Co-funded by the
Erasmus+ Programme
of the European Union

Performance Optimization Techniques and Tools for Distributed Graph Processing

VASILIKI KALAVRI

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016
and
Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium, 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:25
ISBN 978-91-7729-101-5

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen i Informations- och kommunikationsteknik måndagen den 10 oktober 2016 klockan 13:00 i Sal C, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

To all my teachers and mentors

Abstract

Distributed, shared-nothing architectures of commodity machines are a popular design choice for the implementation and deployment of big data platforms. The introduction of MapReduce, a simple programming model for parallel data analysis, has greatly simplified data-parallel programming by abstracting the details of data partitioning, node communication, and fault tolerance. While MapReduce is a powerful model for simple tasks, such as text processing and web log analysis, it is a poor fit for more complex tasks, such as graph analysis. Inter-connected data that can be modeled as graphs appear in several application domains, including machine learning, recommendation, web search, and social network analysis. Graph algorithms are very diverse and expose various computation and communication patterns. MapReduce revolutionized the area of distributed data processing and inspired the development of similar high-level graph processing models and platforms. However, as graphs grow bigger, delivering high performance for graph analysis tasks becomes challenging. Existing distributed graph processing platforms often deliver disappointing performance, while demanding expensive resources, as compared to sequential or multi-threaded algorithms running on a single machine.

Processing graphs on a single machine is often not a viable solution. First of all, graphs rarely appear as raw data. Instead, they are derived from processing, filtering, and transformation of other, often distributed, data sources. Second, graph analysis tasks are usually part of a larger data analysis pipeline. Thus, previous and succeeding processing might require distribution over several machines. Finally, the nature of the graph analysis problem might require distribution. Thus, developing optimization techniques and tools to improve the performance of distributed graph processing platforms is essential.

In this thesis, we propose optimization techniques for distributed graph processing on general-purpose data processing engines and on specialized graph processors. Our optimizations leverage both data and algorithmic properties. Driven by a real-world graph problem, we design performance optimization techniques and tools. First, we describe a data processing pipeline that leverages an iterative graph algorithm for automatic classification of web trackers. Using this application as a motivating example, we examine how *asymmetrical convergence* of iterative graph algorithms can be used to reduce the amount of computation and communication in large-scale graph analysis. We propose an optimization framework for fixpoint algorithms and a declarative API for writing fixpoint applications. Our framework uses a cost model to automatically exploit asymmetrical convergence and evaluate execution strategies during runtime. We show that our cost model achieves speedup of up to 1.7x and communication savings of up to 54%. Next, we propose to use the concepts of *semi-metricity* and the *metric backbone* to reduce the amount of data that needs to be processed in large-scale graph analysis. We provide a distributed algorithm for computing the metric backbone using the vertex-centric programming model. Using the backbone, we can reduce graph sizes up to 88% and achieve speedup of up to 6.7x.

Sammanfattning

Distribuerade shared-nothingärkitekturer för vanligt förekommande datorer är ett populärt designval för att implementera big data-plattformar. Introduktionen av programmeringsmodellen MapReduce, en enkel programeringsmodell för parallell dataanalys, gjorde det möjligt att processa stora mängder data parallellt genom att abstrahera detaljerna för uppdelning av data, kommunikation mellan noder, och feltolerans. Men även om MapReduce är bra för enkla uppgifter som text och logganalys är det betydligt svårare att använda för analys av mera komplexa datatyper som inom grafanalys. Sammankopplade data som kan modelleras som en graf är vanligt förekommande i flera domäner, bland annat maskininlärning, rekommendationssystem, webb-sökning, och sociala nätverk. Dessutom skiljer sig grafalgoritmer avsevärt från varandra, både sett ur beräknings och kommunikationsperspektiv. MapReduce revolutionerade distribuerad databehandling och inspirerade utvecklingen av liknande modeller och plattformar för analys av grafer. Allteftersom graferna har vuxit i storlek har utmaningarna följt efter. Plattformarna för parallell grafanalys som finns idag lever inte upp till de prestandakrav som ställs och kräver allt för stora beräkningsresurser jämfört med dess sekventiella eller multi-trådade motsvarigheter.

Tyvärr är en maskin sällan tillräckligt för att behandla stora grafer. Först och främst förekommer sällan grafer som rådata. Istället tas graferna fram genom att behandla, filtrera, och omvandla ofta distribuerade datakällor. Därutöver är grafanalysen ofta bara en liten del i en större kedja. Slutligen behöver man kunna sprida ut behandlingen över flera maskiner. Därför behöver man utveckla tekniker och verktyg för att förbättra prestandan i distribuerade plattformar för grafanalys.

I den här avhandlingen presenterar vi op timeringar för distribuerad grafbehandling på såväl vanliga datorer och servrar som påspecialiserade grafprocessorer. Våra förbättringar utnyttjar både grafens struktur och algoritmiska egenskaper. Designen av teknikerna och verktygen härstammar från ett realistiskt grafproblem. Avhandlingen börjar med att presentera en pipeline som använder en iterativ grafalgoritm för att automatiskt gruppera webb-trackers. Detta exempel står som grund för en utvärdering av hur såkallad asymmetrisk konvergens kan användas i iterativa grafalgoritmer för att minska antalet beräkningar i stora grafanalyser. Vi föreslår ett optimeringsramverk för fixpunktsalgoritmer och ett deklarativt API för att utveckla fixpunktstillämpningar. Vårt ramverk anvö nder en kostnadsbaserad optimerare som automatiskt kan utnyttja asymmetrisk konvergens och utvärdera exekveringsstrategin när tillämpningen körs. Vi visar att den kostnadsbaserade modellen förbättrar prestandan upp till 1.7 gånger och minskat kommunikationsbehov med 54%. Därefter föreslår vi att man kan förminska datamängden som behäver processas i stora grafer genom begrepp som semimetricitet och *metric backbone*. Vi presenterar en distribuerad algoritm för att beräkna den senare genom att använda programmeringsmodellen som utgår från varje båge i grafen. Resultaten visar att grafens storlek minskar med upp till 88% och en prestandavinst på upp till 6.7 gånger jämfört med dagens alternativ.

Résumé

Des architectures distribuées sans partage sont un choix de conception populaire pour la mise en œuvre et le déploiement de grandes plates-formes de données. L'introduction de MapReduce a grandement simplifié la programmation parallèle de données en faisant abstraction du partitionnement des données, de la communication entre noeuds, et de la tolérance aux pannes. Alors que MapReduce est un modèle puissant pour des tâches simples telles que le traitement de texte et l'analyse de trafic web, il est un mauvais ajustement pour des tâches plus complexes telles que l'analyse des graphes. Des données interconnectées qui peuvent être modélisées sous forme de graphes apparaissent dans plusieurs domaines d'applications, y compris l'apprentissage automatique, les systèmes de recommandation, la recherche sur le Web, et l'analyse des réseaux sociaux. Les algorithmes d'analyse de graphes sont très diverses et exposent divers modèles de calcul et de communication. MapReduce a révolutionné le domaine du traitement de données distribuées et a inspiré le développement des modèles et plate-formes de traitement à haut niveau. Cependant, quand la taille des graphes augmente, offrir des performances élevées pour les tâches d'analyse devient difficile. Les systèmes distribués actuels de traitement de graphes offrent souvent des mauvaises performances, tout en exigeant des ressources coûteuses par rapport aux algorithmes séquentiels ou multi-thread qui s'exécutent sur une seule machine.

Souvent, le traitement des graphes sur une seule machine n'est pas une solution viable. Tout d'abord, les graphes apparaissent rarement comme données brutes. Au contraire, elles sont obtenues par la transformation, le filtrage et la transformation d'autres sources de données, souvent distribuées. Deuxièmement, l'analyse de graphes fait généralement partie d'un plus grande pipeline de traitement de données. Le traitement précédent et suivant peuvent nécessiter la distribution sur plusieurs machines. Enfin, la nature du problème d'analyse pourrait nécessiter la distribution. En conclusion, il est essentiel de développer des techniques et outils d'optimisation pour améliorer la performance des systèmes de traitement de graphes distribués.

Dans cette thèse, nous proposons des techniques d'optimisation pour le traitement des graphes distribués sur des plate-formes d'analyses générales et spécialisées. Nos optimisations exploitent des propriétés des données et des propriétés des algorithmes. Motivé par un problème dans le monde réel, nous concevons des techniques et des outils d'optimisation de la performance. Tout d'abord, nous décrivons un pipeline de traitement de données qui exploite un algorithme de graphe itérative pour la classification automatique des trackers web. En utilisant cette application comme un exemple de motivation, nous examinons comment la convergence asymétrique des algorithmes de graphe itératives peut être utilisée pour réduire la quantité de calcul et de communication dans l'analyse à grande échelle. Nous proposons un cadre d'optimisation pour des algorithmes de point fixe et une API déclarative pour écrire des applications de point fixe. Notre cadre utilise un modèle de coût pour exploiter automatiquement la convergence asymétrique et évaluer des stratégies d'exécution lors de l'exécution. Nous montrons que notre modèle de coût atteint une augmentation de vitesse jusqu'à 1.7x et une réduction du coût de communication jusqu'à 54%. Ensuite, nous proposons d'utiliser les concepts de sémi-métricité et de dorsale métrique pour réduire la quantité de données que doit être traitée dans l'analyse des graphes à grande échelle. Nous proposons un algorithme distribué pour calculer la dorsale métrique en utilisant le modèle de programmation vertex-centrique. En utilisant la dorsale mètrique, nous pouvons réduire la taille des graphes jusqu'à 88% et obtenir une augmentation de vitesse jusqu'à 6,7x.

Acknowledgment

I am deeply thankful to the following people, without the help and support of whom, I would not have managed to complete this work:

My primary advisor Vladimir Vlassov, for his guidance, constant feedback and encouragement throughout this work.

My secondary advisors, Peter Van Roy, Christian Schulte, and Seif Haridi for their valuable advice, insights, and support.

Magnus Boman for agreeing to serve as the internal reviewer of this thesis. His experience and invaluable feedback significantly helped improve the final version of this document.

My co-authors and collaborators, Per Brand, Hui Shang, Vaidas Brundza, Stephan Ewen, Kostas Tzoumas, Volker Markl, Jeremy Blackburn, Matteo Varvello, Dina Papagiannaki, Tiago Simas, and Dionysios Logothetis. Working with them has been a great pleasure and indispensable learning experience.

All anonymous reviewers of accepted and rejected papers, for providing observant and constructive feedback.

My emjd-dc colleagues and colleagues at KTH and UCL, especially Paris Carbone, Ying Liu, Navaneeth Rameshan, Manuel Bravo, Zhongmiao Li, Vamis Xhagjika, Ruma Paul, Hooman Peiro Sajjad, and Kamal Hakizamdeh for always being available to discuss ideas over fika or lunch, for being supportive, and for the good times we spent hanging out together in and out of office.

Colleagues and friends at Telefonica Research, Barcelona, and data Artisans, Berlin, who were so kind to host me during two wonderful internships, allowing me to escape from Swedish winter.

The welcoming cities of Barcelona, Stockholm, Berlin, and Brussels, which hosted me during the past four years and all the wonderful people I met there.

Marcus Ljungblad for translating my abstract in Swedish and Thomas Sjöland for revising it.

Sandra Gustavsson Nylén and Vanessa Maons for taking care of all the complex administrative issues during the course of this joint PhD program.

My family and friends, for their continuous support, their faith in me, their patience and immeasurable understanding.

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent advances in technology and the web, as well as and the decreasing cost of data storage, have motivated all kinds of organizations to capture, store, and analyze vast amounts of data. The term *Big Data* is often used to describe this phenomenon, meaning information that cannot be processed in a practical way, using traditional processes or tools [196], such as relational databases and data warehouses. The term is often used to describe data analysis problems that are defined by certain characteristics, known as the "3 V's of Big Data": *Volume*, *Variety*, and *Velocity* [114]. Volume reflects the fact that the amount of data being collected and analyzed today is incomparable to the amount of data we could collect and store a few years back [171, 124]. Even considering the latest technological advances in hardware, collected data might not fit in the main memory or even on the available disk space of a single machine. Thus, it is often partitioned and distributed in several physical machines [40, 144, 21]. Variety means that data is being collected from all kinds of diverse sources, such as sensor networks, logs, web traffic, images, etc.. Organizations have started storing all kinds of data, with the hope that its analysis and combination with historical data will yield business value. At the same time, data appears in various formats and schemas. Sources like the web, social media, and sensors, produce data in semi-structured or unstructured formats [96, 17, 60]. New technology needs to be developed in order to combine data being represented in different ways, such as raw text, xml, json, streams of clicks, logs, graphics, audio, video. Velocity refers to the rate at which data is generated, collected, and processed. Nowadays, we can generate and capture data at rates we have never experienced before [24, 193, 139]. For example, it is reported that more than 400 hours of video is being uploaded to YouTube every minute [16].

A shared-nothing architecture of commodity machines has proved to be a popular design choice for the implementation and deployment of big data platforms. Clusters of commodity servers are often favored over expensive infrastructure because of their low operational costs [59, 131, 40]. In such clusters, data is partitioned and distributed over several machines, usually leveraging the functionality of a distributed file system [39, 64, 117]. The processing component is often also deployed on the same cluster of machines, in order to leverage data locality; the incentive is to process data on the machine where it is already stored and avoid expensive network transfers.

These ideas were initially introduced by Google's MapReduce paper [59] and adopted by its open-source implementation, Hadoop [7], as well as successor distributed data processing systems. MapReduce proposes a simple programming model for parallel data analysis. It simplifies data-parallel programming by abstracting the details of data partitioning, node communication, and fault tolerance. MapReduce is a powerful model for simple tasks, such as text processing and web log analysis. On the other hand, it is a poor fit for more complex tasks [144, 117, 97], such as machine learning [79, 122], iterative [43, 67, 192], and graph processing [125, 122, 142]. To address these limitations, several other data-intensive frameworks have been developed. Next-generation platforms offer more flexible and efficient runtimes, while aiming to maintain the simplicity and high abstraction level of MapReduce. MapReduce Online [57] implements data pipelining between operators and a simple form of online aggregation. Spark [192] leverages in-memory computation, Dryad [92] uses an arbitrary DAG to describe the application's communication patterns and express data transport mechanisms, Stratosphere [29] offers an extension of the MapReduce programming model with a relational taste and runtime optimizer, ASTERIX [31] offers a storage and computing platform for semi-structured data, HaLoop [43], Twister [67] and CIEL [137] target iterative computations, while Pregel [125] and GraphLab [122] are specialized systems for graph processing.

Building efficient distributed data-intensive computation platforms is a challenging task. Data partitioning, communication, fault-tolerance, scalability, and load-balancing are only a few of the issues that need to be addressed when designing such systems. At the same time, in order to make the power of big data frameworks accessible to non-technical users and relieve the programmer from error-prone, low-level tuning, automatic optimization must be performed [183, 141].

The aforementioned challenges are magnified when aiming to efficiently support complex data processing tasks, rather than simple MapReduce-style applications. In particular, application domains like graph processing are especially interesting. Inter-connected data that can be modeled as graphs appear in several application domains, such as machine learning, recommendation, web search, and social network analysis. Graph algorithms are very diverse and expose various computation and communication patterns. For instance, value-propagation algorithms, like PageRank, require efficient iteration mechanisms, but have predictable communication patterns and can be easily parallelized. On the other hand, graph traversals and path exploration have unpredictable data access patterns and distributed solutions can incur high communication costs.

As graphs grow bigger, delivering high performance for graph analysis tasks becomes even more challenging. Even with the use of specialized graph management systems [125, 81, 122], several graph metrics are painfully slow to compute. As a matter of fact, existing distributed graph processing platforms often deliver disappointing performance, while demanding unreasonable resources, as compared to sequential or multi-threaded algorithms running on a single machine [128, 195, 113]. However, processing on a single machine is often not a viable solution. First of all, graphs rarely appear as *raw* data. Instead, they are derived from processing, filtering, and transforming other, often distributed, data. Second, graph analysis tasks are usually part of a larger data analysis pipeline. Thus, previous and succeeding processing might require distribution over several machines. Finally, the nature of the graph analysis problem might require distribution. For example, when analyzing a social network of users who are *physically* distributed, it might be favorable to process their data at their physical location. Thus, developing optimization techniques and tools to improve the performance of distributed graph processing platforms is essential.

In this thesis, we propose optimization techniques for distributed graph processing on general-purpose data processing engines and on specialized graph processing systems. Our optimizations leverage both data and algorithmic properties. We begin our study by reviewing general-purpose distributed data processing platforms, like MapReduce and its successors. Then, we focus our study on systems and programming abstractions for distributed graph processing. Driven by a real-world graph problem of web tracker classification, we identify optimization opportunities and then proceed to design methods, algorithms, and tools that implement optimization techniques. Finally, we apply our proposed optimization techniques to the algorithms of the web tracker classification problem and we evaluate their effectiveness.

1.1 Research Objectives

We research methods and we build tools for efficient distributed processing of very large graphs. In particular, we aim at making existing graph processing platforms faster, by inventing and implementing optimization techniques. Our main goal is to improve performance, with respect to total execution time and resource requirements. To this end, we analyze existing models and infrastructure, identify limitations, and design concrete optimization techniques that we integrate in popular graph processing platforms. Our secondary goal is to facilitate the development of distributed graph applications and provide automatic optimization that requires minimal user involvement. We aim to make big-graph analytics accessible to non-experts, so that users with limited programming experience and understanding of distributed architectures can benefit from analyzing enormous graph datasets. In this direction, we design our optimization methods to be automatic, with the objective to relieve the programmer from low-level, error-prone instrumentation.

From a high-level view, there are three main factors that dominate the execution time of a distributed data-intensive graph application, assuming stable execution environment conditions and a fixed number of available resources: (a) the input data size, (b) the amount of computations performed, and (c) the amount of communication. Thus, in order to improve performance, we define three main corresponding goals:

1. to reduce the *size of datasets* to be processed,
2. to reduce the *amount of total computation* to be performed, and
3. to reduce the *amount of communication* required.

With respect to these objectives, we first present a study of distributed data processing with MapReduce (Chapter 2) and a study on distributed graph processing abstractions and systems (Chapter 3). Then, we describe a real-world use-case of large-scale graph analytics (Chapter 4). This work exposes limitations in existing implementations and reveals concrete optimization opportunities. In Chapter 5, we propose a framework for automatic optimization of iterative fixpoint graph algorithms. Our optimization framework leverages properties of the fixpoint update functions, in order to reduce the computation and communication required in each iteration step. With respect to the goal of reducing the size of processed datasets, in Chapter 6 we leverage the concept of the metric backbone [154, 169], in the context of large-scale graph analytics. The metric backbone is a reduced representation of a weighted graph, that preserves information about connectivity and distances.

We also define the following design directives for the optimizations presented in this thesis:

— **Application transparency**. It is our goal to make all proposed optimization techniques applicable to existing applications. Ideally, the user should be able to benefit from the optimization, without having to re-write or re-compile their application. In order to achieve this goal, we try to design our optimizations without changing the systems' application programming interface and programming model.

— **System independence**. We strive to make our optimizations as generic as possible, so that they are applicable to a variety of graph processing systems. The optimizations presented in this thesis are implemented in two different, widely used computation platforms; a general-purpose distributed data processing engine and a specialized graph system. To meet this design objective, we decouple our methods from system implementations.

— **Minimum user involvement**. In order to relieve users from error-prone and low-level tuning, optimizations should minimize user involvement. With respect to this goal, we automate the optimization techniques and avoid manual configuration, whenever possible.

1.2 Research Methodology

In this section, we provide a high-level view of the methods used in this research work. We give a summary of the general principles we followed and the design decisions we made in order to achieve our goals. We also discuss several challenges we faced and how we chose to overcome each if these challenges.

1.2.1 General Approach

Among the wide variety of data-intensive applications and platforms, we focus on graph algorithms and distributed graph processing systems for two reasons. First, graph algorithms are vital to a plethora of modern analytics domains, such as social networks, machine learning, anomaly detection, recommender systems, bioinformatics, web security and privacy. Second, distributed graph processing presents interesting research challenges and open issues. Distributed graph applications are complex and hard to implement efficiently. Moreover, they expose diverse computation and communication patterns.

With respect to our main goal of speeding up large-scale graph analytics, we design and apply *performance* improvements. More specifically, we focus on performance optimizations for distributed, batch-processing, computation platforms, including general-purpose data processing engines and specialized graph processors. We approach the problem of

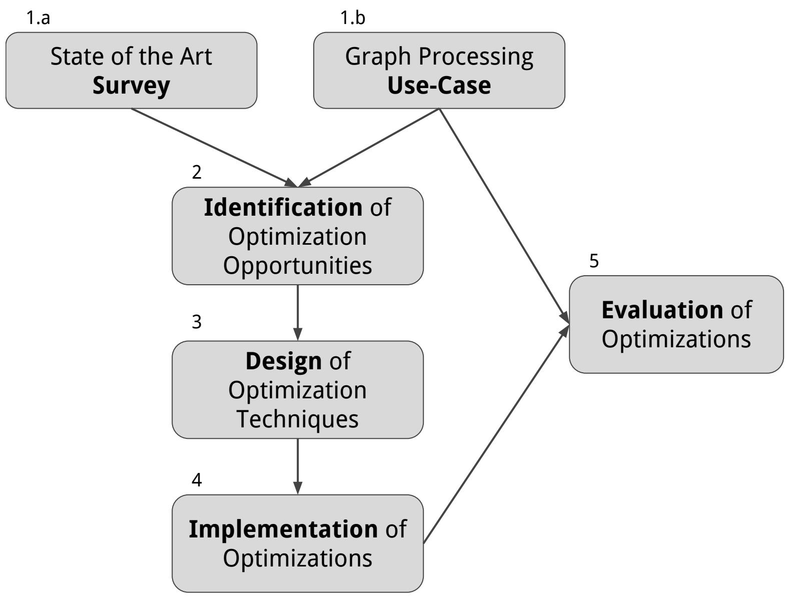

1.a
State of the Art Survey
1.b
Graph Processing Use-Case
2
Identification of Optimization Opportunities
3
Design of Optimization Techniques
4
Implementation of Optimizations
5
Evaluation of Optimizations

Figure 1.1 – A high-level overview of the thesis research methodology.

performance enhancement from two main angles. First, we choose a common application category of graph algorithms, namely fixpoint, value-propagation algorithms. Second, we focus on graphs with commonly encountered characteristics, namely weighted graphs; graphs with attached properties on their edges. We show the applicability of our choices and draw our motivation from a real-world use-case. We focus on techniques that minimize resource requirements and avoid redundancy in computations and data, by targeting:

— **Algorithmic Properties**. We explore how the algorithms we execute can be optimized, so that we avoid redundant computations. For example, the asymmetrical convergence of fixpoint algorithms is exploited to detect inactive parts of the graph and reduce the amount of computations and communication required.

— **Data Properties**. We explore whether the data we process have properties that we can leverage to reduce execution time. For example, we exploit graph semi-metricity to reduce the size of the input graph, while still achieving exact results for applications that depend on the shortest-paths or approximate results for algorithms that do not.

We adopt an empirical approach, common to computer systems research, instead of using analytical, mathematical optimization methods. A high-level overview of our methodology is shown in Figure 1.1. The numbers represent chronological order. First, we identify performance bottlenecks and limitations in data-intensive graph computation platforms and then we design and implement techniques to overcome these limitations. We start by conducting a literature study of recent research results on the limitations and optimizations for big data platforms (Chapter 2). In particular, we focus on platforms implementing or extending the MapReduce programming model. This model revolutionized the area of distributed data processing and inspired the graph processing models and platforms that we target in this work. The results of this study provide us with an overview of the state of the art in the research field and reveal open issues. Next, we review the programming abstractions, architecture, and implementation of modern distributed graph processing platforms (Chapter 3). Most of these platforms are MapReduce successors, and thus, they are often heavily inspired by its design principles. This study helps us identify the techniques developed to extend the MapReduce model to meet the requirements of graph processing applications. In order to invent useful and practical optimizations, we center our research around a specific, real-world graph application. We perform a study in the area of web tracker detection and privacy on the web and we define the problem of automatically detecting web trackers as a graph problem (Chapter 4). We use a real dataset from a large European telecom operator and we implement a data pipeline that solves the problem using an iterative fixpoint graph algorithm. We confirm the presence of performance shortcomings by experiment and we design techniques to eliminate them (Chapters 5 and 6). We implement each optimization technique, while trying to meet our design goals of application transparency, system independence, and minimum user involvement. Finally, we evaluate our implementations by comparing the performance of a modified system, which uses our optimization technique, to the performance of the original unmodified system. For the evaluation, we use the motivating web tracker detection algorithm, as well as a set of additional representative applications and real-world or publicly available datasets. We use job execution time as our main performance measure, add we define secondary measures, such as communication load, where applicable.

1.2.2 Implementations

We use open-source, widely-used, and mature systems and libraries to implement and evaluate our optimization techniques. Specifically, we use Apache Flink [1] to implement the graph processing use-case of Chapter 4 and the cost-based optimization method for fixpoint iterations presented in Chapter 5. We use the Apache Giraph [2] graph processing system and the Neo4j [12] graph database to implement the metric backbone algorithm and evaluate the methods described in Chapter 6. The implementations of our optimization techniques are free to use, open-source, and documented. More details can be found in Section 1.3.2.

1.2.3 Experimental Evaluation

For our experiments, we always choose the latest -at the time of the research work-stable version of the system considered. We use publicly available and real-world datasets, and share our detailed setup configuration in each of our works, in order to facilitate reproducibility. We conducted most of our experiments using virtual machines in a cloud environment. Thus, we are able to create a clean, isolated environment, where only the necessary tools were installed. We choose representative applications for our evaluation, by either using applications that appear often in related research or by implementing algorithms that can demonstrate the benefits of our optimization methods.

1.2.4 Optimizations Overview

In this section, we present a categorization of optimization types and methods for distributed data processing. Most of the available techniques have been inspired by previous research conducted in the context of relational database systems. We categorize optimizations based on their main objective and we describe popular existing techniques that fit into each category. We refer the reader to Chapter 2 for a detailed overview of the state of the art in optimization for data-intensive computation platforms.

1.2.5 Optimization Types and Methods

We categorize optimizations based on their main objective, namely **Performance**, **Ease-of-use** and **Cost reduction**. Performance optimizations aim at reducing job execution time and making data processing faster. The goal is to provide faster answers to user queries. Ease-of-use optimizations aim at making data processing easier, from a user's perspective. These techniques intent to automate the complex parts of data analysis and facilitate application development and deployment. Cost reduction optimizations are to methods whose purpose is minimizing operating costs of data analysis systems.

— **Performance Optimizations**. Techniques in this category can be further divided into two main subcategories: (a) single-program optimizations and (b) batch optimizations. Single-program optimizations target a single application at a time. These techniques consider the characteristics of a specific application when executed in isolation. Thus, these techniques do not take into account interactions with other applications running at the same time and do not consider possible optimization opportunities in this context. Batch optimizations are techniques that intend to optimize a workload of applications as a whole. These techniques view the system as an environment where applications are running concurrently or during a specified period of time.

  Single-program optimizations. A lot of techniques in this category are based on extracting information about the input and intermediate datasets. These optimizations aim at inventing more efficient ways of executing a query. Typically, a user writes a data processing application in some language and the data processing system is responsible for defining an *execution strategy* for running this application, using the available resources. The process of translating the user's application into a series of steps that the system can execute is called *query planning*. Widely used optimizations in this category concern choosing an efficient query execution plan. These techniques usually involve methods for accurate dataset size estimation and estimation of key cardinalities. Static code analysis is another popular method used for this purpose. An example of a concrete query plan optimization technique is operator reordering.

  Another class of optimization techniques in this category is concerned with how to efficiently access data in the processing system's storage. The goal is to minimize communication and random data accesses. In order to reduce data I/O, several smart data placement and partitioning strategies have been developed. For example, one

can predict which data blocks are likely to be accessed at the same time and try to co-locate them. Related issues include efficient data representation, compression techniques, erasure coding and amortizing data skew.
Result approximation is an optimization technique that is very useful when the amount of data to be processed is very large. The goal of this class of techniques is to return an approximate result to the user query, as soon as possible, even before the analysis has finished. Partial job execution can yield very fast results but it has to be paired with a robust results estimation method. Sampling histogram building and wavelets are popular choices in this area. Regarding sampling, techniques can utilize online sampling or use pre-cached samples of the input or pre-processed datasets. These techniques are especially efficient and accurate when the queries are known beforehand and the system has adequate information about the data distribution.

Batch optimizations. The main goal of batch optimization techniques is to increase the performance of an execution environment, such as a cluster or datacenter, as a whole. These techniques often consider metrics like system throughput and query latency. A big class of batch optimization techniques aim at efficiently scheduling and managing applications running in the same environment. In distributed setups, load balancing is another very important issue that significantly affects performance and it is also very closely related to scheduling. Another popular technique makes use of work sharing for simultaneous queries. This optimization considers a batch of jobs submitted for execution and re-orders them to enable sharing of execution pipelines. A related technique is sometimes referred to as non-concurrent work-sharing and aims at avoiding redundant computations by materializing and reusing results of previously executed jobs.

— **Ease-of-use** Ease-of-use optimizations aim at facilitating non-expert users in conducting data analysis. These techniques mainly include tools that analysts can use to boost their productivity and avoid bad development practices that may lead to buggy programs. Tools for easy application development automate parts of the analysis process and abstract low-level details from the user, such as parallelization and data distribution. Another set of techniques facilitates the deployment and maintenance of analysis applications on complex environments. They might perform automatic system configuration and load balancing or handle fault-tolerance. Related tools facilitate testing and debugging of data analysis programs.

— **Cost reduction** This category of optimizations primarily intends to minimize the cost of data processing. Such optimizations include tools for reducing storage requirements, by using data deduplication techniques or erasure coding. A big class of cost reduction optimizations also concerns resource and cost-aware scheduling. By assumption, the system has information about the cost of different resources and can evaluate different execution strategies and deployments, in order to efficiently utilize resources and reduce operational costs. Resource utilization optimizations usually also benefit the performance and throughput of a system.

We summarize the discussed optimization types and methods in Table 1.1.

Table 1.1 – Optimization Types and Methods for Data-Intensive Computation Platforms

Performance	Single-Program	Query Planning I/O Reduction Results Approximation
	Batch	Scheduling Concurrent Work Sharing Results Materialization and Reuse
Ease-of-use	Development Tools Deployment Tools Automatic Configuration Debugging Tools	
Cost	Resource Utilization Cost-aware Scheduling	

1.2.6 Challenges

In this section, we highlight some of the challenges we faced throughout this work, while we expand on the details of problem-specific challenges in the corresponding chapters.

While conducting the state-of-the-art survey on distributed data processing with MapReduce (Chapter 2), we discovered that a lot of the published research in the area lacked open-source or available to use implementations. Even after contacting corresponding authors, we failed to gain access to most of the studied frameworks. As a result, it was impossible to perform an empirical evaluation and we rather focused on a qualitative comparison of system features and optimizations. Another challenge that needs to be addressed when conducting systems comparison is that researchers use widely different environments and applications to evaluate their work. Thus, it is very difficult to compare systems fairly.

A different set of challenges appeared when implementing the graph processing use-case described in Chapter 4. Evaluating the classification accuracy of our solution required some ground truth. We have used the easyprivacy list [6], which was also used to tag the input records of our classifiers. Therefore, even though our classification results reflect how close our solution is to the accuracy of the list, it was impossible to quantify whether our solution could identify trackers that do not appear on the list. Another difficult task was the visualization of large graph datasets. Currently available software is practical to use for input graphs in the order of hundreds of thousands of edges, but struggle to scale for the input graphs that we study in this thesis. When we had to visualize a graph, we created subgraphs that only contained the useful information for the particular case. For example, in Chapter 4, we only visualize the subgraph of a single month of user data and filter out all edges that do not have a tracker endpoint.

We also faced several challenges related to the platforms we used for the implementation of our optimizations, primarily on appropriately configuring and tuning the system parameters for experimentation. Fortunately, both Apache Giraph [2] and Apache Flink [1] have welcoming and helpful open-source communities, whose members provided us with valuable guidance.

1.3 Thesis Contributions

1.3.1 Summary of Contributions

The contributions of this thesis are as follows.

— A survey of the state-of-the-art in Hadoop/MapReduce optimizations. We compare and classify frameworks that adopt and extend the MapReduce programming model and we summarize the state of the research field, while identifying open issues and possible future research directions.

— An overview of MapReduce-inspired, high-level programming abstractions for distributed graph processing. We analyze their execution semantics, user-facing APIs, extensions and proposed optimizations. We identify which classes of applications can be intuitively expressed with each model and which computational patterns might be problematic to express with certain abstractions. We categorize modern distributed graph processing systems, with regard to the graph programming abstractions they expose, their execution models, and their communication mechanisms.

— A study of web tracker behavior and user exposure to the tracking ecosystem, as a representative, real-world graph processing application. We use this study to concretely reason about optimization opportunities, which are general enough to be applied in a wide class of real-word graph applications. We build a classifier that models user access logs as graphs and analyzes those graphs to automatically classify a web request as a tracker or non-tracker. We use the classification algorithms as a motivating example of a real-world, distributed graph processing application that drives the optimization methods developed in this thesis.

— An optimization framework for fixpoint iterative graph processing. We review four fixpoint iteration techniques and we formally describe the necessary conditions, under which each technique can be applied. We map iteration techniques to existing graph processing abstractions and we explain how missing techniques can be implemented. We implement template optimized execution plans, using general dataflow operators and we experimentally evaluate each optimization, using a common runtime. We develop a declarative fixpoint API that supports all four fixpoint iteration techniques and we implement a cost model that evaluates alternative execution strategies. Our framework uses the cost model to automatically switch execution plans during runtime and avoid redundant computations and communication.

— A distributed sparsification algorithm to compute the metric backbone for speeding up computations in large-scale graph analysis. We analyze a variety of real graphs and show that they exhibit high semi-metricity. We categorize the types of applications which benefit from the metric backbone and we propose a practical algorithm for calculating the metric backbone, in a distributed setting. We apply the approach inside two graph management systems and evaluate their efficiency with real datasets. We show that our technique can speed up typical graph queries and improve the performance of several graph applications.

1.3.2 Software

The following software was developed in the course of this thesis.

— Gelly [8], a graph processing library and API for Apache Flink [1]. The author developed the initial version of the Gelly library and API and continues to be a core developer and maintainer of the module. Gelly has been the official graph processing library of Apache Flink since its 0.9 release and can be downloaded from the Apache Flink website [1]. Since then, several Apache Flink community members have contributed library algorithms, optimizations, and documentation to Gelly. In this thesis, we describe Gelly in Chapter 3 and we use its API and library methods to implement the data processing pipeline for automatically detecting web trackers (Chapter 4).

— A framework for fixpoint applications on Apache Flink. The framework provides a high-level Java API for writing fixpoint applications and a cost model that automatically identifies and avoids redundant computations. The framework is described in Chapter 5 and it is available under the Apache 2.0 license [1].

— A distributed algorithm for computing the metric backbone in Apache Giraph [2]. We describe the algorithm in detail in Chapter 6. The algorithm is implemented using the vertex-centric programming model and it is available in the Okapi open source library for machine learning and graph analytics [2].

1.3.3 Publications

Results presented in this thesis have been published as papers in international conference proceedings and workshops as follows.

— *The shortest path is not always a straight line: Leveraging semi-metricity in graph analysis.* Kalavri, Vasiliki; Simas, Tiago; Logothetis, Dionysios; Proceedings of the VLDB Endowment, Vol. 9, No. 9 (PVLDB), 2016

— *Like a Pack of Wolves: Community Structure of Web Trackers.* Kalavri, Vasiliki; Blackburn, Jeremy; Varvello, Matteo; Papagiannaki, Konstantina; 17th International Conference on Passive and Active Measurement. Vol. 9631. Springer, 2016.

— *Asymmetry in large-scale graph analysis, explained.* Kalavri, Vasiliki; Ewen, Stephan; Tzoumas, Kostas; Vlassov, Vladimir; Markl, Volker; Haridi, Seif; Proceedings of Workshop on GRAph Data management Experiences and Systems, 1-7, (GRADES), 2014.

— *MapReduce: Limitations, Optimizations and Open Issues.* Kalavri, Vasiliki; Vlassov, Vladimir; 11th IEEE International Symposium on Parallel and Distributed Processing with Applications 2013.

1. http://github.com/vasia
2. http://grafos.ml/#Okapi

1.3.4 Other Contributions

Other related contributions that are not included in this thesis but were developed during the course of the author's PhD studies can be found in the author's Licentiate thesis [98]. These include the following.

— PonIC, a project that ports the high-level dataflow framework Apache Pig [78] on top of the data-parallel computing framework Stratosphere [23]. In this work, we show that Pig can highly benefit from using Stratosphere as the backend system and gain performance, without any loss of expressiveness. We identify the features of Pig that negatively impact execution time and present a way of integrating it with different backends. We also propose a complete translation process of Pig plans into Stratosphere plans and we evaluate PonIC.

— HOP-S, a system that uses in-memory random sampling to return approximate, yet accurate query results. We propose a simple, yet efficient random sampling technique implementation, which significantly improves the accuracy of online aggregation in MapReduce Online [57]. MapReduce Online is a system that uses online aggregation to provide early results, by partially executing jobs on subsets of the input, using a simplistic progress metric.

— An optimization that exploits computation redundancy in analysis programs. We have built m2r2, a system that stores intermediate results and uses plan matching and rewriting in order to reuse results in future queries. We have built our prototype on top of the Apache Pig framework and report significantly reduced query response times.

These contributions appear in publications [100, 102, 105] as follows.

— *Ponic: Using stratosphere to speed up pig analytics.* Kalavri, Vasiliki; Brand, Per; Vlassov, Vladimir; European Conference on Parallel Processing. Springer Berlin Heidelberg, (Euro-Par), 2013.

— *Block Sampling: Efficient Accurate Online Aggregation in MapReduce.* Kalavri, Vasiliki; Brundza, Vaidas; Vlassov, Vladimir; 5th IEEE International Conference on Cloud Computing Technology and Science (CloudCom) 2013.

— *m2r2: A Framework for Results Materialization and Reuse in High-Level Dataflow Systems for Big Data.* Kalavri, Vasiliki; Shang, Hui; Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on. IEEE, 2013.

1.4 Related Work

We summarize related work in the areas of distributed graph processing, web tracker detection, optimizations for fixpoint graph algorithms, and semi-metricity, graph compression, and sparsification techniques. We compare with more specific related work in the corresponding chapters throughout this thesis.

Graph Processing Surveys

A recent survey of vertex-centric frameworks for graph processing [127] presents a extensive study of frameworks that implement the vertex-centric programming model and compares them in terms of system design characteristics, such as scheduling, partitioning, fault-tolerance, and scalability. Moreover, it briefly introduces subgraph-centric frameworks, as an optimization to vertex-centric implementations. While there is some overlap between this work and ours, the objective of our study in Chapter 3 is to present the first comprehensive comparison of distributed graph processing abstractions, regardless of the specifics of their implementations. While in [127] the discussion revolves around certain frameworks, in our work, we first consider the programming models decoupled from the systems, then build a taxonomy of systems based on their implemented model.

A notable study of parallel graph processing systems is presented in [63]. The work surveys over 80 systems, spanning single-machine, shared-memory, and distributed architectures. The scope of their study is much wider than ours, yet their results are driven by system implementations and platform design choices. With regard to programming models, they consider general-purpose processing systems, vertex-centric, and subgraph-centric, but do not further expand on execution semantics, user-facing interfaces or limitations.

While the focus of our work is to describe and compare available programming abstractions for distributed graph processing, several recent studies focus on the performance of modern distributed graph processing platforms. In [85], the authors compare the performance of six systems, including general-purpose data processing frameworks, specialized graph processors and a non-distributed graph database. The study evaluates these systems across a number of performance metrics, such as execution time, CPU and memory utilization, and scalability, using real-world datasets and a diverse set of graph algorithms. Similarly, [123] presents an experimental evaluation of distributed graph processing platforms, but covering only specialized graph processors. The also study the effectiveness of implemented optimization techniques and compare performance with a single machine system as baseline. A similar but more extensive study can be found in [28]. Both these studies conclude that no single system performs best in all cases and also highlight the need for a standard benchmark solution, that could simplify the performance comparison of graph processing platforms. Even though a standard benchmarking solution for graph processing systems does not exist yet, [86, 45] are two significant steps towards that direction. They propose a clear vision for a benchmark, listing challenges that need to be addressed and share early results in designing a graph processing benchmark, called Graphalytics. Graphalytics supports graph generation with custom degree distributions and structural characteristics and already supports several systems and algorithms.

Web Tracking

TrackAdvisor [120] is a tool that analyzes cookie exchange statistics from HTTP requests to automatically detect trackers. Similar to us, their goal is to identify trackers without having to rely on blacklists. They also identify third-party requests by looking at the referer field. Their dataset is created by visiting Alexa top 10K pages (not real user data) and is an order of magnitude smaller than ours (500k requests in total). Our method does not need to intercept cookie traffic. Our finding that a tracker appears in multiple pages agrees with their results. In conclusion, we could call the two methods complementary; they could be combined to produce a more powerful tool.

Roesner et al. provide a study of web tracking, classifying tracking behaviors and evaluating defense mechanisms [158] using web traces from AOL search logs to simulate real user behavior. Although their work focuses on classifying tracking mechanisms that use cookies and proposes a Firefox add-on, which transparently removes cookies from third-party requests, as a new defense mechanism. They build a classification framework for distinguishing different types of trackers, based on their functionality. In contrast, our classification method distinguishes between trackers and non-trackers, while it is oblivious to the tracker mechanisms and functionality specifics. As in our study, the authors also find that popular trackers can capture more 20% of a user's browsing behavior.

Bau et al. propose a machine learning mechanism for detecting trackers [30]. They evaluate machine learning approaches and present results from a prototype implementation. They use DOM-like hierarchies from the crawl data HTTP content headers as the main features. While they achieve precision of 54% for 1% FPR, our methods achieve much better precision and lower FRP, while not relying on page content collection.

Gomer et al. investigate the graph structure of third-party tracking domains in [80] in the context of search engine results. They obtain their graph by using several popular search engines with a set of pre-defined queries as opposed to real browsing behavior as we do. Their focus is on how users are exposed to trackers via normal search behavior and they find a 99.5% chance of being tracked by all major trackers within 30 clicks on search results. They further found that the graph structure was similar across geographic regions, which reduces the concern of bias in our dataset. Their results agree with our findings regarding the community structures of trackers. They find a dominant connected component that includes the vast majority of tracking nodes and connections. They show that tracking networks expose small-world characteristics, where a small group of entities are highly connected among them.

In [73], Falahrastegar et al. examine the geographical properties of third-party trackers. They show that there are several trackers that focus on specific regions or countries, while there also exists a small number of international corporations with dominant presence across regions.

In [90], Ihm et al. analyze web traffic over a 5-year period. They use a distributed proxy and analyze traffic from 70k users and 187 countries. The work focuses on general web traffic characteristics, such as bandwidth, browser popularity, page complexity and data redundancy. The authors do not make a separate analysis on trackers or tracking behavior.

Another large-scale analysis of third-party trackers on the internet is presented in [163]. The data is extracted from more than 3.5 billion web pages of the CommonCrawl 2012 corpus, creating a dataset with more than 41 million domains. The goals of this study is to reveal the extent of tracking and analyze how the tracking ecosystem differs in different countries.

In agreement with our findings, most of the above works also find that a small number of trackers are able to capture the majority of user behavior. However, our work is, to the best of our knowledge, the first to show that using this community structure as an explicit feature can accurately predict whether an unknown URL is a tracker or not.

Fixpoint Graph Processing Optimizations

GraphLab's [122] adaptive execution and its use of a pull-based model make an interesting comparison with our view of a push-based model where vertices generate updates for their neighbors, described in Chapter 5. GraphLab update functions can access the state of adjacent vertices, even if those are not scheduled for execution in the current superstep. The dependency plan can be implemented in this pull-based model by having a vertex check all of its neighbors' states and decide to participate in the computation if at least one of them has changed value. Note that the trade-offs might be different in this case and the cost model will have to be adjusted accordingly.

GraphX [82] and Pregelix [42] are two graph processing systems that also adopt the idea of performing graph analytics using dataflow operators. Naiad [136] is distributed dataflow system able to perform iterative and incremental computations. It efficiently supports stateful batch and streaming graph computations. High-level graph programming abstractions like hte pregel model can be easily mapped to Naiad primitives. In [142], the authors decompose iterative MapReduce queries into invariant and variant views and incrementally compute the variant view to avoid redundant computations. The main idea is conceptually the same as our the incremental and delta iteration techniques, but incrementalization is abandoned if the aggregation computation is not distributive. In our case, such an application can still benefit by the dependency iteration plan.

Delta and incremental optimizations have been used in several other systems as well. REX [133] is a system for recursive, delta-based data-centric computation, which uses user-defined annotations to support incremental updates. It allows explicit creation of custom delta operations and lets nodes maintain unchanged state and only compute and propagate deltas. The system optimizer discovers incrementalization possibilities during plan optimization, while the user can also manually add delta functions to wrap operators, candidates for incrementalization.

Distributed SociaLite [164] and BigDatalog [166] explore the possibility of using datalog for large-scale graph analysis. Socialite enhances datalog with a set of extensions, efficient data structures, and recursive aggregate functions, which can be efficiently evaluated using semi-naive evaluation. To parallelize the recursive aggregate functions, these have to be monotone, idempotent, commutative, and associative. Under these circumstances, delta stepping [132] is be used to parallelize monotone recursive aggregate functions. BigDatalog uses parallel semi-naive evaluation to efficiently execute recursive queries on Apache Spark. The authors present a parallel evaluation technique that allows using Spark as a Datalog runtime and they propose novel data structures and recursive operators. They also propose several optimizations, most of which are also automatically implemented by the Apache Flink optimizer and its native iteration operators. Specifically, mutable RDD types would correspond to the use of a mutable distributed hash table to represent the iteration solution set in Apache Flink. Input caching and the use of broadcast joins are also handled transparently by Flink's optimizer. Finally, single-job scheduling and caching optimizations are not required in our case, since Flink's iteration operators are native in the execution DAG and no job re-submission is required after an iteration step.

Semi-metricity, Graph Sparsification and Compression

Complex network analysis

The concept of semi-metricity in weighted graphs has been first used in complex network analysis. Semi-metricity was introduced in weighted graphs by Rocha [154, 155], showing that semi-metric edges in a weighted graph encode some latent information between a pair of nodes, which may be useful for information discovery [154, 155, 168, 153, 169]. Simas et al. introduce in [169] a new mathematical framework to the study of networks in general and specifically semi-metric networks. In [169] Simas et al. introduce the concept of distance backbones, a generalization of the metric backbone. In this work they present a few examples of how distance backbones, including the metric backbone, can be useful in network analysis, such as improving modularity in community detection.

Graph compression and query optimization

Graph compression and summarization techniques are closely related to this work. Research in this area aims to provide reduced graph representations, which may have lower storage requirements for large graphs and lower graph query costs. In [134, 19], the authors examine the problem of producing minimal graph representations, while preserving the graph's reachability properties. Their goal is to reduce the memory used for storing the graph and potentially improve the efficiency of certain algorithms that contain reachability queries. More recent works on graph compression, [37], [52], look into specific techniques for compressing web graphs and social networks. These provide methods that preserve the information of the original graph. In these cases, graph decompression is necessary for query evaluation. On the other hand, [74], proposes a query-preserving graph compression method, relative to reachability and graph pattern queries. Similar to our work, the query can be directly issued on the compressed graph representation. The authors also provide a method for incrementally maintaining the compressed graph structure, for dynamic graphs. The proposed algorithms summarize nodes, based on equivalence relations for each query class. Compression algorithms are also studied in [138, 121]. These works explore ways to minimize graph representation overhead and offer several algorithms, both sequential and distributed, for lossless and lossy graph compression. However, their main goal is reducing the graph sizes, while providing guarantees on decompression accuracy.

Graph compression is also closely related to graph query optimization. [194] provides a graph indexing technique that speeds up search in large graphs. It leverages shortest-paths information, but does not generate a reduced graph representation. The indexes encode neighborhood shortest-paths information, which is used to prune the graph search space. The technique is mostly targeted to graph isomorphism queries.

If we view the metric backbone as a reduced graph representation, there are two fundamental differences between our work and aforementioned works. First, our method does not collapse graph vertices; the metric backbone preserves all the vertices of the original graph. Second, none of these works take into account the weights when compressing the graph. Regarding the motivation of our work, an important difference is that we are mainly concerned with the challenge of scalability and we provide a distributed implementation of the metric backbone algorithm. Moreover, existing works focus on optimizing reachability and graph matching queries. In contrast, we go a step further and evaluate the metric backbone in the context of graph analytics and iterative graph processing.

Spanners and Sparsifiers

Spanners [145, 65] and sparsifiers [77] are approximate graph structures that are used to approximate graph distances and shortest paths. They have been mostly used in streaming algorithms as sketches or graph summaries. In this context, the metric backbone can be considered equivalent to the minimum 1-Spanner. A spanner guarantees that distances can be approximated with a certain maximum multiplicative error. On the contrary, the metric backbone preserves all shortest paths and gives the *exact* shortest distances. Thus, it can be used in applications that cannot tolerate errors on the graph distances.

1.5 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we provide a literature study on distributed data processing with MapReduce. We summarize the limitations of the MapReduce programming model and we survey optimizations and open issues. In Chapter 3, we provide an overview of high-level programming abstractions for distributed graph processing. We explore their semantics and their limitations, and we categorize modern distributed graph processing systems, with respect to the abstraction they implement. Chapter 4 presents a use-case scenario of a distributed graph processing application and describes an end-to-end data processing pipeline that implements it. In Chapters 5 and 6 we present two optimization techniques for distributed graph processing on (1) a general-purpose distributed data processing platform and (2) a specialized distributed graph processor, respectively. We summarize the results of this work and conclude in Chapter 7.

Some passages and figures in this thesis have been quoted verbatim from the following sources [104, 99, 103, 101].

Chapter 2

Distributed Data Processing with MapReduce

A *data-intensive application* is an application that, as opposed to a *computation-intensive* application, requires processing of an amount of data that is large in comparison to the amount of computation that needs to be performed. Data-intensive applications are often implemented on top of distributed frameworks and exploit data parallelism in order to achieve good performance. In distributed environments, where network communication is expensive, minimizing data transfers between computation nodes is of essence. One of the prevalent design ideas towards this direction suggests *moving the computation close to the data*. Thus, both the data storage framework and the data processing engine would be deployed on the same set of nodes. This is one of the main ideas introduced by MapReduce [59]. The invention of MapReduce radically transformed the area of distributed data processing and heavily inspired the development of modern data processing frameworks. Next-generation frameworks, including the ones we use for this work, borrow the ideas of high-level, declarative programming abstractions and automatic data distribution and parallelism from MapReduce.

Out of the numerous implementations of the MapReduce programming model that have been developed, the open-source Apache Hadoop framework [7] is the most widely adopted. Apart from the MapReduce programming model, Hadoop offers various other capabilities, including a distributed file system, HDFS [167] and a scheduling and resource management layer. Despite its popularity, the MapReduce model and its Hadoop implementation have also been criticized [10] and have been compared to modern parallel database management systems (DBMSs), in terms of performance and complexity [144]. Several studies [117, 97, 83] have previously identified shortcomings of the model and its current implementations. Researchers and system designers have criticized features such as the static map-shuffle-reduce pipeline, the frequent data materialization (writing data to disk for fault-tolerance), the lack of support for iterations and state transfer between jobs, the lack of indexes and schema, and the sensitivity to configuration parameters. All of the above have been shown to negatively affect the performance of certain classes of MapReduce applications.

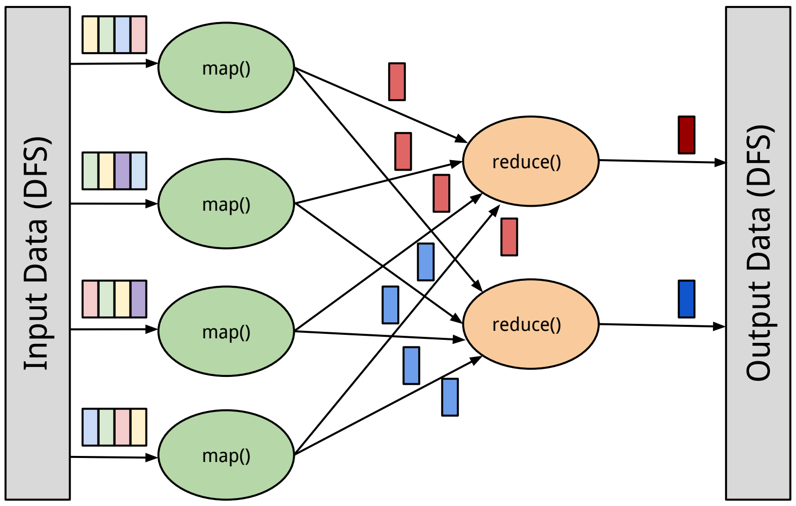


Figure 2.1 – Stages of a MapReduce job.

During the last few years, there has been significant work towards improving MapReduce and Hadoop, proposing performance improvements, programming model extensions, automation of use, and tuning. In this chapter, we describe the motivation of MapReduce and we give an overview of the programming model and the Hadoop implementation. We discuss its limitations, proposed optimizations, and we identify open issues in the area of MapReduce and distributed data processing.

2.1 The MapReduce Programming Model

The MapReduce programming model is designed to efficiently execute programs on large clusters, by exploiting data parallelism. A distributed file system is deployed on the same machines where the applications run, so that execution can benefit from data locality, by trying to schedule computations where the data reside. Figure 2.1 shows the stages of a MapReduce job. The model is inspired by functional programming and consists of two second-order functions, *Map* and *Reduce*, which form a static pipeline, where the Map stage comes first and is followed by the Reduce stage.

Data is read from the distributed file system, in the form of user-defined key-value pairs. These pairs are then grouped into subsets and serve as input for parallel instances of the Map function. A user-defined function must be specified and is applied to all key-value pairs independently. The Map function outputs a new set of key-value pairs, which is then sorted by key and partitioned according to a partitioning function. The sorted data feed the next stage of the pipeline, the Reduce function. The partitioning stage of the framework guarantees that all pairs sharing the same key will be processed by the same Reduce task. A user-defined function is applied to each group of pairs sharing the same key, producing one output file per Reduce task. The results are stored in the distributed file system.

One of the important advantages of this model is that the parallelization is automatically

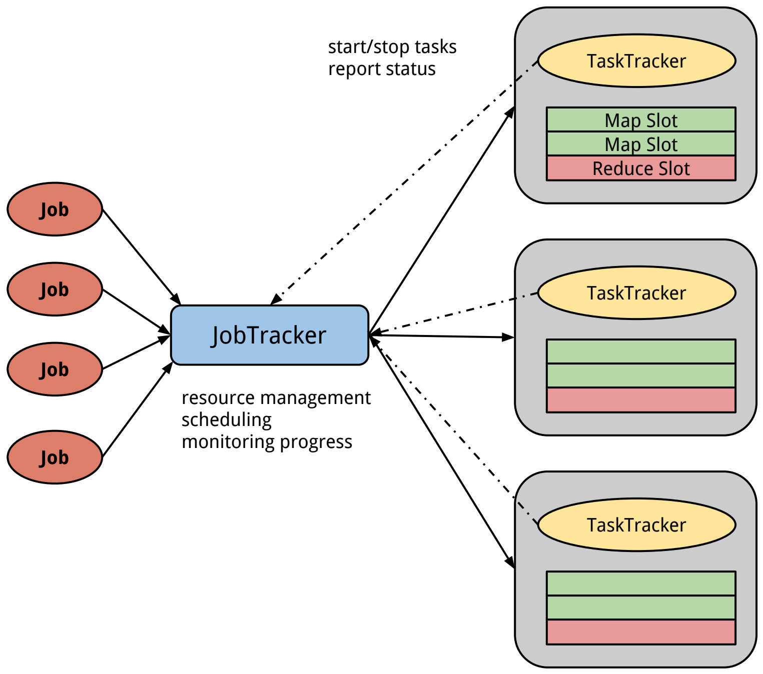


Figure 2.2 – The Master-Slave architecture of early MapReduce frameworks

handled by the framework. The user only has to specify the computation logic that will be wrapped inside the Map and Reduce functions. However, this advantage comes with a certain loss of flexibility. Every job must consist of exactly one Map function followed by an optional Reduce function, and steps cannot be executed in a different order. Moreover, if an algorithm requires multiple Map and Reduce steps, these can only be implemented as separate jobs. Data between different jobs can only be transferred through the file system.

In the initial implementations of Hadoop, MapReduce is designed as a master-slave architecture, as shown in Figure 2.2. The *JobTracker* is the master, managing the cluster resources, scheduling jobs, monitoring progress, and dealing with fault-tolerance. On each of the slave nodes, there exists a *TaskTracker* process, responsible for launching parallel tasks and reporting their status to the JobTracker. The slave nodes are *statically* divided into computing slots, available to execute either Map or Reduce tasks.

The Hadoop community realized the limitations of this static model and recently re-designed the architecture to improve cluster utilization and scalability. They introduced YARN [178], *Yet Another Resource Negotiator*, which allows Hadoop to serve as a general data-processing framework. It supports programming models other than MapReduce, while also improving scalability and resource utilization. YARN makes no changes to the programming model or to HDFS. It consists of a re-designed runtime system, aiming to eliminate the bottlenecks of the master-slave architecture. The responsibilities of the JobTracker are split into two different processes, the *ResourceManager* and the *ApplicationMaster*. The ResourceManager handles resources dynamically, using the notion of *containers*, instead of static Map/Reduce slots. Containers are configured based on information about

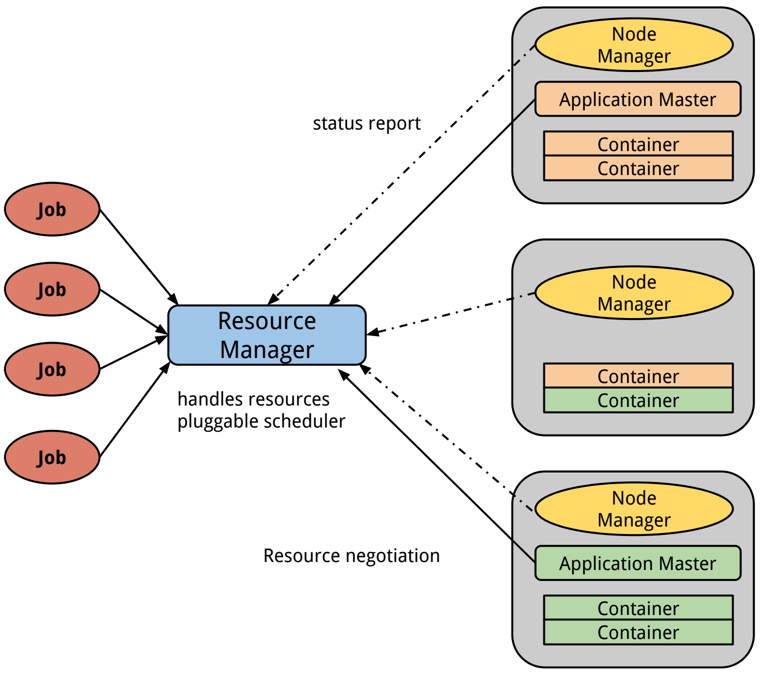


Figure 2.3 – The YARN architecture

available memory, CPU and disk capacity. YARN also has a pluggable scheduler, which can use different strategies to assign tasks to available nodes. The ApplicationMaster is a framework-specific process, meaning that it allows other programming models to be executed on top of YARN. It negotiates resources with the ResourceManager and supervises the scheduled tasks. The YARN architecture is shown in Figure 2.3.

2.2 MapReduce Limitations

Even though YARN manages to overcome the well-known limitations of the Hadoop scheduling framework and improves scalability and resource utilization, there still exist several opportunities for optimizations in Hadoop and MapReduce. We group the optimization opportunities, found in recent literature, in three main categories: performance issues, programming model extensions, and usability enhancements. Next, we discuss the limitations which lead to these optimization opportunities.

2.2.1 Performance Issues

Even though Hadoop/MapReduce has been praised for its scalability, fault-tolerance, and capability of processing vast amounts of data, query execution time can often be in the order of several hours [107]. This is orders of magnitude higher than what modern DBMSs offer and prevents interactive analysis. Performance highly depends on the nature of the application, but is also influenced by inherent system characteristics and design choices. A quite large percentage of the execution time is spent in task initialization, scheduling, coordination, and monitoring. Moreover, Hadoop/MapReduce does not support data pipelining and does not allow overlapping between the execution of the Map phase and the execution of the Reduce phase. Data materialization for fault-tolerance and intensive disk I/O during the shuffling phase have also been found to significantly contribute to the overall execution time. It has been suggested that Hadoop performance would benefit from well-known optimization techniques, already used by database systems and query optimizers. Even though Hadoop lacks a built-in optimizer, similar to what one can find on database systems, many techniques have been borrowed from database research and implemented in Hadoop extensions. Optimizations include index creation [61], data co-location [70], reuse of previously computed results [68, 102], exploiting sharing opportunities [140], mechanisms dealing with computational skew [112] and techniques allowing early approximate query results [57, 100].

2.2.2 Programming Model Issues

Developing efficient MapReduce applications requires advanced programming skills and deep understanding of the system architecture. Common data analysis tasks usually include processing of multiple datasets and performing relational operations, such as joins, which are not trivial to implement in MapReduce. Thus, the MapReduce programming model has often been characterized as being too *low-level* for analysts used to SQL-like or declarative languages. Another limitation of the programming model stems from its *batch* nature. Data must be uploaded to the file system and even when the same dataset needs to be analyzed multiple times, it must be read from scratch every time. Furthermore, the computation steps are fixed and applications must respect the map-shuffle-sort-reduce sequence. Complex analysis queries can only be realized by chaining multiple MapReduce jobs, having the results of one serving as the input for the next. These limitations make the model inappropriate for certain classes of algorithms. Various applications, including machine learning and graph analysis algorithms, often require iterations or incremental computations. Since MapReduce operators are stateless, MapReduce implementations of iterative algorithms require manual management of the state and manual chaining of the iterative steps. In order to facilitate complex application development with MapReduce, several abstractions and high-level languages have been built [78, 175]. Also, a set of domain-specific systems have emerged, extending the MapReduce programming model. We review these systems in Section 2.3.2.

2.2.3 Configuration and Automation Issues

The third category of optimizations are related to automatic tuning and ease of use. There are numerous configuration parameters to set when deploying a Hadoop MapReduce cluster. Performance is often quite sensitive to them and users usually rely on empirical "rules of thumb". Options include the number of parallel tasks, the size of the file blocks, and the replication factor. Proper tuning of these parameters requires knowledge of both the available hardware characteristics and the workload characteristics. Failure to properly

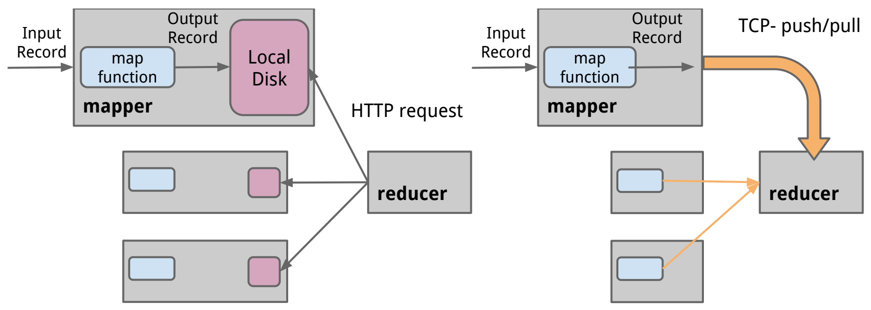


Figure 2.4 – Data pipelining between operators in MapReduce Online.

configure important parameters might lead to inefficient execution and under-utilization of resources [107, 88]. Hadoop variations dealing with automatic tuning are discussed in Section 2.3.3.

2.3 MapReduce Variations

Here, we now look into how specific MapReduce systems implement the presented optimizations in order to address the limitations of the vanilla MapReduce implementation. Some of the optimizations discussed in this section could fall into more than one of the categories we have presented. We base the categorization and comparison that follows on the primary motivation of each examined system.

2.3.1 Performance Optimizations

Operator Pipelining and Online Aggregation

One of the early Hadoop extensions is MapReduce Online [57]. It boosts performance by supporting online aggregation and stream processing, while also improving resource utilization. The motivation of MapReduce Online is to enable *pipelining* between operators, while preserving fault-tolerance guarantees. Pipelining is implemented both between tasks and between jobs. The system uses a ”mixed” push/pull approach for data transfers. Figure 2.4 shows the data pipelining operation in MapReduce Online. The left drawing depicts the original MapReduce execution. In this case, after the mapper tasks have finished execution, they write their intermediate results in local disk, while the reducer tasks remain idle until then. Once all mappers have finished, the reduce tasks get notified and retrieve input data from the local disks of the map tasks, by issuing an HTTP request. On the right, we see how MapReduce Online operates. Reduce tasks do not have to remain idle until the map tasks have finished. Instead, they receive intermediate results as they are being produced.

The pipelining technique is hard to use together with *combiners*. Combiners are tasks that are executed on the mapper side before sending intermediate results to reducers. Their goal is to combine results when possible, so that less data is shipped over the network. In order to use pipelining in conjunction with combiners, MapReduce Online buffers intermediate data up to a specified threshold, applies the combiner function on them, and then spills them to disk.

As a side-effect of the MapReduce Online design, early results can be computed, making approximate answers to queries available to users. This technique is called *online aggregation* and returns useful early results, before job completion. By applying the reduce function to the data that the reducer has seen so far, the system can provide an early snapshot of the current result. Users can approximate the accuracy of the provided snapshot by combining it with job progress metrics.

Approximate Results

A more sophisticated approach to retrieving approximate results in MapReduce is proposed by Laptev et al. [115]. The EARL library is a Hadoop extension which allows incremental computations of early results using sampling and the bootstrapping technique. The system obtains an initial sample of the data and estimates the error using bootstrapping. If the error is too high, the sample is expanded and the error is recomputed. This process is repeated until the error is under a user-defined threshold. In order to implement EARL, Hadoop was extended to support dynamic input size expansion. First, the authors implemented pipelining between mappers and reducers, similarly to MapReduce Online, so that reduce tasks can start processing data as soon as they become available. EARL modifies the mapper task management to keep mapper tasks active and reuse them instead of restarting them in every sampling iteration. This modification saves a significant amount of setup time. Finally, a communication channel was built between mappers and reducers, so that the termination condition can be easily tested. EARL is an addition to the MapReduce API and existing applications require modifications in order to exploit the library.

Indexing and Sorting

Quite a few of the proposed optimizations for Hadoop/MapReduce are inspired by well-known database techniques. Long query runtimes are often caused due to lack of proper schemas and data indexing. Hadoop++ [61] and HAIL [62] are two remarkable works dealing with this issue in the context of MapReduce. Hadoop++ is a transparent addition to Hadoop implemented using User Defined Functions (UDFs). It provides an indexing technique, the *Trojan Index*, which extends input splits with indexes at load time. Additionally to the Trojan Index, the paper also proposes a novel join technique, the *Trojan Join*, which uses data co-partitioning in order to perform the join operation using only map tasks. HAIL proposes inexpensive index creation on Hadoop data attributes, in order to reduce execution times in exploratory use-cases of MapReduce. It modifies the upload pipeline of HDFS and creates a different clustered index per block replica. HAIL uses the efficient binary PAX representation [20] to store blocks and keeps each physical block replica in a different sort order. Sorting and indexing happen in-memory at upload time. If index information is available, HAIL also uses a modified version of the task scheduling algorithm of Hadoop, in order to schedule tasks to nodes with appropriate indexes and sort orders. The block binary representation and in-memory creation of indexes improve upload times for HDFS. Query execution times significantly improve as well when index information is available. HAIL preserves Hadoop's fault-tolerance properties. However, failover times are sometimes higher, due to HAIL assigning more blocks per map task, therefore limiting parallelization during recovery. In a system with the default degree of replication, three different sort orders and indexes are available, greatly increasing the probability of finding a suitable index for the corresponding filtering attribute of the query. HAIL benefits queries with low selectivity, exploratory analysis of data, and applications for which there exists adequate information for index creation.

Work Sharing

MRShare [140] is a Hadoop extension that aims to exploit sharing opportunities among different jobs. It transforms a batch of queries into a new batch, by forming an optimization problem and providing the optimal grouping of queries to maximize sharing opportunities. MRShare works on the following levels. Sharing scans when the input to mapping pipelines is the same and sharing map outputs and map functions. MRShare modifies Hadoop to support tagging of tuples and merges the tags into the keys of tuples, so that their origin jobs can be identified. Moreover, it modifies reducers to enable them to write to more than one output files.

Data Reuse

ReStore [68] is an extension to Pig [78], a high-level system built on top of Hadoop/MapReduce. It stores and reuses intermediate results of scripts, originating from complete jobs or sub-jobs. The input of ReStore is Pig's physical plan; a workflow of MapReduce jobs. ReStore maintains a repository where it stores job outputs together with the physical execution plan, the file name of the output in HDFS, and runtime statistics about the MapReduce job that produced the output. The system consists of a plan matcher and rewriter which searches in the repository for possible matches and rewrites the job workflow to exploit stored data. It also has a sub-job enumerator and a sub-job selector, which are responsible for choosing which sub-job outputs to store, after a job workflow has been executed. Sub-job results are chosen to be stored in the repository based on the input to output ratio and the complexity of their operators. Repository garbage collection is not implemented, however guidelines for building one are proposed.

Skew Mitigation

SkewTune [112] is a transparent Hadoop extension providing mechanisms to detect stragglers and mitigate skew by re-partitioning their remaining unprocessed input data. In order to decide when a task should be treated as a straggler, while avoiding unnecessary overhead and false-positives, SkewTune is using a technique called *Late Skew Detection*. Depending on the size of the remaining data, SkewTune may decide to scan the data locally or in parallel. In Hadoop, skew mitigation is implemented by SkewTune as a separate MapReduce job for each parallel data scan and for each mitigation. When re-partitioning a map task, a map-only job is executed and the job tracker broadcasts all information about the mitigated map to all the reducers in the system. When re-partitioning a reduce task, due to the MapReduce static pipeline inflexibility, an identity map phase needs to be executed before the actual additional reduce task.

Data Colocation

CoHadoop [70] allows applications to control where data is stored. In order to exploit its capabilities, applications need to state which files are related and are likely to be processed together. CoHadoop uses this information to collocate files and improve job runtimes. While HDFS uses a random placement policy for load-balancing reasons, CoHadoop allows applications to set a new file property, in order for all copies of related files to be stored together. This property, the *locator*, is an integer and there is a N:1 relationship between files and locators, so that files with the same locator are stored on the same set of storage nodes. If the selected set of storage nodes runs out of space, CoHadoop simply stores the files in another set of nodes. CoHadoop may lead to skew in data distribution or loss of data in the presence of failures. It performs collocation and special partitioning by adding a pre-processing step to a MapReduce job, which is itself a MapReduce job.

2.3.2 Programming model extensions

High-Level Languages

Developing applications using high-level languages on top of Hadoop has proven to be much more efficient in terms of development time than using native MapReduce. Maintenance costs and bugs could be reduced, as less code is required. Pig [78] is a high-level system that consists of a declarative scripting language, *Pig Latin*, and an execution engine that allows the parallel execution of data-flows on top of Hadoop. Pig offers an abstraction that hides the complexity of the MapReduce programming model and allows users to write SQL-like scripts, providing all common data operations (filtering, join, ordering, etc.).

Another widely-used high-level system for Hadoop is Hive [175]. Initially developed by Facebook, Hive is not just an abstraction, but a data warehousing solution. It provides a way to store, summarize, and query large amounts of data. Hive's high-level language, HiveQL, allows users to express queries in a declarative, SQL-like manner. Very similar to Pig, HiveQL scripts are compiled to MapReduce jobs and executed on the Hadoop execution engine.

Another query language for Hadoop is Jaql [34]. Jaql is less general than Pig and Hive, as it is designed for querying semi-structured data in JSON format only. The system is extensible and supports parallelism using Hadoop. Although Jaql has been specifically designed for data in JSON format, it borrows a lot of characteristics from SQL, XQuery, LISP, and PigLatin.

Cascading [5] is a Java application framework that facilitates the development of data processing applications on Hadoop. It offers a Java API for defining and testing complex dataflows. It abstracts the concepts of map and reduce and introduces the concept of *flows*, where a flow consists of a data source, reusable pipes that perform operations on the data, and data sinks.

Domain-specific Systems

Support for Iterations. Iterative algorithms are very common in data-intensive problems, especially in the domains of machine learning and graph processing. HaLoop [43] is a modified version of Hadoop with built-in support for development and efficient execution of iterative applications. HaLoop offers a mechanism to cache and index *invariant* data between iterations, significantly reducing communication costs. It extends Hadoop's API, allowing the user to define loops and termination conditions easily. The authors also propose a novel scheduling algorithm, which is loop-aware and exploits inter-iteration locality. It exploits cached data in order to co-locate tasks which access the same records in different iterations.

Support for Incremental Computations Incremental computations are useful for jobs which need to be run repeatedly with slightly different, most often augmented input. Performing such computations in MapReduce would obviously lead to redundant computations and inefficiencies. A possible solution is to specially design MapReduce applications to store and use state across multiple runs. Since MapReduce was not designed to reuse intermediate results, writing such programs is complex and error-prone. Incoop's [35] goal is to provide a transparent way to reuse results of prior computations, without demanding any extra effort from the programmer. Incoop extends Hadoop to support incremental computations, by introducing three novel features: (a) Inc-HDFS. A modified HDFS which splits data depending on file contents instead of size. It provides mechanisms to identify similarities between datasets and opportunities for data reuse, while preserving compatibility with HDFS. (b) Contraction Phase. An additional computation phase added before the reduce phase, used to control task granularity. This phase leverages the idea of combiners to *break* the reduce task into a tree-hierarchy of smaller tasks. The process is run recursively until the last level, where the reduce function is applied. In order to result into a data partitioning suitable for reuse, content-based partitioning is again performed on every level of combiners. (c) Memoization-aware Scheduler. An improved scheduler which takes into account data locality of previously computed results, while also using a work-stealing algorithm. The memoization-aware scheduler schedules tasks on the nodes that contain data which can be reused. However, this approach might create load imbalance if some data items are very popular. To avoid this situation, the scheduler implements a simple work-stealing algorithm. When a node runs out of work, the scheduler will locate the node with the largest task queue and delegate a task from the busy node to the idle node.

2.3.3 Automatic tuning

Self-Tuning

Configuring and tuning Hadoop MapReduce is usually not a trivial task for developers and administrators. Misconfiguration is a common cause of poor performance, resource under-utilization, and consequently increased operational costs. Starfish [88] is a self-tuning system, built as an extension to Hadoop, which dynamically configures system properties based on workload characteristics and user input. Starfish performs tuning on three levels. In the *job-level*, it uses a *Just-in-Time Optimizer* to choose efficient execution techniques, a *Profiler* to learn performance models, and build job profiles and a *Sampler* to collect statistics about input, intermediate and output data, and help the Profiler build approximate models. In the *workflow-level*, it uses a *Workflow-aware Scheduler*, which exploits data locality on the workflow-level, instead of making locally optimal decisions. A *What-if Engine* answers questions based on simulations of job executions. In the *workload-level*, Starfish consults the *Workload Optimizer* to find opportunities for data-flow sharing, materialization of intermediate results for reuse or reorganization of jobs inside a batch, and the *Elastisizer* to automate node and network configuration.

Disk I/O Minimization

Sailfish [152] is another Hadoop modification also providing auto-tuning opportunities, such as dynamically setting the number of reducers and handling skew of intermediate data. Additionally, it improves performance by reducing disk I/O due to intermediate data transfers. The proposed solution uses KFS [9] instead of HDFS, which is a distributed file system allowing concurrent modifications to multiple blocks of a single file. The authors propose *I-files*, an abstraction which aggregates intermediate data, so that they can be written to disk in batches. An index is built and stored with every file chunk and an offline daemon is responsible for sorting records within a chunk.

Data-aware Optimizations

Manimal [95] is an automatic optimization framework for MapReduce, transparent to the programmer. The idea is to apply well-known query optimization techniques to MapReduce jobs. Manimal detects optimization opportunities by performing static analysis of compiled code and only applies optimizations which are *safe*. The system's *analyzer* examines the user code and sends the resulting optimization descriptors to the *optimizer*. The optimizer uses this information and pre-computed indexes to choose an optimized execution plan, the *execution descriptor*. The *execution fabric* then executes the new plan in the standard map-shuffle-reduce fashion. Example optimizations performed by Manimal include *Selection* and *Projection*. In the first case, when the map function is a filter, Manimal uses a B+Tree to only scan the relevant portion of the input. In the second case, it eliminates unnecessary fields from the input records.

2.4 Discussion

Table 2.1 shows a brief comparison of the systems discussed in this chapter. We have excluded high-level languages from this comparison, since they share common goals and major characteristics among them.

MapReduce, Hadoop, and similar frameworks still have plenty of optimization opportunities to exploit. However, some techniques can be challenging to apply in architectures of shared-nothing clusters of commodity machines. Scalability, efficiency and fault-tolerance

	Optimization Type	Major Contributions	Open-Source / Available to use	Transparent to Existing Applications
MapReduce Online	performance, programming model	Pipelining, Online aggregation	yes	yes
EARL	performance	Fast approximate query results	yes	no
Hadoop++	performance	Performance gains for relational operations	no	yes
HAIL	performance	Performance gains for relational operations	no	no
MRShare	performance	Concurrent work sharing	no	no
ReStore		Reuse of previously computed results	no	yes
SkewTune	performance	Automatic skew mitigation	no	yes
CoHadoop	performance	Communication minimization by data colocations	no	no
HaLoop	programming model	Iteration support	yes	no
Incoop	programming model	Incremental processing support	no	no
Starfish	tuning, performance	Dynamic self-tuning	no	yes
Sailfish	tuning, performance	Disk I/O minimization and automatic tuning	no	yes
Manimal	tuning, performance	Automatic data-aware optimizations	no	yes

Table 2.1 – Comparative Table of Hadoop Variations

are major requirements for any distributed data processing framework and trade-offs between optimizations and these features have to be carefully considered.

We can identify several trends by reviewing the systems we present in this chapter. In order to achieve good performance for data-intensive, MapReduce applications, it is vital to minimize disk I/O and communication. Therefore, many systems have sought ways to enable in-memory processing and avoid reading from disk when possible. For the same reason, traditional database techniques, such as materialization of intermediate results, caching and indexing have been favored.

Another recurring theme in MapReduce systems is relaxation of fault-tolerance guarantees. The initial MapReduce design from Google assumed deployments in clusters of hundreds or even thousands of commodity machines. In such setups, failures are very common and strict fault-tolerance and recovery mechanisms are necessary. However, after the release of Hadoop, MapReduce has also been used by organizations much smaller than Google. Common deployments may consist of only tenths of machines [13] and exhibit significantly decreased failure rates. Such deployments can benefit from higher performance, by relaxing the strict fault-tolerance guarantees of MapReduce. For example, we can avoid materialization of task results and allow pipelining of data. In this scenario, when a failure occurs, the whole job would have to be re-executed, instead of only rescheduling the tasks running on the failed node.

Despite the significant progress that has been made since the launch of MapReduce and Hadoop, several open issues still exist. Even though it is clear that relaxing fault-tolerance offers performance gains, we believe that this issue needs to be further studied in the context of MapReduce. The trade-offs between fault-tolerance and performance need to be quantified. When these trade-offs have become clear, Hadoop could offer capabilities of tunable fault-tolerance to the users or provide automatic fault-tolerance adjustment mechanisms, depending on cluster and application characteristics.

Another open issue is the lack of a standard benchmark or a set of typical workloads for comparing different Hadoop implementations. Each system is evaluated using different datasets, deployments and set of applications. There have been some efforts in this direction [152, 51], but no clear answer exists to what is *typical* MapReduce workload.

Regarding programming extensions, we believe that the main requirement is to provide transparency to the developer. In our view, programming extensions need to be smoothly integrated into to the framework, so that existing applications can benefit from the optimizations, automatically, without having to significantly change the source code.

Finally, even if successful declarative-style abstractions exist, Hadoop MapReduce is still far from offering interactive analysis capabilities. Developing common analysis tasks and declarative queries has indeed been significantly facilitated. However, these high-level systems still compile their queries into MapReduce jobs, which are executed on top of Hadoop. These systems could greatly benefit from more sophisticated query optimization techniques. Mechanisms such as data reuse and approximate answers could also be more extensively studied and exploited in high-level systems.

Chapter 3

Programming Abstractions and Platforms for Distributed Graph Processing

Efficient processing of large-scale graphs in distributed environments has been an increasingly popular topic of research and systems development in recent years. Inter-connected data that can be modeled as graphs arise in application domains, such as machine learning, recommendation, web search, and social network analysis. Writing distributed graph applications is inherently hard and requires programming models and abstractions that can cover a diverse set of problem domains, including iterative refinement algorithms, graph transformations, graph aggregations, pattern matching, ego-network analysis, and graph traversals. Several high-level programming abstractions have been proposed and adopted by distributed graph processing systems and big data platforms. Even though there exists significant work that experimentally compares distributed graph processing frameworks, no qualitative study and comparison of graph programming abstractions has been conducted yet. In this chapter, we review and analyze the most prevalent high-level programming models for distributed graph processing, in terms of their semantics and applicability. We identify the classes of graph applications that can be naturally expressed by each abstraction. For each model, we also give examples of applications that are hard to express. We review 34 distributed graph processing systems with respect to the graph processing models they implement and we survey the applications that appear in recent distributed graph systems papers.

3.1 The Emergence of Distributed Graph Processing

Graphs are immensely useful data representations, vital to diverse data mining applications. Graphs capture relationships between data items, like interactions or dependencies, and their analysis can reveal valuable insights for machine learning tasks, anomaly detection, clustering, recommendations, social influence analysis, bioinformatics, and other application domains.

The rapid growth of available datasets has established distributed shared-nothing cluster architectures as one of the most common solutions for big data processing. Vast social networks with millions of users and billions of user-interactions, web access history, product ratings, and networks of online game activity are a few examples of graph datasets that might not fit in the memory of a single machine. Such massive graphs are usually partitioned over several machines and processed in a distributed fashion. However, coping with huge data sizes is not the sole motivation for distributed graph processing. Graphs rarely appear as *raw* data; they are most often derived by transforming other datasets into graphs. As we show in Chapter 4, data entities of interest are extracted and modeled as graph nodes and their relationships are modeled as edges. Thus, graph representations frequently appear in an intermediate step of some larger distributed data processing pipeline. Such intermediate graph-structured data are already partitioned upon creation and, thus, distributed algorithms are essential in order to efficiently analyze them and avoid expensive data transfers.

The increasing interest in distributed graph processing is largely visible in both academia and industry. Highly influential papers on parallel and distributed graph processing have been recently published [63, 127] and dominate the proceedings of prime data management conferences [87, 42, 186, 53, 147, 161, 103, 188]. At the same time, open-source distributed graph processing systems have gained popularity [53, 3, 122] and several general-purpose distributed data processing systems offer implementations of graph libraries and connectors to graph databases [82, 8, 42, 4].

Writing distributed graph mining applications is inherently hard. Computation parallelization, data partitioning, and communication management are major challenges of developing efficient distributed graph algorithms. Furthermore, graph applications are highly diverse and expose a variety of data access and communication patterns [69, 135]. For example, iterative refinement algorithms, like PageRank, can be expressed as parallel computations over the local neighborhood of each vertex, graph traversal algorithms produce unpredictable access patterns, while graph aggregations require grouping of similar vertices or edges together.

To address the challenges of distributed graph processing, several high-level programming abstractions and respective system implementations have been recently proposed [122, 125, 176, 174, 148]. Each abstraction is optimized for certain classes of graph applications. For instance, the popular vertex-centric model [125] is well-suitable for iterative value propagation algorithms, while the neighborhood-centric model [148] is designed to efficiently support operations on custom subgraphs, like ego networks. Unfortunately, there is no single model yet that can efficiently and intuitively cover all classes of graph applications.

Although several studies have experimentally compared the available distributed graph frameworks [86, 45, 28, 85, 123], there exists no *qualitative* comparison of the graph programming abstractions these frameworks offer. In this chapter, we review prevalent high-level abstractions for distributed graph processing, in terms of semantics, expressiveness, and applicability. We identify the classes of graph applications that can be naturally expressed by each abstraction and we give examples of application domains for which a model appears to be non-intuitive. We further analyze popular distributed graph processing systems, with regards to their implementations of programming models and semantic restrictions.

3.1.1 Notation

Here, we introduce the notation used for the execution semantics pseudocode and interfaces in the rest of this chapter. Let $G = (V, E)$ be a directed graph, where V denotes the set of vertices and E denotes the set of edges. An edge is represented as a pair of vertices, where the first vertex is the source and the second vertex is the target. For example, for $u, v \in V$, an edge from u to v is represented by the pair (u, v). Vertices and edges might have associated state (value) and local data structures of arbitrary type. For $v \in V$, S_v refers to $v's$ associated value and $S_{(u,v)}$ refers to the value associated with the edge (u, v). We define the set of first-hop, *out-neighbors* of vertex v as $N_v^{out} = \{u|u \in V \wedge (v, u) \in E\}$ and the set of first-hop *in-neighbors* of v as $N_v^{in} = \{u|u \in V \wedge (u, v) \in E\}$.

For the user-facing APIs, we use Java-like notation. We assume that each vertex can be identified by a unique ID of an arbitrary type, to which we refer with I. Similarly, we use VV to refer to the type of the vertex state, EV for the type of the edge value, M for message types, and T for arbitrary intermediate data types.

3.2 Programming Abstractions for Distributed Graph Processing

In this section, we review high-level programming models for distributed graph processing. A distributed graph programming model is an abstraction of the underlying computing infrastructure that allows for the definition of graph data structures and the expression of graph algorithms. We consider a distributed programming model to be *high-level* if it hides data partitioning and communication mechanisms from the end user. Thus, programmers can concentrate on the logic of their algorithms and do not have to care about data representation, communication patterns, and underlying system architecture. High-level programming models are inevitably less flexible than low-level models, and limit the degree of customization they allow. On the other hand, they offer simplicity and facilitate the development of automatic optimization.

We provide a high-level description of main abstractions and user-facing APIs. We give examples of representative applications and examples of algorithms that are difficult to express with certain abstractions. We also review implementation variants, known performance limitations, and proposed optimizations. We describe six models that were developed *specifically* for distributed graph processing, namely the vertex-centric, scatter-gather, gather-sum-apply-scatter, subgraph-centric, filter-process, and graph traversals.

3.2.1 Vertex-Centric

The vertex-centric model, introduced in the Pregel paper [125], is one of the most popular abstractions for large-scale distributed graph processing. Inspired by the simplicity of MapReduce [59], which only requires the user to implement two functions, map and reduce, while hiding the complexity of communication and data distribution, in the vertex-centric model users describe the computation as a vertex program. Also known as the *think like a vertex*-model, it requires users to express the computation logic from the point of view of a single vertex, by providing one user-defined *vertex function*.

A vertex-centric program receives a directed graph and a vertex function as input. A vertex serves as the unit of parallelization and has local state that consists of a unique ID, an optional vertex value, and its out-going edges, with optional edge values. Vertices communicate with other vertices through messages. A vertex can send a message to any other vertex in the graph, provided that it knows the destination's unique ID.

The execution workflow and computation parallelization of the vertex-centric model is shown in Figure 3.1. The dotted boxes correspond to parallelization units. Vertices can be in two states: *active* or *inactive*. Initially, all vertices are active. The computation proceeds in synchronized iterations, called *supersteps*. In each superstep, all active vertices execute the same user-defined computation in parallel. Supersteps are executed synchronously, so that messages sent during one superstep are guaranteed to be delivered in the beginning of the next superstep and be available to the vertex function of the receiving vertex. The output of a vertex-centric program is the set of vertex values at the end of the computation. If the graph is partitioned over several machines, a partition can contain several vertices and may have multiple worker-threads executing the vertex functions.

For simplicity of presentation, we define two auxiliary local data structures for each vertex. During a superstep, $inbox_v$ contains all the messages that were sent to vertex v during the previous superstep. $inbox_v$ is empty during the first superstep for all vertices. $outbox_v$ stores all the messages that a vertex v produces during a superstep. Message-passing is done as follows. At the end of each superstep, the runtime takes care of message delivery, by going through the $outbox$ of each sending vertex and placing the corresponding messages to the $inbox$ of each destination vertex. Message-passing can be implemented in batch or pipelined fashion. In the first case, messages are buffered in the local outbox of each vertex and are delivered in batches at the end of each superstep. In the case of pipelining, the runtime delivers messages to destination vertices as soon as they are produced. Pipelined message-passing might improve performance and lower memory requirements, but it limits ability of pre-aggregating the results at the outbox using combiners [125].

The execution semantics of the vertex-centric model are shown in Algorithm 1. Initially, all vertices are active. At the beginning of each superstep, the runtime delivers messages to vertices ($receiveMessages$ function). Then, the user-defined $compute$ function is invoked in parallel for each vertex. It receives a set of messages as input and can produce one or more messages as output. At the end of a superstep, the runtime receives the messages from the $outbox$ of each vertex and computes the set of active vertices for the superstep. The execution terminates when there are no active vertices or when some user-defined convergence condition is met.

The user-facing interface of a vertex-centric program is shown in Interface 1. The user-defined vertex function can read and update the vertex value and has access to all out-going edges. It can send a message to any vertex in the graph, addressing it by its unique ID. A vertex declares that it wants to become inactive by *voting to halt*. If an inactive vertex receives a message, it becomes active again. In many implementations of the vertex-centric

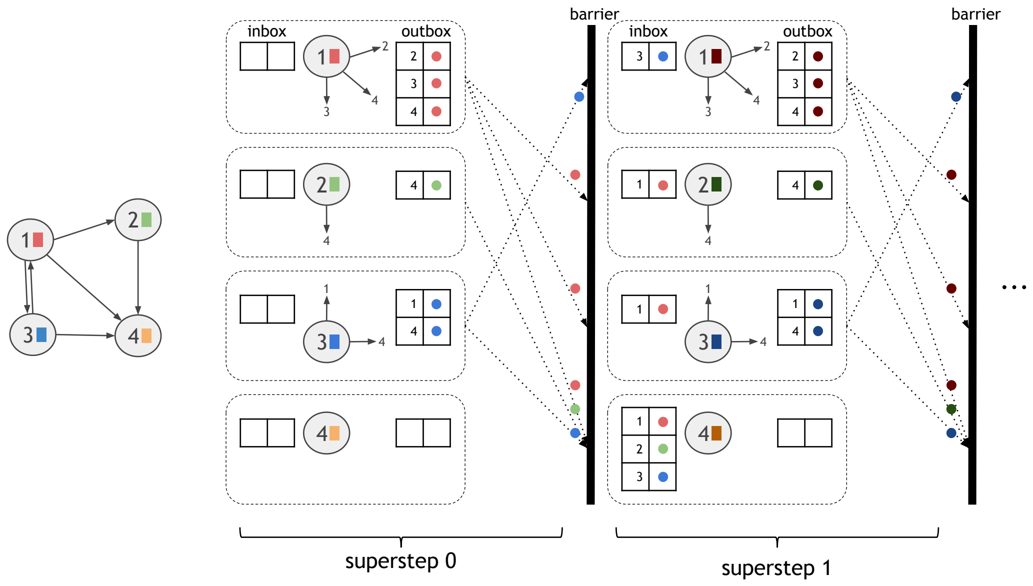


Figure 3.1 – Superstep execution and message-passing in the vertex-centric model. Graph edges can also have associated values, but we omit them here for simplicity. Each dotted box represents the computation scope of a vertex. Vertices in different scopes might reside on the same or different physical partitions. Arrows denote communication actions.

Algorithm 1 Vertex-Centric Model Semantics

```
**Input:** directed graph G=(V, E)
$activeVertices \leftarrow V$
$superstep \leftarrow 0$
**while** $activeVertices \neq \emptyset$ **do**
    **for** $v \in activeVertices$ **do**
        $inbox_v \leftarrow receiveMessages(v)$
        $outbox_v = compute(inbox_v)$
    **end for**
    $superstep \leftarrow superstep + 1$
**end while**
```

model, vertices can also add or remove a local edge or issue a mutation request for adding or removing non-local edges or vertices.

The GraphLab variant

GraphLab [122] generalizes the vertex-centric model by introducing the notion of a vertex *scope*. The scope of a vertex contains the adjacent edges, as well as the values of adjacent vertices. The vertex function is applied over the current state of a vertex and its scope. It returns updated values for the scope (a vertex can mutate the state of its neighbors) and a set of vertices *T*, which will be scheduled for execution.

Interface 1 Vertex-Centric Model

```
void abstract compute(Iterator[M] messages);
VV getValue();
void setValue(VV newValue);
void sendMessageTo(I target, M message);
Iterator getOutEdges();
int superstep();
void voteToHalt();
```

We observe the equivalence mapping between GraphLab's vertex-centric model and the Pregel, described in the previous section. In Pregel, the scope corresponds to the local vertex state together with the messages received from the neighboring vertices, and the set *T* contains the active vertices. Instead of message-passing, GraphLab implements the *pull* model, where vertices can read the values of neighbors in a defined *scope*, and uses a shared-memory model to enable communication between vertices. GraphLab's vertex-centric programming model variant, allows for *dynamic computation*, different consistency models and asynchronous execution. However, it also poses two limitations: (1) vertices can only communicate with their immediate neighbors, and (2) the graph structure has to be static, so that no mutations are allowed during execution. The shared-memory abstraction for communication is also adopted by Cyclops [48], even though its model is more restricted than the one provided by GraphLab.

Applicability and expressiveness

The vertex-centric model is general enough to express a broad set of graph algorithms. The model is a good fit when the computation can be expressed as a local vertex function which only needs to access data on adjacent vertices and edges. Iterative value-propagation algorithms and fixed point methods map naturally to the vertex-centric abstraction.

PageRank [41] is a representative algorithm that can be easily expressed in the vertex-centric model. In this algorithm, each vertex iteratively updates its rank by applying a formula on the sum of the ranks of its neighbors. The pseudocode for the PageRank vertex function is shown in Algorithm 3. Initially, all ranks are set to $1/numVertices()$. In each superstep, vertices send their partial rank along their outgoing edges and use the received partial ranks from their neighbors to update their ranks, according to the PageRank formula. After a certain number of supersteps (30 in this example) all vertices vote to halt and the algorithm terminates. In this pseudocode, we see that the superstep number is used to differentiate the computation between the first iteration and the rest of the iterations. This is a common pattern in vertex-centric applications, since during superstep 0, no messages have been received by any vertex yet.

Non-iterative and asynchronous graph algorithms might be difficult to express in the vertex-centric model for which the concept of synchronized supersteps is central. Furthermore, expressing a computation from the perspective of a vertex can often be challenging, as it also requires expressing all non-local state updates (further than one hop) as messages.

Algorithm 3 PageRank Vertex Function

```
void compute(Iterator[double] messages):
  $outEdges = getOutEdges().size()$
  if $superstep() > 0$ then
      double $sum = 0$
      for $m \in messages$ do
          $sum \leftarrow sum + m$
      end for
      $setValue(0.15/numVertices() + 0.85 * sum)$
  end if
  if $superstep() < 30$ then
      for $edge \in getOutEdges()$ do
          $sendMessageTo(edge.target(), getValue()/outEdges)$
      end for
  else
      $voteToHalt()$
  end if
```

Graph transformations and single-pass graph algorithms, like triangle counting, are not a good fit for the vertex-centric model.

Let us consider an implementation of a triangle counting algorithm in the vertex-centric model. A triangle consists of three vertices which all form edges between them. In a MapReduce-like data processing model, we would solve this problem by generating triads and checking how many of the triads form triangles. In the vertex-centric model, however, we need to formulate the computation logic from the perspective of a vertex. That is, a vertex has to make a local decision on whether it is a member of a triangle. To detect a triangle, a vertex needs to know whether any two of its neighbors are connected. Since vertices can immediately access information about their edges only, they need to ask their neighbors for further information, using message-passing. The main idea is to propagate a message along the edges of a triangle, so that when the message returns to the originator vertex, the triangle can be detected. In order not to count the same triangle multiple times, messages are only propagated from vertices with lower IDs to vertices with higher IDs. The algorithm is shown in Figure 3.2 and proceeds in three supersteps. During the first superstep, each vertex sends its ID to all neighbors with higher ID than its own. During the second superstep, each vertex attaches its own ID to every received message and propagates the pair of IDs to neighbors with higher IDs. During the final superstep, each vertex checks the received pairs of IDs to detect whether a triangle exists.

Another non-intuitive computation pattern is sending messages to the in-neighbors of a vertex. Computing strongly connected components is an algorithm that contains this pattern [160]. Remember that each vertex only has access to its out-going edges and can only send messages to vertices with a known ID. Thus, if a vertex needs to communicate with its in-neighbors, it has to use a pre-processing superstep, during which, each vertex sends a message containing its own ID to all its out-neighbors. This way, all vertices will

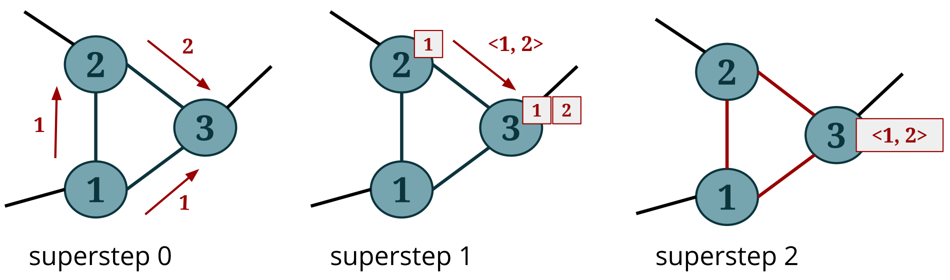


Figure 3.2 – A triangle counting algorithm in the vertex-centric model. Messages produced by a vertex during the current superstep are shown with red arrows. Messages received in the current superstep are shown in grey boxes.

know the IDs of all their in-neighbors in the following superstep.

Performance optimizations

The vertex-centric model has been successful due to its simplicity and linear scalability for iterative graph algorithms and it has inspired research and engineering efforts on optimizing the performance of its implementations. In this section, we summarize some of the results of work on performance optimizations for vertex-centric programs.

Communication can often become a bottleneck in the vertex-centric message-passing model. An overview of the model's limitations with regard to communication bottlenecks and worker load imbalance is presented in [187]. The authors show that high-degree vertices or custom algorithm logic can create communication and computation skew, so that a small number of vertices produce many more messages than the rest, thus also increasing the workload of the worker machines where they reside. The authors address this issue with two message reduction techniques. First, they use *mirroring*, a mechanism that creates copies of high-degree vertices on different machines, so that communication with neighbors can be local. A similar technique is also presented in [159]. Second, they introduce a *request-response* mechanism, that allows a vertex to request the value of any other vertex in the graph, even if it is not a neighbor. For not neighboring vertices, such a process would require three supersteps in the vanilla vertex-centric model. High communication load can also be avoided by using sophisticated partitioning mechanisms [47, 159].

Using synchronization barriers in the vertex-centric model allows programmers to write deterministic programs and easily reason about and debug their code. However, global barriers limit concurrency and may cause unnecessary synchronization, and as a consequence, poor performance, especially for applications with irregular or dynamic parallelism. In fact, various graph algorithms can benefit from asynchronous [184, 33, 122] or hybrid [180, 72] execution models. In [87], the authors propose the *barrierless asynchronous parallel* BAP model, to reduce stale messages and the frequency of global synchronization. The model allows vertices to immediately access messages they have received and utilizes only barriers local to each worker, which do not require global coordination.

Several algorithm-specific optimization techniques for the vertex-centric model are proposed in [160, 189]. Specifically, the authors exploit the phenomenon of *asymmetric convergence* often encountered in graph algorithms. We say that an iterative fixpoint graph algorithm converges asymmetrically, if different parts of the graph converge at different speeds. As a result, the overall algorithm converges slowly because, during the final supersteps, only a small fraction of vertices are still active. The proposed solution is to monitor the active portion of the graph and, when it drops under some threshold, ship it to the master node, which executes the rest of the computation. Other optimizations include trading memory for communication and performing mutations (edge deletions) lazily.

In [161], the authors propose using special data structures, such as bit-vectors, to represent the neighborhood of each vertex. They also suggest compressing intermediate data that needs to be communicated over the network. They find that overlapping computation and communication, sophisticated partitioning, and different message-passing mechanisms have a significant impact on performance. Another issue they identify is that many algorithms have high memory requirements because of the outbox data structures of vertices growing too large. The authors propose to break each superstep into a number of smaller supersteps and processing only a subset of the vertices in each smaller superstep. Similar techniques are used in [53, 103].

3.2.2 Scatter-Gather

The *Scatter-Gather* abstraction, also known as *Signal-Collect* [172], is a vertex-parallel model, sharing the same *think like a vertex* philosophy as the vertex-centric model. Scatter-Gather also operates in synchronized iteration steps and uses a message-passing mechanism for communication between vertices. The main difference is that each iteration step contains two computation phases, *scatter* and *gather*. Thus, the user also has to provide two computation functions, one for each phase.

The model phases are graphically shown in Figure 3.3. Scatter-Gather decouples the sending of messages from the collection of messages and state update. During the scatter phase, each vertex executes a user-defined function that sends messages or *signals* along out-going edges. During the gather phase, each vertex collects messages from neighbors and executes a second user-defined function that uses the received messages to update the vertex state. It is important to note that, contrary to the vertex-centric model, in Scatter-Gather both message sending and receipt happen during the *same* iteration step. That is, during iteration i, the gather phase has access to the messages sent in the scatter phase of iteration i.

The execution semantics of the Scatter-Gather abstraction are shown in Algorithm 4. The input of a Scatter-Gather program is a directed graph and the output is the state of the vertices after a maximum number of iterations or some custom convergence condition has been met. Similarly to the vertex-centric model, vertices can be in an active or inactive state. Initially, all vertices are active. Vertices can either explicitly vote to halt, like in the vertex-centric model, or implicitly get deactivated, if their value does not change during an iteration step. If a vertex does not update its value during a gather phase, then it does not have to execute a scatter phase in the next iteration, because its neighbors have already received its latest information. Thus, it gets deactivated.

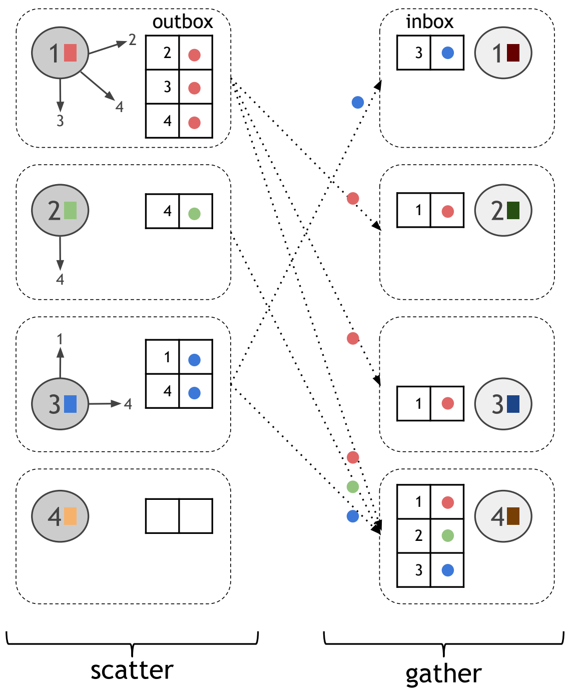


Figure 3.3 – One iteration in the Scatter-Gather model. In the Scatter phase, each vertex has read-access to its state, write-access to its outbox, and no access to its inbox. In the Gather phase, a vertex has read-access to its inbox, write-access to its state, but no access to its outbox.

The user-facing interfaces of Scatter and Gather are shown in Interfaces 2 and 3, respectively. Note that the scopes of the two phases are separate and each interface has different available methods. The scatter interface can retrieve the current vertex value, read the state of the neighboring edges, and send messages to neighboring vertices. The gather interface can access received messages, read and set the vertex value.

Applicability and expressiveness

Scatter-Gather can be used to express a variety of algorithms in a concise and elegant way. Similarly to the vertex-centric model, iterative, value-propagation algorithms like PageRank are a good fit for Scatter-Gather. Since the logic of producing messages is decoupled from the logic of updating vertex values, programs written using Scatter-Gather are sometimes easier to follow and maintain. The vertex-centric PageRank example that we saw in the previous section can be easily expressed in the Scatter-Gather model, by simply separating the sending of messages and the calculation of ranks in two phases. The pseudocode is shown in Algorithm 7. The scatter phase contains only the the rank propagation logic, while the gather phase contains the message processing and rank update logic.

Separating the messaging phase from the vertex value update logic not only makes

Algorithm 4 Scatter-Gather Model Semantics

```
Input: directed graph G=(V, E)
$activeVertices \leftarrow V$
$superstep \leftarrow 0$
while $activeVertices \neq \emptyset$ do
    $activeVertices' \leftarrow \emptyset$
    for $v \in activeVertices$ do
        $outbox_v \leftarrow scatter(v)$
        $S_v' \leftarrow gather(inbox_v, S_v)$
        if $S_v' \neq S_v$ then
            $S_v \leftarrow S_v'$
            $activeVertices' \leftarrow activeVertices' \cup v$
        end if
    end for
    $activeVertices \leftarrow activeVertices'$
    $superstep \leftarrow superstep + 1$
end while
```

Interface 2 Scatter

void abstract **scatter**();
VV **getValue**();
void **sendMessageTo**(I target, M message);
Iterator **getOutEdges**();
int **superstep**();

Interface 3 Gather

void abstract **gather**(Iterator[M] messages);
void **setValue**(VV newValue);
VV **getValue**();
int **superstep**();

some programs easier to follow but might also have a positive impact on performance. Scatter-Gather implementations typically have lower memory requirements, because concurrent access to the inbox (messages received) and outbox (messages to send) data structures is not required. However, this characteristic also limits expressiveness and makes some computation patterns non-intuitive. If an algorithm requires a vertex to concurrently access its inbox and outbox, then the expression of this algorithm in Scatter-Gather might be problematic. Strongly Connected Components and Approximate Maximum Weight Matching [160] are examples of such graph algorithms. A direct consequence of this restriction is that vertices cannot generate messages and update their states in the same phase. In the scatter phase, vertices have read-access to their state and adjacent edges, write-access to their outbox, and no access to their inbox. In the gather phase, vertices have read-access to their inbox, write-access to their state, but no access to their outbox or adjacent edges.

Algorithm 7 PageRank Scatter and Gather Functions

```
void **scatter**():
  $outEdges = getOutEdges().size()$
  **for** $edge \in getOutEdges()$ **do**
    $sendMessageTo(edge.target(), getValue()/outEdges)$
  **end for**
void **gather**(Iterator[**double**] messages):
  **if** $superstep() < 30$ **then**
    **double** $sum = 0$
    **for** $m \in messages$ **do**
      $sum \leftarrow sum + m$
    **end for**
    $setValue(0.15/numVertices() + 0.85 * sum)$
  **end if**
```

Thus, deciding whether to propagate a message based on its content would require storing it in the vertex value, so that the gather phase has access to it, in the following iteration step. Similarly, if the vertex update logic includes computation over the values of the neighboring edges, these have to be included inside a special message passed from the scatter to the gather phase. Such workarounds often lead to higher memory requirements and non-elegant, hard to understand algorithm implementations.

For example, consider the problem of finding whether there exists a path from source vertex a to target vertex b, with total distance equal to a specified user value d. Let us assume that the input graph has edges with positive values corresponding to distances. We can solve this problem in a message-passing vertex-parallel way, by iteratively propagating messages through the graph and aggregating edge weights on the way. Messages originate from the source vertex and are routed towards the target vertex, one neighborhood hop per superstep. When a vertex v receives a message, it decides whether to propagate the message to a neighbor u, based on the current distance that the message contains plus the distance of the edge that connects v with u. If the computed sum is less than or equal to d, v updates the message content and propagates it to u. If the sum exceeds the value of d, then v drops the message. In the vertex-centric model, this logic can be implemented inside the vertex compute function, since vertices receive and send messages during the same phase. However, in Gather-Scatter, vertices receive messages in the gather phase, but can only generate messages in the scatter phase. In order for a vertex to know whether to propagate a message based on its content, it needs a mechanism to allow the scatter phase to access messages received in the previous superstep. One way that this can be achieved is by storing all received messages in the vertex value, so that the scatter interface can access them in the next superstep.

3.2.3 Gather-Sum-Apply-Scatter (GAS)

The Gather-Sum-Apply-Scatter (GAS) programming abstraction, introduced by Powergraph [81], tries to address performance issues that arise when using the vertex-centric or scatter-gather model on power-law graphs. In such graphs, most vertices have relatively few neighbors, while few vertices have a very large number of neighbors. This degree *skew* causes computational imbalance in vertex-parallel programming models. The few high-degree vertices, having much more work to do during a superstep, act as stragglers, thus, slowing down the overall execution.

The GAS model addresses the bottlenecks caused by high-degree vertices by parallelizing the computation over the *edges* of the graph. The abstraction essentially decomposes a vertex-program in separate phases, which allow distributing the computation more effectively over a cluster. The computation proceeds in four phases, each executing a user-defined function. During the *gather* phase, a user-defined function is applied on each of the adjacent edges of each vertex in parallel, where an edge contains both the source vertex and the target vertex values. The transformed edges are passed to an *associative and commutative* user-defined function, which combines them to a single value during the *sum* phase. The gather and sum phases naturally correspond to a *map-reduce* step and are sometimes considered as a single phase [81]. The result of the sum phase and the current state of each vertex are passed to the *apply* user-defined function, which uses them to compute the new vertex state. During the final *scatter* phase, a user-defined function is invoked in parallel per edge, having access to the updated source and target vertex values. In some implementations of the model, the scatter phase is either optional or omitted. The four phases of the GAS model are graphically shown in Figure 3.4 and its execution semantics in Algorithm 8.

Algorithm 8 GAS Model Semantics

```
Input: directed graph G=(V, E)
$a_v \leftarrow$ empty
for $v \in V$ do
    for $n \in N_v^{in}$ do
        $a_v \leftarrow sum(a_v, gather(S_v, S_{(v,n)}, S_n))$
    end for
    $S_v \leftarrow apply(S_v, a_v)$
    $S_{(v,n)} \leftarrow scatter(S_v, S_{(v,n)}, S_n)$
end for
```

The public interfaces of the GAS abstraction are shown in Interface 4. Note how these interfaces are simpler and more restrictive than the vertex-centric and gather-scatter interfaces. All four user-defined functions return a single value and the scope of computation is restricted to local neighborhoods.

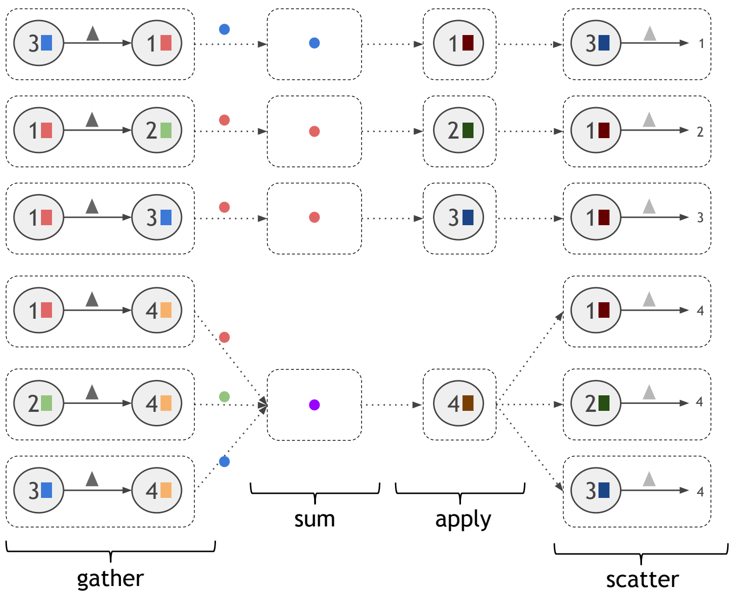


Figure 3.4 – The GAS computation phases. The gather phase parallelizes the computation over the edges of the input graph. The user-defined function has access to an edge, including its source and target vertex states. The sum phase combines partial values using a user-defined associative and commutative function. The vertex states are updated during the apply phase. The scatter phase is optional and can be used to update edge states.

Interface 4 Gather-Sum-Apply-Scatter

```
T abstract gather(VV sourceV, EV edgeV, VV targetV);
T abstract sum(T left, T right);
VV abstract apply(VV value, T sum);
EV abstract scatter(VV newV, EV edgeV, VV oldV);
```

Applicability and expressiveness

The GAS abstraction imposes the restriction of an associative and commutative sum function to produce edge-parallel programs that will not suffer from computational skew. Nevertheless, the model can be used to emulate vertex-centric programs, even if the update function is not associative and commutative. To express a vertex-centric computation, the gather and sum functions can be used to combine the inbound messages (stored as edge data) and concatenate the list of neighbors needed to compute the outbound messages. The vertex compute function is then executed inside the apply phase. The apply user-defined function generates the messages, which can then be passed as vertex data to the scatter phase. Similarly, to emulate a GraphLab vertex program, the gather and sum functions can be used to concatenate all the data on adjacent vertices and edges and then run the vertex function inside the apply phase.

Executing vertex-parallel programs inside the apply function results in complexity linear in the vertex degree, thus defeating the purpose of eliminating computational skew. Moreover, manually constructing the neighborhood in the sum phase and concatenating messages in order to access them in the apply phase, are both non-intuitive and computationally expensive. Fortunately, many graph algorithms can be decomposed into a gather transformation and an associative-commutative sum function. Algorithm 10 shows a PageRank implementation using the GAS interfaces. The gather phase computes a partial rank for each neighbor. The sum phase sums up all the partial ranks into a single value, and the the apply phase computes the new PageRank and updates the vertex value. The scatter phase has been omitted, since edge values do not get updated in this algorithm. Alternatively, the scatter phase can be used to selectively activate vertices for the next iteration [81].

Algorithm 10 PageRank Gather, Sum, Apply

```
double gather(double src, double edge, double trg):
    return $trg.value/trg.outNeighbors$
double sum(double rank1, double rank2):
    return $rank1 + rank2$
double apply(double sum, double currentRank):
    return $0.15 + 0.85 * sum$
```

If the algorithm cannot be decomposed into a gather step and a commutative-associative sum, implementation in the GAS model might require manual emulation of the vertex-centric or scatter-gather models, as described previously. For example, in the Label Propagation algorithm [150], a vertex receives labels from its neighbors and chooses the most frequent label as its vertex value. Computing the most frequent item is not an associative-commutative function. In order to express this algorithm in GAS, the sum phase needs to construct a set of all the neighbor labels. The gather user-defined function returns a set containing a single label for each neighbor. The sum user-defined function receives a pair of sets, each containing a neighbor's label and returns the sets' union. Finally, each vertex chooses the most frequent label in the apply function, which has access to all labels.

Table 3.1 shows a comparison among the vertex-centric, gather-scatter, and GAS programming models, with regard to update functions and communication. The vertex-centric model is the most generic of the three, allowing for arbitrary vertex-update functions and communication logic, while GAS is the most restrictive of the three.

Performance optimizations

In [49], the authors find that, even though the GAS model manages to overcome the load imbalance issues caused by high-degree vertices, at the same time it poses a high memory and communication overhead for the low-degree vertices of the input graph. They propose a *differentiated vertex computation* model, where the high-degree vertices are processed using the GAS model, while the low-degree vertices are processed using a GraphLab-like vertex-centric model.

Table 3.1 – Comparison of the vertex-centric, gather-scatter, and GAS programming models.

	update function properties	**update function logic**	**communication scope**	**communication logic**
Vertex-Centric	arbitrary	arbitrary	any vertex	arbitrary
Scatter-Gather	arbitrary	based on received messages	any vertex	based on vertex state
GAS	associative and commutative	based on neighbors' values	neighborhood	based on vertex state

3.2.4 Subgraph-Centric

All the models we have seen so far, vertex-centric, gather-scatter, and GAS, operate on the scope of a single vertex or edge in the graph. While such fine-grained abstractions for distributed programming are considerably easy to use by non-experts, they might cause high communication overhead when compared to coarse-grained abstractions. In this section, we present two distributed graph processing models that operate on the *subgraph* level, with the objective to reduce communication and exploit the subgraph structure to enable the implementation of optimized algorithms.

Partition-Centric

The partition-centric model offers a lower-level abstraction than the vertex-centric, setting the whole partition as the unit of parallel computation. The model exposes the subgraph of each partition to the user function, in order to avoid redundant communication and accelerate convergence of vertex-centric programs. The abstraction has been introduced by Giraph++[176] and has been further optimized in succeeding works [170, 186].

The the partition-centric model perceives each partition as a proper subgraph of the input graph, instead of a collection of unassociated vertices. While in the vertex-centric model a vertex is restricted to accessing information from its immediate neighbors, in the partition-centric model information can be propagated freely inside all the vertices of the same partition. This simple property of the partition-centric model can lead to significant communication savings and faster convergence.

The partition-centric model is graphically shown in Figure 3.5. Note that the whole partition becomes the parallelization unit on which the user-defined function is applied. As compared to the vertex-centric case, message exchange happens only between partitions, thus resulting in reduced communication costs. Inside a partition, vertices can be *internal* or *boundary*. Internal vertices are associated with their value, neighboring edges, and incoming messages. Boundary vertices only have a local *copy* of their associated value; the primary value resides in the partition where the vertex is internal. In Figure 3.5, ver-

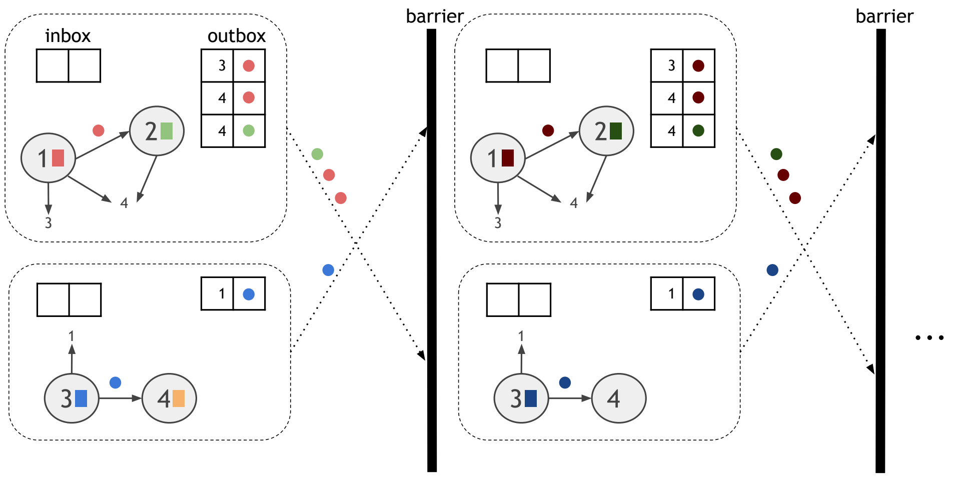


Figure 3.5 – Two iterations in the Partition-Centric model. Each dotted box represents a separate physical partition and dotted arrows represent communication.

tices 1 and 2 are internal in the upper partition, while vertices 3 and 4 are boundary. In the lower partition, vertices 3 and 4 are internal, while vertex 1 is boundary. Message exchange between internal vertices of the same partition is immediate, while messages to boundary vertices require network transfer.

The execution semantics of the partition-centric model are the same as in the vertex-centric model, with the only difference that the user-defined update function is invoked per partition and messages are distributed between partitions, not vertices. The user-facing interface of the model offers most of the methods of the vertex-centric interface, with a few additional ones, shown in Interface 5. The interface needs to provide a mechanism to check whether a vertex belongs to a particular partition and to retrieve internal and boundary vertices inside a partition. Note that since the scope of the user-defined function is not a single vertex, the methods to retrieve and update vertex attributes are not part of the *compute()* interface.

Interface 5 Partition-Centric Model

void abstract **compute**();
void **sendMessageTo**(I target, M message);
int **superstep**();
void **voteToHalt**();
boolean **containsVertex**(I id);
boolean **isInternalVertex**(I id);
boolean **isBoundaryVertex**(I id);
Collection **getInternalVertices**();
Collection **getBoundaryVertices**();
Collection **getAllVertices**();

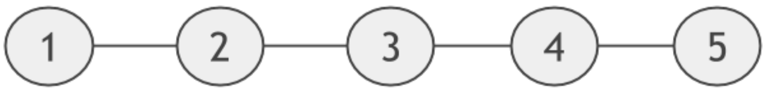


Figure 3.6 – A chain example graph.

Neighborhood-Centric

The neighborhood-centric model [148] sets the scope of computation on *custom* subgraphs of the input graph. These subgraphs are explicitly built around vertices and their multi-hop neighborhoods, in order to facilitate the implementation of graph algorithms that operate on ego-networks; networks built around a central vertex of interest. The user specifies custom subgraphs and a program to be executed on those subgraphs. The user program might be iterative and is executed in parallel on each subgraph, following the Bulk Synchronous protocol (BSP) [177]. In contrast to the partition-centric model, in an implementation of the neighborhood-centric model, a physical partition can contain one or more custom subgraphs. Information exchange happens through shared state updates for subgraphs in the same partition and through replicas and messages for subgraphs belonging to different partitions.

Applicability and expressiveness

The partition-centric model is a good fit when there exists an efficient sequential algorithm which can be executed in each partition and whose partial results can be easily combined to produce the global result. A drawback of the partition-centric model is that users have to switch from "thinking like a vertex" to thinking in terms of partitions. In order to reason about their algorithm, users need to understand what a partition represents and how to differentiate the behavior of an internal vertex versus that of a boundary vertex. The model allows for more control on computation and communication, but at the same time exposes low-level characteristics to users. This loss of abstraction might lead to erroneous or hard-to-understand programs.

Algorithm 12 shows the pseudocode of a partition-centric PageRank implementation. This implementation is quite longer and more complex than the ones we have seen so far. Part of the complexity is introduced because, inside each partition, PageRank computation is asynchronous. Each vertex has an additional attribute, *delta*, besides its PageRank score, where it stores intermediate updates from vertices inside the same partition. At the end of each superstep, boundary vertices with positive *delta* values produce messages to be delivered in other partitions.

The neighborhood-centric model is preferable when applications require accessing multi-hop neighborhoods or ego-centric networks of certain vertices. Such applications include personalized recommendations, social circle analysis, anomaly detection, and link prediction. While in the partition-centric model, the graph is partitioned into non-overlapping, application-independent subgraphs, in the neighborhood-centric model, subgraphs are extracted based on specific application criteria. This way, users can specify subgraphs to be

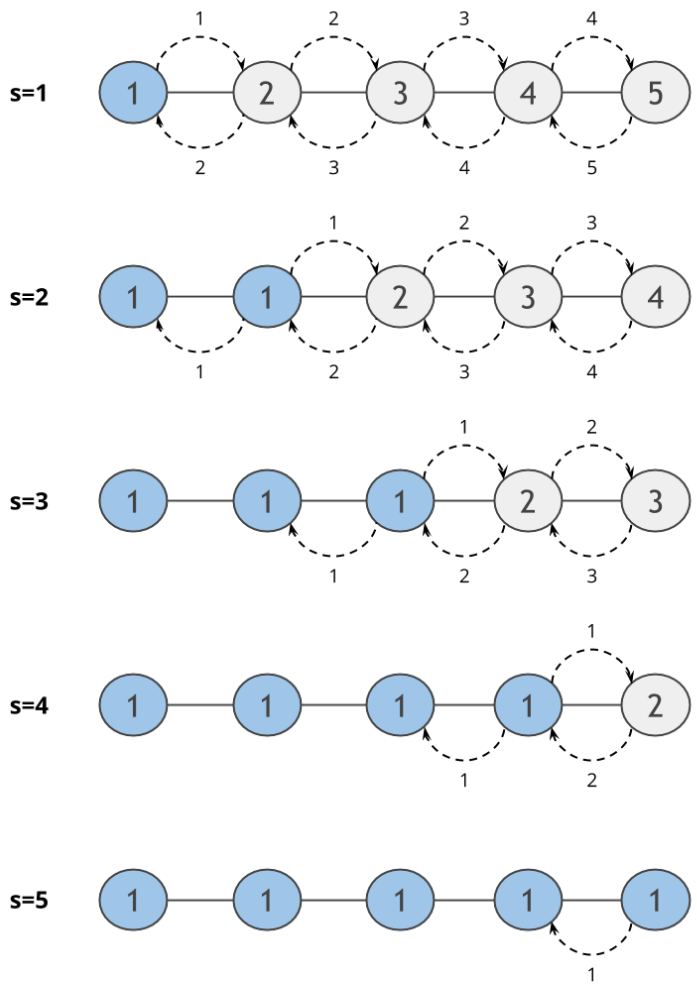


Figure 3.7 – Connected Components via label propagation in the vertex-centric model. The minimum value propagates to the end of the chain after 5 supersteps.

k-hop neighborhoods around a set of query vertices, satisfying a particular predicate. The user program executed on these subgraphs can have arbitrary random access to the state of the whole subgraph. Note that creating these overlapping neighborhoods requires an often expensive pre-processing step, in terms of both execution time and memory.

Performance optimizations

The performance of the partition-centric model is highly dependent on the quality of the partitions. If the partitioning technique used creates well-connected subgraphs and minimizes edge cuts between partitions, it is highly probable that a partition-centric implementation will require less communication than a vertex-centric implementation and that a

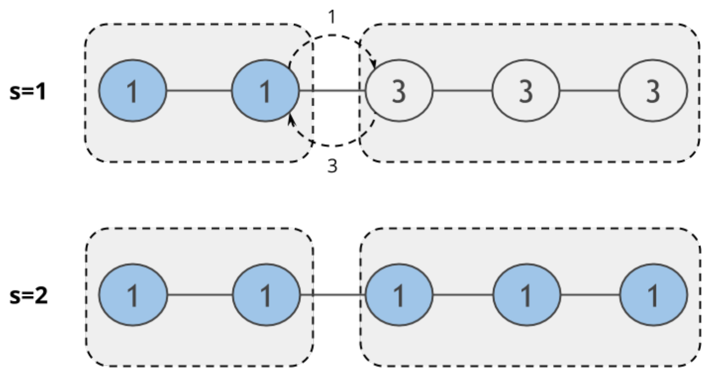


Figure 3.8 – Connected Components via label propagation in the partition-centric model. Vertices 1 and 2 belong to the first partition and vertices 3, 4, 5 belong to the second partition. Each partition converges asynchronously, before initiating communication with other partitions. The minimum value propagates to all vertices after 2 supersteps.

value-propagation algorithm will converge is fewer supersteps. For example, consider the execution of the connected components algorithm on a chain graph, like the one shown in Figure 3.6. In a vertex-centric execution of the algorithm, shown in Figure 3.7, the minimum value can only propagate one hop per superstep. Consequently, the algorithm requires as many supersteps as the maximum graph diameter plus one, in order to converge. In the parition-centric model instead, values can propagate asynchronously inside a partition. If the chain is partitioned in two connected subgraphs, like the ones of Figure 3.8, the algorithm converges after just two supersteps. However, if the graph is partitioned poorly, there will be little to no benefit compared to the vertex-centric execution. For instance, if we partition the given chain into two partitions of odd and even vertices, the algorithm will need as many supersteps to converge as in the vertex-centric case. Sophisticated partitioning can be an expensive task and users must carefully consider the pre-processing cost that it might impose to the total job execution time.

GoFFish [170] and Blogel [186] propose a subgraph-centric model that partitions the input graph in subgraphs which are *connected*. This way, well-known shared-memory algorithms can be applied on the connected subgraphs. A partition can contain multiple subgraphs and execute the computation on each subgraph in parallel. GoFFish also proposes a distributed persistent storage, optimized for subgraph access patterns. On the other hand, Blogel allows a subgraph to define and manage its own state, allowing for subgraph-level communication. Both GoFFish and Blogel are beneficial when the number of subgraphs is sufficiently larger than the number of workers, so that a balanced workload can be achieved.

3.2.5 Filter-Process

The *filter-process* computational model, also known as or "think like an *embedding*", is proposed by Arabesque [174]. Arabesque is a system for efficient distributed graph mining. An embedding is a subgraph instance of the input graph that matches a user-

Algorithm 12 PageRank Partition-Centric Function

```
void **compute**():
  **if** $superstep() > 0$ **then**
    **for** $v \in getAllVertices()$ **do**
      $v.getValue().pr = 0$
      $v.getValue().delta = 0$
    **end for**
  **end if**
  **for** $iv \in internalVertices()$ **do**
    **if** $superstep() > 0$ **then**
      $iv.getValue().delta+ = 0.15$
    **end if**
    $iv.getValue().delta+ = iv.getMessages()$
    **if** $iv.getValue().delta > 0$ **then**
      $iv.getValue().pr+ = iv.getValue().delta$
      $u = 0.85 * iv.getValue().delta/iv.getNumOutEdges()$
      **while** $iv.iterator.hasNext()$ **do**
        $neighbor = getVertex(iv.iterator().next())$
        $neighbor.getValue().delta+ = u$
      **end while**
    **end if**
    $iv.getValue().delta = 0$
  **end for**
  **for** $bv \in boundaryVertices()$ **do**
    **if** $bv.getValue().delta > 0$ **then**
      $sendMessageTo(bv.getVertexId(), bv.getValue().delta)$
      $bv.getValue().delta = 0$
    **end if**
  **end for**
```

specified pattern. The model facilitates the development of graph mining algorithms, which require subgraph enumeration and exploration. Such algorithms are challenging to express and efficiently support with a vertex-centric model, due to immense intermediate state and high computation requirements.

The programming model consists of two primary functions, *filter* and *process*. *Filter* examines whether a given embedding is eligible for processing and *process* executes some action on the embedding and may produce output. The model assumes immutable input graphs and connected graph patterns.

Computation proceeds in a sequence of *exploration steps*, using the BSP model. During each exploration step, the system explores and extends an input set of embeddings. First, a set of candidate embeddings is created, by extending the set of input embeddings. In the first exploration step the set of candidates contains all the edges or vertices of the input graph. Once the candidates have been produced, the filter function examines them and selects the ones that should be processed. The selected embeddings are then sent to the process function, which outputs user-defined values. Finally, the selected embeddings become the input set of embeddings for the next step. The computation terminates when there are no embeddings to be extended.

The filter-process model differs from the partition-centric and neighborhood-centric models, where partitions and subgraphs are generated *once*, at the beginning of the computation as a pre-processing step. In filter-process, embeddings are dynamically generated during the execution of exploration steps. The model is suitable for graph pattern mining problems, which require subgraph enumeration, such as network motif discovery, semantic data processing, and spam detection.

3.2.6 Graph Traversals

Distributed graph analysis through *graph traversals* is the programming model adopted by the Apache Tinkerpop project [4]. The system provides a graph traversal machine and language, called *Gremlin* [157], which supports distributed traversals via the Bulk Synchronous Parallel (BSP) computation model.

In the traversal model of graph databases, *traversers* walk through an input graph, following user-provided instructions. The Gremlin machine supports distributed graph traversals, by modeling traversers as messages. Vertices receive traversers, execute their traversal step, and, as a result, generate other traversers to be sent as messages to other vertices. Halted traversers are stored in a vertex attribute. The process terminates when no more traversers are being sent. The result of the computation is the aggregate of the locations of the halted traversers.

3.3 General-Purpose Programming Models used for Graph Processing

Except from the specialized programming models that we have reviewed so far, several general-purpose distributed programming models have also been used for graph processing. Here, we review five such data processing abstractions that have been used for developing graph applications, namely MapReduce, dataflow, linear algebra primitives, datalog, and shared partitioned tables.

3.3.1 MapReduce

MapReduce [59], extensively presented in Chapter 2, is a popular distributed programming model for large-scale data processing on commodity clusters. However, its programming model is not suitable for graph applications, which are often iterative and require multi-step computations. Several extensions of the model have been proposed in order to support such algorithms [43, 67, 106]. The main idea of these extensions is adding a driver program that can coordinate iterations, containing one or more MapReduce jobs. The main task of the driver is to submit a new job per iteration and track convergence. Iterations are typically chained by using the output of one MapReduce iteration as the input of the next.

Pegasus [106] implements a generalized iterative matrix-vector multiplication primitive, GIM-V, as a two-stage MapReduce algorithm. The graph is represented by two input files, corresponding to the vertices (vector) and edges (matrix). In the first stage, the map phase transforms the input edges to set the destination vertex as the key. The following reduce phase applies a user-defined *combine2* function on each group to produce partial values for each vertex. *combine2* corresponds to a multiplication of a matrix element with a vector element. In the second MapReduce stage, the mapper is an identity mapper and the reducer encapsulates two user-defined functions, *combineAll* and *assign*. *combineAll* corresponds to summing the partial multiplication results and *assign* writes the new result in the vector. GIM-V can be used to express many iterative graph algorithms, such as PageRank, diameter estimation, and connected components. HaLoop [43] supports iterative computations on top of Hadoop [7], by extending Hadoop with a caching and indexing mechanism, to avoid reloading iteration-invariant data and reduce communication costs. It also extends Hadoop's API, offering a way to define loops and termination conditions. Twister [67] also extends the MapReduce API to support the development of iterative computations, including graph algorithms. It offers primitives for broadcast and scatter data transfers and implements a publish-subscribe protocol for message passing.

3.3.2 Dataflow

Dataflow is a generalization of the MapReduce programming model, where a distributed application is represented by a Distributed Acyclic Graph (DAG) of operations. In the DAG, vertices correspond to data-parallel tasks and edges correspond to data *flowing* from one task to another. As opposed to MapReduce, dataflow execution plans are more flexible and operators can support more than one input and output. In the DAG model, iterations can be supported by loop unrolling [192], or by introducing complex iterate operators, as part of the execution DAG [72].

Spark [192], Stratosphere [23], Apache Flink [1], Hyracks [38], Asterix [31], and Dryad [92] are some of the general-purpose distributed execution engines implementing the DAG model for data-parallel analysis. Naiad [136] enriches the dataflow model with timestamps, representing logical points in computation. Using these timestamps, the *timely dataflow* computational model supports efficient incremental computation and nested loops.

Dataflow systems offer different levels of abstraction for writing distributed applications. In Dryad and early versions of Stratosphere, the user describes the DAG by explicitly creating task vertices and communication edges. User-defined functions are encapsulated in the vertices of the graph. Modern dataflow systems, like Apache Spark and Apache Flink, offer declarative APIs for expressing distributed data analysis applications. Data sets are represented by an abstraction that handles partitioning across machines, like RDDs [191], and operators define data transformations, like map, group, join, sort.

It has been shown that the vertex-centric, scatter-gather, and other iterative models can be mapped to relational operations [72, 82, 42, 101]. For example, by representing the graph as two data sets corresponding to the vertices and edges, the vertex-centric model can be emulated by a join followed by a group-by operation. Algorithm 13 shows an implementation of the PageRank algorithm in the Spark Scala API. The input edges ($links$)

are grouped by the source ID to create an adjacency list per vertex and $ranks$ are initialized to 1.0. Then, the $links$ are iteratively joined with the current $ranks$ to retrieve each node's neighbors' rank values. A reduce operation is applied on the neighbor ranks to compute the updated PageRank for each node. Spark, Flink, and AsterixDB, all currently offer high-level APIs and libraries for graph processing on their dataflow engines. Differential dataflow [129] also exposes a set of programming primitives, on top of which, higher-level programming models, such as vertex-centric, can be implemented.

Algorithm 13 PageRank in Apache Spark Scala API

```
Input lines: a list of space-separated ID node pairs
  val links = lines.map{ s =>
      val parts = s.split(" ")(parts(0), parts(1))
  }.distinct().groupByKey().cache()
  var ranks = links.mapValues(v => 1.0)

  for (i < − 1 to iters) {
      val contribs = links.join(ranks).values.flatMap{ case
          (urls, rank) =>
              val size = urls.size
              urls.map(url => (url, rank / size))
      }
      ranks = contribs.reduceByKey(_ + _)
          .mapValues(0.15 + 0.85 * _)
  }
  val output = ranks.collect()
```

Gelly

In this section, we describe Gelly, Apache Flink's graph processing API. Gelly demonstrates how the dataflow model can be used to implement several programming abstractions for distributed graph processing in the same system. Apache Flink is a promising platform for large-scale graph analytics, because of its native support for iterative computations. By leveraging Flink's delta iterations [72], Gelly is able to map various graph processing models, such as vertex-centric, scatter-gather, and gather-sum-apply, to dataflows. Gelly users can perform end-to-end data analysis, without having to build complex pipelines and combine different systems, since the Gelly API can be seamlessly used with Flink's DataSet API. Thus, pre-processing, graph creation, graph analysis, and post-processing can be implemented in the same application.

In Gelly, a graph is represented by a data set of vertices and a data set of edges. A vertex is defined by its unique ID and a value, whereas an edge is defined by its source ID, target ID and value. The library includes common graph metrics, transformations, mutations, and neighborhood aggregations. Graph metrics are methods that can be used to retrieve graph properties, such as the number of vertices, edges, and the node degrees.

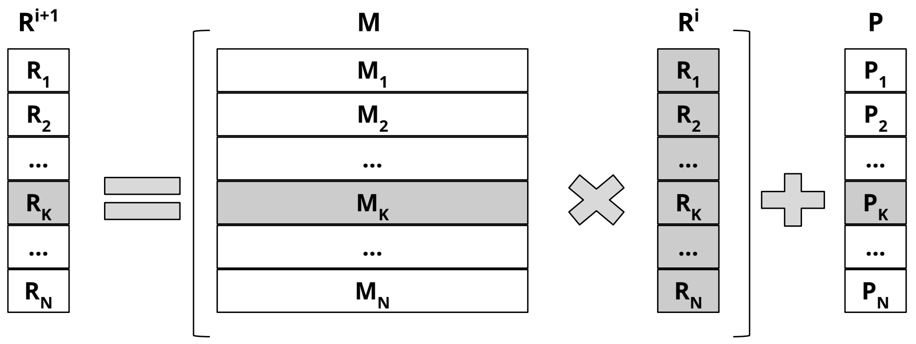


Figure 3.9 – PageRank computation as a vector-matrix multiplication. Vector R_i represents the computed ranks at iteration i, M is the input graph's adjacency matrix, with values scaled to account for the damping factor, and P contains the transition probabilities.

The transformation methods can be used to apply operations on a graph, in a declarative way. Neighborhood methods are single-step operations that allow vertices to perform an aggregation on their first-hop neighborhood. They provides a vertex-centric view, where each vertex can access its neighboring edges and neighbor values. Finally, Gelly provides APIs for writing iterative graph processing programs. It has support for vertex-centric, scatter-gather, and gather-sum-apply iterations. Gelly's iterative methods exploit Flink's efficient delta iteration operators.

3.3.3 Linear Algebra Primitives

Linear algebra primitives and operations have been long used to express graph algorithms. This model leverages the duality between a graph and its adjacency matrix representation [108]. A graph $G = (V, E)$ with N vertices can be represented by a $N \times N$ matrix M, where $M_{ij} = 1$ if there is an edge e_{ij} from node i to node j and 0 otherwise. Using this representation, many graph analysis algorithms can be expressed as a series of linear algebra operations. For example, a breadth-first search (BFS) starting from node i can be expressed as matrix multiplication. Starting from an initial $1xN$ vector y_0, where only the i_{th} element is non-zero, the multiplication $y_1 = y_0 * A$ will give the immediate neighbors of node i, $y_2 = y_1 * A$ will give the 2-hop neighbors of i, and so on. Figure 3.9 shows the computation of the PageRank algorithm in this model. Essentially, PageRank corresponds to the dominant eigenvector of the input graph's adjacency matrix. Starting from an initial vector of ranks R_0, PageRank can be computed by iteratively multiplying R_i with the adjacency matrix and adding the transition probabilities to get R_{i+1}, until convergence.

Combinatorial BLAS [44] and Presto [179] are two seminal works in porting the linear algebra model for graph processing to a distributed setting. In Combinatorial BLAS, the graph adjacency matrix is represented as a distributed sparse matrix and graph operations are mapped to linear algebra operations between sparse matrices and vectors. The main operations perform matrix-matrix multiplication and matrix-vector multiplication and require user-defined functions for addition and multiplication. Presto extends the R language to support a distributed array abstraction, which can represent both dense and sparse matrices. Computation is distributed automatically based on the data partitioning. For instance, in Figure 3.9, the shaded areas correspond to the data that is required for the computation of R_K. We notice that only a single row of the adjacency matrix is accessed and only one element of the transition probability vector. Thus, the computation task of R_K should be sent to the partition that contains M_K and P_K. Extracting such access patterns is essential for partitioning data and computation in the linear algebra model.

3.3.4 Datalog Extensions

Datalog is a declarative logic programming language used as query language for deductive databases. Datalog programs consist of a set of *rules*, which can be recursive. Datalog's support for recursion makes it suitable for expressing iterative graph algorithms. In [164], Datalog is enhanced with a set of extensions that allow users to define data distribution, efficient data structures for representing adjacency lists, and recursive aggregate functions, which can be efficiently evaluated using semi-naive evaluation [26]. Algorithm 14 shows the set of rules to compute an iteration of PageRank, using these extensions.

Algorithm 14 A PageRank iteration in SociaLite [164]

```
PageRank (iteration i+1) Input $N$: number of vertices
  EDGE (int $src$: 0..$N$, (int $sink$)).
  EDGECOUNT (int $src$: 0..$N$, int $cnt$).
  NODES (int $n$: 0..$N$).
  RANK (int $iter$ : 0..10, (int $node$: 0..$N$, int $rank$)).
  RANK($i$+1, $n$, $SUM($r$)) :- NODES($n$), $r$ = 0.15/$N$;
                                :- RANK($i$, $p$, $r_1$), EDGE($p$, $n$),
                                   EDGECOUNT($p$, $cnt$),
                                   $cnt > 0, r = 0.85 * r_1/cnt$.
```

A Datalog program is called *stratifiable* if it contains no negation operations within a recursive cycle. For such a program, there exists a unique greatest fixed point for its rules. In order to parallelize Datalog programs, it is assumed that all input programs are stratifiable. To parallelize the recursive aggregate functions, these have to be *monotone*, i.e. idempotent, commutative, and associative. Under these circumstances, it is shown in [164] that delta stepping [132] can be used to parallelize monotone recursive aggregate functions.

3.3.5 Shared Partitioned Tables

Distributed graph processing through shared partitioned tables is an idea implemented by Piccolo [146]. The computation is expressed as a series of user-provided *kernel* functions, which are executed in parallel and *control* functions, which are executed on a single machine. Kernel instances communicate through shared distributed, mutable state. This state is represented as in-memory tables whose elements are stored in the memory of different compute nodes. Kernels can use a key-value table interface to read and write entries to these tables. The tables are partitioned across machines, according to a user-provided partitioning function. Users are responsible for handling synchronization and resolution functions for concurrent writes in the shared tables.

3.4 Categorization of Distributed Graph Processing Systems

In this section, we present a taxonomy of recent distributed graph processing systems. Table 3.2 contains 34 such systems and compares them in terms of supported graph programming abstractions, execution model, and communication mechanisms. We also review graph analysis applications, as these appear in recent distributed graph processing systems publications. We present the results in Table 3.3.

3.4.1 Programming Model

The vertex-centric model appears to be the most commonly implemented abstraction among the systems that we consider. Pregel, Apache Giraph, Apache Hama, GPS, Mizan, Giraphx, Seraph, GiraphUC, Pregel+, and Pregelix (Apache AsterixDB) implement the full semantics of the model. GraphLab, LFGraph, Cyclops, Gelly, and Trinity do not support graph mutations, while the former three do not support communication with vertices outside of the defined scope / neighborhood either. GraphX provides an operator called *pregel* for iterative graph processing, but, in fact, this operator implements the GAS paradigm. Apache Tinkerpop provides connectors to Giraph and Spark, allowing the execution of graph traversals on top of their computation engines.

3.4.2 Execution Model

We encounter four execution techniques in the graph systems considered in this survey: synchronous (S), asynchronous (A), hybrid (H), and incremental (I). Synchronous execution refers to implementations where a global barrier separates one iteration from the next. In such a model, during iteration i, vertices perform updates based on values computed in iteration $i-1$. On the other hand, in the asynchronous execution model, computation is performed on the most recent state of the graph. Synchronization can happen either through shared memory or through local barriers and distributed coordination. In a hybrid execution model, synchronous and asynchronous modes can coexist. For example, in Giraph++, computation and communication inside each partition happens asynchronously, while cross-partition computation requires global synchronization points. Incremental execution refers to the ability of a system to efficiently update the computation when its input changes, without halting and re-computing everything from scratch.

Synchronous execution is naturally the most common design choice. It simplifies application development and facilitates debugging. Asynchronous execution is usually supported together with synchronous or hybrid. The incremental execution model is only supported in Naiad.

Table 3.2 – Distributed graph processing systems comparison in terms of supported programming models, execution model, and communication mechanisms. S stands for *synchronous*, A for *asynchronous*, H for *hybrid*, and I for *incremental*.

System	**Year**	**Programming Model**	**Execution**	**Communication**
Pegasus [106]	2009	MapReduce	S	Dataflow
Pregel [125]	2010	Vertex-Centric	S	Message-Passing
Signal/Collect [172]	2010	Scatter-Gather	A, S	Message-Passing
HaLoop [43]	2010	MapReduce	S	Dataflow
Twister [67]	2010	MapReduce	S	Dataflow
Piccolo [146]	2010	Partitioned Tables	S	Shared global state
Apache Giraph [53]	2011	Vertex-Centric	S	Message-Passing
Comb. BLAS [44]	2011	Linear Algebra	S	Message-Passing
Apache Hama [3]	2012	Vertex-Centric	S	Message-Passing
GraphLab [122]	2012	Vertex-Centric	A, S	Shared Memory
PowerGraph [81]	2012	GAS	A, S	Shared Memory
Giraph++ [176]	2013	Subgraph-Centric	H	Message-Passing
Naiad [136]	2013	Dataflow	A, S, I	Dataflow
GPS [159]	2013	Vertex-Centric	S	Message-Passing
Mizan [109]	2013	Vertex-Centric	S	Message-Passing
Presto [179]	2013	Linear Algebra	S	Dataflow
Giraphx [173]	2013	Vertex-Centric	A	Shared Memory
X-Pregel [27]	2013	Vertex-Centric	S	Message-Passing
LFGraph [89]	2013	Vertex-Centric	S	Shared Memory
SociaLite [164]	2013	Datalog Extensions	S	Message-Passing
Trinity [165]	2013	Vertex-Centric	A, S	Message-Passing, Shared Memory
Graphx [82]	2014	GAS	S	Dataflow
GoFFish [170]	2014	Subgraph-Centric	H	Message-Passing
Blogel [186]	2014	Vertex-Centric, Subgraph-Centric	H	Message-Passing
Seraph [185]	2014	Vertex-Centric	S	Message-Passing
Cyclops [48]	2014	Vertex-Centric	S	Message-Passing
GiraphUC [87]	2015	Vertex-Centric	A, H	Message-Passing
Gelly [8]	2015	Vertex-Centric, Scatter-Gather, GAS	S	Dataflow
Pregelix [42]	2015	Vertex-Centric	S	Dataflow
Apache Tinkerpop [4]	2015	Graph Traversals	S	Message-Passing
PowerLyra [49]	2015	GAS	S	Shared Memory
NScale [148]	2015	Neighborhood-Centric	S	Dataflow
Arabesque [174]	2015	Filter-Process	S	Message-Passing
Pregel+ [187]	2015	Vertex-Centric	S	Message-Passing

Table 3.3 – Applications used in distributed graph processing systems papers to demonstrate programming models and evaluate performance.

Application	#Appearances	Programming Models
PageRank (and variations)	31	Vertex-Centric, Scatter-Gather, GAS, Dataflow, Linear Algebra, Subgraph-Centric, Partitioned Tables, MapReduce, Datalog Extensions
Shortest Paths (and variations)	15	Vertex-Centric, Scatter-Gather, GAS, Dataflow, Linear Algebra, Subgraph-Centric, Datalog Extensions
Weakly Connected Components	12	Vertex-Centric, GAS, Dataflow, Subgraph-Centric, MapReduce
Graph Coloring	5	Vertex-Centric, Scatter-Gather, GAS, Linear Algebra
Alternate Least Squares (ALS)	4	Vertex-Centric, GAS, Linear Algebra
Triangle count	4	Vertex-Centric, Linear Algebra, Subgraph-Centric, Datalog Extensions
K-Means	4	Vertex-Centric, Linear Algebra, Subgraph-Centric, Partitioned Tables
Minimum Spanning Forest / Tree	4	Vertex-Centric
Belief Propagation (and variations)	2	Vertex-Centric
Strongly Connected Components	2	Vertex-Centric, Dataflow
Label Propagation	2	Vertex-Centric
Diameter Estimation	2	GAS, MapReduce
Clustering Coefficient	2	Subgraph-Centric, Datalog Extensions
Motif Counting	2	Subgraph-Centric, Filter-Process
BFS	2	Vertex-Centric, Subgraph-Centric
Centrality Measures	1	Linear Algebra
Markov Clustering	1	Linear Algebra
Find Mutual Neighbors	1	Datalog Extensions
SALSA	1	Dataflow
n-body	1	Partitioned Tables
Bipartite Matching	1	Vertex-Centric
Semi-Clustering	1	Vertex-Centric
Random Walk	1	Vertex-Centric
K-Core	1	Vertex-Centric
Approximate Max. Weight Matching	1	Vertex-Centric
Graph Coarsening	1	Subgraph-Centric
Identifying Weak Ties	1	Subgraph-Centric
Frequent Subgraph Mining	1	Filter-Process
Finding Cliques	1	Filter-Process

3.4.3 Communication Mechanisms

We come across four different communication mechanisms. In the Message-Passing model, the state is partitioned across worker tasks and updates to non-local state happen by sending and receiving messages. Worker tasks have read-write access to local state but they cannot directly access and modify state on a different machine. On the contrary, the shared memory mechanism allows tasks in different machines to communicate by mutating shared state. Systems that employ this mechanism need to account for race conditions and data consistency. In the dataflow model, operators are usually stateless and data flows from one stage of computation to the next. In order to efficiently support graph computations in this model, dataflow systems offer explicit or automatic caching mechanisms. For example, Spark has a *cache()* method, which can be used to cache the graph structure, which is static. In Stratosphere and Flink, the optimizer will detect loop-invariant data and cache them automatically.

3.4.4 Applications

To further assess distributed graph programming model expressiveness and systems usability, we survey the applications which appear in recent distributed graph processing systems papers. We gather and group graph algorithms that are used in the papers introducing the systems of Table 3.2. We choose applications that appear as examples or pseudocode to demonstrate APIs and programming model interfaces, as well as applications used for performance evaluation. We base this study on the assumption that paper and systems authors choose *representative* algorithms to include in their papers, in order to show their systems' applicability and efficiency. Table 3.3 shows the most commonly encountered applications, sorted by appearance frequency. For each application, we also list the programming models in which they are implemented.

Unsurprisingly, we find the PageRank algorithm to be extremely popular. This algorithm appears in 31 out of the 34 examined systems and we encounter implementations in 9 out of the 11 programming abstractions that we present in Section 3.2. Shortest paths calculation and weakly connected components appear in more than one third of the papers. Graph coloring, ALS, k-means clustering, and minimum spanning forest are also commonly used. However, we notice that the majority of applications are only encountered once or twice. This is partially explained by the fact that some of them, like finding cliques, serve the purpose of introducing a specialized programming model, like filter-process. Nevertheless, it appears that the vertex-centric model has been used to implement most of the applications in the table.

Based on the findings of Table 3.3 and our analysis in the rest of this chapter, we can categorize graph analysis applications and provide guidelines regarding suitable programming models as follows.

— **Value-propagation.** Value-propagation algorithms are fixpoint algorithms that iteratively refine the values of the vertices, until they all converge to their final values. Such applications include PageRank, Single-Source Shortest Paths, Weakly Connected Components, and Label Propagation. In these applications, vertex values are typically computed by applying a function on the values of their first-hop neighbors. Value-propagation algorithms can be easily expressed using the vertex-centric, gather-sum-apply, and scatter-gather models.

— **Traversals.** Graph traversal are algorithms that visit the vertices of a graph with the objective to find a particular value or pattern. Breadth-first search and depth-first search are graph traversal algorithms. Even though the vertex-centric model has been used to implement graph traversals, its synchronized nature might incur unnecessary overhead. Subgraph-centric models are preferable in this case, as they allow for asynchronous computation within each subgraph.

— **Ego-network analysis.** Ego-network applications include algorithms that compute personalized metrics using neighborhood information for each vertex. Among the models that we consider in this chapter, the neighborhood-centric model is the most suitable model for such applications.

— **Pattern matching.** Graph pattern matching applications include graph mining algorithms, like identifying cliques and frequent subgraphs. Among the models that we consider in this chapter, the filter-process and the subgraph-centric models can be used to implement such applications.

— **Machine learning.** Several machine learning tasks can be expressed using graph models. Table 3.3 includes machine learning algorithms, such as k-means clustering, Alternate Least Squares, and Markov Clustering. Such applications are most commonly implemented using linear algebra models.

3.5 Discussion

Similarly to how the introduction of the map-reduce programming model simplified large-scale data analysis to a large extend, the invention of the vertex-centric model revolutionized the area of distributed graph processing. *Thinking like a vertex* has proved to be a valuable abstraction that allows writing comprehensive programs for a variety of graph problems. However, despite its wide adoption, researchers and users have also started to identify its shortcomings [160, 187, 176]. The understanding of the abstraction's limits has spawned novel specialized models, such as the neighborhood-centric for ego-network analysis and filter-process for graph mining tasks. At the same time, lower-level abstractions, such as the partition-centric model, are proposed as alternatives for higher performance.

It is clear from our study that no single model is suitable for all classes of graph algorithms. This is an open challenge for researchers and systems designers to either invent a more expressive and flexible programming model or explore the possibility of supporting multiple programming abstractions on top of the same platform. To that end, general-purpose dataflow systems look promising. It has already been shown how the vertex-centric, scatter-gather, and GAS abstractions can be mapped to relational execution plans, and how distributed dataflow frameworks can efficiently support them [72, 42, 82]. Nevertheless, current systems still rely on the user for efficient implementations.

For our work, we use both a general-purpose system and a specialized graph processing system that implements the vertex-centric model. In the next chapter, we present a real-world graph processing use-case, implemented with a general-purpose dataflow system. Such systems are especially useful when graph processing is only one step in a bigger data analysis pipeline. In Chapter 6, we use the Apache Giraph graph processing system.

Chapter 4

Automatic Detection of Web Trackers: a Real-World Application of Distributed Graph Processing

In this chapter, we present a real-world application of graph processing and its distributed implementation. Our goal is twofold. First, we use this use-case to demonstrate the importance of graph analysis in the context of a real business scenario. We show how the graph abstraction can be used to model user-generated data and how mining the relationships in the induced graphs can provide useful answers to interesting questions. Second, we use the implementation of this graph application as a motivating example for the optimization techniques that we propose in Chapters 5 and 6.

4.1 Advertisements and Tracking on the Web

The massive growth of the web has been funded almost entirely via advertisements shown to users. Web ads have proven superior to traditional advertisements for several reasons, the most prominent being the ability to show personally relevant ads. While the content of the web page the ad is being served on can help provide hints as to relevance, web advertisement agencies also rely on mechanisms to *uniquely identify* and *track* user behavior over time. Known as *trackers*, these systems are able to uniquely identify a user via a variety of methods (e.g., persistent cookies, browser fingerprinting, etc.) and over time can build up enough information about a user to serve extremely targeted ads.

While ad agencies' use of trackers has enabled services to provide access to users free of charge, there is also a certain degree of "creepiness" in the way the current ecosystem works that has also been highlighted in the US Congress [156]. Even worse, recent work [71] has shown that trackers can be easily exploited by government agencies to spy on people.

Privacy concerns have led to the creation of client side applications that block trackers and ads, for example AdBlock, which uses crowdsourcing to build what amounts to black lists of URLs called EasyPrivacy. Since it is crowdsourced, by definition EasyPrivacy requires human effort to identify trackers. Further, the EasyPrivacy list is mostly opaque:

there is no straight forward way to understand why a tracker was added to the list or to get a sense as to how trackers work on an individual or group basis. In the research community, several strategies for detecting and defending against trackers have been introduced [158, 110], in general focusing on understanding the methods that trackers use and techniques for obfuscating a user's browsing behavior.

Targeted advertising and personalization services on the web rely heavily on monitoring the users' browsing behavior. This is often realized using code embedded in web pages that has the capability to log a user's visit, set cookies in their browser, and share this information with a tracker. Trackers enable targeted advertising and personalization services by monitoring user behavior on the web. To understand web tracking, let us consider what happens in the browser when a user visits a URL. First, the browser issues an HTTP request to the site to fetch the contents of the web page. The response contains the page resources, including HTML, and references to embedded objects like images and scripts. These references might then instruct the browser to make additional HTTP requests (e.g., for the image itself) until the page is fully loaded. Embedded objects can be hosted on different servers than the page content itself, in which case they are referred to as *third-party* objects. A fraction of these third-party objects open connections to trackers, e.g., the popular Facebook "like" button, at which point the users' online whereabouts are logged for targeting/personalization purposes.

In this chapter, we use a large-scale dataset of real users extracted from the logs of an explicit mobile proxy to characterize web trackers. We use graph analysis techniques to explore the network level behavior of trackers, such as latency and data transfer sizes. We quantify user exposure to trackers and try to understand what types of browsing behavior are more likely to be exposed to tracking. We then explore trackers' positions within a graph induced from user browsing behavior. To achieve that, we create a bipartite graph of user visited URLs and the 3rd party requests generated from those URLs. We create the graph projection and using a community detection algorithm, we discover that trackers form distinct and tightly coupled clusters. Finally, we show that using the induced graph structure, we can build a scalable tracker classification pipeline, which operates with high precision and very low false positive rate.

4.2 The Referer-Hosts and the Hosts-Projection Graphs

In this section, we introduce the dataset and the graph models that we create to help us solve the tracker classification problem. We also provide the necessary background on graph theory concepts and techniques that we use later in this chapter.

4.2.1 Dataset

Our dataset is derived from 6 months (November 2014 - April 2015) of traffic logs from an explicit web proxy. The proxy is operated by a major telecom located in a large European country. Our data is delivered to us in the form of augmented Apache logs. The logs include fields to identify the user that made the access, the URL that was requested, headers, performance information like latency and bytes delivered. We call this dataset the *proxy log*, and in total it represents 80 million accesses to 2 million individual sites. In the following section, we describe how we use the proxy log to model web tracking as a graph problem. We label URLs in our dataset as *tracker* or *other* based on ground truth derived from the EasyPrivacy list for AdBlock [6].

4.2.2 Modeling Tracking Relationships as Graphs

A *bipartite graph* is a graph with two different modes (or classes) of vertices, where edges are only allowed between vertices belonging to different modes. The interactions between explicit user requests and background requests, both page content and third-party objects like web tracking services, can be naturally modeled as a bipartite graph. The first mode of vertices in the graph are URLs that the user intentionally visits, while the second mode are URLs for objects that are embedded in the visited page.

More precisely, we represent the URLs that a browser accesses as a bipartite graph $G = (U, V, E)$, where U are the URLs that the user explicitly visits, V are the URLs that are embedded within those pages, and E is the set of edges connecting vertices in U (explicitly visited URLs) to vertices in V (URLs embedded within visited pages). We call vertices in U *referers*, vertices in V *hosts*, and G the *referer-hosts graph*.

In graph analysis, *communities* are groups of vertices that are well-connected internally, and sparsely connected with other groups of vertices. An example of graph communities is graphically shown in Figure 4.1. Vertices belonging to the same community are more likely to be similar with respect to connectivity and network position than vertices belonging to different communities. V contains both regular embedded objects and third-party objects potentially associated with trackers. We expect regular embedded objects to only appear on the hosting web page, while tracker objects need to appear on as many web pages as possible to enable successful tracking of users across websites. This implies that: 1) tracker vertices in V should be linked to many different vertices in U and 2) tracker vertices are members of well-defined communities in G.

Unfortunately, working with communities in bipartite graphs can be tricky. For example, the relationships between vertices in the same mode are only *inferred* from relationships that pass *through* vertices in the second mode, which can lead to unexpected results from standard community detection algorithms run on a raw bipartite graph. This is especially a problem when the community structures of the two modes are different, as we might expect in our case [130]. To avoid this problem, it is typical to extract and analyze 1-mode projections of bipartite graphs.

Assuming that users do not intentionally visit tracker sites, U should not contain tracker URLs which are instead contained in V. Accordingly, we can project the bipartite graph into a 1-mode graph that only contains the vertices in V, by creating the *hosts-projection graph* G'. In G', we create an edge between any two vertices in V that share a common neighbor in G. I.e., if two vertices, v and v' from V both share an edge with a vertex u from U, then there is an edge $e = (v, v')$ in G'. Fig. 4.2 illustrates this transformation. This way, G' preserves much of the original graph's structural information and captures implicit connections between trackers through other sites.

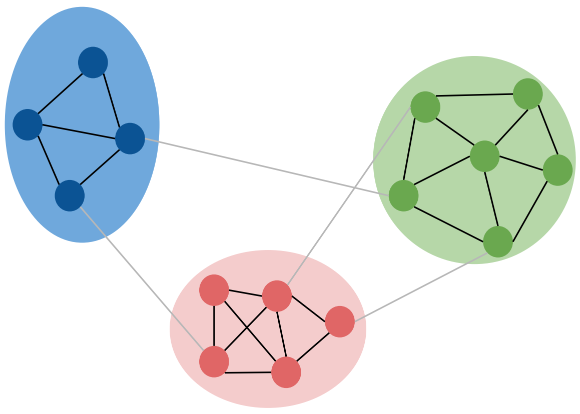


Figure 4.1 – Example of graph communities. Vertices belonging to the same community are well-connected with each other, while vertices belonging to different communities are sparsely connected.

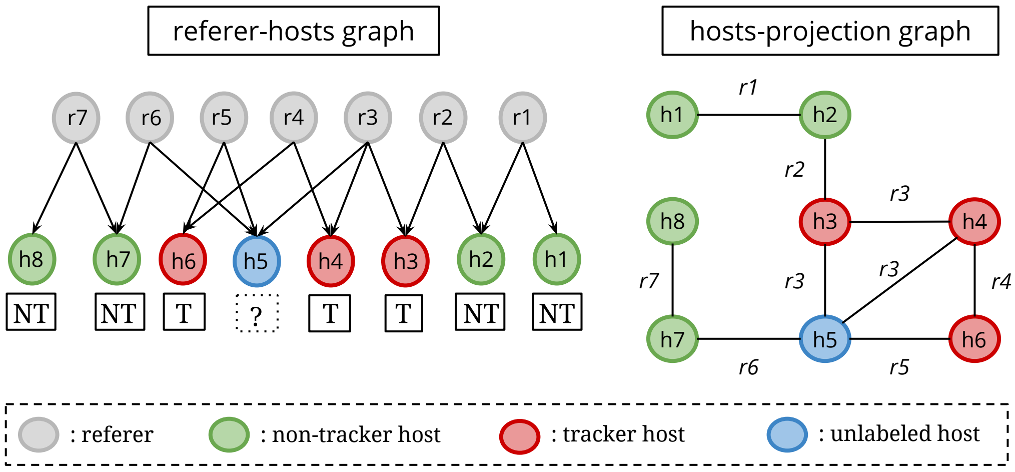


Figure 4.2 – Example of the hosts-projection graph transformation. Vertices prefixed with r are the pages the user explicitly visited while those prefixed with h were embedded within the r vertex they have an edge with. Note that additional information associated with the vertex (e.g., tracker/non-tracker/unknown label) is not affected by the transformation.

4.2.3 Graph Weighting Schemes

The referer-hosts graph and the hosts-projection graph let us encode the relationships between referer-host and host-host pairs respectively. We can enrich these graphs with edge weights to encode the strength of these relationships. The proxy log contains information that can help us quantify the relationship strengths. For example, if a request from referer A to host B appears more often in the proxy log than a request from referer A to host C, it might be the case that the relationship between A and B is more important than the relationship between A and C. In this section, we describe four weighting schemes for the referer-hosts and the hosts-projection graphs.

1. **Common neighbors**. The first weighting scheme is based on the intuition that if two hosts are pointed to by many different referers, then they should probably be similar to each other. We assign weights in the projection graph to represent the number of common neighbors that the connected hosts have in the referer-hosts graph. We use the logs to create the bipartite graph by adding one edge between every *distinct* pair of referer and host URLs. Then, we create the projection graph by connecting two hosts with an edge and add a weight equal to the number of common neighbors that the hosts have in the bipartite graph.
2. **Request frequency**. In the second weighting scheme, we want to encode the importance of a request occurrence frequency in the proxy log. For each request in the proxy log, we extract the referer and host URLs. If no edge between the two exists in the bipartite graph, we add one with initial weight equal to one. If an edge already exists between the URLs, we increment its weight. In a variation of this scheme, we can normalize the weights as follows. We divide each edge weight by the total number of occurrences of its endpoints' pair in the logs. This way, we can express the frequency of a connection between a referer and a host as compared to the rest of the connections of this referer. Figure 4.3 illustrates this idea. Without normalization, edges $(R1, H1)$ and $(R2, H1)$ are assigned the same weight. Since $R1$ appears in the proxy log ten times and in five of them it is connected to $H1$, the normalized weight reflects that $R1$ connects to $H1$ *half* of the times. Similarly, the normalized weight reflects that $R2$ connects to $H1$ *every* time.
3. **Attribute similarity**. The proxy log contains performance information about each request, including the number of bytes exchanged and the request latency. We create a vector of attributes for each host and we use it to weight the projection graph edges by computing the euclidean distance between the vectors of the edge endpoints. The lower the distance, the more similar two hosts are assumed to be.
4. **Attribute similarity and common neighbors**. This weighting scheme combines the attribute similarity weights with the common neighbor weights. We create the projection graph with edge weights computes as the number of common neighbors between the edge end-points multiplied by the inverse of the euclidean distance of the attribute vectors. In this case, the higher the weight, the more similar we assume the hosts to be.

4.3 Tracker Behavior Analysis

In this section we perform a high-level analysis on tracker behavior by analyzing the referer-hosts graph and the hosts-projection graph. We are especially interested in discovering whether trackers have different properties than normal pages in these graphs and whether their network position can be used as a distinguishing feature.

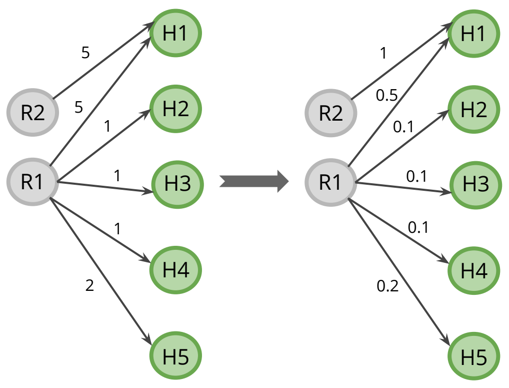


Figure 4.3 – Example of the request frequency weighting scheme. On the left graph, weights correspond to the number of occurrences of each referer-hosts pair in the proxy log. On the right graph, weights are normalized to reflect the occurrence probability of the connections.

4.3.1 Trackers' Position in the Referer-Hosts Graph

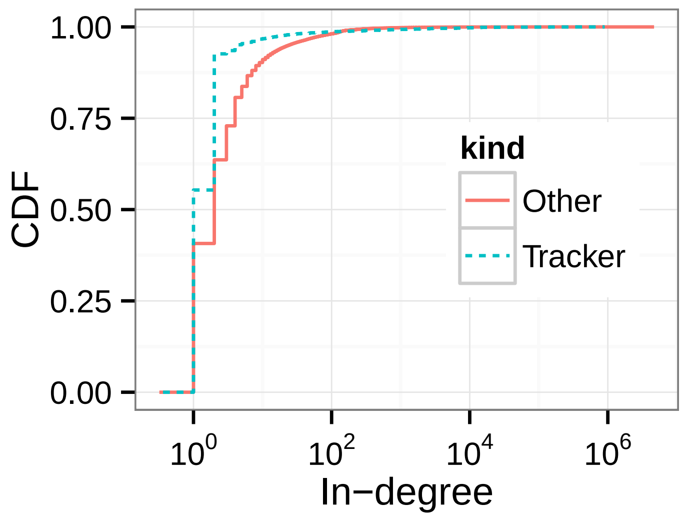


Figure 4.4 – CDF of in-degree per host for trackers and others.

The first question we examine here is how well connected trackers are. Although trackers are essentially required to appear on many different pages to collect meaningful data, we are interested in quantifying this. We begin by plotting the in-degree (i.e., the number of unique source URLs they are requested from) of tracker and other URLs in Figure 4.4.

Interestingly, we find that trackers tend to have a slightly lower in-degree than other URLs. When looking at things a bit closer, we discovered that this is likely due to the use of *user specific tracking URLs*, primarily those from Google. These URLs are in the form, e.g., `unique-hash.metrics.gstatic.com`. Therefore, straight-forward outlier detection techniques on the in-degree would miss them.

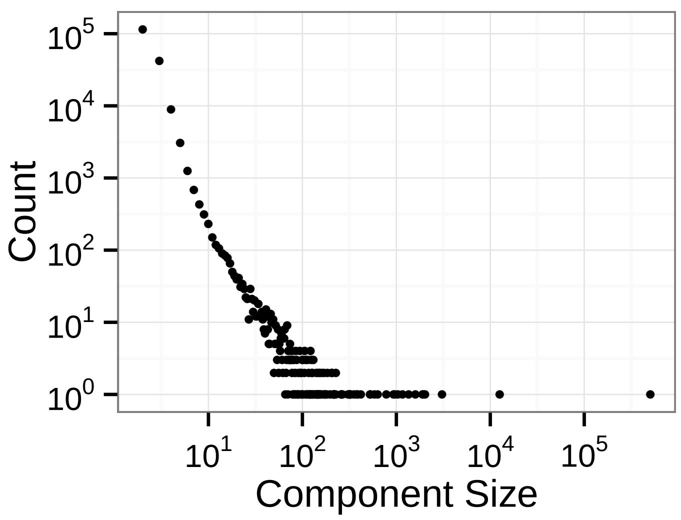


Figure 4.5 – Connected Components Size Distribution in the Referer-Hosts Graph .

Next, to see how well-connected trackers are *to each other* we extract connected components and plot the distribution of their sizes in Figure 4.5. As expected, there are many 2-node components, however, we find that over 94% of the total amount of trackers belong to the largest connected component (LCC). For the remainder of this chapter, we focus on this largest connected component, consisting of of 500,000 vertices and 1 million edges.

4.3.2 Trackers' Position in the Hosts-Projection Graph

We create the hosts-projection graph from the largest connected component in the referer-hosts graph. The projection has 80,000 vertices and 43 million edges.

Figure 4.6 shows the degree distribution of trackers and other hosts in the hosts-projection graph. As opposed to the referer-hosts case, here we observe a clear difference between the degree distribution of trackers and other pages. A higher degree might imply that trackers are more important and central nodes in the projected network than other pages.

Next, we examine trackers' neighborhoods more closely. Figure 4.7 shows the ratio of a nodes's neighbors that are trackers. We observe that the vast majority of trackers' neighbors are other trackers. To further investigate how well-connected trackers are among them, we plot the ratio of a node's neighbors that are trackers over the total number of trackers in the dataset in Figure 4.8. From this figure, we see that trackers appear to be direct neighbors

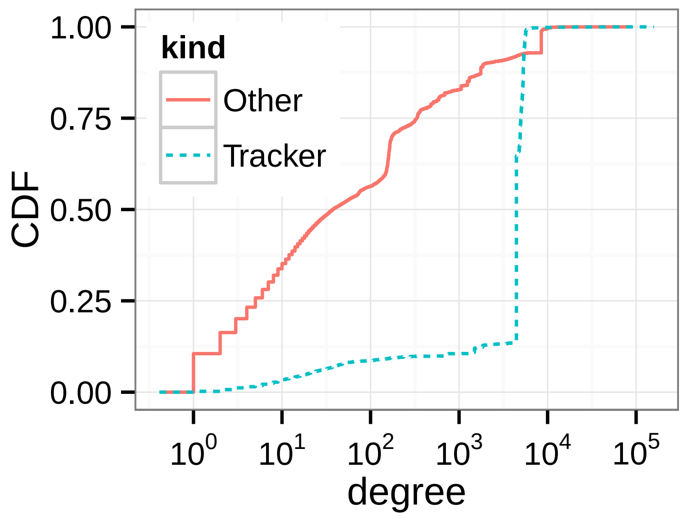


Figure 4.6 – CDF of degrees in the hosts-projection graph for trackers and others.

with most of the existing trackers in the whole network. What this essentially means is that trackers tend to be pointed to by the same pages. Something that makes intuitive sense, since publishers tend to add multiple trackers on their web sites for better ad targetting.

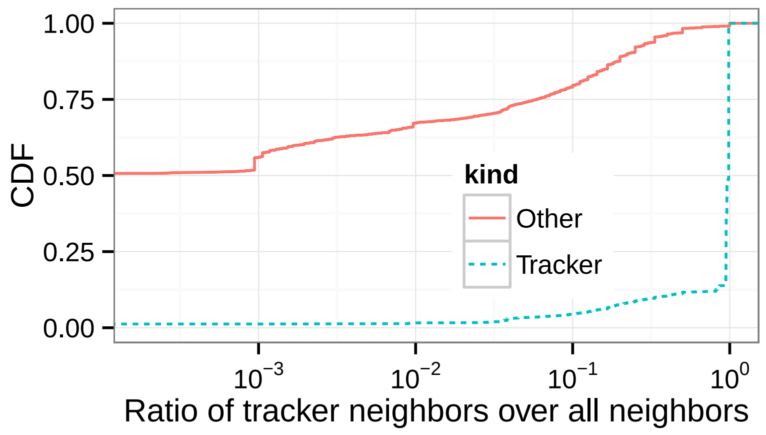


Figure 4.7 – Ratio of the projection graph nodes' neighbors that are trackers.

Figure 4.9 shows co-occurrence ratios for different types of pages. From this figure, we see that not only trackers are mostly connected to other trackers, but also that edges

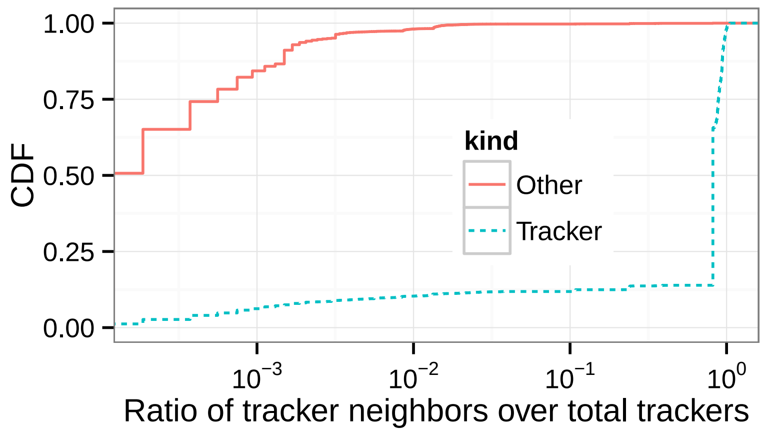


Figure 4.8 – Ratio of a node's tracker neighbors over the total number of trackers in the dataset.

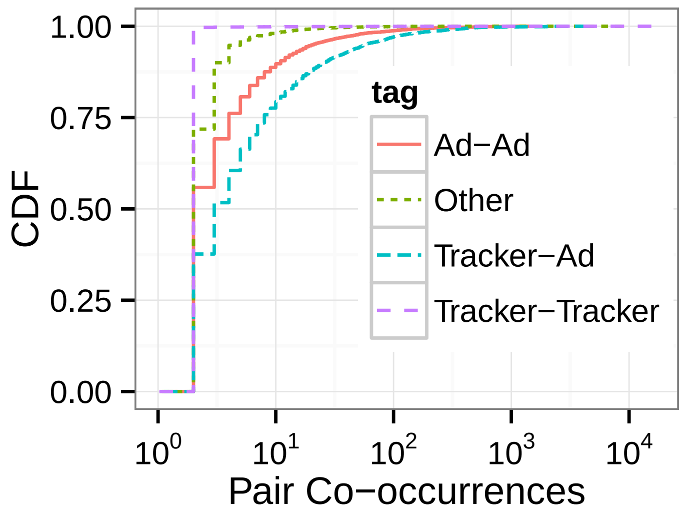


Figure 4.9 – CDF of co-occurrences for different types of pages.

between trackers are the most common, in the hosts-projection graph [1].

All of these findings suggest that trackers form a dense community in the hosts-projection graph. Figure 4.10 shows a visualization of the host-projection graph (from April's logs) focused on trackers' positions. The visualized network contains only edges with at least one tracker endpoint. We observe that the majority of low-degree trackers form a very

1. To increase readability of this figure, we have removed edges that only appear once in the dataset

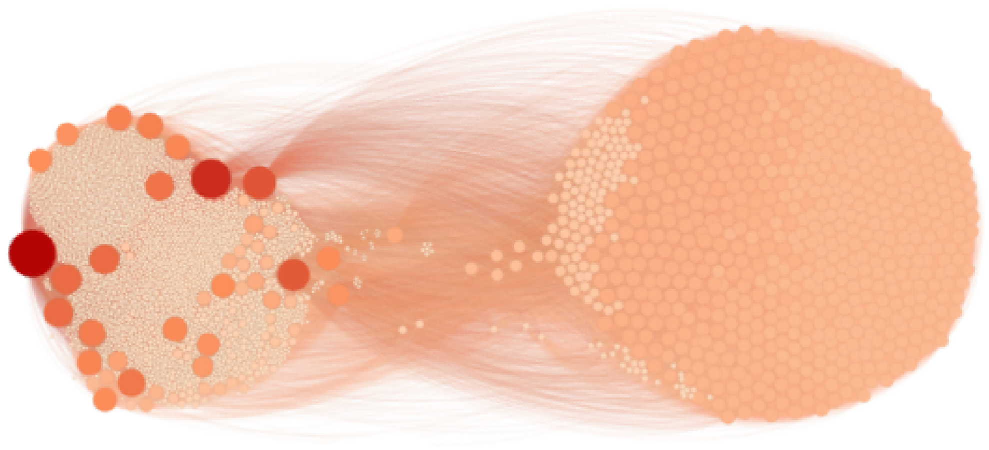


Figure 4.10 – Visualization of the Tracker network structure of the hosts-projection graph, created using the logs of April. The network contains 60k nodes and 340k edges. Nodes are ranked by degree, so that a darker color denotes a higher degree. The community on the right contains tracker nodes and ad server nodes, where ad servers can be seen as having a slightly lighter color and being mostly clustered on the left edge of the community. The left cluster consists of normal webpages and a few popular trackers, which can be distinguished by their larger size and darker color.

dense and easily identifiable community indeed, seen on the right side of the figure. On the other hand, there exist a few *popular* trackers, which are connected to the majority of normal URLs and are also very well-connected among each other.

4.4 Classification Methods

Our main goal is to exploit the hosts-projection graph structure to automatically classify trackers. Our findings so far suggest that trackers form a well-connected cluster and are mostly connected to other trackers. We leverage these findings and confirm that even a simple assessment of the nodes' neighbors in the hosts-projection graph can yield good classification results. We further evaluate a community detection algorithm and show that it succeeds in identifying clusters of trackers.

We evaluate two methods for automatic web tracker classification. As a baseline method, we use a simple structural assessment of the hosts-projection graph to identify trackers. Next, we use a slightly modified version of the well-know Label Propagation algorithm [151] to identify clusters in the hosts-projection graph. We use these methods to identify communities of similar web pages and classify unlabeled hosts as trackers or non-trackers, based on their community membership.

4.4.1 Neighborhood-Based Classification

This is a simple rule-based classification method which analyzes the first-hop neighborhoods of each unlabeled node in the hosts-projection graph. For each unlabeled node, we count the number of trackers among its immediate neighbors and make a classification decision based on a configurable threshold. If the percentage of tracker neighbors is above the threshold, then the node is labeled as a tracker. The intuition behind this method is that we expect most trackers to be mostly connected to other trackers in the host-projection graph. Previous work [110, 73, 80] indicates that web trackers form networks with small-world network characteristics, where small groups of entities are highly connected among them and that tracker networks also tend to cluster based on their geographical location.

4.4.2 Community Detection via Label Propagation

Label Propagation is a scalable iterative algorithm for community detection. It consists of an iterative process, which exploits the network structure to propagate labels and identify densely connected groups of nodes. Initially, nodes are assigned unique labels. Then, the algorithm proceeds in iterations, where nodes exchange labels with their neighbors. In each iteration, a node receives a list of the all the labels of its immediate neighbors and adopts the most frequent label among them. The algorithm converges when no label changes during an iteration. An example is shown in Figure 4.11. In our implementation of the label propagation algorithm, a node considers its own label as well when computing the most frequent label. In order to break ties in the case that two labels have the same frequency, we always choose the label with the highest identifier.

We use this algorithm on the hosts-projection graph as follows. Initially, we assign a unique numeric label to each host in the graph. Upon convergence, we consider that nodes with the same label belong to the same cluster. Then, we use the EasyPrivacy list to identify and tag known trackers inside the clusters. White-listed nodes are tagged as non-trackers. Finally, we assign a tag to each cluster, by choosing the most popular tag among its cluster members. We classify unlabeled nodes by assigning them the tag of the cluster in which they belong. A threshold-based approach for tagging the clusters could be used here as well.

4.4.3 Weighted Label Propagation

The label propagation algorithm can be extended to leverage the edge weights of the projection graph. Using the weighting schemes of Section 4.2.3, we modify the algorithm as follow. During an iteration, nodes send messages to their to their neighbors, containing both the node's label and the edge weight connecting the sender node to the destination node. On the receiving side, a node adds up the received weights for each label and adopts the label with the highest sum.

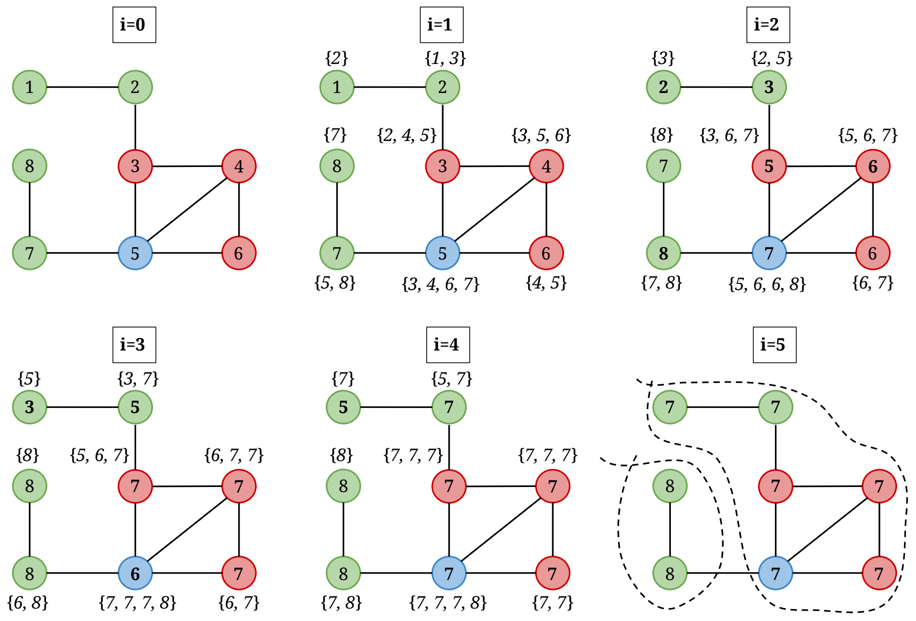


Figure 4.11 – An example of the label propagation algorithm for tracker classification. Green nodes are known to be legitimate pages and red nodes are known to be tracker URLs. The blue node is unlabeled. After the algorithm has converged (i=5), there exist two clusters in the graph. The cluster with label=8 contains two legitimate URLs and is labeled as a non-tracker cluster. The cluster with label=7 contains three tracker URLs and two legitimate URLs. Since the majority of pages are trackers, this cluster is labeled as a tracker cluster and the blue node is classified as a tracker.

4.5 Data processing pipeline implementation

In this section, we describe the system we built to aggregate the data from our logs, construct and transform graphs, and classify URLs as trackers or non-trackers. We provide details on the data processing pipeline and the implementations of the different graph algorithms that we used for classification.

We built an end-to-end data processing pipeline with Apache Flink [1] and its graph processing API, Gelly, to enable our analysis. Flink is an efficient and scalable open-source system for distributed large-scale data processing. It supports both batch and streaming analytics and contains libraries for graph processing and machine learning tasks. Flink can execute applications in single-node mode or cluster mode and scales to very large datasets. It is ideal for our use-case for the following reasons. First, it can easily handle the large datasets we have to process. Second, Flink has native operators which are used to efficiently run iterative graph processing tasks. Finally, Flink can be used to easily build long data processing pipelines that seamlessly combine ETL-style (Extract-Transform-Load) and graph analytics in the same application.

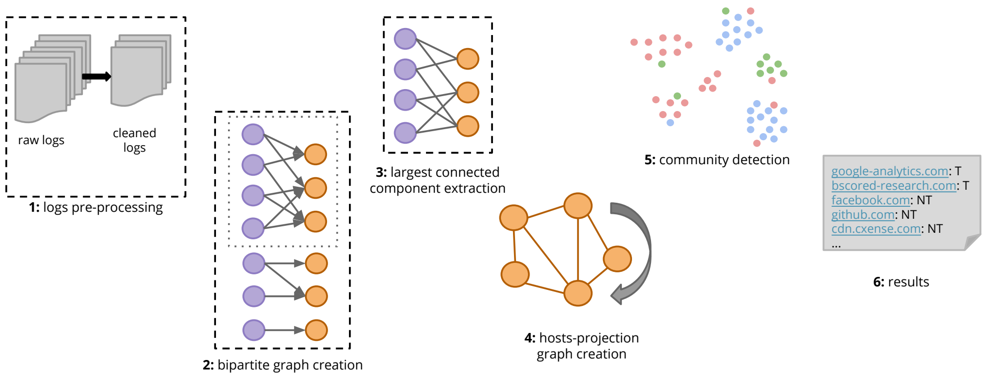


Figure 4.12 – Classification data pipeline

The implemented pipeline is shown in Figure 4.12 and consists of the following steps:

1. *Logs pre-processing*. We process the raw logs of user traffic to filter out bad records and extract useful information. For example, we filter out log records that are missing referer of host information. Then, we parse valid records to extract the fields that we use in our analysis: referer URL, host URL, timestamp, and tag.

2. *Bipartite graph creation*. We use Flink's graph API, Gelly, to parse the extracted fields as edges and create a bipartite graph. Since the same referer-host pair may appear several times, we remove duplicate edges. Finally, we ensure that the result is a bipartite graph and no dual-role nodes exist.

   Upon receiving new logs, we incrementally update the existing bipartite graph as follows. We process the new logs to extract $< refererURL, hostURL >$ pairs and create a new set of edges. We then check whether the new edges contain connections that we have encountered in the previous months. The edges formed by these connections are already present in the bipartite graph and we do not add them again, while we form new edges for connections that we encounter for the first time. Finally, we tag previously identified trackers and non-trackers. The result is a new bipartite graph, where some host nodes (URLs) are not tagged. Figure 4.13 shows the process of merging the logs of a new month to the existing state of the bipartite graph.

3. *Largest Connected Component Extraction*. We use Gelly's scatter-gather implementation of the weakly connected components algorithm to find the connected components of our bipartite graph. We then use Flink's DataSet API to extract the largest component.

4. *Hosts projection graph creation*. We create the hosts-projection graph of the largest connected component of the bipartite graph as follows. We group by referer and create one edge for each pair of its neighbors. This way, if two hosts are referenced by the same referer, they are neighbors in the projection graph.

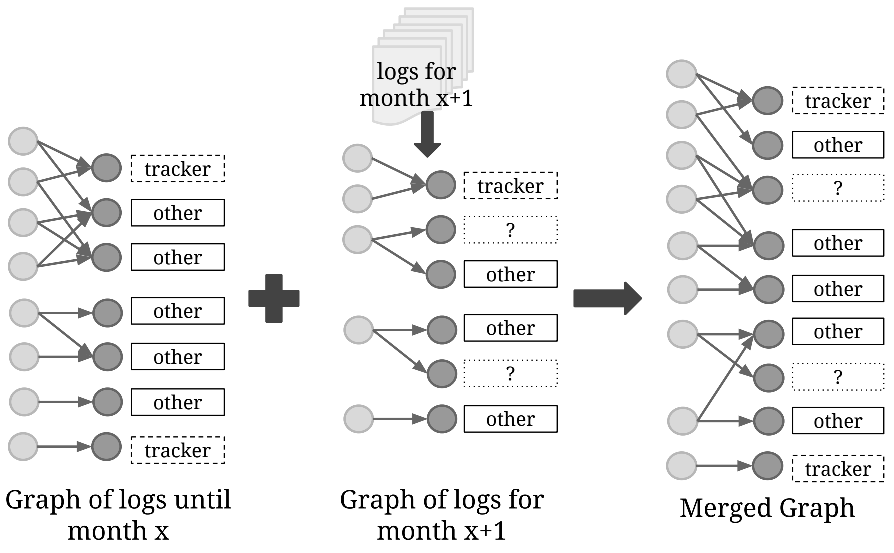


Figure 4.13 – Merging new logs into the current bipartite graph.

5. *Community Detection*. The neighborhood-based classification is implemented using Gelly's neighborhood-based aggregation methods. We implement the Label Propagation algorithm using a custom iterative scatter-gather program. After convergence, we use Flink's DataSet API to extract and tag the clusters and to classify unlabeled nodes.
6. *Results post-processing*. During the post-processing, each unlabeled host is assigned a *tracker* or *non-tracker* label. For the neighborhood method, the label is the most popular label among its neighbors, where popularity is calculated based on a specified threshold. For the community detection method, the label chosen is the cluster label in which the host belongs.

4.6 Classification Evaluation

In this section, we summarize the results of evaluating the proposed classification methods. We evaluate the methods, using three subsets of our dataset. For every subset, we use all the hosts that appear in the last month as the test set and all the previous months as the training set. Thus, we use the logs from November up to January in order to classify hosts seen in February logs, November up to February logs to classify hosts first encountered in March, and logs from November up to March to classify hosts seen in April logs. Note that we remove from the test set all hosts that we have seen before, i.e. nodes for which we already know whether they are trackers or not. The number of test records and new trackers per month are shown in Table 4.1. We use the classification methods to tag each of the untagged nodes as *tracker* or *non-tracker* and measure precision, accuracy, false positive rate (FPR) and recall for each method. Finally, we assess classification stability, by randomly choosing test sets out of the complete dataset.

	Test Records in LCC	**Trackers in LCC**	**Total New Trackers**
Feb	13685	760	811
March	18313	740	774
April	40465	747	792

Table 4.1 – New Trackers per Month

4.6.1 Neighborhood-Based Classification

We evaluate the neighborhood-based method, which utilizes only the first-hop neighborhood information to tag a test item. The results for the months of February, March and April are shown in Figure 4.14. We vary tag popularity, in the following ways:

— the most common tag among all tagged neighbors (Threshold: *0.00*).

— the tag appears on at least $y\%$ of the node's neighbors, where y=30, 40, 50, 60, 70, 80, 85 and 90. This way, a node is classified as *tracker* if at least $y\%$ of its neighbors are tagged as trackers.

In all cases, we achieve a classification precision that varies from 64% up to 83%. We observe that precision increases for higher thresholds: the more tracker neighbors a node has, the higher the probability that it is a tracker itself. Similarly, false positive rate (FPR) and accuracy, both improve for higher thresholds. FPR remains under 2% in all cases and accuracy above 97%. On the other hand, recall decreases as we increase the threshold, which means that we might miss a few trackers, but it is above 90% in all cases.

4.6.2 Label Propagation

The results for the Label Propagation method are shown in Table 4.2. To assess classification stability, we evaluate the classification using random sets of test records of varying sizes. Instead of selecting the test set based on the timestamp of the log record, we create test sets from the complete graph, by randomly choosing log records and marking them as "untagged". We run this experiment for test sets of 5%, 10% , 20% and 30% of the complete dataset and repeat it 3 times.

By exploring further than the first-hop neighborhood of nodes, this method can successfully locate the trackers community and classify test items with extremely high precision, up to 94%. As compared to the simple neighborhood inspection, this method achieves higher accuracy and higher recall, while further lowering the false positive rate. Note that this result does not come at a cost of performance, since the algorithm converged in less than 10 iterations, for all the test sets used. Additionally, this method has the advantage that we do not need to set a threshold.

We also evaluated label propagation on the weighted projection graph, using the weighting schemes introduced in Section 4.2.3 to capture structural and operational similarity between nodes. The results for the attribute similarity weighting scheme are shown in Fig-

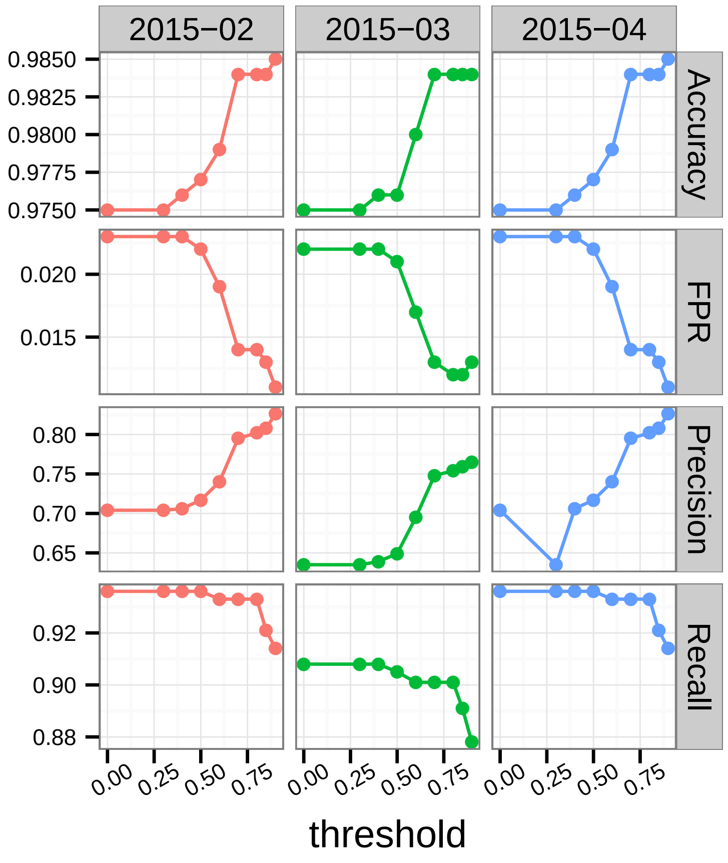


Figure 4.14 – Classifier performance for the neighbor count method.

ure 4.15. We found that community detection method on the weighted graph had similar classification performance to the one of the unweighted graph, but, in some cases, it could lower FPR further. The feature that appeared to improve classification performance most was the size of bytes received. However, the classification results were not stable, as the method is not guaranteed to converge.

4.7 Conclusion

In this chapter, we presented a real-world graph analysis use-case. We characterized tracker behavior using a large-scale dataset of logs from an explicit proxy. We transformed user requests into a bipartite referer-hosts graph where vertices in the first node represent URLs the user visited and vertices in the second node represent requests for objects embedded in those pages. From this graph we discovered that 94% of trackers are in the largest connected component. To further focus in on how trackers are related to each other, we collapsed the bipartite referer-hosts graph into a 1-mode hosts-projection graph. From the

		precision	FPR	accuracy	recall
Monthly Test Sets	**Feb**	0.934	0.004	0.993	0.932
	March	0.946	0.002	0.994	0.9
	April	0.922	0.001	0.997	0.872
Random Test Sets	**5%**	0.923	0.004	0.994	0.958
	10%	0.934	0.004	0.993	0.941
	20%	0.941	0.003	0.994	0.948
	30%	0.939	0.003	0.994	0.951

Table 4.2 – Label Propagation Classification Results

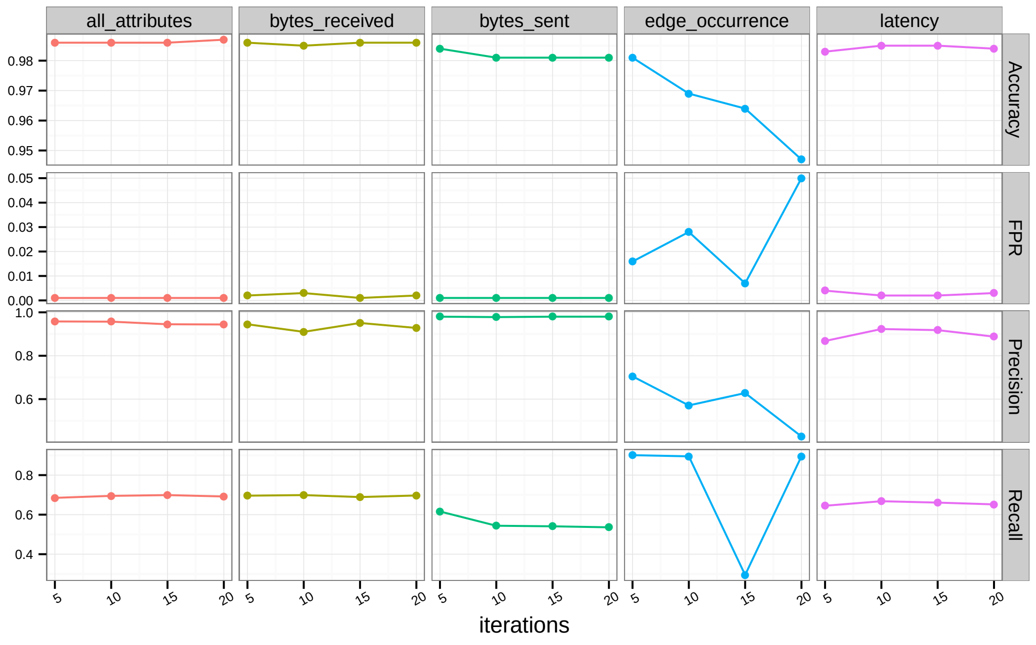


Figure 4.15 – Classifier performance for weighted Label Propagation.

hosts-projection graph we observed an extremely high degree of clustering: trackers tend to co-appear on pages together in a very distinct manner. Next, using the insight from our characterization, we built a classification pipeline which aggregates our proxy logs, processes them into the hosts-referer graph, transforms the hosts-referer graph into the hosts-projection graph, and classifies using two algorithms. From the classification results, we discover that the structure of the induced graph alone resulted in high accuracy.

We have shown a representative graph processing pipeline that consists of a pre-processing step for graph creation, a graph analysis step, and a post-processing step to retrieve the final results. Building and studying the properties of this data processing pipeline has inspired the optimization methods that we present in the following chapters. With respect to the research objectives and goals that we set in the introduction of this thesis, we are going to design and implement performance optimizations for the graph analysis step of this use-case. Specifically, in Chapter 5 we show how we can reduce the amount of computations required in fixpoint graph algorithms, like the label propagation algorithm, and in Chapter 6 we show how we can compress weighted graphs, so that we can reduce the amount of data that we need to process.

Chapter 5

Asymmetrical Convergence in Large-Scale Graph Analysis

As we saw in Chapter 3, fixpoint algorithms like PageRank, Connected Components, and Shortest Paths, are commonly used as representative applications for evaluating graph processing systems. In a recent benchmark for parallel and distributed graph processing systems [91], four out of the six applications used are fixpoint algorithms. One of them, Label Propagation, also lies in the core of the web tracker classification use-case that we described in Chapter 4. Fixpoint algorithms iteratively refine the values of a set of parameters, until they all converge to their final values. Each parameter is associated with a set of *dependency* parameters. Parameter values are computed by applying a *step function* on the parameter's dependency set. In the context of graph processing, fixpoint algorithms can be mapped to the BSP model [177]. The parameters are represented by the graph's vertices and dependencies are represented by edges. In each iteration, vertices compute their values by applying a user-defined function on the values of their incoming neighbors and their current state. The algorithm terminates when no vertex changes its value or when some other convergence criterion is met.

Many iterative refinement graph algorithms expose *non-uniform* behavior, where different vertices require a different number of iterative steps to converge. Consequently, some vertices converge faster than others and do not need to participate in the computation in the following iterations [72, 122, 125]. If this asymmetrical convergence behavior is not accounted for, redundant computations will be possibly performed. Ideally, we would like to detect this phenomenon and stop the computation early for the inactive parts. This would allow the system to avoid redundant computation and communication. Applying this optimization requires detecting inactive vertices and identifying the parts for which computation can halt. However, we must examine how, halting computation for some vertices, could potentially affect the *correctness* of the computation for the rest of them. Even if inactive parts can be accurately identified, is it always possible to halt computation for these parts and still obtain correct results?

To clarify these issues, let us consider the vertex-centric fixpoint version of the single-source shortest paths (SSSP) algorithm. Consider the graph of Figure 5.1, where S is

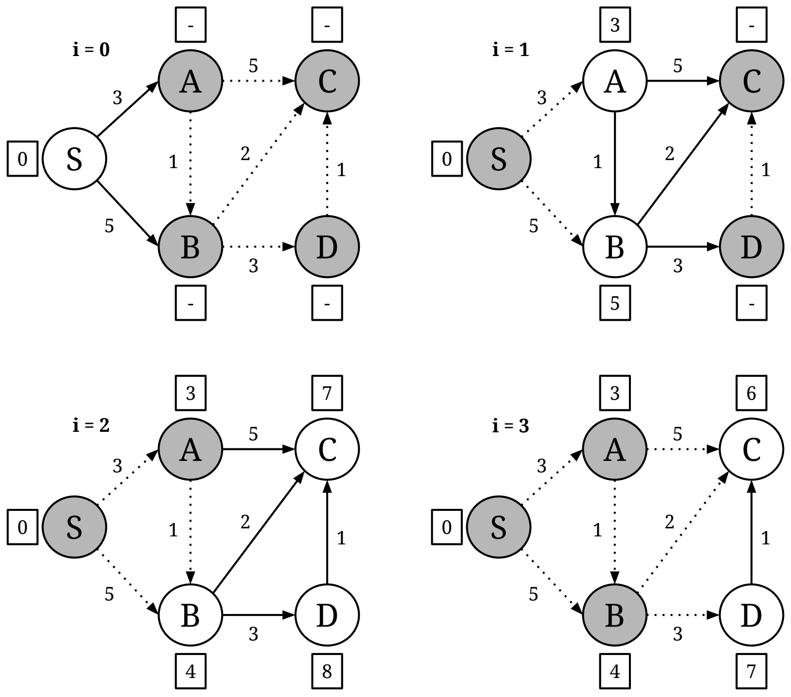


Figure 5.1 – An example execution of SSSP. Grey nodes have not changed their value since the previous iteration and they do not need to participate in the computation. Messages are propagated only along edges depicted with solid lines.

the source, the weights on the edges represent distances, i is the iteration counter and the values in the boxes show the distance computed for each vertex at the current iteration. In this example, the algorithm is refining the distances of vertices from the source vertex S. In each iteration, a vertex receives new candidate distances from its in-neighbors, selects the minimum of these candidates and its current value, adopts this minimum as the new value, and propagates it to its out-neighbors. For this algorithm, it is trivial to observe that if a vertex does not change its distance during an iteration, it does not have to propagate its value to its out-neighbors; the neighbors have already seen this distance in the previous iteration. Vertices whose value does not change between two consecutive iterations are shown in gray in Figure 5.1. If we halt computation for these vertices, so that values propagate only along edges depicted with solid lines, the final result will still be correct. For example, in iteration 3, vertex C does not need to receive the value of A again. Any future distances that C will compute can only be equal or lower its current distance.

Even though applying this optimization to SSSP may seem obvious, it cannot be easily generalized for all similar algorithms. Let us consider computing Label Propagation [150] instead. In this algorithm, each vertex is again initialized with a label, or color as shown in Figure 5.2. Vertices iteratively exchange labels with their neighbors, with the difference that they do not pick the minimum label, but the one that most of its neighbors currently have. Execution A in Figure 5.2 shows what happens if we follow the logic of the shortest

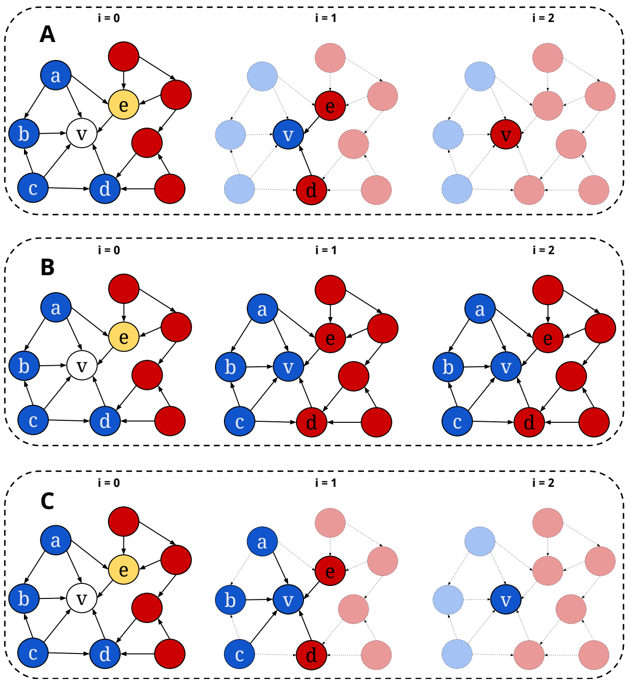


Figure 5.2 – Three possible example executions of the Label Propagation algorithm. In execution A, vertices that do not change value get de-activated and that causes vertex v to decide on an incorrect label. In execution B, all vertices remain active and compute the correct result, but redundant computation and communication is performed. In execution C, only vertices whose values are essential for correct computation are active.

paths example and stop propagating labels from vertices that do not change their values. De-activated vertices are shown in faded colors. All nodes are initially active and send their labels to their neighbors. In iteration 1, node v has received a majority of blue labels and joins the blue community. In iteration 2, v has only received labels from active nodes d and e and incorrectly decides to join the red community, even though the majority of its incoming neighbors have blue labels. A straight-forward way to avoid this situation is to use a *bulk* iteration technique [72], as shown in execution B. In this case all vertices are always active. However, this approach leads to redundant computation and communication. As shown in execution C, a large part of the graph can be safely de-activated without affecting the correctness of the result.

In this chapter, we present an overview of general optimizations for value propagation graph algorithms, in the presence of asymmetrical behavior in computations. We study the characteristics of four iterative techniques and we describe what these characteristics mean and how they can be safely exploited, in order to derive optimized algorithms. More importantly, we give the necessary conditions under which it is safe to apply each of the described optimizations, by exploiting problem-specific properties. We use general-purpose dataflow operators to create template optimized execution plans, which can detect converged parts and avoid redundant computations, while providing functionality equivalent to Pregel and

Table 5.1 – Graph data sets that we use for observing asymmetrical convergence.

Graph	**|V|**	**|E|**	**size**	**avg. degree**	**clust. coeff.**	**diameter**
Twitter [111]	41.6M	1.5B	24GB	70.506	0.0846%	23
Wikipedia [15]	12.2M	378M	6.2GB	62.241	1.63%	11
Livejournal [118]	4.8M	68M	1GB	28.251	11.8%	20
Youtube [190]	1.1M	2.9M	38MB	5.265	0.622%	24

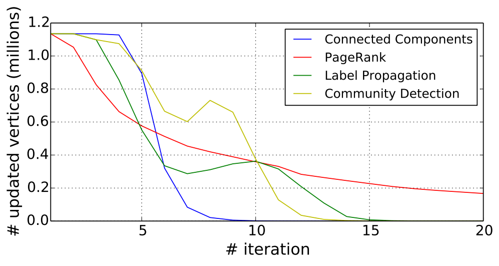
GraphLab. We propose an expressive, high-level fixpoint API for writing graph iterative fixpoint applications in a declarative way. We build this on top of a flexible runtime capable of identifying and avoiding redundant computations. We evaluate the optimizations using four iterative algorithms and we present extensive experiments using real-world datasets of varying sizes and characteristics. We show that optimized algorithms can yield order of magnitude gains compared to the naive execution. We perform a cost analysis for the fixpoint iteration techniques and we build a model that can accurately predict the cost of an iteration step during runtime. We integrate the cost model with our framework and we use it to avoid redundant computation and communication, by switching execution strategy on-the-fly. We evaluate the performance of our cost model by measuring runtime speedup and communication savings as compared to naive bulk execution.

5.1 Observing Asymmetrical Convergence

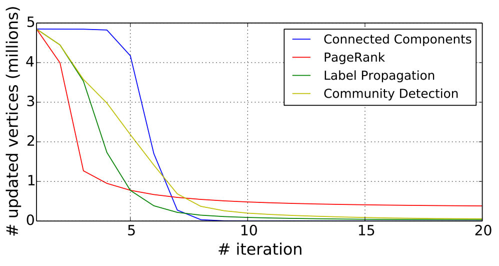
First, we examine several datasets to verify their asymmetrical convergence behavior. We select four real-world graphs of varying sizes and different network properties, shown in Table 5.1. Reported sizes correspond to raw edge list representation. Using these graphs, we run four common value propagation graph algorithms: Weakly Connected Components, PageRank, Label Propagation, and Community Detection [119]. We choose the first two algorithms because they are among the most popular applications used for graph systems evaluation, as we saw in Chapter 3. We choose the latter two applications, because they can both be used for classification in the use-case of Chapter 4. Moreover, both these algorithms require a holistic view of the neighborhood for computation, where the common vertex de-activation techniques not applicable. For every graph-application pair, we measure the number of vertices that *actually* update their values during each iteration, until convergence or until a maximum of 20 iterations. For PageRank, we consider a vertex updated if its value has changed more than 0.001% since the last iteration. Figure 5.3 shows the results.

All four algorithms exhibit asymmetrical convergence behavior on all input graphs. The decline of active vertices appears to be faster in early iterations and slower towards the last iterations. The Connected Components algorithm seems to exhibit the steepest decline. Label Propagation and Community Detection show irregularity for the Youtube graph, where after the initial decrease of active vertices, there is a small increase in later iterations, before the algorithms converge.

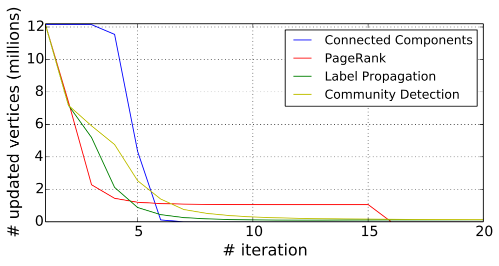
This analysis reveals an optimization opportunity and a challenge. Iterative refinement algorithms expose non-uniform convergence behavior, where big parts of the graph con-



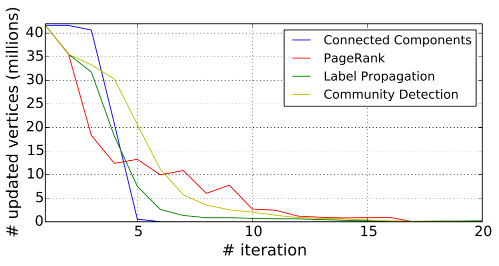
(a) Youtube



(b) Livejournal



(c) Wikipedia



(d) Twitter

Figure 5.3 – Number of updated vertices per iteration

verge early. If we can detect this behavior during runtime, we could potentially deactivate the converged parts and avoid redundant computations. However, we need to make sure that halting computation in some parts of the graph will not affect the correctness of the final result. In the following sections, we introduce four iteration techniques and we show how and when we can use them to safely exploit asymmetrical convergence in fixpoint algorithms.

5.2 Fixpoint Iteration Techniques

During the past few years, a large number of highly-specialized systems for large-scale iterative and graph processing have been developed [122, 125, 82], while there also exist general-purpose analysis systems with support for iterations and graph processing [192, 72, 136]. Some of these systems are designed to exploit dataset dependencies, in order to efficiently execute applications and avoid redundant computations. Existing graph processing frameworks exploit computational dependencies to provide efficient runtimes. However, either they assume certain properties for the update functions or they require caching and expensive state management for more complex cases. For example, Pregel [125] defines that if a vertex does not receive any messages during an iteration, it becomes inactive and does not execute or produce messages in the subsequent superstep. GraphLab [122] realizes similar behavior with its adaptive execution mechanism. However, it is left to the developer to decide when it is safe to deactivate vertices or halt parts of the computation. This requires the user to understand both models and to carefully verify the correctness of the algorithm.

In this section, we formally describe four fixpoint iteration techniques. The first technique is a general, naive approach, while the following three techniques are optimizations that leverage asymmetrical behavior. We specify when each optimization is safe to apply and we prove equivalence to the general approach. Then, we study modern graph processing systems to find which techniques they already offer and how they could support the missing ones.

5.2.1 Fixpoint Iteration Techniques

We use common graph notation to explain the iteration techniques. Let $G = (V, E)$ be an static directed graph, where V is the set of vertices, E is the set of edges, $|V| = n$, and $|E| = m$. Vertices and edges can have associated values of arbitrary type. Given G, we define the following auxiliary problem constructs:

— The *solution set*, S, is the set of all values of the vertices in V. We refer to the instance of S during iteration j as S^j and to the value of vertex i during iteration j as x_i^j.

— The *dependency collection*, D, is a collection of dependency sets, each containing the in-neighbors of a vertex in V. We refer to the set of in-neighbors of vertex i as D_i.

— The *out-dependency collection*, is a collection of out-dependency sets, each containing the out-neighbors of a vetex in V. We refer to the set of out-neighbors of vertex i as U_i.

D and U are invariant and trivial to construct from the set of edges E. In the graph example of Figure 5.1, the initial solution set would be $S^0 = \{0, na, na, na, na\}$ and the final solution set would be $S^3 = \{0, 3, 4, 6, 7\}$. The dependency sets for each vertex would be $D_S = \emptyset$, $D_A = \{S\}$, $D_B = \{S, A\}$, $D_C = \{A, B, D\}$ and $D_D = \{B\}$ and the out-dependency sets $U_S = \{A, B\}$, $U_A = \{B, C\}$, $U_B = \{C, D\}$, $U_C = \emptyset$ and $U_D = \{C\}$, respectively.

Let F be an update function defined over the domain of the values of the vertices, where F is decomposable into partial functions for computing each $x_i \in S$. If F has a fixed point which can be computed in a finite number of iterations, we can compute it by iteratively executing the following procedure:

$$S^{j+1} := F(S^j, D) \; until \; S^{j+1} = S^j \tag{5.1}$$

The Bulk Technique

As we discussed in the beginning of this chapter, our goal is to detect non-uniform behavior in iterative graph algorithms and avoid redundant computations. Essentially, we want to find the *active* parts in the graph and only engage these parts in the computation, while guaranteeing algorithm correctness. A trivial approach to the problem is to assume that the whole graph is constantly active and participates in the computation of each iteration, until convergence. We call this technique the *bulk iteration technique*. During a bulk iteration, *all* the elements of the solution set S are recomputed, by applying the update function F to the result of the previous iteration. In the end of an iteration, S is updated with the newly computed values. The algorithm terminates when none of the values of the solution set changes, i.e. the newly computed S in iteration i is identical to the solution set of the previous iteration. The pseudocode for the bulk technique is given in Algorithm 15.

Algorithm 15 Bulk Iteration Algorithm

```
$S^0 := \{x_1^0, x_2^0, ..., x_n^0\}$
$i := 0$
repeat
    foreach $x_j \in S^i$
        $x_j^{i+1} := f_j(x_j^i, D_j)$
    end
    $S^{i+1} := \{x_1^{i+1}, x_2^{i+1}, ..., x_n^{i+1}\}$
    $i := i + 1$
until $S^i = S^{i-1}$ .
```

The Incremental Technique

In the introductory example of Figure 5.1, we saw that in some cases we can safely detect inactive vertices of a graph based on whether their values have changed during two consecutive iterations. Here, we specify the properties that the update function must have in order to safely apply this optimization. Specifically, if the function f_j is of the form $f_j = t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_n$, where $t_1, t_2, \cdots t_n$ represent independent contributions and f_j is *distributive* over the combination operator $\sqcup$, then we only need to compute f_j on the changed values of the solution set in each iteration and then, combine the result with the previous value. We call this technique *incremental iteration technique*. For example, if t is the identity function and the combination operator is $minimum$, then $f_j = min(t(D_j)) = t(min(D_j))$. In the graph of Figure 5.1, the value of node B depends on the values of nodes S and A, thus, $D_B = \{S, A\}$. Then, $f_B = min(t(\mathit{value}(S)), t(\mathit{value}(A)))$ = $t(min(\mathit{value}(S), \mathit{value}(A)))$ = $min(\mathit{value}(S), \mathit{value}(A))$.

We introduce two auxiliary sets, the *workset* W and the *candidate set* Z. In each iteration, W stores the vertices which have changed value since the last iteration and Z stores the *candidate* vertices for re-computation: vertices whose *at least one* in-neighbor has changed value during the last iteration. Z is essentially an overestimation of the ideal set of vertices that are guaranteed to require recomputation. Figure 5.4 shows an example of how parts of an input graph get de-activated during an execution of the weakly connected components algorithm. In each iteration, W contains the orange vertices. The contents of the Z set in each iteration are shown in Figure 5.5 with green color.

The pseudocode for the incremental technique is given in Algorithm 16.

Proposition 1. *If f_j is of the form $f_j = t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_n$, where $t_1, t_2, \cdots t_n$ represent independent contributions to the value of f_j, i.e. f_j is distributive over the combination operator $\sqcup$ and f_j is also idempotent and weakly monotonic, then the Incremental technique is equivalent to the Bulk technique.*

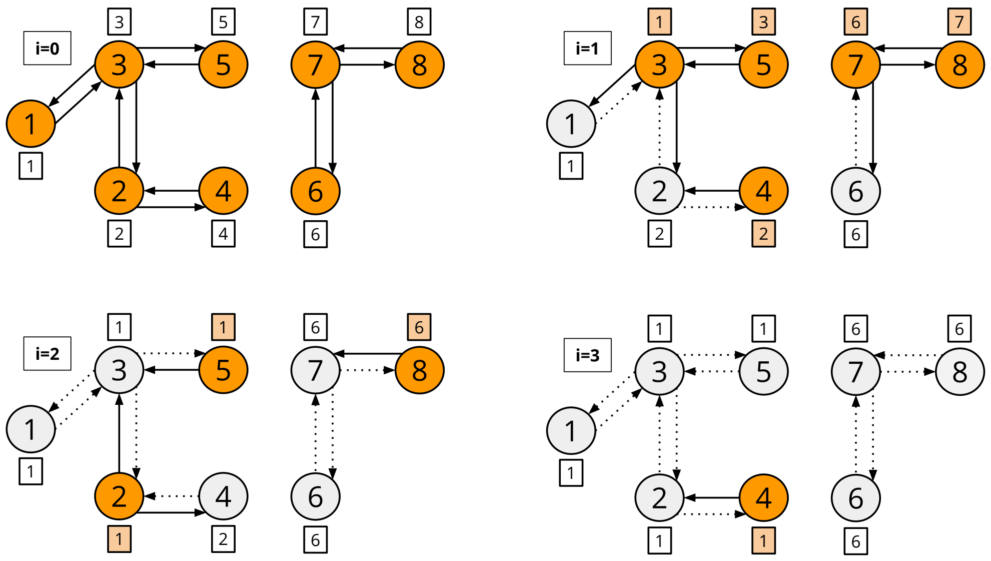


Figure 5.4 – Graph activation in the Connected Components algorithm, using the Incremental Technique. Active nodes are shown in orange color and active edges in solid lines. The set of active nodes during iteration i contains the nodes which updated their values since iteration $i - 1$.

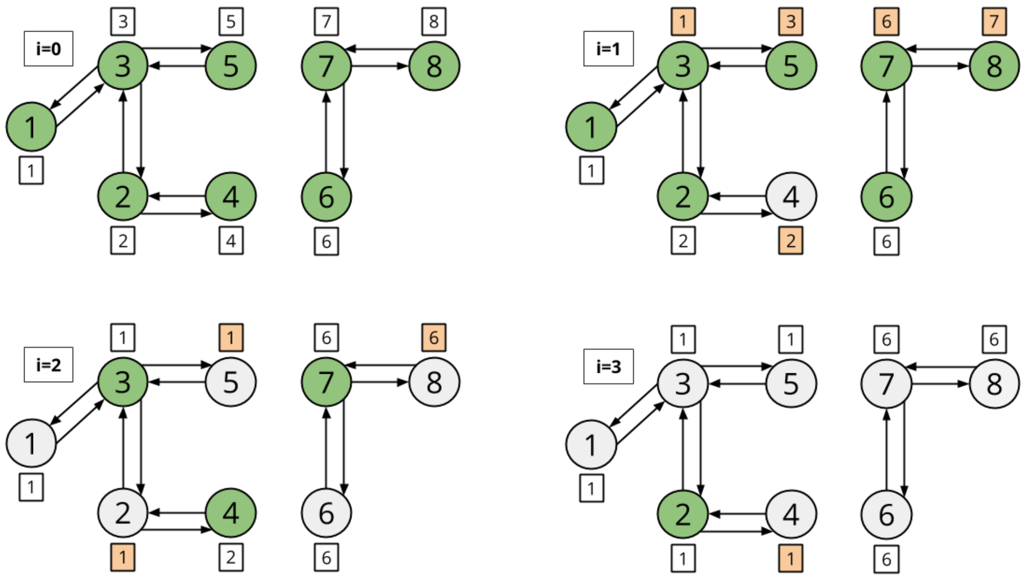


Figure 5.5 – Candidate vertices for re-computation in the Connected Components algorithm. The set of candidates during iteration i contains the nodes whose at least one in-neighbor has updated its value since iteration $i - 1$.

Algorithm 16 Incremental Iteration Algorithm

```
$S^0 := \{x_1^0, x_2^0, ..., x_n^0\}$
$W^0 := S^0$
$i := 0$
**repeat**
    **foreach** $x_j \in W^i$
        $Z^i := Z^i \cup U_j$                                    ▷ generate candidates
    **end**
    **foreach** $x_j \in Z^i$
        $x_j^{i+1} := f_j(x_j^i, W_j^i)$
        **if** $x_j^{i+1} \neq x_j^i$ **then**
            $W^{i+1} = W^{i+1} \cup \{x_j^{i+1}\}$
        **end if**
    **end**
    $S^{i+1} := \{x_1^{i+1}, x_2^{i+1}, ..., x_n^{i+1}\}$
    $i := i + 1$
**until** $W^i = \emptyset$ .
```

Proof. Let x_j^i be the value of an element of S in iteration i. Since f_j is distributive over $\sqcup$ and idempotent then $x_j^i = f_j(x_j^{i-1} \sqcup t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_n) = f_j(x_j^{i-1} \sqcup T \sqcup t_n)$, where $T = t_1 \sqcup t_2 \sqcup \cdots \sqcup \cdots t_{n-1}$. Let us assume that during iteration i, only t_n changed value and therefore

$$x_j^{i+1} = f_j(x_j^i \sqcup T \sqcup t'_n) \tag{5.2}$$

Since f_j is idempotent,

$$f_j(t_n, t_n) = t_n$$

and

$$f_j(x_j^i \sqcup T \sqcup t'_n) = f_j[T \sqcup t'_n \sqcup f_j(T \sqcup t_n \sqcup x_j^{i-1})] \rightarrow$$
$$f_j(x_j^i \sqcup T \sqcup t'_n) = f_j(T \sqcup t_n \sqcup t'_n \sqcup x_j^{i-1}) \rightarrow$$
$$f_j(x_j^i \sqcup T \sqcup t'_n) = f_j[f_j(T \sqcup t_n \sqcup x_j^{i-1}) \sqcup t'_n)] \rightarrow$$
$$f_j(x_j^i \sqcup T \sqcup t'_n) = f_j(x_j^i \sqcup t'_n) \rightarrow$$
$$x_j^{i+1} = f_j(x_j^i \sqcup t'_n)$$ □

Returning to the SSSP example, $minimum$ is also idempotent ($min(a, a) = a$) and also weakly monotonic, since, for $a \leq a'$ and $b \leq b'$, $min(a, b) \leq min(a', b')$.

The Delta Technique

Ideally, for each change δx in the input, we would like to have an efficient function δF, such that: $F(x \oplus \delta x) = F(x) \oplus \delta F(x, \delta x)$ where $\oplus$ is a binary composition operator. In this ideal scenario, we could propagate only the differences of values, or *deltas*, from each iteration to the next one. That would potentially decrease the communication costs and make the execution more efficient. The *delta iteration technique* uses only the *differences* of values to compute the fixpoint.

There are two major factors to consider when implementing the delta technique. First, it might not always be the case that computing $\delta F(x, \delta x)$ is more efficient than simply computing $F(x \oplus \delta x)$. Moreover, even if we are able to find an efficient δF, combining its result with $F(x)$ could still prove to be a costly operation. In the special case where the update function f is *linear* over the composition operator $\oplus$, then

$$F(x \oplus \delta x) = F(x) \oplus F(\delta x) \tag{5.3}$$

in which case we can use the same function f in the place of δf.

For example, if $f = sum(D)$, this optimization is applicable. Let us assume that $D^i = \{a, b\}$ and $D^{i+1} = \{a', b\}$, where $a' = a + \delta a$. Then, $f^{i+1} = sum(a', b) \Rightarrow f^{i+1} = sum(a + \delta a, b) \Rightarrow f^{i+1} = sum(a, b, \delta a) = sum(f^i + \delta a)$.

Proposition 2. *If f_j is linear over the composition operator $\oplus$, then the delta technique is equivalent to the bulk technique.*

The pseudocode is given in Algorithm 17, while the proof is trivial and based on equation 5.3.

Algorithm 17 Delta Iteration Algorithm

```
$S^0 := \{x_1^0, x_2^0, ..., x_n^0\}$
$\Delta^0 := init$
$i := 0$
**repeat**
    **foreach** $x_j \in \Delta^i$
        $Z^i := Z^i \cup U_j$                                  ▷ generate candidates
    **end**
    **foreach** $x_j \in Z^i$
        $\delta x_j^{i+1} := f_j(x_j^i, \Delta^i)$
        **if** $\delta x_j^{i+1} > \epsilon$ **then**
            $\Delta^{i+1} = \Delta^{i+1} \cup \{\delta x_j^{i+1}\}$
        **end if**
    **end**
    $S^{i+1} := S^i \oplus \Delta^{i+1}$
    $i := i + 1$
**until** $\Delta^{i+1} = \emptyset$ .
```

The Dependency Technique

When the bulk technique computes a vertex value, it may produce the same result as the one of the previous iteration. This may happen because (a) either none of the values in the dependency set of the vertex has changed since the previous iteration or (b) applying the update function to the changed values happens to return an identical result. Ideally, we would like to recompute only the values of the vertices that are *guaranteed* to change value during an iteration. Instead, the *dependency iteration technique* exploits the dependency set to safely select the vertices that are *likely* to change value in the next iteration and only recompute those.

The intuition behind this technique is that if a vertex of the graph changes value, then, all its out-neighbors are likely to be affected by this change. On the other hand, if none of the dependencies of a vertex changes value, it is safe to exclude this vertex from the next computation, since recomputing the update function on the same arguments, would return an identical result. The pseudocode of the dependency technique is shown in Algorithm 18.

Algorithm 18 Dependency Iteration Algorithm

```
$S^0 := \{x_1^0, x_2^0, ..., x_n^0\}$
$W^0 := S^0$
$i := 0$
**repeat**
    **foreach** $x_j \in W^i$
        $Z^i := Z^i \cup U_j$                              ▷ generate candidates
    **end**
    **foreach** $x_j \in Z^i$
        $x_j^{i+1} := f_j(x_j^i, D_j)$                     ▷ (5.1.1)
        **if** $x_j^{i+1} \neq x_j^i$ **then**
            $W^{i+1} = W^{i+1} \cup \{x_j\}$               ▷ (5.1.2)
        **end if**
    **end**
    $S^{i+1} := \{x_1^{i+1}, x_2^{i+1}, ..., x_n^{i+1}\}$
    $i := i + 1$
**until** $W^i = \emptyset$ .
```

Proposition 3. *The dependency technique is equivalent to the bulk technique.*

Proof. We prove this statement using the method of contradiction. Let us assume that the two algorithms are not equivalent. Then, there exists an initial input set S_0 and a function f for which the two algorithms converge to different solution sets. Let us assume that the algorithms give identical partial solution sets until iteration i, but the results diverge in iteration $i + 1$. If S_b^{i+1} is the partial solution set produced by the execution of the bulk technique and S_w^{i+1} is the partial solution set produced by the execution of the dependency technique after iteration $i + 1$, there should exist at least one element that is different in the two sets.

Since W is a subset of S, that would mean that the dependency technique failed in identifying all the vertices that required re-computation during iteration i, i.e. there exist $x_{j,b}^i \in S_b^i$, $x_{j,b}^{i+1} \in S_b^{i+1}$, $x_{j,z}^i \in S_w^i$, $x_{j,w}^{i+1} \in S_w^{i+1}$, such that

$$x_{j,b}^i \neq x_{j,b}^{i+1} \tag{5.4}$$

Iteration Technique	Equivalent to Bulk?	Vertex Activation	Vertex Update
Bulk	n/a	always	using values of all in-neighbors
Dependency	always	if any in-neighbor is updated	using values of all in-neighbors
Incremental	f idempotent and weakly monotonic	if any in-neighbor is updated	using values of changed in-neighbors
Delta	f is linear over composition operator	if any in-neighbor is updated	using values of changed in-neighbors

Table 5.2 – Iteration Techniques Equivalence

and

$$x^i_{j,w} = x^{i+1}_{j,w} \tag{5.5}$$

From the relations 5.4 and (5.1.1) we can derive the following relation:

$$f_j(D^{i-1}_j) \neq f_j(D^i_j)$$

From the relations 5.5 and (5.1.2) we can derive the following relation:

$$f_j(D^{i-1}_j) = f_j(D^i_j)$$

and we have therefore arrived at a contradiction. □

Table 5.2 summarizes the equivalence among the different techniques and the conditions for safely applying each optimization.

5.2.2 Iteration Techniques in Graph Processing Systems

The vertex-centric Pregel model [125] naturally translates to the incremental iteration technique. Vertices receive messages from neighbors and compute their new value using those messages only. The candidates set Z can be seen as maintaining the subset of the active vertices for the next superstep. The delta iteration technique can be easily expressed using the vertex-centric model, if vertices produce deltas as messages for their neighbors. To emulate a bulk iteration in the Pregel model, vertices simply need to transfer their state to all their neighbors, in every iteration. Vertices would remain active and not vote to halt, even if they do not have an updated state. Implementing the dependency iteration technique in Pregel is not trivial. Pregel uses a *push* communication model, where vertices send messages to their out-neighbors only. However, in the dependency technique, if a vertex is candidate for re-computation, it needs to *activate* all its in-neighbors. In order to achieve that, we could add a pre-processing step, where all vertices send their IDs to their out-neighbors, so that they can create auxiliary out-going edges to them. The computation can then proceed by using a three supersteps as one dependency iteration: during the first

System	Bulk	Dependency	Incremental	Delta
Pregel	**	***	*	**
GraphLab	*	*	**	**
GraphX	**	***	*	**
Powergraph	*	***	*	*
Stratosphere	*	**	*	**

Table 5.3 – Iteration Techniques in Graph and Iterative systems. *: provided by default, **: can be easily implemented, ***: possible, but non-intuitive

step, vertices with updated values produce messages for their out-neighbors. During the second step, vertices that receive at least one message are candidates for re-computation and produce messages for all their in-neighbors, while the rest of the vertices become inactive. In the third step, the candidates for re-computation receive messages from all their in-neighbors and update their value.

GraphLab's [122] programming abstraction consists of the data graph, an update function and the sync operation. The data graph structure is static, similar to what we assume for the dependency set. GraphLab introduces the concept of the scope of a vertex, which is explicitly declared and refers to the set of values of a vertex and its neighbors. This scope corresponds to the dependency set in the bulk and dependency techniques and to the intersection of the dependency set of a vertex and the workset, in the incremental and delta iterations. Therefore, all four techniques can be implemented in GraphLab by computing the appropriate scopes.

PowerGraph [81] is a graph processing system for computation on natural graphs. It introduces the Gather-Apply-Scatter (GAS) abstraction, which splits a program into these three phases. During the gather phase, a vertex collects information from its neighborhood, which then uses during the apply phase to update its value. During the scatter phase, the newly computed values are used to update the state of adjacent vertices. The GAS abstraction can be used to implement both the bulk and the incremental iteration techniques, while the delta technique is equivalent to PowerGraph's delta caching mechanism. The model does not intuitively support the dependency technique. However, it can be implemented in a similar way to the three-step superstep described for Pregel.

GraphX [82] is a graph processing library built on top of Spark [192], in order to efficiently support graph construction and transformation, as well as graph parallel computations. The programming model of GraphX is similar to that of Pregel and PowerGraph.

Stratosphere [23] supports flexible plans in the form of a Directed Acyclic Graph (DAG) of operators. Iterations are implemented in Stratosphere as composite operators, which encapsulate the step function and the termination criterion. The implementation of the bulk and the incremental algorithms are described in [72]. All of the iteration techniques described above can be easily implemented in Stratosphere.

Table 5.3 summarizes the support of each technique in popular graph processing and iterative systems. Most of the existing systems implement the bulk technique by default and special implementations of operators to support the delta optimization. These models assume that the update function has the required characteristics, or that it can be easily re-written to fulfill the required conditions. Therefore, they do not usually expose the implementation of an equivalent to the more general dependency technique. Indeed, it is often trivial to derive an incremental or delta version of a common aggregate update function, like minimum or average. The dependency technique is useful when an incremental or delta version of the update function cannot be easily derived. Examples include *holistic* functions, such as count distinct, mode, and median. Moreover, the dependency technique can be safely used when the properties of the update function are not known to the user; for instance when calling an update function of an external library.

5.3 Fixpoint Execution Plans Implementation

There has been significant recent work on mapping distributed graph processing models to relational dataflow execution plans [82, 72, 42, 101]. In fact, special optimizations and evaluation strategies proposed in graph programming systems have a foundation in relational query processing primitives. This realization shows a promising avenue to deploy sophisticated query optimization and query execution techniques in Pregel-like environments. Motivated by the increasing interest in using general-purpose distributed dataflow engines for graph processing, we implement the four fixpoint iteration techniques with Apache Flink [1, 46].

Apache Flink is a general-purpose platform for distributed data processing built on a streaming dataflow engine. Flink has native support for iterations and a custom optimizer for batch programs which make it suitable for iterative graph processing. In this section, we give a high-level overview of distributed dataflow processing with Apache Flink and we show how the previously introduced iteration techniques can be mapped to concrete execution plan implementations.

5.3.1 Apache Flink Overview

Apache Flink supports both batch and stream data processing on top of the same execution engine. Users write Flink programs using either the *DataSet* or the *DataStream* API for a batch or stream application respectively. These APIs are expressive and declarative and allow reading data from diverse sources, applying transformations on distributed data sets or streams, and writing data to several data sinks, such as distributed file systems, databases, or message queues. Specialized libraries for graph processing, complex event processing, and machine learning have been built on top of these APIs. Here, we focus on the DataSet API and the development and execution of batch programs in Flink.

A Flink program consists of one or multiple data sources, a set of transformations, and one or multiple data sinks. Transformations are defined on *Datasets*, an abstraction of immutable distributed collections of typed data elements. A transformation takes one or more datasets as input and generates one or more datasets as output. The DataSet API includes implementations of common transformations, such as *map*, *flatMap*, *groupBy*, and *join*. Before execution, Flink programs are mapped to a directed acyclic dataflow graph (DAG) of operations. This DAG is then optimized to create an *execution plan*. The system automatically decides on physical execution strategies and handles data distribution, parallelization, and task assignment to available resources.

The DataSet API

Flink DataSet programs are written in Java or Scala and implement transformations on immutable distributed collections of data. Flink supports simple and complex data types, including Scala case classes and Java POJOs. Flink also provides its own custom *Tuple* type. Tuples are composite types containing up to 25 fields of various types and can be efficiently serialized by the system. We give an overview of basic DataSet transformations below.

— **map**: Receives a *MapFunction* that describes how each element of the input DataSet is mapped to one element of the output DataSet.

— **flatMap**: Receives a *FlatMapFunction* that describes how each element of the input DataSet is mapped to zero, one, or more element of the output DataSet.

— **reduce**: Receives a *ReduceFunction* that combines a group of elements into a single element of the same type. The ReduceFunction repeatedly combines the input elements two-by-two, until a single element remains.

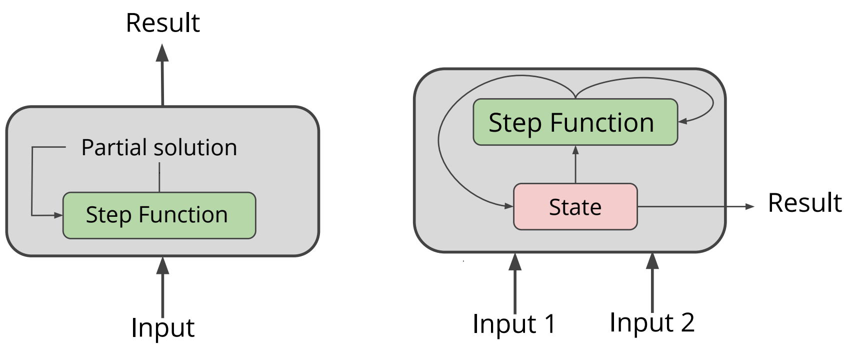
— **join**: Joins two datasets on a specified key. The operator creates all pairs of elements that share the key and applies a user-defined *JoinFunction* on each pair.

For a complete set of available transformations, we refer the reader to the Apache Flink documentation [1].

Iterations

The DataSet API contains two specialized operators for synchronized iterations. Both operators repeatedly execute a user-defined set of transformations, referred to as the *step function*. Iterations proceed in synchronized supersteps according to the BSP model. During a superstep, the step function is evaluated in parallel on different partitions. The iteration operators produce their output after a user-specified maximum number of iteration steps or when a user-defined convergence criterion is met.

The *Iterate* operator implements a simple form of an iterative dataflow program, where a single input feeds a step function. The step function consumes the entire input dataset and produces a new dataset, the *partial solution*, which serves as the input of the following iteration step. The iterate operator is shown in Figure 5.6a. The *Delta Iterate* operator receives two input datasets and maintains state in the form of a distributed hash table. The operator is shown in Figure 5.6b. The first input, the *solution set*, serves as the initial value of the state and feeds the step function together with the second input, the *workset*. During an iteration, the step function is evaluated and produces two outputs. The first output corresponds to updates that are applied on the solution set and the second output becomes the next workset. The iterative computation finishes when the workset is empty or when a custom convergence criterion is met. Then, the solution set becomes the output of the delta iterate operator.



(a) Iterate (b) Delta Iterate

Figure 5.6 – Iteration Operators in Apache Flink

Optimizer

The Apache Flink optimizer is similar to parallel database optimizers. The optimizer chooses join execution strategies (e.g. hash-join, sort-merge join) and data shipping strategies (e.g. broadcast or shuffling). Moreover, it is able to automatically reuse existing partitioning and sort orders across operators. In the case of iterations, the optimizer can detect and cache loop-invariant data and move certain operations out of the loop.

5.3.2 Bulk Plan

The bulk iteration technique can be easily implemented using Flink's *iterate* operator. An execution plan implementing the bulk iteration technique using Flink operators is shown in Figure 5.7a. The solution set S contains the vertices of the input graph and the dependency set D contains directed edges (in case of an undirected graph, each edge appears twice, covering both dependencies). In every iteration, the set of vertices is joined with the set of edges to produce the dependencies (neighbors) of each vertex. For every neighbor-pair, the join emits one record with the target vertex ID as key. Then, the user-defined step function is applied on the neighbor pairs. The output of the step function contains the newly computed vertex values. These are joined with the previous values, in order to update the solution set and check the convergence criterion (if any). The new vertex values completely replace the solution set input for the following iteration.

5.3.3 Incremental and Delta Plans

The Incremental and Delta techniques can be implemented in Flink using its *delta iterate* operator, as shown in Figure 5.7b. The initial workset and solution set inputs consist of the set of vertices and their initial values. The iteration state, the solution set, consists of vertex ID - vertex value pairs and is updated after each iteration. The workset is replaced after each iteration and it contains only *active* vertices; vertices that have updated their values. In the beginning of an iteration, the workset is joined with the dependency set (edges)

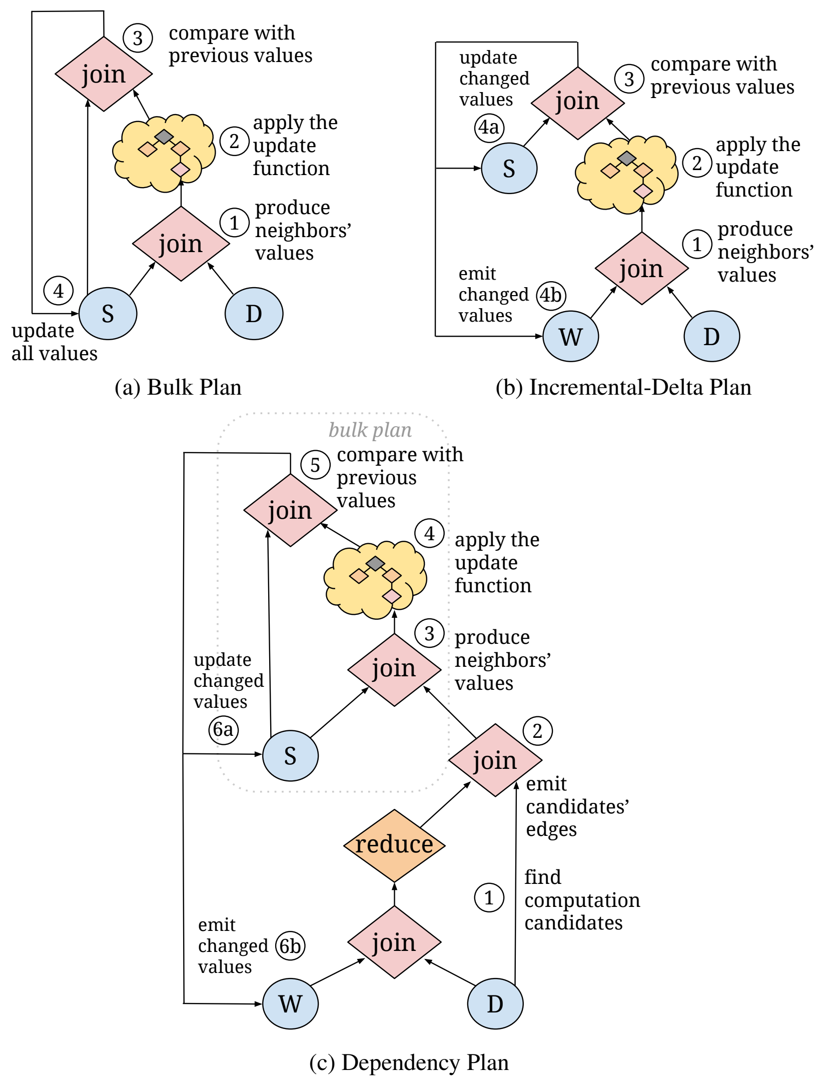

3 compare with previous values
join
2 apply the update function
1 produce neighbors' values
join
4
update all values
S
D
(a) Bulk Plan
update changed values
4a
3 compare with previous values
join
S
2 apply the update function
1 produce neighbors' values
join
emit changed values 4b
W
D
(b) Incremental-Delta Plan
bulk plan
5 compare with previous values
join
4 apply the update function
3 produce neighbors' values
join
update changed values
6a
S
2
join
emit candidates' edges
reduce
find computation candidates
1
emit changed values 6b
join
W
D
(c) Dependency Plan

Figure 5.7 – Fixpoint Graph Iteration Techniques as Data Flow Plans. The yellow *cloud* operator corresponds to the dynamic, user-defined part of the plan.

to produce updates for candidate vertices. The update function is applied to the neighbor values and produces a set of new vertex ID - value pairs. This set is then joined with the solution set, to filter out vertices which have not updated their values. The join output is used to update the solution set and also serves as the new workset. In the case of the Delta technique, the initial value differences have to be computed before the computation begins. This can be easily achieved by chaining a single bulk iteration before the incremental plan.

5.3.4 Dependency Plan

An implementation of the dependency technique is shown in Figure 5.7c. The workset W is isomorphic to the solution set S and contains only the records corresponding to the vertices that changed value during the previous iteration. First, W is joined with the dependency set to generate candidate vertices for computation, emitting a record with the target vertex id as the key for each match. A reduce transformation is then applied on the candidate vertices to remove duplicates. The result is joined with the dependency set on the target vertex ID, producing a subset of the dependency set. This set contains only *active* edges, like the ones shown in solid lines in Figure 5.4. The resulting set serves as input to a subplan equivalent to a bulk iteration. This bulk plan only computes new values for vertices whose at least one in-neighbor has changed value during the previous iteration. The final join operator, only emits records containing active vertices back to the workset.

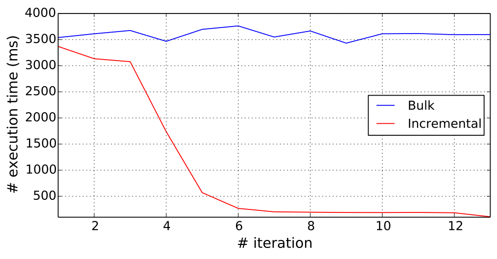
The dependency technique can alternatively be implemented using the incremental plan and a caching mechanism. In this case, each vertex has to maintain a local cache of all neighbor values. Using the incremental plan, vertices receive only values of updated neighbors and retrieve the rest of the neighbors values from the cache. Note that such an implementation is feasible in Flink, because Flink operators are *stateful*. Operator state is maintained across iterations and thus, it can serve as a cache for the dependency plan. However, this implementation would be challenging to realize in systems with stateless operators, like Apache Hadoop, Apache Giraph, and Apache Spark.

5.4 Fixpoint Execution Plans Performance Comparison

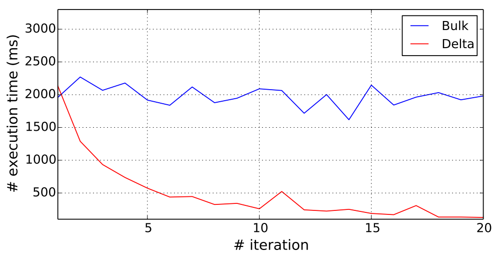
In this section, we provide an evaluation of the iteration techniques and their implementation in Apache Flink. Our goal is to compare the performance of the iteration techniques, using a common execution platform and reveal potential optimization opportunities. For a performance comparison of Flink with other distributed data processing platforms, we refer the reader to [72] and [23].

We evaluate the performance of four iterative algorithms, Weakly Connected Components, PageRank, Label Propagation, and Community Detection, using the datasets of Table 5.1. The update function of the Connected Components algorithm (minimum) satisfies the conditions of the incremental technique. Therefore, we implement this application using the bulk and incremental plans. The update function of PageRank (summation of partial ranks) satisfies the conditions of the delta technique. Thus, we implement this application using the bulk and delta plans. Initial deltas are derived from the difference between the uniform initial rank and the in-degree proportional rank. The Label Propagation and Community Detection algorithms cannot be implemented with the incremental or delta techniques. We implement these algorithms using the bulk and the dependency techniques.

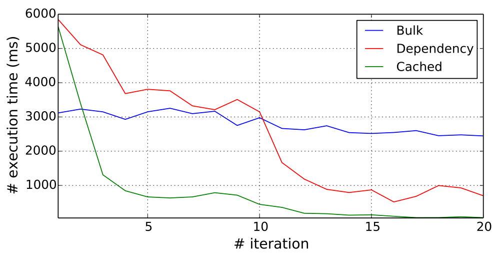
We run our experiments on AWS, using r3.2xlarge instances. Each instance has 8 virtual CPUs, 61 GB of memory, and 160 GB of SSD storage. We use one such instance for the experiments on the Youtube dataset, two instances for Livejournal, four instances for Wikipedia, and eight instances for the Twitter dataset. We read data from HDFS (Hadoop 2.7.2) and we implement all algorithms using Apache Flink 1.0.3.



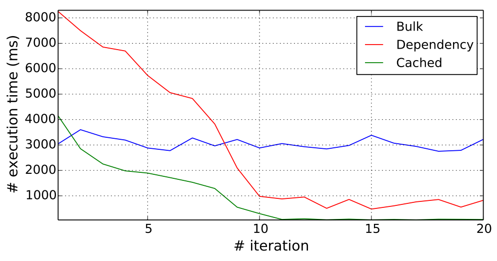
(a) Connected Components



(b) PageRank



(c) Label Propagation



(d) Community Detection

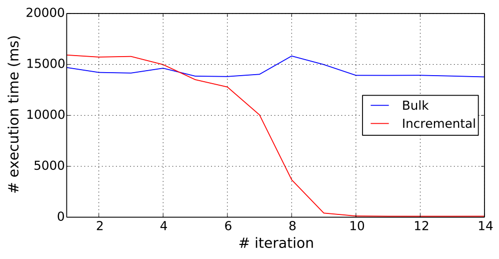
Figure 5.8 – Execution Time per Iteration - Youtube

5.4.1 Results

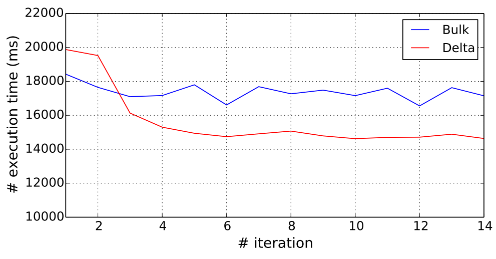
Figures 5.8, 5.9, 5.10, and 5.11 show execution time measurements for the Youtube, Livejournal, Wikipedia, and Twitter graph respectively. In each figure, we have plotted the execution time per iteration for the bulk, dependency, incremental, and delta plan implementations. For the Youtube and Livejournal graphs, we also plot the execution time per iteration for the alternative implementation of the dependency plan, using a cache.

As expected, the time for all bulk iterations is stable throughout execution, for all algorithms and data sets examined. The incremental and delta plans match the bulk plan performance during the first iterations, while they significantly outperform it as the workset shrinks. In all cases, the optimized plans outperform the bulk plan by an order of magnitude or more, as iterations proceed. Regarding the dependency plan, we observe that in the first few iterations, it is consistently less efficient than the bulk plan. This is because the dependency plan first needs to identify the candidate elements for computation and retrieve their dependencies. This pre-processing step imposes an overhead compared to the bulk execution. When the number of elements in the workset is close to the total number of elements in the solution set, the overhead of the pre-processing step is larger than the time we save by updating less elements.

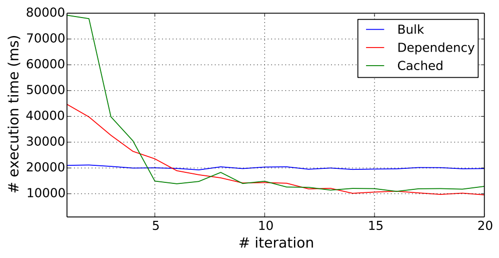
The cached dependency plan performs well for small datasets, but its performance quickly degrades for larger inputs. Indeed, using Livejournal, the first few iterations are almost two times slower compared to the stateless dependency plan and 4-6 times slower than the bulk plan. Using the caching mechanism with the Wikipedia and Twitter datasets



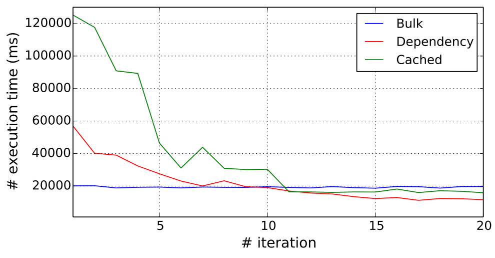
(a) Connected Components



(b) PageRank



(c) Label Propagation

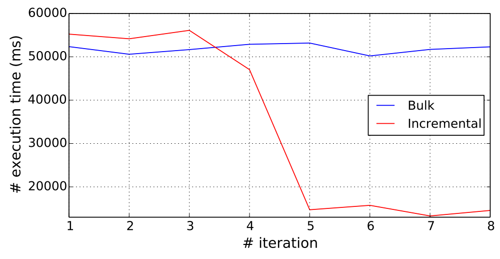


(d) Community Detection

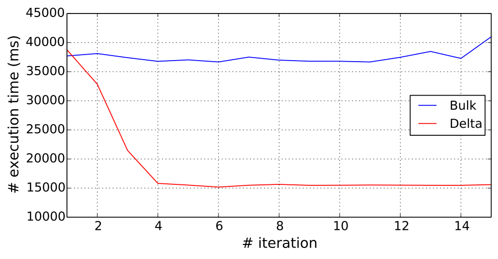
Figure 5.9 – Execution Time per Iteration - Livejournal

proved impractical, given our available resources. Maintaining the cache poses a significant overhead that negates all benefit. In early iterations, almost all vertices change value, thus invalidating their cache entries. Moreover, this implementation has much higher memory requirements than the stateless dependency plan. Even inactive vertices that might not receive any update from neighbors need to maintain their caches until the computation converges.

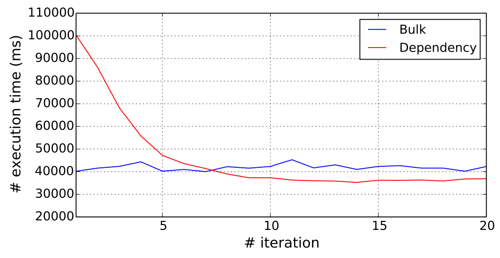
Our experiments show that the incremental and delta plans save a lot of redundant computations and should always be preferred over bulk plans. Using these techniques is a well-known and widely used optimization. However, these plans can be used only when the update function of the algorithm satisfies the conditions described in Section 5.2.1 for the incremental and the delta techniques respectively. What is more interesting to examine is when and how the more general dependency plan can be used to speed up total execution time. Our results show that there is a trade-off that depends on the size of the workset. Two main factors contribute to the cost of the dependency plan. First, there is the cost of the preprocessing step that computes how many vertices are candidates for computation. This cost decreases as the workset decreases. The second factor is the cost of the actual computation, which also decreases as the number of active vertices declines. We observe that there is a threshold of active vertices under which the dependency plan outperforms the bulk plan. In the following section, we build a cost model that is able to capture and quantify the overhead of the dependency plan over the bulk plan. We integrate the cost model with our fixpoint iteration framework that is able to choose the most efficient iteration plan, at runtime.



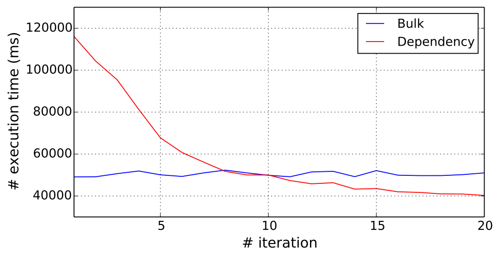
(a) Connected Components



(b) PageRank



(c) Label Propagation



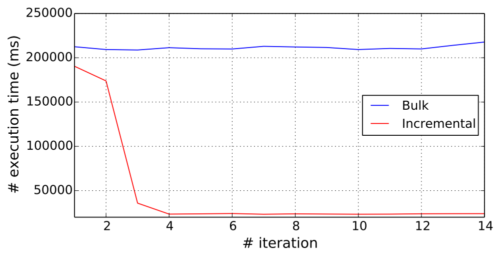
(d) Community Detection

Figure 5.10 – Execution Time per Iteration - Wikipedia

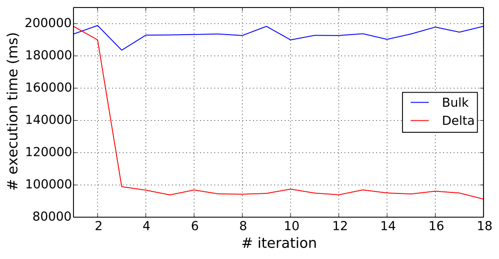
5.5 Cost-Based Optimization for Fixpoint Iterations

As demonstrated by our experimental analysis, the incremental and delta plans always perform better than the bulk plan, for the applications we consider. However, it is still unclear when using the dependency plan instead of the bulk plan can be beneficial. The dependency plan, as shown in Figure 5.7c, consists of a bulk plan (inside the dotted box) plus a pre-processing step. This pre-processing is required in order to find the *candidate set*; vertices that need to participate in the computation. Such vertices are located by filtering out the vertices whose neighbors did not change since the last iteration and, thus, do no need to recompute their value. If the size of the candidate set is in the order of the solution set size, then the cost of the dependency plan will roughly be equal to the cost of the bulk plan plus the overhead of the pre-processing step. As iterations proceed and the workset is getting smaller, the number of candidates also declines. Consequently, both the cost of the pre-processing step and the cost of the bulk sub-plan decrease over time (since the bulk iteration plan operates on decreasing input). The challenge is to identify the workset size threshold for which the dependency plan cost becomes less than the bulk plan cost.

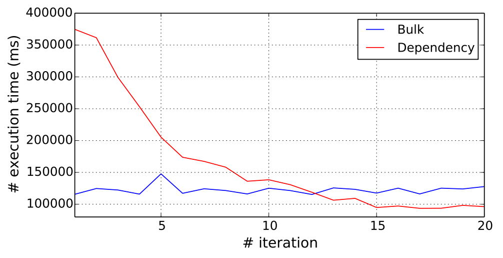
In this section, we present a framework for fixpoint iterations built on top of Apache Flink. We develop a cost model for the fixpoint bulk and dependency techniques. Our framework uses a common fixpoint API for expressing all four iteration techniques and leverages the cost model to choose the best iterative execution plan during runtime. Using the cost model we achieve up to 1.7x speedup on iterative applications and 54% communication savings, as compared to using the bulk execution technique.



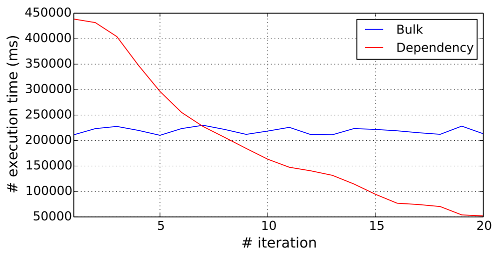
(a) Connected Components



(b) PageRank



(c) Label Propagation



(d) Community Detection

Figure 5.11 – Execution Time per Iteration - Twitter

5.5.1 An Approximate Cost Model for Fixpoint Applications

We approximate the costs of the two plans in terms of their input sizes. We assume that the costs of operators are proportional to the size of their input and that the input size defines the dominant cost of an operator. Therefore, in our model, any two operators of the same type will have the same cost when accepting the same amount of input. We denote the cost of a join operation with C_j and the cost of a reduce operation with C_r. Let us also define the size of the solution set as $|S|$, the size of the dependency set as $|D|$ and the size of the workset as $|W|$. The bulk plan cost is constant throughout iterations and can be approximated as follows.

$$BulkCost = C_j * (|S| + |D|) + C_r * |D| + C_j * 2 * |S| \Rightarrow$$

$$BulkCost = 3 * C_j * |S| + (C_j + C_r) * |D| \tag{5.6}$$

The cost of the dependency plan is different in every iteration and depends on the convergence rate of the algorithm and the properties of the input dependency graph. Consider that in iteration k, the workset contains $\lambda_k * S$ vertices and that, given a their neighborhoods, these $\lambda_k * S$ vertices will produce $\mu_k * S$ candidates for computation. Assuming a regular graph so that each node has approximately the same number of neighbors, we can then express the dependency plan cost in iteration k as follows.

$$\begin{aligned} DepCost_k = C_j * (\lambda_k * |S| + |D|) + C_r * \lambda_k * |D| + \\ C_j * (|D| + \mu_k * |S|) + C_j * (|S| + \mu_k * |D|) + \\ C_r * \mu_k * |D| + C_j * (|S| + \mu_k * |S|) \Rightarrow \end{aligned}$$

$$\begin{aligned} DepCost_k = C_j * (\lambda_k + 2 * \mu_k + 2) * |S| + \\ [C_j * (\mu_k + 2) + C_r * (\mu_k + \lambda_k)] * |D| \end{aligned} \tag{5.7}$$

Based on the assumption that operation cost depends solely on the amount of input, we can simplify, equations 5.6 and 5.7 by substituting $C_j, C_r = 1$

$$BulkCost = 3 * |S| + 2 * |D| \tag{5.8}$$

$$DepCost_k = (\lambda_k + 2 * \mu_k + 2) * (|S| + |D|) \tag{5.9}$$

We seek the values of λ and μ, such that

$$DepCost_k \leq BulkCost \Rightarrow$$

$$(\lambda_k + 2 * \mu_k + 2) * |S| + |D| \leq 3 * |S| + 2 * |D| \tag{5.10}$$

For a regular graph, the average node degree d of is equal to the number of edges divided by the number of the vertices:

$$d = \frac{|D|}{|S|} \Rightarrow |D| = d * |S|$$

Substituting in equation 5.10, we get:

$$(\lambda_k + 2 * \mu_k) * (d + 1) \leq 1 \Rightarrow$$

$$\lambda_k + 2 * \mu_k \leq \frac{1}{d + 1} \tag{5.11}$$

Conceptually, λ expresses how fast the workset is shrinking, while μ depends on the value of λ and the clustering coefficient of the input graph.

Unfortunately, neither λ nor μ can be measured without pre-processing the input. Moreover, such a step could prove very expensive for the orders of input sizes that we are considering. Another challenge is that the decrease rate of the workset is not only a graph property but also depends on the algorithm, so a pre-processing step would give us λ only for one specific application. It seems that computing the cost of a dependency plan for the whole execution of an arbitrary iterative application would be either expensive or highly inaccurate. Nevertheless, we know that the bulk plan performs better than the dependency plan, during the initial iterations. Moreover, we can easily measure how many elements of the solution set have changed value at the synchronization barrier of a bulk iteration. That is, in iteration k, we can measure λ_{k+1}. If we could also measure μ_{k+1}, we could use equation 5.11 to decide whether to switch to a dependency plan execution strategy in iteration $k+1$, at runtime. Note that, the candidate vertices for computation in iteration k should be roughly equal to the workset elements in the next iteration. This would be true if we expect that from the $\mu_k * |S|$ elements that are likely to change in iteration k, most of them *do* change. In fact, μ_k is an *upper bound* for λ_{k+1} and λ_k. Substituting $\lambda_k = \mu_k$, equation 5.11 becomes

$$3 * \lambda_k \leq \frac{1}{d+1} \tag{5.12}$$

This is an inequality that we can easily and efficiently compute during runtime.

5.5.2 Cost Model Implementation

When a Flink program is submitted for execution, the system creates an optimized execution plan, allocates resources and schedules the job for execution. This execution plan cannot be dynamically modified during job execution. To implement the cost model and be able to switch iteration technique during runtime, we use equation **??** inside a custom *convergence criterion*. In Flink, convergence criteria are implemented using aggregators. Aggregators are auxiliary iteration constructs that can be used to maintain simple global statistics during the iteration, such as the number of processed elements. At the end of each iteration step, local aggregates are combined to produce one global aggregate that represents the statistic across all parallel instances. Aggregator values computed during an iteration step are made available to the operators of the the next iteration step. Similarly, a convergence criterion is evaluated at the end of each iteration step. The convergence criterion decides whether the iteration should terminate, based on the value of a global aggregate.

We create a bulk execution plan chained to a dependency execution plan and we use the convergence criterion to implement the cost model. The optimization logic is depicted in Figure 5.12. If an incremental or delta plan is provided, it is chosen and executed until convergence. If no incremental or delta plan is available, the framework starts executing the bulk plan for the given application. At the end of iteration i of the bulk plan, the framework counts the number of updated vertices and uses equation 5.12 to compute whether $DepCost_{i+1} < BulkCost_{i+1}$. If the dependency plan cost is estimated to be lower than the bulk plan cost, the framework switches the execution strategy to a dependency plan in the next iteration.

5.5.3 Fixpoint API on Apache Flink

We build a fixpoint API as a thin layer on top of the Apache Flink Java DataSet API. Our API simplifies the development of fixpoint graph applications by allowing users to specify a single update function. We provide implementations for all four iteration techniques and let users decide whether to use a specific technique or activate the fixpoint cost model optimization that we described in the previous section. The fixpoint API hides the

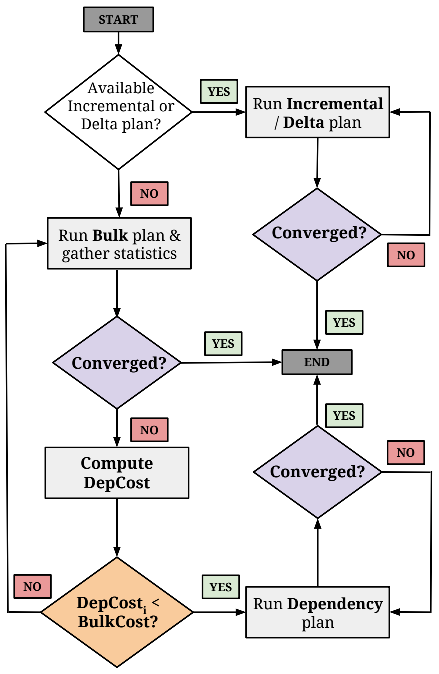


Figure 5.12 – The Fixpoint Cost Model Control Flow

complexity of delta iterations, neighborhood construction, candidate vertices calculation, cost model evaluation, and plan switching from the user.

The main components of the API are shown in Listing 5.1. Vertices are represented as a DataSet of `Tuple2` type, where the first element, $f0$, denotes the unique vertex identifier and the second element, $f1$, corresponds to the vertex value. Edges are represented with a DataSet of `Tuple3` type, where $f0$ is the identifier of the source vertex, $f1$ is the identifier of the target vertex, and the third element, $f2$, is an optional edge value. The API main abstraction is `FixedPointIteration`. It is implemented as a complex operator that consists of a delta iteration operator and encapsulates the vertices, the edges, and a user-defined a step function. `FixedPointIteration` only exposes one method, `iterate`, which takes the following parameters:

— `edges`: the data set of edges.

— `stepFunctIon`: the step function that defines how the state of vertices is to be updated.

— `maxIterations`: the maximum number of iterations.

— `execMode`: the execution mode. This parameter corresponds to the iteration techniques discussed in Section 5.2.1 and can take the values `BULK`, `INCREMENTAL`, `DELTA` or `COST_MODEL`. The `COST_MODEL` execution mode utilizes the cost-model optimization described in Section 5.5.1.

To use the fixpoint API, users only need to define the input vertices with their initial values, the edges data set, and an implementation of the `StepFunction` abstract class. The `StepFunction` requires the implementation of one method, `updateState()`.

Listing 5.1 Fixed Point API

```
FixedPointIteration iterate(
    DataSet<Tuple3> edges,
    StepFunction stepFunction,
    int maxIterations,
    ExecutionMode mode);

abstract class StepFunction {

    abstract DataSet<Tuple2> updateState(DataSet<Tuple4> inNeighbors);

    DataSet<Tuple2> deltaInput(DataSet<Tuple2> in_0, DataSet<Tuple2> in_1);

    Tuple2 deltaUpdate(Tuple2 previousValue, Tuple2 deltaValue);

    boolean deltaEquals(Tuple2 previousValue, Tuple2 currentValue);
}
```

This method contains the fixpoint iteration logic and defines how vertices update their values based on the state of their neighbors. The method receives one input DataSet, `inNeighbors`, which contains records of `Tuple4` type. The tuple fields correspond to the target vertex ID, the source vertex ID, the source vertex value, and the edge value, respectively. Note that records are intentionally not grouped by vertex ID. This design choice offers greater implementation flexibility than using a vertex-centric approach. Users can apply transformations to the `inNeighbors` DataSet to simulate different computation models. For example, grouping by source vertex ID and applying an aggregation would correspond to the vertex-centric model. Alternatively, applying a map transformation and then a combinable reduce would simulate the gather-sum-apply model.

The `StepFunction` class contains three additional methods that must be implemented when using the delta iteration technique.

— `deltaInput()` defines how to produce the input for the delta iteration plan. Its first parameter is the initial vertex dataset and the second parameter corresponds to the result DataSet after one first bulk iteration. The method returns a DataSet containing the initial delta values.

— `deltaUpdate()` defines how to produce the next DataSet of delta values. The *previousValue* DataSet contains the vertex deltas of the previous iterations and the *deltaValue* DataSet contaons the delta values computed in the current iteration.

— `deltaEquals()` defines when two delta values should be considered equal and it is used to control he convergence of the fixpoint algorithm.

Listing 5.2 Fixpoint Application Skeleton

```
// retrieve the Apache Flink execution environment
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Tuple2> vertices = env.read(...) // read the vertices input
DataSet<Tuple3> edges = env.read(...) // read the edges input

DataSet<Tuple2> result = vertices.runOperation(
  FixedPointIteration.iterate(
    edges, new UDF(), maxIterations, ExecutionMode.COST_MODEL));

result.write("path/to/result"); // store the output
env.execute("Fixpoint Application");
```

Examples

Here, we present some usage examples of the fixpoint API. Listing 5.2 shows the skeleton code for a typical fixpoint application. Like in any other Apache Flink application, we first retrieve the execution environment, which contains methods and utilities for reading input from external sources and writing output to sinks. We use those methods to read the vertex and edge data and represent them as Flink DataSets. Next, we call DataSet's `runOperation()` method, which allows running complex operators that are composed of multiple steps. In our case, the complex operator is created by `FixedPointIteration.iterate()`. Finally, we write the result to a sink, like a distributed file system, and we call `execute()`, which triggers the program execution.

Listing 5.3 Weakly Connected Components Step Function

```
updateState(inNeighbors):
  return inNeighbors
    .groupBy(0) // create neighborhood groups
    .min(2) // choose min label
    .project(0, 2); // project source vertex ID and min label
```

Listing 5.3 shows the step function implementation for the weakly connected components algorithm. The input DataSet is grouped by the first tuple element, which corresponds to the source vertex ID, resulting in creating groups of in-neighbors for each vertex. A *min()* aggregation is applied on each group, in order to choose the minimum label among all neighbors. The output contains the source vertex ID and the the minimum label value.

An implementation of a delta PageRank step function is shown in Listing 5.4. Here, we have implemented all four methods. First, we apply a map transformation to each neighbor record in order to compute the rank contribution of this neighbor. Then, records are grouped by source vertex ID and the ranks are summed up. The *deltaInput()* method simply computes the difference between the rank computed during the first bulk iteration and the initial rank value. *deltaUpdate()* adds the computed delta to the previous value

Listing 5.4 Delta PageRank StepFunction

```
updateState(neighbors):
  return neighbors
    .map(new PartialRank()) // compute partial ranks
    .groupBy(0).sum(1) // create neighborhood groups and sum up ranks
    .project(0, 1); // project the source vertex ID and new rank

deltaInput(in0, in1):
 // join the two inputs on vertex source ID
 initialDeltas = in0.join(in1).where(0).equalTo(0)
  .with(new JoinFunction() {
    join(Tuple2 initial, Tuple2 afterBulk, Collector out)
      // the initial delta is the difference of the two ranks
      out.collect(new Tuple2(initial.f0, afterBulk.f1 - initial.f1));
  }
  return initialDeltas;

deltaUpdate(prev, delta):
  // the new rank is the previous rank plus the delta value
  return (prev.f0, prevl.f1+delta.f1);

deltaEquals(prev, delta):
  // check if the rank has changed more than epsilon
  return Math.abs(prev.f1 - delta.f1) < epsilon;

PartialRank implements MapFunction:
  map(val):
    partialRank = 0.85+0.15*(val.f2/val.f3);
    return (val.f0, partialRank);
```

Listing 5.5 Label Propagation StepFunction

```
updateState(neighbors):
  return neighbors.map(v ->(v.f0, v.f2, 1))
     // groupBy vertexID, label
    .groupBy(0, 1).sum(2)
    // groupBy vertexID
    .groupBy(0).max(2).project(0, 1);
```

to compute the rank, and *deltaEquals()* checks if a rank has changed more than a user-specified threshold value.

Listing 5.5 shows an implementation of the Label Propagation step function. Initially, each neighbor record is mapped to a `Tuple3` containing the source vertex ID, the neighbor label, and the a constant equal to one. The tuples are then grouped by the composite key of vertex ID and label. A *sum()* aggregation is applied on each group, in order to produce the frequency of each label for each source vertex. The result is grouped by source vertex ID and the label with the highest frequency is chosen as the new value. The same algorithm can alternatively be implemented in a vertex-centric way similar to that of the weakly connected components implementation of Listing 5.3. In this case, a `GroupReduceFunction` needs to be applied on each neighborhood group. Inside this function, each vertex would have to store its neighbors' labels (e.g. in a HashMap) and compute the one that appears more frequently.

5.6 Cost Model Evaluation

Here, we present evaluation results for our fixpoint framework and cost model. Using the fixpoint API, we implement the Label Propagation and Community Detection algorithms. We run these applications on the graphs of Table 5.1 until convergence, in `BULK`, `DEPENDENCY`, and `COST_MODEL` execution mode. We use the same environment as in Section 5.4. For each run, we measure the total execution time, the time per iteration, and the values of λ and μ. We also measure the number of updates that each vertex generates for its neighbors in each iteration. In a vertex-centric model, these updates would correspond to the number of messages exchanged per superstep.

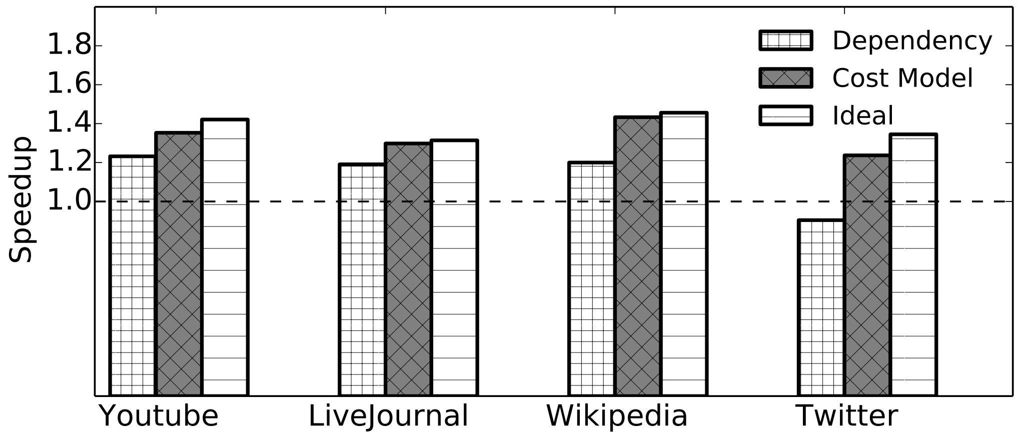


Figure 5.13 – Runtime speedup for Label Propagation when using the dependency plan or the cost model execution as compared to a bulk execution. The third bar corresponds to the ideal speedup we could get with an exact cost model.

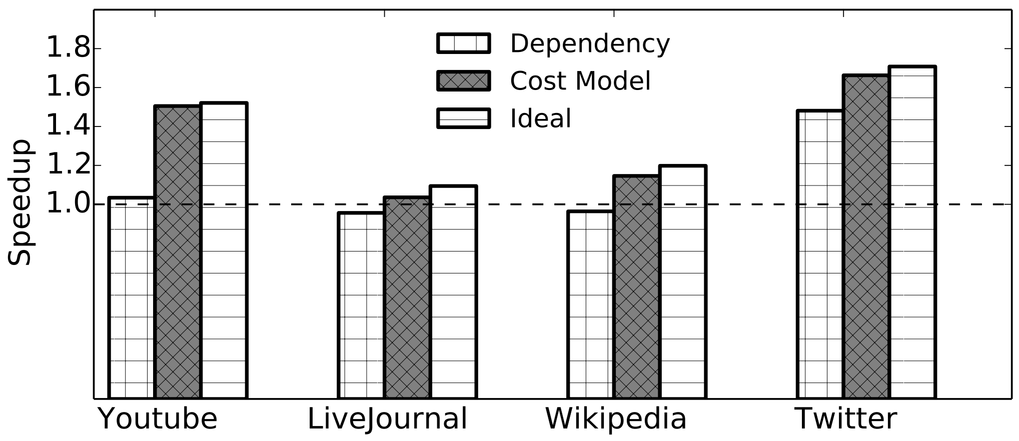


Figure 5.14 – Runtime speedup for Community Detection when using the dependency plan or the cost model execution as compared to a bulk execution. The third bar corresponds to the ideal speedup we could get with an exact cost model.

Figures 5.13 and 5.14 show the runtime speedup when using the dependency or cost model execution mode, as compared to when using the bulk execution mode, for Label Propagation and Community Detection, respectively. In the plots, the third bar corresponds to the *ideal* speedup we would achieve, if we used an exact cost model. That is, the ideal speedup corresponds to a perfect optimizer that would switch execution plan *immediately* after the cost of the dependency technique would become lower than the cost of the bulk technique. Knowing the execution time per iteration, we can easily compute the ideal speedup. Our cost model uses the approximation of Equation 5.12, thus, we expect it to switch plans a few iterations later than an ideal cost model would.

Our results show that even using the dependency plan for the complete execution is beneficial in some cases. For the Label Propagation algorithm, the dependency execution mode yields lower total execution time than the bulk execution mode for three input graphs, and for two graphs in the case of Community Detection. However, as we saw in Section 5.4, the dependency plan imposes a significant overhead in early iterations, when many vertices update their values. Using the cost model execution mode, we can successfully decide when this overhead has become small enough and switch from a bulk execution strategy to a cheaper dependency execution strategy. Using the cost model execution mode, we see speedup of 1.1-1.7x for all input graphs and for both applications. Note that the execution time measured in the cost model case *does* include any overhead imposed by monitoring the value of λ, recalculating the dependency cost, and evaluating the cost model convergence criterion at the end of each iteration. Finally, by comparing with the ideal speedup, we verify that our approximate cost model performs almost as well as an ideal cost model would. Indeed, the additional benefit that a perfect cost model would provide is small in all cases.

Speedup is a good metric for assessing our optimization's performance, but it is a metric naturally dependent on the underlying system's properties and implementation details. What is more interesting to consider is how much communication load we can save by using the fixpoint cost model. Figure 5.15 shows the communication savings for the considered algorithms when using a cost model execution strategy over a bulk execution strategy. We measure communication load by counting the number of updates generated per iteration. This number is a equivalent to the number of exchanged messages in a vertex-centric model. We see that communication savings are substantial, ranging from 19% for community detection on the Wikipedia graph to 54% for the same algorithm on the Youtube graph.

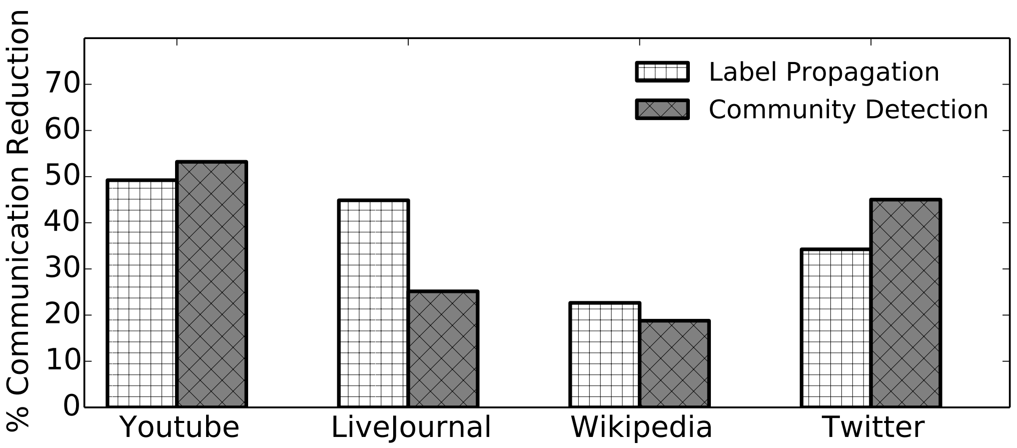


Figure 5.15 – Percentage of communication reduction when using the cost model. We use the number of updated generated per vertex as a measure of communication load. In a vertex-centric model, this would correspond to exchanged messages.

5.7 Conclusion

Iterative value-propagation algorithms often expose *non-uniform* convergence behavior. In this chapter, we have shown that failure to detect this behavior often leads to redundant computations and communication. We have experimentally observed this asymmetrical converge and we have presented ways to exploit. We have presented a taxonomy of optimizations for iterative fixpoint algorithms. We have experimentally observed asymmetrical converge of iterative graph algorithms and we have presented ways to exploit this behavior. We have implemented template execution plans, using common dataflow operators and we have presented experimental evaluation, using a common runtime. Our results demonstrate order of magnitude gains in execution time, when the optimized plans are used. Using our evaluation results, we built a framework for fixpoint algorithms on top of Apache Flink. Our framework consists of a simple declarative API for fixpoint applications and a cost-based optimizer that chooses the best iterative execution plan at runtime. We propose an approximate cost model, which can accurately predict the cost of subsequent iterations, by monitoring the number of active vertices. We have shown that using our cost model, we can achieve substantial computation and communication savings. Our approximate cost model demonstrates performance close to that of an ideal cost model. To the best of our knowledge, this is the first work that builds a distributed fixpoint iteration framework, capable of switching execution strategy, based on a cost model, during run time. Nevertheless, our technique is general and easy to integrate in any distributed dataflow execution engine.

Chapter 6

Semi-metricity in Large-Scale Graph Analysis

In the previous chapter, we explored how algorithmic properties can be exploited to optimize the execution of graph applications like Label Propagation. In this chapter, we explore how properties of the graph data can be leveraged in order to reduce execution time. Specifically, we propose to use the concepts of *semi-metricity* [154] and the *metric backbone*, introduced in [58], for efficient large-scale graph analysis. In a weighted graph, semi-metric edges are direct links for which there exists an indirect shorter path. In the example of Figure 6.1, the dashed lines represent such semi-metric edges. The metric backbone is the subgraph of a weighted graph that includes no semi-metric edges. In Figure 6.1, the solid lines represent the metric backbone of the depicted social graph. Effectively, the metric backbone is a reduced representation of a graph, that preserves information about shortest paths. This property has been used to improve recommendation algorithms [155, 153, 168] and more recently, to improve the modularity in community detection [169].

In this chapter, we explore how the performance of various large-scale graph analysis tasks can benefit from the concept of the metric backbone. In particular, we apply the metric backbone concept in the context of large-scale graph analysis systems, such as graph databases [12] and batch processing systems [53, 125]. First, we show that, for applications that depend explicitly on the calculation of shortest paths, we can get exact answers, but significantly improve performance by computing on the reduced metric backbone instead. Second, even when shortest paths are not explicitly used, such as when performing reachability queries, the metric backbone can still yield correct answers faster, by reducing the amount of paths that must be explored. Third, we study algorithms, for which the metric backbone does not yield exact answers, but an approximation. Here, we consider PageRank and show that executing the algorithm on the metric backbone produces a good approximation, while considerably improving efficiency. In particular cases, the edges removed from the backbone may act as noise for specific algorithms. Running these algorithms on the backbone can often give better quality results. For example, it has been shown that the metric backbone can improve the modularity of community detection, on weighted

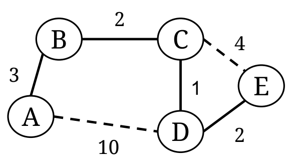


Figure 6.1 – An example graph representing a social network. The edge weights represent the proximity of the social tie. Dashed lines represent semi-metric edges, indicating there is a shorter, indirect path between two nodes.

graphs [169].

Despite its usefulness, the metric backbone has not been used in large-scale graph analysis yet. This is mainly due to the fact that the calculation of the backbone itself, on big graphs, is challenging. Computing the metric backbone requires solving the APSP (All-Pairs-Shortest-Paths) problem and storing $O(N^2)$ paths, which is prohibitive for large graphs. We address this challenge in two ways. First, we provide an algorithm for the calculation of the backbone that does not require the computation of APSP. Our algorithm starts by detecting and removing only those semi-metric edges that violate the triangle inequality. Subsequently, it iteratively labels metric edges by performing short breadth-first searches. Second, we show that even an approximation of the metric backbone, where only the semi-metric triangles have been removed, reduces the size of the original graph significantly.

We start by describing the concept of the metric backbone. To motivate its use in graph analysis, we first discuss how different graph analysis tasks may benefit from the concept at the algorithm level. Then, we analyze a variety of real data sets, to validate that graphs exhibit high degree of semi-metricity, making this approach effective in real scenarios. Further, we outline how we can apply the concept in graph management systems.

6.1 The metric backbone

The metric backbone is introduced in [58], primarily to improve the accuracy of community detection algorithms [169]. To describe it more formally, let us first introduce the necessary notation. Let $G = (V, E)$ be a graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges, with $(u, v) \in E$ denoting an edge from u to v. We use $d(u, v)$ to denote the weight of an edge (u, v), representing a distance. The weight may represent any application-defined distance metric imposed on the graph, such as communication latency or euclidean distance. Finally, given an acyclic path p, $d(p)$ denotes the distance of the path.

The utility of the metric backbone is based on the role of *semi-metric* edges in a graph. In a weighted graph, we say that an edge between two nodes is semi-metric, if there exists an indirect path between these two nodes, with a shorter distance.

Definition 1. *An edge (u, v) is n^{th}-*order semi-metric *if there exists an alternative path, $u, x_1, ..., x_n, v$, with $n + 1$ edges $(u, x_1), ..., (x_n, v) \in E$, such that $d(u, v) > d(u, x_1) +$*

$... + d(x_n, v)$.

For instance, in Figure 6.1, edge CE is 1^{st}-order semi-metric and edge AD is 2^{nd}-order semi-metric. The metric backbone is essentially a subgraph of the original graph that contains no semi-metric edges.

Definition 2. *Given a weighted graph, where weights represent non-negative distances, the metric backbone is the minimum subgraph that preserves the shortest paths of the original graph.*

In Figure 6.1 the solid lines represent the metric backbone of the depicted social graph.

The utility of the backbone is not limited to weighted graphs where weights represent distances. Often, instead of a distance metric, relations in graphs are described by a similarity metric [169, 50]. For instance, the Jaccard index [94] is a popular similarity metric that relies on the number of common neighbors between two nodes in a graph. Alternatively, in the context of an social network, similarity is sometimes related to the amount of interaction between two users, like the number of messages exchanged. In such cases, we can transform similarity to distance, through appropriate functions [169, 50], and still take advantage of the metric backbone for analysis [1].

Once we have computed the metric backbone, we can use it to calculate metrics based on shortest distances, since it maintains this information. Additionally, recent work has shown that removing semi-metric edges from a graph also allows us to perform community detection [169] or recommendations [153] with improved accuracy.

6.2 Semi-metricity in real graphs

The utility of the metric backbone is based on the observation that many real-world graphs exhibit high degree of semi-metricity. As it has been shown in various contexts, especially in social networks, indirect connections are often stronger than direct ones. For instance, OSN interactions between users, a common metric of social proximity [84], are often more frequent between users who are not directly connected [36, 182]. In fact, this principle has been used, for example, to predict information propagation paths [197], to improve link prediction in OSNs, to provide better recommendations and even to design more efficient storage systems that back OSNs [36].

In general, different real-world graphs present different degrees of semi-metricity. Semi-metricity also varies with the distance metric imposed on the graph. To study the practicality of our approach, we analyze a variety of real-world graph datasets, measuring the degree of semi-metricity. We present results for graphs in several domains, such as OSN, web, authorship, air traffic and biological graphs. We measure semi-metricity under various commonly used metrics, such as the Jaccard [94] and the Adamic-Adar [18] metrics.

In Table 6.1, we describe the datasets we analyzed and summarize the results. The degree of semi-metricity ranges from 9% to 88% depending on the dataset and metric. The

1. While there are various functions available, a simple and commonly used function to convert a similarity metric x to distance is $\varphi(x) = \frac{1}{x} - 1$.

Graph	$\|V\|$	$\|E\|$	**metric**	%
Facebook [2]	190M	49.9B	Custom	26.5%
Twitter [111]	40M	1.5B	Jaccard	39%
Tuenti [14]	12M	685M	Jaccard	59%
LiveJournal [25]	4.8M	34M	Jaccard	40%
NotreDame [22]	0.3M	1.5M	Jaccard Adamic	45% 29%
DBLP [190]	318K	1M	Jaccard Adamic	23% 9%
Twitter-ego [126]	81K	1.7M	Jaccard Adamic	57% 39%
Movielens [11]	1.6K	1.9M	Jaccard	88%
Facebook [143]	1K	143K	#messages message size	78% 77%
US-Airports [56]	0.5K	6K	#passengers	72%
C-Elegans [181]	0.3K	2.3K	#connections	17%

Table 6.1 – The percentage of semimetric edges on various real graphs under different similarity metrics. In the small Facebook dataset, we use the number of messages or size of messages exchanged between users to measure similarity. In the US-Airports graph we measure similarity as the number of passengers that travel between cities.

Movielens [11] movie preference graph exhibits the highest semi-metricity, while among the analyzed OSN graphs, Tuenti [14] has the highest semi-metricity of 59%. Further, we see that the same graph may exhibit different semi-metricity for different distance metrics. For instance, the Jaccard similarity metric [94] typically results in more semi-metric edges than the Adamic-Adar metric [18].

Further, in Figure 6.2 we plot the percentage of semi-metric edges for different orders of semi-metricity, for some of the networks of Table 6.1. Notice that, in most of the graphs, the vast majority of the semi-metric edges are 1^{st}-order semi-metric. In other words, there are *few* indirect paths with three or more edges that are shorter than any direct edges. Previous work has also identified that the strength of indirect social connections decreases with the length [76, 54].

This analysis reveals an opportunity. First, based on Table 6.1, we see that in practice, we can run a variety of analytical tasks on a graph that is significantly reduced in size, in some cases, with more than half the edges removed. Indeed, when we apply the concept in large-scale graph management systems later in this chapter, even seemingly modest degrees of semi-metricity have a significant impact on application performance. Second, the analysis of Figure 6.2 shows that we can compute a good approximation of the metric backbone by removing only the 1^{st}-order semi-metric edges. We use this intuition to guide the design

2. A subgraph of the Facebook social network representing a geographic area. The graph is weighed with a custom similarity score that integrates a number of user features.

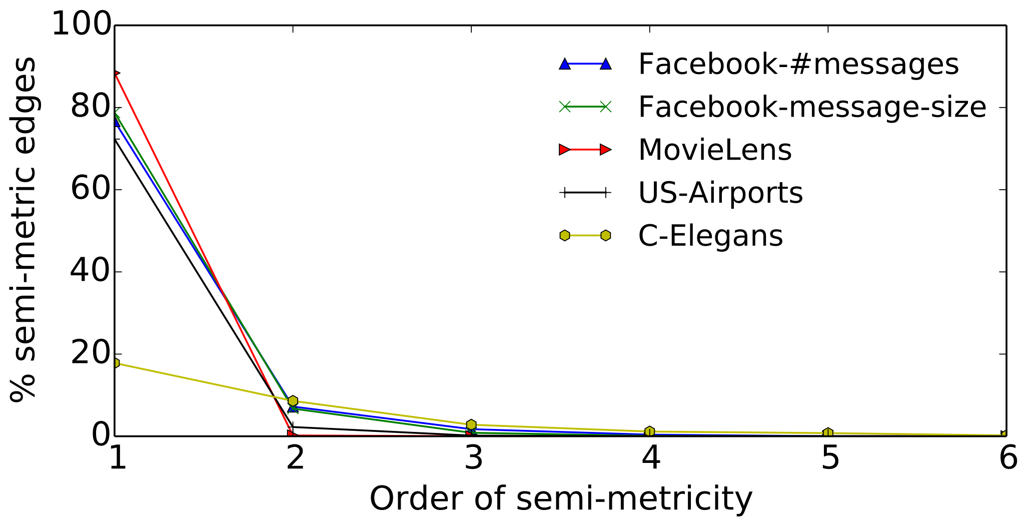


Figure 6.2 – Percentage of semi-metric edges over their order of semi-metricity. For most of the graphs, the majority of edges are 1st-order semi-metric edges.

of the algorithm for the computation of the metric backbone. This approximation is not the optimal subgraph that gives us the shortest paths, but it is very close to the optimal. Note that the approximate metric backbone preserves the connectivity and reachability properties just like the exact metric backbone and it generates the same shortest paths distribution.

6.3 Exploiting Semi-metricity in Graph Algorithms

Graph processing may benefit from the metric backbone in different ways. Here, we divide graph algorithms in two classes and give examples for each class. Note that the classification does not depend on the framework or model in which the corresponding algorithms may be programmed and executed. We discuss framework-specific impact in Section 6.3.1.

We summarize the classification in Table 6.2. Class A consists of algorithms that we can run unmodified on the metric backbone and get the *exact* same answer, as when running the algorithm on the original graph. Examples include the calculation of shortest distances or algorithms that depend on computing shortest distances, like betweenness centrality. The metric backbone also maintains the connectivity of the graph, therefore, we can compute exact answers for connected components and reachability queries.

Class B includes algorithms that we can run unmodified on the metric backbone but may return an *approximation* of the metric they are intended to calculate. Examples in this category include PageRank and various community detection algorithms [169, 75]. We empirically validate the accuracy of PageRank approximation when using the metric backbone in Section 6.6.

The metric backbone reduces the information and thus, there are graph metrics for

Class	Description	Examples
A	Algorithms we can run unmodified and produce exact result.	Shortest weighted paths, betweeness centrality, closeness centrality, connected components, radius, reachability queries.
B	Algorithms we can run unmodified and produce an approximation.	PageRank, eigenvector centrality, random walks, community detection, clustering.

Table 6.2 – A classification of graph algorithms and metrics that may be computed on the metric backbone. For each class we provide a list of example algorithms.

which the metric backbone may yield highly inaccurate answers. For instance, we cannot use the metric backbone to calculate the unweighted shortest distances, as it will overestimate the distance for all the pairs of nodes connected by semi-metric edges.

Finally, note that this is not meant to be an exhaustive list of the algorithms we can or cannot use with the metric backbone. We believe that the concept of the metric backbone opens up an opportunity to define more metrics and characterize how to benefit from it.

6.3.1 Applications in Graph Management Systems

While the benefit of the metric backbone is not specific to a computation model or framework, it manifests in distinct ways when used in the context of graph management systems. We use the metric backbone to improve the performance of two types of systems, graph databases and distributed batch processing systems.

Graph databases. Graph databases are used to store and query large graphs. They are optimized for traversals, reachability, and pattern matching queries [12]. For example, users can query for paths that satisfy criteria, such as length or the properties of the nodes. For several queries, the metric backbone preserves the semantics. At the same time, executing a query on the metric backbone only, reduces the path search space and may provide significant query speedups. We apply this technique manually, by re-writing queries to use the metric backbone but we envision that a closer integration with automatic query rewriting will allow for more optimizations and a more user-friendly interface. We evaluate the impact in Section 6.6.3.

Batch processing systems. We consider large-scale graph processing systems, such as Pregel [125, 53] and Graphlab [122]. In such systems, graph algorithms are implemented as parallel per-vertex computations and typically, vertices communicate by exchanging messages. This communication usually occurs along the edges of the graph. In these systems, the CPU and memory requirements depend on the number of messages that have to be processed, which is typically proportional to the number of the edges of the graph. In Section 6.6.4, we validate that by reducing the edges of a graph, we reduce communication overhead and resource requirements, eventually improving runtime performance.

Graph compression The metric backbone can also be used as a lossy compression mechanism, as the amount of semi-metric edges directly translates to storage reduction.

The last column of Table 6.1 corresponds to size reduction, when the backbone is used in the place of the original graph.

6.3.2 Discussion

For algorithms that do not depend on the edge weights of a graph, it might not be clear whether they could benefit from the metric backbone. Actually, iterative applications that use the graph structure to propagate information, might require more iterations to converge, when run on top of the metric backbone. For example, let us consider the Connected Components problem. In its typical distributed implementation, in every iteration, a node receives the IDs of its neighbors, adopts the minimum of these IDs and, if its value has changed since the previous iteration, it propagates the new value to its neighbors. Computation stops when none of the nodes changes value. This computation does not utilize the edge weights of the graph, but only the graph structure. The maximum number of iterations necessary for convergence is equal to the maximum graph diameter + 1. When removing edges to generate the metric backbone, the *absolute* paths between some nodes become longer; the removed edges might be *shortcuts* in the unweighted graph. Thus, by removing them, we increase the graph diameter and consequently, the number of iterations required for convergence. However, as we show in Section 6.6, such applications can still benefit from the metric backbone. Even if the algorithm does not depend on the edge weights, when removing a large amount of edges, we notably decrease the communication required, thus, speeding up execution.

6.4 Computing the Metric Backbone

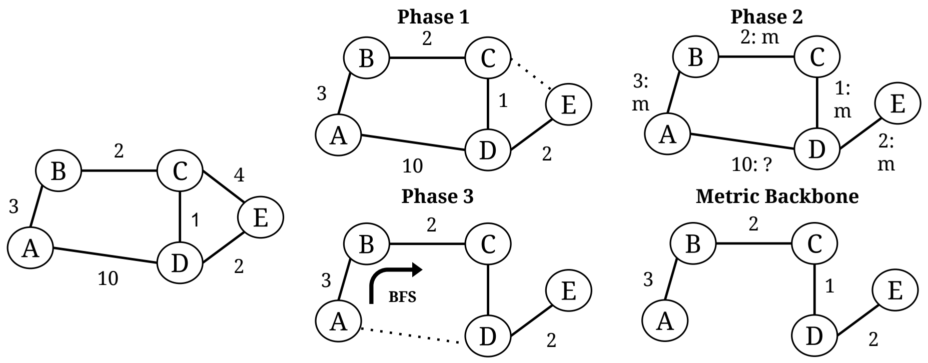


Figure 6.3 – The three phases of the backbone calculation. In the first phase, the algorithms removes 1st-order semi-metric edges, in this case edge CE, marked with a dotted line. In the second phase, the algorithm identifies metric edges, within the two-hop neighborhood of each node. Here, edges AB, BC, CD and DE are identified as metric, while edge AD remains unlabeled. In the third phase, the algorithm discovers all remaining higher-order semi-metric edges, by running a BFS for each unlabeled edge (in this case AD).

In this section, we describe the algorithm for computing the metric backbone. We show how to implement it in an efficient and scalable manner in Section 6.5. The algorithm we propose in this paper assumes a static graph that does not change over time. We believe that incremental maintenance of the backbone is an important topic and we plan to address it in future work. In Section 6.4.4, we provide a brief description of an incremental algorithm and explain how changes in the original graph affect the backbone. We target undirected graphs with *symmetric* relations. We focus on undirected graphs as they appear naturally in the scenarios we examined. For instance, commonly used similarity metrics, like Adamic-Adar, are symmetric. Note, however, that the concept of the metric backbone applies to directed graphs with asymmetric relations too. For a detailed description of the conditions under which an edge in a directed graph is semi-metric, we refer the reader to [169].

6.4.1 Naive algorithm

The most straight-forward approach to computing the metric backbone is to identify semi-metric edges through multiple breadth-first searches (BFS): to test an edge (u, v) for semi-metricity, we start a breadth first search from node u and we accumulate path weights, while visiting new nodes. During the BFS, if vertex v is visited, we check whether the weight of the newly discovered path is lower than $d(u, v)$. If it is, then (u, v) is semi-metric. Otherwise, we stop exploring towards this direction. If the BFS finishes without encountering vertex v, then there is no alternative path from u to v, and thus, (u, v) is metric. This process is essentially equivalent to solving the APSP problem.

This approach incurs high overhead and does not scale to large graphs. Even if we start several BFSs in parallel, a lot of communication and substantial storage is required to keep track of the visited paths and their weights. Next, we present a three-phase algorithm, that uses optimizations and empirical heuristics to considerably speed up the computation of the metric backbone. We show that, by using simple scalable steps, we can identify the majority of the semi-metric edges and significantly prune the paths that ultimately need to be explored by a BFS.

6.4.2 Core algorithm

We divide the algorithm in three phases. In the first phase, we discover and remove all the 1st-order semi-metric edges. In the second phase, we identify metric edges within the two-hop neighborhood of each node in the induced subgraph. Finally, we discover remaining semi-metric edges with breadth-first searches and remove them to produce the metric backbone. Figure 6.3 illustrates the algorithm phases.

We divide the algorithm in these three phases for different reasons. First, we can easily parallelize and scale the removal of 1st-order semi-metric edges, by detecting triangles. Specifically, in Section 6.5.1, we show how to implement this phase on top of a distributed graph processing system. Second, as we already saw in the analysis results of Section 6.2, the largest fraction of semi-metric edges are typically *1st-order* semi-metric. This allows us to significantly reduce the size of the graph early in the process and provide a fair and practical approximation of the metric backbone, after having executed just the first phase.

Third, the second phase can exploit the knowledge that there are no 1st-order semi-metric edges to efficiently discover metric edges. Next, we describe the three phases in detail.

In the first phase, for every triangle in the graph, we test whether one of its edges violates the triangle inequality. Such edges are by definition semi-metric, and we can remove them from the graph. Triangle enumeration is a well-studied problem in graph theory and several algorithms have been proposed for its solution [93, 162, 55]. Here, we use a variation of the node-iterator algorithm to demonstrate the removal of first-order semi-metric edges. Algorithm 24 shows the pseudocode for this phase.

Algorithm 24 Detect semi-metric edges in triangles.

```
1: Input: the set of vertices, $V$ and the set of edges, $E$
2: for all $v \in V$ do                                   ▷ Iterate over all vertices
3:     for all $x, y \in neighbors(v)$ do
4:         if $x, y \in E$ then                           ▷ Check if there exists a triangle
5:             if $d(x,y) + d(y,v) < d(x,v)$ then
6:                 remove $(x,v)$
7:                 remove $(v,x)$
```

In the second phase, we reverse the logic of the algorithm and aim to identify metric edges. Each node exploits information in its two-hop neighborhood to reason about the semi-metricity of its adjacent edges. The initialization of this phase is based on the following proposition:

Proposition 1. *The lowest-weight edge of every vertex in a graph, belongs to the metric backbone.*

Let v be a vertex in G and $(v, u_1), (v, u_2), ..., (v, u_k)$ be $v's$ edges, in increasing weight order. If the edge with the lowest weight, (v, u_1), is semi-metric, then there exists a path $p = (v, u_x, ..., u_1)$, such that $d(p) < d(v, u_1)$. This cannot be true, since $d(v, u_1) \leq d(v, u_x), \forall x \neq 1$.

For example, consider node C of the network in Figure 6.1. Its lowest-weight edge, CD, belongs to the metric backbone: any indirect path between C and D would contain either edge CB or edge CE and thus, have a larger weight than the direct edge.

After each node has marked its lowest-weight edges as metric, it checks whether it can reason about the semi-metricity of the rest of its edges, by comparing their weights to the minimum weights of its two-hop paths, which contain metric edges. This process is shown in Figure 6.4. Node u decides whether edge e_1 is metric, by checking the weights of the two-hop paths along its metric edges, m_1 and m_2. Any alternative path would include edges e_2 or e_3, which already have a larger weight than e_1. If u discovers that the minimum two-hop paths containing its metric edges have larger distance than the direct edge e_1, then e_1 is metric.

Proposition 2. *Given a node with metric edges $m_1, m_2, ..., m_k$ and unlabeled edges $e_1, e_2, ..., e_l$, in increasing weight order, edge e_1 is metric if its weight is lower than all the weights of the node's two-hop paths, which contain edges $m_1, m_2, ..., m_k$.*

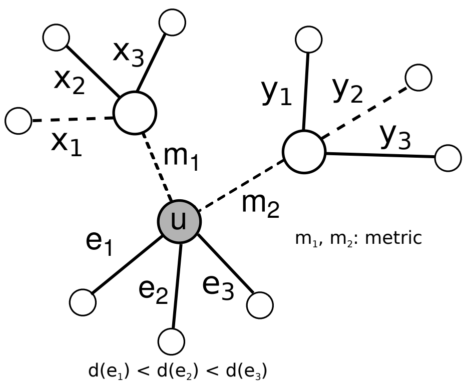


Figure 6.4 – The second phase of the algorithm. Edge indices are ordered by increasing weight value. u has already discovered that m_1 and m_2 are metric. The minimum paths containing its metric edges are shown with dashed lines. If edge e_1 has a lower weight than both these paths, then e_1 is metric.

Proof. Let edge e_1 of Figure 6.4 be the edge in consideration and let v be the label of its target node. According to the proposition,

$$d(e_1) < d(m_1) + d(x_1) \tag{6.1}$$

and

$$d(e_1) < d(m_2) + d(y_1) \tag{6.2}$$

We will assume that e_1 is semi-metric and prove that this cannot be true. If e_1 is semi-metric, then there exists a path p, from v to u, which does not contain e_1, has length at least two and its weight is lower than the weight of e_1, i.e.

$$d(p) < d(e_1). \tag{6.3}$$

Obviously, this path cannot contain e_2 and e_3, since $d(e_1) < d(e_2) < d(e_3)$. Thus, p passes through m_1 or m_2. If p passes through m_1, then its lowest weight possible would be $d(m_1) + d(x_1)$. According to 6.1, e_1 has a weight smaller than the lowest possible weight of a path passing through m_1, thus, equation 6.3 cannot be true. We arrive at the same contradiction assuming that p passes through m_2. Therefore, e_1 is metric. □

Algorithm 25 shows the pseudocode for the second phase. We assume a partial ordering on the edge weights and that a node's access to its edges respects this order. We represent the edges of a node v as an ordered set, U_v, with two additional methods, $first$ and $remove$. If the set is not empty, a call to $first$ will return the edge in the set with the minimum weight. If the set contains more than one edge with the minimum weight, $first$ will return all of them. A call to $remove$ will return the same edge(s) as a call to $first$, while also removing them from the set.

When no further local metric edges can be found, we proceed to the third phase, where we characterize the remaining unlabeled edges, by performing breadth first search. For each unlabeled edge (u, v), we start a BFS from node u. If the BFS discovers an indirect path from u to v with a lower weight than the weight of the direct edge, then (u, v) is semi-metric. Otherwise, if the BFS finishes without finding a shorter indirect path, then (u, v) is metric. We present the pseudocode for the third phase in Algorithm 26. The method $bfs(u, v)$ returns the set of all indirect paths, starting from node u and ending in node v.

Algorithm 25 Identify local metric edges.

```
Input: the set of vertices, $V$ and the set of edges, $E$
$M \leftarrow \emptyset$                                  ▷ Metric edges found so far
for all $v \in V$ do                                      ▷ Iterate over all vertices
    $U_v \leftarrow E_v$                                  ▷ All edges are initially unlabeled
    $W \leftarrow \emptyset$                              ▷ Set of weights for comparison
    $metric \leftarrow TRUE$
    $M \leftarrow M \cup (U_v.remove)$                    ▷ See Proposition 1
    while $U_v \neq \emptyset$ do
        $e \leftarrow U_v.remove$
        for all $m \in M$ do
            $x \leftarrow m.target$                       ▷ The target node of m
            $w_x = d(v, x) + d(U_x.first)$                ▷ The min 2-hop
                                                          ▷ path weight, that includes m
            $W \leftarrow W \cup w_x$
        for all $w \in W$ do
            if $d(e) > w$ then
                $metric \leftarrow FALSE$
                break
        if $metric$ then                                  ▷ All 2-step paths were larger
            $M \leftarrow M \cup e$                       ▷ e is metric
            $W \leftarrow \emptyset$
        else
            return $M$                                    ▷ Cannot label further edges
```

Based on the results of Figure 6.2, we expect the majority of metric edges to be identified during the first phase. Moreover, since most of the semi-metric edges are discovered during the first phase, we also expect the BFSs to finish early. Indeed, in the same figure, we observe that this is true for all the networks we analyze. In the worst case, a total of 6 hops is required to label all the edges of the graph.

6.4.3 Complexity analysis discussion

While our algorithm does not lower the worst-case complexity of the naive algorithm, our heuristic makes the computation practical for large-scale graphs. Here, we analyze the conditions under which our approach is faster than solving APSP.

Let U be the set of vertices which are sources of unlabeled edges, after removing semi-metric triangles. U is the upper bound of the number of BFSs we have to run, after semi-

Algorithm 26 Characterize remaining unlabeled edges

```
1: **Input:** the set of unlabeled edges, $U$
2: **for all** $(u, v) \in U$ **do**
3:     $label \leftarrow METRIC$
4:     $P_u \leftarrow bfs(u, v)$                        ▷ Indirect paths from $u$ to $v$
5:     **for all** $p \in P$ **do**
6:         **if** $d(p) < d(u, v)$ **then**               ▷ Shorter indirect path found
7:             $label \leftarrow SEMIMETRIC$
8:             **break**
9:     **if** $label = SEMIMETRIC$ **then**
10:        $remove(u, v)$
11:    **else**
12:        $label \leftarrow METRIC$
```

metric triangle removal. The worst-case complexity of the metric-backbone algorithm is *complexity of semi-metric triangles + complexity of computing shortest paths (or BFSs) on U*. The worst-case complexity of basic triangle listing algorithms is $\Theta(E * d_{max})$, where d_{max} is the maximum vertex degree [116]. This leads to $\Theta(E * V)$ in the worst case. Thus, computing the metric backbone has a worst-case complexity of $\Theta(E * V) + O(U^3)$. If the graph is highly metric, $U \approx V$, and the worst case is equivalent to running APSP, i.e. $O(V^3)$. If the graph is highly semi-metric (and most of the semi-metric edges are discovered in the first step), then $U \lll V$ and the practical run time is lower.

6.4.4 Maintaining the backbone incrementally

We only consider one-at-a-time edge removals and edge additions. We can simulate all other graph structure and value changes, using edge additions and removals. Changing the weight of an edge is equivalent to removing the edge and then adding the same edge, but with the new weight. Adding a node with no edges results in simply adding the new vertex to the metric backbone. Adding a node with one or more edges is equivalent to a combination of adding a single vertex and each of the edges one by one. Finally, removing a node is equivalent to first removing its edges one by one and then removing the vertex.

— *Edge Removal*: If the edge to be removed is semi-metric, the metric backbone does not change. The edge can be simply removed from the original graph. If the edge to be removed is metric, some of the semi-metric edges might now become metric. Note that the metric edges do not get affected. Let (u, v) be the metric edge to be removed. A semi-metric edge (x, y) is potentially affected by the removal of (u, v) if there is a path $p = (x, ..., u, v, ...y)$ in G, such that $d(p) < d(x, y)$. Thus, the only edges that might be affected are the semi-metric edges in indirect paths from u to v. We check these edges with the following steps. First, remove edge (u, v) from G. Then, for every semi-metric edge in all remaining paths from u to v, execute the backbone algorithm, starting from phase 2.

— *Edge Addition*: When adding an edge (u, v) to the original graph, we first check whether this edge is semi-metric. In order to do so, we can easily find the current shortest path from u to v, using the metric backbone. If the weight of the new edge is larger than the current shortest path, then this edge is semi-metric and the metric backbone does not change. If the edge weight of the new edge is lower than the current shortest path from u to v, then the edge is metric. In this case, some of the metric edges of the backbone might become semi-metric, if a shorter path is introduced through the new edge. Again, the only edges that we need to consider are the ones on the indirect paths from node u to node v.

Note that the edges that change from semi-metric to metric (or vice-versa), in one of the above cases, do not cause further changes in the metric backbone, since all weights remain unchanged.

Proposition 3. *When removing a metric edge (u, v) from the original graph G, the only edges that might be affected are the semi-metric edges in indirect paths from u to v.*

Proof. We prove this statement using the method of contradiction. Proving that no metric edges are affected is trivial. Let G' denote the graph after the removal of edge (u, v) and (x, y) be a semi-metric edge in G, which does not participate in any path from u to v and which becomes metric in G'. Since (x, y) is semi-metric in G, there exist one or more a paths $x, v_1, ..., v_k, y$ in G, such that $d(x, v_1, ..., v_k, y) < d(x, y)$. Let p be the minimal of these paths. Given that (x, y) is metric in G', its weight is smaller than the weights of all indirect paths from x to y, including p. Thus, p in G' has a smaller weight than p in G. Since (x, y) does not participate in any path from u to v, (u, v) cannot belong to p. Therefore, path p in G and G' are identical and we have arrived at a contradiction. □

6.5 Vertex-Centric Implementation of the Metric Backbone Algorithm

To be usable in real scenarios, the computation of the metric backbone must be practical for large graphs. We have implemented the computation of the backbone on top of the Pregel programming model [125], and specifically the Apache Giraph system [53]. Pregel-like platforms [125, 53, 81, 122] are widely adopted and are common components of data centers. We have made the implementation of the algorithm available as open source [3].

Here, we describe our vertex-centric implementation in Giraph. It consists of three phases: (i) detection of the 1^{st}-order semi-metric edges (ii) iterative labeling of local metric edges and (iii) labeling of all remaining metric edges.

6.5.1 Phase 1: Detect semi-metric triangles

This phase is based on the BSP-model algorithm for triangle detection, as described in [66] for a weighted, undirected graph with total ordering on the vertex IDs. The algorithm consists of four supersteps and discovers each triangle exactly once. In the first

3. Implementation available at http://grafos.ml

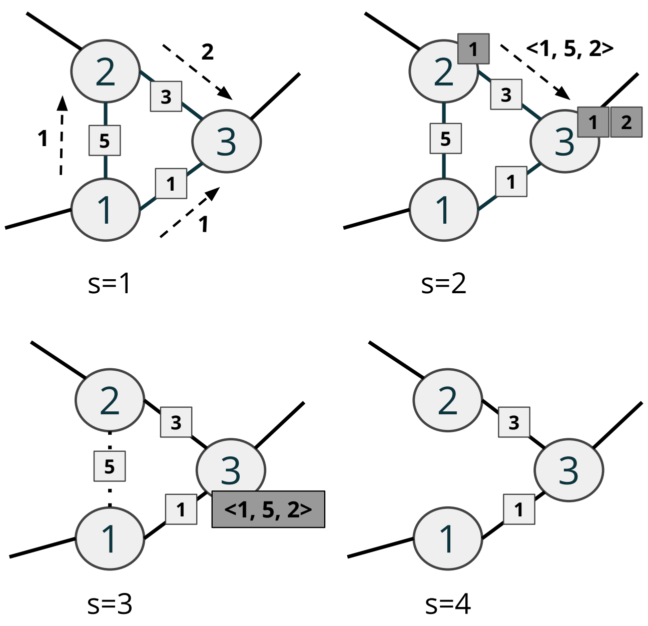


Figure 6.5 – Detecting semi-metric triangles in the vertex-centric model. Messages in transit are shown with dotted arrows. Received messages are shown in dark grey boxes. At the third superstep, vertex 3 detects the semi-metric triangle and issues a request to remove edge 1-2.

superstep, each vertex propagates its ID to neighbors with higher IDs. For example, if vertex 5 is a neighbor of vertices 1 and 6, it only propagates its ID to vertex 6. In the second superstep, each vertex iterates over received messages and augments each one with (1) its own ID and (2) the edge weight connecting this vertex with the message sender. It then propagates the augmented messages to all neighbors with higher IDs. In the third superstep, each vertex checks whether each of the received messages forms a triangle. If a triangle is found, the vertex compares the edge distances to discover whether there exists a semi-metric edge. If a semi-metric edge is found, it is marked for removal. In the final superstep, all marked edges are removed. Figure 6.5 illustrates an example.

6.5.2 Phase 2: Identify local metric edges

This phase consists of three supersteps. The first superstep is executed once, while steps two and three are executed in an alternate fashion, until no further metric edges can be discovered.

1. *Mark the lowest-weight local edges as metric*: Each vertex marks its lowest-weight edges as metric (according to Proposition 1). Then, it sends a message with its ID and the edge weight, along the identified metric edges.
2. *Send lowest alternative path distance to metric edges*: Vertices which have received a message are endpoints of metric edges found in the previous superstep. Thus,

Graph	% of unlabeled edges
Tuenti [14]	1.17
LiveJournal [25]	4.36
NotreDame-web [22]	9.09
DBLP [190]	8.08
Twitter egonet [126]	1.15

Table 6.3 – Percentage of unlabeled edges, after the second phase of the mertric backbone algorithm.

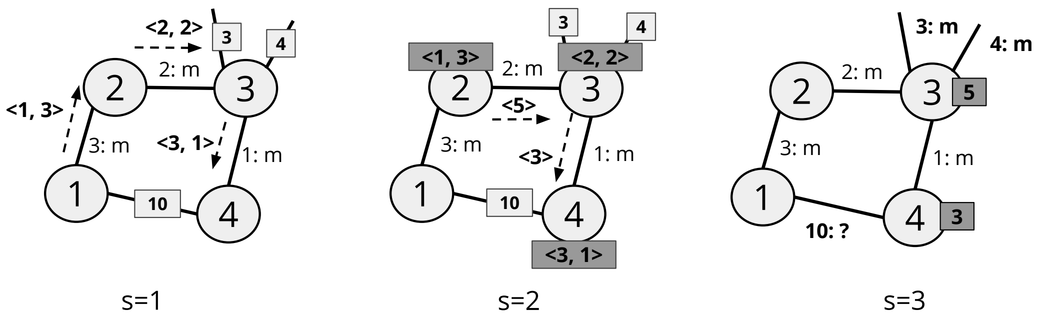


Figure 6.6 – Identifying local metric edges in the vertex-centric model. Messages in transit are shown with dotted arrows. Received messages are shown in dark grey boxes. At the third superstep vertex 3 has identified both of its unlabeled edges as metric.

these vertices mark opposite-direction edges as metric. Then, every vertex sends one message along all its metric edges. For metric edge (u, v), the message contains the distance of the shortest two-hop path, that passes through u and contains (u, v). This distance is computed by adding the weight of (u, v) and the smallest weights of the rest of u's edges.

3. *Check lowest-weight unlabeled edge* In the third superstep, each vertex checks whether it can reason about the semi-metricity of its smallest-weight unlabeled edge. If all of the weights in the received messages are larger than the weight of this edge, then both this edge and the opposite-direction edge can be safely marked as metric.

Figure 6.6 illustrates an example.

6.5.3 Phase 3: Label remaining metric edges

In order to characterize the remaining unlabeled edges, we initiate parallel breadth-first searches. For every unlabeled edge (u, v), u propagates a message to its neighbors to explore paths that have weight lower than the weight of (u, v). After one initialization superstep the computation iteratively runs custom breadth-first searches, until no unlabeled edges remain. During the initialization superstep, each vertex gathers its unlabeled edges. For each unlabeled edge, it creates a message and propagates it along all edges that have weight lower than the unlabeled edge weight. In the next supersteps, each vertex performs the following computation until convergence. Upon receiving a message, it checks whether it is the target of the message edge. If it is, then this edge is labeled as semi-metric. Otherwise, it propagates the message to neighbors that could produce shorter paths, making sure not to forward the message back to its source. Note that, the percentage of unlabeled edges for which we need to execute a BFS is usually very small. For the networks we analyze, the percentage of unlabeled edges, after executing the second step of the algorithm, is under 10% in all cases, and as low as 1% for the Twitter and Tuenti graphs. We present these results in Table 6.3. Also, since most of the semi-metric edges have already been discovered, the majority of the BFSs terminate after only a few steps.

6.5.4 Spreading the communication overhead over multiple supersteps

The computational complexity of the first phase of the metric backbone algorithmis equivalent to the one of triangle enumeration in undirected graphs. Even though there exist several heuristics that significantly reduce the practical computation time [32, 149, 66], the memory requirements in a message-passing system like Apache Giraph might still be fairly high. In order to avoid memory problems, we apply a simple, yet practically effective optimization in this phase.

We make the observation that the computations for detecting semi-metric edges can be performed completely independently. None of the parts of the algorithm require any message aggregation or combining. Thus, in order to reduce the communication load, we spread the algorithm execution into several identical superstep-groups, which we call *megasteps*. Each megastep contains the three supersteps of the first phase of the algorithm, as described in 6.5.1. Throughout the program execution, we keep all vertices active. However, during each megastep, only some of the vertices execute the computation, while the rest remain idle. Since all computations are independent from each other, there is no need to maintain or transfer any state across supersteps. When all vertices have executed their computation, we have encountered all first-order semi-metric edges. Note that for this optimization to work, we do not remove any edges before all vertices have completed their computation phase. Instead, when we encounter a semi-metric edge, we simply put a mark on it. We then remove semi-metric edges during a single finalization superstep.

In our implementation, we decide which vertices to activate in which megastep, based on their numeric vertex IDs. More sophisticated load balancing methods might yield better performance. In our evaluation, we varied the number of megasteps for each the of the experiments. Intuitively, the number of megasteps should increase with the graph size, but it also depends on the amount of available memory. For example, we found that, for our experimental setup, 10 megasteps result in a fairly fast execution for finding first-order semi-metric edges in the Livejournal dataset, while we used 100 megasteps for the Tuenti network.

Graph	SSSP	MSSP
DBLP [190]	120	11
Twitter-ego [126]	177	14

Table 6.4 – Ratio of the projected execution time of (1) SSSP from all vertices and (2) MSSP for all vertices of the input graphs over the total execution time of our algorithm for computing the metric backbone.

6.6 Evaluation

We evaluate our approach in different respects. First, we measure the performance of our algorithm and show that it is orders of magnitude faster than APSP. Second, we show that even when computing the exact backbone incurs high overhead, we can compute a backbone approximation by removing only 1^{st}-order semi-metric edges, to scale to large graphs. Third, we measure the impact on the performance on different graph management systems. We evaluate this using a variety of real-world data sets and graph queries.

6.6.1 Comparing to APSP

We first compare our algorithm's performance with that of APSP. Computing APSP on a distributed platform like Apache Giraph is a challenging task. To perform this computation, every vertex must compute and store $|V|$ distances, resulting in excessive communication and memory overhead. Instead, we implement APSP in two alternative ways. The first approach computes Single-Source Shortest Paths (SSSP) for each vertex individually, in successive jobs. The second approach runs multiple instances of Multi-Source Shortest Paths (MSSP) [4]. MSSP batches a configurable number of simultaneous SSSPs in the same job to improve efficiency.

First, we run 100 instances of SSSP from different sources and average the execution time. Second, we run MSSP using 1% of the vertices as sources. Because of the time it takes to run the entire APSP computation, we estimate the total time by projecting the measured time to the entire graph. We compare the projected execution times with the total execution time of our algorithm for the Twitter-egonet and DBLP graphs. Running APSP for larger graphs was impossible with our available computing resources.

We show the results in Table 6.4. The projected execution times for SSSP are in the order of months, while the projected execution times for MSSP are in the order of days. Instead, our algorithm was able to compute the metric backbone in 8 hours for the DBLP graph and 2 hours for the Twitter graph.

6.6.2 Scalability

Even though computing the exact backbone can incur high overhead, we can still compute a backbone approximation in a scalable manner by removing only 1^{st}-order semi-metric edges. As we show in the beginning of this chapter, the first phase of our algorithm

4. Our implementation of MSSP is available as open source.

Graph	|E|	Size (GB)	Time (s)
Twitter [111]	1.5B	72	3792
Tuenti [14]	685M	33	1305
LiveJournal [25]	34M	1.6	62
NotreDame-web [22]	1.5M	0.07	25
Twitter egonet [126]	1.7M	0.08	32
DBLP [190]	1M	0.05	20

Table 6.5 – Execution runtime of semi-metric triangle detection and removal for real-world graphs.

removes the majority of semi-metric edges, practically providing a good approximation for many applications. Here, we show that this phase affords a scalable implementation on top of distributed graph processing frameworks, such as Giraph. We measure execution time as the size of the graph increases. In the experiments following, we use this backbone approximation to measure speedup of graph analysis applications.

We run the first phase of our algorithm on synthetic graphs constructed with the Watts-Strogatz model [181], using Giraph's built-in graph generator. The generator produces unweighted graphs with high clustering coefficient and low average path length. Using synthetic graphs for these experiments allows us to gradually increase the number of vertices and edges in a controlled manner, still working with a graph that resembles a real-world social network or web-graph characterized by small-world properties. Even though the synthetic graphs do not have edge weights, the execution time of this phase depends only on the size of the graph and the number of triangles. Edge weights impact only the the number of edges removed at the end (see Section 6.5.1). Edge removals have a constant overhead that does not affect scalability. We configure the generator so that vertices have 30 edges on average and set the model rewiring probability to 0.3. We performed this experiment on an AWS cluster consisting of 32 r3.4xlarge instances (16 vCPUs, 122GB memory).

We show the result in Figure 6.7. As the graph size increases from 240 million to 3.8 billion edges, the running time increases almost linearly. On the largest graph, which has a size of 123GB and 4.6 billion triangles, the computation finishes in 14 minutes. This is the largest graph we could process on the 32 compute nodes due to the message overhead. Further, in Section 6.6.4, we provide results on a $\sim$50 billion-edge subgraph of the Facebook social network, demonstrating that our algorithm is practical for even larger graphs. In Table 6.5, we also provide the execution time of semi-metric triangle detection for the real-world graph datasets we consider in this paper. The table shows the number of edges and size in GB of each graph. This experiment was performed on an Amazon EC2 cluster of 16 r3.2xlarge instances (8 vCPUs, 61GB memory).

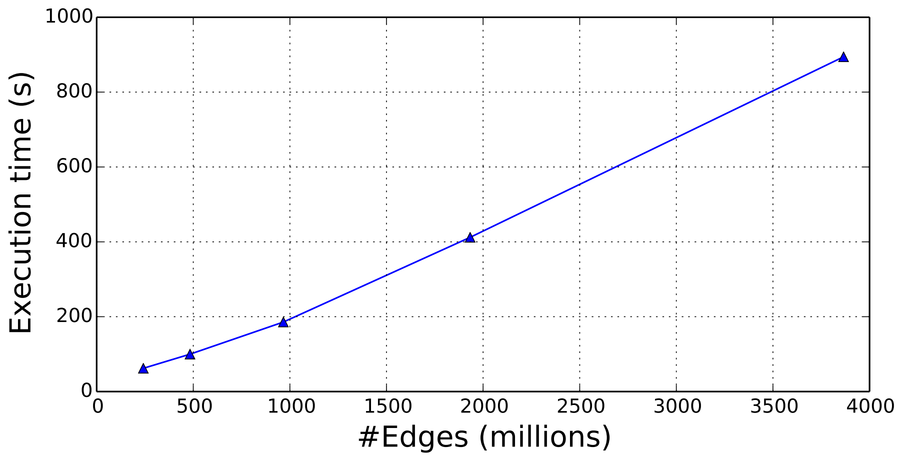


Figure 6.7 – Scalability evaluation of the backbone algorithm. The figure shows execution time of the first phase for synthetic graphs of increasing size.

Graph	**% of size reduction**
Tuenti [14]	59.14
LiveJournal [25]	39.66
NotreDame-web [22]	16.67
DBLP [190]	22.62
Twitter egonet [126]	57.14

Table 6.6 – The Neo4j relationship store size reduction when 1st-order semi-metric edges have been removed from the graph.

6.6.3 Graph databases

Here, we want to measure the impact on query latency and storage reduction by using the backbone transparently in a graph database. We load the original graph in the Neo4j database and then run two different queries: (i) we run a shortest path query for 1000 randomly selected pairs of nodes in the graph, (ii) we run a connected components query 10 times. For both queries we measure the average latency. We perform this measurement for different graphs. Subsequently, we start another instance of the database, where we load the graph after removing 1^{st}-order semi-metric edges, and repeat the same experiment. We run this experiment on an Intel Xeon E5530 2.40GHz server with 128GB of RAM, running Ubuntu 2.6.38.

Table 6.6 shows the size reduction of the Neo4j database when we use the approximate backbone for query evaluation. We see that for highly semi-metric graphs, the database files have close to 60% less storage requirements.

Figure 6.8 shows the speedup when we execute the queries on the approximate metric backbone compared to the original graph for all the workloads. First, we observe the high-

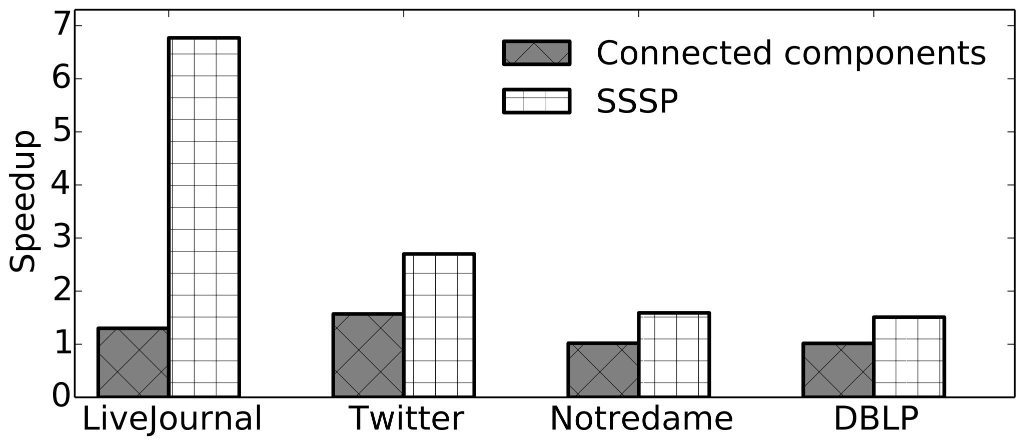


Figure 6.8 – Query speedup on the Neo4j graph database. Graphs in the x-axis are ordered from larger to smaller.

Graph	Spearman Coefficient
Facebook	0.98
Tuenti [14]	0.98
LiveJournal [25]	0.95
NotreDame-web [22]	0.76
DBLP [190]	0.98
Twitter egonet [126]	0.97

Table 6.7 – The Spearman correlation coefficient between (a) the ranking computed by PageRank on the original weighted graph and (b) the ranking computed by PageRank on the same graph, after removing first-order semimetric edges.

est speedups occur for the shortest path query. Second, the larger the graph, the higher the speedup. For the three smaller graphs, the speedup ranges from 1.51 to 2.7, while for the LiveJournal graph the speedup is 6.7.

Notice that the connected components query does not consider the edge weights. For smaller graphs the speedup is 1.01. However, for larger graphs we still measured significant performance improvement, with the speedup ranging from 1.30 to 1.5.

In general, even in a highly optimized system such as a commercial graph database, we still derive a significant speedup by applying the backbone approach transparently. We believe that integrating the approach inside the system can yield even higher speedups.

6.6.4 Distributed graph analytics

Reducing the edges of a graph impacts directly the performance of algorithms developed on top of distributed graph processing systems, such as Pregel and Graphlab. In such systems, programs are typically communication-intensive and communication coincides with the edges of the graph. Further, reducing the size of a distributed graph also reduces the runtime memory requirements of such systems, which may indirectly affect performance as well.

We measure how removing semi-metric edges from a graph improves the performance of different applications, developed on the Apache Giraph graph processing system. We run three applications on the original graph and on the graph with no 1^{st}-order semi-metric edges; Connected Components (CC), Single-Source-Shortest-Paths (SSSP), and weighted PageRank (15 iterations). For each application, we measure (a) the runtime speedup, and (b) the total amount of messages sent. For PageRank, we also compute the Spearman correlation coefficient of the resulting rankings, shown in Table 6.7. The Spearman correlation measures the statistical dependence between the ranking of two variables. Intuitively, we expect two variables to have a high Spearman correlation when their values have similar rankings. We perform this experiment on an Amazon EC2 cluster with 16 r3.2xlarge instances[5].

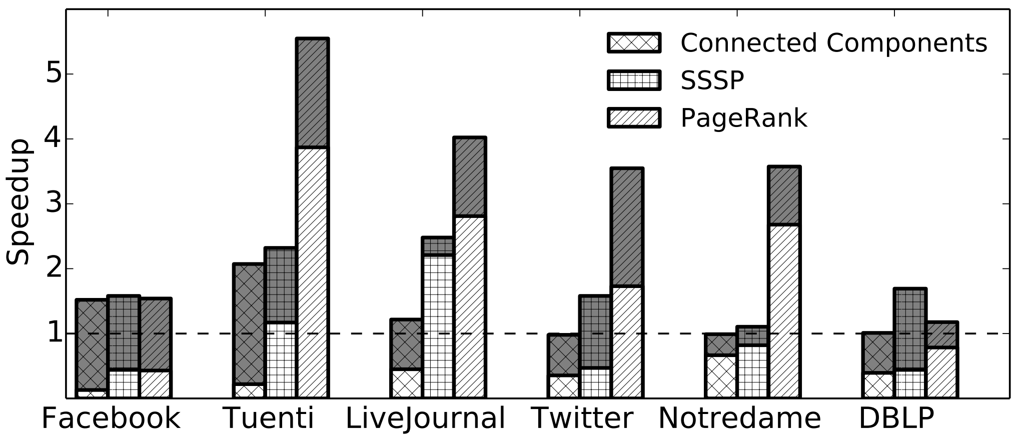


Figure 6.9 – Runtime speedup on Apache Giraph. Graphs in the x-axis are ordered from larger to smaller. Bars are overlayed, not stacked.

In Figure 6.9, we plot the runtime speedup for different applications. Bars are overlayed, not stacked. The white bars show the speedup $S_1 = T_o/(T_b + T)$, where T_o is the time to run the analysis on the original graph, T_b is the time to calculate the backbone approximation, that is, remove the 1^{st}-order semi-metric edges, and T is the time to run the analysis on the backbone approximation. The gray bars show the speedup $S_2 = T_o/T$ that considers only the time to run the analysis on the two graphs. This is the speedup after the overhead of our algorithms gets amortized.

For more compute-intensive applications, such as PageRank, the total runtime, including the removal of semi-metric edges, is typically lower than running the application on the original graph. For SSSP, this is also true for the Tuenti and Livejournal datasets. In some

5. The analysis on the Facebook graph was run on an experimental cluster with 50 machines, each with 16 cores and 10Gbs Ethernet.

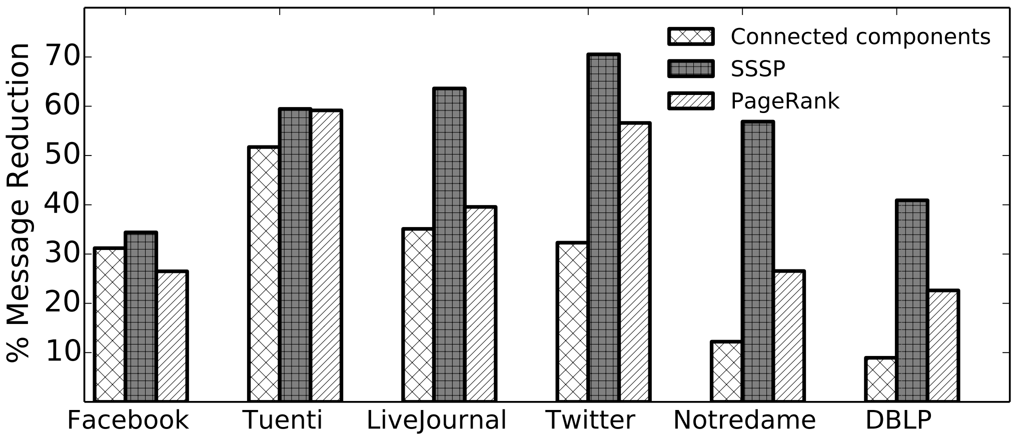


Figure 6.10 – Communication reduction on Apache Giraph. Graphs in the x-axis are ordered from larger to smaller.

cases, though, if we consider only one run of the analysis, the overhead of our approach exceeds the analysis runtime on the original graph. This is true for the Facebook and DBLP datasets, and all instances of the Connected Components experiment.

Note, however, that computing the backbone is intended to be run once and re-used multiple times across applications, so S_2 represents the practical speedup we see in the applications. For example, approximating betweenness centrality in a graph requires running *multiple* SSSP instances with different source vertices. For such an application, the overhead of removing semi-metric edges gets amortized after only the first two runs of the analysis in the worst case.

With regard to graph semi-metricity, as expected, we notice larger gains for highly semi-metric graphs. Tuenti, Livejournal, Twitter, and Notredame (40-60% semi-metric) give better speedups than Facebook and DBLP (23-27% semi-metric). With regard to application complexity, we observe a higher benefit for more compute-intensive applications. We see the highest speedup for PageRank, which runs close to six times faster for the Tuenti graph. We believe that this is because, in contrast to the SSSP and CC applications, in PageRank, all the nodes communicate with all their neighbors, in every iteration (in SSSP and CC nodes do not send outgoing messages to their neighbors, if their value does not change). The shortest paths application also benefits significantly, running about two times faster on average. Not surprisingly, connected components experiences the lowest speedup, since it does not make use of the edge weights. Even so, for a highly semi-metric graph, such as Tuenti, the benefit is substantial.

Figure 6.10 shows the communication reduction, when running the same three applications, for the different networks. For SSSP and PageRank, we observe tremendous communication reduction, ranging from 30% to 70%, in terms of messages exchanged, throughout the application execution. Even for connected components, the reduction is remarkable, ranging from 10% to 50%.

6.7 Conclusion

In this chapter, we examined the property of semi-metricity in real-world graphs and we showed how we can use the metric backbone to compute graph metrics efficiently. Our experimental results show that using the backbone, even on modestly semi-metric graphs, reduces the size of the original graph enough to allow for runtime speedups of up to 6x.

We have proposed an algorithm that computes the metric backbone, avoiding the computation of APSP. We have showed that we can closely approximate the metric backbone by removing only 1st-order semi-metric edges. Computing the approximation is a scalable step that make the application of the backbone practical in large-scale scenarios.

We have shown that the metric backbone can be used to compute exact values or approximations of graph metrics. Algorithms that depend on the weighted shortest paths can be executed on the metric backbone without any change and provide exact results. The backbone also preserves the graph connectivity and provides exact answers to reachability queries. Further, we have shown that the metric backbone can also be used to speed up algorithms that do not explicitly depend on the shortest paths. We have evaluated the computation of PageRank on the backbone and we have shown that the results are highly correlated to the exact ranking. Previous work has also demonstrated that the backbone provides good results in the case of community detection algorithms [169]. We have verified this result by using the metric backbone in the context of the web tracker classification of Chapter 4. We found the hosts-projection graph to be highly semi-metric. We were able to remove 71% of its edges, while preserving the classification accuracy of our data pipeline.

Chapter 7

Conclusion

In this thesis, we have proposed performance optimization techniques and tools for distributed graph processing. Our goal is to improve the performance of distributed graph processing, both on general-purpose and specialized platforms. To that end, we aim to reduce the size of datasets to be processed while getting accurate results. We also aim to reduce the amount of total computation to be performed and the amount of communication required. We have accomplished these objectives by designing transparent and system-independent optimizations for distributed graph processing applications.

7.1 Summary of Results

We have reviewed the evolution of distributed graph processing in recent years and we have examined how shared-nothing clusters and high-level programming abstractions have emerged as popular design choices. We have studied the evolution the MapReduce programming model and we have provided an overview of recent research that proposed enhancements and extensions to the popular computing paradigm.

Next, we have presented a survey of programming abstractions and platforms for distributed graph processing. We have analyzed and compared high-level programming models that have been recently used to build distributed graph applications. For each model, we have identified representative applications and we have reviewed proposed performance optimizations. We have categorized recent distributed graph processing systems, with respect to their programming and execution models. Further, we have reviewed graph analysis applications and we have identified open issues and future research directions in the area.

Following, we have presented a real-world application of large-scale graph processing. We have used web traffic log data to build an end-to-end graph processing pipeline that uses an iterative community detection algorithm to automatically classify web trackers. We have shown how web log data can be modeled as a graph and we have demonstrated how such a graph can be transformed and analyzed to provide highly accurate classification results. We have drawn inspiration by this use-case and we have proposed two optimization methods to improve the performance of similar graph processing pipelines.

Our first optimization method fulfills the objective of reducing computation and communication in large-scale iterative graph processing. We have presented a unified framework for fixpoint applications and we have implemented template execution plans for fixpoint iteration techniques, using common dataflow operators. We have built a declarative API and a cost model that can automatically exploit the asymmetrical convergence of fixpoint algorithms. Our framework leverages the cost model to choose the best iterative execution plan during runtime. Using our framework we achieve up to 1.7x speedup on iterative applications and up to 54% communication savings, as compared to using the naive bulk execution technique. We have further evaluated our framework and we have demonstrated that our cost-based optimization is close to ideal.

Our second optimization method fulfills the objective of reducing the size of the data sets to be processed, while it also reduces communication overhead in several cases. We have used the concepts of semi-metricity and the metric backbone to significantly reduce the sizes of weighted graphs. We have proposed a scalable algorithm that computes an approximation of the backbone and we have implemented it in a popular specialized graph processing system. We have evaluated the application of the backbone and we have shown that we can achieve remarkable speedup in distributed graph processing and graph database queries. Our results show that, even on modestly semi-metric graphs, we can reduce the size of the original graph enough to allow for runtime speedups of up to 6x. Further, we have applied our technique on the web tracker classification graph and we were able to remove 71% of its edges, while preserving the classification accuracy of our data pipeline.

7.2 Results Generality

Even though we have used the web trackers classification as a specific application for inspiration, our proposed optimization methods are general and can be applied in a wide range of distributed graph processing scenarios. The fixpoint optimization method is applicable to any iterative value-propagation graph algorithm with a complex update function. Such algorithms include update functions that are not combinable, associative-commutative or cannot be expressed as an incremental computation, functions whose properties are not easy to reason about, and "black box" update functions to which the user application has no access. Further, our optimization framework, including the fixpoint API and the cost-based optimizer, can be easily implemented on top of general-purpose or specialized data processing engines. By using common relational operators and dataflow plans, we have made the design independent of the underlying processing engine.

The concept of semi-metricity and the metric backbone are also general and can be applied in a variety of scenarios that are not considered in this thesis. The distributed three-phase algorithm for the backbone calculation can be implemented in any graph processing system that supports the vertex-centric programming model. Examples of such systems are provided in Table 3.2 of Chapter 3. The metric backbone technique can be integrated in batch graph processing systems or graph databases as a pre-processing optimization step to improve the performance of a variety of queries and analysis jobs. Algorithms that depend on the shortest paths, , and connectivity queries can be computed on top of the metric backbone or the approximate backbone and return exact results. Further, the backbone can be used to speed up community detection, page rank, and recommendations, in which cases it provides approximate results.

7.3 Future Work

In the future, we would like to refine the proposed cost model and also extend it to support complex iterative algorithms beyond value-propagation. We are particularly interested in working towards support for efficient iterations on graphs with a dynamic edge set and iterative processing of continuous graph streams. We envision building a fixpoint iteration optimizer able to extract the update function properties and decide on an iteration strategy with full independence. We plan to refine our fixpoint cost model, by adding a more sophisticated estimation of graph properties. While we currently use the average node degree to estimate how many vertices will be active in the following iteration, this might not be a good metric for all types of graphs. For example, clustering coefficient might provide a better estimation. Computing such metrics can be challenging, especially if no the graph properties are known beforehand. We would like to investigate solutions such as sampling and gathering statistics during the initial bulk iterations to approximate the graph properties. Considering the metric backbone, we plan to further use the concept to re-design certain algorithms explicitly on top of the backbone. For instance, we could possibly compute shortest paths more efficiently by intelligently choosing which nodes to traverse. We believe that the metric backbone and distance backbones, in general, offer a framework for the design of such algorithms and we plan to investigate this in the future.

Bibliography

[1] Apache Flink. http://www.flink.apache.org. [Online; Last accessed June 2016].

[2] Apache Giraph. http://www.giraph.apache.org. [Online; Last accessed June 2016].

[3] Apache Hama . http://www.hama.apache.org. [Online; Last accessed June 2016].

[4] Apache Tinkerpop . http://www.tinkerpop.apache.org. [Online; Last accessed June 2016].

[5] Cascading. http://www.cascading.org/. [Online; Last accessed 2012].

[6] EasyPrivacy List. https://hg.adblockplus.org/easylist/. [Online; Last accessed June 2015].

[7] Hadoop: Open-Source implementation of MapReduce. http://hadoop.apache.org. [Online; Last accessed March 2014].

[8] Introducing Gelly: Graph Processing with Apache Flink. http://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html. [Online; Last accessed June 2016].

[9] Kosmos distributed file system. http://code.google.com/p/kosmosfs/. [Online; Last accessed Jan 2013].

[10] MapReduce: A Major Step Backwards. http://homes.cs.washington.edu/billhowe/mapreduce_a_major_step_backwards.html. [Online; Last accessed Jan 2013].

[11] The movielens dataset. http://grouplens.org/datasets/movielens.

[12] Neo4j. http://neo4j.com/. [Online; Last accessed June 2016].

[13] PoweredByHadoop. http://wiki.apache.org/hadoop/PoweredBy. [Online; Last accessed Jan 2013].

[14] The Tuenti Social Network. http://www.tuenti.com.

[15] Wikipedia links, english network dataset. http://konect.uni-koblenz.de/networks/wikipedia_link_en. [KONECT, June 2016].

[16] YouTube Statistics. http://www.youtube.com/yt/press/statistics.html. [Online; Last accessed March 2014].

[17] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[18] Lada A Adamic and Eytan Adar. Friends and neighbors on the Web. *Social Networks*, 25(3):211–230, July 2003.

[19] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[20] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[21] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. Storage infrastructure behind facebook messages: Using hbase at scale. *IEEE Data Eng. Bull.*, 35(2):4–13, 2012.

[22] Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. Internet: Diameter of the World-Wide Web. *Nature*, 401(6749):130–131, September 1999.

[23] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[24] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 577–588. ACM, 2013.

[25] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *ACM SIGKDD'06*, August 2006.

[26] Francois Bancilhon. *Naive evaluation of recursively defined relations*. Springer, 1986.

[27] Nguyen Thien Bao and Toyotaro Suzumura. Towards highly scalable pregel-based graph processing platform with x10. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 501–508. International World Wide Web Conferences Steering Committee, 2013.

[28] Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Ahmed Barnawi, Sherif Sakr, et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.

[29] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[30] Jason Bau, Jonathan Mayer, Hristo Paskov, and John C. Mitchell. A promising direction for web tracking countermeasures. In *Web 2.0 Security and Privacy*, 2013.

[31] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: Towards a scalable, semistructured data platform for evolving-world models. *Distrib. Parallel Databases*, 29(3):185–216, June 2011.

[32] Jonathan W. Berry, Luke K. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. Why do simple algorithms for triangle enumeration work in the real world? In *Conference on Innovations in Theoretical Computer Science*, pages 225–234. ACM, 2014.

[33] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*, volume 23. Prentice hall Englewood Cliffs, NJ, 1989.

[34] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl C. Kanne, Fatma Özcan, and Eugene J. Shekita. *PVLDB*, 4(12):1272–1283, 2011.

[35] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[36] Jeremy Blackburn, Xiang Zuo, Nicolas Kourtellis, John Skvoretz, and Adriana Iamnitchi. The power of indirect ties in friend-to-friend storage systems. *IEEE International Conference on Peer-to-Peer Computing*, September 2014.

[37] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *ACM WWW'11*, pages 587–596, 2011.

[38] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1151–1162. IEEE, 2011.

[39] Vinayak R Borkar, Michael J Carey, and Chen Li. Big data platforms: what's next? *XRDS: Crossroads, The ACM Magazine for Students*, 19(1):44–49, 2012.

[40] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[41] Sergey Brin and Lawrence Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[42] Yingyi Bu, Vinayak Borkar, Jianfeng Jia, Michael J Carey, and Tyson Condie. Pregelix: Big (ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment*, 8(2):161–172, 2014.

[43] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[44] Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, page 1094342011403516, 2011.

[45] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15*, page 7. ACM, 2015.

[46] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, 2015.

[47] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 3. ACM, 2012.

[48] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Computation and communication efficient graph processing with distributed immutable view. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 215–226. ACM, 2014.

[49] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, page 1. ACM, 2015.

[50] Shihyen Chen, Bin Ma, and Kaizhong Zhang. On the similarity metric and the distance metric. *Theoretical Computer Science*, 410(24-25):2365–2376, May 2009.

[51] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.

[52] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *ACM SIGKDD'09*, pages 219–228. ACM, 2009.

[53] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[54] Nicholas A. Christakis and James H. Fowler. *Connected: The Surprising Power of Our Social Networks and how They Shape Our Lives*. 2009.

[55] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *ACM SIGKDD'11*, pages 672–680, 2011.

[56] Vittoria Colizza, Romualdo Pastor-Satorras, and Alessandro Vespignani. Reaction-diffusion processes and metapopulation models in heterogeneous networks. *Nature Physics*, 2007.

[57] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[58] Tiago Manuel Louro Machado De Simas. *Stochastic Models and Transitivity in Complex Networks*. PhD thesis, Indiana University, 2012.

[59] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[60] David J DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1255–1266. ACM, 2013.

[61] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

[62] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only aggressive elephants are fast elephants. *Proc. VLDB Endow.*, 5(11):1591–1602, July 2012.

[63] Niels Doekemeijer and Ana Lucia Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, 2014.

[64] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014.

[65] Feodor F. Dragan, Fedor V. Fomin, and Petr A. Golovach. Spanners in sparse graphs. *Journal of Computer and System Sciences*, 77(6):1108 – 1119, 2011.

[66] David Ediger and David A. Bader. Investigating graph algorithms in the bsp model on the cray xmt. In *IPDPS Workshops*, pages 1638–1645. IEEE, 2013.

[67] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[68] Iman Elghandour and Ashraf Aboulnaga. Restore: reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, February 2012.

[69] Radwa Elshawi, Omar Batarfi, Ayman Fayoumi, Ahmed Barnawi, and Sherif Sakr. Big graph processing systems: State-of-the-art and open challenges. In *Big Data Computing Service and Applications (BigDataService), 2015 IEEE First International Conference on*, pages 24–33. IEEE, 2015.

[70] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.*, 4(9):575–585, June 2011.

[71] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th international conference on World Wide Web*, WWW '15, 2015.

[72] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5(11):1268–1279, 2012.

[73] Marjan Falahrastegar, Hamed Haddadi, Steve Uhlig, and Richard Mortier. Anatomy of the third-party web tracking ecosystem. *CoRR*, abs/1409.1066, 2014.

[74] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query preserving graph compression. In *ACM SIGMOD'12*, pages 157–168, 2012.

[75] Santo Fortunato and Alessandro Flammini. Random walks on directed networks: the case of pagerank. *International Journal of Bifurcation and Chaos*, 17(07):2343–2353, 2007.

[76] NE Friedkin. Horizons of observability and limits of informal control in organizations. *Social Forces*, 62(1):54 – 77, 1983.

[77] Wai Shing Fung, Ramesh Hariharan, Nicholas J.A. Harvey, and Debmalya Panigrahi. A general framework for graph sparsification. In *ACM Symposium on Theory of Computing*, pages 71–80, New York, NY, USA, 2011. ACM.

[78] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[79] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

[80] Richard Gomer, Eduarda Mendes Rodrigues, Natasa Milic-Frayling, and M. C. Schraefel. Network analysis of third party tracking: User exposure to tracking cookies through search. In *Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 01*, pages 549–556. IEEE Computer Society, 2013.

[81] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[82] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[83] Amit Goyal and Sara Dadizadeh. A survey on cloud computing. Technical report, University of British Columbia, 2009.

[84] Mark Granovetter. The strenght of weak ties. *American Journal of Sociology*, 78(6):1360–1380, May 1973.

[85] Yong Guo, Marcin Biczak, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 395–404. IEEE, 2014.

[86] Yong Guo, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L Willke. Benchmarking graph-processing platforms: a vision. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 289–292. ACM, 2014.

[87] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[88] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272, 2011.

[89] Imranul Hoque and Indranil Gupta. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, page 9. ACM, 2013.

[90] Sunghwan Ihm and Vivek S. Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 295–312, 2011.

[91] A Iosup, T Hegeman, W Ngai, S Heldens, A Prat, T Manhardt, H Chafi, M Capota, N Sundaram, M Anderson, et al. Ldbc graphalytics: A benchmark for large-scale

graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(12), 2016.

[92] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72. ACM, 2007.

[93] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[94] P Jaccard. The Distribution of the Flora in the Alpine Zone. *New Phytologist*, 11(2):37 – 50, 1912.

[95] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, March 2011.

[96] Dawei Jiang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Sai Wu. epic: an extensible and scalable system for processing big data. *Proceedings of the VLDB Endowment*, 7(7):541–552, 2014.

[97] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: an in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, September 2010.

[98] Vasiliki Kalavri. *Performance Optimization Techniques and Tools for Data-Intensive Computation Platforms*. Licentiate Thesis in Information and Communication Technology, KTH Royal Institute of Technology, Sweden, 2014.

[99] Vasiliki Kalavri, Jeremy Blackburn, Matteo Varvello, and Konstantina Papagiannaki. Like a pack of wolves: Community structure of web trackers. In *International Conference on Passive and Active Network Measurement*, pages 42–54. Springer, 2016.

[100] Vasiliki Kalavri, Vaidas Brundza, and Vladimir Vlassov. Block sampling: Efficient accurate online aggregation in mapreduce. In *CloudCom (1)*, pages 250–257, 2013.

[101] Vasiliki Kalavri, Stephan Ewen, Kostas Tzoumas, Vladimir Vlassov, Volker Markl, and Seif Haridi. Asymmetry in large-scale graph analysis, explained. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–7. ACM, 2014.

[102] Vasiliki Kalavri, Hui Shang, and Vladimir Vlassov. m2r2: A framework for results materialization and reuse in high-level dataflow systems for big data. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 894–901. IEEE, 2013.

[103] Vasiliki Kalavri, Tiago Simas, and Dionysios Logothetis. The shortest path is not always a straight line: leveraging semi-metricity in graph analysis. *Proceedings of the VLDB Endowment*, 9(9):672–683, 2016.

[104] Vasiliki Kalavri and Vladimir Vlassov. Mapreduce: Limitations, optimizations and open issues. In *TrustCom/ISPA/IUCC*, pages 1031–1038, 2013.

[105] Vasiliki Kalavri, Vladimir Vlassov, and Per Brand. Ponic: Using stratosphere to speed up pig analytics. In *Euro-Par*, pages 279–290, 2013.

[106] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 229–238. IEEE, 2009.

[107] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103. IEEE Computer Society, 2010.

[108] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*, volume 22. SIAM, 2011.

[109] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[110] Balachander Krishnamurthy and Craig Wills. Privacy diffusion on the web: A longitudinal perspective. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 541–550.

[111] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *International Conference on World Wide Web*. ACM Press, April 2010.

[112] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36. ACM, 2012.

[113] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[114] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[115] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. Early accurate results for advanced analytics on mapreduce. *Proc. VLDB Endow.*, 5(10):1028–1039, June 2012.

[116] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, November 2008.

[117] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Rec.*, 40(4):11–20, January 2012.

[118] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM, 2008.

[119] Ian XY Leung, Pan Hui, Pietro Lio, and Jon Crowcroft. Towards real-time community detection in large networks. *Physical Review E*, 79(6):066107, 2009.

[120] Tai-Ching Li, Huy Hang, Michalis Faloutsos, and Petros Efstathopoulos. Trackadvisor: Taking back browsing privacy from third-party trackers. In *Passive and Active Measurement*, pages 277–289. Springer, 2015.

[121] Xingjie Liu, Yuanyuan Tian, Qi He, Wang-Chien Lee, and John McPherson. Distributed graph summarization. In *ACM CIKM'14*, pages 799–808, 2014.

[122] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[123] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, 2014.

[124] Sam Madden. From databases to big data. *IEEE Internet Computing*, (3):4–6, 2012.

[125] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

[126] Julian McAuley and Jure Leskovec. Learning to Discover Social Circles in Ego Networks. In *Advances in Neural Information Processing Systems*, 2012.

[127] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[128] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[129] Frank McSherry, Derek G Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[130] David Melamed. Community structures in bipartite networks: A dual-projection approach. *PLoS ONE*, 9(5):e97823, 05 2014.

[131] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[132] Ulrich Meyer and Peter Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[133] Svilen R Mihaylov, Zachary G Ives, and Sudipto Guha. Rex: recursive, delta-based data-centric computation. *Proceedings of the VLDB Endowment*, 5(11):1280–1291, 2012.

[134] Dennis M Moyles and Gerald L Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *Journal of the ACM (JACM)*, 16(3):455–460, 1969.

[135] Kameshwar Munagala and Abhiram Ranade. I/o-complexity of graph algorithms. In *SODA*, volume 99, pages 687–694. Citeseer, 1999.

[136] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[137] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9. USENIX Association, 2011.

[138] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *ACM SIGMOD International Conference on Management of Data*, pages 419–432, 2008.

[139] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[140] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, September 2010.

[141] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[142] Makoto Onizuka, Hiroyuki Kato, Soichiro Hidaka, Keisuke Nakano, and Zhenjiang Hu. Optimization for iterative queries on mapreduce. *Proceedings of the VLDB Endowment*, 7(4):241–252, 2013.

[143] Tore Opsahl. Triadic closure in two-mode networks: Redefining the global and local clustering coefficients. *Social Networks*, 35(2):159–167, May 2013.

[144] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[145] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[146] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14, 2010.

[147] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2014.

[148] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26, 2014.

[149] Louise Quick, Paul Wilkinson, and David Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *International Conference on Advances in Social Networks Analysis and Mining*, Washington, DC, 2012.

[150] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[151] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[152] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: a framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[153] L.M. Rocha, T. Simas, A. Rechtsteiner, M. Di Giacomo, and R. Luce. MyLibrary@LANL: Proximity and Semi-metric Networks for a Collaborative and Recommender Web Service. In *ACM International Conference on Web Intelligence*, 2005.

[154] Luis M Rocha. Proximity and semi-metric analysis of social networks. In *Internal Report of Advanced Knowledge Integration In Assessing Terrorist Threats LDRD-DR - Network Analysis Component. LAUR*, pages 02–6557, 2002.

[155] Luis M Rocha. Semi-metric behavior in document networks and its application to recommendation systems. *Soft Computing Agents: A New Perspective for Dynamic Information Systems*, 83:137, 2002.

[156] John D Rockefeller. Do-Not-Track Online Act of 2013. US Senate, 2013.

[157] Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.

[158] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12. USENIX Association, 2012.

[159] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[160] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[161] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 979–990. ACM, 2014.

[162] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*, pages 606–609. Springer, 2005.

[163] Sebastian Schelter and Jérôme Kunegis. Tracking the trackers: A large-scale analysis of embedded web trackers. In *Tenth International AAAI Conference on Web and Social Media*, 2016.

[164] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

[165] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 505–516. ACM, 2013.

[166] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD. ACM*, 2016.

[167] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[168] Tiago Simas and Luis Rocha. Semi-metric Networks for Recommender Systems. *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 2012.

[169] Tiago Simas and Luis M. Rocha. Distance closures on complex networks. *Network Science*, 3:227–268, 6 2015.

[170] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par 2014 Parallel Processing*, pages 451–462. Springer, 2014.

[171] Michael Stonebraker. What does 'big data' mean. *Communications of the ACM, BLOG@ ACM*, 2012.

[172] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *The Semantic Web–ISWC 2010*, pages 764–780. Springer, 2010.

[173] Serafettin Tasci and Murat Demirbas. Giraphx: Parallel yet serializable large-scale graph processing. In *Euro-Par 2013 Parallel Processing*, pages 458–469. Springer, 2013.

[174] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.

[175] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[176] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[177] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[178] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[179] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 197–210. ACM, 2013.

[180] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[181] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 1998.

[182] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In *ACM European Conference on Computer Systems*, page 205, April 2009.

[183] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[184] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *ACM SIGPLAN Notices*, volume 50, pages 194–204. ACM, 2015.

[185] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 227–238. ACM, 2014.

[186] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[187] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1307–1317. International World Wide Web Conferences Steering Committee, 2015.

[188] Da Yan, James Cheng, M Tamer Özsu, Fan Yang, Yi Lu, John Lui, Qizhen Zhang, and Wilfred Ng. A general-purpose query-centric framework for querying big graphs. *Proceedings of the VLDB Endowment*, 9(7):564–575, 2016.

[189] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[190] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[191] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[192] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[193] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[194] Peixiang Zhao and Jiawei Han. On graph query optimization in large networks. *Proc. VLDB Endow.*, 3(1-2):340–351, September 2010.

[195] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, 2015.

[196] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[197] Xiang Zuo, Jeremy Blackburn, Nicolas Kourtellis, John Skvoretz, and Adriana Iamnitchi. The Influence of Indirect Ties on Social Network Dynamics. In *International Conference on Social Informatics*, November 2014.