top-tier master. Each community member is then informed of the elements of its own community: required to proceed to Phase Two.

Phase Two: Leader Election. Phase Two receives a set of communities and elects

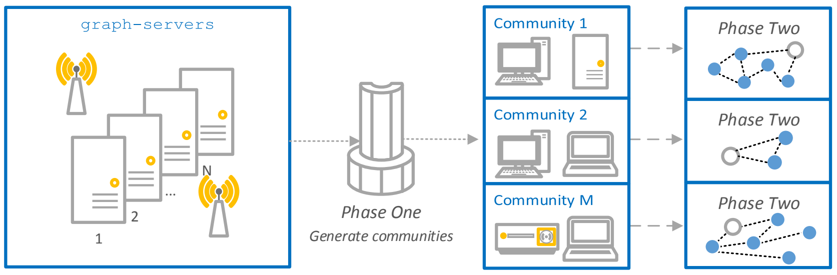


Figure 5.2: The graph-servers on the left send the network heuristics to the master node; the master node in the middle decides on the community configuration through Phase One; communities concurrently elect an internal leader during Phase Two.

a leader for each one. This election phase is self-contained for each community, in the sense that a distributed implementation of this phase can be carried out concurrently with respect to communities and in parallel within each community with our graph-based approach. The right-side of Figure 5.2 illustrates this. There may be more than one connected component in geographical zones of guifi.net. Due to this, for every community network G, only the nodes belonging to the largest connected component of G are used to choose a leader for service placement. This election consists of Phase Two of our algorithm and is detailed in Algorithm 3. This phase serves the purpose of identifying the best node for service placement. Leadership is attributed through a scoring, where the score s_i of each node i lies in defining a linear combination of two sets of heuristics (described in Table 5.2). One set is based on system-centric values: availability β_1 and latency β_2 as defined by graph-servers (Cerdà-Alabern 2012), as well as computational class β_3 as per Table 5.1 and defined as part of this work; the other is calculated as part of this algorithm and consists of betweenness α_1 and closeness α_2 centralities. We defined heuristic β_3 as a score in three computational categories for nodes: *i)* server-type nodes which typically have stronger computational power to support more demanding services; *ii)* non-server nodes with more than one device; *iii)* non-server nodes with a single device. Table 5.1 shows the representation of β_3 for each category for the data we analysed. The values we attribute to β_3 were selected arbitrarily to represent computational power of a given node. This categorization serves the purpose of approximating realistic tiers of computational capabilities for nodes in the network – information which, as far as the authors know, is not readily-available in the guifi.net CNML dataset. Thus, as an example, the initial score of a node i will be defined as:

$$s_i = w_1\,\alpha_1 + w_2\,\alpha_2 + w_3\,\beta_1 + w_4\,\beta_2 + w_5\,\beta_3 \tag{5.1}$$

Table 5.2 details the specifics of each heuristic, namely their meaning and how they are obtained. Notation-wise, i is the node to be scored while u and v represent arbitrary nodes in the community graph G with n nodes, $\sigma_{u,v}$ is the number shortest paths from u to v, $\sigma_{u,v}(i)$ is the number of those that pass through i, and $d(i,v)$ is the geodesic

Table 5.1: Frequency of per-node device count categories. The most frequent services are Internet proxies, a consequence of guifi.net existing as an alternative to the standard ISP model.

Class	#Nodes = 23,468	Percentage	β_3
i) Strong	337	1.436%	1
ii) Medium	1666	7.099%	0.5
iii) Weak	21 465	91.465%	0.1

Algorithm 3 Phase Two: Leader Election

```
1: INPUT: $G$ =($V, E$), $W$ (heuristic weights)          ▷ Heuristic weight array
2: OUTPUT: $R$                                           ▷ Decreasing-order ranks
3: $\alpha : Array \longleftarrow$[ ], $\beta : Array \longleftarrow$[ ]
4: for all $v \in V$ do
5:     $\alpha[v]$ = $[\alpha_1 \;\; \alpha_2] \longleftarrow v$.calculateAlphas()
6:     $\beta[v]$ = $[\beta_1 \;\; \beta_2 \;\; \beta_3] \longleftarrow v$.getBetas()
7:     $v$.setScore($W *$ [ $\alpha[v] \quad \beta[v]$ ])
8: end for
9: $R \longleftarrow G$.getVertices().orderByScore()
10: return $R$
```

distance between i and v. Phase Two was designed under two types of evaluation based on configuration of heuristics: **Absolute heuristics** - in this case, leader selection is guided exclusively by exactly one of the heuristics. We analyse the impact of each individual heuristic, setting the weights of others to zero. **Combined heuristics** - we consider a linear combination of two heuristics. We set unbalanced weights in order to better determine the more significant contributions, in the sense that for two arbitrary heuristics m_1 and m_2, we may define the node score to be $v_s = (1 - f)m_1 + fm_2$, or the reverse. If for example one heuristic weights in for $f = 60\%$ of the score, the other will account for the remaining 40%.

5.2.1 Implementation

Regarding our implementation over `Flink`, we implement our algorithms over the `Gelly` graph processing library. We use its API and did not modify its internal components.

Scores of heuristics α_1 and α_2 were obtained for each community G using the `Python NetworkX` library, for use in Phase Two. Overall, the time to calculate them is negligible when compared to the total amount of time required to compute Phase One plus Phase Two. There are common aspects to generating samples for bandwidth and round-trip time, but each was based on different statistical artifices.

Bandwidth. Let BW ˜ $K(k, h, \xi, \alpha)$ represent the empirical bandwidth distribution.

Table 5.2: Algorithm's heuristic symbols and meanings.

α_1	**Betweenness Centrality** $\sum_{u \neq i \neq v} \frac{\sigma_{uv}(i)}{\sigma_{uv}}$, fraction of shortest paths from u to v, for all nodes u and v, passing through node i.
α_2	**Closeness Centrality** (Newman 2010) $(n-1)/\sum_v d(i,v)$, where $d(v,i)$ is the geodesic distance from node i to node v.
β_1	**Availability** Percentage of ping responses received by a graph-server (%) over a specific time period.
β_2	**Latency** Ping response timing, measured by a graph-server (ms) over a specific time period.
β_3	**Computational Class** Defined by the number of devices handled by the node, as well as its role.

K stands for the four-parameter Kappa distribution (Hosking 1994), where k and h denote the shape of the distribution, ξ denotes its location and α is a scaling factor. These four parameters were estimated using L-moment statistics, namely through the `lmoms` function which computes the sample L-moments and the `parkap` function which estimates the four parameters of K based on the sample L-moments. Both functions are part of the `R lmomco` library. The four-parameter Kappa distribution is used for simulating additional samples based on the empirical distribution made by using the `rkappa4` function of the `R FAdist` library for random generation purposes.

Round-trip time. Let $RTT \sim GEV(\mu, \sigma, \xi)$ represent the empirical round-trip time distribution. GEV is the generalized extreme value distribution. It has four parameters: μ, which is the location of the distribution, σ which represents the scale and ξ which represents the shape of GEV (influencing the behaviour of the distribution tail). The previously-referenced `lmoms` function was used as well, with `pargev` now being the function (also present in library `lmomco`) responsible for estimating the GEV parameters based on the sample L-moments. Additional round-trip time samples were simulated using the `rgev` function. GEV exists as a family of continuous probability distributions, stemming from extreme value theory (Coles, Bawa, Trenner, and Dorazio 2001).

The method of L-moments is used to understand insights of analysed data and to estimate distributions (Hosking 1990; Hosking and Wallis 2005) using efficient techniques (Hosking 2000). Figure 5.3 shows the $\log_{10}$ plot of the bandwidth, while Figure 5.4 shows the same for round-trip time.

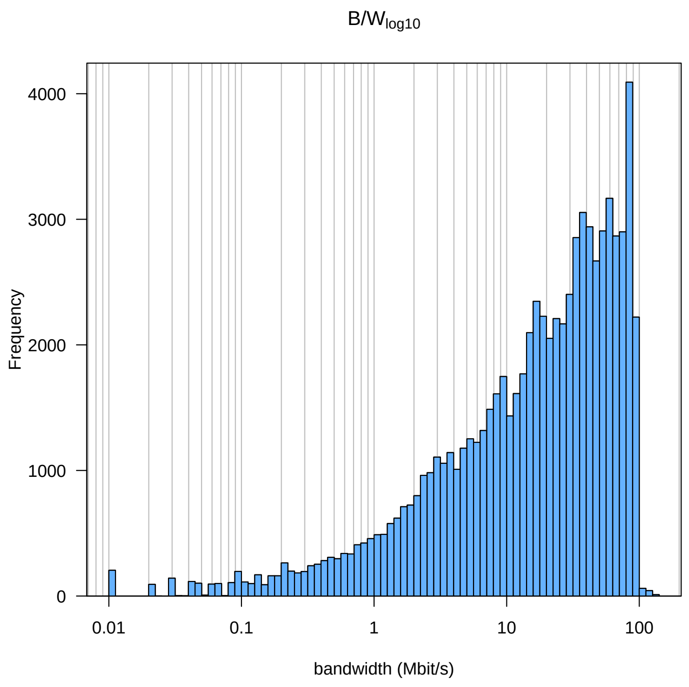


Figure 5.3: Bandwidth (B/W) in logarithmic scale.

5.3 Experimental Evaluation

There are 23,391 nodes identified as working and, for the whole guifi.net, there are 878 nodes defined as servers. This implies that, at most, 3.75% of the working nodes could actually be sustaining full fledged services, which adds importance to how leader election is performed to assign services. We believe guifi.net, while it is in fact an open community network, has a type of topology which allows for extrapolating results into other sorts of networks. This claim is made based on previous research work in the literature (Selimi, Freitag, Cerdà-Alabern, and Veiga 2016), which both analysed the impact of prioritizing different heuristics on the computational and network resources available (Selimi, Vega, Freitag, and Veiga 2016) and studied practical issues with microservice architectures (Selimi, Cerdà-Alabern, Artigas, Freitag, and Veiga 2017). We used available statistical processing tools to attempt to fit several distributions and compare them. For the bandwidth, we present plots of the distribution fitting and Empirical Cumulative Distribution Function in Figures 5.5 and 5.6. In the same order, we also present the aforementioned plots for round-trip time in Figures 5.7 and 5.8. We then modelled the ECDF of both network properties with the use of the `lmomco` and `FAdist` libraries in `R`.

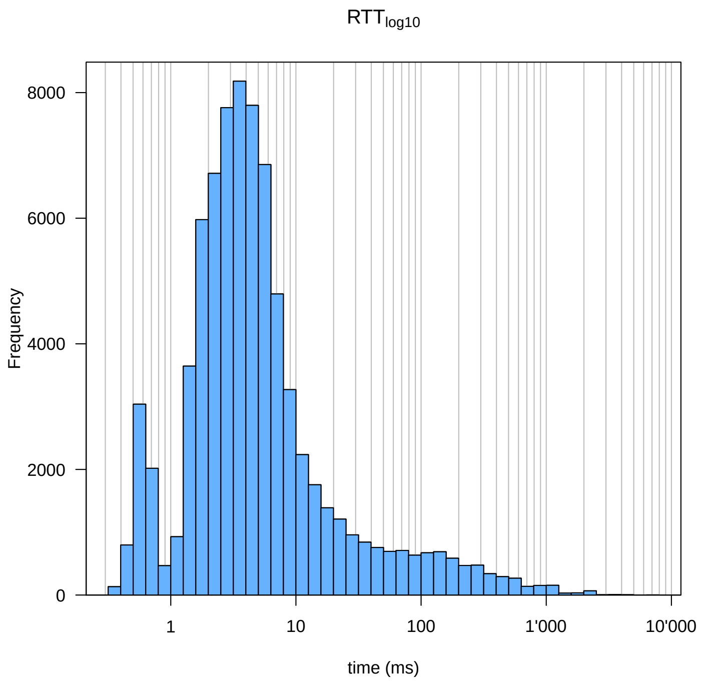


Figure 5.4: Round-trip time (RTT) in logarithmic scale.

Network Characteristics. Part of guifi.net exists as an instance of the *Quick Mesh Project* (QMP) [1], a system for easily deploying MESH/MANET networks using Wi-Fi technology. QMP is an urban mesh network in Barcelona and it is a subset of the guifi.net community network sometimes called Sants-UPC network. It was designed for use in scenarios such as free community networks, of which guifi.net is a rich example (Selimi, Cerdà-Alabern, Artigas, Freitag, and Veiga 2017). We use measurements of round-trip time (RTT) and bandwidth (B/W) from the Sants-UPC wireless mesh QMP instance to establish a model of these telecommunication heuristics for the remainder of the network. It would be through a hierarchy of graph-server nodes that one would acquire a view of all the nodes in the network. However, due to privacy and maintenance issues, many of these graph-server types fail to provide any type of information about queried nodes. Due to this, we employed a one-week snapshot of this seventy-node QMP instance to establish ground-truth relevance for our work. The measurements were taken for seven days from the 1st to the 8th of March 2017, with a snapshot taken every hour (Cerdà-Alabern 2012). The measurement period and frequency produced enough samples for evaluating guifi.net in light of the results of our method. We remark that node links in QMP and guifi.net, in general, are not symmetrical: the bandwidth and round-trip time from node u to node v isn't necessarily the same from v to u.

Firstly, although QMP has a more uniform set of nodes compared to guifi.net, it

[1]`Quick Mesh Project` website. Access date: 28 Dec '17. http://qmp.cat/

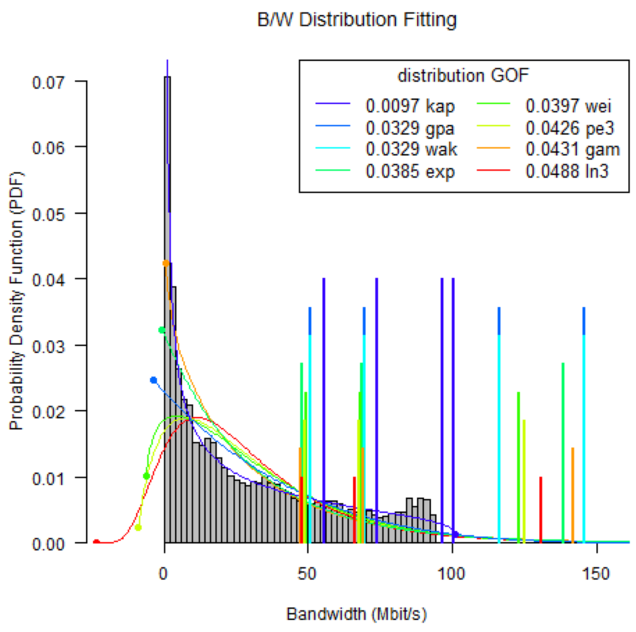


Figure 5.5: Bandwidth (B/W) observation comparison and goodness-of-fit of different candidate distributions. Lower goodness-of-fit is better.

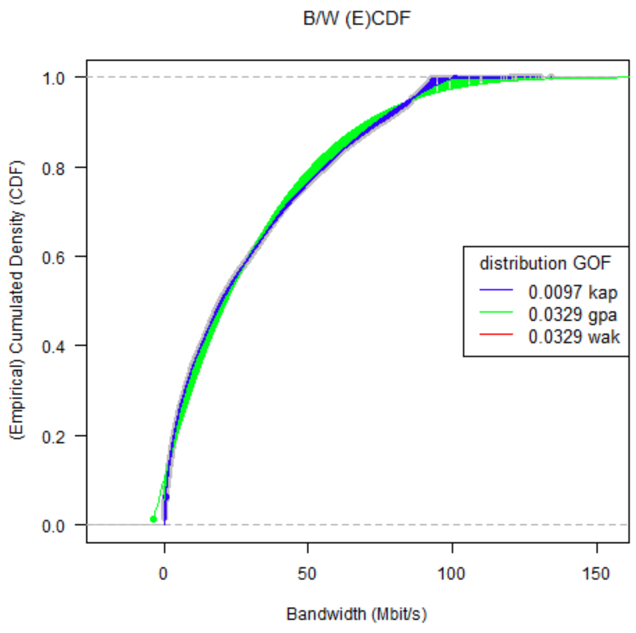


Figure 5.6: Bandwidth (B/W) Top 3 Empirical Cumulative Distribution Function (ECDF).

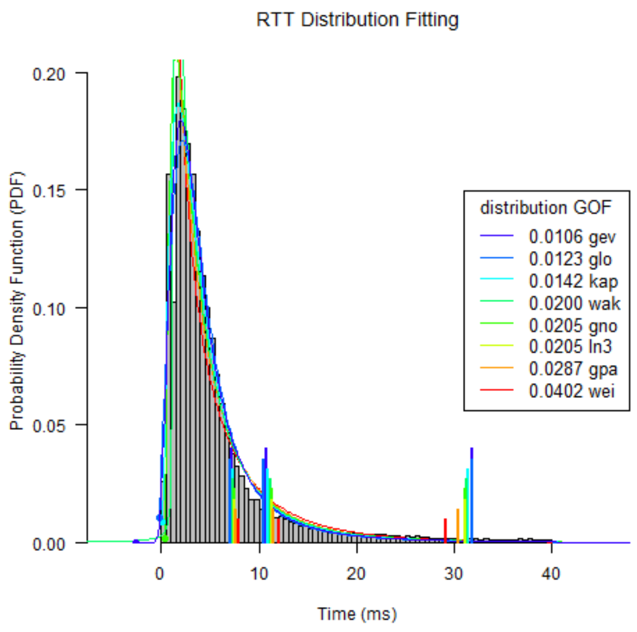


Figure 5.7: Round-trip time (RTT) observation comparison and goodness-of-fit of different candidate distributions. Lower goodness-of-fit is better.

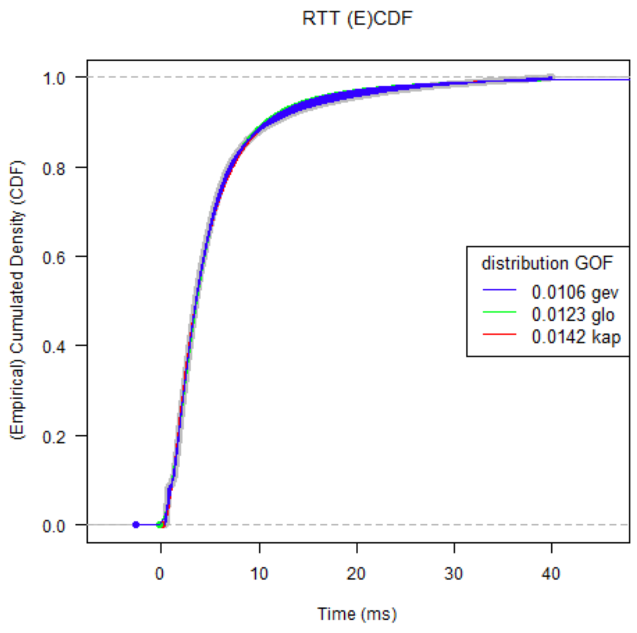


Figure 5.8: Round-trip time (RTT) Top 3 Empirical Cumulative Distribution Function (ECDF).

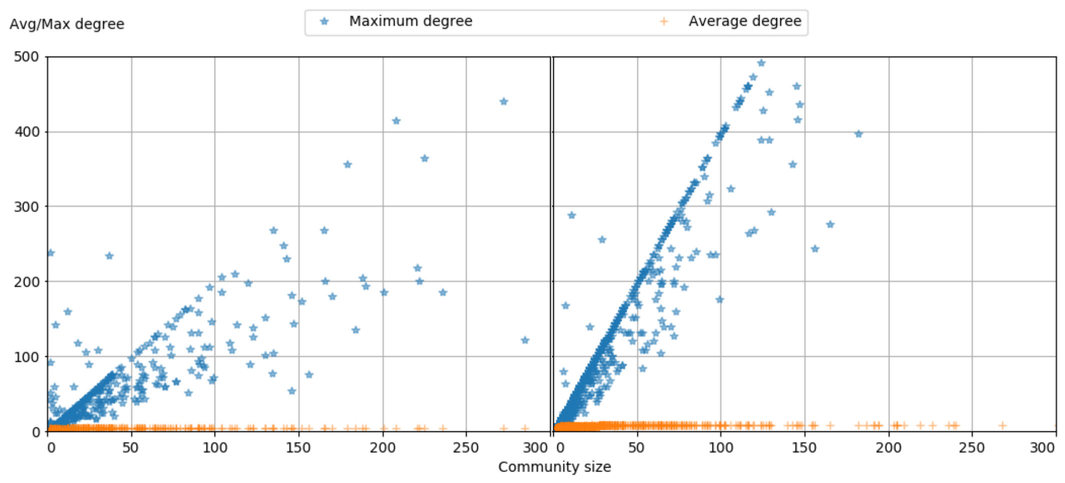


Figure 5.9: Plot of maximum and average degree distributions for each community. The left image is the geographical configuration of node sets, while the right side is based on Phase One of our algorithm.

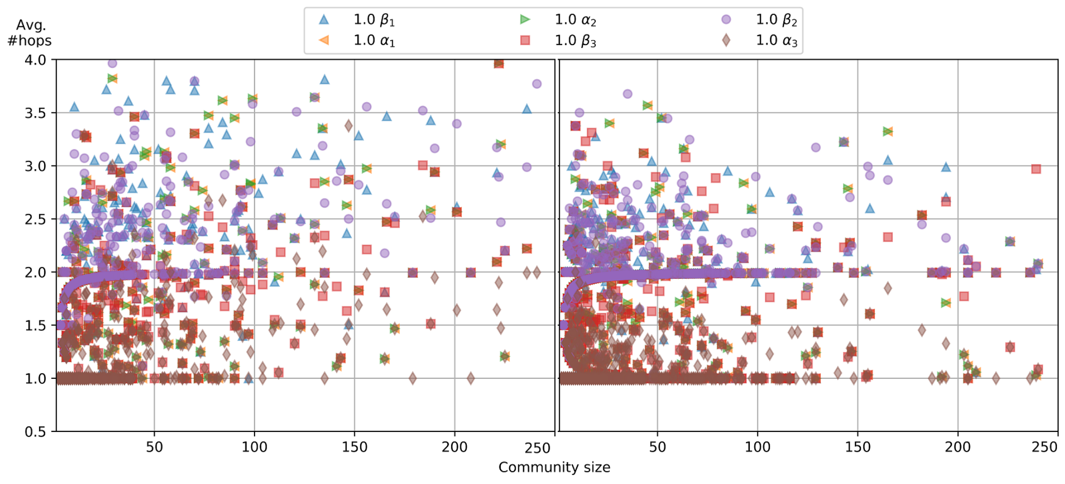


Figure 5.10: Average number of hops-to-leader plotted against each community's size. The left image is the geographical configuration of node sets, while the right side is based on Phase One of our algorithm.

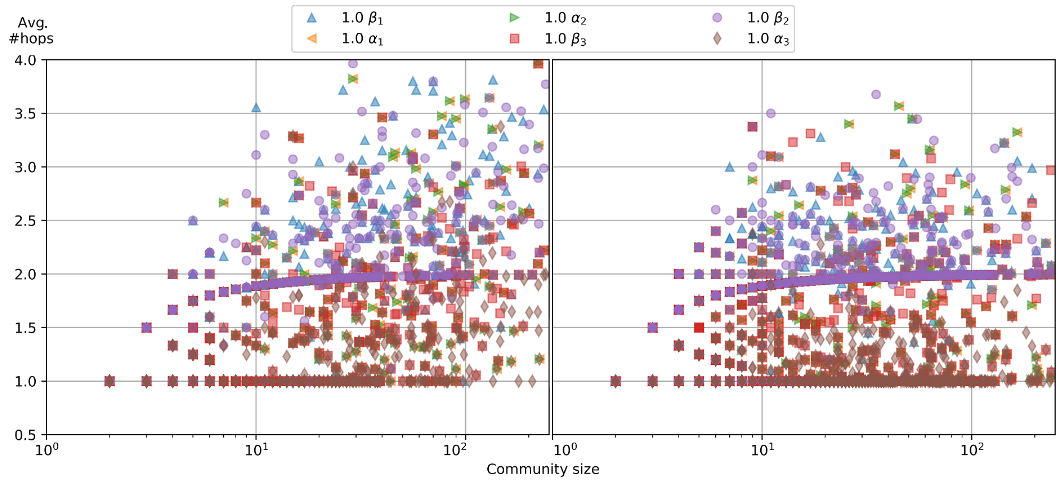


Figure 5.11: Average number of hops-to-leader plotted against a logarithmic scale of each community's size. The left image is the geographical configuration of node sets, while the right side is based on Phase One of our algorithm.

is also subject to the same behavioural user factors which influence the whole network (Vega, Cerdà-Alabern, Navarro, and Meseguer 2012). This means that it may be considered as a representative sampling of guifi.net. Secondly, the obtained number of samples is high enough to enable us to apply statistical techniques to define empirical models of bandwidth and round-trip time. This allows us to fit different distributions to the measurements and evaluate the resulting goodness-of-fit (GOF) values. Selecting the most fitting distributions, we then synthesize their parameters in order to generate functions to produce artificial observations. Qualitatively, these simulated values are representative of the behaviour of the QMP network (and thus of guifi.net) and were used to populate the bulk of our dataset (guifi.net snapshot of January, 2017) nodes, which were missing data.

Phase One: Community Detection Impact. We present in Figure 5.9 the distribution of maximum and average degree versus the size of the communities. The left side pertains guifi.net zone-based communities (from the dataset as-is), while the right side is related to the configuration of network node groups obtained with Phase One of our algorithm. We derive from this that our algorithm produces groupings with a tendency for greater node inter-connectivity. Moving on, we further evaluate this derivation by producing a visualization of the average number of hops-to-leader for each community versus community size. Figure 5.10 presents this with respect to natural geographical zones of guifi.net in the left, with our algorithm's results on the right side. Our algorithm led to an overall reduction in the number of hops, in particular for smaller and more frequent communities. Figure 5.11 highlights interesting tendencies with regard to the impact of absolute heuristic weights and their influence on the average number of hops. In particular, we achieve this by isolating the range of community sizes to a maximum size of 250 members. Plotting these ranges over a logarithmic scale, it can

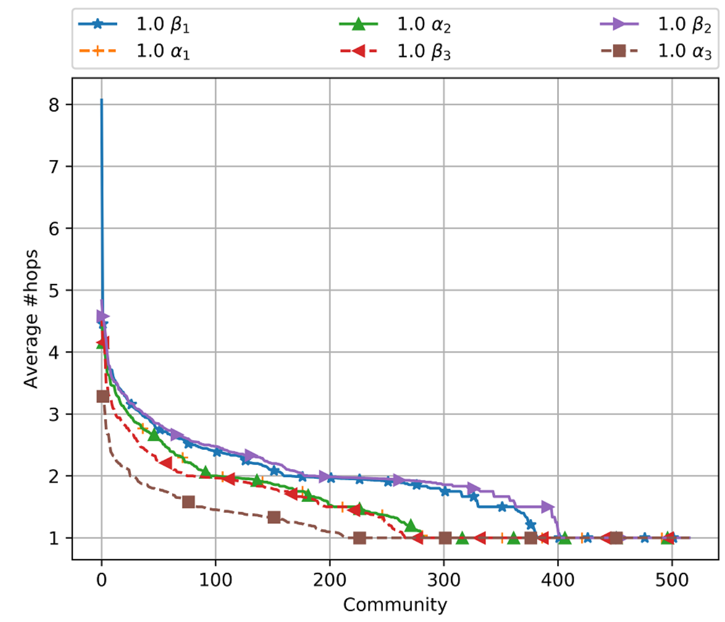


Figure 5.12: Average number of hops-to-leader against community in decreasing order for the original geographical configuration.

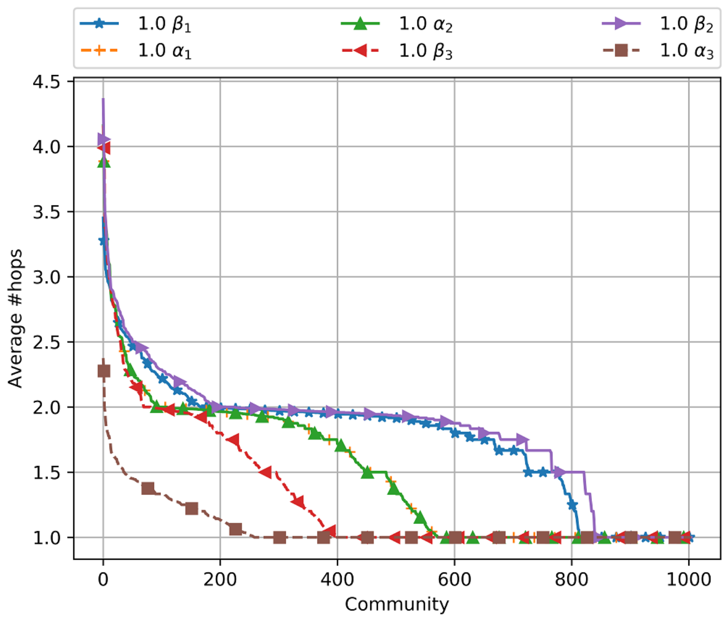


Figure 5.13: Average number of hops-to-leader against community in decreasing order for Phase One's communities.

be seen that the contained communities exhibit a lower number of hops. This tendency is particularly manifested with heuristics α_1 and β_3 (betweenness centrality and computational class of the node, respectively). We extrapolate from this finding that the fixed-region geographical definition of guifi.net may be too rigid and that it may in fact provide a user experience which is probably below-optimal regarding typical services offered in CNMCs. Usage of the Phase One technique shows promise with respect to optimizing the length of the path taken from each community's node to the community leader, a sure benefit for many services.

Phase Two: Leader Election Results. It is relevant to note that after Phase Two of our algorithm, the application of heuristics over the propagation-based node sets yielded more outliers than the geographical zones. While there were more outliers in the results of Phase One of our algorithm, lower values were achieved when compared to the geographical node groups. We presented obtained results evaluated under different criteria. Our focus is not on producing a one-size-fits-all hierarchy of heuristics: other real-world scenarios upon which to test our algorithm will have specific objective functions, bound by application needs. The results are promising as they highlight that our algorithm is a valid alternative to traditional computational approaches to optimizing responsibility assignment to network nodes. We present Figures 5.12 and 5.13, which depict the number of average hops-to-leader in decreasing order. Orthogonally to node group definitions, the tendencies in the influence of the heuristics remain valid, with the same patterns appearing for each of the cases. It is interesting to note that, for the right side (based on Phase One of our algorithm), heuristics α_2 and β_3 produced greater differences between them. Accounting for the computational class of nodes in the case of the right side led to a lower number of hops-to-leader compared to simply electing leaders based on centrality.

SLA Assessment. We also evaluate the quality of leaders in the context of the sampling performed for the QMP network. Namely, we modelled round-trip time (RTT) in milliseconds and bandwidth B/W in Mbit/s distributions based on around 70,000 samples (of bandwidth and round-trip time) obtained from QMP in guifi.net over a period of seven days. These two features are relevant to types of SLAs inherent to services such as (RTT) web caching, web content requests, NoSQL cloud storage as well as (B/W) streaming and file download services. For each community (or zone), we compute the community's RTT by summing the RTT of the path from each specific node to the elected community leader and then computing an average to represent the RTT of the community. For B/W, we perform a similar operation, but instead note for each path (from a node to the community leader) the minimum B/W value, with the community's B/W consisting of the average value of the minimum B/W recorded for each path in the community. From these two features, we modelled their distribution and simulated their values for all of the guifi.net network snapshot mentioned earlier. Figures 5.14, 5.15, 5.16, and 5.17 were produced using the `Python statsmodel` package (Seabold and Perktold 2010), which has a set of utilities to automate statistical processing tasks.

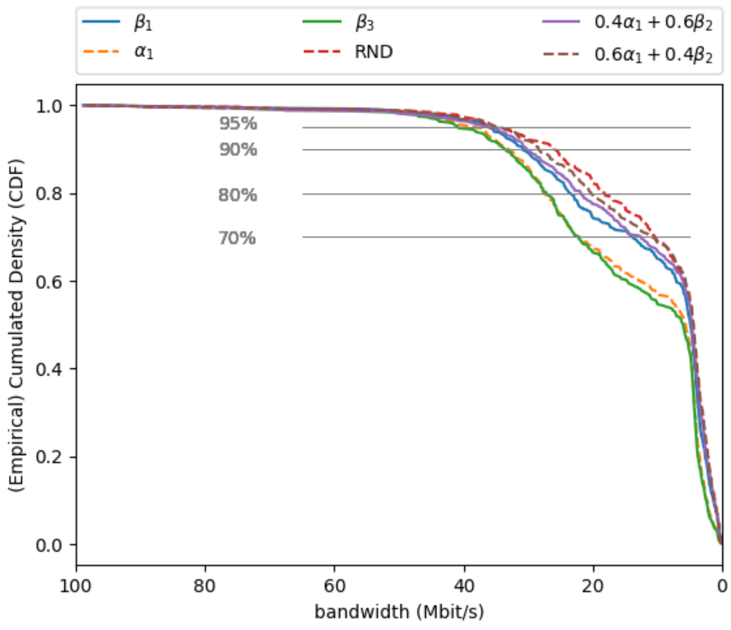


Figure 5.14: Zone-based bandwidth ECDF.

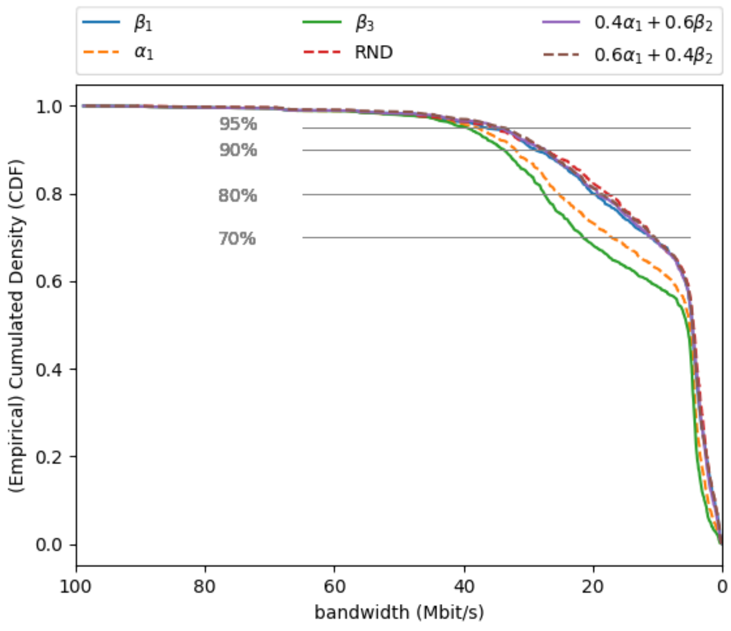


Figure 5.15: Community-based bandwidth ECDF.

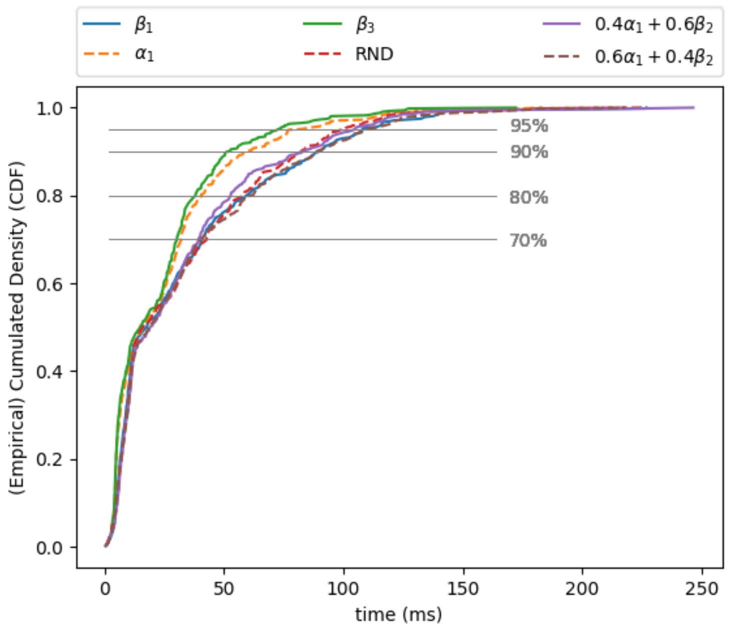


Figure 5.16: Zone-based round-trip time ECDF.

Figures 5.14 and 5.15 show the empirical cumulative distribution function of the bandwidth for the original guifi.net zones and for communities produced by Phase One of our algorithm, respectively. Interestingly, the bandwidth interval for [10; 30] Mbit/s in Figure 5.14 shows that Phase Two of our algorithm (the stage of leader election within a community – in this test case there is a one-to-one mapping between guifi.net zones and communities) fared better by using singular heuristics for electing the leader. That is, heuristics β_3 (computational class) and α_1 (betweenness centrality) produced, on average, more efficient bandwidth paths from a community's nodes to their leader. This tendency was also reproduced in the execution of Phase Two of our algorithm after Phase One (custom communities generated by Algorithm 2 instead of one-to-one mapping to guifi.net's original zones), which can be seen in Figure 5.15.

For the round-trip time, the same two heuristic weights fared better than the others as well. All combinations seemingly max out (in terms of cumulative distribution) at around 125 milliseconds. However, up to about 100 milliseconds, heuristics β_3 (computational class) and α_1 (betweenness centrality) produced lower round-trip time. This occurs for the zone-based communities in Figure 5.16 and also the communities resulting from Phase One of our algorithm, as illustrated in Figure 5.17. Curiously, we observe, as far as round-trip time is concerned, that the random leader election yielded practically the same results as the combined usage of betweenness centrality α_1 and latency β_2 (alternating between $0.4\alpha_1 + 0.6\beta_2$ and $0.6\alpha_1 + 0.4\beta_2$). We did not perform an exhaustive analysis of all possible combinations of heuristics and their weights. The combinations we present herein are relevant in terms of what the heuristics represent. Bandwidth samples were modelled as a four-parameter Kappa distribution, while round-trip time

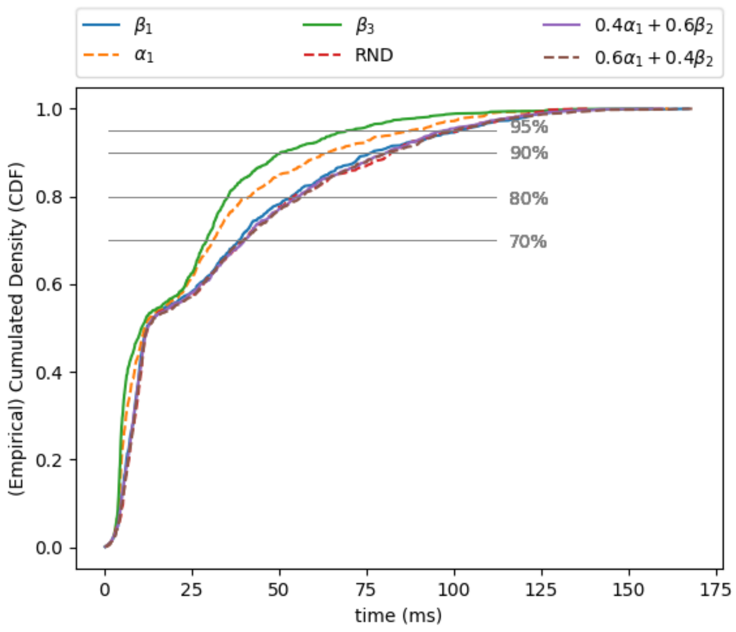


Figure 5.17: Community-based round-trip time ECDF.

was modelled as a generalized extreme value (GEV) distribution. It is relevant to say that the empirical distributions of these two metrics exhibited a considerable degree of independence. In fact, $corr(BW, R) = -0.134$ for the sampled values, which means they appear to be only slightly inversely related. We assumed them to be independent with respect to results.

Summary. The method we present is inherently parallel and distributed, a break from traditionally-centralized often exhaustive optimization-driven solutions, opening possibilities for scalability. Phase Two of our algorithm was designed to be distributed with the purpose of executing concurrently among all communities. This implies that the computational time of this phase has an upper bound associated to the slowest-computing community. As far as the authors are aware, this work is the first that attempts to optimize service placement by defining communities using an analysis based purely on network theory and distributed graph processing. The guifi.net telecommunications network is one upon which different research projects have been executed (Cerdà-Alabern 2012; Vega, Cerdà-Alabern, Navarro, and Meseguer 2012).

5.4 Related Work

Herein we go over alternative approaches to Phase One and Phase Two of our solution as a whole. We note that our work is novel, as far as we know, in the sense that it combines these two multidisciplinary phases, whose literature we analyse.

Community Networks. Different studies on guifi.net have drawn several insights:

the network is not homogeneous – rural areas have topology properties different from those of metropolitan areas, such as density; the topology observed in rural areas is not scale-free (degree distribution does not fit a power law) due to the high number of terminals connected to some nodes; removing terminal nodes (with degree one) from the graphs in rural areas, however, reveals a scale-free *core-network* as in (Vega, Cerdà-Alabern, Navarro, and Meseguer 2012). On the one hand, it is necessary to be aware of the challenges inherent to service allocation in different types of networks in the context of distributed systems. On the other hand, we highlight the existence of community detection techniques (as discussed in Chapter 2) as a novel approach to these challenges. In recent years, metrics have been proposed for evaluating the quality of calculated communities have emerged: the most notorious one being that of modularity. However, focusing exclusively on modularity incurs community resolution penalties with smaller communities often not being detected. Considering this and focusing on scalability, other methods in the literature which do not use domain-specific heuristics were devised, such as the class of label propagation algorithms (Leung, Hui, Liò, and Crowcroft 2009; Raghavan, Albert, and Kumara 2007). These algorithms are inherently parallel and work well in practice for real world networks (Boldi, Rosa, Santini, and Vigna 2011).

Service Placement. Typically, by monitoring all the physical and virtual resources on a system, service placement aims to balance load through the allocation, migration and replication of tasks. This can take place in cloud data-centres and in wireless networks that power a significant part of CNs. Most of the work in the data centre environment, including distributed data centres, is not applicable to our case because we have a strong heterogeneity given by the limited capacity of nodes and links, as well as asymmetric quality of wireless links. The authors in (Vega, Meseguer, Cabrera, and Marquès 2014) introduce a service allocation algorithm that provides near-optimal overlay allocations without the need to verify the whole solution space. They use static data from the network to identify node traits and minimize the coordination and overlay cost along a network. The work in (Novotny, Urgaonkar, Wolf, and Ko 2015) analyses network topology and service dependencies, and combined with set of system constraints determines the placement of services within the wireless network. The authors use a multi-layer model to represent a service-based system embedded in a network topology and then apply an optimization algorithm to this model to find where best to place or reposition the services as the network topology and workload on the services changes.

In distributed micro-cloud environment (i.e., similar to our case), the work of (Tärneberg, Mehta, Wadbro, Tordsson, Eker, Kihl, and Elmroth 2017) takes into account rapid user mobility and resource cost when placing applications in Mobile Cloud Networks (MCN). A recent work (Tantawi 2016) uses biased statistical sampling methods for cloud workload placement. Regarding the service placement through migration, the authors in (Wang, Urgaonkar, He, Chan, Zafer, and Leung 2017) study the dynamic service migration problem in mobile edge-clouds that host cloud-based services at the network edge. They formulate a sequential decision making problem for service migration using the framework of Markov Decision Process (MDP) and illustrate the effectiveness of their approach by simulation using real-world mobility traces of taxis in San Francisco. As a whole, service placement approaches are predominantly based on resource (CPU, memory) and node availability, and when they are network-aware, they are able just to employ static network information or at most process historical network data for availability predictions. Moreover, they are batch-oriented and execute sequentially in centralized settings and therefore cannot scale to larger network sizes, number of services, or greater network dynamism. Our approach is the first, to the best of our knowledge, that is dynamic, parallel and distributed, and therefore able scale seamlessly, by employing distributed graph processing systems, such as the `Gelly` library of `Apache Flink`. Thus, we are able to continually monitor service quality and perform service placement decisions continually/incrementally based on data gathered from the network (e.g., graph-servers in guifi.net).

5.5 Remarks

With this work, we presented a novel take on the processing steps that underlie service placement, a multi-objective problem. Compared to traditional system techniques (which, as far as we know, have not seen developments regarding parallel implementations and scalability with network size), our algorithm is expressed purely over state-of-the-art graph techniques which have inherent parallelism, making our algorithm a very competitive approach. While we evaluated GELLY-SCHEDULING with a static version of the network data, this study could also be adapted to consider more realistic dynamic scenarios, potentially making use of the techniques of VEILGRAPH.

6 Dynamic Graph Representations

The work on succinct graph representations presented in this chapter is an extended journal version submission (Coimbra, Hrotkó, Francisco, Russo, de Bernardo, Ladra, and Navarro 2021) of a publication (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020) resulting from an international collaboration with: Centro de Investigación de Galicia "CITIC" (University of A Coruña); Enxenio SL.; the Millennium Science Initiative Program (University of Chile).

We address the problem of representing dynamic graphs using k^2-trees. The k^2-tree data structure is one of the succinct data structures proposed for representing static graphs, and binary relations in general. It relies on compact representations of bit vectors. Hence, by relying on compact representations of dynamic bit vectors, we can also represent dynamic graphs. However, this approach suffers of a well known bottleneck in compressed dynamic indexing. We present a k^2-tree based implementation which follows instead the ideas by (Munro, Nekrich, and Vitter 2015) (PODS 2015) to circumvent this bottleneck. We present two dynamic graph k^2-tree implementations, one as a stand-alone implementation and another as a `C++` library. The library includes efficient edge and neighbourhood iterators, as well as some illustrative algorithms. Our experimental results show that these implementations are competitive in practice.

6.1 Introduction

Most succinct data structures for representing graphs are static (Boldi and Vigna 2004b; Brisaboa, Ladra, and Navarro 2014). Only recently, by relying on compact representations of dynamic bit vectors, succinct representations for dynamic graphs were presented (Brisaboa, Cerdeira-Pena, de Bernardo, and Navarro 2017). These representations suffer however from a well known bottleneck in compressed dynamic indexing (Munro, Nekrich, and Vitter 2015; Navarro 2016), namely that a slowdown is caused by the need to decompress components of the representation in order to perform their union. We adopt the ideas proposed by Munro *et al.* (Munro, Nekrich, and Vitter 2015) to represent dynamic graphs through collections of static and compact graph representations.

A note must be made on the meaning of certain terms. With *"succinct graph representation"* we are referring to one which, through compression techniques, enables the computational representation of a graph, while requiring less space, in a lossless way. The expression *"summary graph"* is used to describe an incomplete representation of the graph. In VEILGRAPH we build a summary graph and even compute the importance of individual vertex scores and the contribution of their edges when defining the big vertex. The summary graph is built in a one-way fashion, while our succinct graph representation (k^2-tree) can be converted to the original graph and vice-versa.

Our approach relies on k^2-trees to represent static graphs, but it supports edge insertions and removals. The edge insertion time is almost the same as the average construction time per edge of static k^2-trees. We describe the ideas behind our previous standalone implementation `sdk2tree` and compare it, together with several other k^2-tree implementations[1], with our new `sdk2sdsl` implementation[2] based on the `sdsl-lite` data structure library.[3]

Section 6.2 provides an overview of the k^2-tree data structure. In Section 6.3, we explain the technique (Munro, Nekrich, and Vitter 2015) employed to implement dynamic graphs using collections of static k^2-trees and the details of our library implementation `sdk2sdsl`. We present extensive experimental analysis in Section 6.4 and final remarks in Section 6.5.

6.2 The Static k^2-tree

Let $G = (V, E)$ be a graph where V is the set of vertices, of size n, and $E \subseteq V \times V$ is the set of edges, of size m. The k^2-tree data structure provides a static succinct representation of G (Brisaboa, Ladra, and Navarro 2014). At a high level, this data structure corresponds to an adjacency matrix representation, where a bit set to 1 indicates the existence of an edge and a bit set to 0 its absence. To reduce the space requirements for sparse graphs, a hierarchical decomposition of the matrix is used, where a sub-division consisting only of zeros is represented by a single 0 bit.

More concretely, the k^2-tree can be conceptually defined as a non-balanced k^2-ary tree that represents the recursive partition of the adjacency matrix into $k \times k$ sub-matrices. The root node contains k^2 children, each of them corresponding to one sub-matrix and sorted following a Z-order. The nodes of the tree store just one bit indicating if the sub-matrix is non-empty (1) or if it is all zeroes (0). Then, the non-empty sub-matrices are subdivided again until reaching an empty sub-matrix, or until no more subdivision is possible; thus, bits at last level of the tree correspond to cell values of the original adjacency matrix. The resulting tree is thus of height $\lceil \log_{k^2} n^2 \rceil = \lceil \log_k n \rceil$.

An example of this tree-shaped representation is shown in Figure 6.1. This conceptual tree is stored in one single bitmap following a level-wise traversal of the tree (i.e., concatenating the 4 bitmaps of the figure). Queries over the graph can be solved efficiently by performing top-down traversals over the tree representation. Those traversals are efficiently implemented thanks to the use of fast rank operations (Navarro 2016) over the bitmap.

The maximum length of this bitmap is $k^2 m (\log_{k^2} \frac{n^2}{m} + \mathcal{O}(1))$. A sub-linear number of extra bits are needed to enable constant-time rank operations on the bitmaps. Testing

[1] https://github.com/aplf/sdk2tree
[2] https://github.com/joo95h/dynamic_k2tree
[3] https://github.com/simongog/sdsl-lite

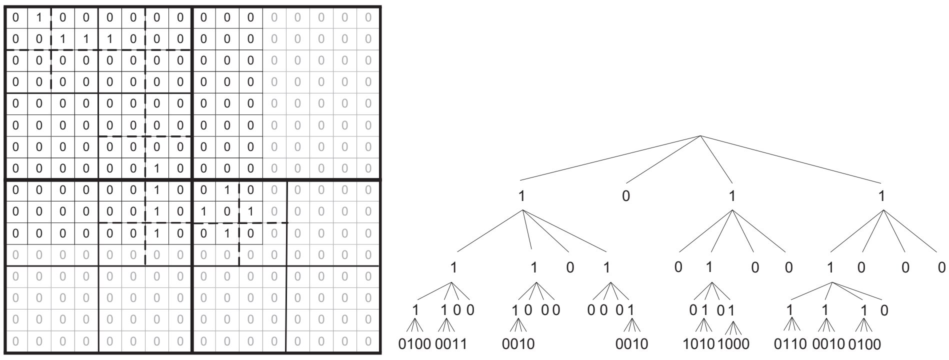


Figure 6.1: Example of adjacency matrix (left) and its corresponding k^2-tree with $k = 2$ (right).

the existence of an edge is done in $\mathcal{O}(\log_k n)$ time by traversing the k^2 down to the desired matrix cell, until an empty sub-matrix (a 0) is found or we reach the 1 of the cell in the last level. Obtaining the neighbours of a node is done in $\mathcal{O}(n)$ worst-case time by reaching all the cells in the corresponding matrix row, via entering all the children of each node that intersect the row; the reverse neighbours are obtained similarly by extracting the corresponding matrix column. In this paper, k is a fixed value across the trees and remains constant along time.

6.3 From Static k^2-trees to Dynamic Graphs

The main idea to represent G dynamically, supporting edge insertions and deletions, as well as listing the neighbours of a given vertex v, is to use a collection of static edge sets $\mathcal{C} = \{E_0, \ldots, E_r\}$. Each static edge set E_i is then represented using a static k^2-tree, except E_0 which is represented as a dynamic and uncompressed adjacency list. Figure 6.2 depicts a link query over the different E_i sets of the data structure.

Let m_i be the number of edges in each set E_i. As discussed by Munro *et al.* (Munro, Nekrich, and Vitter 2015), we must control both the number of edges m_i in each set E_i and the number r of such sets to achieve the optimal amortized cost for each operation. The first set (E_0) contains at most $m/\log^2 n$ elements. In general we require that m_i is at most $m/\log^{2-i\varepsilon} n$, for some constant $\varepsilon > 0$. We must also have that $r \leq 2/\varepsilon$, so when ε is a fixed constant so is r. For example when $\varepsilon = 1/4$ we get that r is at most $2/(1/4) = 8$. Hence the maximum number of edges per static set follows a geometric progression. Each set E_i is static (except for E_0) and has a maximum allowed size $m/\log^{2-i\varepsilon}$. Whenever we reach the maximum size for E_0 (overflow), we find a set E_j, with $i < j \leq r$ such that $\sum_{\ell=0}^{j} m_\ell \leq m/\log^{2-j\varepsilon} n$ and (re)build E_j with all edges in it and in the previous sets, and reset all previous sets to empty. By construction, E_j has a maximum capacity enabling it to store the content of sets E_0 through E_{j-1}. We detail this process below.

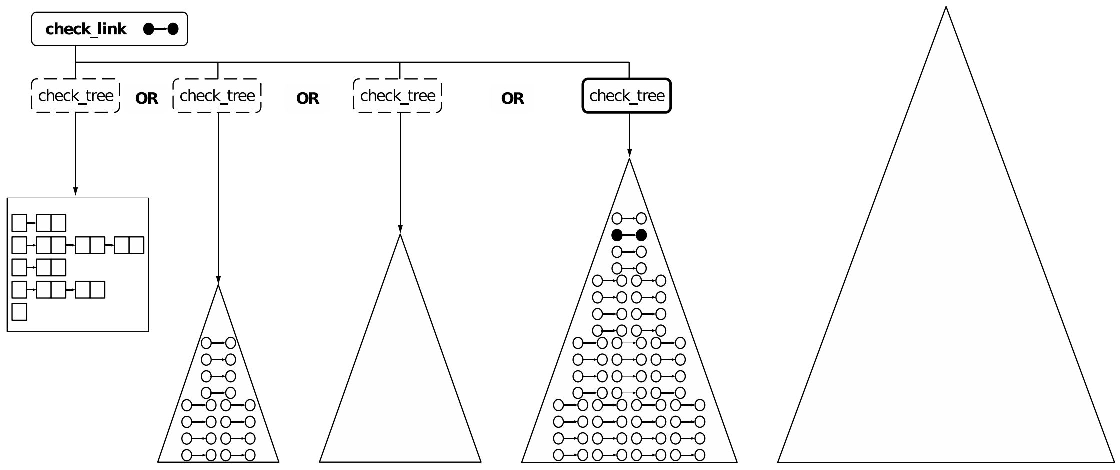


Figure 6.2: `check_link` query going through E_i sets to find an edge (depicted in black). The algorithm first checks the E_0 structure which consists of a dynamic and uncompressed adjacency list (represented by the square on the left side). If it does not find the edge, the following static k^2-tree data structures are iterated in order of growing size: $E_1, E_2, ..., E_r$ until the edge is found. We represent this sequence of data structure checks by the composition of boolean `OR` operations seen in the image.

6.3.1 Space

Let us analyse the required space to represent the data structure. The set E_0 is represented in an uncompressed adjacency list coupled with a hash table to allow answering queries on edge existence in constant expected time. This requires $\mathcal{O}(m_0 \log n)$ bits, where $m_0 \leq m/\log^2 n$ is the number of edges in E_0. Each set E_i, for $1 \leq i \leq r$, is represented with a static k^2-tree requiring $k^2 m_i \left(\log_{k^2}(n^2/m_i) + \mathcal{O}(1)\right)$ bits (plus sublinear-order terms to support rank), where $m_i \leq m/\log^{2-i\varepsilon} n$. Hence, overall, the space required is

$$\mathcal{O}(m_0 \log n) + \sum_{i=1}^{r} k^2 m_i \left(\log_{k^2}(n^2/m_i) + \mathcal{O}(1)\right)\ (1 + o(1)) \tag{6.1}$$

bits. The first term in Equation 6.1 is sublinear, $\mathcal{O}\left((m/\log^2 n)\log n\right) = \mathcal{O}\left(m/\log n\right)$. We now bound the main part of second term as follows, exploiting the fact that the formula is monotonic on every m_i:

$$\sum_{i=1}^{r} k^2 m_i \log_{k^2}\frac{n^2}{m_i} \;\leq\; k^2 \sum_{i=1}^{r} \frac{m}{\log^{2-i\varepsilon} n} \log_{k^2}\left(\frac{n^2}{m}\log^{2-i\varepsilon} n\right)$$

$$= \frac{k^2 m}{\log^2 n} \sum_{i=1}^{r} \log^{i\varepsilon} n \left(\log_{k^2}\frac{n^2}{m} + (2 - i\varepsilon)\log_{k^2}\log n\right). \tag{6.2}$$

We can set $r = 2/\varepsilon$ because the sum is monotonic on r. Then, because

$$\sum_{i=1}^{r} \log^{i\varepsilon} n \;=\; \log^{r\varepsilon} n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right) \;=\; \log^2 n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right), \text{ and}$$

$$\sum_{i=1}^{r} i \log^{i\varepsilon} n \;=\; r \log^{r\varepsilon} n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right) \;=\; r \log^2 n \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right),$$

Equation (6.2) is

$$k^2 m \left(\log_{k^2} \frac{n^2}{m} \left(1 + \mathcal{O}(\log^{-\varepsilon} n)\right) + (2 - r\varepsilon + \mathcal{O}(\log^{-\varepsilon} n)) \log_{k^2} \log n\right)$$

$$= k^2 m \log_{k^2}(n^2/m)(1 + o(1)).$$

The whole Equation (6.1), since $\sum_{i=1}^{r} m_i \leq m$, is then upper bounded by

$$k^2 m(\log_{k^2}(n^2/m) + \mathcal{O}(1))(1 + o(1)),$$

which asymptotically coincides with the space of the static k^2-tree representation, even considering the sublinear extra space to support rank operations.

6.3.2 Insertion, Deletion and Queries

We rely on efficient set operations over k^2-trees (Quijada-Fuentes, Penabad, Ladra, and Gutiérrez 2019). Given C and C' represented as two k^2-trees, we are able to compute k^2-trees representing $C \cup C'$, $C \cap C'$ and $C \setminus C'$ in linear time on the size $|C|$ and $|C'|$ of the representations. Moreover these operations are done without decompressing C and C', with only some negligible extra space being used.

Insertion works as follows. Given a new edge (u, v), if $|E_0| < m_0$, then just add (u, v) to E_0 and we are done; otherwise, build a k^2-tree for E_0, find $0 < j \leq r$ such that $\sum_{i=0}^{j} m_i \leq m/\log^{2-j\varepsilon} n$, and rebuild E_j with all edges in $E_0, \ldots, E_j$ by successive unions of k^2-trees. Figure 6.3 illustrates the process.

If $|E_0| < m_0$, then insertion takes constant expected time since we are relying on an adjacency list coupled with a hash table to maintain adjacencies, as described before. Otherwise, we need to build a k^2-tree to contain the edges in E_0, which requires finding some E_j to accommodate all previous collections E_i, for $0 \leq i \leq j$. Note that the construction of the k^2-tree for E_0 takes $\mathcal{O}(m_0 \log_k n)$ time (Brisaboa, Ladra, and Navarro 2014), and the pairwise union of at most j k^2-trees representing collections $E_0 \ldots E_{j-1}$ takes $\mathcal{O}(m_j \log_k n)$ time, using only the required space to store a k^2-tree representing E_j. The amortized analysis of the insertion cost follows the argument presented by Munro *et al.* (Munro, Nekrich, and Vitter 2015) for the general case. Either E_j is new and m has at least doubled, in which case the amortized cost is $\mathcal{O}(\log_k n)$ per edge insertion, or E_j is not new and we are adding to it all edges in collections $E_0, \ldots, E_{j-1}$. In this last case the building cost can be charged to the new edges added to E_j, which are at least $m/\log^{2-(j-1)\varepsilon} \geq m_j/\log^{\varepsilon} n$. Therefore, the amortized cost of inserting an

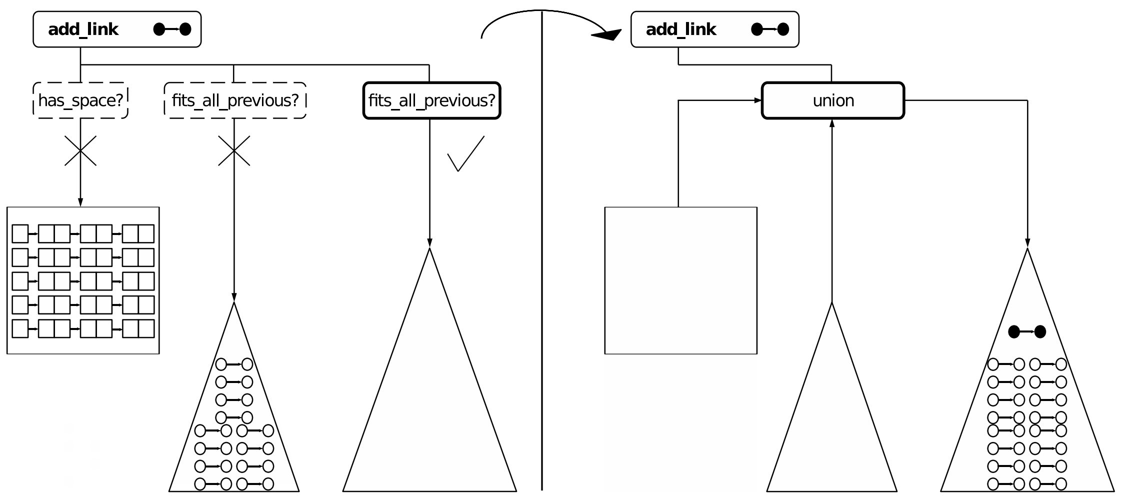


Figure 6.3: `add_link` searching for j such that E_j has enough space to hold all edges of sets $E_i, i < j$ because E_0 was full.

edge in E_j is $\mathcal{O}(\log_k n \log^\varepsilon n)$ and, since each edge can be moved once to each E_j, with $0 < j \leq r = \lfloor 2/\varepsilon \rfloor$, the amortized cost of inserting an edge is $\mathcal{O}((1/\varepsilon) \log_k n \log^\varepsilon n)$. This is then the overall amortized cost of inserting an edge.

Deletion works as follows. Given an edge $(u, v) \in E$, if $(u, v) \in E_0$, then just remove it and we are done; otherwise, find $0 < j \leq r$ such that $(u, v) \in E_j$ and, if there is such j, set the corresponding bit to zero in E_j k^2-tree, update the number m' of deleted edges, and if $m' > m/ \log \log n$, rebuild $\mathcal{C}$.

Deleting an edge in E_0 takes constant expected time. Checking and deleting an edge in our collections takes $\mathcal{O}((\log_k n)/\varepsilon)$, since checking if an edge exists in a given k^2-tree takes $\mathcal{O}(\log_k n)$ (Brisaboa, Ladra, and Navarro 2014), and we might have to look in each collection E_i, with $0 < i \leq r = \lceil 2/\varepsilon \rceil$. Once an edge is found, setting the corresponding bit to 0 in the static k^2-tree takes constant time. Note that there is a bit set to 1 for each edge in a k^2-tree. We are just exploiting that fact when we delete an edge and, hence, we do not use extra space. This means in particular that the k^2-tree data structure remains the same, we are not updating it and any query for a deleted edge will reach the corresponding leaf of the tree, which now is marked as 0. The full rebuild after $m/ \log \log n$ edges are deleted costs $\mathcal{O}(m \log_k n)$, *i.e.*, it has an amortized cost of $\mathcal{O}(\log_k n \log \log n)$ per deleted edge. Overall deleting an edge has then an amortized cost of $\mathcal{O}((1/\varepsilon + \log \log n) \log_k n)$.

Querying works just as in k^2-trees with the difference that we need to query all sets in the collection. Therefore, the querying cost increases by a factor of $\mathcal{O}(1/\varepsilon)$.

6.3.3 Graph Library

The library proposed in the context of this work exposes an API supporting also edge and neighbourhood iterators. This API was built having in mind an easy and familiar interface, compared to other libraries such as: `SNAP`[4], `igraph` (Csardi, Nepusz, et al. 2006), among others. And the library was built after the `sdk2tree` project, but the underlying k^2-tree implementation is based on the `sdsl-lite` k^2-tree implementation, which uses static bit vectors, in `C++`.

The two types of iterators present a similar interface: `edge_begin`, `edge_end` for edges and `neighbour_begin(x)`, `neighbour_end()` for neighbours. The neighbour iterator receives the node whose neighbourhood is desired. The iterators may be used as other iterators in `C++` and follow its iterator pattern. If there are no edges to iterate then `edge_begin()=edge_end()`. The iterators do not return edges in any particular order, since they first iterate over the E_0 container and then over each E_i k^2-tree.

6.3.3.1 Edge Iterator

As previously mentioned, the edge iterator iterates over the container E_0 and then over the k^2-tree for each E_i. In the uncompressed container E_0, it takes linear time to retrieve all its edges. However, for each k^2-tree, it relies on a k^2-tree edge iterator and it takes time proportional to the size of the k^2-tree. This iterator is implemented by saving in a queue all states where the search was still not finished in a depth-first approach over the k^2-tree. Thus, this queue has at most size $\mathcal{O}(\lfloor \log(V)/\log(k) \rfloor)$ which is the maximum level of the k^2-tree. Iterating over edges in a k^2-tree is performed then by visiting internal nodes and, if there are any children, then we check in all the k^2 children of that node; otherwise we backtrack. When we reach the last level, we check if the bit position is 1, which means we have found an edge and we return it.

6.3.3.2 Neighbourhood Iterator

The neighbourhood iterator is very similar to the edge iterator in the sense that we keep a queue of the states from where we last evaluated each node of the tree. If there are neighbours of node x in the first container E_0, it iterates over them first. Once the uncompressed container is iterated, it goes to the k^2-tree collections. Similarly to the edge iterator, it follows a depth-first search keeping a queue with the incomplete searched states. The neighbour iterator follows the same algorithm from the listing neighbours operation from the k^2-tree, however it saves the state from all the incomplete searched states. The running time for listing all neighbours of a given vertex is the same of the listing neighbours operation.

6.3.4 Comparison with Other Constructions

Given a graph G, for a fixed ε, the presented data structure uses essentially the same space as a static k^2-tree, and supports insertions and deletions in $\mathcal{O}(\log_k n \log^{\varepsilon} n)$ and $\mathcal{O}(\log_k n \log\log n)$ amortized time, respectively. The implementation of dynamic k^2-trees using dynamic bit vectors (Brisaboa, Cerdeira-Pena, de Bernardo, and Navarro 2017) requires a small space overhead, and it supports insertions and deletions in

[4] `http://snap.stanford.edu`

$\mathcal{O}(\log_k n \log n)$ time, which implies a slowdown factor of $\Theta(\log^{1-\varepsilon} n)$ with respect to the proposed data structure.

Edge queries over the proposed data structure take the same time as in static k^2-trees. Although dynamic k^2-trees using dynamic bit vectors (Brisaboa, Cerdeira-Pena, de Bernardo, and Navarro 2017) work similarly to static k^2-trees, they run on dynamic bit vectors, thereby having a slowdown of $\Omega(\log n/\log\log n)$ (Navarro 2016, Chapter 12).

We also compare our approach with a new representation, k^2-tries, proposed recently (Arroyuelo, de Bernardo, Gagie, and Navarro 2019). This data structure uses $\mathcal{O}(m \log(n^2/m)+m \log k)$ bits, and supports edge queries in $\mathcal{O}(\log_k n)$ time and updates in $\mathcal{O}(\log_k n)$ amortized time. The implementation provided by the k^2-tries authors supports only edge additions and queries, with slightly worse time complexities.

6.4 Experimental Analysis

In previous work (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020), we presented `sdk2tree`, our dynamic k^2-tree `C` implementation based on the techniques proposed by Ian Munro *et al.* (Munro, Nekrich, and Vitter 2015). We now also introduce a `C++` library version named `sdk2sdsl` which is based on the `sdsl-lite` data structure library. We present experimental analysis comparing different implementations: our new `sdk2sdsl` library version; our previous dynamic graph `sdk2tree` implementation (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020); the dynamic graph `dk2tree` based on dynamic bit vectors (Brisaboa, Cerdeira-Pena, de Bernardo, and Navarro 2017); two dynamic graph implementations (differing only on the parametrization to trading compression for speed) `k2trie{1,2}` based on dynamic tries (Arroyuelo, de Bernardo, Gagie, and Navarro 2019); the original static bit vector implementation `k2tree` (Brisaboa, Ladra, and Navarro 2014). We make available the source code for all implementations as well as usage instructions at `https://github.com/aplf/sdk2tree`. Our new `sdk2sdsl` implementation was written in `C++` and the others are in `C`, with every implementation single-threaded and compiled with `gcc 7.5.0` using the `-O3` optimization flag. Experiments were performed on an 8-core AMD Ryzen 7 2700X Eight-Core Processor @ 2.04GHz machine with 32K L1d cache, 64K L1i cache, 512KB L2 cache, 8192K L3 cache and system memory of 64 GB RAM. We implemented a common interface to test each implementation. All dynamic data structures `dk2tree`, `sdk2tree`, `sdk2sdsl` and `k2trie{1,2}` are initialized empty. The static `k2tree` is initialized by reading the whole graph from secondary storage. Once initialized, the interface starts a main loop which reads instructions from `stdin` representing all supported edge operations, with additions and deletions not available in `k2tree`, and `k2trie{1,2}` supporting only edge additions and queries. We also implemented and tested known graph algorithms on our `sdk2sdsl` implementation: breadth-first search (BFS), depth-first search (DFS), global clustering coefficient and variants of triangle counting.

Table 6.1: Bit/edge ratio (post-serialization) is presented for each data structure. First four datasets were synthetically generated using a duplication model. Last four datasets are real-world Web graphs made available by the Laboratory for Web Algorithmics (LAW) (Boldi and Vigna 2004b; Boldi, Rosa, Santini, and Vigna 2011) (`uk-2007-05` is actually `uk-2007-05-100000` in the LAW website).

Dataset	$\|V\|$ (M)	$\|E\|$ (M)	`k2tree` (bit/edge)	`dk2tree` (bit/edge)	`sdk2tree` (bit/edge)	`sdk2sdsl` (bit/edge)	`k2trie1` (bit/edge)	`k2trie2` (bit/edge)
`dm50K`	0.05	1.11	21.10	23.64	21.26	25.26	43.16	298.99
`dm100K`	0.10	2.59	22.66	25.27	22.76	27.16	47.31	257.61
`dm500K`	0.50	11.98	27.87	30.85	27.97	33.31	57.92	187.91
`dm1M`	1.00	27.42	29.48	32.63	29.49	35.33	58.78	132.92
`uk-2007-05`	0.10	3.05	2.98	3.39	3.16	3.63	5.62	11.11
`in-2004`	1.38	16.92	2.99	3.40	3.14	3.64	3.90	6.97
`uk-2014-host`	4.77	50.83	9.47	10.55	9.58	11.42	13.07	21.88
`indochina-2004`	7.42	194.11	2.46	2.79	2.59	3.00	2.88	4.91
`eu-2015-host`	11.26	386.92	5.61	6.26	5.71	6.74	7.02	11.64

6.4.1 Datasets and Methodology

We use both real and synthetic datasets. In Table 6.1 we identify the datasets and their properties. For each dataset, we present its vertex and edge counts written as $|V|$ and $|E|$, respectively, and bits per edge (after serialization) for each implementation.

Real-world graphs were obtained from the Laboratory of Web Algorithmics[5] (Boldi and Vigna 2004b; Boldi, Rosa, Santini, and Vigna 2011). Synthetic datasets were generated from the partial duplication model (Chung, Lu, Dewey, and Galas 2003). Although the abstraction of real networks captured by the partial duplication model, and other generalizations, is rather simple, the global statistical properties of, for instance, biological networks and their topologies can be well represented by this kind of model (Bhan, Galas, and Dewey 2002). We generated random graphs with selection probability $p = 0.5$, which is within the range of interesting selection probabilities (Chung, Lu, Dewey, and Galas 2003). The number of edges for those graphs is approximately 25 times the number of vertices.

We should note that bits per edge for real datasets in Table 6.1 are affected by the natural order of vertices, given in that case by URL lexicographic order, which favours Web graph compressibility. If ids were randomly assigned to vertices, then the bits per edge would be similar to those observed for random synthetic graphs.

We consider four major operations: edge additions, removals, querying/checking and vertex neighbourhood listing. Elapsed time was measured using the `clock()` function[6]. Each time and memory result is the average of 5 individual executions. Although the `k2tree` implementation does not support additions, we include it in the

[5] http://law.di.unimi.it/datasets.php
[6] http://man7.org/linux/man-pages/man3/clock.3.html

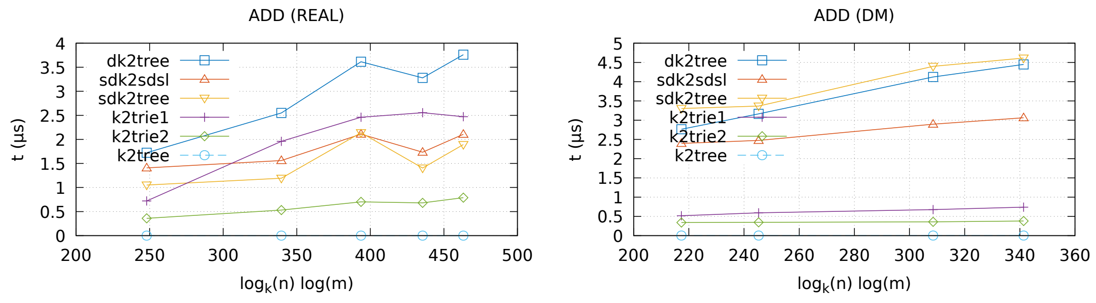


Figure 6.4: Average time taken for adding an edge in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

comparison. For that we build a k2tree for each dataset and divide the time it takes by the number of edges, obtaining the average construction time per edge. This allowed us to evaluate the overhead introduced by dynamic data structures. The removal operation is compared between sdk2tree, sdk2sdsl and dk2tree. This operation was evaluated by adding all edges and removing a sample of 50% of them. All implementations except the one based on dynamic tries were directly compared for the listing operation. After adding all edges, we evaluated this operation by asking for the neighbourhoods of a sample of 50% of the vertices. We measure for each implementation the average time per individual operation, the maximum resident set size (memory peak was obtained with GNU time[7]), and the disk space taken by the serialization of data structures.

6.4.2 Cost Analysis

Let us analyse the cost of each operation over the different datasets and for the different implementations. Figure 6.4 shows the average running time for adding an edge. As mentioned before, we include k2tree in this comparison to observe the slowdown introduced by dynamic data structures. As expected, dynamic implementations take in general more time per add operation than k2tree. As expected also from the theoretical analysis, the add operation on sdk2tree is faster than on dk2tree, in particular for real Web graphs. The library version sdk2sdsl is faster or as fast as sdk2tree for this operation.

Figure 6.5 shows the average running time for removing an edge. For both dataset types, dk2tree was slower than others. The sdk2tree and sdk2sdsl implementations achieved close execution times and similar behaviour among datasets, with the library implementation sdk2sdsl being faster. We note that costs seem to correlate well with the predicted bounds.

Figures 6.6 and 6.7 show the average running time for listing vertex neighbourhoods and querying/checking edges. Across all datasets, sdk2tree was faster than

[7]https://www.gnu.org/software/time/

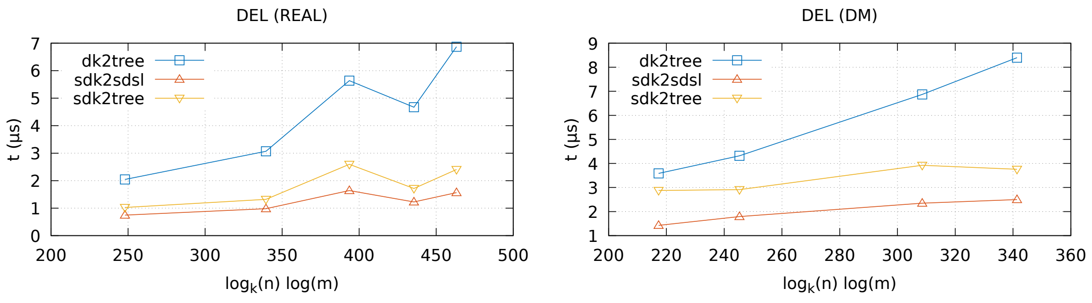


Figure 6.5: Average time taken for deleting an edge in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

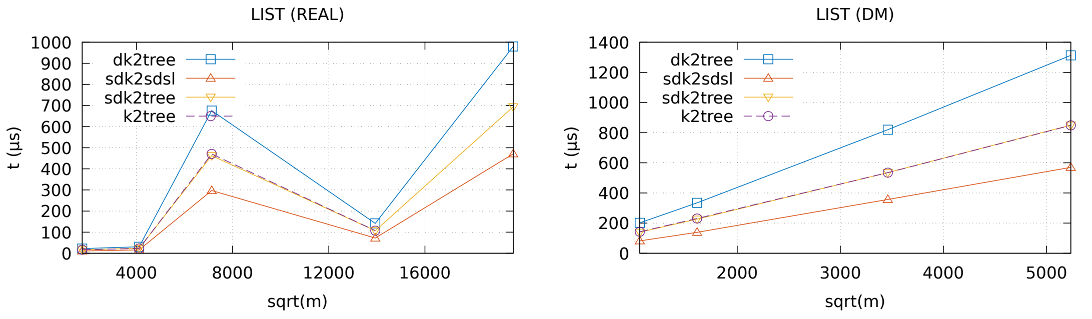
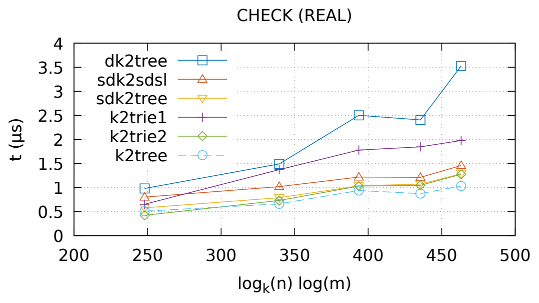


Figure 6.6: Average time taken for listing neighbours of random vertices in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

`dk2tree` and on-par with `k2tree` and `k2trie`{`1,2`}. In the case of listing, we are plotting against $\mathcal{O}(\sqrt{m})$, the average-case bound on the cost of listing vertex neighbourhoods with `k2tree` (Brisaboa, Ladra, and Navarro 2014). This bound is valid also for `sdk2tree` and `dk2tree` as discussed previously in the theoretical analysis. `dk2tree` (dynamic bit vectors) was slower than `sdk2tree` which was matched with the static `k2tree` implementation. Our `sdk2sdsl` library version was consistently the fastest for the listing operation. For the edge query operation, `dk2tree` was constantly the slowest implementation, with the others coming very close.

Let us now analyse how much memory is used by each implementation. In this analysis we consider resident memory while we are performing operations. For the space that each data structure takes once serialized on secondary memory, we refer the reader to Table 6.1. We note that our `sdk2sdsl` library implementation serialized format has a higher ($\approx 20\%$) bits/edge ratio compared to `sdk2tree`. This seems to be related to using a 64-bit index for the data structures.

Figure 6.8 shows the maximum resident memory while adding edges in dynamic implementations. We can observe that `sdk2tree` requires more memory than `dk2tree`, although the growth rate is similar. This can look unexpected given the theo-



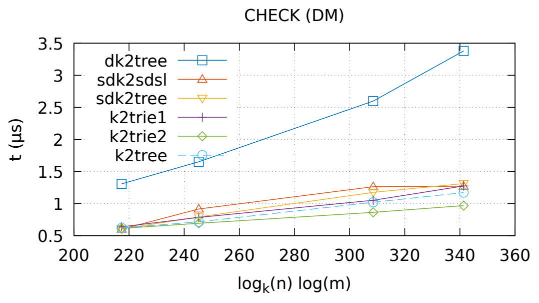
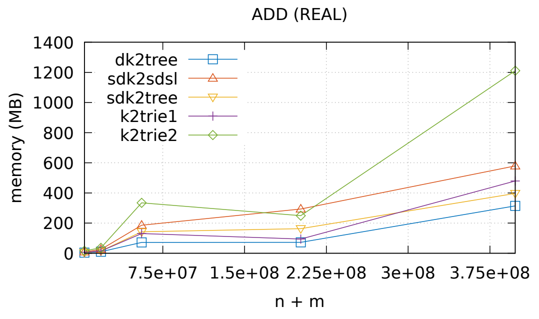


Figure 6.7: Average time taken for querying edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.



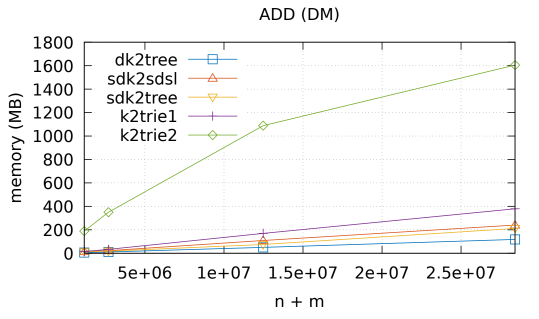


Figure 6.8: Max. resident memory while adding edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

retical bounds derived previously, but we must recall that we are periodically merging together static collections in the `sdk2tree` implementation. The `sdk2sdsl` implementation followed the same pattern as `sdk2tree`, albeit consuming more memory than `sdk2tree`. Note that we use 64 bit integers in `sdk2sdsl` and 32 bit integers in `sdk2tree`.

Figure 6.9 shows the maximum resident memory while removing edges. Since we are adding all edges before removing about 50% of them, the memory requirements for `sdk2tree` are exactly the same as in Figure 6.8. This also means that the edge removal operation does not increase the space requirements in this implementation. `sdk2sdsl` consumed more memory than `sdk2tree`, with the memory requirements for `dk2tree` being the lowest on this operation.

Figure 6.10 shows the maximum resident memory while adding edges and listing vertex neighbourhoods. Since we are adding all edges as before, the memory requirements for `sdk2tree`, `sdk2sdsl` and `dk2tree` are identical to those observed in Figures 6.8 and 6.9. We include now also the static `k2tree` in our analysis. We should note however that once constructed, `k2tree` requires much less space as shown in Table 6.1. For instance, for the dataset `dm100K`, `k2tree` had a peak resident memory footprint of

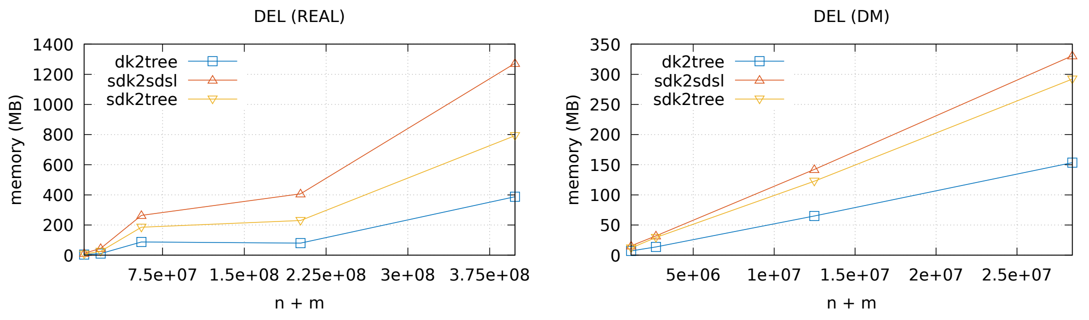


Figure 6.9: Max. resident memory while deleting edges in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

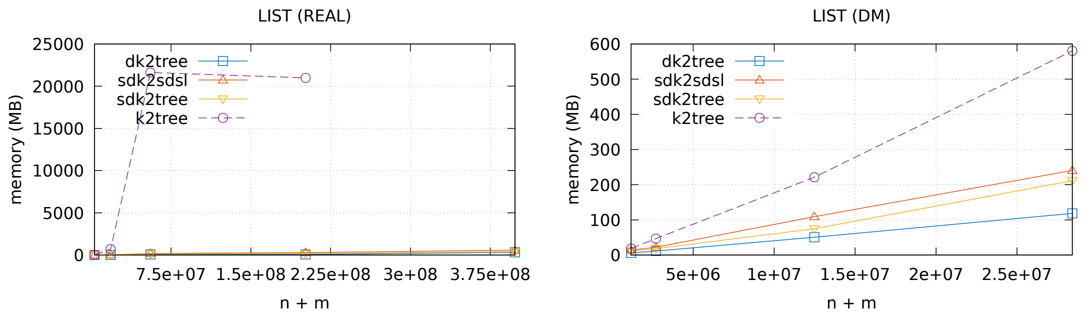


Figure 6.10: Max. resident memory while listing neighbours of random vertices in real Web graphs and in synthetic graphs (generated from a duplication model), respectively.

around 503.11 MB during construction, while its k^2-tree structure stored on disk uses 22.66 bits per edge, i.e, a total of 7.01 MB. Although we are using the exact same implementation of k^2-trees for representing the static collections within our `sdk2tree` and `sdk2sdsl` implementations, we do not observe such a high memory footprint while adding edges in our implementations. This highlights the fact that we are merging those collections without decompressing them as mentioned before.

6.4.3 Graph Library Performance

We implemented some well known graph algorithms over `sdk2sdsl`, for which we compare consumed memory and execution time against expected theoretical results. For each algorithm, we present in Table 6.2 the running time and peak resident memory usage. Each cell holds the ratio of observed value to corresponding theoretical complexity. For example, the value of the cell of the first row and first column on the top represents the execution time (nanoseconds) of the `sdk2sdsl` breadth-first search algorithm (applied to dataset `dm50K`) divided by its theoretical temporal complexity of $\mathcal{O}(n\sqrt{m}+m)$. We omit dataset `indochina-2004` from the graph algorithm tests for `sdk2sdsl` as its topology does not allow for an adequate assessment of algorithms

Table 6.2: Ratios between observed values and corresponding theoretical graph algorithm complexities. The top part is the execution time ratio (nanoseconds) and the bottom part is the peak resident memory ratio (bytes).

Time ratio (ns)	`BFS` $n\sqrt{m}+m$	`DFS` $n\sqrt{m}+m$	`CC` $m\sqrt{m}$	`CT (hash)` $m\sqrt{m}$	`CT (neighbour)` $m\sqrt{m}\log_k n \log m$
`dm50K`	66.2884	67.3630	6.4365	0.0474	0.4566
`dm100K`	74.2812	73.8288	5.5574	0.0386	0.4350
`dm500K`	82.8276	84.9108	5.7892	0.0214	0.3033
`dm1M`	88.1807	91.0187	5.2030	0.0203	0.2773
`uk-2007-05`	5.0637	5.0362	0.8456	0.0173	N/A
`in-2004`	2.7018	2.6811	0.7807	0.0039	N/A
`uk-2014-host`	13.5329	13.2120	2.3418	0.0745	N/A
`eu-2015-host`	10.3745	10.1891	0.4817	0.0007	N/A
Memory ratio (B)	`BFS` $n+m$	`DFS` $n+m$	`CC` $n+m$	`CT (hash)` $n+m$	`CT (neighbour)` $n+m$
`dm50K`	16.465	16.526	78.771	75.848	14.581
`dm100K`	11.015	11.017	74.837	72.110	9.231
`dm500K`	12.744	12.748	72.662	70.569	10.561
`dm1M`	11.112	11.113	70.923	68.940	10.813
`uk-2007-05`	4.715	4.149	69.023	67.497	N/A
`in-2004`	6.119	4.970	74.324	70.670	N/A
`uk-2014-host`	7.686	6.596	72.019	67.283	N/A
`eu-2015-host`	3.174	2.625	13.585	12.442	N/A

expected efficiency.

The first column of Table 6.2 shows the behaviour of breadth-first search (BFS). For the time ratio, the implementation is such that the observed execution times increase by small amounts compared to the growing dataset sizes, with the peak resident memory values being more intimately connected to the topology of the datasets. Note the $\sqrt{m}$ due to the cost of listing of neighbourhoods.

The second column of Table 6.2 shows the behaviour of depth-first search (DFS). It has a behaviour similar to BFS for all dataset graph types (for both time and memory), as expected.

For the (global) clustering coefficient (CC), the observed time ratios highlight the influence of graph density. This is shown with dataset `uk-2014-host`, whose ratio of 2.3418 (ns) is around 3x greater than the time ratio of the smaller dataset `in-2004` and close to 5x greater than the time ratio of the bigger `eu-2015-host`. The peak resident memory ratios for CC are more closely related to graph structure, with `eu-2015-host` (biggest of the tested web graphs) achieving a memory ratio 5x lower than `uk-2007-05` (smallest of the tested web graphs).

Note that we use a classic algorithm for computing both the clustering coefficient and counting triangles. This algorithm iterates over all edges (u, v) and, without loss of generality, it iterates over the neighbourhood of u, checking if each neighbour w of u

is such that an edge (w, v) exists in the graph, where edge existence is checked against a hash table with all edges. Neglecting heavy hitters, *i.e.* vertices with more than $\sqrt{m}$, neighbours which are uncommon for large scale-free networks, the expected running time is $\mathcal{O}(m\sqrt{m})$. We can observe in Table 6.2 the third (CC) and fourth (CT hash) columns of the memory ratio section. Their memory ratio values are similar.

Since we can answer queries on edge existence with our proposed data structures in $\mathcal{O}(\log n \log m)$ time, we implemented an algorithm for counting triangles using edge queries directly against the data structure, without relying on a hash table. Note that the expected running time becomes now $\mathcal{O}(m\sqrt{m} \log n \log m)$ since we can no longer have edge queries in expected constant time. But now we need much less memory since we do not need a hash table to track edges, with memory usage essentially being the space required by the compact graph data structure. As observed in Table 6.2, the time ratio for this implementation (fifth column, CT neighbour) is around an order of magnitude greater than the the time ratio for CT hash.

6.4.4 Memory Allocation Analysis

Our implementation of the dynamic k^2-tree is based on the technique presented in (Munro, Nekrich, and Vitter 2015), whose authors claim additional space is necessary to perform a union of two collections (which would be decompressed before the union operation taking place). The implementation we present is able to perform the union operation without decompressing the collections, effectively avoiding this pitfall. We show for dataset `uk-2007-05`, in Figure 6.11, a detailed analysis of heap memory usage. The analysis was performed using `valgrind`, with parameters `--tool=massif --max-snapshots=200 --detailed-freq=5`, and the visualizations using the `massif-visualizer`[8].

It can be observed that during execution where edges are continuously added, there are memory peaks associated with the union operation, temporarily increasing the heap usage by a factor of at most 2. This explains also the difference in maximum resident memory between `sdk2tree` and `dk2tree` observed before in Figures 6.8 and 6.9. The number of rebuilds/unions performed for dataset `uk-2007-05`, and for each static set in $\{E_1, \ldots, E_8\}$, is respectively 508, 127, 63, 32, 17, 9, 4 and 1.

6.5 Remarks

We presented the `sdk2tree` implementation for representing dynamic graphs, based on the k^2-tree graph representation and relying on a collection of static k^2-trees. It is a dynamic data structure that supports edge additions and removals with competitive performance, showing faster execution times than the `dk2tree` implementation, a dynamic version of k^2-trees based on dynamic bit vectors, and on par with k^2-tries with respect to additions and queries.

We also present a `C++` implementation `sdk2sdsl`, a modular version which makes

[8] https://github.com/KDE/massif-visualizer

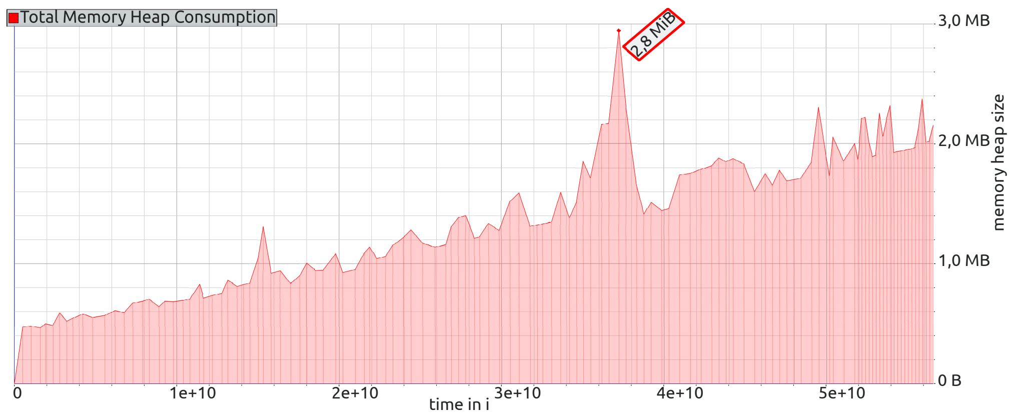


Figure 6.11: `valgrind` heap allocation profile for dataset `uk-2007-05`. The label `time in i` in the x axis denotes the number of instructions executed.

use of the succinct data structure library `sdsl-lite`. It achieves competitive performance compared to the other implementations analysed in this document. `sdk2sdsl` also provides efficient implementations of edge and neighbourhood iterators, and of elementary graph algorithms, with empirical time and space complexity in tune with theoretical bounds.

Implementations like those analysed in this paper, when implemented carefully, are of crucial importance for the efficient analysis and storage of evolving graphs, while drastically reducing the requirements of secondary storage compared to traditional dynamic graph representations. Hence, as future work, we envision further refinements to these data structures to achieve greater efficiency, namely in what concerns listing vertex neighbourhoods. `sdk2sdsl` is first step towards a reusable library for the analyses of large evolving graphs. We are also aiming to research how these representations may be used within distributed graph processing systems in order to reduce the memory pressure observed often in these systems.

While these have been studied extensively in the past (Boldi and Vigna 2004b), (Brisaboa, Ladra, and Navarro 2014), we did not find initiatives on their application to distributed graph processing. Most distributed systems (such as those based on dataflow programming) merely distribute the graph across the worker nodes in a cluster. For skewed graphs, this is not an optimal strategy, and in fact work has been done to address this (Chen, Shi, Chen, and Chen 2015). These compact representations are typically used in single-machine execution environments and haven't been applied in distributed systems. If their memory-efficiency is that good, perhaps each worker node in the cluster could have its own smart representation of the whole graph. In this sense, it becomes relevant to ask: how would global performance evolve when we reduce network communication (as every node can observe the whole graph) at the cost of the additional overhead of manipulating a smart graph representation? The synchronization of data between cluster nodes and the usage of statistics to keep track of error bounds will grow in importance in this scenario.

7 Conclusion

We present the conclusions of our research, encompassing our observations on experiments and on the field of graph processing as a whole. As it is an overreaching field, our research vectors were motivated by both improvements to existing graph processing solutions as well as their novel applications to existing problems. Focusing on these two points of innovation, it followed that our analysis of the state-of-the-art (Chapter 3) was not performed linearly. It was the result of a constant iterative analysis of the state-of-the-art, in function of new perspectives that emerged in the literature and those pertaining our own research development.

7.1 Graph Processing

7.1.1 VeilGraph Implementation, Flink Architecture

As we developed the ideas of VEILGRAPH, we stretched the architectural limitations of batch processing in `Apache Flink`, while also gaining awareness of them through direct communication with developers in the project's mailing lists. This posed a challenge for our research effort, as we only realized the limitations of the framework after discovering them as we invested more development effort on it.

A number of technical difficulties emerged when the concept of a dynamic graph was implemented over the batch API. To update a `DataSet` in `Flink` and use it in another batch job execution, it is recommended to spill it to disk and then read it back in. This becomes problematic for huge graphs due to the I/O latency. While it may appear acceptable for a one-time execution, some of the experiments comprise the execution of fifty (50) jobs, one job per chunk of updates received in the stream. As this cost adds up, we experimented reusing the `DataSet` instances between job executions. Unfortunately, this incurs the overhead of executing *everything* from start for every job. That is, when execution of job i is triggered (consisting of applying update i to the graph and executing the graph algorithm), the operators that were executed in job $i-1$ all the way from the initial job are executed again, resulting in exponential execution time, which is not a viable option.

When a Solution Becomes a Problem. Currently, as far as we know, there is no possible way by which to reuse results which are already in memory (and potentially distributed throughout a `Flink` cluster). Such a mechanism would be essential to efficiently implement VEILGRAPH over `Flink` using the appropriate stream processing semantics. A caching operator could allow for this behaviour. During development of VEILGRAPH, active contributors of Flink were contacted to ascertain the feasibility of this task. As programs are converted to dataflow operator graphs (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015), it would require an analysis over the execution plan optimizer and graph builder. An attempt was also made to evaluate a controlled experiment with a stream of updates represented as a simple Java collection. A set of ten-thousand (10, 000) edge additions was divided into buckets of a hundred (100) edges, each bucket representing an update. Between updates, the tested algorithm (PageRank) was executed. Although it is theoretically possible to define these in a single execution plan, in practice there was another prohibitive limitation. To build the execution dataflow graph, the plan optimizer has to consider a configuration space which can easily become exponential in size as more operators are added.

Thus, the experiment to execute in a single execution plan is also prohibitive as it would be stuck *forever* in the plan creation phase. The fact is that there is no concept of a dynamic graph over updates represented in Flink's streaming API. The closest to such a concept that exists over Flink is an experimental API (Kalavri, Carbone, Bali, and Abbas 2019) for single-pass graph streaming analytics also by the authors of Gelly.

A part of the detection of these problems resulted from an analysis of open-source project mailing lists. For example, we discovered that a caching operator feature would also be relevant for projects such as the Apache Mahout machine learning platform.[1] Reusing results while keeping them in memory is also relevant in a batch processing context as it would enable the visualization of intermediate results.

Surpassing these technical difficulties would target not only the execution of the described experiments. The caching mechanism would allow for an implementation operating more similarly to real streams, opening up the scope of research validation over real-world scenarios. Having said that, another possibility was to implement the dataflow-based components of VEILGRAPH in Apache Spark. Spark supports explicit RDD caching in memory, which as far as we know, is the equivalent to this missing functionality of result reuse in Flink. While caching may be employed with this framework, as far as we know it does not support incrementally adding or removing edges and vertices. In our experiments with GraphX, we found the same growing execution time problem that we encountered in Gelly. In Apache Flink the case of maintaining data for iterative computation is offered in the batch processing DataSet API, but manipulating data streams with rich logic with respect to the windowing and data aggregation behaviours are exclusive to the DataStream API.

Analysis of mailing list discussions and ongoing brainstorms[2] led us to the conclusion that Flink, while offering stream and batch processing, if it does so in a unified way at all, is only at a very low implementation level. The contract offered to programmers in the form of the aforementioned APIs establishes a clear separation between

[1] Apache Mahout use-case for the caching operator. Access date: 16 Dec '17. https://issues.apache.org/jira/browse/MAHOUT-1817

[2] Flink design proposal for unified stream and batch processing: https://docs.google.com/document/d/1G0NUIaaNJvT6CMrNCP6dRXGv88xNhDQqZFrQEuJ0rVU/edit#heading=h.ob9i0lcn7ulz

them. We see the lack of representativity of this use-case in the `Flink` API as another indicator that this vector of research in graph processing systems is still in early development. To stimulate the development of `Flink` in the direction of this use-case, we proposed a feature request using the project's open-source collaboration system, JIRA.[3] Some initial positive feedback was received.

7.1.2 Gelly-Scheduling - Use Case

Effectively, GELLY-SCHEDULING (Coimbra, Selimi, Francisco, Freitag, and Veiga 2018) emerged as an innovative application of graph processing for the challenge of leader placement in community network micro-clouds (CNMC). To pursue this possibility, a preliminary analysis of state-of-the-art graph processing systems was performed, leading to experimentation with solutions such as `Weaver` (Dubey, Hill, Escriva, and Sirer 2016), `Apache Flink` (Carbone, Katsifodimos, Ewen, Markl, Haridi, and Tzoumas 2015) and `Apache Spark` (Zaharia, Chowdhury, Franklin, Shenker, and Stoica 2010). `Weaver` in particular was challenging to analyse as it was a system which did not boast an active open-source community and required us to analyse its undocumented source code, only to discover that many of its features were not complete. We considered a multitude of available graph processing solutions, many published in high-quality venues. Considering the potential pitfalls (devaluing research time) of engaging the route of finely analysing their source code when confronted with apparent malfunctioning features, we took note of systems with active and large communities, of which `Apache Flink` and `Apache Spark` are prime examples.

As such, and with `Apache Flink` being originally designed with the stream processing scenario in mind, we performed our initial experiments with this framework. And it was through it that we explored the ideas validated in GELLY-SCHEDULING (Coimbra, Selimi, Francisco, Freitag, and Veiga 2018). `Apache Flink` is an open-source project, receiving contributions from distinct groups of people internationally. Many of its developments are implemented and validated in the realm of scientific publications, and it also has user-friendly documentation with examples to begin using it. Despite this, some aspects were only to be found in developers' posts in blogging platforms or software development online outlets. In original descriptions, the case of batch processing was described as a special case of stream processing.

7.2 Graph Storage

In light of the observations we made throughout our research, experiments and state-of-the-art analysis, the optimization of computational graph representations gained prominence. Tolerating different trade-offs between storage economy at the expense of processing speed became an interesting area to explore, with the potential of also applying its idea to the representations of graphs in existing tools. As part of an in-

[3]JIRA issue FLINK-10867: https://issues.apache.org/jira/browse/FLINK-10867?page=com.atlassian.jira.plugin.system.issuetabpanels%3Aall-tabpanel

ternship at the University of Chile, we researched different techniques for compressed and compact graphs. From this work, we implemented and compared a dynamic version of the k^2-tree data structure, achieving competitive results (Coimbra, Francisco, Russo, de Bernardo, Ladra, and Navarro 2020).

7.3 Future Work

This section identifies and proposes vectors of research activity. The first two consist in further exploring the ideas of current contributions while the last aims to synthesize assumptions and different domains spanned by graph processing, as well as the applicability of smart graph data structures to improve distributed system solutions.

Approximate graph processing. In Chapter 4, we presented VEILGRAPH, a proposed model with an API for approximate graph processing over a stream of graph updates. This work could benefit from certain technological developments which would act as enablers of more performance improvements. VEILGRAPH was developed over the `Flink` stream processing platform, and some of the platform's limitations were only understood during development. This project made use of the `Gelly` API, which is the platform's syntactic sugar for graph operations. However, it provides only abstractions in the context of batch processing. Enriching the model with error management findings from other contributions (e.g., (Mariappan and Vora 2019)) and implementing it on other platforms (or updated versions of the already considered ones, in the future) could pave the way for new validations and extensions of the model.

Scaling community network micro-clouds. We stated that our contribution described in Chapter 5, GELLY-SCHEDULING, could be further evaluated in a production network environment. Specifically, this evaluation as another use-case of our technique could be achieved by real-time monitoring in the *Quick Mesh Project* (QMP) by installing dedicated monitoring services in a set of nodes of the network. QMP is useful to perform additional model validations, as was done in (Selimi, Cerdà-Alabern, Artigas, Freitag, and Veiga 2017). Furthermore, another dimension to explore in this topic of community networks is to consider a need for increasing system size, higher frequency of sensor data point output and the speed of adaptation of the service placement. As these requirements become more complex, complete computation may be unable to satisfy the required optimizations. This problem could potentially benefit from incorporating the techniques of VEILGRAPH into GELLY-SCHEDULING.

Algebraic formulation. A higher level direction which is worth exploring is the algebraic definition of the operations performed over the graph, especially in the context of VEILGRAPH. The creation and deletion of vertices and edges are elementary operations over graphs. They are fine-grained building blocks of graph dynamism. However, there may be complex operations built over these which could also be relevant to various graph processing algorithms. Synthesizing a set of operations to use in graph processing systems could aid in establishing further gains. We base this claim on the inherent multidisciplinary nature of our work.

Exploring graph theory and the continuous study of graphs, we note that operations over them have been formalized (Bollobás 1979), defining concepts which describe and name their transformations. Bridging towards the execution of operations over graphs which are computationally represented, one will find in the literature the application of matrix algebra to perform them (e.g., (Buluç and Gilbert 2011b)). However, ensuring efficient execution is far more complex if we consider different contexts of execution: *a)* with parallelism and distributed systems (e.g., (Gregor and Lumsdaine 2005)); *b)* with different result accuracy criterion (e.g., (Mariappan and Vora 2019)); *c)* streams of incoming data (e.g., (Shi, Cui, Shao, and Tong 2016; Eksombatchai, Jindal, Liu, Liu, Sharma, Sugnet, Ulrich, and Leskovec 2018)). Throughout this study, we have come to believe there is a need for an algebraic formulation over graph processing in the context of stream processing and distributed systems in order to bring to the same level of importance these different dimensions.

Applications of smart graph data structures. In this work we analysed different computational graph representations, from the more primitive adjacency lists or matrices to represent the adjacencies of a graph, to sparse vectors and more refined structures such as the `WebGraph` as well as the k^2-tree data structure in Chapter 6, of which we compared several implementations. These efficient structures enable analysis of graphs on infrastructures which do not have unlimited storage. There is the potential to apply them to other types of graph processing architectures. For example, different distributed systems may represent graph data directly with graph-tailored structures, or they may implement graph constructs over underlying generic data representations. Topics such as redundancy, replication and distribution of graph data across workers nodes are important to achieve performance and scalability.

It would be interesting to integrate state-of-the-art smart graph representations into such systems and explore how the dynamics between these topics evolve. Consider that a large graph may have to be distributed across the cluster nodes to be processed. If it can be fully-represented in each worker, then all nodes have access to all of the graph. This has the potential of reducing certain types of communication between workers in order to improve performance, possibly raising the need to reinvent how conceptual operations (graph theory) over the graph will ultimately translate to the underlying (distributed) architecture.

Final remarks. As a field, graph theory has laid the foundations to structure the reasoning over interlinked information and inherent problems (Gross and Yellen 2005). However, processing large graphs using distributed systems adds another layer of conceptual challenges. What is the best approach to split graph elements across workers (or nodes) of a distributed computational infrastructure? How may we maximize performance and reduce I/O bottlenecks? Several works in the literature have emerged to address these questions, many of which we detailed in Chapter 3. Frameworks like `Apache Spark` (Zaharia, Chowdhury, Franklin, Shenker, and Stoica 2010) and `Flink` (Katsifodimos and Schelter 2016) were devised with general-purpose problem domains in mind, but soon led to libraries to process graphs over these infrastructures like `GraphX` (Xin, Gonzalez, Franklin, and Stoica 2013) and `Gelly`.

Other solutions were designed from the start to target graph-based data, enabling a direct association between graph-specific concerns and workload nature. For example, providing a solution for a scenario where new volumes of data are arriving (e.g. by having dedicated workers in clusters for data ingestion) and enabling their efficient integration due to awareness of the graph structure of data (e.g., `Tornado`, `Pixie`, `GraphBolt`). From these and other examples, we observe that elevating the awareness of the graph-structured nature of data, as a primary concern, enables the design of more efficient systems. This contrasts with the approach of building graph-tailored functionalities over general-purpose frameworks, which face more limitations and abstraction barriers to achieving the same levels of efficiency.

Our contributions herein presented have synergy between themselves. GELLY-SCHEDULING, upon deployment in more complex scenarios, could benefit from the concepts presented in VEILGRAPH in order to increase the speed with which communities are detected and leaders elected by tolerating a certain degree of approximation in result accuracy. Upon the arrival of network topology changes, for example the *big vertex* model could be constructed in order to update the desirability for impacted nodes to play the role of leader within their communities. It would also be interesting to explore how to create an adapted summary graph (different from the presented VEILGRAPH summary model) in order to apply approximate community detection algorithms as well.

The use of compact graph data structures such as the k^2-tree would be implemented in `Flink`'s internals in order to benefit VEILGRAPH. Such an effort would comprise a careful analysis of `Flink`'s code base in order to explore how to offer programmers an option (as a configuration file entry in `flink-conf.yaml` or as a function to set its value) to activate this internal graph representation. The research and implementation effort of integrating the k^2-tree into `Flink` to benefit VEILGRAPH would in our opinion be costly compared to the effort of applying VEILGRAPH techniques for the purpose of GELLY-SCHEDULING.

There are aspects to take in consideration when undertaking this to maximize the usability of the combined solution:

- `Gelly`'s powerful API should remain usable, thus ensuring all its graph algorithms and functionalities remain functional (while benefiting from the compact graph structure).

- It would be necessary to implement the logic in a way that intercepts the use of `Flink`'s `DataSet` API when its methods are invoked through the `Gelly` library to then use the k^2-tree as opposed to this `Flink` batch data representation. This would require appropriate changes to the code that distributes records across `TaskManager` instances. The conceptual change in data distribution would have the underlying assumption of the ability to fully represent the graph in each of the cluster's nodes, achieving redundancy through the compact representation with the aim of promoting distributed graph processing performance in `Flink`.

- To minimize the entropy in changing the code base of `Flink`, the offering of a k^2-tree implementation could perhaps be implemented using aspect-oriented programming to make a series of surgical changes to `Flink`'s flow of logic (data shuffling, scheduling, operator semantics) to use the added logic of the k^2-tree. This could be perhaps offered as a `Flink` library, much like `Gelly` currently is. To ensure this compact graph representation extension will remain in use, focus must be kept on implementing this in a way that minimizes conflicts from the advances in `Flink`'s own core code by the developer community.

- Our `C/C++` k^2-tree implementation uses low level bit operations. It would be very useful to explore the code base of `WebGraph`, also written in `Java` to reuse the low-level (bit) data manipulation functions to then produce a `Java`-based implementation of the k^2-tree to be incorporated with the previous points.

As the ventures of industry and academia began offering solutions for the reality of big data, the evolution of information also gained relevance. Techniques for processing incoming information have existed for over a decade now (Babcock, Datar, Motwani, et al. 2003; Tatbul, Çetintemel, and Zdonik 2007), but the analysis of dynamic graphs evolving through streams has been gaining traction as its own field (McGregor 2014; Ahmed, Duffield, Willke, and Rossi 2017; Besta, Fischer, Kalavri, Kapralov, and Hoefler 2019). Orthogonally, the tendency remains regarding the need to improve existing systems (Kalavri, Vlassov, and Haridi 2017; Heidari, Simmhan, Calheiros, and Buyya 2018; Coimbra, Francisco, and Veiga 2021) and develop new ones to process the big graphs of tomorrow (Sakr, Bonifati, Voigt, Iosup, Ammar, Angles, Aref, Arenas, Besta, Boncz, et al. 2021). An awareness of these different multidisciplinary perspectives coupled with a *unifying theory of graph processing* has the potential to enrich the current approaches to graph processing.

Bibliography

Agarwal, S., H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica (2014). Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, pp. 481–492. ACM.

Agarwal, S., B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica (2013). BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 29–42. ACM.

Aggarwal, G., M. Datar, S. Rajagopalan, and M. Ruhl (2004). On the streaming model augmented with a sorting primitive. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pp. 540–549. IEEE.

Ahmed, N. K., N. Duffield, T. L. Willke, and R. A. Rossi (2017, August). On Sampling from Massive Graph Streams. *Proc. VLDB Endow. 10*(11), 1430–1441.

Ahn, K. J., S. Guha, and A. McGregor (2012). Graph Sketches: Sparsification, Spanners, and Subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, New York, NY, USA, pp. 5–14. ACM.

Al-Molhem, N. R., Y. Rahal, and M. Dakkak (2019). Social network analysis in Telecom data. *Journal of Big Data 6*(1), 99.

Alexandrov, A., R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke (2014). The stratosphere platform for big data analytics. *The VLDB Journal 23*(6), 939–964.

Alverson, B., E. Froese, L. Kaplan, and D. Roweth (2012). Cray XC series network. *Cray Inc., White Paper WP-Aries01-1112*.

Amazon, I. (2020). Amazon Neptune Samples - Source Code. [Online, GitHub; accessed 24-April-2020].

Andreev, K. and H. Racke (2006). Balanced graph partitioning. *Theory of Computing Systems 39*(6), 929–939.

Angles, R. (2018). The Property Graph Database Model. [Online; accessed 24-April-2020].

Angles, R. and C. Gutierrez (2008). Survey of graph database models. *ACM Computing Surveys (CSUR) 40*(1), 1–39.

Apostolico, A. and G. Drovandi (2009). Graph compression by BFS. *Algorithms 2*(3), 1031–1044.

Armbrust, M., R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia (2015). Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, pp. 1383–1394. ACM.

Arroyuelo, D., G. de Bernardo, T. Gagie, and G. Navarro (2019). Faster dynamic compressed d-ary relations. In *String Processing and Information Retrieval (SPIRE)*, pp. 419–433.

Aurelius (2015). Titan: Distributed Graph Database. [Online; accessed 24-April-2020].

Authors, J. (2020). JanusGraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Babcock, B., M. Datar, and R. Motwani (2002). Sampling from a Moving Window over Streaming Data. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, Philadelphia, PA, USA, pp. 633–634. Society for Industrial and Applied Mathematics.

Babcock, B., M. Datar, R. Motwani, et al. (2003). Load Shedding Techniques for Data Stream Systems. In *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS*, pp. 1–3.

Bahrami, R. A., J. Gulati, and M. Abulaish (2017). Efficient processing of SPARQL queries over graphframes. In *Proceedings of the International Conference on Web Intelligence*, pp. 678–685.

Bajardi, P., C. Poletto, J. J. Ramasco, M. Tizzoni, V. Colizza, and A. Vespignani (2011). Human mobility networks, travel restrictions, and the global spread of 2009 H1N1 pandemic. *PloS one 6*(1).

Balaban, A. T. (1985). Applications of graph theory in chemistry. *Journal of chemical information and computer sciences 25*(3), 334–343.

Balasundaram, B. and S. Butenko (2006). Graph domination, coloring and cliques in telecommunications. In *Handbook of Optimization in Telecommunications*, pp. 865–890. Springer.

Bao, N. T. and T. Suzumura (2013). Towards highly scalable pregel-based graph processing platform with x10. In *Proceedings of the 22nd International Conference on World Wide Web*, pp. 501–508.

Baritchi, A., D. J. Cook, and L. B. Holder (2000). Discovering Structural Patterns in Telecommunications Data. In *FLAIRS Conference*, pp. 82–85.

Bearman, P. S., J. Moody, and K. Stovel (2004). Chains of affection: The structure of adolescent romantic and sexual networks. *American journal of sociology 110*(1), 44–91.

Bebee, B. R., D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, et al. (2018). Amazon Neptune: Graph Data Management in the Cloud. [Online; accessed 24-April-2020].

Benisis, A. (2010). *Business Process Management: A Data Cube to Analyze Business Process Simulation Data for Decision Making*. VDM Publishing.

Berkhin, P. (2005). A survey on PageRank computing. *Internet mathematics 2*(1), 73–120.

Besta, M., M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler (2019). Practice of Streaming and Dynamic Graphs: Concepts, Models, Systems, and Parallelism. *CoRR abs/1912.12740*.

Besta, M., D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler (2018). Log (graph) a near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pp. 1–13.

Beyene, Y., M. Faloutsos, D. H. Chau, and C. Faloutsos (2008). The eBay Graph: How do online auction users interact? In *IEEE INFOCOM Workshops 2008*, pp. 1–6. IEEE.

Bhan, A., D. J. Galas, and T. G. Dewey (2002). A duplication growth model of gene expression networks. *Bioinformatics 18*(11), 1486–1493.

Blondel, V. D., J. loup Guillaume, R. Lambiotte, and E. Lefebvre (2008). Fast unfolding of communities in large networks.

Blum, A., T. H. Chan, and M. R. Rwebangira (2006). A random-surfer web-graph model. In *2006 Proceedings of the Third Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pp. 238–246. SIAM.

Boldi, P., B. Codenotti, M. Santini, and S. Vigna (2004). Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience 34*(8), 711–726.

Boldi, P., A. Marino, M. Santini, and S. Vigna (2014). BUbiNG: Massive Crawling for the Masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, pp. 227–228. International World Wide Web Conferences Steering Committee.

Boldi, P., M. Rosa, M. Santini, and S. Vigna (2011). Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar (Eds.), *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, New York, NY, USA, pp. 587–596. ACM.

Boldi, P. and S. Vigna (2004a). The WebGraph Framework II: Codes For The World-Wide Web. In *2004 Data Compression Conference (DCC 2004), 23-25 March 2004, Snowbird, UT, USA*, pp. 528. IEEE Computer Society.

Boldi, P. and S. Vigna (2004b). The WebGraph framework I: Compression techniques. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills (Eds.), *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, New York, NY, USA, pp. 595–602. ACM.

Bollobás, B. (1979). Graph theory. graduate texts in mathematics.

BrightstarDB (2015). BrightstarDB - Source Code. [Online; accessed 24-April-2020].

Brin, S. and L. Page (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems 30*(1-7), 107–117.

Brisaboa, N. R., A. Cerdeira-Pena, G. de Bernardo, and G. Navarro (2017). Compressed representation of dynamic binary relations with applications. *Information Systems 69*, 106–123.

Brisaboa, N. R., G. de Bernardo, G. Gutiérrez, S. Ladra, M. R. Penabad, and B. A. Troncoso (2015). Efficient Set Operations over k2-Trees. In *Data Compression Conference (DCC)*, pp. 373–382.

Brisaboa, N. R., S. Ladra, and G. Navarro (2014). Compact representation of web graphs with extended functionality. *Information Systems 39*, 152–174.

Brockmann, D. and D. Helbing (2013). The hidden geometry of complex, network-driven contagion phenomena. *science 342*(6164), 1337–1342.

Buehrer, G. and K. Chellapilla (2008). A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pp. 95–106.

Buerli, M. and C. Obispo (2012). The current state of graph databases. *Department of Computer Science, Cal Poly San Luis Obispo, mbuerli@ calpoly. edu 32*(3), 67–83.

Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson (2009). Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pp. 233–244.

Buluç, A. and J. R. Gilbert (2011a). Combinatorial BLAS - Source Code. [Online, Berkeley; accessed 24-April-2020].

Buluç, A. and J. R. Gilbert (2011b). The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications 25*(4), 496–509.

Buluç, A., H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz (2016). Recent advances in graph partitioning. In *Algorithm Engineering*, pp. 117–158. Springer.

Busato, F., O. Green, N. Bombieri, and D. A. Bader (2018). Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7. IEEE.

Busato, F., O. Green, N. Bombieri, and D. A. Bader (2020). Hornet - Source Code. [Online, GitHub; accessed 24-April-2020].

Cailliau, P., T. Davis, V. Gadepally, J. Kepner, R. Lipman, J. Lovitz, and K. Ouaknine (2019). RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 285–286. IEEE.

Cambridge Semantics (2020). AnzoGraph® DB. [Online; accessed 24-April-2020].

Carbone, P., A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas (2015). Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36*(4), 28–38.

Cerdà-Alabern, L. (2012, Oct). On the topology characterization of Guifi.net. In *2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 389–396.

Chaudhry, H. N. (2019). FlowGraph: Distributed temporal pattern detection over dynamically evolving graphs. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, pp. 272–275.

Chen, C., X. Yan, F. Zhu, J. Han, and P. S. Yu (2008, Dec). Graph OLAP: Towards Online Analytical Processing on Graphs. In *2008 Eighth IEEE International Conference on Data Mining*, pp. 103–112.

Chen, R., J. Shi, Y. Chen, and H. Chen (2015). PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1. ACM.

Chen, R., J. Shi, Y. Chen, and H. Chen (2018). PowerLyra - Source Code. [Online, GitHub; accessed 24-April-2020].

Chen, X. (2019). GraphCage: Cache Aware Graph Processing on GPUs. *arXiv preprint arXiv:1904.02241*.

Cheng, R., J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen (2012). Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, New York, NY, USA, pp. 85–98. ACM.

Chien, S., C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar (2003). Link Evolutions: Analysis and Algorithms. *Internet Math. 1*(3), 277–304.

Chinazzi, M., J. T. Davis, M. Ajelli, C. Gioannini, M. Litvinova, S. Merler, A. P. y Piontti, K. Mu, L. Rossi, K. Sun, et al. (2020). The effect of travel restrictions on the spread of the 2019 novel coronavirus (COVID-19) outbreak. *Science*.

Ching, A. (2013). Scaling Apache Giraph to a trillion edges. *Facebook Engineering Blog*, 25.

Ching, A., S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan (2015, August). One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow. 8*(12), 1804–1815.

Chung, F. (2010). Graph theory in the information age. *Notices of the AMS 57*(6), 726–732.

Chung, F., L. Lu, T. G. Dewey, and D. J. Galas (2003). Duplication models for biological networks. *Journal of Computational Biology 10*(5), 677–687.

Chung, F. and O. Simpson (2015). Distributed algorithms for finding local clusters using heat kernel pagerank. In *International Workshop on Algorithms and Models for the Web-Graph*, Eindhoven, The Netherlands, pp. 177–189. Springer: Springer, Cham.

Cloud, A. (2020). Alibaba Graph Database. [Online; accessed 24-April-2020].

Coimbra, M. E., S. Esteves, A. P. Francisco, and L. Veiga (2021). VeilGraph: Streaming Graph Approximations. Submission: Journal of Big Data, Springer.

Coimbra, M. E., A. P. Francisco, L. M. S. Russo, G. de Bernardo, S. Ladra, and G. Navarro (2020, January). On dynamic succinct graph representations. In *Data Compression Conference (DCC)*, pp. 10. IEEE.

Coimbra, M. E., A. P. Francisco, and L. Veiga (2021). An analysis of the graph processing landscape. *Journal of Big Data 8*(1), 1–41.

Coimbra, M. E., J. Hrotkó, A. P. Francisco, L. M. S. Russo, G. de Bernardo, S. Ladra, and G. Navarro (2021). A practical succinct dynamic graph representation. Submission: Information and Computation, Elsevier.

Coimbra, M. E., M. Selimi, A. P. Francisco, F. Freitag, and L. Veiga (2018, April). Gelly-Scheduling: Distributed Graph Processing for Service Placement in Community Networks. pp. 151–160.

Coles, S., J. Bawa, L. Trenner, and P. Dorazio (2001). *An introduction to statistical modeling of extreme values*, Volume 208. Springer.

Colizza, V., A. Barrat, M. Barthélemy, and A. Vespignani (2007). Predictability and epidemic pathways in global outbreaks of infectious diseases: the SARS case study. *BMC medicine 5*(1), 34.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Corporation, M. (2019). Hub for Azure Cosmos DB - Source Code. [Online, GitHub; accessed 24-April-2020].

Corporation, O. (1999, Dec). Application and System Performance Characteristics. [Online, GitHub; accessed 24-April-2020].

Csardi, G., T. Nepusz, et al. (2006). The igraph software package for complex network research. *InterJournal, complex systems 1695*(5), 1–9.

DataStax, Inc. (2016). DataStax Enterprise Graph. [Online; accessed 24-April-2020].

Dathathri, R., G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali (2018). Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 752–768.

Dathathri, R., G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali (2020a). faimGraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Dathathri, R., G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali (2020b). Gluon - Source Code. [Online, GitHub; accessed 24-April-2020].

Dave, A., A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia (2016). GraphFrames: an integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pp. 1–8.

David, E. and K. Jon (2010). *Networks, Crowds, and Markets: Reasoning About a Highly Connected World.* New York, NY, USA: Cambridge University Press.

de Bruijn, N. G. (1946, June). A Combinatorial Problem. *Koninklijke Nederlandsche Akademie Van Wetenschappen 49*(6), 758–764.

De Virgilio, R., A. Maccioni, and R. Torlone (2014). Model-driven design of graph databases. In *International Conference on Conceptual Modeling*, pp. 172–185. Springer.

Demetrescu, C., I. Finocchi, and A. Ribichini (2009). Trading off space for passes in graph streaming problems. *ACM Transactions on Algorithms (TALG) 6*(1), 1–17.

Deutsch, A., Y. Xu, M. Wu, and V. Lee (2019). TigerGraph: A Native MPP Graph Database. *arXiv preprint arXiv:1901.08248*.

Dgraph Labs, Inc. (2020a, April). A Tour of Dgraph. [Online; accessed 24-April-2020].

Dgraph Labs, Inc. (2020b). Dgraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Dhulipala, L., G. Blelloch, and J. Shun (2017). Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 293–304.

Dhulipala, L., G. E. Blelloch, and J. Shun (2019). Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 918–934.

Dhulipala, L., G. E. Blelloch, and J. Shun (2020). Aspen - Source Code. [Online, GitHub; accessed 24-April-2020].

Dubey, A. (2016). Weaver - Source Code. [Online, GitHub; accessed 24-April-2020].

Dubey, A., G. D. Hill, R. Escriva, and E. G. Sirer (2016). Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *PVLDB 9*(11), 852–863.

Ediger, D., R. McColl, J. Riedy, and D. A. Bader (2012). Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pp. 1–5. IEEE.

Eksombatchai, C., P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec (2018). Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, Republic and Canton of Geneva, Switzerland, pp. 1775–1784. International World Wide Web Conferences Steering Committee.

Erb, B., D. Meißner, F. Kargl, B. A. Steer, F. Cuadrado, D. Margan, and P. Pietzuch (2018). GraphTides: a framework for evaluating stream-based graph processing platforms. In *Proceedings of the 1st ACM SIGMOD joint international workshop on graph data management experiences & systems (GRADES) and network data analytics (NDA)*, pp. 1–10.

Erling, O. (2012). Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull. 35*(1), 3–8.

Euler, L. (1956). *The seven bridges of Konigsberg*. Wm. Benton.

Facebook (2020). Newsrooms. [Online; accessed 05-May-2020].

Facebook Database Engineering Team (2012). RocksDB - Source Code. [Online, GitHub; accessed 24-April-2020].

Facebook, Inc. (2016, October). GraphQL. [Online, GitHub; accessed 24-April-2020].

Fan, W., J. Li, X. Wang, and Y. Wu (2012). Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 157–168.

Fauna, I. (2020). FaunaDB — The database built for serverless, featuring native GraphQL. [Online; accessed 24-April-2020].

Feder, T. and R. Motwani (1995). Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences 51*(2), 261–272.

Feigenbaum, J., S. Kannan, A. McGregor, S. Suri, and J. Zhang (2005). On graph problems in a semi-streaming model. *Departmental Papers (CIS)*, 236.

Fortunato, S. and M. Barthélemy (2007). Resolution limit in community detection. *Proceedings of the National Academy of Sciences 104*(1), 36–41.

Fortunato, S., A. Flammini, and F. Menczer (2006). Scale-free network growth by ranking. *Physical review letters 96*(21), 218701.

Fortunato, S. and D. Hric (2016). Community detection in networks: A user guide. *CoRR abs/1608.00163*, 1–44.

Foundation, T. A. S. (2019a). Apache Giraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Foundation, T. A. S. (2019b). Apache TinkerPop. [Online; accessed 24-April-2020].

Foundation, T. A. S. (2020a). Apache Flink - Source Code. [Online, GitHub; accessed 24-April-2020].

Foundation, T. A. S. (2020b). Apache Spark - Source Code. [Online, GitHub; accessed 24-April-2020].

Francis, N., A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor (2018). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1433–1445.

Freeman, L. C. (1977). A Set of Measures of Centrality Based on Betweenness. *Sociometry 40*(1), 35–41.

Fu, Z., M. Personick, and B. Thompson (2014). MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pp. 1–6.

Fu, Z., M. Personick, and B. Thompson (2016). MapGraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Gagie, T., J. I. González-Nova, S. Ladra, G. Navarro, and D. Seco (2015). Faster compressed quadtrees. In *Data Compression Conference (DCC), 2015*, pp. 93–102. IEEE.

George, G. and S. M. Thampi (2018). A graph-based security framework for securing industrial IoT networks from vulnerability exploitations. *IEEE Access 6*, 43586–43601.

George, L. (2011). *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly.

Goiri, I., R. Bianchini, S. Nagarakatte, and T. D. Nguyen (2015, March). ApproxHadoop: Bringing Approximations to MapReduce Frameworks. *SIGARCH Comput. Archit. News 43*(1), 383–397.

Gonzalez, J. E., Y. Low, H. Gu, D. Bickson, and C. Guestrin (2012). PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, Berkeley, CA, USA, pp. 17–30. USENIX Association.

Gonzalez, J. E., R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica (2014). GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Berkeley, CA, USA, pp. 599–613. USENIX Association.

Google (2017). Google Cayley - Source Code. [Online, GitHub; accessed 24-April-2020].

Gordon Donnelly (2020, March). 75 Super-Useful Facebook Statistics for 2018. [Online; accessed 05-May-2020].

Grando, F., D. Noble, and L. C. Lamb (2016). An analysis of centrality measures for complex and social networks. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6. IEEE.

Green, A., M. Junghanns, M. Kießling, T. Lindaaker, S. Plantikow, and P. Selmer (2018). openCypher: New Directions in Property Graph Querying. In *EDBT*, pp. 520–523.

Gregor, D. and A. Lumsdaine (2005). The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC) 2*, 1–18.

Gregor, D. and A. Lumsdaine (2018). Parallel Boost Graph Library - Source Code. [Online, GitHub; accessed 24-April-2020].

Gregory, S. (2010). Finding overlapping communities in networks by label propagation. *New journal of Physics 12*(10), 103018.

Gross, J. L. and J. Yellen (2005). *Graph theory and its applications*. CRC press.

Grujić, J. (2008). Movies recommendation networks as bipartite graphs. In *International Conference on Computational Science*, pp. 576–583. Springer.

Gu, Q., J. Zhou, and C. Ding (2010). Collaborative filtering: Weighted nonnegative matrix factorization incorporating user and item graphs. In *Proceedings of the 2010 SIAM international conference on data mining*, pp. 199–210. SIAM.

Gubichev, A. and M. Then (2014). Graph Pattern Matching: Do We Have to Reinvent the Wheel? In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pp. 1–7.

Guidotti, R. and M. Coscia (2017). On the equivalence between community discovery and clustering. In *International Conference on Smart Objects and Technologies for Social Good*, pp. 342–352. Springer.

Guo, Y., Z. Pan, and J. Heflin (2005). LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics 3*(2-3), 158–182.

Gustafson, J. L. (2011). *Gustafson's Law*, pp. 819–825. Boston, MA: Springer US.

Haeusler, M., T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu (2017). ChronoGraph: A Versioned TinkerPop Graph Database. In *International Conference on Data Management Technologies and Applications*, pp. 237–260. Springer.

Haeusler, M., T. Trojer, J. Kessler, M. Farwick, E. Nowakowski, and R. Breu (2020). ChronoDB - Source Code. [Online, GitHub; accessed 24-April-2020].

Han, M., K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin (2014, August). An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. VLDB Endow. 7*(12), 1047–1058.

Hartig, O. (2014). Reconciliation of RDF* and property graphs. *arXiv preprint arXiv:1409.3288*.

Hartig, O. and J. Pérez (2017). An initial analysis of Facebook's GraphQL language. [Online; accessed 24-April-2020].

Hartig, O. and J. Pérez (2018). Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference*, pp. 1155–1164.

He, H. and A. K. Singh (2008). Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 405–418.

Heidari, S., Y. Simmhan, R. N. Calheiros, and R. Buyya (2018). Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Computing Surveys (CSUR) 51*(3), 60.

Henderson, C. E. (2014). System and method for creating, deploying, integrating, and distributing nodes in a grid of distributed graph databases. US Patent 8,775,476.

Hernández, C. and G. Navarro (2014). Compressed representations for web and social graphs. *Knowl. Inf. Syst. 40*(2), 279–313.

Holzschuher, F. and R. Peinl (2013). Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4J. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, New York, NY, USA, pp. 195–204. ACM.

Horn, P. and M. Rydberg (2020). openCypher - Source Code. [Online, GitHub; accessed 24-April-2020].

Hosking, J. (2000). FORTRAN routines for use with the method of L-moments, Version 3.04. *IBM Research*.

Hosking, J. R. (1994). The four-parameter kappa distribution. *IBM Journal of Research and Development 38*(3), 251–258.

Hosking, J. R. M. (1990). L-moments: Analysis and estimation of distributions using linear combinations of order statistics. *Journal of the Royal Statistical Society. Series B (Methodological) 52*(1), 105–124.

Hosking, J. R. M. and J. R. Wallis (2005). *Regional frequency analysis: an approach based on L-moments*. Cambridge University Press.

Hu, P. and W. C. Lau (2013, aug). A Survey and Taxonomy of Graph Sampling.

Huang, J., D. J. Abadi, and K. Ren (2011). Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment 4*(11), 1123–1134.

Hwang, M. (2018). Graph Processing Using SAP HANA: A Teaching Case. *e-Journal of Business Education and Scholarship of Teaching 12*(2), 155–165.

IBM System G Team (2015, Oct). The World is Big and Linked: Whole Spectrum Industry Solutions towards Big Graphs - Graph Computing and Tutorial of IBM System G. [Online; accessed 24-April-2020].

Inc., F. (1984). AllegroGraph. [Online; accessed 24-April-2020].

InternetLiveStats.com (2020). Total number of Websites. [Online; accessed 24-April-2020].

Iordanov, B. (2010). HyperGraphDB: a generalized graph database. In *International Conference on Web-age Information Management*, pp. 25–36. Springer.

Iordanov, B. (2020). Distributed HypergraphDB: partial replication. [Online; accessed 24-April-2020].

Ivanciuc, O. (2013). Chemical Graphs, Molecular Matrices and Topological Indices in Chemoinformatics and Quantitative Structure-Activity Relationships §. *Current computer-aided drug design 9*(2), 153–163.

Iyer, A. P., L. E. Li, T. Das, and I. Stoica (2016). Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pp. 1–6.

Jain, A. K., G. Arora, and R. Agrawal (2020). Graph Regularization for Multi-lingual Topic Models. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1741–1744.

Jain, N., G. Liao, and T. L. Willke (2013). Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, pp. 1–6.

JanusGraph Authors (2017). JanusGraph: Distributed Graph Database. [Online; accessed 24-April-2020].

JCC Consulting, I. (2020). GQL Standard. [Online; accessed 24-April-2020].

Jia, Z., Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken (2017). A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment 11*(3), 297–310.

Jia, Z., Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken (2018). Lux - Source Code. [Online, GitHub; accessed 24-April-2020].

Jin, C., S. S. Bhowmick, X. Xiao, J. Cheng, and B. Choi (2010). GBLENDER: towards blending visual query formulation and query processing in graph databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 111–122.

Jones, N. and P. Pevzner (2004). *An Introduction to Bioinformatics Algorithms*. A Bradford book. London.

Junghanns, M., M. Kießling, A. Averbuch, A. Petermann, and E. Rahm (2017). Cypher-based Graph Pattern Matching in Gradoop. In P. A. Boncz and J. Larriba-Pey (Eds.), *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pp. 3:1–3:8. ACM.

Junghanns, M., M. Kießling, N. Teichmann, K. Gómez, A. Petermann, and E. Rahm (2018). Declarative and distributed graph analytics with GRADOOP. *PVLDB 11*(12), 2006–2009.

Kakwani, D. and Y. Simmhan (2019). Distributed Algorithms for Subgraph-Centric Graph Platforms. *arXiv preprint arXiv:1905.08051*.

Kalavri, V., P. Carbone, D. Bali, and Z. Abbas (2019). Gelly Streaming - Source Code. [Online, GitHub; accessed 24-April-2020].

Kalavri, V., S. Ewen, K. Tzoumas, V. Vlassov, V. Markl, and S. Haridi (2014). Asymmetry in Large-Scale Graph Analysis, Explained. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, New York, NY, USA, pp. 4:1–4:7. ACM.

Kalavri, V., T. Simas, and D. Logothetis (2016). The shortest path is not always a straight line: leveraging semi-metricity in graph analysis. *Proceedings of the VLDB Endowment 9*(9), 672–683.

Kalavri, V., V. Vlassov, and S. Haridi (2017). High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering 30*(2), 305–324.

Kang, U. and C. Faloutsos (2011). Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *2011 IEEE 11th International Conference on Data Mining*, pp. 300–309. IEEE.

Kankanamge, C., S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu (2017). Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1695–1698.

Kankanamge, Chathura and Sahu, Siddhartha and Mhedbhi, Amine and Chen, Jeremy and Salihoglu, Semih (2017). Graphflow - Source Code. [Online, GitHub; accessed 24-April-2020].

Katsifodimos, A. and S. Schelter (2016). Apache Flink: Stream Analytics at Scale. In *2016 IEEE International Conference on Cloud Engineering Workshop, IC2E Workshops, Berlin, Germany, April 4-8, 2016*, pp. 193. IEEE Computer Society.

Katz, L. (1953). A new status index derived from sociometric analysis. *Psychometrika 18*(1), 39–43.

Kevin Gómez, Christopher Rost, M. J. (2020). GRADOOP - Source Code. [Online, GitHub; accessed 24-April-2020].

Khorasani, F., K. Vora, R. Gupta, and L. N. Bhuyan (2014). CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pp. 239–252.

Khorasani, F., K. Vora, R. Gupta, and L. N. Bhuyan (2015). CuSha - Source Code. [Online, GitHub; accessed 24-April-2020].

Ko, S. and W.-S. Han (2018). TurboGraph++ A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 395–410.

Kolomičenko, V., M. Svoboda, and I. H. Mlỳnková (2013). Experimental comparison of graph databases. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, pp. 115–124.

Koschützki, D., K. A. Lehmann, L. Peeters, S. Richter, D. Tenfelde-Podehl, and O. Zlotowski (2005). Centrality indices. In *Network analysis*, pp. 16–61. Springer.

Kostakos, V. (2009). Temporal graphs. *Physica A: Statistical Mechanics and its Applications 388*(6), 1007–1023.

Krawczyk, B., L. L. Minku, J. Gama, J. Stefanowski, and M. Woźniak (2017). Ensemble learning for data stream analysis: A survey. *Information Fusion 37*, 132–156.

Kruskal, J. B. (1956). On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society 7*(1), 48–50.

Laboratory for Web Algorithmics (2020). Datasets. [Online; accessed 05-May-2020].

Lakshman, A. and P. Malik (2010, April). Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*(2), 35–40.

Langville, A. N. and C. D. Meyer (2004). Updating Pagerank with Iterative Aggregation. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, New York, NY, USA, pp. 392–393. ACM.

Langville, A. N. and C. D. Meyer (2011). *Google's PageRank and beyond: The science of search engine rankings*. 41 William Street, Princeton, New Jersey: Princeton University Press.

Leskovec, J., A. Rajaraman, and J. Ullman (2014). *Mining of Massive Datasets*. Cambridge University Press.

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

Li, R., H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, S. Li, H. Yang, J. Wang, and J. Wang (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research 20*(2), 265–272.

Lightenberg, W., Y. Pei, G. Fletcher, and M. Pechenizkiy (2018). Tink: A temporal graph analytics library for Apache Flink. In *Companion Proceedings of the The Web Conference 2018*, pp. 71–72.

Lightenberg, W., Y. Pei, G. Fletcher, and M. Pechenizkiy (2019). Tink - Source Code. [Online, GitHub; accessed 24-April-2020].

Liljeros, F., C. R. Edling, and L. A. N. Amaral (2003). Sexual networks: implications for the transmission of sexually transmitted infections. *Microbes and infection 5*(2), 189–196.

Lim, J., S. Ryu, K. Park, Y. J. Choe, J. Ham, and W. Y. Kim (2019). Predicting drug–target interaction using a novel graph neural network with 3D structure-embedded graph representation. *Journal of chemical information and modeling 59*(9), 3981–3988.

Lim, Y., U. Kang, and C. Faloutsos (2014). Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering 26*(12), 3077–3089.

LinkedIn Corporation (2020). Quarterly results. [Online; accessed 05-May-2020].

Liu, X. and T. Murata (2010). Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Statistical Mechanics and its Applications 389*(7), 1493–1500.

Liu, Y., C. Zhou, J. Gao, and Z. Fan (2016). GiraphAsync: Supporting Online and Offline Graph Processing via Adaptive Asynchronous Message Processing. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, New York, NY, USA, pp. 479–488. ACM.

Lofgren, E. T. and N. H. Fefferman (2007). The untapped potential of virtual game worlds to shed light on real world epidemics. *The Lancet infectious diseases 7*(9), 625–629.

Low, Y., J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein (2010). GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, Arlington, Virginia, USA, pp. 340–349. AUAI Press.

Low, Y., J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein (2014a). GraphLab - Source Code. [Online, GitHub; accessed 24-April-2020].

Low, Y., J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein (2014b). GraphLab: A new framework for parallel machine learning. *CoRR abs/1408.2041*.

Ltd., R. L. (2020). RedisGraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Lyon, W. (2020). GRANDstack. [Online; accessed 24-April-2020].

Maass, S., C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim (2017a). Mosaic - Source Code. [Online, GitHub; accessed 24-April-2020].

Maass, S., C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim (2017b). Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, New York, NY, USA, pp. 527–543. ACM.

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski (2010). Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, pp. 135–146. ACM.

Malicevic, J., A. Roy, and W. Zwaenepoel (2014). Scale-up Graph Processing in the Cloud: Challenges and Solutions. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*, CloudDP '14, New York, NY, USA, pp. 5:1–5:6. ACM.

MariaDB (2016). Open Query GRAPH computation engine. [Online; accessed 24-April-2020].

Mariappan, M. and K. Vora (2019). GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, pp. 25:1–25:16. ACM.

Mariappan, M. and K. Vora (2020). GraphBolt - Source Code. [Online, GitHub; accessed 24-April-2020].

Martella, C., D. Logothetis, A. Loukas, and G. Siganos (2017). Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pp. 1083–1094. Ieee.

Martinez-Bazan, N., S. Gomez-Villamor, and F. Escale-Claveras (2011). DEX: A high-performance graph database management system. In *2011 IEEE 27th International Conference on Data Engineering Workshops*, pp. 124–127. IEEE.

Matam, K. K., G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram (2019). GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 116–128.

Mayer, C., M. A. Tariq, R. Mayer, and K. Rothermel (2016). GraphH - Source Code. [Online, GitHub; accessed 24-April-2020].

Mayer, C., M. A. Tariq, R. Mayer, and K. Rothermel (2018). GrapH: Traffic-aware graph processing. *IEEE Transactions on Parallel and Distributed Systems 29*(6), 1289–1302.

McGregor, A. (2014, May). Graph Stream Algorithms: A Survey. *SIGMOD Rec. 43*(1), 9–20.

Meusel, R., S. Vigna, O. Lehmberg, and C. Bizer (2015). The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science 1*.

Meyer, S. M., J. Degener, J. Giannandrea, and B. Michener (2010). Optimizing Schema-last Tuple-store Queries in Graphd. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, pp. 1047–1056. ACM.

Mhedhbi, A., P. Gupta, S. Khaliq, and S. Salihoglu (2020). A+ Indexes: Lightweight and Highly Flexible Adjacency Lists for Graph Database Management Systems. arXiv preprint arXiv:2004.00130.

Mhedhbi, A. and S. Salihoglu (2019, July). Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow. 12*(11), 1692–1704.

Miao, Y., W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen (2015). Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS) 11*(3), 1–34.

Michail, O. (2016). An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics 12*(4), 239–280.

Microsoft (2017). Graph Engine (GE): Serving Big Graphs in Real-time. [Online; accessed 24-April-2020].

Miller, J. J. (2013). Graph database applications and concepts with neo4j. *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA 2324*(36).

Mishra, R. K. and S. R. Raman (2019). GraphFrames. In *PySpark SQL Recipes*, pp. 297–315. Springer.

Moffat, A. (2018). Computing Maximized Effectiveness Distance for Recall-Based Metrics. *IEEE Transactions on Knowledge and Data Engineering 30*(1), 198–203.

Mofrad, M. H., R. Melhem, and M. Hammoud (2018). Revolver: vertex-centric graph partitioning using reinforcement learning. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 818–821. IEEE.

Muggli, M. D., A. Bowe, N. R. Noyes, P. S. Morley, K. E. Belk, R. Raymond, T. Gagie, S. J. Puglisi, and C. Boucher (2017). Succinct colored de Bruijn graphs. *Bioinformatics 33*(20), 3181–3187.

Munro, J. I., Y. Nekrich, and J. S. Vitter (2015). Dynamic Data Structures for Document Collections and Graphs. In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 277–289.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Murray, D. G., F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi (2013). Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, pp. 439–455. ACM.

Nathan, A. and R. K. Even (1967). The inversion of sparse matrices by a strategy derived from their graphs. *The Computer Journal 10*(2), 190–194.

Navarro, G. (2016). *Compact data structures: A practical approach*. Cambridge University Press.

Neo4j, I. (2018). What is openCypher? [Online; accessed 24-April-2020].

Neo4j, I. (2019). Cypher for Gremlin - Source Code. [Online, GitHub; accessed 24-April-2020].

Neo4j, I. (2020a). Neo4j and GraphQL. [Online; accessed 24-April-2020].

Neo4j, I. (2020b). What is a Graph Database? [Online; accessed 24-April-2020].

Neo4j Inc. (2020). Neo4j - Source Code. [Online; accessed 24-April-2020].

Newman, M. (2010). *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc.

Novotny, P., R. Urgaonkar, A. L. Wolf, and B. Ko (2015, October). Dynamic placement of composite software services in hybrid wireless networks. In *Military Communications Conference, MILCOM 2015-2015 IEEE*, pp. 1052–1057. IEEE.

Objectivity (2016). Objectivity/DB. [Online; accessed 24-April-2020].

Ontotext (2020). GraphDB — The Best RDF Database for Knowledge Graphs. [Online; accessed 24-April-2020].

OpenLink Software (2020). Virtuoso Open-Source Edition - Source Code. [Online, GitHub; accessed 24-April-2020].

Optiver (2016). Ruruki - In-Memory Directed Property Graph. [Online; accessed 24-April-2020].

Oracle (2020). Spatial and Graph features in Oracle Database. [Online; accessed 24-April-2020].

OrientDB LTD (2020). OrientDB - Source Code. [Online; accessed 24-April-2020].

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

Panda, B., J. S. Herbach, S. Basu, and R. J. Bayardo (2009). Planet: massively parallel learning of tree ensembles with MapReduce. *Proceedings of the VLDB Endowment 2*(2), 1426–1437.

Paredaens, J., P. Peelman, and L. Tanca (1995). G-Log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering 7*(3), 436–453.

Paz, J. R. G. (2018). Introduction to Azure Cosmos DB. In *Microsoft Azure Cosmos DB Revealed*, pp. 1–23. Springer.

Pearce, R., M. Gokhale, and N. M. Amato (2013). Scaling techniques for massive scale-free graphs in distributed (external) memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 825–836. IEEE.

Pearce, R., M. Gokhale, and N. M. Amato (2019). HavocGT - Source Code. [Online, GitHub; accessed 24-April-2020].

Pell, J., A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown (2012). Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences 109*(33), 13272–13277.

Pérez, J., M. Arenas, and C. Gutierrez (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS) 34*(3), 1–45.

Perez, Y., R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec (2015). Ringo: Interactive Graph Analytics on Big-Memory Machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, pp. 1105–1110. ACM.

Perez, Y., R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec (2016). Ringo - Source Code. [Online, GitHub; accessed 24-April-2020].

Petroni, F., L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni (2015). Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 243–252.

Pfluke, C. (2019). A history of the Five Eyes Alliance: Possibility for reform and additions: A history of the Five Eyes Alliance: Possibility for reform and additions. *Comparative Strategy 38*(4), 302–315.

Plantikow, S. (2019, March). Towards an International Standard for the GQL Graph Query Language. [Online; accessed 24-April-2020].

Pokornỳ, J. (2015). Graph databases: their power and limitations. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pp. 58–69. Springer.

Prim, R. C. (1957). Shortest Connection Networks And Some Generalizations. *Bell System Technical Journal 36*(6), 1389–1401.

Qi, X., E. Fuller, Q. Wu, Y. Wu, and C.-Q. Zhang (2012). Laplacian centrality: A new centrality measure for weighted networks. *Information Sciences 194*, 240–253.

Quijada-Fuentes, C., M. R. Penabad, S. Ladra, and G. Gutiérrez (2019). Set operations over compressed binary relations. *Information Systems 80*, 76–90.

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

Rathore, M. M., A. Ahmad, A. Paul, and U. K. Thikshaja (2016). Exploiting real-time big data to empower smart transportation using big graphs. In *2016 IEEE Region 10 Symposium (TENSYMP)*, pp. 135–139. IEEE.

Research, M. (2018). Naiad - Source Code. [Online, GitHub; accessed 24-April-2020].

Rickett, C. D., U.-U. Haus, J. Maltby, and K. J. Maschhoff (2018). Loading and querying a trillion rdf triples with cray graph engine on the cray xc. *Cray User Group*.

Robinson, I., J. Webber, and E. Eifrem (2013). *Graph Databases*. O'Reilly Media, Inc.

Robinson, I., J. Webber, and E. Eifrem (2015). *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.".

Rodriguez, M. A. (2015a). The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pp. 1–10.

Rodriguez, M. A. (2015b). The Gremlin Graph Traversal Machine and Language. *CoRR abs/1508.03843*.

Ronhovde, P. and Z. Nussinov (2010, Apr). Local resolution-limit-free potts model for community detection. *Phys. Rev. E 81*, 046114.

Roy, A., L. Bindschaedler, J. Malicevic, and W. Zwaenepoel (2015). Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, pp. 410–424. ACM.

Roy, A., L. Bindschaedler, J. Malicevic, and W. Zwaenepoel (2016). Chaos - Source Code. [Online, GitHub; accessed 24-April-2020].

Roy, A., I. Mihailovic, and W. Zwaenepoel (2013). X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, pp. 472–488. ACM.

Roy, A., I. Mihailovic, and W. Zwaenepoel (2015). X-Stream - Source Code. [Online, GitHub; accessed 24-April-2020].

Rudolf, M., M. Paradies, C. Bornhövd, and W. Lehner (2013). The graph story of the SAP HANA database. *Datenbanksysteme für Business, Technologie und Web (BTW) 2037*.

Sahu, S., A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu (2017, December). The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow. 11*(4), 420–431.

Sakr, S., A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al. (2021). The future is big graphs: a community view on graph processing systems. *Communications of the ACM 64*(9), 62–71.

Sakr, S., F. M. Orakzai, I. Abdelaziz, and Z. Khayyat (2016). *Large-Scale Graph Processing Using Apache Giraph*. Springer.

Salihoglu, S. and J. Widom (2013a). GPS - Source Code. [Online, Stanford; accessed 24-April-2020].

Salihoglu, S. and J. Widom (2013b). GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pp. 1–12.

Samet, H. (2006). *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

SAP SE (2018). SAP HANA Graph Academy - Source Code. [Online, GitHub; accessed 24-April-2020].

Schätzle, A., M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen (2015). S2X: graph-parallel querying of RDF with GraphX. In *Biomedical Data Management and Graph Online Querying*, pp. 155–168. Springer.

Schelter, S., A. Palumbo, S. Quinn, S. Marthi, and A. Musselman (2016). Samsara: Declarative machine learning on distributed dataflow systems. In *NIPS Workshop MLSystems*.

Seabold, S. and J. Perktold (2010). Statsmodels: Econometric and Statistical modeling with Python. In *9th Python in Science Conference*.

Securities and Exchange Commission (2019). One billion. [Online; accessed 05-May-2020].

Sedgewick, R. and K. Wayne (2011). *Algorithms*. Addison-wesley professional.

Selimi, M., L. Cerdà-Alabern, M. S. Artigas, F. Freitag, and L. Veiga (2017, April). Practical Service Placement Approach for Microservices Architecture. In *IEEE/ACM 17th International Symposium On Cluster, Cloud And Grid (CCGRID 2017)*. ACM/IEEE.

Selimi, M., F. Freitag, L. Cerdà-Alabern, and L. Veiga (2016, August). Performance Evaluation of a Distributed Storage Service in Community Network Clouds. *Concurrency and Computation: Practice and Experience 28*(11), 3131–3148.

Selimi, M., A. M. Khan, E. Dimogerontakis, F. Freitag, and R. P. Centelles (2015). Cloud services in the Guifi. net community network. *Computer Networks 93*, 373–388.

Selimi, M., D. Vega, F. Freitag, and L. Veiga (2016, August). Towards Network-Aware Service Placement in Community Network Micro-Clouds. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing*. Springer.

Sengupta, D., N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan (2016). GraphIn: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pp. 319–333. Springer.

Sheng, L., Z. M. Ozsoyoglu, and G. Ozsoyoglu (1999). A graph query language and its query processing. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*, pp. 572–581. IEEE.

Shi, X., B. Cui, Y. Shao, and Y. Tong (2016). Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, pp. 417–430. ACM.

Shi, X., X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin (2017). Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Transactions on Knowledge and Data Engineering 30*(1), 29–42.

Shi, X., X. Luo, J. Liang, P. Zhao, S. Di, B. He, and H. Jin (2018). Frog - Source Code. [Online, GitHub; accessed 24-April-2020].

Shi, X., Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua (2018). Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR) 50*(6), 1–35.

Shimpi, D. and S. Chaudhari (2012). An overview of graph databases. In *Int. Conf. on Recent Trends in Information Technology and Computer Science*, pp. 16–22.

Shun, J. and G. E. Blelloch (2013). Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 135–146.

Shun, J. and G. E. Blelloch (2020). GraphH - Source Code. [Online, GitHub; accessed 24-April-2020].

Silva, N. B., R. Tsang, G. D. Cavalcanti, and J. Tsang (2010). A graph-based friend recommendation system using genetic algorithm. In *IEEE congress on evolutionary computation*, pp. 1–7. IEEE.

Simmhan, Y., A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna (2014). GoFFish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pp. 451–462. Springer.

Simmhan, Y., A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna (2017). GoFFish - Source Code. [Online, GitHub; accessed 24-April-2020].

Slota, G. M., K. Madduri, and S. Rajamanickam (2014). PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *2014 IEEE International Conference on Big Data (Big Data)*, pp. 481–490. IEEE.

Smola, A. and S. Narayanamurthy (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment 3*(1-2), 703–710.

Soudani, N. M., A. Fatemi, and M. Nematbakhsh (2019). An investigation of big graph partitioning methods for distribution of graphs in vertex-centric systems. *Distributed and Parallel Databases*, 1–29.

Sparsity Technologies (2015). Scalable high-performance graph database. [Online, GitHub; accessed 24-April-2020].

Stanton, I. and G. Kliot (2012). Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1222–1230.

Stardog (2020). Enterprise Knowledge Graph platform. [Online; accessed 24-April-2020].

Stutz, P., A. Bernstein, and W. Cohen (2010). Signal/collect: graph algorithms for the (semantic) web. In *International Semantic Web Conference*, pp. 764–780. Springer.

Stutz, P., M. Verman, L. Fischer, and A. Bernstein (2013). TripleRush: a fast and scalable triple store. [Online; accessed 24-April-2020].

Sundaram, N., N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey (2015). GraphMat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*.

Sundaram, N., N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey (2017). GraphMat - Source Code. [Online, GitHub; accessed 24-April-2020].

Surveillances, V. (2020). The epidemiological characteristics of an outbreak of 2019 novel coronavirus diseases (COVID-19)—China, 2020. *China CDC Weekly 2*(8), 113–122.

Systap (2020). Blazegraph High Performance Graph Database. [Online; accessed 24-April-2020].

Tantawi, A. N. (2016, July). Solution Biasing for Optimized Cloud Workload Placement. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 105–110.

Tatbul, N., U. Çetintemel, and S. Zdonik (2007). Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pp. 159–170. VLDB Endowment.

Tesoriero, C. (2013). *Getting started with OrientDB*. Packt Publishing Ltd.

Thomas Schmidts, Jan Steemann, F. C. (2020). ArangoDB - Source Code. [Online, GitHub; accessed 24-April-2020].

Tian, Y., A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson (2013, November). From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow. 7*(3), 193–204.

TigerGraph (2020). The World's Fastest and Most Scalable Graph Platform. [Online; accessed 24-April-2020].

Tsourakakis, C., C. Gkantsidis, B. Radunovic, and M. Vojnovic (2014). Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pp. 333–342.

Twitter, Inc. (2020). Quarterly results. [Online; accessed 05-May-2020].

Tärneberg, W., A. Mehta, E. Wadbro, J. Tordsson, J. Eker, M. Kihl, and E. Elmroth (2017). Dynamic application placement in the mobile cloud network. *Future Generation Computer Systems 70*, 163 – 177.

UC Berkeley, MIT, and Databricks (2020). GraphFrames - Source Code. [Online, GitHub; accessed 24-April-2020].

Unsalan, C. and B. Sirmacek (2012). Road network detection using probabilistic and graph theoretical methods. *IEEE Transactions on Geoscience and Remote Sensing 50*(11), 4441–4453.

Vaikuntam, A. and V. K. Perumal (2014). Evaluation of Contemporary Graph Databases. In *Proceedings of the 7th ACM India Computing Conference*, COMPUTE '14, New York, NY, USA, pp. 6:1–6:10. ACM.

van Rest, O., S. Hong, J. Kim, X. Meng, and H. Chafi (2016). Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pp. 1–6.

Vaquero, L., F. Cuadrado, D. Logothetis, and C. Martella (2013). xDGP: A dynamic graph processing system with adaptive partitioning. *arXiv preprint arXiv:1309.1049*.

Vassilevich, D. V. (2003). Heat kernel expansion: user's manual. *Physics reports 388*(5-6), 279–360.

Vega, D., L. Cerdà-Alabern, L. Navarro, and R. Meseguer (2012, Oct). Topology patterns of a community network: Guifi.net. In *2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 612–619.

Vega, D., R. Meseguer, G. Cabrera, and J. M. Marquès (2014, October). Exploring local service allocation in community networks. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference on*, pp. 273–280. IEEE.

VESoft Inc. (2020). Negula Graph - Source Code. [Online; accessed 24-April-2020].

Vora, K., R. Gupta, and G. Xu (2017). KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, New York, NY, USA, pp. 237–251. ACM.

Waltman, L. and N. J. van Eck (2013). A smart local moving algorithm for large-scale modularity-based community detection. *The European Physical Journal B 86*(11), 1–14.

Wang, G., W. Xie, A. J. Demers, and J. Gehrke (2013). Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, Volume 13, pp. 3–6.

Wang, S., R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung (2017, April). Dynamic Service Placement for Mobile Micro-Clouds with Predicted Future Costs. *IEEE Transactions on Parallel and Distributed Systems 28*(4), 1002–1016.

Wang, Y., A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens (2016). Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12.

Wang, Y., A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens (2020). Gunrock - Source Code. [Online, GitHub; accessed 24-April-2020].

Wang, Y., Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, et al. (2017). Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC) 4*(1), 1–49.

Wang, Z., A. Scaglione, and R. J. Thomas (2010). Electrical centrality measures for electric power grid vulnerability analysis. In *49th IEEE conference on decision and control (CDC)*, pp. 5792–5797. IEEE.

Webber, J. (2012). A Programmatic Introduction to Neo4J. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, New York, NY, USA, pp. 217–218. ACM.

Webber, W., A. Moffat, and J. Zobel (2010, November). A Similarity Measure for Indefinite Rankings. *ACM Trans. Inf. Syst. 28*(4), 20:1–20:38.

WhatsApp Inc. (2016). One billion. [Online; accessed 05-May-2020].

Winter, M., D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger (2018). faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 754–766. IEEE.

Xiao, W., J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou (2017). Tux^2: Distributed Graph Computation for Machine Learning. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 669–682.

Xin, R. S., J. E. Gonzalez, M. J. Franklin, and I. Stoica (2013). GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, New York, NY, USA, pp. 2:1–2:6. ACM.

Yan, D., Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang (2017). Graphd: Distributed vertex-centric graph processing beyond the memory limit. *IEEE Transactions on Parallel and Distributed Systems 29*(1), 99–114.

Yang, K. and L. Toni (2018). Graph-based recommendation system. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 798–802. IEEE.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica (2010). Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA, pp. 10–10. USENIX Association.

Zerbino, D. and E. Birney (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res 18*, 821.

Zhang, K., R. Chen, and H. Chen (2015). NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 183–193.

Zhang, K., R. Chen, and H. Chen (2018). Polymer - Source Code. [Online, GitHub; accessed 24-April-2020].

Zhang, Y., X. Liao, H. Jin, L. Gu, and B. B. Zhou (2017). FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping. *IEEE Transactions on Knowledge and Data Engineering 30*(5), 895–907.

Zhao, B., H. Zhou, G. Li, and Y. Huang (2018). ZenLDA: Large-scale topic model training on distributed data-parallel platform. *Big Data Mining and Analytics 1*(1), 57–74.

Zhao, H., Q. Yao, J. Li, Y. Song, and D. L. Lee (2017). Meta-graph based recommendation fusion over heterogeneous information networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 635–644.

Zheng, D., D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay (2015). FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pp. 45–58.

Zheng, Da and Mhembere, Disa and Burns, Randal and Vogelstein, Joshua and Priebe, Carey E and Szalay, Alexander S (2014). FlashGraph - Source Code. [Online, GitHub; accessed 24-April-2020].

Zhu, X., W. Chen, W. Zheng, and X. Ma (2016). Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 301–316.

Zhu, X. and Z. Ghahramani (2002). Learning from labeled and unlabeled data with label propagation. [Online; accessed 24-April-2020].