Methods and Algorithms for Data-Intensive Computing

Streams, Graphs, and Geo-Distribution

HOOMAN PEIRO SAJJAD

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2019

School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2019:13
ISBN 978-91-7873-094-0

Akademisk avhandling som med tillstånd av Kungliga Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen i Informations- och kommunikationsteknik fredagen den 15 mars 2019 klockan 13:30 i sal C, Electrum, Kungliga Tekniska högskolan, Kistagången 16, Kista.

© Hooman Peiro Sajjad, February 2019

Tryck: Universitetsservice US AB

List of Papers

This doctoral thesis is based on the following papers:

I Stream processing in community network clouds [1]
Ken Danniswara, Hooman Peiro Sajjad, Ahmad Al-Shishtawy, and Vladimir Vlassov
Published in IEEE 3rd International Conference on Future Internet of Things and Cloud (FiCloud), 2015
Contribution: The author of this doctoral thesis was the main contributor in this work. The author contributed to writing most of the article and proposing the evaluation methodology of the Apache Storm in the Community Network Cloud. He suggested the placement methods for the Storm components and provided the dataset for the evaluation.

II Smart partitioning of geo-distributed resources to improve cloud network performance [2]
Hooman Peiro Sajjad, Fatemeh Rahimian, and Vladimir Vlassov
Published in IEEE 4th International Conference on Cloud Networking (CloudNet), 2015
Contribution: The author was the main contributor in this work, including writing most of the article, formulating the idea of clustering geo-distributed resources into a community detection problem, designing and implementing the decentralized community detection algorithm and the evaluations.

III Optimizing windowed aggregation over geo-distributed data Streams [3]
Hooman Peiro Sajjad, Ying Liu, and Vladimir Vlassov
Published in IEEE 2nd International Conference on EDGE, 2018
Contribution: The author was the major contributor in this work. He formulated the problem, wrote most of the paper, designed the coordinated method, proposed the evaluation methodology and conducted the experiments.

IV SpanEdge: towards unifying stream processing over central and near-the-edge data centers [4]
Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov
Published in IEEE/ACM Symposium on Edge Computing (SEC), 2016
Contribution: The author was the major contributor in this work. He wrote most of the paper, designed the architecture of SpanEdge, proposed the two new task groupings, proposed the evaluation methodology, and the future work.

V Boosting vertex-cut partitioning for streaming graphs [5] (**Best Paper Award**)
Hooman Peiro Sajjad, Amir H. Payberah, Fatemeh Rahimian, Vladimir Vlassov, and Seif Haridi
Published in IEEE International Congress on Big Data (BigData Congress), 2016
Contribution: The author was the main contributor in this work, including writing the paper, designing and implementing the graph partitioning framework, and designing and implementing the experiments with real-world and synthetic data sets.

VI Efficient representation learning using random walks for dynamic graphs [6]
Hooman Peiro Sajjad, Andrew Docherty, and Yuriy Tyshetskiy

ArXiv article.
Contribution: The author was the main contributor in this work, including the formulation of the problem, developing the idea of the incremental random walk algorithms, implementing the solutions and the experiments, and writing most of the article.

The papers are referred by their Roman numerals. The author of this doctoral thesis is the main author and the main contributor of all the papers listed above. He has also contributed to the following publications that are not included in this thesis:

- Scaling HDFS with a strongly consistent relational model for metadata [7]
  Kamal Hakimzadeh, Hooman Peiro Sajjad, and Jim Dowling
  Published in IFIP International Conference on Distributed Applications and Interoperable Systems, 2014

- Reproducible distributed clusters with mutable containers: to minimize cost and provisioning time [8]
  Hooman Peiro Sajjad, Kamal Hakimzadeh, and Shelan Perera
  Published in Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems, ACM, 2017

Abstract

Struggling with the volume and velocity of Big Data has attracted lots of interest towards stream processing paradigm, a paradigm in the area of data-intensive computing that provides methods and solutions to process data in motion. Today's Big Data includes geo-distributed data sources. In addition, a major part of today's Big Data requires exploring complex and evolving relationships among data, which complicates any reasoning on the data. This thesis aims at challenges raised by geo-distributed streaming data, and the data with complex and evolving relationships.

Many organizations provide global scale applications and services that are hosted on servers and data centers that are located in different parts of the world. Therefore, the data that needs to be processed are generated in different geographical locations. This thesis advocates for distributed stream processing in geo-distributed settings to improve the performance including better response time and lower network cost compared to centralized solutions. In this thesis, we conduct an experimental study of Apache Storm, a widely used open-source stream processing system, on a geo-distributed infrastructure made of near-the-edge resources. The resources that host the system's components are connected by heterogeneous network links. Our study exposes a set of issues and bottlenecks of deploying a stream processing system on the geo-distributed infrastructure. Inspired by the results, we propose a novel method for grouping of geo-distributed resources into computing clusters, called micro data centers, in order to mitigate the effect of network heterogeneity for distributed stream processing applications. Next, we focus on the windowed aggregation of geo-distributed data streams, which has been widely used in stream analytics. We propose to reduce the bandwidth cost by coordinating windowed aggregations among near-the-edge data centers. We leverage intra-region links and design a novel low-overhead coordination algorithm that optimizes communication cost for data aggregation. Then, we propose a system, called SpanEdge, that provides an expressive programming model to unify programming stream processing applications on a geo-distributed infrastructure and provides a run-time system to manage (schedule and execute) stream processing applications across data centers. Our results show that SpanEdge can optimally deploy stream processing applications in a geo-distributed infrastructure, which significantly reduces the bandwidth consumption and response latency.

With respect to data with complex and evolving relationships, this thesis aims at effective and efficient processing of inter-connected data. There exist several domains such as social network analysis, machine learning, and web search in which data streams are modeled as linked entities of nodes and edges, namely a graph. Because of the inter-connection among the entities in graph data, processing of graph data is challenging. The inter-connection among the graph entities makes it difficult to distribute the graph among multiple machines to process the graph at scale. Furthermore, in a streaming setting, the graph structure and the graph elements can continuously change as the graph elements are streamed. Such a dynamic graph requires incremental computing methods that can avoid redundant computations on the whole graph. This thesis proposes incremental computing methods of streaming graph processing that can boost the processing time while still obtaining high quality results. In this thesis, we introduce HoVerCut, an efficient framework for boosting streaming graph partitioning algorithms. HoVerCut is Horizontally and Vertically scalable. Our evaluations show that HoVerCut speeds up the partitioning process significantly without degrading the quality of partitioning. Finally, we study unsupervised representation learning in dynamic graphs. Graph representation learning seeks to learn low dimensional vector representations for the graph elements, i.e. edges and vertices, and the whole graph. We propose novel and computationally efficient incremental algorithms. The computation complexity of our algorithms depends on the extent and rate of changes in a graph and on the graph density. The evaluation results show that our proposed algorithms can achieve competitive results to the state-of-the-art static methods while being computationally efficient.

Keywords: *stream processing; geo-distributed infrastructure; edge computing; streaming graph; dynamic graph*

Sammanfattning

Att kämpa med volymen och hastigheten hos Big Data har väckt mycket intresse för strömbehandlingsparadigmet, ett paradigm inom dataintensiv databehandling som ger metoder och lösningar för att bearbeta data i rörelse. Det som idag kallas "Big Data" omfattar geodistribuerade datakällor. Dessutom kräver en stor del av dagens Big Data att man utforskar komplexa och föränderliga relationer mellan data, vilket komplicerar alla resonemang om datamängden. Denna avhandling riktas mot utmaningar som uppkommer i samband med geodistribuerade strömmande data och data med komplexa och föränderliga relationer.

Många organisationer tillhandahåller globala applikationer och tjänster som finns på servrar och datacenter lokaliserade i olika delar av världen. Därför genereras de data som behöver bearbetas på olika geografiska platser. Avhandlingen förespråkar distribuerad strömbehandling i geodistribuerade tillämpningar för att förbättra prestanda, inklusive bättre svarstid och lägre nätverkskostnad jämfört med centraliserade lösningar. I den här avhandlingen genomför vi en experimentell studie av Apache Storm, ett allmänt använt open source-processbehandlingssystem, på en geografisk distribuerad infrastruktur som består av resurser "nära-kanten". De resurser som är värdar för systemets komponenter är anslutna med heterogena nätverkslänkar. Vår studie avslöjar en uppsättning problem och flaskhalsar med att distribuera ett strömbehandlingssystem på den geodistribuerade infrastrukturen. Inspirerade av resultaten, föreslår vi en ny metod för gruppering av geodistribuerade resurser i datakluster, benämnda "mikrodatacenter", för att mildra effekten av nätverkets heterogenitet för distribuerade strömbehandlingsapplikationer. Därefter fokuserar vi på en fönsterbaserad aggregering av geo-distribuerade dataströmmar, vilket har använts i stor utsträckning i analys av dataströmmar. Vi föreslår en reduktion av bandbreddskostnaden genom att samordna fönstrade aggregeringar bland datacentra "nära kanten". Vi erbjuder intra-regionala länkar och utformar en ny koordinationsalgoritm med låg overhead som optimerar kommunikationskostnaden för dataaggregering. Därefter föreslår vi ett system, som kallas SpanEdge, som ger en uttrycksfull programmeringsmodell för att förena programmeringen av strömbehandlingsapplikationer i en geografiskt distribuerad infrastruktur och ger ett runtime-system för att hantera (schemalägga och exekvera) strömbehandlingstillämpningar som spänner över flera datacentra. Våra resultat visar att SpanEdge optimalt kan distribuera flödesbehandlingstillämpningar i en geo-distribuerad infrastruktur vilket avsevärt minskar bandbreddskonsumtionen och svarsfördröjningen.

Med avseende på data med komplexa och föränderliga relationer, syftar denna avhandling till fungerande och effektiv behandling av sammankopplade data. Det finns flera domäner som social nätverksanalys, maskininlärning och webbsökning där dataströmmar modelleras som länkade enheter av noder och kanter, nämligen en graf. På grund av sammankopplingen mellan enheterna är utarbetandet av grafdata utmanande. Sammankopplingen mellan grafenheterna gör det svårt att fördela grafen mellan flera maskiner för att bearbeta grafen storskaligt. Vidare, i en strömuppställning kan grafstrukturen och grafelementen kontinuerligt förändras när grafelementen strömmar. En sådan dynamisk graf kräver stegvisa beräkningsmetoder som kan undvika överflödiga beräkningar på hela grafen. I denna avhandling föreslås inkrementella beräkningsmetoder för behandling av dataströmmar som kan minska behandlingstiden medan som högkvalitativa resultat bibehålls. I denna avhandling introducerar vi HoVerCut, ett effektivt ramverk för att öka strömningsgrafikpartitioneringsalgoritmer. HoVerCut är horisontellt och vertikalt skalbar. Våra utvärderingar visar att HoVerCut påskyndar partitioneringen väsentligt utan att försämra kvaliteten på partitioneringen. Slutligen studerar vi oövervakat representationslärande i dynamiska grafer. Grafrepresentationsinlärning syftar till att lära sig lågdimensionella vektorrepresentationer för grafelementen, dvs kanter och bågar och hela grafen. Vi föreslår nya och beräkningsmässigt effektiva inkrementella algoritmer. Beräkningskomplexiteten för våra algoritmer beror på omfattningen och graden av förändringar i ett diagram och på grafens densitet. Utvärderingsresultaten visar att våra föreslagna algoritmer kan uppnå konkurrenskraftiga resultat jämfört med state-of-the-art statiska metoder samtidigt som de är beräkningseffektiva.

To Aziz (my grandmother), Maryam, Gloria, Soheil, and Cesar

Acknowledgments

Life as a Ph.D. student is a path of growth. I would like to express my deep gratitude to all of the people who have inspired and supported me on this path, especially:

My primary advisor Vladimir Vlassov, for his mentorship, help, and encouragements through this work, and for putting his trust in me for conducting my own research and developing my skills to become an independent researcher.

My (ex-)co-advisor Fatemeh Rahimian for her valuable insights, advice, and feedback; my co-authors Amir Hossein Payberah, and Ahmad Al-Shishtawy, for sharing their knowledge, and experience eagerly and for their tremendous help in brainstorming and designing the solutions; my co-authors Ken Danniswara, Ying Liu, Yuriy Tyshetskiy, and Andrew Docherty, who I have truly enjoyed collaborating with them; my co-advisor Seif Haridi, for his precious feedback and advice on writing my doctoral thesis; my advance reviewer Sarunas Girdzijauskas, for his valuable feedback and comments on my doctoral thesis; my co-advisor Jim Dowling, for sharing his insights on open research topics at the early stages of my doctoral studies.

All anonymous reviewers of the research papers, for providing valuable and constructive feedback.

Our head of department Alf Thomas Sjöland, for his quick actions, and for helping me in writing the Swedish abstract; our program director for the doctoral program Christian Schulte, for his availability to consult and for his valuable advice; Sandra Nylén, our department's passionated administrator for handling all the complex administrative works.

My friends and (ex-)colleagues at KTH and RISE SICS, especially Kamal Hakimzadeh, Kambiz Ghoorchian, Edward Tjörnhammar, Jingna Zeng, Navaneeth Rameshan, Vasiliki Kalavri, and Shatha Jaradat, Leila Bahri, Roberto Castañeda Lozano, Gabriel Hjort Blindell, and Amira Soliman for the interesting discussions that we had during our coffee and lunch breaks and for all the inspirations and motivations that they gave me; my colleagues and friends at CSIRO's DATA61, Sydney, for hosting me for a wonderful and fruitful internship during my Ph.D.

Last but not least, my family and friends, especially, my mother Maryam for her constant support on pursuing my passions; my brothers Soheil and Cesar for their help, love, and faith in me; my dear Gloria for her full support during my research, studies, and deadlines, and for her love and continuous encouragement.

Contents

List of Figures

List of Tables

Part I

Thesis Overview

Chapter 1

Introduction

More than a decade ago, the term *Big Data* was coined to address emerging compute and storage challenges caused by massive growth in the amount of data. The growth in the amount of data is due to several factors among them are the growing realization on the insights that can be extracted from data and data-driven decision making, and the decreasing cost and the advancement of data storage technologies [9]. Organizations have realized opportunities in collecting and processing data and have started to collect more data even if they may not know exactly what kind of useful information they can extract from the data at the moment. Big Data is like a gold mine that one needs to dig into it and there may be a piece of gold in it.

In particular, Big Data refers to data that are considerably larger, faster, and more complex such that the previous generation of solutions and tools for maintaining and processing the data (such as relational databases) fall short [10]. In the literature, Big Data is usually recognized by the three "V"s *volume*, *velocity*, and *variety* [10]. Volume refers to the amount of data that is generated. The amount of data has grown from the Terabyte to Petabyte scale and pretty soon is expected to reach the Zettabyte scale. Velocity refers to the speed that the data is generated. Organizations are dealing with data being generated at a high rate such as streams of voice, video, text messages, and social interactions in social network applications such as Facebook, mobile applications data, streams of bank transactions, streams of server logs being generated, and streams of sensor data. Finally, variety represents all types of data including unstructured, semi-structured, and structured data.

By the advancements in data-intensive computing, new methods and paradigms have emerged to scale compute and storage systems to cope with the volume of Big Data [11]. Distributed storage systems such as Hadoop Filesystem [12] and batch programming models such as MapReduce [13] have enabled collection and process of massive amount of data sets through highly scalable and distributed batch processing across hundreds or thousands of servers [10]. Dealing with the velocity of Big Data has attracted lots of interest towards the stream processing paradigm, which provides methods and solutions to process data in motion (data streams) [14]. There are many organizations that have already passed beyond ingesting millions of events a day. For example, Walmart reported in 2010 one million customer transactions every hour [15]; T-Mobile needs to process more than 17

billion events per day for performing network quality analysis [16]. The high velocity of Big Data has created a need for continuous gathering, processing, and analyzing streams of data efficiently and in a timely manner.

In recent years, several advanced distributed stream processing systems have been developed [17, 18] that have contributed to the success of stream processing paradigm. In addition to the systems (such as Apache Storm [17], and Apache Samza [19]) that support pure stream processing, some data processing frameworks (such as Apache Spark [20], and Flink [21]) provide a unified environment for implementing both batch and stream processing applications [22]. However, there are still several open challenges for effective and efficient stream processing. Today's Big Data includes autonomous data sources with distributed and decentralized control [23]. Global scale applications are deployed on a geo-distributed infrastructure, i.e., a large number of servers distributed all over the world. In addition, a major part of today's Big Data requires exploring complex and evolving relationships among data, which complicates any reasoning on the data [23]. The rest of this section explains the challenges raised by geo-distributed data, and the data with complex and evolving relationships.

Many organizations provide global scale applications and services that are hosted on servers and data centers that are located in different parts of the world [24, 25]. Social network, media streaming, and online store services are few examples among several others that usually span the globe. In such geo-distributed settings consist of geographically distributed servers, the data to process and analyze are generated in different geographical locations. The common approach to process geo-distributed data is the *centralized* approach, in which raw data is transferred from different geographical locations to a central data center [26]. In accordance with the centralized approach, most of the existing stream processing systems [27] are designed to work on a single data center. Even though the centralized approach simplifies the design and implementation of stream processing systems and applications, there are several factors that make the centralized approach inefficient. Transferring geo-distributed data to a central data center introduces significant communication over wide-area network (WAN) between the original sources of data on the network edge and the analytic applications hosted in the central data center. However, the WAN bandwidth is expensive and can be scarce [28] and the amount of data generated at the network edge has been growing rapidly. Furthermore, there are applications with stringent time requirements (for example predictable and low latency processing time) that can not tolerate the long communication latency over the WAN links. Another problem with the centralized approach is the legal constraints on data collection. Sometimes data can not be transferred to another geographical area because of the legal bounds to a certain jurisdiction. The increase in the privacy concerns is expected to result in more regulatory constraints on data movement [29]. To overcome the drawbacks of the centralized stream processing, this thesis provides methods, algorithms, and a system for effective and efficient distributed stream processing over geo-distributed settings. This thesis argues that well designed distributed stream processing in geo-distributed settings can improve the performance including better response time and lower network cost compared to the centralized approach.

With respect to data with complex and evolving relationships, this thesis aims at ef-

fective and efficient processing of inter-connected data. Inter-connected data modeled as linked entities of nodes and edges is called a graph. Processing of graph data is important in many domains such as social network analysis [30], machine learning [31], and web search [32]. Because of the inter-connection among the entities in graph data, processing of graph data is challenging. The inter-connection among the graph entities makes it difficult to distribute the graph among multiple machines to process the graph at scale [33]. Furthermore, in a streaming setting, the graph structure and the graph elements can continuously change as the graph elements are streamed. Such a dynamic graph requires incremental computing methods that can avoid redundant computations on the whole graph [34]. This is because graph computation algorithms are expensive and recomputing the algorithms on the whole graph is too costly. Currently, existing solutions for processing of streaming and dynamic graphs often trade-off between processing time and the quality of results. However, this thesis proposes incremental computing methods of streaming graph processing that can boost the processing time while still obtaining high quality results.

The rest of this chapter first presents the thesis statement. Second, the thesis objectives are presented. Following that research methodology of this thesis is explained. Then, the contributions of this thesis are described. In the end, the thesis outline is given.

1.1 Thesis Statement

Well designed distributed stream processing in geo-distributed settings leads to improved performance including better response time and lower network cost compared to centralized solutions. In addition, well designed incremental computing methods of streaming graph processing boost the processing time while still obtaining high quality results.

1.2 Research Objectives

This thesis aims at improving the performance of data-intensive computing, specifically focusing on distributed stream processing in geo-distributed settings, and processing of streaming graphs. The main objectives of this thesis are set towards: (i) reducing the response time and the network cost of distributed stream processing in geo-distributed settings, and (ii) improving the processing time in the processing of streaming graphs.

With respect to distributed stream processing in geo-distributed settings, the primary objectives in the thesis are:

- to mitigate the effect of network heterogeneity on distributed stream processing.
- to reduce the amount of data being transferred over expensive network links.
- to reduce the amount of communication over long latency network links.

As a secondary objective, this thesis seeks to provide a system solution that facilitates programming stream processing applications for geo-distributed infrastructure.

With respect to the processing of streaming graphs, the main objectives are:

- to improve the parallelization and state-sharing.
- to reduce the amount of computation.

Based on the thesis objectives, Chapter 3 presents the thesis contributions related to distributed stream processing in geo-distributed settings. Section 3.1 presents an experimental study, which addresses some of the main issues and bottlenecks of deploying a stream processing system on a geo-distributed infrastructure made of near-the-edge resources. Section 3.2 proposes and evaluates a solution based on grouping near-the-edge resources into micro data centers in order to mitigate the effect of network heterogeneity for distributed data-intensive application components, such as stream processing applications. Section 3.3 presents an efficient method for *windowed grouped aggregation* (see Section 2.1.2) over geo-distributed data streams. Finally, Section 3.4 presents a stream processing system to provide a unified environment for programming stream processing in a geo-distributed infrastructure.

With respect to streaming graph data, specifically this thesis targets two important stages in a typical pipeline of processing large graphs, namely graph partitioning and graph representation learning. Graph partitioning (see Section 2.3.3) is to partition a graph into sub-graphs such that it can be efficiently distributed between multiple computation units (e.g., multiple machines). Graph representation learning (see Section 2.3.5) is to learn a low-dimensional representation of the graph vertices, which can be input to downstream machine learning tasks such as vertex classification, clustering, and graph visualization. Chapter 4 presents the contributions of this thesis related to efficient processing of streaming graph data, specifically, a parallel and distributed framework for online partitioning of graph data presented in Section 4.1 and an efficient solution for incremental graph representation learning presented in Section 4.2.

1.3 Research Methodology

The research work of this thesis is based on the empirical study. Specifically, we observe a problem and propose a solution based on some testable hypothesis. Then, we evaluate the hypothesis based on running some reproducible experiments. This section describes the methods that we have used in this thesis to conduct the research work including the process that we have taken to define the problems, and to design and evaluate the solutions. This section also discusses the challenges that we have faced in different steps of the process and how we have overcome them.

Our general approach for defining a problem domain is by studying related works and literature. We have studied the state-of-the-art approaches for stream processing and we have spotted their important limitations and unsolved problems. We set up experimental environments and run extensive experiments to verify the existence of the problems empirically. For example, in Paper I we evaluated Apache Storm in an emulated geo-distributed infrastructure to find the limitations and the bottlenecks of using Apache Storm in such an environment.

We evaluate our work, by implementing the solutions from scratch or by augmenting an existing implementation whenever possible. Our implementations are open source and are publicly available. The references to the source codes are available in the corresponding papers. To compare our solutions with the state-of-the-art solutions, we use either the actual source code of the solutions (e.g., in Paper I, Paper II, and Paper V) or we implement the solutions according to their published papers (e.g., in Paper III, and Paper VI). In order to run the experiments, we use simulation, emulation, or real settings whenever it is practical. For example, in Paper II, we simulate the network based on the real-world dataset and we implement the network as a weighted graph. We assume the routing protocol is based on the weighted shortest path among each pair of vertices. We use simulation because of the high complexity of the network. In Paper IV, we use the CORE network emulator [35] in order to emulate a geo-distributed infrastructure. We use emulation because it enables us to run the actual software components while designing complex networks. In the case of Paper V, we run the experiments on our servers. We use real-world datasets, as well as synthetic datasets, to evaluate our solution. In Paper VI, we run our experiments on the Cloud.

One of the main challenges in our research has been acquiring information about near-the-edge data centers and resources on the network edge. This information is mainly confidential and is not publicly available. Therefore, we collected the only open data available from the community network Guifi.net [36], in order to simulate and emulate the resources at the network edge. The other challenge is that the implementation of some of the state-of-the-art solutions is not openly available. For example, in Paper III, we had to implement the solution in the literature from scratch.

1.4 Thesis Contributions

The contributions of this doctoral thesis are as follows:

- An experimental study of Apache Storm, a widely used open-source distributed stream processing system, on a geo-distributed infrastructure made of near-the-edge resources (Paper I [1]). We investigate different placements of the Apache Storm components on the geo-distributed resources. Our study exposes a set of requirements for the distributed stream processing system in order to be deployed and operate in a geo-distributed infrastructure, as well as, a set of challenges related to the underlying network connectivity.

- A novel method for grouping of geo-distributed resources into computing clusters, called micro data centers, in order to mitigate the effect of network heterogeneity for distributed data-intensive applications (Paper II [2]). We model geo-distributed resources and their connections as a weighted graph. We estimate the connectivity inside each micro data center by the known modularity metric in graph theory. We propose a novel decentralized community detection algorithm that increases the modularity metric competitive to the state-of-the-art centralized community detection algorithm.

- Reducing the bandwidth cost by coordinating windowed aggregations over data streams among edge data centers (Paper III [3]). We define the theoretical minimum bandwidth cost for aggregating data streams by means of coordination. Based on that, we propose a low-overhead coordination method that can identify relevant data among edge data centers and aggregate them effectively and efficiently, and send data streams among the data centers in a timely manner.

- A geo-distributed stream processing system (Paper IV [4]). We propose SpanEdge that provides an expressive programming model to unify programming stream processing applications on a geo-distributed infrastructure and provides a run-time system to manage (schedule and execute) stream processing applications across data centers.

- A parallel and distributed vertex-cut partitioner for streaming graphs (Paper V [5]). We address the scalability problem of the online vertex-cut partitioning algorithms for streaming power-law graphs. We propose HoVerCut, a framework for boosting the partitioning of streaming graphs. It can employ different partitioning heuristics in a scalable fashion. HoVerCut utilizes an efficient tumbling window model to process edges of the graph and efficiently shares the state information between multiple instances of the partitioning algorithm.

- Efficient representation learning of dynamic graphs using random walks(Paper VI [6]). Our method is based on the state-of-the-art unsupervised vertex representation learning, which contains two steps: (i) random samples (corpus) are generated by executing truncated random walks on a graph, and (ii) inspired from the state-of-the-art word representation learning technique, the skip-gram model is trained by the generated random samples. We propose computationally efficient algorithms for vertex representation learning that extend random walk based methods to dynamic graphs. The computation complexity of our algorithms depends upon the extent and rate of changes—the number of edges changed per update—and on the density of the graph.

1.5 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 explains the background information and the related works to this thesis. Chapter 3 presents the contributions of this thesis related to stream processing across a geo-distributed infrastructure. Chapter 4 discusses our contributions on two important problems for streaming graph processing, namely graph partitioning and graph representation learning. Finally, Chapter 5 discusses the summary of our work, conclusions, and discusses the future works.

Chapter 2

Background

As explained in Chapter 1, this thesis spans methods and solutions for data-intensive computing with the focus on streams, graphs, and geo-distribution. Therefore, this chapter presents the theories, concepts, and tools used in this doctoral thesis under the sections stream processing, geo-distributed infrastructure, and graph analysis. First, this chapter explains the stream processing paradigm and the related topics including windowing, windowed aggregation, and Apache Storm, a widely used stream processing system in the industry. Next, this chapter gives background information about geo-distributed infrastructure, and Community Network Cloud as a special case of resources at the network edge. In the end, the concepts related to graph analysis including graph partitioning, community detection, and graph representation learning are explained.

2.1 Stream Processing

Stream processing plays an important role in the area of Big Data. In many domains data are generated continuously and hence, there is a need for continuous processing of the data. The core assumption in stream processing is that the data is continuous and unbounded. Stream processing is applicable in almost any domain where we are challenged with the velocity and volume of Big Data. For example, it can be applied in stock markets for the fast analysis of market data [37], and in transportation for intelligent traffic management [38] and navigation [39]. Other applications of stream processing include spam detection [40], fraud detection [41], online detection of denial of service [42], sentiment analysis in social networks [43], and many others [44].

In stream processing, the data in motion is called *data stream*, for example a stream of sensor readings, a stream of tweets, and a stream of bank transactions. A data stream is made of atomic data items each called a *tuple* (Figure 2.1), e.g., a sensor reading tuple, a tweet tuple, and a bank transaction tuple. A source of a data stream is called a *streaming data source* [14]. For example, a temperature sensor is a streaming data source that continuously emits temperature data at a specific rate. Some other examples of data sources include web browsers, sensors such as traffic and sound sensors, mobile devices, network and telecommunication switches, and server logs.

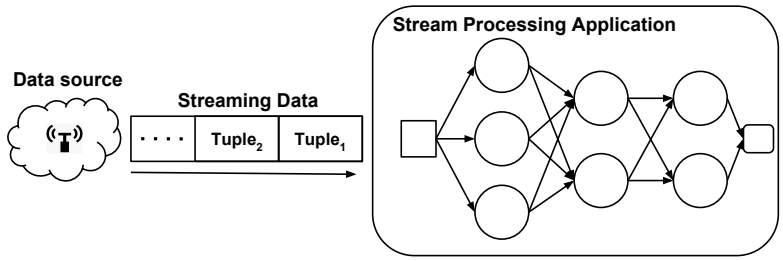


Figure 2.1: A demonstration of a data stream and a stream processing application.

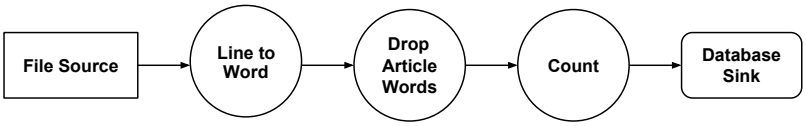


Figure 2.2: A dataflow graph for the word count example.

An application that processes one or multiple data streams is called a *stream processing application* (Figure 2.1). The dataflow in a stream processing application is usually expressed as a graph (usually directed acyclic) called a *stream processing graph* or *dataflow graph*. Nodes of the graph include *operators*, *sources*, and *sinks*. An operator has an input channel to receive tuples and may generate outputs and contains some basic function or complex logic unit that applies on the tuples. For example, in the case of a simple fire alarm application, a filter operator takes temperature tuples as input and drops those tuples with the temperature value below a specific threshold. The filter operator outputs a fire signal only if the temperature is above the threshold. The source enables reading a data stream from outside of a stream processing application, e.g., from a file system or a message broker. The source receives data and acts as an adaptor to generate the data as a stream of tuples. The sink is an endpoint that sends output results to an external service, e.g., a database or a monitoring service. Similar to the source, a sink also acts as an adaptor for those external services that consume the outputs of a stream processing application.

Figure 2.2 depicts the dataflow graph of an example stream processing application for counting the number of words in a text file. In this example, there is a source node that reads the file and converts it to a stream of lines. There are three operator nodes. The first operator "Line to Word" converts the lines of texts to words. The second operator "Drop Article Words" ignores article words and only emits non-article words to the next operator. The last operator counts the number of each word and sends the number of each word to the database sink to store the word count results.

The links between the nodes of a dataflow graph can be defined explicitly or implicitly depending on the *stream processing system*, which provides the environment for development and execution of stream processing applications. In the explicit model, the dataflow graph of a stream processing application is built by explicitly connecting the elements of the graph, i.e., sources, sinks, and operators. Apache Storm [17] is an example of a stream processing system that supports the explicit model. The implicit model provides a high

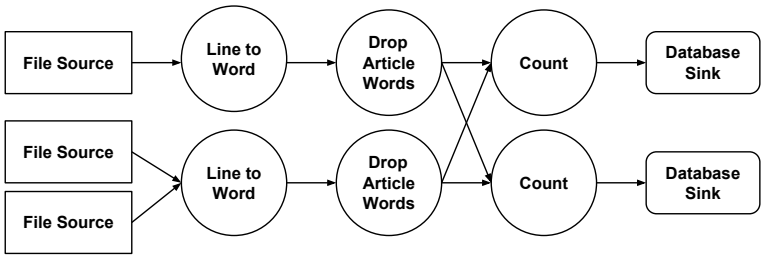


Figure 2.3: An example of an execution graph.

level abstraction in order to define the dependencies between the operators and types of the data streams as their input and output. Then, the stream processing system builds the graph in run time. Flink [21] and Spark [45] are two stream processing systems that support the implicit model.

Every stream processing system offers a runtime environment in order to execute stream processing applications. The common architecture of the runtime environment in the contemporary distributed stream processing systems is the *master-worker* architecture [17, 21, 46, 47]. The primary task of the *master* is to distribute and deploy stream processing applications across the *workers*. Upon the submission of a stream processing application to the master, the master converts the dataflow graph of the application to an *execution graph*. Each node of the execution graph is a *task* corresponding to a node in the given dataflow graph. For each node of a dataflow graph there can be multiple task instances in the corresponding execution graph. The tasks run in parallel and the number of parallel tasks is configurable in order to scale the application according to a workload. The configuration of the number of parallel tasks depends on the stream processing system. Depending on a stream processing system, the parallelism for nodes of a dataflow graph can be either directly hinted or it can be dynamically adjusted by the system depending on a workload. To execute a stream processing application, the master uses a *scheduler* that allocates the tasks to the workers. The scheduler may assign multiple tasks to one worker or multiple workers and may distribute the workers among several machines.

Figure 2.3 shows an example execution graph for the dataflow graph shown in Figure 2.2. In the execution graph (Figure 2.3), the File source runs with three parallel tasks, and operators Line to Word, Drop Article Words, and Count each run with two parallel tasks. Figure 2.4 demonstrates a sample deployment of the word count execution graph on two workers. The figure also depicts the inter-worker dependencies among the tasks, which incur communication over the network between the workers. The all-to-all dependency between the Drop Article Words operator and the Count operator is because the Count operator is split into two tasks. Each of the two tasks is responsible for counting a set of words.

2.1.1 Windowing

Windowing is an important concept in stream processing. Windowing allows the stream processing operators to temporarily store input tuples in a buffer called *window* before pro-

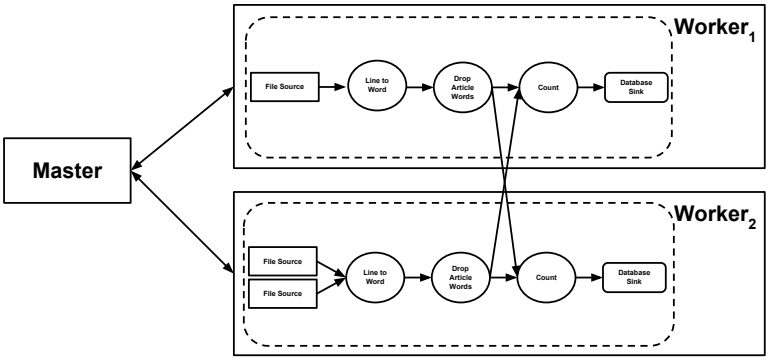


Figure 2.4: The master-worker architecture.

cessing the tuples. This temporary buffer enables to split an unbounded data stream into a smaller batch of tuples. Windowing is especially useful in aggregation queries, for example, standing queries such as top 10 songs listened in the last hour or the most frequent event in every 1000 events. Many stream processing systems support the windowing semantics in their APIs [21, 45, 46, 48]. On the other hand, there are stream processing systems that do not explicitly provide windowing primitives but it is possible to implement user-defined windowing logics [17].

A window can be defined based on the number of tuples that the window includes before processing the tuples, *count-based window*, or based on the time of the tuples, *time(-based) window*. A count-based window triggers computation on the buffered tuples when the number of tuples reaches a certain threshold. A time-based window triggers the computation based on a specified elapsed time from the start time of each window.

Two of the most common window types are *tumbling window* and *sliding window*. A tumbling window splits a stream into non-overlapping windows. This means that there is no overlap between the sets of tuples buffered in the two consecutive tumbling windows. In case of a sliding window, consecutive windows can overlap and hence, there may be tuples that belong to multiple windows.

Both tumbling window and sliding window can be either time-based or count-based. A time-based tumbling window is defined by a time parameter, e.g., every 30 seconds, or every 1 hour. But a count-based tumbling window is defined by the number of tuples, e.g., every 1000 tuples. Both, time-based and count-based sliding windows, have two parameters: the window length and the trigger interval. For example, a time-based sliding window with a length of 30 seconds and the trigger interval of 1 second allows processing every second the tuples buffered in the last 30 seconds.

2.1.2 Windowed Grouped Aggregation

Aggregation of data streams has been widely used in streaming analytics [49, 50]. For example, data stream aggregation is supported and provided in commercial services such as Amazon Kinesis Analytics [51], and Azure Stream Analytics [52]. Some examples of aggregate operators are counts, sum, min, and max. As mentioned in Section 2.1.1,

windowing is mainly used in stream processing so that an unbounded stream data is split into finite windows of tuples. Based on that, a *windowed aggregation* computes aggregates of the tuples for each window. In addition, it is common to group tuples by one or a set of tuple attributes and to compute aggregates for each group separately. For example, a set of tuples may be grouped by their attributes such as city, country, age group, gender, bandwidth class, and etc. More formally, a *windowed grouped aggregation* is defined by an aggregate operator, a window, and a group-by clause. The group-by clause defines one or a set of attribute(s) that the tuples with the same value for the given attribute(s) are grouped together.

Some of the aggregate operators are associative and commutative, such as sum, min, and max. This means that the aggregate can be computed by applying the aggregate operator on tuples in any order. Both associative and commutative properties are desirable because these properties enable to distribute and to parallelize the computation among multiple machines. For example, each machine can compute a partial aggregate on a subset of tuples, e.g., a partial sum, and then send the partial aggregate to a single machine to compute the final aggregate, e.g., the sums of partial sums. However, some of the aggregation operators such as average that are not associative cannot be partitioned in the straight way as explained. But it is still possible to partition and distribute the computation of intermediate results. For example, in the case of average aggregation, each machine can compute the sum and count of tuples in its partition and then send the result to a single machine to compute the average by having all the partial sums and counts.

Due to the importance of data stream aggregation, there have been many works on efficient implementation of this widely used operation. One series of research works aim to reduce the redundant computations that occur in aggregation with sliding windows [49,53–55]. Another series of research works aim to optimize execution of aggregations between multiple queries [49, 56–59]. Multiple aggregation queries may share redundant data and computation due to their overlapping windows, predicates, or group-by clauses. In some applications, the exact aggregate results are not required and therefore it is possible to degrade the result when the processing resources are scarce to deal with a high workload. This situation could happen for example because of a spike in the input stream rate. There have been several works [60–63] on designing load shedding algorithms to remove some parts of data while keeping the degradation of output data low. In Paper III, we propose a solution for optimizing windowed aggregation in geo-distributed infrastructure.

2.1.3 Distributed Stream Processing with Apache Storm

Apache Storm [17] is an open-source, distributed, and scalable stream processing system widely used in the industry. In Paper I we evaluate Apache Storm on a geo-distributed infrastructure and in Paper IV, we augment our solution on Apache Storm.

The dataflow graph in Storm's terminology is called a *topology*. The topology is made of two types of nodes, the *spouts* and the *bolts*. A spout is similar to the source node in a dataflow graph. The bolt can be either an operator node or a sink node. Each node in the topology, depending on its *parallelism* parameter, can run with one or multiple instances of the task.

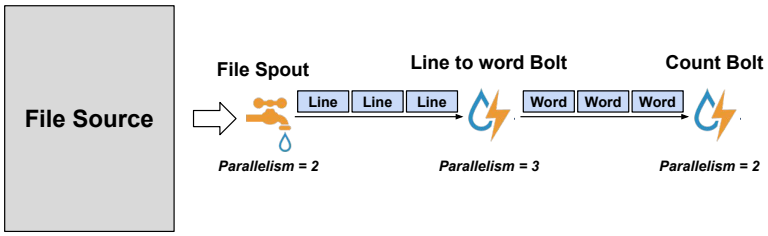
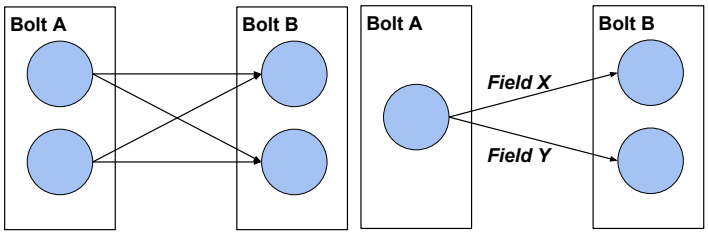


Figure 2.5: An example of a Storm topology for counting the number of words in a file.



(a) Shuffle grouping (b) Field grouping

Figure 2.6: A demonstration of two types of stream groupings.

Figure 2.5 shows an example of the Storm topology for counting the number of words in a text file. In this topology, a spout called File Spout reads the file and transforms its content to a stream of tuples each wrapping a line of the text. The parallelism is set to two for the spout so that there will be two instances of the spout tasks running in parallel. File Spout sends the tuples to the bolt that splits each line to words and sends them as tuples to the next bolt to count the number of the occurrences for each word.

The tasks are executed concurrently by multiple threads on a predefined number of workers. Each thread is called an *executor* [64] and each worker is a process, which can host one or multiple executors. In Storm, the parallelism of each node of a dataflow graph can be defined manually. The *stream grouping* specifies how a stream should be partitioned among the parallel tasks of a bolt. Storm offers some predefined stream groupings and also supports the implementation of custom stream groupings. Two of the predefined stream groupings in Storm are as follows:

- **Shuffle:**Tuples are uniformly randomly distributed across a bolt's tasks (Figure 2.6(a)).
- **Fields:**Tuples are partitioned by a specified field. Tuples with the same value for that field will always go to the same task (Figure 2.6(b)).

Storm has a master-worker architecture (Figure 2.7). The master is called *Nimbus* and each worker is called a *Supervisor*. Nimbus is responsible to deploy stream processing application topologies across supervisors. Nimbus employs a *scheduler* to plan the execution of tasks. The default scheduler of Nimbus uses a round-robin strategy, which aims to evenly distribute the tasks among the Supervisors. Each Supervisor is instructed by Nimbus

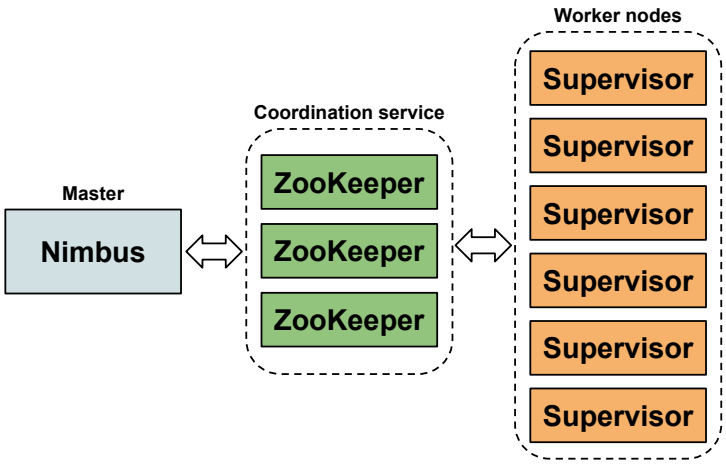


Figure 2.7: The architecture of Apache Storm.

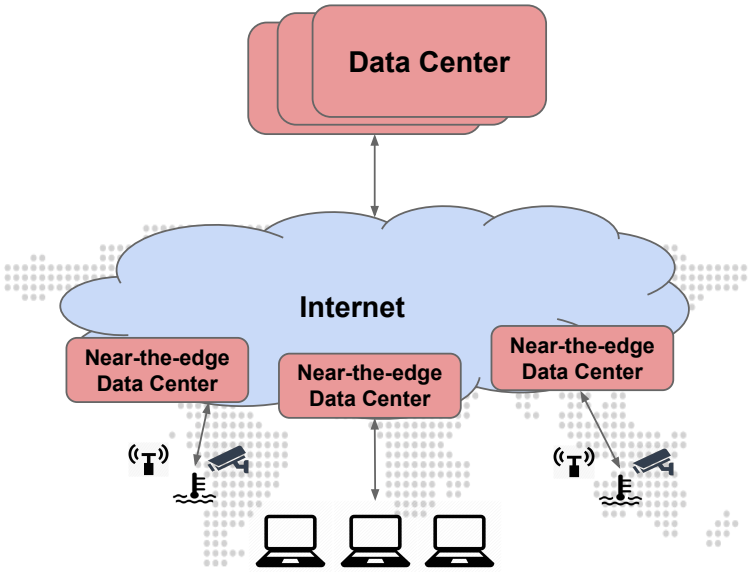


Figure 2.8: A demonstration of the two-tier geo-distributed infrastructure.

and manages processes (workers) hosted in its machine. Nimbus and Supervisors exchange coordination messages through *ZooKeeper* [65], a reliable distributed coordination system.

2.2 Geo-Distributed Infrastructure

A *geo-distributed infrastructure* consists of a set of geographically distributed resources that can host Internet-based applications and services. An example of a geo-distributed infrastructure is a group of data centers located in different countries and continents. One of the main reasons to use a geo-distributed infrastructure is to improve the quality of service

by hosting applications closer to end-users. Placing applications closer to end-users will reduce the network latency caused by distance and routing.

A geo-distributed infrastructure can have a hierarchical architecture with at least two tiers. In a geo-distributed infrastructure with the hierarchical two-tier architecture, the first tier consists of central data centers and the second tier consists of *near-the-edge* data centers (Figure 2.8). Two examples of near-the-edge data centers are Cloudlets [66] and micro data centers [67]. In the literature, the term Fog computing [68] has been extensively used to refer to distributed computing on near-the-edge and edge devices that can host applications. Central data centers are large data centers that are farther from the network edge but have considerably more available resources than near-the-edge data centers. As we move from the first tier toward the lower tiers in a multi-tier distributed infrastructure, the distance to the network edge is reduced and hence, the network latency is reduced. It is usually the case that the compute and storage resources are more limited in the lower tiers of a geo-distributed infrastructure.

Recently, there have been many research works aimed at reducing the network cost and latency of data-intensive computing applications over geo-distributed data by distributing the application components over geo-distributed infrastructure. However, most of the research work has focused on the optimization of batch data processing [26, 69–73], in which the data is assumed to be non-continuous and finite.

There have been few works on optimizing data stream processing in geo-distributed infrastructure. Some works [74–76] are based on the assumption that the approximated output results are acceptable and therefore, propose load-shedding or adaptive result degradation techniques. Heintz et al. [77] provide a solution to optimize network cost and latency in windowed aggregation queries without any result degradation. There are also a series of research works [78, 79] that provide solutions for the placement of stream processing operators on arbitrary networks with the aim of reducing the bandwidth consumption and response time.

This thesis proposes a method that coordinates windowed aggregation among edge data centers to optimize the network cost (Paper III). Furthermore, this thesis presents SpanEdge (Paper IV) that provides a run-time system to manage stream processing applications and an expressive programming model to unify programming on a geo-distributed infrastructure.

2.2.1 Community Network Cloud

Section 3.1 of this thesis presents a case study of enabling stream processing on a geo-distributed infrastructure built on top of a community network. Therefore, this section provides background information about community networks and Community Network Clouds.

Community network is an IP-based bottom-up network built by the citizens of a community and therefore, the topology of the network is self-provisioned. It is built of heterogeneous hardware that are geographically distributed across an area. Community networks often emerge as a solution for providing Internet access to the areas abandoned by commercial telecoms [36]. Examples of community networks are Guifi.net [80] in Spain, AWMN [81] in Greece, and Funkfeuer [82] in the German region.

Community Network Cloud is an infrastructure built on top of a community network in order to provide Cloud services [83, 84]. In Community Network Cloud, Cloud resources are owned by community members. The resources are mainly co-located with the community network devices. Due to the proximity of the Cloud resources to end-users, these resources can be considered as the edge resources in order to host low latency services.

Stream processing in Community Network Clouds share several similar network-related challenges with stream processing in geo-distributed infrastructure, especially, edge resources. Geographical distribution of the resources and network heterogeneity among the resources in terms of bandwidth and latency are some of the challenges. In Paper I and Paper II, we conduct our evaluations on a dataset that we collected from community network Guifi.net [80].

2.3 Graph Analysis

As mentioned in Section 1.2, one of the objectives of this thesis is to provide solutions for the efficient processing of streaming data modeled as a graph data structure. In addition, in Paper II, we leverage graph analysis techniques to optimize the network performance for distributed data-intensive applications. Therefore, this section covers the background information and related works about graphs and graph mining techniques in the scope of this thesis.

Graph or network data are fundamental structures that occur naturally in the real world, for example, social networks, protein networks, communication networks, and the world wide web. Graphs are the natural representation of a collection of entities and the relationships between them. In graph theory, an entity is usually called a *vertex* or *node* and a relation between two entities are presented as an *edge* or *link* between the two corresponding vertices. Graph's vertices and edges can have attributes that represent some characteristics of corresponding entities and their relations. For example, in case of a graph of a social network, the vertices that represent the social network's members (users) can have attributes such as age, gender, and location while the edges can have attributes such as a friend, spouse, sister, colleague, and so on.

Graphs can be *directed* or *undirected*. In a directed graph the direction of an edge is only in one direction from the source vertex to the target vertex. However, in an undirected graph, each edge is bidirectional between the two end vertices. Graphs can also be *weighted* or *unweighted*. In a weighted graph, a value (weight) is assigned to each edge that defines the importance of that edge. The weight can be derived from the edge attributes such as cost, capacity, and etc. However, in an unweighted graph, all edges have the same importance (weight).

Graphs as structured knowledge repositories already encode useful information about their elements. For example, in a social network, vertices that are directly connected by an edge can be considered as friends. One can further analyze to find common friends among two vertices by a simple algorithm. However, graph analysis is more powerful than simply finding common friends. It enables us to decode more interesting and deeper insights from a given data. Graphs are being used in several domains such as optimizing routes for airlines,

recommendation systems, detecting crimes, social network analysis, and genomics.

The rest of this section explains streaming graphs and dynamic graphs that are the main focus of this thesis. It follows by explaining three of the important topics in graph analysis namely *graph partitioning*, *community detection*, and *graph representation learning*. Graph partitioning and vertex representation learning are two of the important stages in large scale graph analysis that are aimed by Paper V and Paper VI respectively. In our research on partitioning geo-distributed resources to improve the network performance (presented in Paper II), we use a novel technique based on community detection, which is primarily used for social network analysis.

2.3.1 Streaming Graphs

In a *streaming graph*, the graph elements are processed as they are being streamed [85]. There are two different models for streaming graphs. A graph can be either streamed by edges or by vertices. In the streaming by edge model, each tuple includes an edge with its two end-vertices whereas in the streaming by vertex model, each tuple includes the data of a vertex and its adjacent vertices.

The order in which the graph elements are streamed is called *stream ordering*. Three common stream orderings include random, breadth-first search (BFS), and depth-first search (DFS) [86]. In the random ordering, the graph elements are streamed by a random permutation of the graph elements. In BFS and DFS, the ordering is generated by selecting an element of each connected component of a graph uniformly at random and then the ordering is given as a result of a BFS or DFS starting from the given element accordingly.

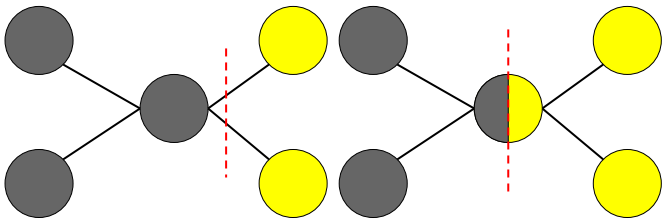
2.3.2 Dynamic Graphs

In the streaming setting, a graph can be seen as an unbounded stream of insertions, deletions, and updates of the graph elements. Therefore, the graph elements and the graph structure can continuously change over time. For example, users of a social network actively develop their connections, new web pages and new hyperlinks between the web pages are created over time, and new bank accounts are created and new transactions between the accounts are made every day.

Specifically, a graph that changes over time is called a *dynamic graph*. A dynamic graph can be represented as a series of graphs $G^t = \{V^t, E^t\}$, where $V^t = \{v_1^t, ..., v_{n(t)}^t\}$, edges $E^t = \{e_1^t, ..., e_{m(t)}^t\}$, and t is a discrete series of times. A set of updates on the elements of graph G^t creates a new snapshot of the graph at $t + 1$, graph G^{t+1}. Efficient analytics for dynamic graphs require to take an incremental approach that can update the analytic's model based on the changes in the graph.

2.3.3 Graph Partitioning

Graph partitioning is a technique that is used for any application that involves distributing large graphs across disks, machines, or data centers. The graph partitioning problem is formulated as dividing a graph into a predefined number of balanced partitions (subgraphs)



(a) Edge-cut partitioning. (b) Vertex-cut partitioning.

Figure 2.9: Graph partitioning models.

while minimizing cut cost [87]. The cut cost refers to the number of connections between the graph elements of different partitions. For example in partitioning a graph among several machines for distributed processing, the cut cost determines the extent of the network communication across the machines.

There are two approaches to graph partitioning, namely *edge-cut* and *vertex-cut* partitioning. The edge-cut partitioning divides vertices of a graph into disjoint partitions of nearly equal size and tries to minimize the number of edges across the partitions (Figure 2.9(a)). The vertex-cut partitioning divides edges into balanced partitions while minimizing the number of replicated vertices (Figure 2.9(b)). The vertex-cut partitioning creates better partitions than the edge-cut partitioning in case of the *power-law* graphs [88]. In a power-law graph the majority of vertices are a relatively low degree (have few neighbors), while a small fraction of them are the high degree (have many neighbors) [89]. Therefore, in a power-law graph, cutting high-degree vertices with many neighbors allows achieving better balanced partitioning compared to cutting edges.

Streaming graph partitioning is an approach for dividing graph elements among the partitions as graph elements are received continuously over time. In fact, the goal is to find good partitions with as little computation as possible [86]. In contrary to the traditional graph partitioning methods, the streaming graph partitioning does not have a global knowledge about the structure of the graph. Lack of global knowledge about the graph structure makes it more challenging to make good partitioning, i.e., balanced partitions with minimum connections between the partitions.

Most of the recent works on streaming graph partitioning lie in the category of edge-cut partitioning algorithms. The research works [33,86,90] propose one-pass edge-partitioning algorithms. Nishimura et al. [91] show that they can achieve a better performance than the contemporary one-pass algorithms by re-streaming the data and improving graph partitions iteratively. Existing streaming vertex-cut partitioning algorithms can be grouped into two main categories: *hashing algorithms* and *greedy algorithms*. Hashing algorithms [92–94] ignore the history of the edge assignments and rely on the presence of a predefined hash function, while the greedy algorithms [92, 95, 96] use the history of the edge assignments to make the next decision. Many of the aforementioned graph partitioning techniques are integrated into several distributed graph processing systems such as PowerGraph [92] and Apache Spark's Graphx [88]. In Paper V, we address scalability problem in the state-of-the-art streaming vertex-cut partitioning algorithms and propose a framework for scalable

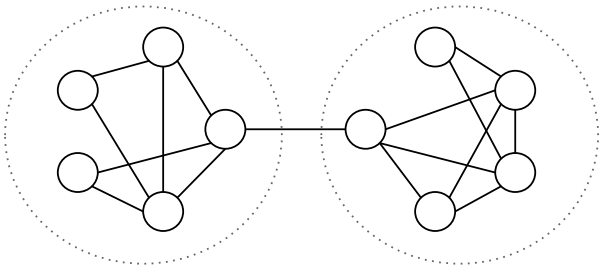


Figure 2.10: An example of a graph with two communities.

and parallel vertex-cut partitioning of streaming graphs.

2.3.4 Community Detection

A *community* in a graph is a subset of the vertices in the graph that the density of the connections among them are much higher than the rest of the vertices in the graph [30]. The notion of community reveals meaningful information about the structure of graphs. For example in a social network graph, vertices that belong to a community can be part of groups, families, and friends. Another example is communities of a web graph that are groups of pages having topical similarities. Figure 2.10 illustrates an example of a graph with two communities.

The identification of communities is possible only for sparse graphs, where the number of edges is considerably higher than the number of vertices and the distribution of edges are heterogeneous. Detecting communities in a graph is an NP-hard problem and there has been a lot of research in this area [30]. The idea in *community detection* is to cluster vertices into communities without having knowledge about the number of communities. The main difference between community detection and graph partitioning (see Section 2.3.3) is that in graph partitioning the number of groups is given to the algorithm as input, but in case of community detection we expect the algorithm to provide this information in its output. This is why graph partitioning algorithms cannot be used for community detection.

There are several methods to measure the quality of the communities identified by a community detection algorithm. The *modularity* is the most popular quality function [30]. The modularity quality function is based on the idea that random graphs are not expected to have cluster structures. Therefore, we can reveal the possible existence of a community by comparing the actual density of edges in a subgraph and the expected density of the subgraph regardless of the community structure [97]. The expected edge density is calculated based on a selected *null model*. A null model is a graph that matches with the original graph in some structural properties but without the community structure. Therefore, the modularity can be written as follows:

$$Q = \frac{1}{2m} \sum_{c} \sum_{i,j \in V_c} (A_{ij} - P_{ij})$$

where the first sum runs over all communities c, V_c is the set of all the vertices in community

c, A is the adjacency matrix, m is the total number of edges, P_{ij} is the expected number of edges between vertices i and j in the null model. Even though in principle one can choose any arbitrary null model, a null model with the same degree distribution of the original graph is preferable. Modularity can be easily extended to weighted and directed graphs. For example, in the case of a weighted graph, the degrees of the vertices can be replaced by their strengths, i.e., sum of the edge weights.

Modularity-based methods [98, 99] are by far the most common methods for community detection [30]. This category of methods directly or indirectly use modularity and the motivation is to partition the graph such that to maximize the modularity. In Paper II, we propose to use the modularity metric. We propose a novel decentralized community detection method to detect resource "communities" in a graph built of near-edge resource nodes that increases the modularity metric.

2.3.5 Graph Representation Learning

Incorporating structural information about graphs into machine learning models is the major problem in machine learning on graphs. This is unlikely in image processing systems, where images can be simply presented by vectors of raw pixel-intensities and we know that all the information that are needed to recognize objects in images are encoded in the vectors. However, in the case of vertex (node) classification on graphs, it is not clear how to encode local neighborhoods of a vertex or its global position in the graph into a feature vector [31].

Feature engineering has been long studied for vertex feature extraction. However, feature engineering requires domain experts and it is often done by handcrafting features and using feature extraction techniques [100, 101]. In general, handcrafted features are limited and inflexible, and designing them can be time-consuming and expensive [31].

Recently, there has been a growing interest in general purpose methods for graph representation learning, which is to learn latent, low-dimensional representation (representations) of graph vertices while preserving the graph topology structure and other useful information [102]. Specifically, the graph representation learning learns a d-dimensional vector for each vertex, where d is much smaller than the number of vertices. The representations can be used as the features for vertices in many downstream machine learning tasks such as classification of vertices, clustering, and link prediction [103, 104]. The existing graph representation learning techniques can be categorized into two groups of *unsupervised* and *semi-supervised* representation learning [102]. Unsupervised representation learning techniques aim to learn the representations only based on the graph structure, i.e., vertices and edges. However, in semi-supervised techniques, the assumption is that the vertices are labeled so beside the graph structure they can leverage the labels in learning representations.

Contemporary graph representation learning methods can be grouped in five categories of *matrix factorization based*, *edge modelling based*, *deep learning based*, *random walk based*, and *hybrid* [102]. Matrix factorization based methods [105, 106] represent graphs in the form of matrices and obtain representations by means of matrix factorization. These category of methods are memory intensive and computationally expensive and their scala-

bility is a major bottleneck. Edge modeling based methods use vertex-vertex connections to learn graph representations. For example, LINE [107] captures the first order and second order proximities among vertices. Since these methods only capture the observable vertex connectivity information, they cannot capture the global structure of graphs. Deep learning based methods [108, 109] apply deep learning techniques such as autoencoder [110] for learning network representations. Random walk based representation learning methods create a collection of vertex sequences by means of truncated random walks. Random walk based methods leverage the recent advancement in word representation learning [111, 112] to learn the representations. Specifically, in random walk based methods, the sequences of vertices are treated as sample sentences extracted from text documents. DeepWalk [113] and node2vec [114] are two of the random walk based representation learning algorithms. Among different categories of graph representation learning from the algorithmic perspective, random walk based methods are one of the scalable methods. Hybrid methods [115] use a combination of the other four aforementioned methods.

In Paper VI, we propose an incremental method for graph representation learning that can learn graph representations competitive to state-of-the-art methods DeepWalk [113] and node2vec [114] while being several times faster.

Chapter 3

Optimizing and Unifying Stream Processing over Geo-Distributed Infrastructure

THIS chapter targets some of the important issues related to stream processing across geo-distributed infrastructure. Resource nodes of a geo-distributed infrastructure are connected through heterogeneous network connections. This means that the network bandwidth, latency, and even the traffic cost can considerably vary between every two resources of a geo-distributed infrastructure. The network heterogeneity arises new challenges for stream processing. The network heterogeneity complicates the process of placement of stream processing tasks on the resources. A naive non-optimal placement of the components on the resources can lead to severe performance penalties by creating performance bottlenecks and over-utilizing some network links. Furthermore, optimizing the amount of data being exchanged between the resources becomes more important in such a heterogeneous network. To that end, novel solutions are required to optimize allocation of resources and inter-task communication in order to improve stream processing performance and to provide required quality of service at a minimal cost.

This chapter first summarizes some of the problems related to the deployment of a stream processing system on a geo-distributed infrastructure made of near-the-edge resources connected with heterogeneous networks. Second, this chapter explains a novel solution based on community detection in order to partition the near-the-edge resources into clusters based on their network performance. It follows with the case of windowed grouped aggregation and the proposed coordination algorithm that optimizes communication cost for data aggregation. In the end, this chapter explains a novel approach to unify stream processing across central and near-the-edge data centers. The content of the next sections are taken from Paper I, Paper II, Paper III, and Paper IV respectively.

3.1 Evaluation of Apache Storm in a Geo-distributed Infrastructure

In Paper I, we address some of the main issues and bottlenecks of deploying a stream processing system on a geo-distributed infrastructure made of near-the-edge resources, where resources that host the system's components are connected by heterogeneous network links. We take an experimental study approach in this regard. We have evaluated Apache Strom [64], a widely used open-source distributed stream processing system, on an emulated Community Network Cloud [83]. We have chosen the community network as a geo-distributed infrastructure because of its similarity to other means of hosting applications near-the-edge, in which the network topology and connections are heterogeneous. Another reason to use the emulated Community Network Cloud as a test-bed was availability of real-world dataset of community network Guifi.net [80]. The dataset that we collected from a small part of the network, named QMP Sants-UPC [116] allowed us to perform realistic emulation by using the CORE network emulator [35]. The network contains 52 nodes and 112 bidirectional links.

In Paper I, we consider different placements of the Apache Storm components on the geo-distributed resources. The evaluation exposes the following requirements for the stream processing system in order to be deployed and operate in a geo-distributed infrastructure, as well as, the challenges related to the underlying network connectivity:

- Different placements of the Storm components on the distributed resources, can lead to different tasks scheduling time, failure detection time, failure detection accuracy, throughput, and the traffic overhead on the network. This is mainly because in a geo-distributed infrastructure, network links are heterogeneous, some links can be over-utilized, and the number of hops between the communicating resources can vary. In order to facilitate efficient placement of the stream processing components, we propose a component placement method based on partitioning of the geo-distributed resources, that is described in Section 3.2.

- Stream sources, called spouts in Apache Storm (see Section 2.1.3), can be spread all over the network. Therefore, the placement of bolts closer to their relevant stream sources (spouts) can reduce the network traffic and increase the throughput of the system. It is required for the Storm scheduler to be aware of the network topology and the location of the stream sources in order to assign bolts efficiently to the resources.

- Apache Storm supports stream grouping that specifies how a task should distribute outbound tuples between the parallel instances of the next task in an application dataflow (see Section 2.1.3). However, the existing stream groupings in Apache Storm are inefficient and not expressive for a geo-distributed infrastructure. There is a need for novel stream groupings that can define streams to be partitioned among bolts with respect to the architecture of a geo-distributed infrastructure.

3.2 Smart Partitioning of Geo-Distributed Resources

Near-the-edge resources that can be used to host applications closer to data sources and sinks are often connected by and co-located with network devices such as routers [68], base-stations [117], or community network nodes [83]. As mentioned in the previous section, placement of stream processing components on near-the-edge resources affects the performance of the stream processing system. The problem is that the topology of the network that connects near-the-edge resources is not necessarily optimal for hosting stream processing components or any other distributed data-intensive application. Therefore, a naive non-optimal placement of the components on the resources can lead to severe performance penalties. For example, some links can become over-utilized, creating performance bottlenecks. Data can flow through excessive number of links in the network, which can degrade the performance of the whole network. Furthermore, some stream processing components with high dependency and high data transfer rate may be dispersed among the resources connected with poor links. In fact, all these issues can raise due to the heterogeneous network that connects the near-the-edge resources.

For an optimal placement of data-intensive application components on a geo-distributed infrastructure, knowledge about the underlying network can be exploited. In Paper II, we propose a novel method for partitioning a distributed infrastructure of near-the-edge resources to computing clusters, each called a *micro data center*. Communications within the resources of a micro data center are faster and cheaper than communications across micro data centers and communications between micro data centers and a central data center. Topology-aware distributed applications can increase their performance by exploiting the information about the micro data centers when placing their application components. Techniques such as rack locality have been extensively used inside central data centers for efficient scheduling of data-intensive applications [118,119]. For example, in a central data center, it is more efficient to place components of a distributed application inside the same rack. This is because there is more bandwidth available for the nodes inside a rack and the intra-rack communications are faster.

In Paper II, we model near-the-edge resources and their connections as a weighted graph, where vertices represent near-the-edge resources and edges represent connections between the resources. In the graph model, edge weights represent connection attributes such as network latency and bandwidth. The model enables us to define the problem of partitioning resources into micro data centers as a community detection problem (Section 2.3.4), which is originally used in social network analysis [30]. In particular, the high connectivity inside each micro data center of a distributed infrastructure represented as a graph can be modeled and estimated by the known modularity metric in graph theory [30]. The modularity measures the fraction of edges that fall within the given groups minus the expected such fraction if edges were distributed at random. We propose to use the modularity metric to detect resource "communities" in a distributed infrastructure by clustering near-edge resource nodes that increase the modularity metric. Figure 3.1 shows an example of a graph of geo-distributed resources. In the graph on the left side, the nodes with the same color are grouped as the resources of the same community (micro data center). However, the detection of community structures in a graph is an NP-hard problem and hence, it

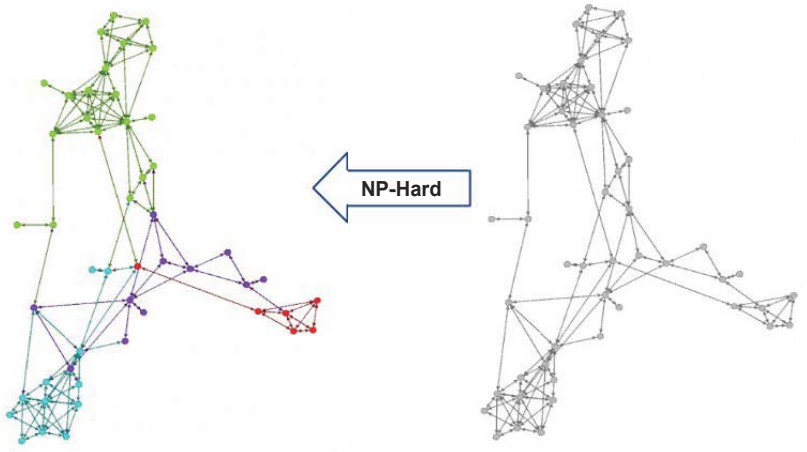


Figure 3.1: Graph of a geo-distributed infrastructure before and after grouping into micro data centers. The visualization does not represent the geographical location of the nodes.

is not feasible to find an exact solution.

To provide a general solution for near-the-edge resources with a centralized or decentralized controller, in Paper II, we propose a novel decentralized community detection algorithm in which each node of the graph only needs to communicate with its adjacent nodes through message passing. The algorithm is an agglomerative algorithm [30], in which bigger communities are built by merging smaller communities. Our solution is based on biased random walk, in which the random walker either stays in its current node or leaves it with a probability that is inversely proportional to its degree. When performing a random walk on a graph, starting from a random source node, the walker is more likely to get trapped in the dense region where the source node belongs to. For example, in Figure 3.2, if a random walker starts from node A and takes a few random steps, with higher probability the random walker will still remain in subgraph G_1. However, if the walker eventually enters subgraph G_2 then it is more likely to stay in G_2 than to leave it.

Inspired by the idea of the random walk, we use multiple random walkers, starting from each node of the graph. Each random walker is identified with a unique color. At the beginning, each node is initiated with a unit of a unique color and in each iteration the node sends out a fraction of its color through its outgoing edges to its adjacent nodes. The amount of color to send out is proportional to the weight of edges. Finally, each node selects the color that has the highest quantity in its neighborhood as its *dominant color*. Nodes that observe the same dominant color are the ones that are located in the same dense region of the graph, i.e., in the same community. More detailed description of the algorithm can be found in Paper II.

Here, we present a summary of the evaluation of the proposed decentralize community detection algorithm by using the dataset of the QMP Sants-UPC network (the dataset is also explained in Section 3.1). Paper II includes the detail of the algorithm and the evaluation settings. We compare three different algorithms: (i) a geolocation-based algorithm using KMeans clustering; (ii) a centralized community detection algorithm integrated in Gephi

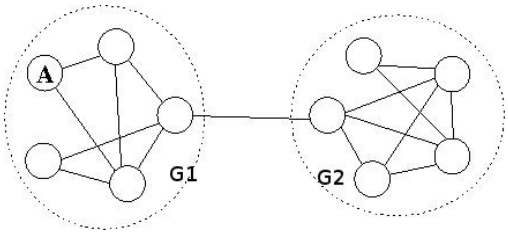


Figure 3.2: An example graph with two sub-graphs. A random walker starting from node A has a higher probability to ends up at a node in subgraph G_1 rather than a node in subgraph G_2.

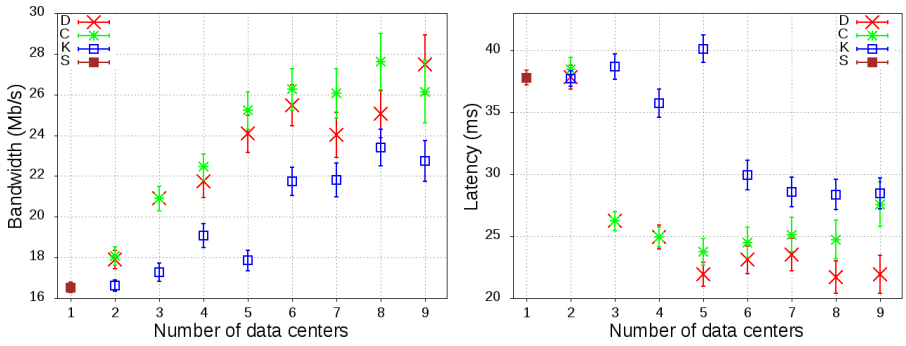
graph visualization software [120] based on [99] and [105]; and (iii) our decentralized community detection algorithm. The centralized and decentralized community detection algorithms are denoted by C and D respectively and the KMeans clustering is denoted by K. We also present the result of considering the whole graph as a single data center, denoted by S. We consider bandwidth as the edge weights and assume that the routing protocol is based on weighted shortest paths among every two node. Community detection algorithms are run to find different number of communities, even though community detection algorithms are not parameterized directly to find exact number of communities. For example, in case of the decentralized algorithm, increasing the number of iterations leads to fewer number of communities.

Figure 3.3 demonstrates quality of the network links in terms of minimum bandwidth and latency inside micro data centers formed using the aforementioned algorithms. It is clear that the decentralized community detection is performing competitive to the centralized community detection, forming micro data centers with higher connectivity, having higher bandwidth, and lower latency compared to other algorithms. The micro data centers obtained from KMeans clustering have considerably lower bandwidth and higher latency. In addition, we can see that if we consider the whole resources as a single infrastructure, we can end up placing our distributed application components among nodes that have very low network connectivity.

Further evaluation results are available in Paper II. According to the results, by placing distributed applications inside micro data centers we can considerably reduce the network overhead and the network latency. Furthermore, the decentralized community detection algorithm, proposed in Paper II, creates clusters with qualities competitive to the centralized community detection method.

3.3 Optimizing the Communication Cost in Stream Data Aggregation in Geo-Distributed Infrastructure

The previous section proposed a solution to mitigate the effect of network heterogeneity on distributed stream processing in near-the-edge resources. The solution is based on grouping the resources into micro data centers with relatively more efficient communication within the micro data centers compared to inter-data center communication. Micro data centers



(a) (b)

Figure 3.3: Mean of minimum available bandwidth (a) and latency (b) between each pair of nodes co-located in the same micro data center when the routing protocol is the shortest path between every two node.

enable us to place interacting stream processing components on the resources of the same micro data center, expecting a better network performance between the resources of a micro data center.

However, inter-micro data center communications and communications between micro data centers and central data centers are unavoidable in many stream processing applications. One of the applications that is common to run on data streams across multiple data centers is *data aggregation*. Stream data aggregation is a widely used task (as a standing or ad-hoc query) in streaming analytics, which is to compute an aggregate, such as sum, average, etc., on a data stream. Stream data aggregation is typically performed as windowed aggregation (Section 2.1.2). Global scale organizations run windowed aggregation queries for constant monitoring and real-time analytics of their data, including user actions, server logs, and sensor readings. Thus, data are constantly being collected on tens to hundreds of geographically distributed edge data centers (such as micro data centers) that are proximate to users.

A common approach for processing aggregate queries of geo-distributed data streams follows a hub-and-spoke model, in which edge data centers stream tuples directly to a central location (*core*). This approach is inefficient because it does not consider heterogeneity of networks among edge data centers and the core. For example, the cost of communication among data centers in the same region, country, or continent is mostly, if not always, lower than inter-region, inter-country, or inter-continent communications. Examples of this phenomenon are evident in pricing policies of Amazon AWS, and Google Cloud.

In Paper III, we propose a solution that coordinates windowed aggregations among edge data centers, and can significantly reduce bandwidth cost. Essentially, edge data centers connected with low-cost links can transfer and aggregate data streams among each other before communicating with the core over expensive links. In Paper III, we define the theoretical minimum bandwidth cost for aggregating data streams by means of coordi-

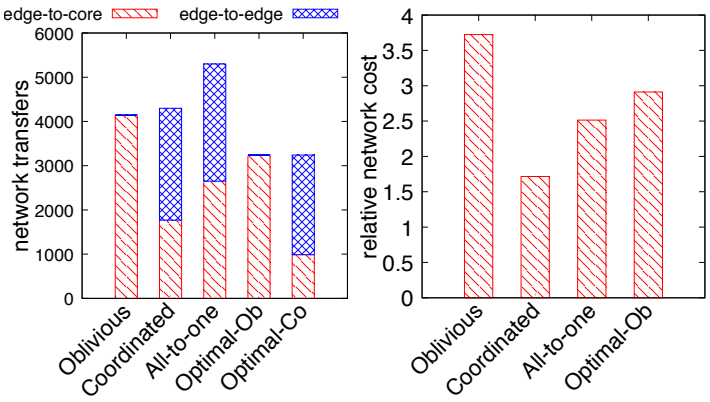
nation. Based on that, we propose a low-overhead coordination method that can identify relevant data among edge data centers and aggregate them effectively and efficiently, and send the data streams among the edge data centers and the core in a timely manner.

In Paper III, we assume that a data stream is a stream of key-value tuples. Our coordinated method uses predicted arrival rates of tuples with identical keys in each time window. Each edge data center obtains the arrival rates by profiling its workload. The edge data centers use the predictions for coordinating aggregations among themselves and deciding the time to emit updates for each key. In general, accurate prediction of workloads requires deep knowledge of the incoming workload and sophisticated selection and tuning of one or several prediction algorithms. Like in previous research [121], we assume that the incoming workload within a window follows a Poisson distribution. Therefore, we predict the expected arrival rate for each key as a weighted average of historical arrival rates.

The edge data centers in a region use our coordination method to aggregate tuples with identical keys. In every region, one edge data center is assigned as a role of *coordinator*. All edge data centers in the region send to the coordinator information about the keys they have observed. Based on this information, for each key, the coordinator selects one of the edge data centers to serve as an *aggregation point (AP)* for that key, and sends the list of APs to all edge data centers in the region. An edge data center assigned the role of AP for a key is responsible to collect and aggregate updates for that key from all data centers in the region that have received tuples with that key. An AP is also responsible for sending aggregate updates to the core. Each edge data center sends aggregate updates for keys with assigned APs to corresponding APs; whereas updates for the keys with no assigned APs are sent directly to the core.

To schedule the updates departure from edge data centers, we use the method proposed by Heintz et al. [121], where the authors model the available bandwidth in a edge data center as a cache. In the cache model, the cache size determines the number of keys that can be held for further local aggregation and the cache size dynamically changes during a window, which is divided into multiple *time steps*. Each edge data center recomputes the cache size and schedules updates at every time step, which may evict a set of keys from the cache and hence, emit their updates. We design the coordinated method such that an edge data center may emit each update either to the core or to an AP.

The rest of this section presents a summary of the evaluation from Paper III. The evaluation compares the coordinated aggregation method with the following four methods: (i) the state-of-the-art *oblivious* aggregation method, in which edge data centers do not interact with each other for computing partial aggregates, and independently optimize the aggregation over data streams [121], (ii) an optimal version of the oblivious method, (iii) a simple method called all-to-one, in which edge data centers send their updates for all keys to a fixed aggregation point and that single aggregation point sends all the updates to the core, and (iv) an optimal version of the coordinated method. The optimal methods (ii) and (iv), which are based on offline batch algorithm processing the entire stream dataset, are not practical and are considered here only for the purpose of performance evaluation of other methods. As a proof of concept, we have implemented a prototype of all the aforementioned methods in Java and we evaluate them by means of simulation. As for the data stream traces, we use the dataset "UserVisits" from Big Data Benchmark [122], and syn-



(a) Network transfers in terms of number of aggregate updates.

(b) Network cost relative to the cost of optimal coordinated method.

Figure 3.4: Comparison of the different aggregation methods with respect to network transfers in terms of number of aggregate updates and network cost.

thetically generated workloads. The details about the stream traces are available in Paper III.

Figure 3.4(a) presents edge-to-edge and edge-to-core network transfers in terms of number of updates and the incurred network cost for the edge data centers in a single region. In the experiment, there are 6 edge data centers in the region and the data set is generated for 1000 unique keys. The keys are generated such that the distribution of identical keys among any possible combination of edge data centers are uniform. Figure 3.4(a) shows that the coordinated method reduces the network transfer to the core by at least two times compared to the oblivious method. Figure 3.4(b) depicts the relative network cost of aggregation incurred by the different methods compared to the traffic cost incurred by the optimal coordinated method. We assume that an edge to core update costs 18 times larger than an edge to edge update. This is the average of inter-region network traffic cost compared to intra-region traffic cost in Google Compute Engine [123]. Figure 3.4 shows that even though the total number of updates are almost the same between the coordinated and the oblivious methods, majority of the updates in the coordinated method are sent through the cheap intra-region network. Therefore, as depicted in Figure 3.4(b), the coordinated method incurs much lower network cost compared to the oblivious method.

3.4 A Unified Stream Processing System over Geo-Distributed Infrastructure

Section 3.1 explained that, in general, streaming data sources and sinks are geographically distributed that calls for distributed stream processing over a geo-distributed infrastructure in order to enable and achieve efficient distributed stream processing. Distributed stream

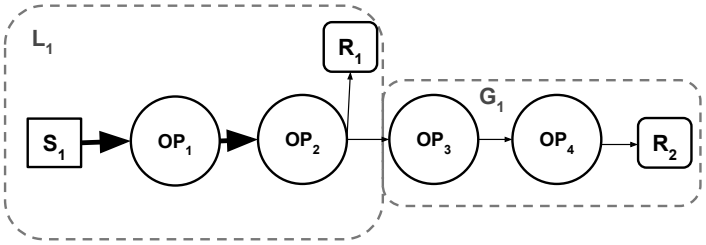


Figure 3.5: The dataflow graph of an example stream processing application and the local and global task groupings.

processing over a geo-distributed infrastructure allows to place components of a stream processing application close to their relevant data sources and sinks in order to reduce the network traffic and improve the application performance. Most, if not all, of distributed stream processing systems such as Apache Storm were designed to be deployed and operate in a centralized data center and are not optimized for a geo-distributed infrastructure with heterogeneous network, compute, and storage resources. For example, as discussed in Section 3.1, the existing stream groupings in Apache Storm are inefficient and not expressive for a geo-distributed infrastructure. Programming and deploying complex stream processing applications across (near-the-) edge data centers and central data centers are challenging. This is because that data sources and sinks are large in numbers, and they are geographically dispersed.

In Paper IV, we propose a solution called *SpanEdge* to tackle the aforementioned issues in enabling and achieving efficient distributed stream processing in a geo-distributed infrastructure. SpanEdge provides an expressive programming model to unify programming stream processing applications on a geo-distributed infrastructure, and provides a run-time system to manage (schedule and execute) the stream processing applications across data centers. In SpanEdge, data centers are categorized into two tiers, where central data centers are in the first tier and edge data centers are in the second tier. The edge data centers are close to data sources and sinks and they are meant to host low-latency services. The data centers in the first-tier and second-tier communicate via WAN links that incur higher response time and traffic cost compared to the communication between the edge data centers in the second-tier and their proximate data sources and sinks.

SpanEdge has a master-worker architecture, which consists of a *manager* as the master and several *workers*. The manager receives a stream processing application and schedules the application's components among the workers as tasks. A worker runs on a cluster of *compute nodes* and executes the tasks assigned to it by the manager. There are two types of workers: the *hub-worker* and the *spoke-worker*, where a hub-worker resides in a data center in the first tier and a spoke-worker resides in a data center in the second tier near the edge. SpanEdge is designed for an ecosystem with multiple heterogeneous streaming data sources that are geographically dispersed. The data sources are different in types depending on what kinds of data they collect, e.g., weather or traffic sensors, and there can be several instances of each source type in different geographical areas.

To facilitate programming stream processing applications for a two-tier architecture,

SpanEdge provides two new operator groupings, *local-task* grouping and *global-task* grouping, which enable development of stream processing applications for both edge and central data centers in a unified environment. In SpanEdge, a stream processing application is presented as a dataflow graph of operators. By means of task groupings, the operators of a stream processing application can be grouped either as a local-task or a global-task, where a local-task refers to a group of operators that require to be close to the streaming data sources; whereas a global-task refers to a group of operators that will be placed on a hub-worker. The operators grouped in a local-task are placed close to the sources from which they consume data. SpanEdge creates a replica of the local-task at each spoke-worker with the corresponding data source types. The task groupings provide a general and extensible model in order to develop any arbitrary operations.

Figure 3.5 shows an example of a dataflow graph of a stream processing application. In this example, there is one type of data source S_1 (e.g., a specific type of sensor, or a camera) and two data sinks R_1 (e.g., an alarm) and R_2 (e.g., a monitoring system). Operators OP_1 and OP_2 (e.g., image compression, tuple sampling, filtering, aggregations, or any arbitrary logic) are grouped as local-task L_1. This means that OP_2 can emit partial local results to sink R_1. Operators OP_3 and OP_4 are grouped as global-task G_1, which receives data from all the replicas of local operator OP_2.

The role of the manager is to deploy the dataflow graph of stream processing applications across the two-tier infrastructure formed of edge data centers in the first tier and central data centers in the second tier. The manager employs a component called *scheduler*, which converts the dataflow graph to an execution graph by assigning the tasks to hub-workers and spoke-workers. To assign the tasks, the scheduler requires three types of information: (i) a dataflow graph, (ii) a map of the streaming data sources to the spoke-workers, (iii) the network topology between the workers. The scheduler uses the map of streaming data sources to the spoke-workers in order to deploy the local-tasks in the spoke-workers with the required data source types. The algorithms of the scheduler are explained in detail in Paper IV.

As a proof of concept, we implemented a prototype of SpanEdge by augmenting Apache Storm [64] with our solution. In order to evaluate SpanEdge in different deployment scenarios, we emulate a geo-distributed infrastructure using the CORE network emulator [35]. We consider a geo-distributed infrastructure consists of 2 central and 9 edge data centers. The designed stream processing application is inspired by Yahoo's Storm performance test [124]. SpanEdge is compared against a standard central deployment architecture for stream processing. Figure 3.6 shows the overall bandwidth consumption in the two different deployment scenarios, namely, central deployment and SpanEdge geo-distributed deployment. The data generation rate is fixed but the number of data sources are increased from 4 to 16. By increasing the number of data sources, the amount of data that needs to be processed also increases. As it can be seen, SpanEdge significantly reduces the overall bandwidth consumption between the data centers. This is because, in the case of a central deployment, the raw streaming data needs to be transferred to the central data centers. While in the case of SpanEdge, the data before being transferred between data centers is already processed near the edge by the spoke-workers. More details about the evaluation are available in Paper IV.

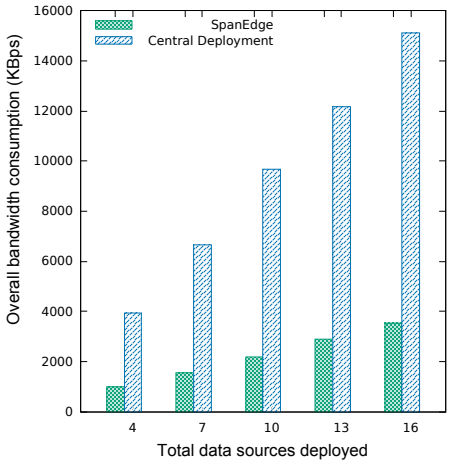


Figure 3.6: The overall bandwidth consumption as the number of data sources increases.

Chapter 4

Boosting the Performance of Streaming Graph Processing

DATA-INTENSIVE computing in many domains deals with data streams that are in the form of interconnected data, i.e. streaming graph data [125]. In a streaming graph, the data stream is modelled as a stream of graph elements, namely edges and vertices. For example, in a social network new users joining the network can be modelled as a stream of vertices and existing users adding friendships can be modelled as a stream of edges. In a streaming setting, the graph is dynamic and its structure changes over time. Effective and efficient processing of streaming and dynamic graphs call for fast online methods and algorithms that process the data streams as they are generated. In addition, well-designed incremental methods and algorithms are required to reduce the computation overheads incurred by the changes in the graph while producing high quality results.

Graph partitioning (see Section 2.3.3) and graph representation learning (embedding) (see Section 2.3.5) are two important operations in processing of large graphs. Graph partitioning is necessary to reduce inter-machine communications in distributed processing of large graphs [86]. Graph representation learning seeks to learn low dimensional vector representations for the graph elements and the whole graph. Graph representation facilitates machine learning tasks and visualization on graph data [31].

This section aims at the two aforementioned problems, i.e., partitioning and representation learning, on streaming graphs. First, this section discusses a summary of our framework for fast partitioning of streaming graphs, which is explained in detail in Paper V. Second, a summary of our incremental method for representation learning of dynamic graphs is explained. The details of our work on graph representation learning is available in Paper VI.

4.1 Boosting Vertex-Cut Partitioning for Streaming Graphs

Vertex-cut partitioning is an effective partitioning method for graphs that have few high-degree vertices and many low-degree vertices, also known as power-law graphs (see Sec-

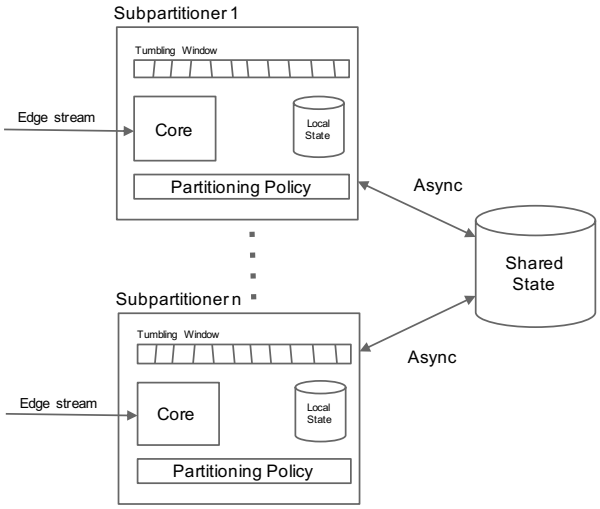


Figure 4.1: HoVerCut's architecture.

tion 2.3.3). Many real-world graphs have power-law degree distribution such as graphs of social networks, Internet, and the graph of the world wide web. In vertex-cut partitioning, edges are divided into nearly balanced partitions while some of the vertices may be replicated among the partitions.

To the best of our knowledge the PowerGraph Greedy algorithm [92] and HDRF [95] have shown the best partitioning results for power-law streaming graphs. Even though these algorithms are online and incremental, they are centralized and do not scale. To the best of our knowledge, the only distributed implementation of these algorithms have been by implementing multiple oblivious partitioners, each partitioning a part of the graph without sharing any information with other partitioners. Even though employing oblivious partitioners scales well by increasing the number of parallel partitioners, but the quality of partitions dramatically drop [126].

In Paper V, we present a parallel and distributed vertex-cut partitioner for streaming graphs, namely *HoVerCut* (Paper V). HoVerCut is scalable while it creates partitions as good as the centralized algorithms. HoVerCut partitions a graph by streaming an exclusive subset of the graph edges to a configurable number of *subpartitioners*. Each subpartitioner uses a windowing technique that brings about a light-weight state sharing with other subpartitioners. HoVerCut decouples the partitioning policy from the partitioning state for efficient parallelism. Each subpartitioner holds a local state, which contains the information required by the partitioning policy. HoVerCut maintains the global system state in a shared storage accessible by all subpartitioners. Subpartitioners read and update the shared state periodically and asynchronously. The more subpartitioners refer to the shared state, the more updated information they receive, and consequently better decisions they make, but the slower the system will be. Figure 4.1 depicts the architecture of HoVerCut.

In Paper V, we present the results of a comprehensive evaluation of HoVerCut. HoVerCut is compared to the state-of-the-art algorithms HDRF [95] and Greedy [92] on open

	Partitioning time [s]				Replication Factor			
Dataset	HDRF	HoVerCut (H)	Greedy	HoVerCut (G)	HDRF	HoVerCut (H)	Greedy	HoVerCut (G)
AS	33	1	28	1	1.99	2.00	2.24	2.24
PSN	72	2	66	3	3.90	3.89	3.89	3.89
LSN	138	5	123	5	2.76	2.76	2.83	2.83
OSN	415	11	371	11	5.57	5.54	5.29	5.29

Table 4.1: Summary of the partitioning results for HDRF, Greedy, HoVerCut(H) and HoVerCut(G). The LSRD is 0.00% in all the experiments.

real-world and synthetic datasets with power-law degree distribution. The four real-world datasets that are used in the evaluation include autonomous systems (AS) [127] with $|V| = 1.7M$ and $|E| = 11M$, Pokec social network (PSN) [128] with $|V| = 1.6M$ and $E = 22M$, Live journal social network (LSN) [129, 130] with $|V| = 4.8M$ and $|E| = 48M$, and Orkut social network (OSN) [131] with $|V| = 3.1M$ and $|E| = 117M$. AS is the dataset of a subgraph of routers comprising the internet. PSN is the dataset of a popular social network in Slovakia. LSN is a subgraph of a free online community that allows its members to maintain journals and blogs. OSN is the dataset of a free online social network.

Table 4.1 summarizes the performance of the different partitioning algorithms on the aforementioned real-world datasets. This experiment is done on a single machine. The window size and the number of subpartitioners are set to 32 for HoVerCut. In the table, the replication factor is the average number of replicated vertices, and the load relative standard deviation (LRSD) is the relative standard deviation of edge size in each partition. The value zero for LRSD indicates equal size partitions. In addition, HoVerCut(H) and HoVerCut(G) represent the results of HoVerCut with the partition policy set to HDRF and Greedy respectively. As Table 4.1 shows, HoVerCut is performing 28 up to 37 times faster than the other two algorithms while the replication factor and the LRSD of the partitions made by HoVerCut is as good as the other algorithms. More details about the experimentation are available in Paper V.

4.2 Efficient Representation Learning Using Random Walks for Dynamic Graphs

An important part of many machine learning workflows on a graph is vertex representation learning, i.e., learning a low-dimensional vector representation for each vertex in the graph. Recently, several powerful techniques [113,114,132] for unsupervised representation learning have emerged, which learn low dimensional vector representations for the vertices only based on the graph structure. These unsupervised representation learning techniques have shown state-of-the-art performance in downstream predictive tasks such as vertex classification and edge prediction. The unsupervised learning is based on two main stages of neighborhood sampling of vertices in a graph and training the skip-gram neural network of [112], which is a model for learning representations of words in natural language processing. The neighborhood sampling is done by executing truncated random walks in a graph. Random walks can capture structural properties of the graph at scale [113].

However, most of the previous research on unsupervised representation learning is based on static graphs whereas most real-world graphs are dynamic, i.e., they change over time. Currently, it is an open research question that how methods based on random walks and skip-gram model can be used for representation learning of dynamic graphs. For a dynamic graph, the random walks must be re-generated for every snapshot of the graph in time, in order to capture the changing graph structure and update the vertex representations, and the downstream predictive model, accordingly.

The workflow for vertex representation learning on a dynamic graph consists of the following steps: (i) update the graph, (ii) generate random walks, (iii) learn vertex representations (embeddings), (iv) train the downstream learning task (e.g., vertex classification). Steps (ii) and (iii), i.e., generating random walks and learning vertex representations, are the most resource-intensive steps in the workflow. In Paper VI, we focus on steps (ii) and (iii) and introduce incremental algorithms with different trade-offs for generating random walks and learning vertex embeddings efficiently.

The baseline solution for neighborhood sampling for dynamic graphs is to re-generate random walks for the latest snapshot of the graph every time the graph is changed. We call this algorithm Static. Static is equivalent to running the first-order/second-order random walk proposed in DeepWalk [113] / node2vec [114]. However, re-generating all random walks on every graph snapshot, is inefficient and incurs unnecessary computation. In Paper VI, we propose two algorithms that can track changes in the graph and their computation cost depends on the extent of the changes. The algorithms consider changes in a graph based on first order affected vertices, which are those vertices that are in the set of added or removed edges in a graph. The formal definition of an affected vertex is explained in Paper VI.

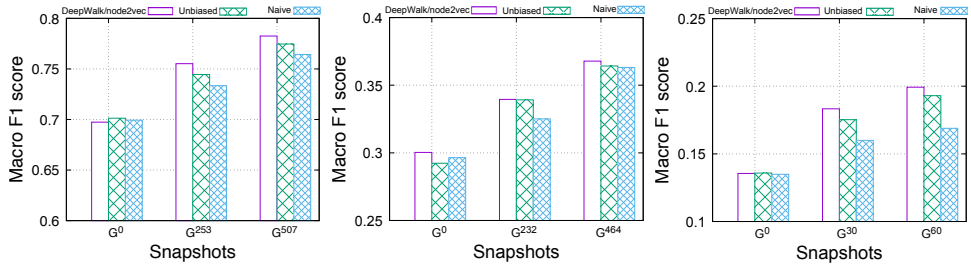
Our first proposed algorithm, called Unbiased Update, is based on modifying the random walks generated on the previous snapshot of the graph (G^t) such that the updated set of walks are statistically indistinguishable from the walks created by Static on G^{t+1}, i.e., re-generating all the random walks. The Unbiased Update algorithm finds all the random walks that have at least one affected vertex, namely *affected walks*, and resumes the random walk for only the affected walks. Unbiased Update ensures that the random walks are unbiased, i.e., statistically indistinguishable from re-generating all the random walks. To that end, Unbiased Update resumes an affected walk from the first occurrence of an affected vertex in that random walk. Our second algorithm Naïve Update, instead of modifying a random walk by searching for affected vertices among random walks, only re-generates random walks for the set of affected vertices. Therefore, Naïve Update has the least computation cost and is faster than Static and Unbiased Update. However, Naïve Update generates biased random walks due to its bias towards the random walks initiated for affected vertices. The details about the computation complexity of the algorithms are available in Paper VI.

The baseline algorithm to learn vertex representations is by training the skip-gram neural network of [112]. Vertex representations are learnt from a random initialization for each vertex. The skip-gram model is trained by the sample target-context pairs generated from a corpus, which is created by random walks. For simplicity, we call this baseline algorithm Retrain. In Paper VI, we propose algorithm Incremental, which trains the skip-gram

model incrementally by initializing the skip-gram neural network variables from the previous snapshot including the corresponding vector representations for each vertex. For a new snapshot of the graph, the skip-gram model is only trained by newly generated samples. Our evaluations show that training the skip-gram model with only the new samples generated from our random walk algorithms can result vertex representations competitive to Static and Retrain but several times faster.

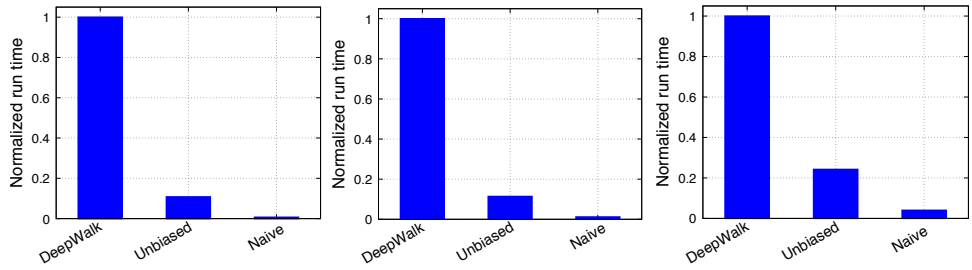
To evaluate our algorithms, we have implemented the random walk methods in Scala, and the skip-gram model in python with TensorFlow framework [133]. This section presents a summary of the results on three real-world datasets Cora ($|V| = 2,485$ and $|E| = 5,069$), Wikipedia ($|V| = 2,357$ and $|E| = 11,592$), and CoCit ($|V| = 42,452$ and $|E| = 194,410$). The detail about the evaluation setup and more evaluation results are available in Paper VI. As in previous research [134, 135], we initiate a graph by randomly selecting a subset of edges and at each step we add a number of randomly selected edges to the initial graph in order to create a new snapshot of the graph. The vertex representations are given to a one-vs-rest logistic regression classifier. The classifier is set to split the train and test data 10 times and we present the mean Macro-F1 score. We evaluate the performance of our incremental algorithms for learning vertex representations, i.e., random walk algorithms Unbiased Update and Naïve Update combined with our incremental method for training the skip-gram model Incremental. We compare the incremental representation learning algorithms against our implementation of DeepWalk [113]/node2vec [114], which re-generates all the random walks and re-trains the skip-gram model, i.e., algorithm Static combined with Retrain.

Figure 4.2 depicts the Macro-F1 score of the downstream classification task using the vertex representations on different snapshots of the networks. We present the scores for the first snapshot and for the two snapshots chosen from the middle and the end of experiment. We have observed the same trend for the results of the other snapshots. Figure 4.3 also demonstrates the total run time of the random walk and the skip-gram training algorithms on the final snapshot of the graphs normalized to the run time of the DeepWalk algorithm. As Figure 4.2 shows, the representations computed incrementally at every snapshot of the graphs are competitive to the representations that are created by processing the whole graph for every snapshot. This is while our incremental algorithms are several times faster than DeepWalk (Figure 4.3). In addition, Unbiased Update results slightly better representations than Naïve Update as Unbiased Update considers all the affected vertices in the expense of more computation time (Figure 4.3).



(a) Cora (b) Wikipedia (c) Cocit

Figure 4.2: Multi-class vertex classification results for different representation learning methods on different datasets. As it can be seen, the performance of our methods are competitive to that of DeepWalk and Unbiased Update performs slightly better than Naïve Update. 9% of labelled data are used for training. The initial number of edges in G^0 for Cora (a), Wikipedia (b), and CoCit (c) are 50%, 10%, and 4% of total number of edges accordingly.



(a) Cora (b) Wikipedia (c) Cocit

Figure 4.3: Run time of different methods normalized to the run time of DeepWalk (Static and Retrain). As it can be seen, our incremental methods are several times faster than DeepWalk. The run time includes the time for generating random walks and training the skip-gram model.

Chapter 5

Conclusion and Future Work

THIS thesis has proposed methods, algorithms, and tools for improving the performance of data-intensive computing, specifically focusing on distributed stream processing in geo-distributed settings, and processing of streaming graphs. With respect to the performance of distributed stream processing in geo-distributed settings, this thesis aims at mitigating the effect of network heterogeneity, reducing the amount of data being transferred over expensive network links, reducing the amount of communication over long latency network links, and facilitating programming stream processing applications for geo-distributed settings. With respect to processing of streaming graphs, this thesis targets to improve the parallelization and state-sharing, and to reduce the amount of redundant computation.

In the rest of this chapter, first, a summary of the results of this thesis on achieving its objectives are discussed. Second, this chapter explains the possibilities, limitations, and environmental aspects of this thesis. In the end, the future work is presented.

5.1 Summary of Results

Most of the existing stream processing systems are designed and optimized to work inside a central data center. To study and detect the limitations of deploying a distributed stream processing system on a geo-distributed infrastructure, we evaluated Apache Storm, an open-source distributed stream processing system, on a geo-distributed infrastructure made of near-the-edge resources. The evaluation exposed some new requirements for the stream processing system in order to be deployed and operate in a geo-distributed infrastructure, as well as, the challenges related to the heterogeneous network that connects the near-the-edge resources: (i) the performance of Apache Storm highly depends on the placement of the Storm components on distributed resources of the geo-distributed infrastructure, (ii) the placement of bolts closer to their relevant stream sources (spouts) can reduce the network traffic and increase the throughput of the system, and (iii) there is a need for novel stream groupings that can define streams to be partitioned among bolts with respect to the architecture of a geo-distributed infrastructure.

To mitigate the network heterogeneity and to facilitate the placement of stream pro-

cessing components on geo-distributed resources, we proposed partitioning of the geo-distributed resources based on community detection. To that end, we modeled near-the-edge resources and their connections as a weighted graph. We call each partition of the resources a micro data center. In particular, we estimated the high connectivity inside each micro data center of a distributed infrastructure by the modularity metric. We proposed to use the modularity metric to detect resource "communities" in a distributed infrastructure by clustering near-edge resource nodes that increase the modularity metric. To provide a general solution for near-the-edge resources with a centralized or decentralized controller, we proposed a novel decentralized community detection algorithm that increases the modularity metric competitive to the state-of-the-art centralized community detection algorithm. We showed that micro data centers increase the minimum available bandwidth in the network to up to 62%. Likewise, the average latency can be reduced to 50%.

Next, we discussed the windowed aggregation of geo-distributed data streams, which is widely used in streaming analytics. We defined the theoretical minimum bandwidth cost for aggregating data streams by means of coordination among edge data centers that receive partial streams from data sources and coordinate their actions to achieve efficient stream aggregation. Based on that, we proposed a novel low-overhead coordination method that identifies relevant data among edge data centers and aggregates them effectively and efficiently, and sends the aggregates among the data centers in a timely manner. We showed that our method reduces the bandwidth cost up to $\sim 6\times$, as compared to the state-of-the-art solution.

In order to provide a unified stream processing system in a geo-distributed infrastructure, we proposed SpanEdge that utilizes near-the-edge micro data centers in order to reduce network communication over WAN links and consequently, to avoid the incurred network latency. SpanEdge categorizes the data centers in two-tiers, where the first tier includes central data centers and the second-tier includes near-the-edge data centers. To facilitate programming stream processing applications for a two-tier architecture, SpanEdge provides two new operator groupings, *local-task* grouping and *global-task* grouping, which enable development of stream processing applications for both edge and central data centers in a unified environment. SpanEdge provides a run-time system to manage (schedule and execute) stream processing applications across data centers. We implemented a prototype of SpanEdge as a proof of concept by augmenting the Apache Storm stream processing system with our solution. Our results show that SpanEdge significantly reduces the bandwidth consumption and the response latency.

With respect to graph data, this thesis has aimed at two important problems in graph processing, namely graph partitioning, and graph representation learning. We discussed the scalability problem of vertex-cut partitioning algorithms for streaming power-law graphs. Our solution, HoVerCut, deploys the state-of-the-art partitioning heuristics in a distributed and scalable fashion without degrading the quality of partitions. HoVerCut decouples the partitioning policy from the partitioning state, and utilizes an efficient tumbling window model to share the state between multiple parallel and distributed instances of the partitioning algorithm. Our evaluations, on both real-world and synthetic graphs, showed that HoVerCut significantly speeds up the partitioning process. For example, HoVerCut partitions a social network graph with 117 million edges about 37 times faster than a state-of-the-art

partitioner.

In the end, we proposed an incremental method for vertex representation learning of dynamic graphs. Our method is based on unsupervised vertex representation learning, which contains two steps. First, random samples are generated by executing truncated random walks on a graph. Second, the skip-gram model is trained by the generated random samples to learn the representation of vertices. We defined the notion of affected vertices that represent changes in a graph when the graph is updated. Based on that, we proposed three incremental and computational efficient random walk algorithms that their computation cost depends on the extent of the changes in the graph. In addition, we proposed an incremental algorithm for learning vertex representations using the skip-gram model. We showed that our proposed algorithms can achieve competitive results to the state-of-the-art static methods while being computationally efficient and 9 times up to 160 times faster.

5.2 Possibilities

In our research on partitioning geo-distributed resources to improve the network performance (presented in Paper II), we introduced a method for grouping geo-distributed resources into micro data centers based on their network connectivity. Our method facilitates network-aware placement of stream processing applications by introducing micro data centers and therefore, improving the performance of stream processing applications.

In the work aimed at unifying stream processing over central and near-edge data center (presented in Paper IV), we proposed SpanEdge, which enables new applications with low-latency requirements, e.g., mission-critical applications that require quick decision making. To that end, application components with low-latency response time requirements are deployed in the second-tier data centers in proximity to the areas where their output results are consumed.

In the research on distributed stream processing in geo-distributed settings (presented in Paper II, Paper III, and Paper IV) one of the main objectives is to reduce the bandwidth cost, which is a significant component of the operating expense (OPEX) among other infrastructure expenses.

With respect to the privacy and data movement limitations, in Paper III and Paper IV, our solutions advocate for keeping the raw data with legal constraints within the area of their jurisdiction, avoiding transferring the raw data to other geographical areas.

In the research on streaming graph data (presented in Paper V and Paper VI), we propose efficient solutions for partitioning and embedding of streaming graphs. Streaming graphs are a large part of the today's typical workloads that many organizations and companies are struggling with, e.g., graphs of social networks, bank transactions, IoT devices, protein interactions, and etc. Our solutions for graph partitioning and graph representation learning boost machine learning pipelines on streaming graphs. Our graph partitioning and graph representation learning methods are parallel and scalable for large graphs.

5.3 Limitations

There are some limitations to this work that can be considered as open problems as well. In the research on partitioning geo-distributed resources to mitigate the effect of network heterogeneity (presented in Paper II), we assume that the computing nodes are homogeneous and have the same amount of resources (CPU and memory). Furthermore, our assumption in the evaluation is that the network is static, even though, the nodes can be dynamically added/removed to/from the network. In our work SpanEdge (presented in Paper IV), it is needed to specifically annotate components of a stream processing application in order to be placed close to the network edge. Another limitation with SpanEdge is our assumption on the amount of provisioned resources on near-the-edge data centers. We assume that near-the-edge data centers are provisioned with enough resources to handle the workload received from their proximate data sources. However, near-the-edge data centers may have limited resources and hence, may require to prioritize applications or use load shedding techniques.

5.4 Environmental Aspects

This thesis by its own nature is a step toward efficient processing of streaming data. This thesis does not rely on any method that could be a threat or could harm society. From the privacy point of view, the methods proposed in this thesis do not rely on exploiting any information of individuals or any sensitive data. For example, in research on distributed stream processing in geo-distributed settings (presented in Paper II and Paper IV), our solutions exploit network data without requiring knowledge about the content of the data being transferred over the network. In research on streaming graph processing (presented in Paper V and Paper VI), our methods leverage meta-data of a graph (the graph structure) and do not rely on data of the graph entities. From the point of view of sustainability and environmental impact, our solutions presented in this thesis optimize network consumption and reduce computation, which reduce energy consumption. Furthermore, by improving the response time of stream processing applications, this thesis enables novel mission-critical applications that can reduce risks and can increase safety in many areas such as traffic management and smart grids.

5.5 Future Work

The results of this thesis encourage to extend the research works discussed in this thesis in several dimensions. In the work aimed at unifying stream processing over central and near-edge data centers (presented in Paper IV), further research is required to enable SpanEdge to dynamically schedule stream processing applications according to changes in the network conditions and available resources on near-the-edge data centers. In addition to bandwidth, CPU and memory may be scarce resources in edge data centers. Considering the scarcity of CPU and memory in edge data centers creates new challenges and increases the complexity of both defining the optimization problem and its solution. To that end,

studying the existing online schedulers [136, 137] that are proposed for central data centers can be a good start point.

In the research aimed at optimizing windowed aggregation over geo-distributed data streams (presented in Paper III), we plan to investigate on fault tolerance in the windowed aggregation of geo-distributed data streams. Data centers may become inaccessible due to network and server failures. Different type of failures may cause redundant or late updates. Another dimension for future work is to consider workload distributions other than Poisson, having long-term trends, peaks, or other patterns. We would like to investigate other workload prediction methods based on ARIMA or machine learning techniques.

In the research on streaming graph data (presented in Paper V and Paper VI), we propose and evaluate our solutions for the addition of edges and vertices to a dynamic graph. Even though the addition of edges cover many applications, it is an interesting dimension to direct our research towards the removal of edges and vertices. Therefore, we would like to investigate the importance and impact of edge and vertex removal from a dynamic graph on problems such as graph partitioning and graph representation learning in our future work.

Bibliography

[1] K. Danniswara, H. P. Sajjad, A. Al-Shishtawy, and V. Vlassov, "Stream processing in community network clouds," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 800–805.

[2] H. P. Sajjad, F. Rahimian, and V. Vlassov, "Smart partitioning of geo-distributed resources to improve cloud network performance," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 112–118.

[3] H. P. Sajjad, Y. Liu, and V. Vlassov, "Optimizing windowed aggregation over geo-distributed data streams," in *2018 IEEE International Conference on Edge Computing (EDGE)*, 2018, pp. 33–41.

[4] H. Peiro Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processingover central and near-the-edge data centers," in *Edge Computing, 2016 The First IEEE/ACM*. IEEE/ACM, 2016, pp. 123–132.

[5] H. P. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi, "Boosting vertex-cut partitioning for streaming graphs," in *Big Data (BigData Congress), 2016 IEEE International Congress on*. IEEE, 2016, pp. 1–8.

[6] H. P. Sajjad, A. Docherty, and Y. Tyshetskiy, "Efficient representation learning using random walks for dynamic graphs," *arXiv preprint arXiv:1901.01346*, 2019.

[7] K. Hakimzadeh, H. P. Sajjad, and J. Dowling, "Scaling hdfs with a strongly consistent relational model for metadata," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2014, pp. 38–51.

[8] H. P. Sajjad, K. Hakimzadeh, and S. Perera, "Reproducible distributed clusters with mutable containers: To minimize cost and provisioning time," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM, 2017, pp. 18–23.

[9] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile networks and applications*, vol. 19, no. 2, pp. 171–209, 2014.

[10] P. Zikopoulos, C. Eaton *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[11] C. Doulkeridis and K. NØrvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 23, no. 3, pp. 355–380, 2014.

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[13] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[14] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[15] (2018) Data, data everywhere. [Online]. Available: https://www.economist.com/special-report/2010/02/25/data-data-everywhere

[16] P. Zikopoulos, D. Deroos, K. Parasuraman, T. Deutsch, J. Giles, and D. Corrigan, *Harness the power of big data: The IBM big data platform*. McGraw-Hill New York, NY, 2013.

[17] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156. [Online]. Available: http://dx.doi.org/10.1145/2588555.2595641

[18] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 239–250.

[19] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[20] (2019) Apache spark. [Online]. Available: http://spark.apache.org/

[21] (2019) Apache flink. [Online]. Available: https://flink.apache.org/

[22] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[23] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 1, pp. 97–107, 2014.

[24] V. K. Adhikari, S. Jain, and Z.-L. Zhang, "Youtube traffic dynamics and its interplay with a tier-1 isp: an isp perspective," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 431–443.

[25] I. Narayanan, A. Kansal, A. Sivasubramaniam, B. Urgaonkar, and S. Govindan, "Towards a leaner geo-distributed cloud infrastructure." in *HotCloud*, 2014.

[26] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "Wanalytics: Geo-distributed analytics for a data intensive world," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM, 2015, pp. 1087–1092.

[27] G. Hesse and M. Lorenz, "Conceptual survey on data stream processing systems," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2015, pp. 797–802.

[28] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[29] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "Gupt: privacy preserving data analysis made easy," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 349–360.

[30] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

[31] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.

[32] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[33] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 2014, pp. 333–342.

[34] W. Fan, C. Hu, and C. Tian, "Incremental graph computations: Doable and undoable," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 155–169.

[35] Common open research emulator (core). [Online]. Available: http://www.nrl.navy.mil/itd/ncs/products/core

[36] J. Jiménez, R. Baig, P. Escrich, A. M. Khan, F. Freitag, L. Navarro, E. Pietrosemoli, M. Zennaro, A. H. Payberah, and V. Vlassov, "Supporting cloud deployment in the guifi. net community network," in *Global Information Infrastructure Symposium-GIIS 2013*. IEEE, 2013, pp. 1–3.

[37] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, "Distributed event stream processing with non-deterministic finite automata," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, 2009, p. 3.

[38] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1093–1104.

[39] D. Yang, D. Zhang, K.-L. Tan, J. Cao, and F. Le Mouël, "Cands: continuous optimal navigation via distributed stream processing."

[40] Z. Miller, B. Dickinson, W. Deitrick, W. Hu, and A. H. Wang, "Twitter spammer detection using data stream clustering," *Information Sciences*, vol. 260, pp. 64–73, 2014.

[41] A. Hinze, K. Sachs, and A. Buchmann, "Event-based applications and enabling technologies," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM, 2009, p. 1.

[42] A. Hofmann and B. Sick, "Online intrusion alert aggregation with generative data stream modeling," *IEEE transactions on dependable and secure computing*, vol. 8, no. 2, pp. 282–294, 2011.

[43] P. C. Guerra, W. Meira Jr, and C. Cardie, "Sentiment analysis on evolving social streams: How self-report imbalances can help," in *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 2014, pp. 443–452.

[44] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[45] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[46] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://dx.doi.org/10.1145/2723372.2742788

[47] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*. USENIX Association, 2012, pp. 2–2.

[48] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[49] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 623–634.

[50] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online mapreduce computations," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1441–1451, 2014.

[51] Amazon kinesis analytics. [Online]. Available: https://aws.amazon.com/kinesis/analytics/

[52] Azure stream analytics. [Online]. Available: https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction

[53] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *Acm Sigmod Record*, vol. 34, no. 1, pp. 39–44, 2005.

[54] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid, "Incremental evaluation of sliding-window queries over data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 1, pp. 57–72, 2007.

[55] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 702–713, 2015.

[56] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Optimized processing of multiple aggregate continuous queries," in *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 2011, pp. 1515–1524.

[57] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "F1: Accelerating the optimization of aggregate continuous queries," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 2015, pp. 1151–1160.

[58] K. Naidu, R. Rastogi, S. Satkin, and A. Srinivasan, "Memory-constrained aggregate computation over data streams," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 852–863.

[59] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1201–1210.

[60] N. Tatbul and S. Zdonik, "Window-aware load shedding for aggregation queries over data streams," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 799–810.

[61] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Data Engineering, 2004. Proceedings. 20th International Conference on*. IEEE, 2004, pp. 350–361.

[62] B. Mozafari and C. Zaniolo, "Optimal load shedding with aggregates and mining queries," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 76–88.

[63] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 159–170.

[64] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[65] (2019) What is zookeeper? [Online]. Available: https://zookeeper.apache.org/

[66] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, 2009.

[67] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[68] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[69] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: optimizing data parallel jobs in wide-area data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.

[70] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: Wan-aware optimization for analytics queries." in *OSDI*, vol. 16, 2016, pp. 435–450.

[71] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints." in *NSDI*, vol. 7, no. 7.2, 2015, pp. 7–8.

[72] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 421–434.

[73] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds." in *NSDI*, 2017, pp. 629–647.

[74] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 361–373.

[75] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area." in *NSDI*, vol. 14, 2014, pp. 275–288.

[76] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, "Themis: Fairness in federated stream processing under overload," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 541–553.

[77] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing timeliness and cost in geo-distributed streaming analytics," *IEEE Transactions on Cloud Computing*, 2017.

[78] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 49–49.

[79] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 69–80.

[80] Guifi.net. [Online]. Available: http://guifi.net/en/node/38392

[81] Athens wireless metropolitan network (awmn). [Online]. Available: http://www.awmn.net

[82] Funkfeuer. [Online]. Available: https://funkfeuer.at/

[83] R. Baig, J. Dowling, P. Escrich, F. Freitag, R. Meseguer, A. Moll, L. Navarro, E. Pietrosemoli, R. Pueyo, V. Vlassov *et al.*, "Deploying clouds in the guifi community network," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 1020–1025.

[84] M. Selimi, N. Apolónia, F. Olid, F. Freitag, L. Navarro, A. Moll, R. Pueyo, and L. Veiga, "Integration of an assisted p2p live streaming service in community network clouds," in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*. IEEE, 2015, pp. 202–209.

[85] A. McGregor, "Graph stream algorithms: a survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[86] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 1222–1230.

[87] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[88] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[89] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[90] I. Stanton, "Streaming balanced graph partitioning algorithms for random graphs," in *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2014, pp. 1287–1301.

[91] J. Nishimura and J. Ugander, "Restreaming graph partitioning: simple versatile algorithms for advanced balancing," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1106–1114.

[92] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. 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)*, 2012, pp. 17–30.

[93] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Advances in Neural Information Processing Systems*, 2014, pp. 1673–1681.

[94] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: scalable graph etl framework," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 4.

[95] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 2015, pp. 243–252.

[96] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 1.

[97] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[98] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[99] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[100] K. Henderson, B. Gallagher, L. Li, L. Akoglu, T. Eliassi-Rad, H. Tong, and C. Faloutsos, "It's who you know: graph mining using recursive structural features," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 663–671.

[101] J. Tang and H. Liu, "Unsupervised feature selection for linked social media data," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 904–912.

[102] D. Zhang, J. Yin, X. Zhu, and C. Zhang, "Network representation learning: A survey," *IEEE Transactions on Big Data*, 2018.

[103] L. Backstrom and J. Leskovec, "Supervised random walks: predicting and recommending links in social networks," in *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 2011, pp. 635–644.

[104] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, pp. 1–13, 2007.

[105] M. Belkin and P. Niyogi, "Laplacian eigenmaps and spectral techniques for embedding and clustering," in *Advances in neural information processing systems*, 2002, pp. 585–591.

[106] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *IJCAI*, 2015, pp. 2111–2117.

[107] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.

[108] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 1225–1234.

[109] S. Cao, W. Lu, and Q. Xu, "Deep neural networks for learning graph representations." in *AAAI*, 2016, pp. 1145–1152.

[110] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[111] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[112] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[113] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 701–710.

[114] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 855–864.

[115] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, "Harp: hierarchical representation learning for networks," *arXiv preprint arXiv:1706.07845*, 2017.

[116] Qmpsu. [Online]. Available: http://dsg.ac.upc.edu/qmpsu/index.php

[117] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer, 2014, pp. 48–55.

[118] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Tech. Rep., 2009.

[119] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[120] M. Bastian, S. Heymann, M. Jacomy *et al.*, "Gephi: an open source software for exploring and manipulating networks." *Icwsm*, vol. 8, no. 2009, pp. 361–362, 2009.

[121] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing grouped aggregation in geo-distributed streaming analytics," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 133–144.

[122] Big data benchmark dataset. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/v1/

[123] (2019) Google compute engine pricing. [Online]. Available: https://cloud.google.com/compute/pricing

[124] Yahoo's storm performance test. [Online]. Available: https://github.com/yahoo/storm-perf-test

[125] F. Sheng, Q. Cao, H. Cai, J. Yao, and C. Xie, "Grapu: Accelerate streaming graph analysis through preprocessing buffered updates," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 301–312.

[126] S. M. S. Khamoushi, "Scalable streaming graph partitioning," Master's thesis, School of Information and Communication Technology (ICT), 2017, supervisor: Hooman Peiro Sajjad.

[127] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 177–187.

[128] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference and International Workshop Present Day Trends of Innovations*, 2012, pp. 1–6.

[129] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 44–54.

[130] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[131] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[132] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 135–144.

[133] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[134] L. Du, Y. Wang, G. Song, Z. Lu, and J. Wang, "Dynamic network embedding: An extended approach for skip-gram based network embedding." in *IJCAI*, 2018, pp. 2086–2092.

[135] J. Li, H. Dani, X. Hu, J. Tang, Y. Chang, and H. Liu, "Attributed network embedding for learning in a dynamic environment," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 2017, pp. 387–396.

[136] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 207–218.

[137] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 535–544.