Scalable Graph Partitioning for Distributed Graph Processing

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Christian Mayer

aus Breisach

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichterin:	Prof. Dr. rer. nat. Melanie Herschel
Tag der mündlichen Prüfung:	24.07.2019

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2019

Acknowledgements

First and foremost, I would like to express my deep gratitude to my mentor, Prof. Dr. Kurt Rothermel. His trust, guidance, and role model formed the basis of my progress in research and life. Most importantly, he showed me how to think critically and analyze situations rationally —serving as a role model of fairness, integrity, and work ethics from which I will benefit all my life. For his constant support, guidance, and mentoring I will be forever grateful.

I would like to give my thanks to Prof. Dr. Melanie Herschel for being a part of my Ph.D. committee and reviewing my thesis.

Over the years, I have had three postdoctoral supervisors who all shared their great wisdom with me and gave me guidance and support. I would like to thank my supervisor and mentor Dr. Muhammad Adnan Tariq for trusting me and for being a beacon in a confusing ocean of possible research directions—his contribution to my research career was outstanding. Moreover, I would like to thank my supervisor Dr. Ruben Mayer for his years of friendship, guidance, and inspiration—he truly made the ship go faster! Finally, from the bottom of my heart, I would like to thank my supervisor Dr. Sukanya Bhowmik who was both a friend and a role model for positive thinking and focus—she showed me how to enjoy the journey and gave me a glimpse into the true value of collaboration.

I would like to thank my esteemed colleagues for creating a work environment full of friendship, fun, and inspiration. I am truly honored about having been given the opportunity of working together with such inspiring friends. Zohaib Riaz was the second half of my brain in our “thinking-aloud sessions”. Thomas Bach and David Schäfer who took me in and made my time so much more enjoyable. Johannes Kässinger showed me the value of true comradeship. Henriette Röger inspired me to produce and perform on a higher level. Ben Carabelli made me always feel heard—he truly is a great scientist. Thomas Kohler, Otto Bibartiu, and Christoph Dibak always made me smile and enjoy the time in the department. My students Larissa Laich, Jonas Grunert, Heiko Geppert, Lukas Rieger, and Lukas Epple contributed greatly to this research and I am very grateful for their creativity and “hands-on” mentality. Special thanks go to Eva Strähle who is an inspiring and positive woman—never tired of helping one out. I consider it a great privilege to have worked together with such admirable people.

My deepest gratitude goes to my wife Anna Altimira—she is the love of my life. All I have achieved in life is as much her accomplishment as it is mine. She helped me raise two wonderful kids, Amalie and Gabriel, who were my main source of motivation and recreation. I cannot put in words how grateful I am to have such a supporting family.

I would like to thank my family and friends for supporting me during this process. My parents Vera and Ulrich Mayer, my sister Kerstin Mayer, my brother Sven Mayer, and my in-laws always supported me and encouraged me to pursue my goals. I cannot give enough thanks to all my friends who play an integral part in my life—especially my very best friend and intellectual sparring partner, Nathan Prins, as well as Tobias Miltenberger, without whom I probably wouldn't have started my newest venture.

Contents

Abstract **19**

Deutsche Zusammenfassung **21**

1 Introduction **23**

1.1 Research Statement 24

1.1.1 Traffic-aware Graph Partitioning and Processing 25

1.1.2 Latency Trade-off between Graph Partitioning and Processing 25

1.1.3 Query-centric Graph Partitioning and Processing 25

1.1.4 Neighborhood-centric Partitioning of Skewed Hypergraphs 26

1.2 Contributions 26

1.3 Structure of the Thesis 28

2 Background **29**

2.1 Distributed Graph Processing 29

2.1.1 System Model 29

2.1.2 Data Graph 30

2.1.3 Synchronization Model 31

2.1.4 Programming Model 31

2.2 Graph Partitioning 33

2.2.1 Edge-cut Partitioning 33

2.2.2 Vertex-cut Partitioning 34

2.2.3 Comparison Vertex-cut and Edge-cut Partitioning 34

3 Heterogeneous Partitioning for Distributed Graph Processing **37**

3.1 Preliminaries and Problem Formulation 39

3.1.1 Preliminaries 39

3.1.2 Network- and Traffic-aware Dynamic Vertex-cut 42

3.2 Partitioning Algorithms 45

3.2.1 H-load: Initial Partitioning 46

3.2.2 H-adapt: Distributed Migration of Edges 48

3.3 Graph Algorithms . . . 55
3.3.1 Subgraph Isomorphism . . . 55
3.3.2 Cellular Automaton . . . 59
3.4 Evaluations . . . 61
3.4.1 Summary of Evaluation Results . . . 68
3.5 Related Work . . . 69
3.6 Chapter Summary . . . 70

4 Adaptive Window-based Streaming Partitioning 71
4.1 Problem Statement and Analysis . . . 73
4.1.1 The Vertex-cut Graph Partitioning Problem . . . 74
4.1.2 Streaming Partitioning . . . 74
4.1.3 Window-based Streaming Partitioning . . . 77
4.2 ADWISE . . . 78
4.2.1 Adaptive Window Algorithm . . . 80
4.2.2 Lazy Window Traversal . . . 81
4.2.3 Scoring Window Edges . . . 82
4.2.4 Spotlight Partitioning . . . 85
4.3 Evaluation . . . 86
4.3.1 Efficacy of ADWISE to Minimize Total Graph Latency . . . 87
4.3.2 Spotlight . . . 93
4.3.3 Summary of Evaluation Results . . . 93
4.4 Related Work . . . 94
4.5 Chapter Summary . . . 95

5 Query-aware Multi-query Graph Processing 97
5.1 Research Gap and Contributions . . . 99
5.2 Problem Description . . . 100
5.3 Q-Graph System . . . 103
5.3.1 System Overview . . . 103
5.3.2 Q-cut: Centralized Query-aware Partitioning . . . 104
5.3.3 Hybrid Barrier Synchronization for Multi-Query Graph Processing . . . 109
5.3.4 Adapting to Dynamic Query Workload . . . 111
5.4 Evaluations . . . 112
5.4.1 Experimental Setup . . . 112
5.4.2 Adaptive Q-cut Partitioning: . . . 114
5.4.3 Scalability and Hybrid Barrier . . . 117
5.4.4 Summary of Evaluation Results . . . 118
5.5 Related Work . . . 119
5.6 Chapter Summary . . . 119

6 Massive Hypergraph Partitioning with Neighborhood Expansion 121
6.1 Research Gap and Contributions 121
6.2 Problem Formulation 122
6.3 Hypergraph Partitioning with Neighborhood Expansion 124
6.3.1 Neighborhood Expansion Idea 125
6.3.2 HYPE Algorithm 126
6.3.3 Balancing Considerations 132
6.3.4 HYPE Pseudocode 133
6.3.5 Complexity Analysis 134
6.4 Evaluations 136
6.4.1 Performance Evaluations 138
6.4.2 Discussion of the Results 140
6.5 Related Work 141
6.6 Chapter Summary 142

7 Summary and Future Work 143
7.1 Thesis Summary 143
7.2 Conclusions 145
7.3 Future Work 146

Bibliography 147

List of Figures

2.1 Scaling mechanisms for distributed graph processing (one CPU core per worker). 30

2.2 Example PageRank algorithm. 32

2.3 Vertex-cut and edge-cut partitioning. 33

2.4 This star graph with hub vertex u can be cut through either one vertex or four edges. 35

3.1 (a)-(c) Distribution functions of vertex traffic for PageRank, subgraph isomorphism, and cellular automaton. (d) Latency between worker machines intra-Availability Zone (AZ), inter-AZ and intra-region, and inter-region. 40

3.2 Message Flow of a single GAS iteration for vertex v. 42

3.3 (a) Vertex-cut minimizing replication degree. (b) Network- and traffic-aware vertex-cut minimizing communication costs. 43

3.4 Approach Overview H-load. 46

3.5 Example: bag-of-edges migration to reduce inter-partition traffic. 49

3.6 Lost update problem for parallel edge migration. 54

3.7 (a) Subgraph isomorphism example – perspective of vertex v_2. We show the path of a single message. (b) The algorithm considers only messages that travel *within the matching subgraph*. 56

3.8 Agent-based social simulations via cellular automaton. 59

3.9 (a) Traffic- and (b) network-awareness reduce communication costs. 62

3.10 (a) H-adapt reduces communication costs with low overhead. (b) Pre-partitioning with H-load reduces communication costs. 62

3.11 (a)-(b) H-adapt reduces network traffic. 63

3.12 (a) H-adapt reduces replication degree. (b) GraphH reduces total workload. (c)-(d) H-adapt reduces latency. 63

3.13 Heterogeneity in single data center, scale-up scenario. 65

3.14 (a)-(b) Prediction methods influence migration efficiency. 67

3.15 (a)-(b) Adaptive-α outperforms other prediction methods. 67

4.1 Research gap – adaptive window-based streaming vertex-cut partitioning. Partitioning strategies: Hash [34], Grid [50], DBH [140], Greedy [34], HDRF [101], NE [150], H-move [87], and Ja-Be-Ja-VC [106] 72
4.2 Streaming partitioning model. 77
4.3 Approach overview ADWISE. 79
4.4 Degree-aware vertex-cut partitioning. 83
4.5 Clustering Score Example. 85
4.6 Spotlight partitioning reduces spread of partitioners to reduce the impact of balancing considerations on the partitioning decisions. 86
4.7 Trade-off graph partitioning latency against processing latency. 88
4.8 Replication degree for different partitioning strategies and settings. For all presented results, the partitions are balanced, i.e., $\frac{maxsize-minsize}{maxsize} < 0.05$ (see Section 4.2.3). 89
4.9 Efficacy of spotlight optimization on Brain. 93

5.1 Query-agnostic partitioning optimizes edge-cut and query-aware partitioning optimizes query-cut.[1] 103
5.2 System Architecture. 104
5.3 The Q-cut algorithm operates on a high-level query representation to improve low-level graph partitioning. 106
5.4 Perturbation example. 108
5.5 The hybrid barrier synchronization integrates: A) limited and B) local query barriers, and C) global barriers. 110
5.6 Hybrid Barrier Protocol for a single query q_1. The multi-query approach uses this protocol for each query independently. 111
5.7 (a)-(b) Adaptive query-aware partitioning reduces query latency over time. . . 115
5.8 (a)-(b) Q-cut reduces query latency for different partitioning strategies for SSSP and (c) for POI compared to static partitioning Hash and Domain. (d) The query latency decreases with better partitioning and hybrid barrier synchronization. . 116
5.9 (a) Workload Balancing. (b) Percentage of local query executions. 116
5.10 Perturbation overcomes local minima (SSSP on BW). 117
5.11 Q-Graph scalability. 118

6.1 A small extract of the global Reddit graph. 123
6.2 High-level idea of neighborhood expansion. 126
6.3 Limiting the fringe size s to a small value (e.g. $s = 10$) keeps partitioning quality intact while reducing runtime significantly (StackOverflow hypergraph). 127
6.4 The external neighbors metric determines which vertex to move into the fringe. 129
6.5 Limiting the number of fringe candidate vertices r to $r = 2$ leads to the best partitioning quality (StackOverflow hypergraph). 131

6.6 The caching optimization for external neighbors score computation keeps partitioning quality intact while reducing runtime by up to 20% on the Stackoverflow hypergraph. 132
6.7 Evaluations on the Github hypergraph (lower is better). 135
6.8 Evaluations on the StackOverflow hypergraph (lower is better). 137
6.9 Evaluations on the Reddit hypergraph (lower is better). 139
6.10 Evaluations on the Reddit-L hypergraph (lower is better). 140

List of Tables

3.1 Notation overview. 44
3.2 Heterogeneous network link costs for AWS (Jan 2018). 45
3.3 Real-world graphs for evaluations. 61

4.1 Notation overview. 75
4.2 Real-world graphs for evaluations. 87

5.1 Research gap: multi-query graph processing with adaptive partitioning maximizing query locality. 97
5.2 Notation overview. 101
5.3 Q-Graph API. 105

6.1 Notation overview. 124
6.2 Real-world hypergraphs used in evaluations. 130

List of Algorithms

1 Migration algorithm on worker m. 48
2 Determining the bag-of-edges to exchange. 50
3 Implicit locking scheme. Does worker m have the lock for vertex v? 55
4 Subgraph isomorphism algorithm. 58
5 Cellular automaton algorithm of vertex u. 60
6 Window-based streaming vertex-cut algorithm. 81
7 Iterated local search algorithm for Q-cut partitioning. 106
8 Local search heuristic to find local minimum. 107
9 HYPE algorithm for hypergraph $G = (V, E)$. 133
10 The function `upd8_fringe()` updates the fringe F_i with vertices from the vertex universe V'. 133
11 The function `upd8_core()` updates the core C_i with vertices from the fringe F_i. 134

List of Abbreviations

ACID	Atomicity Consistency Isolation Durability
ADWISE	Adative Window-based Streaming Edge Partitioning
API	Application Programming Interface
AWS	Amazon Web Services
AZ	Availability Zone
BSP	Bulk Synchronous Parallel
BW	Baden-Wuerttemberg
CA	Cellular Automaton
CDF	Cumulative Distribution Function
CGA	Concurrent Graph Query Analytics
CPU	Central Processing Unit
DBH	Degree-Based Hashing
EA	Exponential Averaging
EC2	Amazon Elastic Compute Cloud
GAS	Gather Apply Scatter
GB	Gigabyte
GPS	Global Positioning System, Graph Processing System
GPU	Graphics Processing Unit
GY	Germany
HDRF	High-Degree are Replicated First
ILS	Iterated Local Search
KB	Kilobyte
LDG	Linear Deterministic Greedy
MA	Moving Average
MB	Megabyte
NP	Nondeterministic Polynomial Time
NY	New York
PG	PowerGraph
POI	Point of Interest
PR	PageRank
RAM	Random-Access Memory
SI	Subgraph Isomorphism
SSSP	Single-Source Shortest Path
TCP	Transmission Control Protocol
VLSI	Very Large-Scale Integration

Abstract

Distributed graph processing systems such as Pregel, PowerGraph, or GraphX have gained popularity due to their superior performance of data analytics on graph-structured data such as social networks, web document graphs, and biological networks. These systems scale out graph processing by dividing the graph into k partitions that are processed in parallel by k worker machines. The graph partitioning problem is NP-hard. Yet, finding good solutions for massive graphs is of paramount importance for distributed graph processing systems because it reduces communication overhead and latency of distributed graph processing. A multitude of graph partitioning heuristics emerged in recent years, fueled by the challenge of partitioning large graphs quickly.

The goal of this thesis is to tailor graph partitioning to the specifics of distributed graph processing and show that this leads to reduced graph processing latency and communication overhead compared to state-of-the-art partitioning. In particular, we address the following four research questions. (I) Recent partitioning algorithms unrealistically assume a uniform and constant amount of data exchanged between graph vertices (i.e., uniform *vertex traffic*) and homogeneous network costs between workers hosting the graph partitions. The first research question is: how to consider dynamically changing and heterogeneous graph workload for graph partitioning? (II) Existing graph partitioning algorithms focus on minimal partitioning latency at the cost of reduced partitioning quality. However, we argue that the mere minimization of partitioning latency is not the optimal design choice in terms of minimizing total latency, i.e., the sum of partitioning and graph processing latency. The second research question is: how much latency should we invest into graph partitioning when considering that we often have to pay higher partitioning latency in order to achieve better partitioning quality (and therefore reduced graph processing latency)? (III) Popular *user-centric* graph applications such as route planning and personalized social network analysis have initiated a shift of paradigms in modern graph processing systems towards *multi-query analysis*, i.e., processing multiple graph queries in parallel on a shared data graph. These applications generate a dynamic number of *localized* queries around query hotspots such as popular urban areas. However, the employed methods for graph partitioning and synchronization management disregard query locality and dynamism which leads to high query latency. The third question is: how to dynamically adapt the graph partitioning when multiple localized graph queries run in parallel on a shared graph structure?

(IV) Graphs are special cases of hypergraphs where each edge does not necessarily connect exactly two but an arbitrary number of vertices. Like graphs, they need to be partitioned as a pre-processing step for distributed *hypergraph* processing systems. Real-world hypergraphs have billions of vertices and a skewed degree distribution. However, no existing hypergraph partitioner tailors partitioning to the important subset of hypergraphs that are very large-scale and have a skewed degree distribution. Regarding this, the fourth research question is: how to *partition* these large-scale, skewed hypergraphs in an efficient way such that neighboring vertices tend to reside on the same partition?

We answer these research questions by providing the following four contributions. (I) We developed the graph processing system GrapH that considers both, diverse vertex traffic and heterogeneous network costs. The main idea is to *avoid frequent communication over expensive network links* using an adaptive edge migration strategy. (II) We developed a static partitioning algorithm ADWISE that allows to control the trade-off between partitioning latency and graph processing latency. Besides providing evidence for efficiency and effectiveness of our approach, we also show that state-of-the-art partitioning approaches invest too little latency into graph partitioning. By investing more latency into partitioning using ADWISE, total latency of partitioning and processing reduces significantly. (III) We developed a distributed graph system QGraph for multi-query graph analysis that allows multiple localized graph queries to run in parallel on a shared graph structure. Our novel query-centric dynamic partitioning approach yields significant speedup as it repartitions the graph such that queries can be executed in a localized manner. This avoids expensive communication overhead while still providing good workload balancing. (IV) We developed a novel hypergraph partitioning algorithm, called HYPE, that partitions the hypergraph by using the idea of neighborhood expansion. HYPE grows k partitions separately—expanding one vertex at a time over the neighborhood relation of the hypergraph. We show that HYPE leads to fast and effective partitioning performance compared to state-of-the-art hypergraph partitioning tools and partitions billion-scale hypergraphs on a single thread.

The algorithms and approaches presented in this thesis tailor graph partitioning towards the specifics of distributed graph processing with respect to (I) dynamic and heterogeneous traffic patterns and network costs, (II) the integrated latency of partitioning plus graph processing, and (III) the graph query workload for partitioning and synchronization. On top of that, (IV) we propose an efficient hypergraph partitioner which is specifically tailored to real-world hypergraphs with skewed degree distributions.

Deutsche Zusammenfassung

Verteilte Systeme zur Analyse von Graphen, wie Pregel, PowerGraph, oder GraphX, verbreiteten sich rapide in den letzten Jahren aufgrund ihrer überlegenen Performanz auf großen Graphen wie sozialen Netzwerken, Webgraphen, und biologischen Netzwerken. Diese Systeme skalieren die Graphanalyse indem sie einen Graphen in k Partitionen unterteilen und diese parallel auf k Maschinen verarbeiten. Das Problem der optimalen Partitionierung von Graphen ist NP-hart. Gleichzeitig werden gute Lösungen benötigt um den Kommunikationsmehraufwand und die Latenz der verteilten Graphanalyse zu reduzieren. In den letzten Jahren wurde eine große Anzahl von Heuristiken zur Partitionierung von Graphen entwickelt. Diese Heuristiken können große Graphen schnell partitionieren, und erreichen gleichzeitig eine moderate Partitionierungsqualität.

Das Ziel dieser Dissertation ist die Entwicklung von Algorithmen zur Partitionierung von Graphen, die speziell auf die verteilte Graphanalyse optimiert sind. Diese spezifischen Algorithmen führen zu schnellerer Latenz der verteilten Graphanalyse und reduzieren den Kommunikationsaufwand während der verteilten Graphanalyse im Vergleich zu modernen Partitionierungsalgorithmen. Um dieses Ziel zu erreichen, beantworten wir die folgenden vier Forschungsfragen. (I) Aktuelle Partitionierungsalgorithmen basieren auf der unrealistischen Annahme, dass der Datenverkehr zwischen Knoten, sowie die Netzwerkkosten zwischen Maschinen, homogen und konstant ist. Wie können wir diese sich dynamisch ändernde und heterogene Arbeitslast bei der Partitionierung von Graphen berücksichtigen? (II) Moderne Partitionierungsalgorithmen streben minimale Latenz an, insbesondere auf Kosten der Qualität der Partitionierung. In dieser Arbeit zeigen wir, dass die einfache Minimierung der Partitionierungslatenz nicht die optimale Entwurfsentscheidung darstellt. Stattdessen sollte die Minimierung der Gesamtlatenz das Ziel sein, also der Summe aus der Latenz der Partitionierung und der Graphanalyse. Wieviel Latenz sollten wir in die Partitionierung investieren unter Berücksichtigung, dass eine bessere Partitionierungsqualität (und damit reduzierte Latenz der Graphanalyse) oftmals eine höhere Partitionierungslatenz erfordert? (III) Moderne Anwendungen im Bereich der Graphanalyse rücken den Endnutzer ins Zentrum, beispielsweise bei der Routenplanung und der personalisierten Analyse von sozialen Netzwerken. Dieser Paradigmenwechsel verlangt von Graphsystemen die Unterstützung von gleichzeitiger Verarbeitung von mehreren Anfragen auf einem gemeinsamen Graphen. Diese Anfragen sind *lokalisiert*, beispielsweise Routenanfragen in der

Nähe eines populären Stadtzentrums, und treten in dynamischer Häufigkeit auf. Die existierenden Arbeiten zur Partitionierung und Synchronisation von Graphen ignorieren die Lokalität und dynamische Häufigkeit dieser Anfragen. Dies führt zu einer hohen durchschnittlichen Anfragelatenz. Wie können wir die Graphanalyse dynamisch an die sich ändernden Anfragelokationen anpassen und gleichzeitig mehrere Anfragen parallel auf dem gemeinsamen Graphen ausführen? (IV) Graphen sind spezielle Fälle von Hypergraphen, wobei jede Kante nicht notwendigerweise exakt zwei, sondern eine beliebige Anzahl von Knoten verbindet. Wie Graphen müssen Hypergraphen partitioniert werden als Vorverarbeitungsschritt einer verteilten *Hypergraphanalyse*. In der Praxis häufig vorkommende Hypergraphen haben Milliarden von Knoten und besitzen eine sogenannte "asymmetrische Wahrscheinlichkeitsverteilung" der Knoten- und Kantengrade. Allerdings gibt es derzeit keinen Hypergraphpartitionierer, der sich speziell auf große Hypergraphen fokussiert, die eine assymetrische Wahrscheinlichkeitsverteilung der Knoten- und Kantengrade aufweisen. Deshalb lautet die vierte Forschungsfrage: Wie können wir diese großen, asymmetrischen Hypergraphen effizient partitionieren, sodass Nachbarknoten mit möglichst hoher Wahrscheinlichkeit auf derselben Partition liegen?
Wir beantworten diese vier Forschungsfragen folgendermaßen. (I) Unser System zur Graphanalyse GrapH berücksichtigt den heterogenen Datenverkehr zwischen Knoten, sowie die heterogenen Netzwerkkosten zwischen Maschinen. Die Grundidee ist die systematische Vermeidung der Kommunikation über teure Netzwerkverbindungen durch eine adaptive Kantenmigrationsstrategie. (II) Der statische Partitionierungsalgorithmus ADWISE ermöglicht eine Kosten-Nutzen-Abschätzung zwischen Latenz der Partitionierung und der Graphanalyse. Wir demonstrieren die Effizienz und Effektivität unseres Ansatzes und gelangen zu der Erkenntnis, dass moderne Partitionierungsansätze zu wenig Latenz in die Partitionierung von Graphen investieren. Indem wir mithilfe von ADWISE mehr Latenz in die Partitionierung investieren, kann die Gesamtlatenz der Partitionierung und Graphanalyse signifikant reduziert werden. (III) Wir entwickelten ein verteiltes System zur Analyse von mehreren gleichzeitigen Anfragen auf einem gemeinsamen Graphen, genannt QGraph. Die Idee, die Lokalität der Anfragen in den Fokus der Partitionierung zu rücken, führt zu einer signifikanten Reduzierung der durchschnittlichen Anfragelatenz. Der Grund hierfür ist, dass unser Algorithmus den Graphen dynamisch partitioniert, sodass Anfragen zu jedem Zeitpunkt möglichst lokalisiert ausgeführt werden. Dies vermeidet teuren Kommunikationsaufwand und ermöglicht gleichzeitig eine balancierte Arbeitslast der Maschinen. (IV) Wir entwickelten einen neuen Algorithmus zur Hypergraphpartitionierung, genannt HYPE, der auf der Idee der Nachbarschaftsexpansion basiert. HYPE vergrößert sukzessive k Partitionen indem er Knoten über die Nachbarschaftsbeziehung expandiert. Wir zeigen, dass HYPE zu schnellerer und effektiverer Partitionsperformanz führt im Vergleich zu gängigen Hypergraphpartitionierern—auf nur einem einzigen Thread.

1

Introduction

Graphs are fundamental data structures in computer science. Wide-spread digitalization generates massive amounts of interconnected, graph-structured data in diverse areas. These include information retrieval from semantic web graphs [114], product recommendation based on product rating graphs [126], communities detection in social networks [97], centrality computation in web document graphs [8], probabilistic inference on graphical models [146], social movement pattern simulation on urban cellular networks [87], shortest path calculation on road networks [149], cellular clustering in biological networks [2], disease propagation analysis in socio-temporal networks [10], and images retrieval from image similarity graphs [52]. This non-exhaustive list shows the variety and importance of emerging graph processing applications. Graphs have been, are, and will stay ubiquitous in the field of computer science.

In the last decade, two trends have had a sustainable impact on graph processing: (i) growing data volume and (ii) increasing processing complexity. (i) Web graphs link trillions of documents [22], social networks connect hundreds of billions of users [21], the semantic web consists of trillions of RDF triples describing relations between entities [148], recommendation graphs connect millions of people to billions of products, movies, or songs [16], and deep neural networks comprise of billions of highly connected artificial neurons [25]. (ii) Many important graph algorithms are iterative, complex, and long-running. Examples are PageRank [96], subgraph isomorphism [79], k-clique [139], belief propagation [53], graph coloring [34], single source shortest path [80], graph clustering [117], and collaborative filtering [72]. As data tends to grow faster than processing power [92], parallelizing computation of complex graph algorithms on large graphs is crucial for many real-world applications [90]. This has led to the advent of specialized distributed graph processing systems such as, in chronological order, Pregel [80], GraphLab [77], PowerGraph [34], GPS [115], GraphX [35], Giraph [21], PowerLyra [18], GraM [138], and GrapH[1] [87]. These systems adopt a user-friendly programming

[1]See Chapter 3

paradigm, i.e., application programmers, *think like a vertex* [80] by defining vertex functions that workers execute in parallel on the vertices of the distributed graph. Vertices iteratively update their local state at runtime based on the state of neighboring vertices.

To parallelize graph execution on k workers, these systems divide the graph into k partitions such that the number of vertices and/or edges are balanced and maximally localized, i.e., neighboring vertices and edges preferably reside on the same partition. For instance, if neighboring vertices reside on the same partition, exchanging data between two vertices is inexpensive. Otherwise, if vertices reside on different partitions, data has to be sent via TCP or other means of inter-process communication. This problem is denoted as *balanced k-way graph partitioning*. The graph partitioning problem is computationally hard, even simplified variants are NP-complete [32]. But graph partitioning is also an important problem: Developing fast and effective algorithms provides a large value for industry and academia in terms of reduced latency, communication overhead, and monetary costs. The combination of hardness and practical importance has driven research in the area of graph partitioning for decades — with constantly changing scale, performance, and assumptions.

In this thesis, we motivate, develop, and analyze novel graph partitioning algorithms tailored to distributed graph processing. Graph partitioning plays a key role in distributed graph processing regarding several important performance metrics such as scalability, latency, CPU utilization, and communication overhead [133]. Moreover, efficient graph partitioning algorithms are crucial for various other domains such as image segmentation [121], identifying communities in biological protein interaction networks [112] and in social networks [102], parallel processing of specific graph algorithms using message passing [45], speeding up Dijkstra's algorithm in road networks [93], and very large-scale integration (VLSI) [58]. The rest of this chapter presents research challenges, gaps, and hypotheses, followed by our contributions to the field of graph partitioning for distributed graph processing.

1.1 Research Statement

This thesis focuses on four research gaps and problem domains in the area of graph partitioning (and the more generalized *hypergraph partitioning*). The main goal is to design dynamic and static partitioning algorithms that are tailored to the unique characteristics of distributed graph and hypergraph processing on real-world data sets. In this thesis, we show that a careful consideration of these specific properties reduces latency and communication overhead of partitioning and graph processing compared to state-of-the-art benchmarks and generalized "out-of-the-box" graph partitioning algorithms.

1.1.1 Traffic-aware Graph Partitioning and Processing

A wide-spread belief in the field of graph partitioning for distributed graph processing is the following: *The higher the number of neighboring vertices or edges residing on the same partition, the lower the communication costs during graph processing* [34, 35]. But this holds only under two assumptions: *vertex traffic homogeneity*, i.e., processing each vertex involves the same amount of communication overhead, and *network costs homogeneity*, i.e., the underlying network links between each pair of workers have the same usage costs [34, 106, 115]. We show in Chapter 3 that both assumptions are not generally true. Instead, vertex traffic and network costs are highly heterogeneous for many real-world applications.

This leads us to the research question: How can we tailor graph partitioning to distributed graph processing by considering heterogeneous vertex traffic and network costs? Our hypothesis is that this heterogeneity-aware graph partitioning leads to reduced communication costs and graph processing latency.

1.1.2 Latency Trade-off between Graph Partitioning and Processing

Suboptimal graph partitioning causes higher graph processing latency due to the increased synchronization and communication overhead between the worker machines [64, 78, 133]. In other words, *finding good solutions to the partitioning problem speeds up graph processing*. But finding better partitioning solutions is slow because of the NP-hardness of the partitioning problem. There is a trade-off between the *partitioning latency* and the *graph processing latency*. Existing graph systems are positioned on one extreme of the trade-off: minimal partitioning latency (which is close to the *minimum* partitioning latency of random partitioning [87, 123]) and, thus, only moderate partitioning quality.

A fundamental research question is whether it is always optimal to invest minimal partitioning latency as done by the established systems. Our hypothesis is that for complex and long-running graph algorithms that run on large graphs, investing more than minimal time into graph partitioning leads to reduced total latency of graph partitioning *and* graph processing.

1.1.3 Query-centric Graph Partitioning and Processing

Novel graph applications have given rise to a shift of paradigms towards interactive (online) graph queries on a shared graph structure [27, 143]. We identified three challenges for those applications. (i) *Locality:* how to ensure locality of query execution? (ii) *Multi-query:* how to manage the execution of multiple queries in parallel on a shared graph? (iii) *Adaptivity:* how to adapt to dynamic query hotspots and query workloads? None of the existing graph systems account for all three challenges.

The research question here is whether taking into account the dynamic workload of graph queries for graph partitioning (*query-awareness*) reduces total graph processing latency. Our hypothesis is that this is in fact the case when executing multiple localized queries on the same graph.

1.1.4 Neighborhood-centric Partitioning of Skewed Hypergraphs

Graphs are special cases of hypergraphs. In a hypergraph, each edge does not necessarily connect exactly two but an arbitrary number of vertices. Real-world hypergraphs have billions of vertices (e.g. the group-based WhatsApp social network [83]) and a skewed degree distribution (i.e., a small percentage of vertices have extremely high degree while most vertices have a small to medium degree). We show in Chapter 6 that both edge and vertex degrees resemble a power-law distribution.

Regarding this, an interesting research question is how to *partition* these large-scale, skewed hypergraphs in an efficient way such that neighboring vertices tend to reside on the same partition? There is a significant research gap because no existing hypergraph partitioner tailors partitioning to the important subset of hypergraphs that are very large-scale and have a skewed degree distribution—considering the practical need to optimize for partitioning speed. Our hypothesis is that a simple neighborhood-centric partitioning algorithm—with a bias towards optimal partitioning of hypervertices with low degree—generates competitive partitioning quality with reduced partitioning latency compared to state-of-the-art approaches, even for billion-scale hypergraphs.

1.2 Contributions

This thesis focuses on improving scalability, communication overhead, and latency of distributed graph computation by solving the graph partitioning problem for distributed graph processing. The following contributions are based on work performed and published as part of the PhD thesis [86], [87], [89], [85], [84], [82], [88], [83].

1. To address the research statement in Section 1.1.1, we developed GrapH, a distributed graph processing system for in-memory data analytics on graph-structured data[2]. GrapH is aware of both, heterogeneous dynamic vertex traffic and heterogeneous network costs. Considering this information, it adaptively partitions the graph at runtime to minimize overall communication costs by systematically *avoiding frequent communication over expensive network links*.

[2] https://github.com/GrapH2/GrapH2.0

In particular, we propose a fast linear-runtime partitioning algorithm and a fully distributed edge migration algorithm for runtime refinement. The distributed edge migration algorithm incorporates a method for online vertex traffic prediction that treats *each vertex as an independent learner* and thereby is able to predict diverse and dynamic vertex traffic patterns. Moreover, we propose two novel distributed algorithms plus implementation in the vertex-centric (*think like a vertex*) API solving two important graph problems: subgraph isomorphism to find arbitrary subgraphs in the graph, and agent-based cellular automaton to simulate physical phenomena such as mobility of citizens. These contributions have been published in [86, 87, 89]. The author of this thesis contributed around 70%, 45%, and 70% of the scientific content, respectively.

2. To address the research statement in Section 1.1.2, we developed ADWISE, a novel *window-based* streaming partitioning approach[3]. Our result regarding the trade-off between partitioning latency and graph processing latency is the following. For complex graph problems, we can reduce runtime significantly when investing approximately three times the partitioning latency of state-of-the-art partitioning algorithms that optimize mainly for partitioning latency.

   In particular, we propose an algorithm that uses an *edge window* rather than only a single edge to determine the best partitioning decisions. We employ methods to *automatically adapt* the window size at runtime in order to control the partitioning latency via a latency preference parameter. Our algorithms considers multiple objectives – including diversity and skewness of the graph edges – to quantify partitioning decisions pertaining to the edges in the window. We employ multiple performance optimizations such as focusing computation on a subset of most promising window edges to avoid redundant computations. Finally, we propose an optimization for parallel graph partitioning on multiple ADWISE instances where partitioning instances work on disjoint sets of partitions. This tremendously improves partitioning quality and can be applied on top of any existing streaming graph partitioning algorithm. These contributions have been published in [85]. The author of this thesis contributed around 70% of the scientific content.

3. To address the research statement in Section 1.1.3, we developed the open-source system QGraph for `CGA` applications[4]. We use a centralized controller to maintain global knowledge about the queries running on each worker to perform *high-level query-aware partitioning*. We show that query-awareness of partitioning and synchronization speeds up `CGA` applications – as a result of improved query locality and workload balancing compared to query-agnostic and static edge-cut partitioning algorithms.

   In particular, the query-aware partitioning algorithm partitions the graph based on a history of queries reaching high query locality. Moreover, we introduce a novel hybrid

[3] `https://github.com/GraphPartitioning/WISE`
[4] `https://gitlab.com/qgraph/GRADES2018`

synchronization model to account for both local and global graph queries. Finally, our partitioning adaptation algorithm optimizes the graph partitioning at runtime to dynamic query hotspots using the centralized global query knowledge. These contributions have been published in [84]. The author of this thesis contributed around 70% of the scientific content.

4. To address the research statement in Section 1.1.4, we developed a novel hypergraph partitioning algorithm, called HYPE, and made the source code publicly available[5]. The algorithm partitions the hypergraph by using the idea of neighborhood expansion. It grows k partitions separately—expanding one vertex at a time over the neighborhood relation of the hypergraph. We show that HYPE leads to fast and effective partitioning performance compared to state-of-the-art hypergraph partitioning tools and even scales to billion-scale hypergraphs on a single thread.

   In particular, we introduce methods to reduce the algorithmic complexity of neighborhood expansion on hypergraphs. We observe that for real-world hypergraphs, the degrees of both vertices and hyperedges resemble a power-law degree distribution. These hypergraphs have dense local communities connected by a small number of high-degree vertices. Based on this observation, we develop the idea of reinforcing the algorithm to expand each partition towards the small and dense local communities and punishing expansions towards high-degree vertices and hyperedges. These contributions have been published in [83]. The author of this thesis contributed around 50% of the scientific content.

A number of student theses contributed parts of system implementations and evaluations to this thesis [9, 29, 33, 36, 71, 74, 109, 118, 137].

1.3 Structure of the Thesis

The rest of the thesis is structured as follows. Chapter 2 provides background about distributed graph processing and partitioning. Chapter 3 presents the idea of traffic-aware graph processing, where we consider heterogeneous network and vertex communication in our algorithms for graph partitioning. Moreover, we present vertex-centric implementations of two real-world graph algorithms: subgraph isomorphism and cellular automaton. Chapter 4 presents the idea of window-based streaming graph partitioning in order to trade partitioning latency against improved partitioning quality. Chapter 5 examines the idea of query-centric graph processing and partitioning in a multi-query graph system that processes multiple queries in parallel. Chapter 6 introduces the novel concept of fast and scalable hypergraph partitioning using neighborhood expansion. Finally, Chapter 7 concludes the thesis.

[5]https://github.com/mayerrn/HYPE

2

Background

In this chapter, we establish background information about distributed graph processing and graph partitioning.

2.1 Distributed Graph Processing

Initiated by Google's Pregel system [80], a multitude of distributed graph processing systems emerged. These systems share many core design decisions such as the execution model and the programming model. In the following, we introduce the models used in this thesis.

2.1.1 System Model

We assume a distributed environment consisting of a number of virtual or physical machines, which are connected via network links. Multiple processes communicate over the network via inter-process communication technology (e.g. via TCP) and work together on the distributed graph processing task. The graph system comprises of two types of processes: a single master process and k worker processes. The master orchestrates graph processing and performs house-keeping tasks such as synchronization between the workers (see Section 2.1.3). The workers perform the distributed graph processing task using inter-process communication to synchronize and exchange data.

Each worker is responsible for a single *graph partition*, i.e., a subgraph of the graph to be analyzed (see Section 2.2). The union of the disjoint graph partitions is the global *distributed data graph* (see Section 2.1.2). The workers are decoupled but need to communicate as given by the specific graph partitioning. With better partitioning quality comes a higher degree of decoupling and independence of different worker machines. We examine this dimension in detail

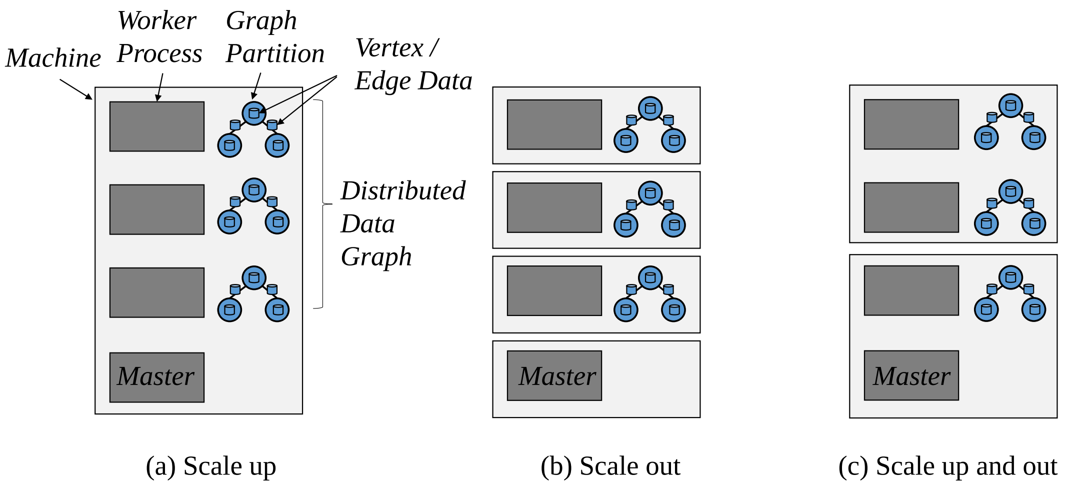


Figure 2.1: Scaling mechanisms for distributed graph processing (one CPU core per worker).

in the next chapters of this thesis. Decoupling the workers in this way enables parallel processing of the respective graph partitions and scaling computation to large graphs and complex algorithms.

Each process runs on a physical or a virtual machine and there can be multiple processes per machine. In Figure 2.1, we show the three different options that arise from this model. First, we can *scale up* computation to large multi-core machines by executing multiple workers on the same machine. Second, we can *scale out* computation by executing one worker process per machine and using multiple machines. Third, we can *scale up and out* by combining these ideas of assigning multiple workers to a single machine and increasing the number of machines.

2.1.2 Data Graph

Each graph processing application defines two types of data: (i) the graph structure and (ii) the vertex and edge data. (i) The graph structure, denoted as $G = (V, E)$ with the set of vertices V and edges E, captures the relationship between data entities. For example, in a social network, each user is represented as a graph vertex and each friendship as an edge between two vertices. The graph structure represents the underlying topology of the data—it encodes data dependencies (graph edges) between data entities (vertices). (ii) The *vertex data* and *edge data* hold the concrete data values defined by the application and maintained on the respective vertices and edges. For example, in a social network, each user vertex (object) maintains vertex data that quantifies the influence of this user, and each friendship edge (object) maintains edge data that quantifies the strength of the friendship relation. Distributed graph processing systems use the graph structure to optimize efficiency (e.g. via graph partitioning or message batching) [34, 35, 80].

2.1.3 Synchronization Model

In the synchronous vertex-centric execution model introduced by Pregel, each vertex acts as a virtual processing unit—each calculating a similar *vertex function* virtually in parallel[1]. The overall computation proceeds in a sequence of iterations. Each iteration can be seen as an isolated graph snapshot. The global state of the data graph proceeds in lock-step from the state in iteration i to the state in iteration $i+1$. To this end, vertex v calculates its own vertex data in iteration $i+1$ based on the vertex data of neighboring vertices in iteration i. This provides clear semantics to the application programmer: there are no race conditions among the millions of vertices which compute their vertex data in parallel. The model is said to be *synchronous* because no worker proceeds with iteration $i+1$ until all workers have finished iteration i. To reach this synchronous execution, a synchronization barrier is introduced: a worker that has finished execution for all vertices in iteration i sends a synchronization request to the master machine. The master waits for all synchronization requests from all workers before it initiates the next iteration $i+1$.

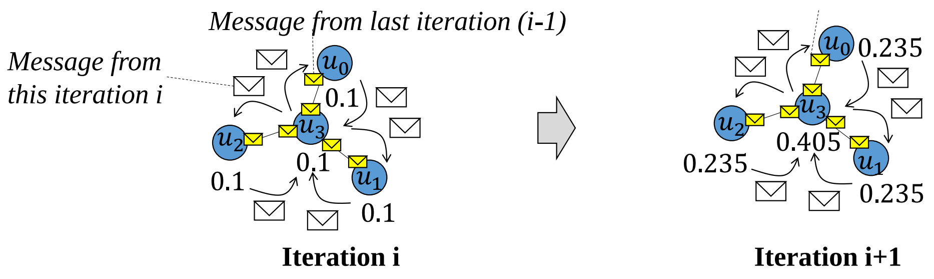
There is also work on asynchronous graph processing, where each vertex recomputes its own vertex data at its own pace. However, asynchronous graph processing is, in general, not superior to synchronous graph processing with respect to key performance metrics such as total graph processing latency. The reason is that asynchronous graph processing relies on locking of vertices to prevent read/write conflicts on the vertices: *"Async's poor performance is due to lock contention, the lack of message batching, and communication overheads in distributed locking."* [40]. Hence, we use the synchronous execution model which provides clearer semantics to the application programmer.

2.1.4 Programming Model

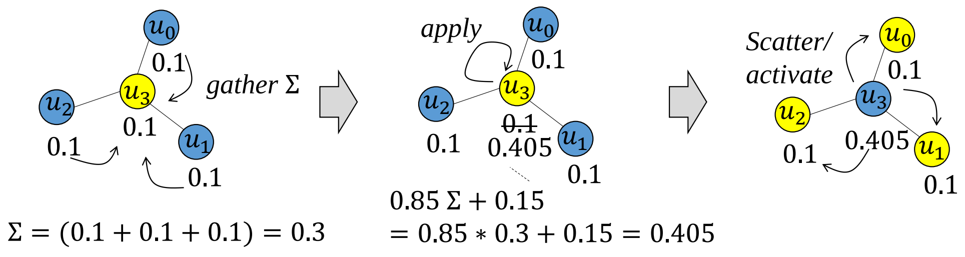
There are two types of popular vertex-centric programming models: the vertex compute model and the gather-apply-scatter model.

The *vertex compute model*, as proposed by Pregel, is implemented by a wide range of popular graph systems [20,21,66,80,90]. This programming abstraction is vertex-centric and message-based. The application programmer "*thinks like a vertex*" and specifies the vertex function that is executed by each graph vertex in an iterated manner. The vertex function expects a set of messages from neighboring vertices as input arguments. Based on these messages and the previous state of the vertex data, the vertex function recomputes and updates the vertex data. In the following, we denote the execution of the vertex function on a single vertex as *one* vertex function execution. During this execution, a vertex can send messages to neighboring vertices. Each message activates a neighboring vertex and transports updated information over

[1] A detailed explanation is given by Malewicz et al. [80].



(a) The PageRank algorithm in the vertex compute model



(b) The PageRank algorithm in the GAS programming model

Figure 2.2: Example PageRank algorithm.

the graph structure. Because of the vertex-centric programming abstraction, the application programmer is able to deploy complex graph algorithms on a distributed graph without the burden of implementing message transporting, synchronization, and partitioning.

In Figure 2.2(a), we give an example for the vertex compute model. Each vertex has an associated numerical data value that is the current relative PageRank of this vertex (not normalized here). Vertices exchange messages that contain the numerical vertex data. Based on the sum Σ of all input messages *from the last iteration*, vertex u_3 updates its own PageRank according to the formula $0.85\Sigma + 0.15$ (see [80]).

The **g**ather-**a**pply-**s**catter (GAS) model of computation operates on a higher abstraction level. The application programmer writes vertex-centric code with the convenient assumption that a vertex can *read* vertex data of neighboring vertices. The message flow is transparent to the application programmer—messages are created and propagated by the system. GAS computation works as follows. In the gather phase, the vertex collects information from all neighboring vertices. In the apply phase, the vertex computes its own vertex data based on the gathered data. In the scatter phase, the vertex updates the value of *edge data* and schedules neighboring vertices for execution (see [34]).

In Figure 2.2(b), we give an example for the GAS model. Vertices *gather* data from neighboring vertices in a gathered sum Σ. Based on the sum Σ of all input messages, vertex u_3 updates its own PageRank in a similar fashion (for clarity, we omit the results for the other vertices u_0, u_1, u_2). In contrast to the vertex compute model, all vertices perform the gather, apply, and

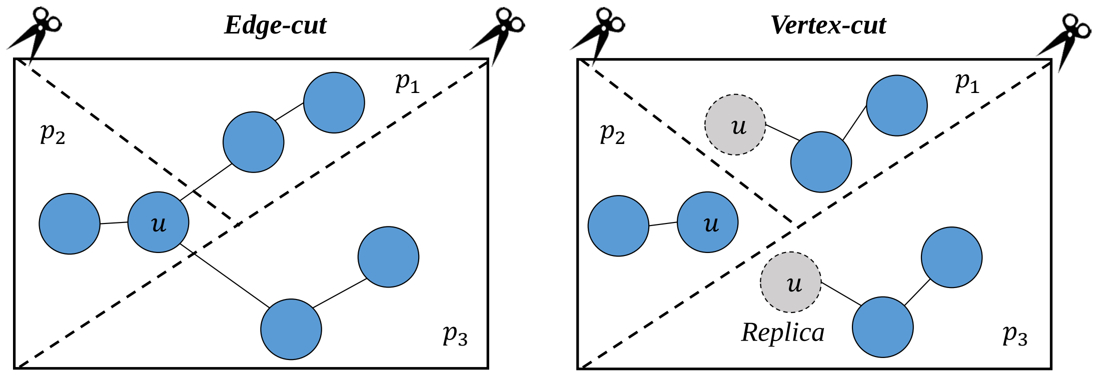


Figure 2.3: Vertex-cut and edge-cut partitioning.

scatter phases in lockstep. In other words, there is a synchronization barrier after each phase. Note that we skipped edge data for this minimal example.

Both programming models are popular design choices for distributed graph processing systems. In the next section, we investigate the two different partitioning strategies *edge-cut* and *vertex-cut* partitioning. In literature, systems that use edge-cut partitioning are defined in terms of the vertex compute model [80] and systems that use vertex-cut partitioning are defined in terms of the GAS model [34]. The reason is that the GAS model employes an *edge-centric* computation model specifically tailored to vertex-cut partitioning where each edge is assigned to a single partition [34]. We follow this common practice and use the vertex compute model in Chapter 5 and the GAS model in Chapters 3 and 4—because the systems use edge-cut partitioning and vertex-cut partitioning, respectively.

2.2 Graph Partitioning

To mitigate inefficient communication and synchronization overhead, graph processing systems require suitable partitioning methods improving the locality of vertex communication. Mainly, there are two types of partitioning strategies: edge-cut and vertex-cut. These strategies minimize the number of times an edge or vertex spans multiple partitions (*cut-size*).

2.2.1 Edge-cut Partitioning

Edge-cut partitioning algorithms divide the graph into multiple partitions by assigning each vertex exclusively to a single partition. In other words, vertices can not be replicated on multiple partitions. Figure 2.3 shows an edge-cut partitioning where six vertices are divided among three partitions. Vertex u has one neighbor on each partition. We have seen in Section 2.1.4 that in vertex-centric graph processing systems, neighboring vertices exchange information to recompute the vertex data values. For instance, when executing the vertex function of vertex u, the graph system propagates data from both non-local partitions p_1 and p_3 to partition p_2 and vertex u. Thus, existing graph systems assume that each edge that is cut by partition boundaries induces network communication during graph processing.

To this end, the main optimization objective for edge-cut graph partitioning algorithms is to minimize the number of cut-edges in the distributed graph. The hope is that this leads to reduced network communication during graph processing. To divide the graph processing workload evenly among k partitions, the number of graph vertices should be balanced. This problem is therefore denoted as *balanced k-way edge-cut partitioning* [123].

2.2.2 Vertex-cut Partitioning

Vertex-cut partitioning algorithms divide the graph to multiple partitions by assigning each edge exclusively to a single partition. Here, vertices can be replicated on multiple partitions. In Figure 2.3, we can see such a partitioning for the same graph. Vertex u is assigned to all three partitions p_1, p_2, p_3, i.e., it is replicated on all partitions. A vertex replica can only access a subset of all neighbors. Thus, the different replicas must exchange information to reach a consistent view of vertex u and its neighbors. We denote the average number of replicas per vertex the *replication degree* of a given partitioning.

The main optimization goal is to minimize the replication degree when dividing the graph among k partitions. To balance workload, each partition should be responsible for the same number of edges. This problem is denoted as *balanced k-way vertex-cut partitioning* [34].

2.2.3 Comparison Vertex-cut and Edge-cut Partitioning

Both partitioning approaches have advantages and disadvantages. Edge-cut partitioning leads to less overhead as vertices are not replicated and synchronized during graph processing [18]. On the other hand, vertex-cut partitioning produces better cuts for many real-world graphs with skewed degree distributions where some hub vertices are highly connected and central in the graph [34]. For example, the star graph with $|E|$ edges has an edge-cut size of $\Omega(|E|)$ but a vertex-cut size of $O(1)$. This scenario is given in Figure 2.4 where dividing the graph through the hub vertex u is superior to dividing the graph through the edges. Since real-world graphs often resemble a star-like degree distribution (e.g. *President Obama* in the Twitter social network), vertex-cut is often preferred for graph processing applications. Note that optimal edge-cuts can not be transformed into close-to-optimal vertex-cuts for graphs with high-degree vertices [30].

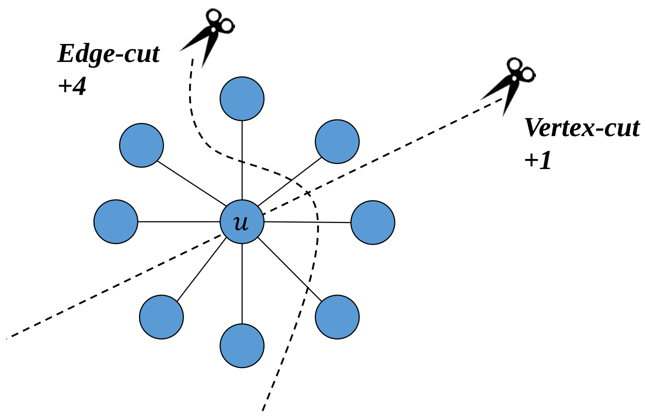


Figure 2.4: This star graph with hub vertex *u* can be cut through either one vertex or four edges.

In this thesis, we investigate both edge-cut partitioning in Chapter 5 and vertex-cut partitioning in Chapters 3 and 4.

3

Heterogeneous Partitioning for Distributed Graph Processing

As described in Chapter 2, distributed graph processing systems rely on suitable graph partitioning algorithms to reduce overhead for communication and synchronization at runtime. In this chapter, our target metric is *communication costs*, i.e., the observed network traffic during graph processing between two workers weighted by the actual costs of using the network link (for a more formal definition, please refer to Equation 3.2 in Section 3.1.2). This chapter is based on previously published work [86], [87], [89]. Regarding this objective, we tailor vertex-cut partitioning algorithms to the specific properties of distributed graph processing. We motivate these properties in the following paragraphs.

Existing partitioning approaches for distributed graph systems work under the assumption that decreasing the cut-size reduces communication costs due to less inter-worker traffic [34, 35]. But this holds only under two assumptions: *vertex traffic homogeneity*, i.e., processing each vertex involves the same amount of communication overhead, and *network costs homogeneity*, i.e., the underlying network links between each pair of workers have the same usage costs (e.g., [34, 115]). However, these assumptions oversimplify the target objective, i.e., *minimize overall communication costs*, for two reasons.

First, real-world vertex traffic is rarely homogeneous. This is due to computational hotspots, where processing is unevenly distributed across graph areas and vertices. Hotspots mainly arise for three causes.

I Vertices process different amounts of data. Examples are large-scale simulations of heart cells, liquids or gases in motion, and car traffic in cities [125, 127], where each vertex is responsible for a small part of the overall simulation. Vertices simulating real-world hotspots (e.g., the Times Square in NY) have to process more data.

II Graph systems execute vertices a different number of times. This can be observed for algorithms defining a convergence criteria for vertices. The graph system skips execution of converged vertices (*dynamic scheduling* [34]) leading to inactive graph areas and therefore different frequencies of vertex execution. Concerning this, a popular example is the PageRank algorithm [34].

III Graph algorithms inherently focus on specific graph areas. This includes user-centric graph algorithms such as k-hop random walk, finding a k-clique [89], and graph pattern matching. A prominent example is Facebook Graph Search, where users pose search queries to the system ("find friends who tried this restaurant").

In general, our evaluations show that vertex traffic often resembles a Pareto distribution, whereby a higher percentage of the total traffic is contributed by a much lower percentage of the vertices (see Section 3.1.2). Hence, we argue that assuming homogeneous vertex traffic misfits real-world, heterogeneous and dynamic traffic conditions in modern graph processing systems.

Second, network-related costs, such as bandwidth, latency, or monetary costs, are subject to large variations. Today, it is common to run graph analytics in the cloud, because of low deployment costs and high scalability [19, 34, 68]. Network heterogeneity exists even in a single data center where worker machines are connected via a tree-structured switch topology. Machines connected to the same switch experience high-speed communication, while distant machines suffer from degraded performance because of multi-hop networks [99]. Nevertheless, modern cloud infrastructures are *geo-distributed* [94]. Cloud providers such as Amazon, Google, and Microsoft are deploying dozens of data centers world-wide to provide low latency user-access. These data centers host global services such as Twitter that need to analyze large amounts of data (e.g., user friendship relations). Geo-distributed data analytics spanning *multiple* data centers is often the only option [134]. For instance, data should be stored close to the geo-distributed users to reduce access latency, but replication may be prohibitive for legal reasons or efficiency considerations [51, 104]. In these scenarios, network link costs can differ by orders of magnitudes (see Figure 3.1d, 3.13a, and Table 3.2). These heterogeneous network costs significantly influence overall communication costs and therefore should be considered when partitioning the graph.

To overcome these limitations, we developed GrapH, a distributed graph processing system for in-memory data analytics on graph-structured data[1]. GrapH is aware of both, dynamic vertex traffic and underlying network link costs. Considering this information, it adaptively partitions

[1] https://github.com/GrapH2/GrapH2.0

the graph at runtime to minimize overall communication costs by systematically *avoiding frequent communication over expensive network links*. It provides the following contributions. (i) An analysis of real-world vertex traffic for three different graph algorithms showing that vertex traffic is highly skewed. (ii) A fast single-pass partitioning algorithm, named *H-load*, and a fully distributed edge migration algorithm for runtime refinement, named *H-adapt*, solving the dynamic vertex traffic- and network-aware partitioning problem. H-adapt regulates the migration frequency between workers (*constant back-off migration*) and eliminates the need to exchange locking messages (*lock-free migration*). (iii) A method for online vertex traffic prediction, named *Adaptive-*α that treats *each individual vertex as an independent learner* and thereby is able to predict diverse and dynamic vertex traffic patterns. Compared to standard methods for time series prediction, Adaptive-α reduces the prediction error by 10-30%. Moreover, we show that accurate vertex traffic prediction is of major importance for efficiency of repartitioning in terms of overall communication costs. (iv) Two distributed algorithms plus implementation in the vertex-centric API solving two important graph problems: subgraph isomorphism to find arbitrary subgraphs in the graph, and agent-based cellular automaton to simulate physical phenomena such as mobility of citizens. In literature, there are currently no other vertex-centric graph algorithms for distributed graph systems solving these two graph problems. (v) Extensive evaluations on PageRank, subgraph isomorphism, and cellular automaton – for multiple graph data sets with up to 1.4 billion edges – showing that GraphH reduces communication costs by up to 60% and end-to-end graph processing latency (including overhead of dynamic repartitioning) by more than 10% compared to state-of-the-art partitioners.

This chapter is structured as follows. We formulate the network- and traffic-aware dynamic vertex-cut partitioning problem in Section 3.1, present the partitioning algorithms in Section 3.2, introduce the vertex functions for subgraph isomorphism and cellular automaton in Section 3.3, evaluate in Section 3.4, discuss related work in Section 3.5, and summarize in Section 3.6.

3.1 Preliminaries and Problem Formulation

In this section, we provide preliminaries and present the network- and traffic-aware dynamic partitioning problem.

3.1.1 Preliminaries

We assume the widely-used vertex-centric, iterative graph computation model from PowerGraph [34]. In each *iteration*, the system executes the user-defined vertex function for all *active* vertices and waits until they terminate (synchronized model, see Section 2.1.3). The vertex function operates on user-defined vertex data and consists of three phases, **G**ather, **A**pply

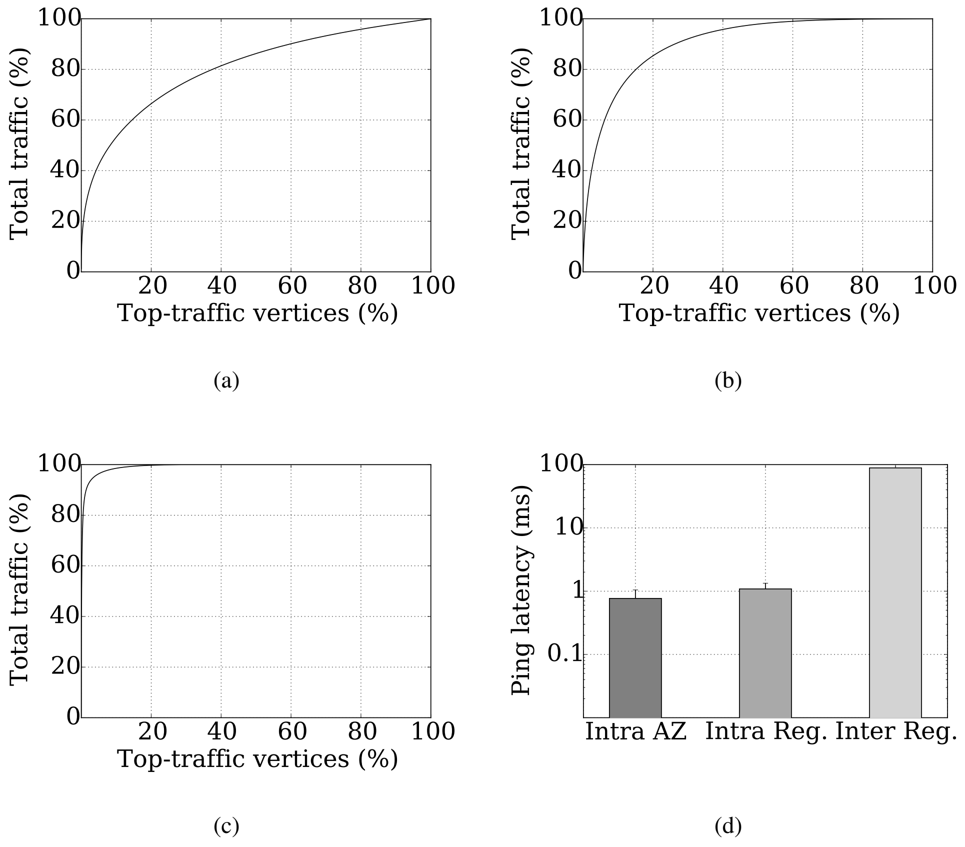


Figure 3.1: (a)-(c) Distribution functions of vertex traffic for PageRank, subgraph isomorphism, and cellular automaton. (d) Latency between worker machines intra-Availability Zone (AZ), inter-AZ and intra-region, and inter-region.

and **S**catter (GAS). In the gather phase, each vertex aggregates data from its neighbors into a *gathered sum* σ. In the apply phase, a vertex changes its local data using σ. In the scatter phase, a vertex activates neighboring vertices for the next iteration. For example, in PageRank each vertex has vertex data $rank \in \mathbb{R}$, gathers the sum σ over all neighbors' *rank* values, changes its vertex data *rank* using σ (i.e., $rank = 0.15 + 0.85 * \sigma$), and activates neighbors, if *rank* has changed more than a threshold.

To parallelize GAS execution, a graph is distributed onto multiple workers by cutting it through edges or vertices (*edge-cut* or *vertex-cut*) [34]. Edge-cuts distribute vertices to partitions, i.e., an edge can connect vertices on different partitions, and vertex-cuts distribute edges, i.e., a vertex can span multiple partitions, each having its own *vertex replica*. We formally denote the set of partitions, where vertex u is replicated, as *replica set* R_u. The *replication degree* is defined as the total number of vertex replicas. Because of its superior partitioning properties for real-world graphs with power-law degree distribution [34], we use vertex-cut in this chapter.

Inter-partition communication happens only in the form of *vertex traffic* between replicas. More precisely, if vertex v is distributed, replicas must communicate to access neighboring data. In this paragraph and Figure 3.2, we describe the message flow of the GAS protocol as introduced in [34]. The GAS protocol is well-researched for distributed graph systems that are based on vertex-cut partitioning [18, 34, 111]. We follow this computational model in this work. However, the conceptual contributions of heterogeneous-aware vertex-cut partitioning are not dependent on the concrete choice of the synchronization protocol. A dedicated *master replica* $\mathcal{M}_v$, manages the distributed vertex function execution and keeps vertex data consistent on the *mirror replicas* by exchanging three types of GAS messages (see Figure 3.2). First, a master sends a *gather request* to each mirror, which returns a *gather response* containing an aggregation of local neighboring data (e.g., summed page *rank*s). We denote the number of bytes, exchanged in the gather phase between the master of vertex v and a mirror r in iteration i as $g_r^v(i)$. Second, after computing the new vertex data in the apply phase, the master sends a *vertex data update* to all mirrors (e.g., the new *rank*). The size of this message, $a^v(i)$, depends on the local vertex data on the master. Third, in order to schedule neighbors of v for future execution, *scatter requests* of constant size s are exchanged between master and mirrors.

In general, vertex traffic between the master and the replicas of vertex v can vary due to a different size of the vertex data and the different amount of gathered data (e.g. gather sum being a union operation, see Section 3.3). In Equation 3.1, we define vertex traffic $t^v(i)$ of vertex v in iteration i as the summed byte size of gather, apply, and scatter messages. In order to make vertex traffic independent from the number of replicas to prevent partitioning oscillations (i.e., due to the wrong assumption that a vertex with good locality induces low traffic), we averaged vertex traffic $t^v(i)$ of vertex v in iteration i over all replicas in the replica set $R_v(i)$. Thus, the formula describes the *average vertex traffic between the master and the mirrors of vertex v in iteration i.*

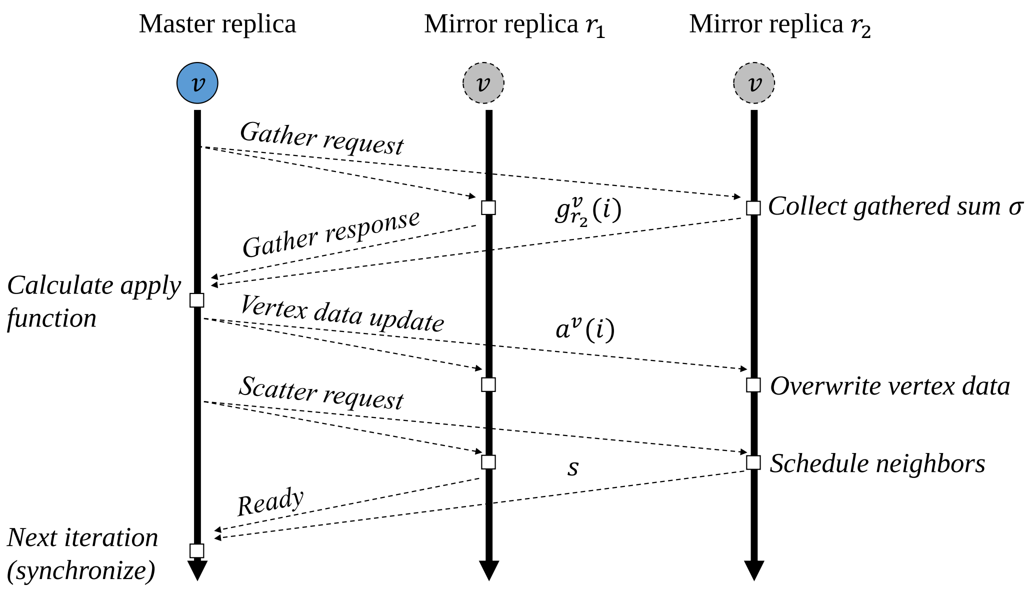


Figure 3.2: Message Flow of a single GAS iteration for vertex v.

$$t^v(i) = \frac{1}{|R_v(i)|} \sum_{r \in R_v(i)} (g^v_r(i) + a^v(i) + s) \tag{3.1}$$

3.1.2 Network- and Traffic-aware Dynamic Vertex-cut

Vertex traffic and network costs are heterogeneous. In Figure 3.1(a)-(c), we show vertex traffic heterogeneity for three algorithms: PageRank, subgraph isomorphism and cellular automaton (see Section 3.4 for details). The graph shows the x/y distribution: x% of the top-traffic vertices are responsible for y% of overall traffic. For instance, PageRank is 20/65 distributed because of different convergence behaviors of vertices as mentioned in the introduction of this chapter. Subgraph isomorphism is more extreme with a 20/84 distribution because some vertices match more subgraphs than others. Cellular automaton is highly imbalanced (20/100) because vertices simulating unpopular regions in Beijing have almost zero traffic (see Section 3.4). Besides vertex traffic, network communication is also subject to significant variations in terms of bandwidth and latency, even in a single data center [19, 70, 142]. For Amazon EC2 workers, we show orders-of-magnitude variations of latency (see Figure 3.1d). Likewise, many cloud providers charge variable *prices* for intra and inter data center communication. For instance, Amazon charges nothing for communication within the same availability zone (AZ), but for communication across different AZs and regions, i.e., respectively 0.01$/GB and 0.02$/GB

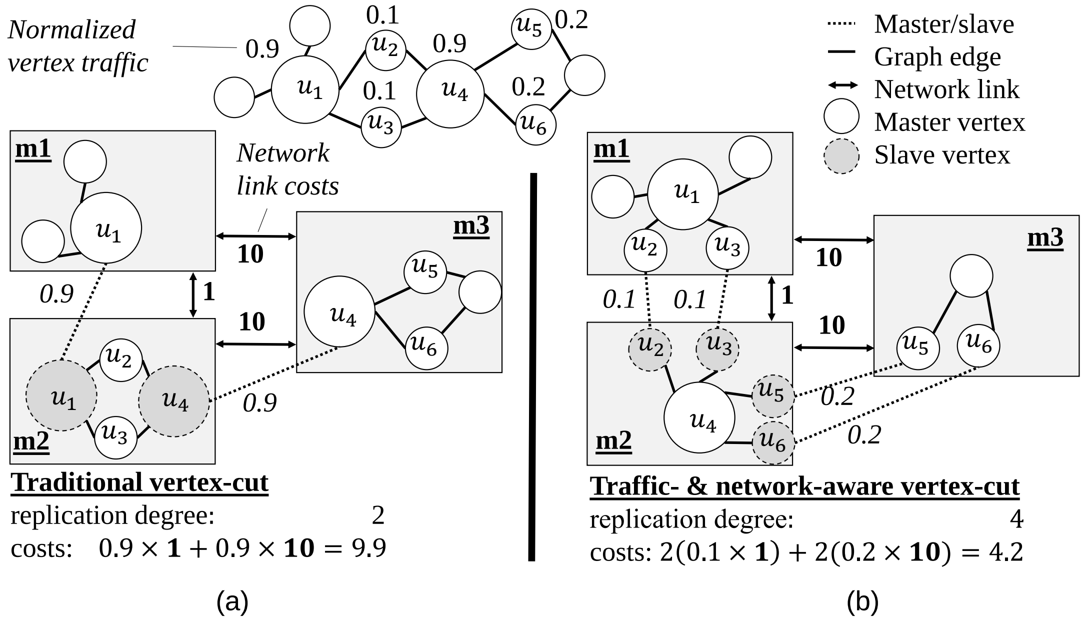


Figure 3.3: (a) Vertex-cut minimizing replication degree. (b) Network- and traffic-aware vertex-cut minimizing communication costs.

(see Table 3.2). We model this heterogeneity using a **static, weighted worker topology**, where each pair of workers is associated with specific network-related costs. In Section 3.4, we state precisely how the weighted worker topology can be determined in practical scenarios with very low overhead.

Efficient graph partitioning should take into consideration these diverse costs. For example in Figure 3.3(a), vertices are annotated with their (normalized) vertex traffic, also indicated by the vertex size. Workers $m1 - m3$ communicate via network links with different costs, given by the weights in bold. The vertex-cut distributes vertices u_1 and u_4, both have traffic 0.9. *Communication costs* via a network link are the costs of the link multiplied by the traffic sent over this link. For example, when measuring communication costs as monetary costs, sending 2GB over a link that costs 0.02$ per GB leads to communication costs of 0.04$. We define the *total communication costs* as the summed communication costs over all network links. In the example, total communication costs are $(0.9*1)+(0.9*10) = 9.9$. Here, traditional vertex-cut leads to minimal replication degree, but high communication costs, because high-traffic vertices u_1 and u_4 induce more network overhead. To this end, we introduce the network- and traffic-aware dynamic vertex-cut partitioning. The idea is to cut the graph on the low-traffic vertices to decrease inter-partition communication. In Figure 3.3(b), we minimize communication by cutting the graph on vertices u_2,u_3 and u_5,u_6. This increases the replication degree, but decreases overall communication costs. Note that this partitioning could be improved even further by ex-

M	The set of workers.
k	The number of workers.
$G = (V,E)$	The graph with vertex set V and edge set E.
V_m	The set of vertices replicated on worker m.
I	The set of all iterations.
$T_{m,m'}$	The network costs between workers m and m'
$\mathbb{A}$	Function mapping edges to workers.
R_v	Replica set of vertex v.
$\mathcal{M}_v$	Master of replica set of vertex v.
$t^v(i)$	Vertex traffic of vertex v in iteration i.
$\hat{t}^v(i)$	Vertex traffic estimation of vertex v for iteration i.
$L_m(i)$	Load (*summed vertex traffic*) of worker m in iteration i.
C	Capacity of exchange partner worker.
$\beta(x)$	Byte size of serialized information x (e.g. vertex).
μ	Aggressiveness parameter specifying *willingness-to-move*.
c_+	Investment costs of migrating edges.
c_-	Payback costs in terms of saved future traffic.

Table 3.1: Notation overview.

ploiting heterogeneous network link costs. Suppose, the subgraph assignments of m1 and m3 were swapped. Then, the (relative) high traffic vertices u_5,u_6 communicate over the inexpensive link (m1,m2), decreasing overall communication costs to $2(0.2*1)+2(0.1*10) = 2.4$. In Table 3.1, we summarize notation used in this chapter.

Problem Formulation: Let $G = (V,E)$ be a directed graph with the vertex set V and edge set $E \in V \times V$. Let $M = \{m_1,...,m_k\}$ be the set of all participating workers. The network cost matrix $T \in \mathbb{R}^{k\times k}$ assigns a cost value to each pair of workers (e.g., monetary costs for sending one byte of data). Hence, $T_{m,m'}$ represents the network costs between workers m and m'. The set of all iterations needed for the graph processing task be $I = \{0,1,2,...\}$. Vertex traffic for all iterations $i \in I$ and vertices $v \in V$ is denoted as $t^v(i)$. The assignment function $\mathbb{A} : E,I \to M$ specifies the mapping of edges to workers in a given iteration. The *replica set* of vertex v in iteration i based on assignment function $\mathbb{A}$ is denoted as $R_v^{\mathbb{A}}(i)$. It represents the set of workers maintaining a replica of v: $R_v^{\mathbb{A}}(i) = \{m|\mathbb{A}((u,v),i) = m \vee \mathbb{A}((v,u),i) = m\}$. In the following, we denote R_v to be v's replica set under the assignment in the present context. One dedicated replica $\mathcal{M}_v \in R_v$ is the *master replica* of vertex v.

Our goal is to *find an optimal dynamic assignment of edges to workers minimizing overall communication costs*:

Worker placement	Incoming traffic	Outgoing traffic
Same AZ	0.00-0.01 \$/GB	0.00-0.01 \$/GB
Different AZ, same region	0.01 \$/GB	0.01 \$/GB
Different region	0.00 \$/GB	0.02 \$/GB
Internet	0.00 \$/GB	0.00-0.09 \$/GB

Table 3.2: Heterogeneous network link costs for AWS (Jan 2018).

$$\mathbb{A}_{opt} = \operatorname*{argmin}_{\mathbb{A}} \sum_{i} \sum_{v \in V} \sum_{m \in R_v^{\mathbb{A}}(i)} t^v(i)\; T_{m,\mathcal{M}_v} \tag{3.2}$$

The load $L_m(i)$ of worker m in iteration i is defined as the summed vertex traffic over all vertices replicated on m in iteration i. To balance worker load, we require for each iteration i and worker m (having vertices V_m) that load deviation is bounded by a small balancing factor $\lambda > 1$:

$$L_m(i) = \sum_{v \in V_m} t^v(i) < \lambda \frac{\sum_{v \in V} t^v(i)}{|M|}. \tag{3.3}$$

Theorem: The dynamic network- and traffic-aware partitioning problem is NP-hard.

Proof sketch: Reduce the NP-hard balanced vertex-cut problem to Equation 3.2. Set input: $I = \{1\}$, $t^v(i) = 1$, $T_{m_1,m_2} = 1$. By, $\mathbb{A}_{opt} = \operatorname{argmin}_{\mathbb{A}} \sum_i \sum_{v \in V} \sum_{m \in R_v^{\mathbb{A}}(i)} 1 * 1 = \operatorname{argmin}_{\mathbb{A}} \sum_{v \in V} |R_v^{\mathbb{A}}|$, Equation 3.2 becomes the network- and traffic-*unaware* vertex-cut problem, which is NP-hard (e.g., [101]).

□

3.2 Partitioning Algorithms

In this section, we present two algorithms addressing the network- and traffic-aware partitioning problem: **H-load** for pre-partitioning the graph and **H-adapt** for runtime refinement using dynamic migration of edges.

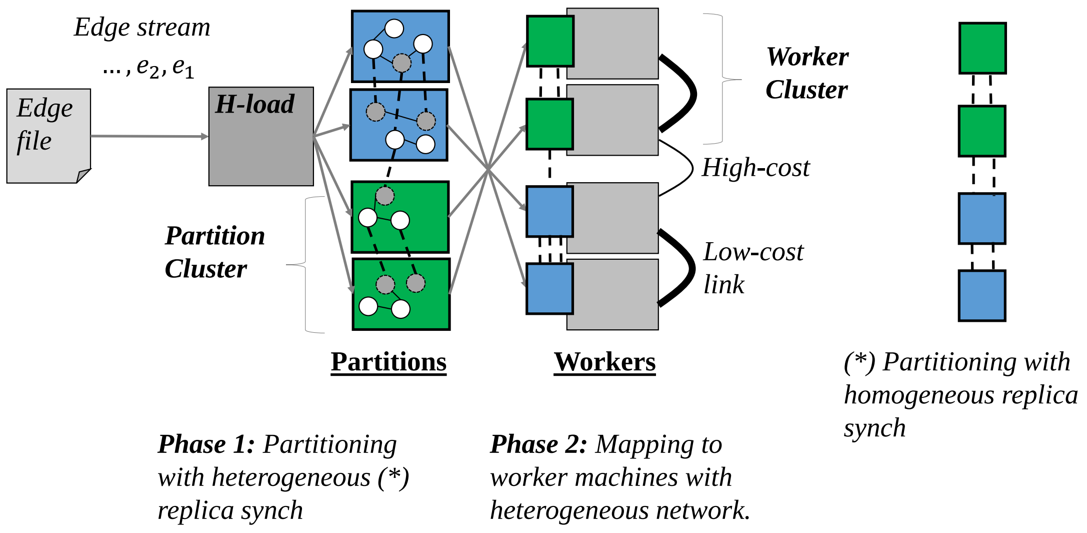

Edge stream
$\ldots, e_2, e_1$
Edge file
H-load
Partition Cluster
Partitions
Workers
Worker Cluster
High-cost
Low-cost link
(*) Partitioning with homogeneous replica synch
Phase 1: Partitioning with heterogeneous (*) replica synch
Phase 2: Mapping to worker machines with heterogeneous network.

Figure 3.4: Approach Overview H-load.

3.2.1 H-load: Initial Partitioning

Graph processing systems pre-partition the graph, so that each worker can load its partition into local memory. To this end, we developed H-load, a fast pre-partitioning algorithm that consists of two phases (see Figure 3.4). First, it divides the graph into k partitions using a vertex-cut algorithm. Second, it determines a cost-efficient mapping from partitions to workers. We describe these two phases in the following.

1) The goal of the first phase is to partition the graph into k balanced parts, while ignoring the concrete mapping of partitions to workers. In order to improve partitioning performance of billion-scale graphs, we assume a streaming setting: the graph is given as a stream of edges $e_1, e_2, \ldots, e_{|E|}$ with $e_i \in E$ and we consecutively read and assign one edge at a time to a partition until all edges are assigned. For instance, PowerGraph [34] greedily reduces the replication degree by moving edges to the partitions where incident vertices already reside. However, the PowerGraph partitioning leads to homogeneous replica synchronization between each pair of partitions as they share a similar amount of replicas (see Figure 3.4 right).

In order to exploit heterogeneity of network link costs, inter-partition traffic must be heterogeneous as well: partitions exchanging more traffic should be mapped to workers with low-cost network links (see Figure 3.4 center). Therefore, our aim in the first phase of H-load is to produce a clustered partitioning consisting of *partition clusters* such that two partitions with many shared replicas are likely to reside in the same partition cluster. The partition clusters are designed to have high intra- and low inter-cluster traffic, i.e., more traffic will be exchanged between partitions in the same partition cluster.

How can we ensure that a mapping from these partition clusters to the workers exists such that communication costs during graph processing are minimal? H-load addresses this problem by assuming that the network consists of several *worker clusters* with low intra-cluster and high inter-cluster costs (e.g., EC2 instances running in different availability zones). This clustering is reflected in the cost matrix T. The number of worker clusters c can be determined from the matrix T using well-established clustering methods [39]. We use the number of worker clusters c in the first phase of H-load to generate c partition clusters by grouping the partitions into equal-sized clusters and assigning edges to partitions such that replicas preferentially lie in the same partition cluster. Hence, the relation between the worker clusters and the partition clusters is the following: we use the number of worker clusters c to produce c partition clusters. Therefore, the first phase operates independently of the concrete mapping of partitions to workers.

Each edge (u,v) is assigned to a partition p as follows. If neither vertex u nor vertex v resides in any partition, assign (u,v) to the least loaded partition. If there are partitions where both vertices u *and* v reside, assign (u,v) to the least loaded of those partitions. Otherwise, a new replica has to be created (say, of vertex v). We choose partition p, such that the new replica of v preferentially lies in the same partition cluster as already existing replicas of v – simply by counting the number of replicas of v in each partition cluster. With this method, our algorithm ensures a clustered traffic behavior: partitions in the same partition cluster are more likely to share the same vertices than partitions in different clusters. Thus, two partitions in the same partition cluster are expected to exchange more traffic than two partitions in different partition clusters.

2) The second phase of the algorithm determines a mapping from the $|M|$ partitions to $|M|$ workers while minimizing overall communication costs. A minimal-cost mapping of partitions to workers would assign two partitions with high inter-partition traffic to workers connected via a low-cost network link (see Figure 3.4). This is an instance of the well-known quadratic assignment problem: map $|M|$ factories (i.e., partitions) to $|M|$ locations (i.e., workers), so that the mapping has minimal costs of factories sending their goods to other factories (i.e., communication costs). We used the iterated local search algorithm of Stützle et al. [124] to minimize (communication) costs. Initially, partitions are randomly mapped to workers. Then the algorithm iteratively improves the total costs using the following method. Find two workers such that exchanging partition assignments results in lower total communication costs. For example in Figure 3.3, exchanging partition assignments of workers $m1$ and $m3$ results in lower total communication costs. If an improvement is found, it is applied immediately. In order to address convergence to local minima, we perturb a local optimal solution by exchanging two random assignments. Note, that this algorithm is computationally feasible, because the problem

set is relatively small with size $|M| << |V|$. Clearly, the above method assumes that the traffic exchanged between each pair of partitions is known (i.e., cumulative traffic exchanged between vertex replicas shared by each pair of partitions). This information can be determined from previous executions of the GAS algorithm. Otherwise, homogeneous traffic between vertex replicas is assumed.

3.2.2 H-adapt: Distributed Migration of Edges

The H-load algorithm is suitable for a static network-aware and traffic-aware partitioning. However, often the vertex traffic changes dynamically at runtime. To this end, we developed the distributed edge-migration algorithm *H-adapt* solving the dynamic heterogeneity-aware partitioning problem (note that this algorithm extends our previous algorithm H-move [86] by three optimizations (i) adaptive-α, (ii) constant back-off migration, and (iii) lock-free migration). The idea is that each worker locally reduces the communication costs by migrating edges to distant workers. To this end, we define the term *bag-of-edges* as the set of edges to be migrated. Workers migrate bag-of-edges in parallel after each GAS iteration.

Approach overview: The overall migration strategy is given in Algorithm 1. After activation of the migration algorithm (line 1), worker m first selects partner worker m' (line 2) and then calculates the bag-of-edges to be send to m' (line 3). In order to prevent inconsistencies due to parallel updates on the distributed graph, worker m requests locks for all vertices in the bag-of-edges (line 4). Afterwards, m updates the bag-of-edges to contain only those edges, whose endpoint vertices could be locked (line 5) and determines, whether sending the updated bag-of-edges results in lower total communication costs (line 6). When sending the bag-of-edges, communication costs change due to modifications of the vertex replica sets. To calculate Δc in line 6, worker m considers both: the migration overhead c_+ of sending the bag-of-edges, as well as the decrease of communication costs c_- when improving the partitioning. If Δc is negative, the bag-of-edges is migrated to m'. Finally, worker m releases all held locks in line 9.

Algorithm 1 Migration algorithm on worker m.

```
1: $waitForActivation()$
2: $m' \leftarrow selectPartner()$
3: $b \leftarrow bagOfEdges(m')$
4: $lock(b)$
5: $b \leftarrow updateLocked(b)$
6: $\Delta c \leftarrow c_+ - c_-$
7: if $\Delta c < 0$ then
8:     $migrateBag(b)$
9: $releaseLocks(b)$
```

We give an example of this procedure in Figure 3.5. Two workers m and m' have replicas of high-traffic vertices u_2 and u_3. In order to reduce communication costs, m decides to send the

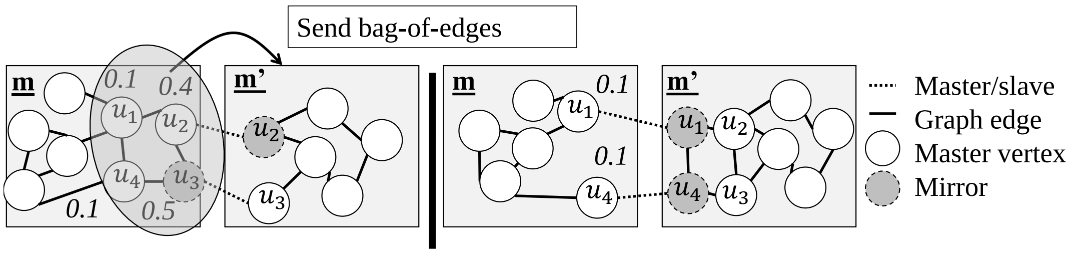


Figure 3.5: Example: bag-of-edges migration to reduce inter-partition traffic.

bag-of-edges $b = \{(u_1,u_2),(u_2,u_3),(u_3,u_4),(u_4,u_1)\}$ to m'. Worker m' receives b and adds all edges in b to the local subgraph. The right side of the Figure 3.5 shows the final state after migration of b. Here, low-traffic vertices u_1 and u_4 are cut leading to less inter-partition traffic. In the following, we describe the proposed H-adapt algorithm (see Algorithm 1) in more details.

Selection of partner and bag-of-edges

Which worker to select as exchange partner? Intuitively, two workers sharing high-traffic replicas are strong candidates for exchanging bag-of-edges, because improving their partitioning can greatly reduce the overall communication costs. On the other hand, two workers sharing no or only low-traffic replicas have low potential to improve overall costs. Hence, some workers are more promising partners than other workers for the edge migration. Therefore, each worker m maintains a candidate list of potential exchange partners (with decreasing priority). Worker m computes the candidate list by sorting neighboring workers w.r.t. the total amount of exchanged traffic. In each round of Algorithm 1, we iteratively select the top-most worker from the list as exchange partner and remove it from the list. Once the list is empty (i.e., all workers have exchanged bag-of-edges with each other), it is recomputed using the most recent traffic statistics.

However, this strategy leads to suboptimal or redundant selections of exchange partners in the following three cases. First, worker m has already selected worker m' in one of the previous i' iterations and can only find minor improvements of the partitioning. Second, worker m has less load than worker m' and the load balancing requirement prohibits any migration from m to m'. Third, workers m and m' share no common replicas. In these cases, we do not expect to find significant improvements of the current partitioning. Therefore, we remove worker m' from the candidate list for a constant number of iterations. During this time period, worker m' can not be selected as exchange partner (hence, denoted as *constant back-off migration*). We set the parameter for back-off migration to half the number of partitions, i.e., $k/2$, and set it to infinity as soon the algorithm detects a convergence that is assumed when the bag-of-edges for an exchange partner is empty.

Next, we determine the maximal size of the bag-of-edges to be sent to m'—depending on how balanced the workload is between workers—in order to improve workload balancing between these two workers (see Equation 3.3). Therefore, we introduce the notion of *capacity* of a worker m', i.e., the maximum amount of additional load, worker m' can carry. Capacity is defined as half the difference of loads $L_{m'}$ and L_m of the receiving and the sending worker: $C = (L_{m'} - L_m)/2$. To learn about the current load $L_{m'}$ of worker m', worker m sends a request to m'. Using the capacity, worker m can control the size of the bag-of-edges, such that load deviation is still bounded. For example, if sending the bag-of-edges results in a new replica of vertex v on m', this increases load of m' by the vertex traffic of v. If this violates load balancing between m and m', worker m will not include v into the bag-of-edges.

Once the exchange partner is selected and we know its capacity, we determine a bag-of-edges (in short: *bag*) to be send. Selecting a suitable bag is crucial for optimizing communication costs and migration overhead. Theoretically, the perfect bag could be any subset out of p edges on a worker (i.e., 2^p subsets). In order to keep the migration phase lean, we developed a fast heuristic to find a bag improving communication costs (see Algorithm 2). Initially, worker m determines the set of candidate vertices, those replicated on both workers, because they are responsible for all the traffic between m and m'. Worker m sorts the candidates by descending vertex traffic in order to focus on the high-traffic vertices first (line 3). Then, m iterates the following steps until m' has no more capacity. It checks for the top-most candidate vertex (line 5), whether sending all adjacent edges results in lower total communication costs of the overall graph processing (line 6-7, cf Section 3.2.2). If the total communication costs would decrease when sending the edges, worker m adds them to the bag (line 8-9).

Algorithm 2 Determining the bag-of-edges to exchange.

```
1: function BAGOFEDGES($m'$):
2:     $bag \leftarrow []$
3:     $candidates \leftarrow sort(adjacent(m'))$
4:     while $hasCapacity(m', bag)$ do
5:         $v \leftarrow candidates.removeFirst()$
6:         $b \leftarrow \{(u,v) | u \neq v\}$
7:         $\Delta c \leftarrow c_+ - c_-$
8:         if $\Delta c < 0$ then
9:             $bag \leftarrow bag + b$
    return $bag$
```

Calculation of costs

Clearly, migrating bag b from one worker to another is only beneficial, if it results in lower overall costs (i.e., line 6 in Algorithm 1, and line 7 in Algorithm 2). In general, two types of costs have to be considered in calculating the resulting overall costs: *investment costs* and *payback costs*. Investment costs represents the overhead for migrating the bag and should be avoided. Payback costs are the saved costs after migrating bag b in the form of less future inter-partition traffic. In the following, we formulate both costs.

Investment costs: After sending b to m', m can remove isolated replicas that have no local edges anymore (Figure 3.5 vertices u_2, u_3). If worker m is the master $\mathcal{M}_v$ of a vertex v to be removed, i.e., $\mathcal{M}_v = m$, we have to select a new master after removing v from m. We set the partner worker m' to be the new master of v: $\mathcal{M}'_v = m'$. On the other hand, some vertices may not exist on m' leading to creation of new replicas (Figure 3.5 vertices u_1, u_4). In both cases, the replica set R_u of a vertex u might have changed (i.e., remove m or add m' to R_u). Because of this, m has to send an update to all workers in R_u with the new vertex replica set, denoted as R'_u. Additionally, when creating a new replica on m', worker m has to send the state of v, i.e., vertex data and meta information such as the vertex id. This can be very expensive for large vertex data, and should be taken into account when deciding whether to migrate a bag. Together, the investment costs are the sum of three terms. The first term calculates the costs of sending the bag b to m'. The second term calculates the costs of sending new replicas to m', if needed. The third term calculates the costs of updating workers in all replica sets that have changed.

$$c_+ = \sum_{e=(u,v)\in b} \beta(e) T_{m,m'} + \sum_{u \in V_b} \delta(u)\beta(u) T_{m,m'} + \sum_{u\in V'_b, r \in R_u \cup R'_u} \beta(R_u) T_{m,r}, \tag{3.4}$$

where (i) the function $\beta(x)$ returns the number of bytes needed to encode x (to be sent over the network), (ii) the indication function $\delta(u)$ returns 1, if worker m' has no local replica of u, otherwise 0, (iii) V_b is the set of all vertices in bag b, and (iv) V'_b is the set of all vertices whose replica sets will change when sending the bag b to m'.

Payback costs: we can also save costs when sending bag b from m to m'. Suppose the replication degree decreases because of sending (u, v), i.e., $|R'_u| < |R_u|$ or $|R'_v| < |R_v|$. Then, we save *for each iteration* (starting from the current iteration i_0) the costs of exchanging gather, apply, and scatter messages across replicas, i.e., the vertex traffic $t^v(i)$ of vertex v in iteration i. Theoretically, exact payback costs are given by the following formula that calculates for all future iterations and each vertex in the bag the difference of the new costs and the old costs of v's replica set.

$$c^*_- = \sum_{i>i_0} \sum_{v\in V_b} \Big(\sum_{r\in R'_v} t^v(i) T_{r,\mathcal{M}'_v} - \sum_{r\in R_v} t^v(i) T_{r,\mathcal{M}_v} \Big) \tag{3.5}$$

Here, we assumed that vertex traffic is known for all future iterations. This is not the case in real systems. Therefore, we describe next, how to estimate the payback costs in the presence of uncertainty about future vertex traffic.

Vertex Traffic Prediction

To estimate payback costs, we first need to predict vertex traffic in future iterations. More formally, given vertex traffic $t^v(0), t^v(1), ..., t^v(i)$ of vertex v, we estimate traffic values $t^v(i+1), ..., t^v(|I|)$. The prediction should be fast with low computational overhead and low memory requirements, because we have to predict vertex traffic in each migration phase for millions of vertices. We investigate three well-known methods (compare [46]) for time series prediction of the next traffic value $t^v(i+1)$ that fit our requirements.

- The first method is *most recent value* (denoted as *Last*) taking the last traffic value as prediction for the next traffic value: $\hat{t}^v(i+1) = t^v(i)$.
- The second method is *incremental moving average* (abbreviated as *MA*) with the idea of using the moving average of the last w observations, while not storing the values in the window: $\hat{t}^v(i+1) = \frac{\hat{t}^v(i)(w-1)+t^v(i)}{w}$.
- The third method is *incremental exponential average* (abbreviated as *EA*) calculating the prediction based on the previous prediction and the last observed traffic value: $\hat{t}^v(i+1) = \alpha t^v(i) + (1-\alpha)\hat{t}^v(i)$. The parameter $\alpha \in [0,1]$ specifies the amount of decaying older traffic values and thus, the importance of recently observed traffic.

Which prediction method performs best? In Figure 3.14, we compare overall system performance for these methods with different parameter choices, i.e., window sizes w and decay parameters α. In summary, the third method, i.e., exponential averaging, leads to the best overall system performance for certain choices of the parameter α. The reason is that exponential averaging is able to express both rapid (i.e., short term) and gradual (i.e., long term) changes in vertex traffic by varying the parameter α. For instance, setting α to a high value (e.g. $\alpha = 0.9$) is better for predicting rapidly changing vertex traffic whereas setting α to a low value (e.g. $\alpha = 0.1$) leads to more accurate long term traffic pattern prediction. A major challenge is to select the correct parameter α according to the concrete traffic pattern of the graph algorithm. Furthermore, assuming a *global and static* parameter value α for all vertices is unrealistic for many applications such as social simulations, where some vertices are responsible for simulating long-term, repeating movement patterns (e.g. in a work environment) and others for simulating short-term, unique movement patterns (e.g. a flash mob). In such cases, the graph system should select the parameter α automatically and specifically for each vertex.

Adaptive-α: To this end, we developed our method Adaptive-α where each vertex learns its individual prediction model by dynamically optimizing α at runtime. This releases the system administrator from the burden of choosing the optimal parameter value and empowers vertices to catch their individual diverse and dynamic traffic patterns. In more details, each vertex continuously monitors the accuracy of its predictions by comparing its previous vertex traffic predictions with the actually observed vertex traffic. For a given α, the prediction accuracy of a vertex v in iteration i can be quantified by using the error function $e_{v,i}(\alpha)$ as shown in Equation 3.6.

$$e_{v,i}(\alpha) = (t^v(i) - \hat{t}^v(i))^2 \tag{3.6}$$

In order to find the value of α that minimizes the error function, we set the first order derivative of Equation 3.6 to zero, i.e., $\frac{\partial e_{v,i}}{\partial \alpha} = 0$, and calculate the extremum of $e_{v,i}(\alpha)$ as follows:

$$\frac{\partial}{\partial \alpha}(t^v(i) - (\alpha t^v(i-1) + (1-\alpha)\hat{t}^v(i-1)))^2 = 0 \tag{3.7}$$

$$\Leftrightarrow \alpha_{min} := \alpha = \frac{\hat{t}^v(i-1) - t^v(i)}{\hat{t}^v(i-1) - t^v(i-1)} \tag{3.8}$$

If the extremum is a local minimum (i.e., the second order derivative is greater than zero) and $\alpha_{min} \in [0,1]$, we set $\alpha = \alpha_{min}$. Note that each vertex can quickly calculate α_{min} by simply applying the closed formula in Equation 3.8 (see Section 3.4, Figure 3.12c).

With the above mentioned methods, we can determine the vertex traffic estimation for the next iteration. However, Equation 3.5 expects a vertex traffic value for all future iterations. In general, accuracy of the predicted vertex traffic $\hat{t}^v(i)$ can decrease with increasing i, because vertex traffic patterns may change over time. Therefore, we introduce a factor μ representing the minimum number of iterations we expect to save communication costs as a result of migrating bag b. This parameter specifies the aggressiveness with which migration should be performed. We set it to the total number of iterations specified by the GAS algorithm minus the current iteration, i.e., to the (expected) number of iterations left in the algorithm. Together, our estimated payback costs are the following (denoting the new master of vertex v as $\mathcal{M}_v'$ and the old master as $\mathcal{M}_v$ – see Equation 3.5).

$$c_- = \mu \sum_{v \in V_b} \Big(\sum_{r \in R_v'} \hat{t}^v(i+1) T_{r,\mathcal{M}_v'} - \sum_{r \in R_v} \hat{t}^v(i+1) T_{r,\mathcal{M}_v} \Big) \tag{3.9}$$

Lock-free Migration

When two workers independently migrate edges and change replica sets of vertices, inconsistencies of the data graph can arise. In Figure 3.6, we give an example. Workers $m1$-$m4$ maintain a replica of vertex u. Initially, all workers have the same view of the replica set $\{m1, m2, m3, m4\}$. Now, suppose worker $m4$ migrates edge (u, v_1) to worker $m3$. At the same

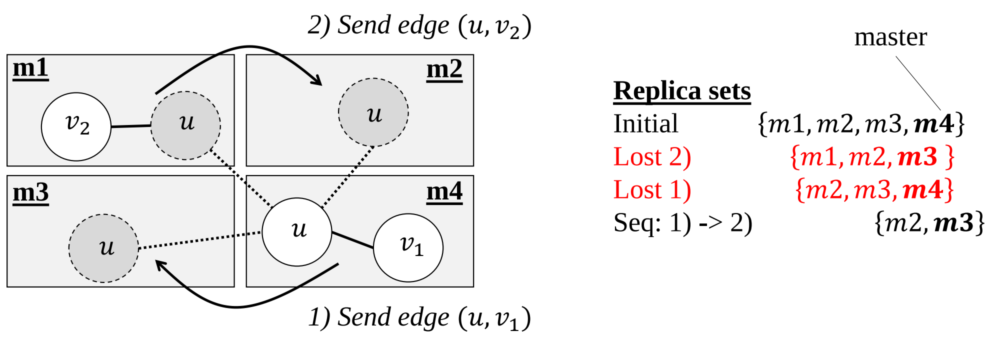

2) Send edge (u, v_2)
m1
m2
m3
m4
v_2
u
u
u
u
v_1
1) Send edge (u, v_1)
master
Replica sets
Initial $\{m1, m2, m3, \mathbf{m4}\}$
Lost 2) $\{m1, m2, \mathbf{m3}\}$
Lost 1) $\{m2, m3, \mathbf{m4}\}$
Seq: 1) -> 2) $\{m2, \mathbf{m3}\}$

Figure 3.6: Lost update problem for parallel edge migration.

time, worker $m1$ migrates edge (u, v_2) to $m2$. Both workers $m1$ and $m4$ remove the local replica of vertex u, if no incident edge exists on the respective worker. In order to synchronize the replica sets, worker $m4$ ($m1$) has to update all other workers having a replica of u with the new replica set. Worker $m4$ sends the new replica set $\{m1, m2, m3\}$ to all workers, while worker $m1$ sends $\{m2, m3, m4\}$. However, different workers can receive these updates in different orders leading to inconsistent views on the replica sets (lost update problems). Instead, sequential updates would result in a consistent state (e.g. worker $m4$ changes the replica set *before* worker $m1$).

To guarantee sequential updates during edge migration, the standard procedure is to send locking requests to the masters of the vertices to be migrated [86]. More precisely, a worker m locks endpoint vertices in the bag-of-edges to be sent to the partner worker m'. For vertex u it sends a locking request to the master $\mathcal{M}_u$. If vertex u is already locked, the master worker returns $false$, otherwise it locks u and returns $true$. If a worker holds a lock, no other worker can change the replica set of a vertex and the distributed graph is always in a consistent state.

However, acquiring all locks via locking messages leads to significant latency penalty (at least one round trip time per migration phase) and message overhead (one locking message per vertex in the bag-of-edges). To save this additional overhead, we designed a so called *implicit locking scheme* that enables workers to acquire locks for vertices without the need for explicitly exchanging lock messages. In Algorithm 3, we describe our implicit locking scheme. The given function returns true if worker m owns the lock for vertex v. In order to enable each worker m to migrate each combination of vertices eventually, worker m possesses locks for all vertices every k-th iteration (line 2)—the value k being the number of participating workers. This ensures fairness because each worker has an equal chance of getting all locks. However, a worker does not need the lock for vertex v if it does not possess a vertex replica of vertex v. Therefore, the lock of this vertex v is given to another worker that contains a replica of vertex v and therefore might actually need it for migration (see formula in line 4). This formulaic approach does not require any communication overhead at runtime: the workers execute the same formula having the same values for the current iteration and the replica set R_v. Thus, they also calculate the same value for the worker currently having the lock of vertex v. To sum up, this simple technique allocates vertex locks without inducing any locking message.

Algorithm 3 Implicit locking scheme. Does worker m have the lock for vertex v?

```
1: function HASLOCK($m, v$)
2:     $\hat{m} \leftarrow iteration()$ mod $k$
3:     if $\hat{m} \notin R_v$ then
4:         $\hat{m} \leftarrow R_v.sort()[iteration()$ mod $|R_v|]$
5:     return $m == \hat{m}$
```

3.3 Graph Algorithms

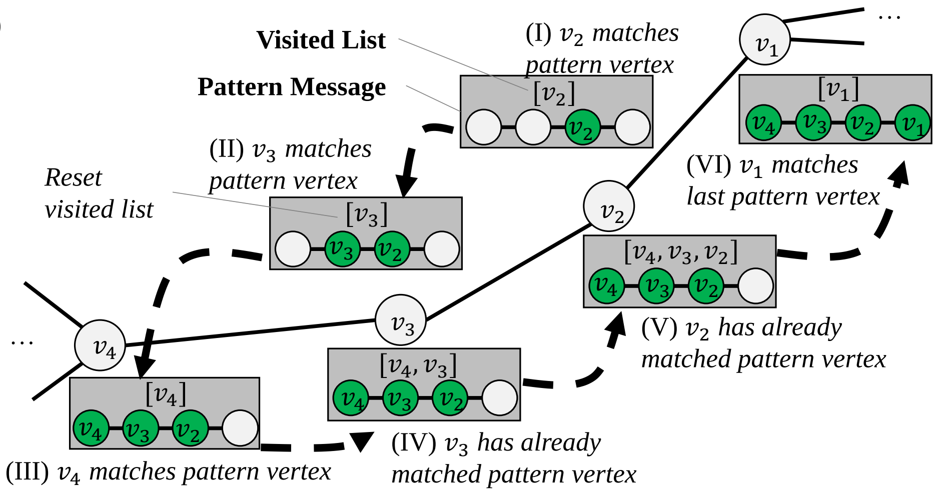
In order to evaluate our partitioning methods and heterogeneity of vertex traffic, we have implemented three important graph algorithms: PageRank (see [34]), subgraph isomorphism, and social simulations via agent-based cellular automaton, denoted as PR, SI, and CA, respectively. For SI and CA there is, to the best of our knowledge, no vertex-centric algorithm, so we have designed novel algorithms and implemented them in the GAS API.

3.3.1 Subgraph Isomorphism

The NP-complete subgraph isomorphism problem addresses the question, whether a graph contains a subgraph that is isomorphic to a specified subgraph [130]. Given an undirected graph $G = (V,E)$ and a graph pattern $P = (V_P, E_P)$, the problem of subgraph isomorphism is to find subgraphs $G_{sub} = (V_{sub}, E_{sub})$, with $V_{sub} \subseteq V, E_{sub} \subseteq E$, that are isomorphic to the graph pattern P (see [130]). More precisely, there is a bijective mapping (in the following denoted as *matching*) $f : V_{sub} \to V_P$, such that $(u,v) \in E_{sub} \Leftrightarrow (f(u), f(v)) \in E_P$. Each graph vertex can have an optional label (e.g., a category to which the graph vertex belongs). In this case, the labeled SI additionally requires both vertices in V_{sub} and V_P to have matching labels. Specifically, the following condition has to hold for vertex $v \in V_{sub}$: $l(v) = l(f(v))$ for the label function $l : V \cup V_P \to \mathbb{R}$ (see [79]).

The main idea of our GAS algorithm is to solve SI using a vertex-centric message passing scheme. Conceptually, a message represents a partially matched graph pattern, i.e., we associate graph vertices with pattern vertices (assuming they have matching labels). We denote a message as *partially matched* if we have not yet associated all pattern vertices with graph vertices. In contrast, a *fully matched* message associates one unique graph vertex with *each* pattern vertex in the message. If a message is fully matched, we have found a matching subgraph. Eventually, we are guaranteed to retrieve matching subgraphs (if they exist) by repeatedly attempting to match a pattern vertex in the message and sending the message to all neighbors,

(a)


Visited List
Pattern Message
(I) v_2 matches pattern vertex
$[v_2]$
v_1
…
$[v_1]$
v_4 v_3 v_2 v_1
(II) v_3 matches pattern vertex
Reset visited list
$[v_3]$
v_2
(VI) v_1 matches last pattern vertex
$[v_4, v_3, v_2]$
v_3
v_4
$[v_4]$
$[v_4, v_3]$
(V) v_2 has already matched pattern vertex
(III) v_4 matches pattern vertex
(IV) v_3 has already matched pattern vertex

(b)

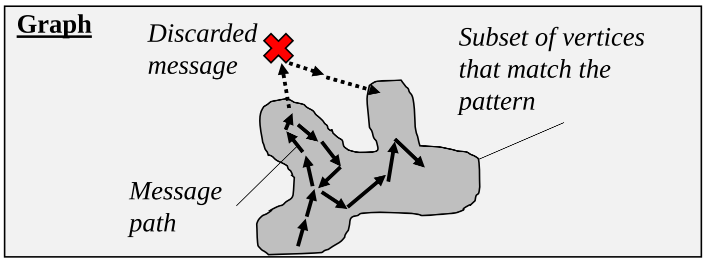


Figure 3.7: (a) Subgraph isomorphism example – perspective of vertex v_2. We show the path of a single message. (b) The algorithm considers only messages that travel *within the matching subgraph*.

because this creates a message for each possible path in the graph and one of these paths leads to a fully matched message.

However, the number of paths and therefore the number of messages grows exponentially – considering that a message is gathered by *all* neighbors in each iteration. To limit the number of concurrent messages traversing the graph, we focus on these messages that traverse only graph vertices that can match exactly one pattern vertex and discard all other messages. In Figure 3.7(a), we exemplify the path of a single message (I) to (VI) that subsequently traverses graph vertices attempting to match a pattern vertex in the message. Vertices that have already matched a pattern vertex (IV)-(V) simply forward the message to their neighbors. The example shows that the message traverses some vertices twice, if they have already matched a pattern vertex. In Figure 3.7(b), we sketch a graph containing a subgraph that matches the pattern. Also, we give a specific message traversing the graph to find the matching subgraph. A message that has traversed all vertices in the subgraph would successfully return the found subgraph. Messages that leave the subgraph of vertices matching a pattern vertex are discarded immediately (in contrast to the naive approach given above). This reduces the number of concurrent messages traversing inefficient paths. Furthermore, we break infinite message forwarding cycles if there is no progress in terms of newly matched pattern vertices. To this end, we maintain a list of vertices, denoted as $visited$, that have already processed this message (see line 14).

In Algorithm 4, we specify the vertex data and the gather, apply, and scatter phases of the vertex function. The vertex data consists of the graph pattern $P = (V_P, E_P)$ and the set of messages ω to be processed by graph vertex u. We define a message as a tuple $(\mathcal{X}, visited)$ where $\mathcal{X} \subset V \times V_p$ specifies the set of current matches between graph vertices and pattern vertices. In the gather phase (line 4), vertex u collects the messages stored at neighboring vertex v and performs a union operation over all gathered messages from neighboring vertices (line 6). In the apply phase, vertex u processes all gathered messages by matching pattern vertices if possible (lines 8-23). If a graph vertex u has already matched a pattern vertex, it forwards the message to neighbors by adding them to the message set $D_u.\omega$ (see line 15). Note that the function $match(\ldots)$ (line 18) returns true, if for all edges in the pattern there are corresponding edges in the graph (i.e., $\forall (u_1, u'_1), (u_2, u'_2) \in \mathcal{X} : (u'_1, u'_2) \in E_p \implies (u_1, u_2) \in E$) and the labels of associated graph and pattern vertices match. Graph vertices that can not match a pattern vertex (or that detect a message loop) drop the message. In the scatter phase (line 24-26), vertex u schedules neighboring vertex v, if u has partially matched messages to be processed by vertex v.

Algorithm 4 Subgraph isomorphism algorithm.

```
1: Vertex data $D_u$:
2: $P = (V_P, E_P)$
3: $\omega$ // The set of partially matched pattern messages
4: function GATHER($u, v$)
5:     return $D_v.\omega$
6: function SUM($S_1, S_2$)
7:     return $S_1 \cup S_2$
8: function APPLY($D_u, S$)
9:     if First execution of apply then
10:        $D_u.\omega \leftarrow \{(\{(u, v_P)\}, [u]) | u \text{ matches } v_P \in V_P\}$ // Create message for each matching of graph vertex $u$ and a pattern vertex
11:    else
12:        $D_u.\omega = \emptyset$
13:        for all $(X, visited) \in S$ do
14:            if $\exists (u, v_P) \in X \wedge u \notin visited$ then
15:                $D_u.\omega \leftarrow D_u.\omega \cup \{(X, visited + [u])\}$
16:            else if not $\exists (u, v_P) \in X$ then
17:                for all $c \in neighbors(visited[-1])$ do
18:                    if $match(X \cup \{(u, c)\})$ then
19:                        $X \leftarrow X \cup \{(u, c)\}$
20:                        $visited \leftarrow [u]$
21:                        $D_u.\omega \leftarrow D_u.\omega \cup \{(X, visited)\}$
22:                        if $|X| == |V_P|$ then
23:                            $X$ is a correct matching
24: function SCATTER($D_u, D_v$)
25:    if $D_u.\omega \neq \emptyset$ then
26:        $Activate(\{u, v\})$
```

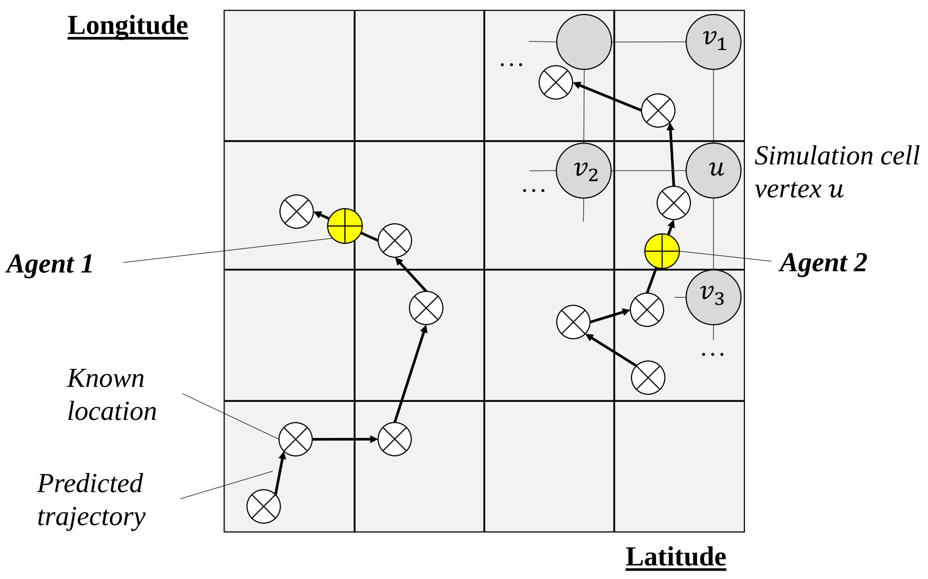


Figure 3.8: Agent-based social simulations via cellular automaton.

3.3.2 Cellular Automaton

The powerful and well-established model of *cellular automaton* is able to express various problems in several research areas [57] such as simulations of complex systems. A cellular automaton models the problem space as a grid of cells, each with a finite number of states. A cell iteratively calculates its own state based on the states of neighboring cells. Cellular automata fit well into the vertex-centric programming model of recent graph processing systems because of the neighborhood relation of the cells and the local nature of iterative state computation based on neighboring cells.

We implemented an agent-based variant for simulating real-world movements of people in Beijing[2]. [125] In Figure 3.8, we give an example of agent-based simulations. Given are two moving agents in an area described as a two-dimensional euclidean space using latitude and longitude coordinates. For each agent, a series of known locations at certain points in time (i.e., movement trajectory L) is given, e.g., extracted from GPS signals of user smartphones [107]. The area is divided into a grid of cells, whereby each cell is assigned to a graph vertex. A graph vertex is connected to all vertices of neighboring cells and is responsible for all agents within its assigned cell. In the example, vertex u is connected to vertices v_1, v_2, v_3 and simulates movement of agent 2. More precisely, the vertex data consists of a set of agents A,

[2]http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

each agent maintains its current location *loc* and the set of known locations L, both defined by (latitude,longitude,time) coordinates (see Algorithm 5).

We use a simple linear interpolation between the known locations of the agent to estimate its location for each intermediate time step. In each iteration, we move forward Δt time steps along the estimated movement direction of the agent. Each vertex u only processes agents whose current locations $loc = (x, y, t)$ fall into vertex u's responsibility area, i.e., $x_{min} \leq x < x_{max}$ and $y_{min} \leq y < y_{max}$. As soon as the agent leaves the responsibility area, the agent is sent to the neighboring vertex v_1.

In the gather phase, vertex u collects agents from a neighboring vertex v that are in u's responsibility area (line 5). The sum function performs a simple union operation over all gathered agents. In the apply phase, vertex u adds all gathered agents to its local agents A (line 9) and removes all agents that have left the responsibility area of vertex u (line 10). Then the location of all agents is updated by incrementing time by Δt (lines 11-14) using linear interpolation to estimate the agent's current location *loc*. In the scatter phase, vertex u activates itself if he possesses local agents. Finally, vertex u activates neighboring vertices if u has local agents in A that have left v's responsibility area (line 18).

Algorithm 5 Cellular automaton algorithm of vertex u.

```
1: **Vertex data** $D_u$:
2: $A$ // The set of local agents
3: $Cell$ // Responsibility area
4: **function** GATHER($u, v$)
5:     **return** $\{((x,y,t),L) \in D_v.A \mid (x,y) \in Cell\}$
6: **function** SUM($S_1, S_2$)
7:     **return** $S_1 \cup S_2$
8: **function** APPLY($u, S$)
9:     $D_u.A \leftarrow D_u.A \cup S$
10:    $D_u.A \leftarrow D_u.A \setminus \{((x,y,t),L) \in D_u.A \mid (x,y) \notin Cell\}$ // Remove the agents that have left the cell
11:    **for all** $(loc, L) \in D_u.A$ **do**
12:        $last \leftarrow previous\ location\ in\ L$
13:        $next \leftarrow next\ location\ in\ L$
14:        $loc \leftarrow interpolateLocation(last, next, \Delta t)$
15: **function** SCATTER($u, v$)
16:    **if** $D_u.A \neq \emptyset$ **then**
17:        $Activate(u)$
18:    **if** $|\{((x,y,t),L) \in D_u.A \mid (x,y) \notin Cell\}| > 0$ **then**
19:        $Activate(v)$
```

Name	$\|V\|$	$\|E\|$
Twitter	81,308	1,768,149
GoogleWeb	875,713	5,105,039
Web	41,291,594	1,150,725,436
TwitterLarge	41,652,230	1,468,365,182

Table 3.3: Real-world graphs for evaluations.

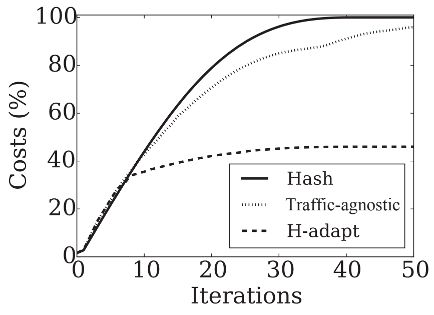
3.4 Evaluations

In the following, we present evaluations for GrapH on two computing clusters for three different algorithms—PageRank, subgraph isomorphism, and cellular automaton—on several real-world graphs given in Table 3.3 with up to 1.4 billion edges[3]. We compare H-load and H-adapt against both existing static vertex-cut approaches, i.e., hashing of edges (Hash) and PowerGraph (PG) [34, 35], as well as traffic- and network-agnostic algorithmic variants.

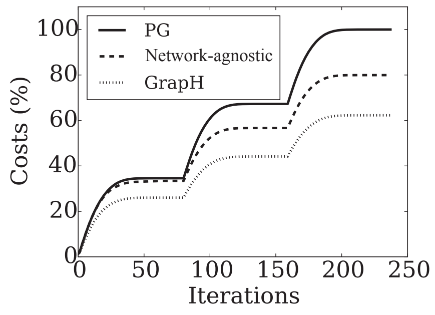
Evaluation Setup: We have implemented GrapH in the Java programming language ($>$ $10,000$ lines of code). GrapH consists of a master worker and multiple client workers performing graph analytics. The master receives a sequence of graph processing queries $q_1, q_2, q_3, \ldots$ consisting of user specified GAS algorithms. The graph system executes these queries in a synchronized manner, i.e., one after another. All workers communicate with each other directly via TCP/IP.

We used four computing infrastructures with homogeneous and heterogeneous network costs. (i) The homogeneous computing cluster (ComputeC) consists of 12 machines, each with 8 cores (3.0GHz) and 32GB RAM, interconnected with 1 Gbps Ethernet. (ii) Furthermore, we performed evaluations on an in-house shared memory machine (ComputeM) with 32 cores (2.3GHz) and 280GB RAM. Unless stated otherwise, we modeled network costs in ComputeC and ComputeM as follows, $\forall m, m' \in M : T_{m,m} = 0 \wedge m \neq m' \rightarrow T_{m,m'} = 1$. (iii) The heterogeneous computing cluster (CloudC) is deployed in the Amazon cloud using 8 geographically distributed EC2 instances (1 virtual CPU with 3.3 GHz and 1 GB RAM) that are distributed across two regions, US East (Virginia) and EU (Frankfurt), and four different availability zones. As network costs between these instances, we used the monetary costs charged by Amazon (see Table 3.2). If not mentioned otherwise, the experiments are performed on CloudC. (iv) The computing infrastructure (CloudM) consists of two powerful EC2 multi-core machines in the same availability zone (m4.16xlarge) with 64 virtual CPUs and 256GB RAM. Scaling up computation on a small number of powerful machines becomes more and more common in

[3] http://konect.uni-koblenz.de/networks/twitter, http://snap.stanford.edu/data

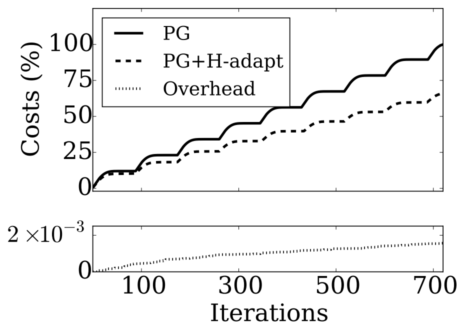


(a) PR on *Twitter*.

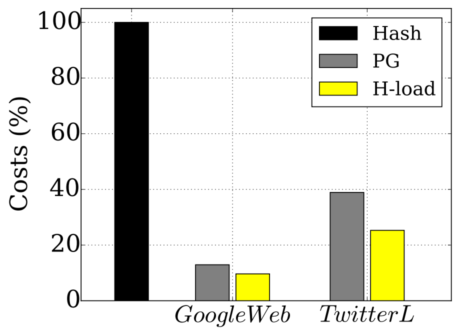


(b) PR on *GoogleWeb*.

Figure 3.9: (a) Traffic- and (b) network-awareness reduce communication costs.



(a) PR on *GoogleWeb*.



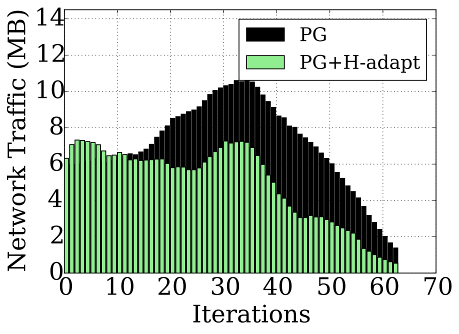
(b) Pre-partitionings.

Figure 3.10: (a) H-adapt reduces communication costs with low overhead. (b) Pre-partitioning with H-load reduces communication costs.

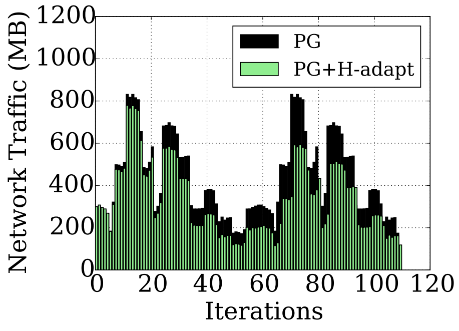
recent years [155]. GrapH supports scaling up by simply creating multiple workers running on the same machine (see Chapter 2). This leads to heterogenous network costs: workers placed on the same multi-core machine have relatively low network costs while remote workers have relatively high network costs.

However, as *distributed* graph processing mainly focuses on scaling out, our default setting is ComputeC or CloudC with one worker per machine. The network costs matrix T is calculated based on packet round-trip times and presented in Figure 3.13a. Note that for large-scale deployments, even with hundreds of thousands of servers, there are scalable methods with low overhead (i.e., $< 1\%$ CPU overhead, < 45 MB RAM footprint, and $\approx 50KB/s$ probing traffic) to determine the estimated round-trip time between any two servers in a dynamic environment [37].

Communication Costs: The main idea in this chapter is to consider network- and traffic-heterogeneity while constantly repartitioning the graph during computation. In the following,

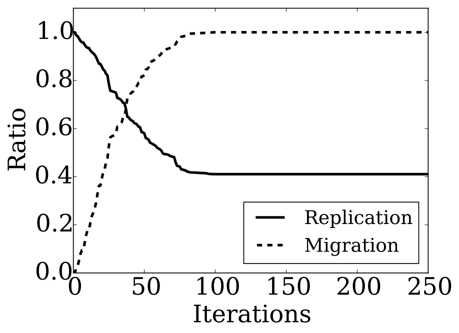


(a) CA on grid (10^5 vertices)

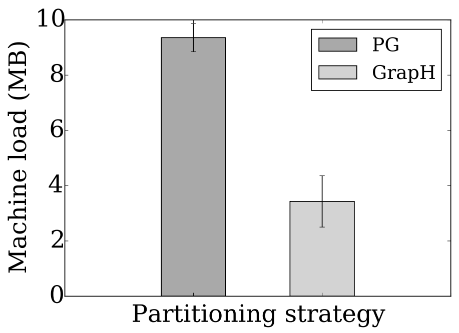


(b) SI on *Twitter*.

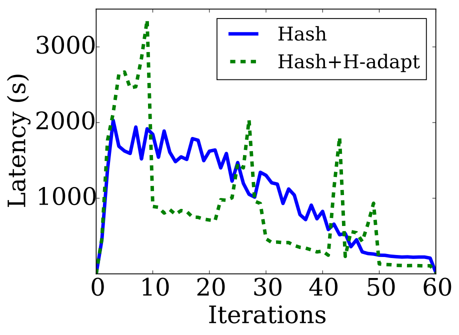
Figure 3.11: (a)-(b) H-adapt reduces network traffic.



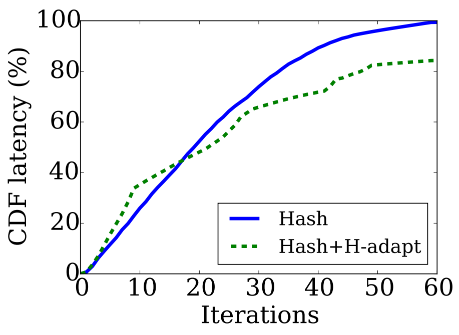
(a) PR on *Web*.



(b) PR on *GoogleWeb*.



(c) PR on *TwitterLarge*.



(d) PR on *TwitterLarge*.

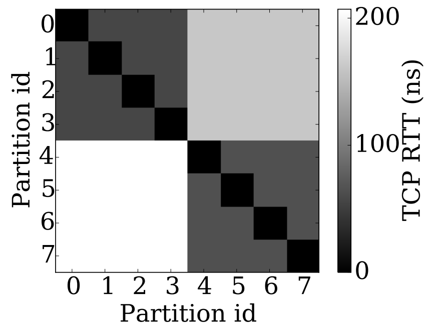
Figure 3.12: (a) H-adapt reduces replication degree. (b) GrapH reduces total workload. (c)-(d) H-adapt reduces latency.

we evaluate the effect of traffic- and network-awareness on total communication costs as defined in Equation 3.2: *total traffic sent via each network link, weighted by the costs of the network link*. In our first experiment (see Fig 3.9a), we compared three different partitioning methods: (i) hashing of edges to partitions without dynamic migration, (ii) our dynamic migration strategy assuming *homogeneous* vertex traffic on the hash partitioned graph (Traffic-agnostic), and (iii) our dynamic migration strategy on the hash partitioned graph considering heterogeneous vertex traffic (H-adapt). We accumulated communication costs over 50 iterations of PR on *Twitter*. The results indicate that considering heterogeneous vertex traffic greatly improves total communication costs by up to 50% compared to Traffic-agnostic. The reason is that H-adapt implicitly prioritizes high-traffic vertices, that are the major sources of total traffic (see Section 3.1, Figure 3.1).

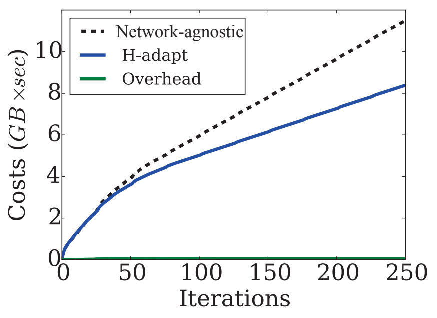
How does network-awareness improve total communication costs? In Figure 3.9b, we compare three different partitioning methods: (i) PowerGraph partitioning without dynamic migration (PG), (ii) our strategy H-adapt on the H-load partitioned graph, while both strategies assume homogeneous network costs (Network-agnostic), and (iii) our dynamic migration strategy H-adapt on the H-load partitioned graph considering heterogeneous network costs (GrapH). We performed 250 iterations of PageRank on *GoogleWeb* (and restarted computation after termination of one PageRank instance). It can be seen, that Network-agnostic already reduces communication costs by 20% compared to PG. However, taking a heterogeneous network into account reduces total costs by additional 20%. Our experiments therefore indicate, that the awareness of traffic and network heterogeneity improves partitioning quality significantly.

What is the overhead of migrating edges in terms of additional communication costs? In Figure 3.10a, we show total communication costs and migration overhead of 700 iterations of PageRank on *GoogleWeb*. While communication costs decreased by 33% compared to PG, the costs for migration itself (investment costs, see Equation 3.4) are very low compared to the saved costs. The total costs of migration approaches only 2×10^{-3} percent of the overall communication costs due to the following three reasons: (i) optimizing the placement of a vertex replica *once* saves replica communication *in each iteration*, (ii) optimizing the placement of a master replica can reduce replica communication over *multiple* high-costs network links, and (iii) traffic awareness allows for more focused optimizations, i.e., H-adapt automatically targets optimal placement of the high-traffic vertices dominating overall network usage (see Figure 3.1a-c).

Finally, we evaluated communication costs improvements of H-load compared with PowerGraph and Hashing for two different graphs: *GoogleWeb*, and *TwitterLarge* (Fig 3.10b). We assume homogeneous vertex traffic and heterogeneous network costs. H-load greatly reduces total graph processing costs by 70-90% compared to Hash and by 25-38% compared to PowerGraph partitioning even if no vertex traffic statistics are known (e.g. from previous executions).



(a) Round-trip time in CloudM.



(b) PR on *Web*.

Figure 3.13: Heterogeneity in single data center, scale-up scenario.

Considering Network Heterogeneity when Scaling Up: Many real-world computing clusters such as ComputeC and CloudM consist of a number of multi-core workers connected via Ethernet. These types of infrastructures require both operations, scaling up and scaling out. Similar to other graph systems (e.g. [138]), we support both scaling operations by assigning multiple workers to the same machine (each running on a different core). This introduces another source of network heterogeneity: loop-back versus plain network communication.

In Figure 3.13a, we plot the round-trip time of 8 workers that are evenly divided among two machines in CloudM. Hence, we used four workers per machine for the following experiment. Clearly, the round-trip time differs by an order of magnitude: workers on the same machine can communicate with very low delay of only 0.06 milliseconds – compared to workers on different machines with up to 0.2 milliseconds. We set the network costs matrix $T_{m,m'}$ for workers m and m' according to these round-trip time values.

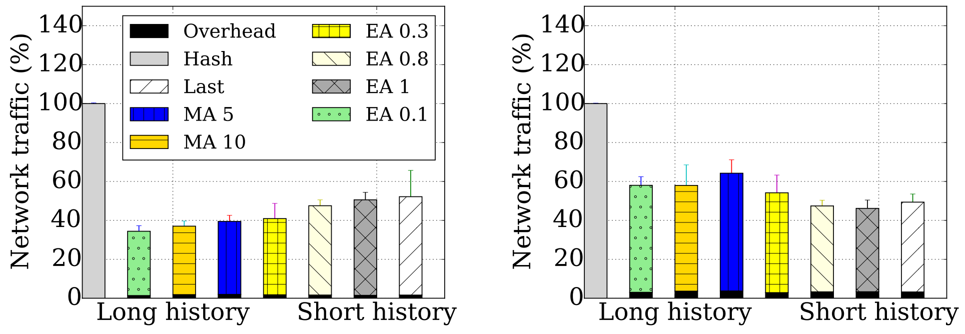
In Figure 3.13b, we exploit these heterogeneities in CloudM in order to show that single data center applications can benefit from network awareness as well. We performed 250 iterations of PageRank on the web graph [11] and took the round-trip times between the workers as network costs T as described previously. We measured the total accumulated costs (i.e., the product of traffic sent over a link and network link costs) for H-adapt when switching network awareness on and off. The result shows that being network aware reduces graph processing costs by 29% while introducing neglectable costs overhead.

Network Traffic: Network costs may be unknown or relatively homogeneous. In this case, H-adapt automatically reduces the total amount of worker communication, i.e., *network traffic* (compare Section 3.2.2). In Figure 3.11a and b, we show the extent of this improvement for SI and CA (similar results for PR are omitted). We show the current network traffic (averaged over a sliding window of 10 iterations) for these algorithms on ComputeC. The SI algorithm searched for a series of ten randomly chosen patterns of sizes 3-6, such as triangles and circles.Our migration strategy reduces total network traffic by up to 50% compared to PowerGraph partitioning.

Replication Degree: A standard metric to measure the partitioning quality of vertex-cut algorithms is the replication degree, i.e., the number of vertex replicas in the system. In Figure 3.12a, we performed 250 iterations of PageRank on the web graph [11] using the CloudM infrastructure. We measured the replication degree in each iteration divided by the replication degree of a hash partitioned graph, as well as the current migration overhead divided by the total migration overhead (CDF). During the first 70 iterations replication degree is reduced by 60%, followed by saturation. Hence, H-adapt reduces replication degree as a side effect when optimizing for total communication costs.

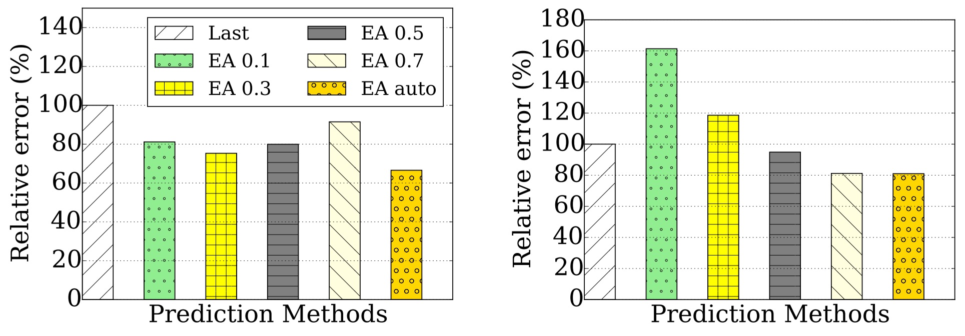
Latency: The partitioning problem is computationally hard and solving it during execution can increase overall processing latency, despite the benefits of reduced communication. To evaluate how our dynamic repartitioning method H-adapt influences latency of graph processing, we measured the latency per iteration of a single execution of PageRank on our shared memory machine. We used *TwitterLarge* with more than 1.4 billion edges. In Figure 3.12c, we plot the latency per iteration for the Hash pre-partitioned graph (Hash) and for the Hash repartitioned graph using H-adapt (8 partitions). The figure highlights the trade-off between migration latency overhead and gain: there are four instances where H-adapt temporarily invests more time (approximately at iterations $3, 25, 41, 48$). However, investing time to compute and implement an improved partitioning results in large reductions of latency up to 60%. In Figure 3.12d, we show the cumulated distribution function for latency to compare end-to-end latency of both methods. Surprisingly, the end-to-end latency for a single execution of PageRank improves by 10% using our dynamic repartitioning method H-adapt. Thus, compared to the predecessor method H-move [86], H-adapt reduces end-to-end latency by approximately 20% due to the novel methods of constant back-off and lock-free migration. Although included into the total runtime, the time spent on partitioning is 12%, averaged over all iterations. Note that we have executed only one instance of the PageRank algorithm, subsequent executions and long-running graph algorithms would benefit even more from the improved partitioning.

We also tested the impact of network-awareness on graph processing latency for the PageRank algorithm in CloudM. However, we observed a non-significant reduction of graph processing latency by less than 1 percent for H-adapt compared to its network-agnostic variant. We attribute this to the fact that our cost function minimizes the *weighted summed vertex traffic over all network links* (see Equation 3.9) but latency mainly depends on the bottleneck link with maximal latency because of the straggler problem [40]. In other words, at least one high-cost link between two workers residing on different machines experienced the same amount of traffic (although the *summed* high-cost network link traffic was reduced). However, tuning the cost function to minimize latency instead of communication costs is beyond the scope of this thesis.



(a) PR on *Twitter*. (b) SI on *Twitter*.

Figure 3.14: (a)-(b) Prediction methods influence migration efficiency.



(a) SI on *Twitter*. (b) CA on grid (2,500 vertices).

Figure 3.15: (a)-(b) Adaptive-α outperforms other prediction methods.

Load Balancing: GraphH balances the *summed vertex traffic over all vertices on a partition* (see Equation 3.3). In Figure 3.12b, we show the average worker workload after one PR execution (78 iterations) for PG and GraphH as well as the deviation of the workers from this average workload. GraphH leads to a slightly higher workload imbalance because we balance for (more volatile) vertex traffic, while PG balances the number of edges. However, because of the reduced overall communication overhead, GraphH's maximally loaded worker still has less workload than the least loaded worker in PG. Thus, GraphH reduces total workload by more than 60%.

Prediction Methods: Choosing the right prediction method for future vertex traffic is important for overall performance of our system. For instance, overestimating vertex traffic of vertex u leads to biased decisions towards migrating edges incident to vertex u. To learn about prediction accuracy, we compared three methods (see Section 3.2): last value (Last), moving average (MA) with window sizes 5 and 10, and exponential averaging (EA) with decay parameter $\alpha = 0.1, 0.3, 0.8, 1.0$ in Figure 3.14a-b. We evaluated the reduction of total network traffic for PR and SI, compared to Hash. The position of the bars from left to right reflects the size of the considered history for prediction in descending order. For example, since *Last* considers only the last value, we plotted it on the right. Clearly, all prediction methods meet the goal of reducing overall network traffic. However, none of these methods leads to consistently better results for all algorithms, because of the different stability of vertex traffic patterns. For example in PR, considering a longer history shows better results, because vertex traffic patterns remain stable over time. Nevertheless, considering a large history in SI actually harms performance, because the subsequent short-lived queries lead to diverse vertex traffic patterns.

We have seen that there is no single prediction method that is optimal for all algorithms. Thus, determining the best parameter α is a crucial task for effective vertex traffic prediction. In the following, we show that our method for automatically selecting α individually for each vertex consistently outperforms all other static (and global) parameter choices. In Figure 3.15a-b, we compared the *relative prediction error* w.r.t. decay parameter α. We define the relative prediction error as the ratio of the absolute prediction errors of the tested method and the benchmark method *Last*. The absolute error of a prediction method is determined by predicting vertex traffic for the next w iterations and accumulating the difference between predicted and observed vertex traffic value, averaged over 5 randomly selected vertices. We performed evaluations for SI on *Twitter* and CA on a grid with 2500 vertices (and window sizes w of $10, 5$, and 3 respectively). On the right, we plotted the relative error for our method Adaptive-α. Our experiments lead to two interesting conclusions. First, the extremely simple method *Last* results in relatively robust vertex traffic predictions with high accuracy and low computational overhead. Second, Adaptive-α consistently outperformed all other prediction methods in terms of prediction accuracy while introducing little computational overhead (see Equation 3.7).

3.4.1 Summary of Evaluation Results

In summary, the experimental results validate the main thesis of this chapter: considering heterogeneous vertex traffic and network costs in graph partitioning reduces communication costs using different realistic computational infrastructures – when executing practical graph algorithms such as PageRank, subgraph isomorphism, and cellular automaton on large real-world graphs.

In particular, the results validate that the distribution of vertex traffic resembles a pareto distribution where a small percentage of vertices are responsible for a large percentage of vertex traffic. By considering the skewed vertex traffic for dynamic graph partitioning with our strategy H-adapt, we could reduce communication costs by up to 50% compared to traffic-agnostic partitioners on the Twitter graph (PageRank). On top of that, we experimentally validated that considering heterogeneous networks reduces communication costs by up to 20% compared to network-agnostic partitioners on the GoogleWeb graph. The remaining experiments show robustness of those results for different computational environments, graphs with up to 1.4 billion edges, and graph algorithms — and validate efficiency of our dynamic vertex traffic prediction mechanisms.

3.5 Related Work

Several existing research efforts in the area of distributed graph processing are relevant to our work. PowerGraph [34] suggests the vertex-centric GAS API. We benchmarked our system against their greedy streaming heuristic *Coordinated* for static vertex-cut partitioning. Petroni et al. (*HDRF* [101]) consider the vertex degree in order to find a minimum vertex-cut in the streaming setting arguing that optimal placement of low-degree vertices should be preferred. ADWISE [85] uses a window over the edge stream as a basis for more informed partitioning decisions. PowerLyra [18] extends PowerGraph with hybrid-cuts: cutting vertices with high degree and edges of low-degree vertices decreasing expensive replica communication overhead. These strategies minimize the replication degree, but ignore diverse and dynamic vertex traffic.

On the other hand, the graph systems Mizan [66] and GPS [115] propose adaptive edge-cut partitioning using vertex migration. Mizan considers the traffic sent via each edge to balance workload at runtime. Vaquero et al. [131] apply edge-cut to changing graphs using a decentralized algorithm for iterative vertex migration to avoid costly re-execution of static partitioning algorithms. Shang et al. [120] nicely identified and categorized three types of vertex activation patterns for graph processing workload: always-active-, traversal-, and multi-phase-style. They exploit these workload patterns to dynamically adjust the partitioning during computation. Yang et al. [144] (*Sedge*) improve localization of processing small-sized queries by introducing a two-level complimentary partitioning scheme using vertex replication. While these edge-cut systems adapt to changing graphs or traffic behaviors, they do not consider network topology and migration costs. Furthermore, these approaches focus on edge-cut and optimal edge-cuts can not be transformed into close-to-optimal vertex-cuts for graphs with high-degree vertices [30].

Surfer [19] tailors graph processing to the cloud by considering bandwidth unevenness to map graph partitions with a high number of inter-partition links to workers connected via high-bandwidth networks. However, they assume homogeneous traffic on each edge. GraphIVE [68] strives for a minimal *unbalanced* k-way vertex-cut for workers with heterogeneous computation and communication capabilities, in order to put more work to more powerful workers — searching for the optimal number of edges for each worker. This approach is orthogonal to heterogeneity-aware partitioning algorithms. Xu et al. [142] consider network and vertex weights to find a static minimal costs edge-cut. They do not consider adaptive vertex-cut, and vertex weights reflect only the number of executions, but not the real vertex traffic. Zheng et al. [153] propose ARGO, an architecture-aware edge-cut graph repartitioning method focusing on RDMA-enabled networks that also considers the amount of communication going over each edge and heterogeneous network costs. Similarly, GrapH is also applicable to the scenarios described by ARGO while focusing on vertex-cut partitioning and the GAS execution model.

General data processing in the geo-distributed setting is addressed by Pu et al. [104] and Jayalath et al. [51]. They argue that aggregating geographical distributed data into a single data center can significantly hurt overall data processing performance for MapReduce-like computations. LaCurts et al. [70] point out that considering network heterogeneity for an optimal task placement, improves overall end-to-end data analytics performance, even in a single data center. Therefore, they place communicating tasks in such a way that most communication flows over fast network links. However, task placement is orthogonal to graph partitioning and could be used on top of our system.

Finally, significant research efforts addressed the problem of parallelizing subgraph isomorphism (e.g. [79, 147]) and cellular automata (e.g. [41, 125]). However, we have not found any algorithm in the GAS programming interface despite the suitability of distributed graph processing systems for these kind of problems.

3.6 Chapter Summary

Modern graph processing systems use vertex-cut partitioning due to its superiority of partitioning real-world graphs. Existing partitioning methods minimize the replication degree, which is expected to be the dominant factor for communication costs. However, the underlying assumptions of uniform vertex traffic and network costs do not hold for many real-world applications. To this end, we propose GrapH, a graph processing system taking dynamic vertex traffic and diverse network costs into account, to adaptively minimize communication costs. Our evaluations show, that GrapH outperforms state-of-the-art by up to 60% with respect to communication costs, while improving end-to-end latency of graph computation by more than 10%.

4

Adaptive Window-based Streaming Partitioning

In Chapter 3, we propose two partitioning algorithms: a static algorithm to partition the graph as it is loaded into the graph system, and a dynamic algorithm that repartitions the graph at runtime. In the previous chapter, we focus on the latter algorithm, i.e., dynamic repartitioning. The focus of this chapter is the static, initial partitioning of the graph. The initial partitioning problem is NP-hard [30]. Therefore, the standard choice for partitioning billion-scale graphs is to use very fast heuristics at the costs of partitioning quality [123]. Consider the following observation: graph processing latency strongly correlates with *partitioning quality* (cf. *replication degree* in Chapter 2) which can be improved by investing more latency into partitioning. In this chapter, we investigate the trade-off between partitioning latency and graph processing latency. Hereby, the main question is whether minimizing partitioning latency also leads to minimal total latency (consisting of graph partitioning *and* graph processing). Neither extreme is very attractive: investing a lot of latency in partitioning to reduce graph processing latency; or investing minimal latency in partitioning which increases graph processing latency. The question is whether there is a sweet spot between partitioning latency and graph processing latency because graph practitioners are interested in minimizing the sum of both, i.e., the perceived total latency of graph partitioning and processing. This chapter is based on our published work [85].

In literature, there are two basic approaches to practically address the vertex-cut partitioning problem. (i) *Single-edge* streaming algorithms perform partitioning decisions on one edge at a time, minimizing the partitioning latency. (ii) *All-edge* algorithms load the complete graph into memory and employ global placement heuristics to optimize the partitioning quality. The existing algorithms follow either of the methods: Figure 4.1 illustrates the landscape of state-of-the-art vertex-cut partitioning algorithms. The x axis describes the partitioning latency, i.e., the total amount of time invested into the graph partitioning. The y axis describes the partitioning

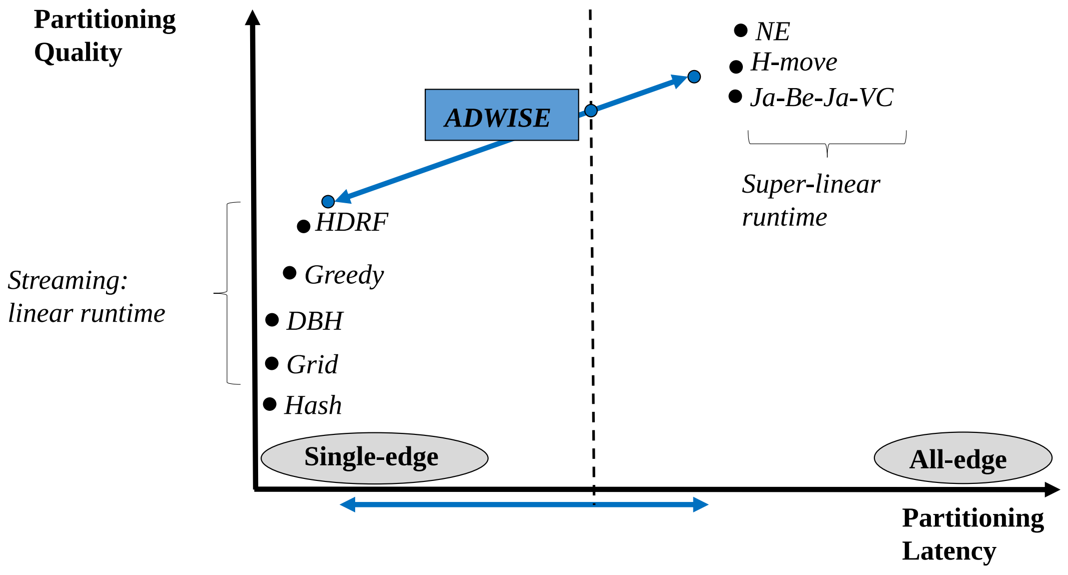


Figure 4.1: Research gap – adaptive window-based streaming vertex-cut partitioning. Partitioning strategies: Hash [34], Grid [50], DBH [140], Greedy [34], HDRF [101], NE [150], H-move [87], and Ja-Be-Ja-VC [106]

quality which impacts the efficiency of graph processing, i.e., high partitioning quality leads to reduced overhead and faster graph processing. In general, increasing the partitioning latency allows for better partitioning quality. Single-edge streaming algorithms have minimal partitioning latency due to their linear algorithmic complexity, while the partitioning latency of all-edge algorithms is high but leads to better partitioning quality. Modern graph processing systems use streaming partitioning when loading massive graphs due to their superior scalability and minimal runtime complexity [18, 34].

In this chapter, we investigate whether it is always optimal to invest minimal partitioning latency as done by the established streaming partitioning algorithms. Clearly, there is a trade-off between partitioning latency and partitioning quality—and thus, graph processing latency. Our hypothesis is that for complex and long-running graph algorithms that run on large graphs, investing more than minimal time into graph partitioning leads to reduced total latency of graph partitioning *and* graph processing: To minimize the total latency, this trade-off must become controllable, i.e., the partitioning algorithm should be able to control the time invested into optimizing the partitioning quality. However, none of the current streaming partitioning algorithms allows for that.

To close this gap, we propose to consider a *window of edges* from the graph stream for making the partitioning decisions—instead of either a single edge or all edges. The basic idea is that considering more edges at a time enables improvements on the partitioning quality, but imposes a larger partitioning latency. While this is an intuitive idea, it poses a number of interesting research questions that need to be addressed: (1) How many edges should be taken into account when making a partitioning decision, i.e., how large should the window be? (2) Which of the edges should be assigned to which partition, i.e., how to design the *scoring* function that assigns the highest score to the best edge placement? (3) How to avoid unnecessary computations, i.e., how to limit score calculations to the high-potential edges in the window?

To address these questions, we developed ADWISE[1], a novel *window-based* streaming partitioning approach. Our main contributions are as follows. (i) We employ methods to *automatically adapt* the window size at runtime in order to control the trade-off between partitioning latency and quality according to a partitioning latency preference. (ii) We propose a novel *scoring function* tailored to window-based partitioning. It considers multiple objectives – including diversity and skewness of the graph edges – to quantify partitioning decisions pertaining to the edges in the window. (iii) We employ a *lazy traversal* score calculation method that limits score (re-)calculations to a subset of most promising window edges in order to reduce partitioning latency on a given window. (iv) We introduce the *spotlight partitioning* optimization for parallel graph partitioning on multiple ADWISE instances. Spotlight partitioning reduces the spread of the partitioning instances such that each instance works on a disjoint set of partitions. This tremendously improves partitioning quality and can be applied on top of any existing streaming graph partitioning algorithm. (v) Our evaluations show that for large real-world graph processing problems, it is beneficial to invest more latency into partitioning in order to minimize the total latency. Using ADWISE, the total latency could be reduced by up to $23-47\%$ compared to traditional single-edge streaming partitioning algorithms.

The rest of the chapter is structured as follows. In Section 4.1, we state the problem formulation and analyze challenges of window-based streaming partitioning. In Section 4.2, we describe our algorithm ADWISE in detail. We evaluate our methods in Section 4.3, present related work in Section 4.4, and summarize in Section 4.5.

4.1 Problem Statement and Analysis

In this section, we introduce the graph partitioning problem, define the window-based streaming partitioning model proposed by ADWISE, and discuss the research questions in window-based streaming partitioning that need to be solved.

[1] `https://github.com/GraphPartitioning/WISE`

4.1.1 The Vertex-cut Graph Partitioning Problem

Many graph processing systems rely on vertex-cut partitioning [34, 35, 86]. In the following, we quickly recap the vertex-cut partitioning problem described in Chapter 2 to introduce the specific notation used in this chapter. Let graph $G = (V,E)$ consist of a set of vertices $V = \{v_1,...,v_n\}$ and edges $E \subseteq V \times V$. The goal is to divide the graph into k partitions with identifiers $P = \{1,...,k\}$. Vertex-cut graph partitioning can be achieved by *assigning edges to partitions*, which leads to cut vertices spanning multiple partitions. Suppose, a graph is cut through vertex u into two partitions p_1 and p_2. Vertex u is replicated on both partitions, because both contain edges incident to vertex u. We denote the set of partitions where vertex u is replicated as *replica set* R_u. During graph processing, replicas of u communicate to provide remote vertex data access to vertices residing on different partitions. By minimizing the number of replicas (denoted as *replication degree*), the amount of communication during graph computation is minimized as well [34]. Therefore, the goal of vertex-cut partitioning is to minimize the replication degree (see Equation 4.1), such that the partitions are balanced in the number of edges (see Equation 4.2) to ensure workload balancing during graph processing (see [101]). The maximal deviation between the number of edges assigned to any pair of partitions is controlled via the parameter $\tau \in [0,1]$. In Table 4.1, we give an overview about the notation used in this chapter, in the order of occurrence.

$$minimize \frac{1}{|V|} \sum_{v \in V} |R_v|, \tag{4.1}$$

$$s.t. \forall i,j \in P, |P_i| > |P_j| : \frac{|P_j|}{|P_i|} > \tau. \tag{4.2}$$

Note that this partitioning problem is a version of the problem in Chapter 3, but without considering dynamic and heterogeneous vertex traffic and network costs. Before executing the graph algorithm, it is difficult to obtain the heterogeneous vertex traffic for each vertex. Instead, we follow the standard approach and optimize for replication degree [34, 101]. However, the problem is still NP-hard [30].

4.1.2 Streaming Partitioning

In the following, we analyze the streaming vertex-cut partitioning method in more detail, pointing out the commonalities and shortcomings of existing algorithms.

In vertex-cut streaming partitioning, partitioning algorithms perform a single pass over the stream of graph edges and assign all edges to partitions as they arrive in the stream. More

$G = (V, E)$	Graph with set of vertices V and edges E.
$P \subset \mathbb{N}$	The set of partition ids.
$k \in \mathbb{N}$	The number of partitions, i.e., $\|P\| = k$.
$R_u \subseteq P$	Replica set of vertex u.
$P_i \subseteq E$	The set of edges assigned to partition $i \in P$.
$\tau \in [0, 1]$	Maximal imbalance between any two partitions.
$g(e, p) \in \mathbb{R}$	Score for edge e and partition p.
$w \in \mathbb{N}$	Number of edges in the window.
$W \subseteq E$	The set of edges in the window with $\|W\| = w$.
$L \in \mathbb{N}$	User-defined latency preference (milliseconds).
$S = \langle E \rangle$	The edge stream, an ordered sequence of edges
$C \subseteq W$	Set of high-score edges (*candidate set*).
$Q \subseteq W$	Set of low-score edges (*secondary set*).
$\Theta \in \mathbb{R}$	Score threshold to determine a candidate edge.
$B(p) \in \mathbb{R}$	Balancing score of partition $p \in P$.
$\lambda \in \mathbb{R}$	Balancing parameter and adaptive balancing function.
$R(e, p) \in \mathbb{R}$	Replication score for $e \in W$ and $p \in P$.
$deg(v) \in \mathbb{N}$	Degree of vertex v.
$N(u) \subseteq V$	Set of neighbors of vertex $u \in V$.
$N_i(u) \subseteq V$	Set of neighbors of vertex $u \in V$ on partition $i \in P$.
$CS(e, p) \in \mathbb{R}$	Clustering score for $e \in W$ and $p \in P$.

Table 4.1: Notation overview.

precisely, given a sequence of edges $\langle e_1,...,e_{|E|} : e_i \in E\rangle$, edge e_i is assigned to partition $p_j \in P$ considering only previous assignment information from edges $\langle e_1,...,e_{i-1}\rangle$. As each edge is accessed exactly once, the runtime complexity is linear to the number of edges.

We illustrate the streaming partitioning model at the top of Figure 4.2. The graph data is stored in a large file, a graph database, or a distributed file system. The streaming partitioning algorithm loads the data as a stream of graph edges and subsequently assigns them to partitions. Finally, these partitions are used for distributed graph processing. The streaming partitioning model consists of three building blocks. All state-of-the-art streaming algorithms fit into this model.

Definition: The *edge universe* (cf. Figure 4.2 (i)) is a container data type that stores a set of edges with the following purpose: The streaming partitioner selects an edge from the edge universe and assigns this edge to the best partition according to the scoring function (see next paragraph). Existing algorithms allow only a single edge in the edge universe. In this case, the partitioning decision answers the question *"to which partition to assign the edge?"* rather than *"which edge to assign to which partition?"*.

Definition: The *scoring function* (cf. Figure 4.2 (ii)) measures how well an edge fits to a certain partition. In this way, the scoring function quantifies the partitioning decisions—existing streaming partitioning algorithms differ only in the selection of the scoring function.

Definition: The *vertex cache* (cf. Figure 4.2 (iii)) maintains replica sets for all vertices that were assigned in any previous edge assignment. This information is used by the scoring function to determine the best edge assignment.

Shortcomings of single-edge streaming. Due to the narrowness of the edge universe, existing partitioning algorithms enforce an assignment decision for each edge before populating the edge universe with the next edge. As a consequence, edge assignment decisions are often uninformed, i.e., based on insufficient knowledge about the replica sets of incident vertices. This can lead to low partitioning quality. Figure 4.2(a) provides an example. The scoring function $g(e_1, p_j)$ returns the number of times a vertex incident to edge e_1 is already replicated on partition p_j (see [34]). Unfortunately, the vertex cache does not contain any information about the replica set of a vertex incident to edge e_1. Therefore, the score is zero for all partitions and the algorithm assigns edge e_1 to any partition (here: p_2) – an uninformed assignment decision. Next, the algorithm loads edge e_2 into the edge universe and assigns it to partition p_2 as selected by the scoring function. The assignment of both edges e_1 and e_2 leads to three new replicas (black, blue, and green vertex) on partition p_2.

Definition: An *uninformed edge assignment* of edge $e = (u, v)$ is an edge assignment where both vertices u and v are not replicated on any of the partitions $p_i \in P$ during the execution of the partitioning algorithm.

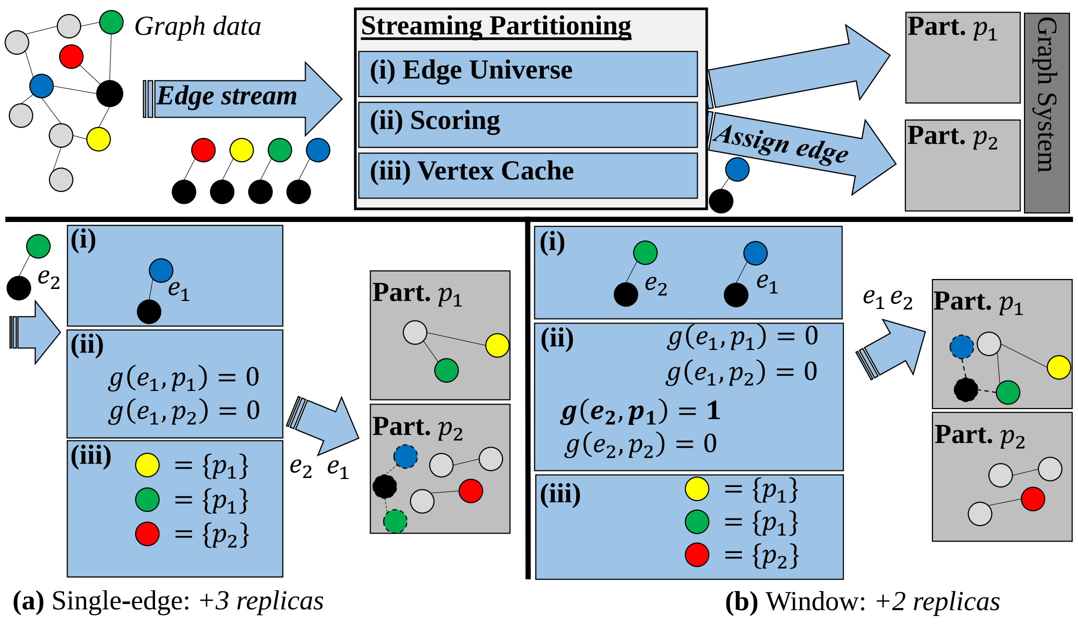


Figure 4.2: Streaming partitioning model.

4.1.3 Window-based Streaming Partitioning

To mitigate the problem of uninformed edge assignments, the idea of this work is to extend the edge universe to more than a single edge. When more edges are available in the edge universe, the partitioning algorithm can choose the next edge assignment from multiple edges. Note that if the edge universe consists only of a single edge, the algorithm is forced to assign this edge to any of the partition—even if this leads to an uninformed edge assignment. In the following, we extend the streaming partitioning model by the proposed windowing mechanism and point out the research questions that need to be solved.

Basic Approach

To improve partitioning quality, ADWISE extends the edge universe to contain multiple edges and iteratively assigns the edge with the highest score in the edge universe – thus *preferring informed and delaying uninformed edge assignments*. While the partitioning algorithm assigns more edges, it enriches the vertex cache with more information about the replica sets. In this way, the idea of ADWISE is to delay uninformed edge assignments until they become informed. For example, in Figure 4.2(b), the edge universe contains edges e_1 and e_2. The scoring function prefers assignment of edge e_2 to partition p_1 because an incident vertex is already replicated on p_1 (green vertex). By assigning edge e_2 first (i.e., *before* e_1), the algorithm learns relevant information for edge e_1 ("black vertex replicated on p_1"). It assigns edge e_1 to partition p_1 and has saved one replica compared to the single-edge streaming algorithm.

Research Questions

Adding the concept of an edge window to the streaming edge partitioning model will only improve the total latency when the window-based partitioning algorithm is carefully designed. This is a challenging task that has not been addressed in literature yet. In particular, the following questions have to be addressed.

How to set and adapt the optimal window size? Although partitioning quality can be improved by increasing the window size $w \in \mathbb{N}, w \geq 1$, this also incurs more score computations leading to higher partitioning latency. There is a complex relation between partitioning latency, partitioning quality and graph processing latency. To be able to optimize the total latency, it is necessary that the partitioning latency can be controlled, i.e., a preference on partitioning latency can be set. How this can be achieved has not been investigated in previous works, as the single-edge streaming partitioning algorithms do not allow for such a degree of freedom.

How to reduce computational complexity of partitioning? Calculating a score for each edge-partition pair in the window from scratch would lead to $O(w)$ times the computational complexity of single-edge streaming algorithms. However, from-scratch calculations might not always be necessary because of significant computational overlap between two consecutive windows. An efficient window traversal algorithm should only compute the significant score deltas to the previous window.

How to tailor the scoring function to window-based streaming? The scoring function in window-based streaming partitioning should effectively exploit the window‘s main advantage: the ability *to choose among multiple edges*. This increases flexibility – but only for carefully designed scoring functions that account for this additional dimension. Existing scoring functions from single-edge streaming partitioning can only decide about the best partition for a given edge.

In developing ADWISE, we have thoroughly investigated these research questions and have developed practical solutions, as described in the following section.

4.2 ADWISE

ADWISE, the **AD**aptive **WI**ndow-based **S**treaming **E**dge partitioning algorithm, addresses the shortcomings of single-edge streaming algorithms by extending the edge universe with multiple edges, thus enabling more flexibility in the edge assignment decisions. Figure 4.3 provides an

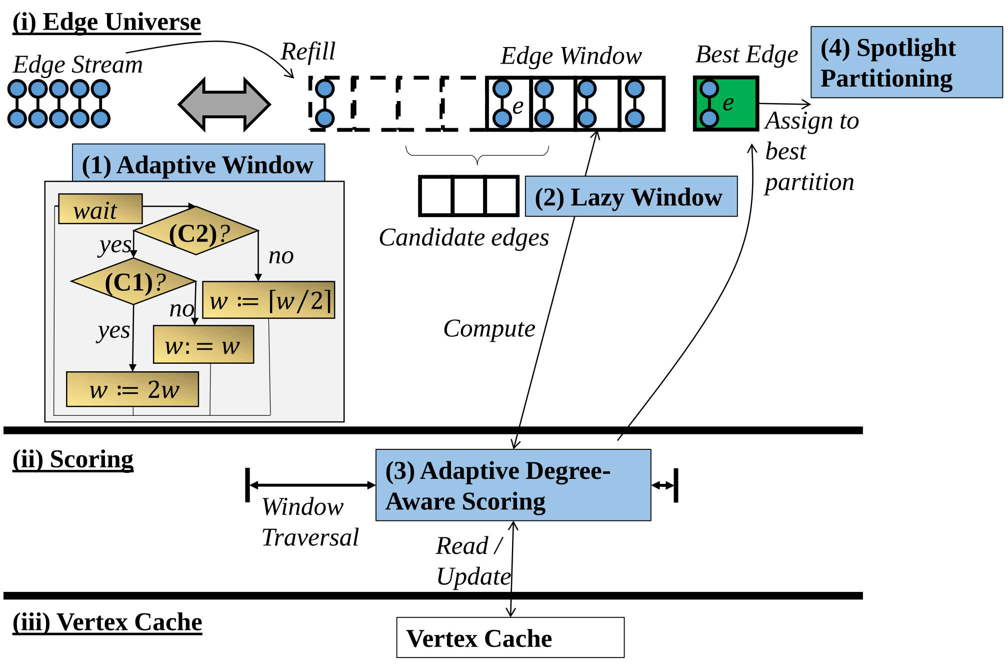


Figure 4.3: Approach overview ADWISE.

overview of the ADWISE algorithm. The edge universe consists of a window of w edges. ADWISE iteratively selects the best edge from the edge window, assigns it to the best partition, and refills the window from the edge stream to contain w edges again. In the following, we outline the general approach and highlight the main concepts of ADWISE.

(1) Adaptive Windowing: ADWISE allows to control the partitioning latency by automatically adapting the window size w at runtime such that the algorithm keeps a partitioning latency preference $L \in \mathbb{N}$ (specified in milliseconds) with high probability. In the presence of sufficient partitioning time, the window size is increased to maximize partitioning quality; if the latency preference L is likely to be violated, the window size is decreased. Section 4.2.1 provides a detailed description.

(2) Lazy Window Traversal: ADWISE exploits the property that high-score edges in one window are likely to remain high-score edges in the subsequent window. Hence, complete re-computation of the whole window after each edge assignment would lead to redundancies. We developed the optimization of *lazy window traversal* that exploits this property by calculating scores only for a subset of high-score edges in the window. The scores for the remaining edges are updated only if significant changes in the vertex cache require re-computation of individual scores (see Section 4.2.2).

(3) Adaptive Degree-Aware Scoring Function: To exploit the freedom to choose among multiple edges in a window when making the partitioning decisions, we introduce our scoring function $g(e,p)$ in Section 4.2.3. It consists of three parts:

- **Adaptive load balancing score:** The partitioning decision of single-edge streaming approaches is significantly influenced by the objective of *balancing the number of edges* among partitions (see Equation 4.2) [101]. However, we argue that balancing partitions is not equally important throughout the algorithm's execution. Instead, we introduce an optimization of adapting at runtime how much the balancing objective influences the partitioning decisions. This optimization is based on the relative progress in the stream and the current imbalance of the partitions at any point in time.
- **Degree-aware score:** The degree-aware score quantifies how good edge $e \in W$ in edge window $W \subset E$ fits to partition p by taking into account information about current replica sets from the vertex cache.
- **Clustering score:** The clustering score prioritizes assignment of edges towards the local communities of the incident vertices – exploiting the cliquishness of real-world graphs.

(4) Spotlight Partitioning: If multiple instances of a streaming partitioning algorithm partition the graph in parallel (by processing disjoint graph edge files), it is of great importance to carefully consider how many partitions are filled by the different workers (i.e., the spread). To address this problem, we propose our optimization *"Spotlight"* that is reducing the spread of each partitioner such that partitioners can maintain locality by working on their own set of partitions. Details are provided in Section 4.2.4.

4.2.1 Adaptive Window Algorithm

In the following, we explain our method for trading partitioning latency against quality. The basic idea is to increase the window size as long as this leads to better partitioning quality while the latency preference L still can be met. Otherwise, we decrease (or keep) the window size. To decide whether the latency preference can be met, ADWISE measures the average latency lat_w of *assigning a single edge* (for current window size w). The algorithm starts by setting the window size to $w = 1$. After assigning w edges and updating the average edge assignment latency lat_w, the algorithm either increases, keeps or decreases the window size (see the flow diagram in Figure 4.3). More precisely, the window size is set to $w \leftarrow 2w$, if the following two conditions (**C1**) and (**C2**) are met. (**C1**) The *last* increasing of the window size led to better edge assignment decisions (quantified by averaging the score $g(e,p)$ over w edge assignments). (**C2**) The latency preference L can be met – assuming stable average latency and a known number of edges in the stream[2]. In more detail, (**C2**) is true, if the average latency lat_w is smaller than the

[2]The graph size is usually known or can be determined efficiently using line count on the graph file

maximal latency per edge assignment, i.e., $lat_w < \frac{L'}{|E'|}$, where $|E'|$ is the number of edges left in the stream and L' is the time until the latency preference would be exceeded. This ensures that there is only a small risk of not meeting the latency preference. If the average latency is too large to meet the latency preference L, i.e., ($\neg$**C2**), the algorithm decreases the window size to $w \leftarrow \lceil w/2 \rceil$. Note that if the latency preference L is too tight (e.g. 0 seconds), the algorithm decreases w until $w = 1$ leading to single-edge streaming partitioning.

Algorithm 6 Window-based streaming vertex-cut algorithm.

```
1:  $W \leftarrow \{\}$ // Set of window edges
2:  $S$ // Edge stream
3:  $c \leftarrow 0$// Assignment counter
4:  while $S \neq \emptyset$ do
5:      if $|W| < w$ then $W \leftarrow W \cup \{S.next()\}$
6:      $(\hat{e}, \hat{p}) \leftarrow$ GETBESTASSIGNMENT()
7:      assign $\hat{e}$ to partition $\hat{p}$
8:  function GETBESTASSIGNMENT()
9:      $(\hat{e}, \hat{p}) \leftarrow argmax_{(e,p) \in W \times P} g(e, p)$
10:     $W \leftarrow W \setminus \{\hat{e}\}$
11:     if $c$ mod $w = 0$ then
12:         if (C1) $\wedge$ (C2) then
13:             $w \leftarrow 2w$
14:             while $|W| < w$ do $W \leftarrow W \cup \{S.next()\}$
15:         else if $\neg$(C2) then
16:             $w \leftarrow \lceil w/2 \rceil$
17:     $c \leftarrow c + 1$
18:     return $(\hat{e}, \hat{p})$
```

We give an algorithmic description in Algorithm 6. There are three global variables: the edge window W (initially empty), the edge stream S, and an assignment counter c tracking the number of assigned edges since the last window change. In lines 4-7, the algorithm performs the main loop: reading an edge from the stream and adding it to the window, retrieving the best edge-partition pair $(\hat{e}, \hat{p})$ from the window, and assigning edge $\hat{e}$ to partition $\hat{p}$. The algorithm retrieves the edge-partition pair $(\hat{e}, \hat{p})$ with highest score $g(\hat{e}, \hat{p})$ by iterating over all edges in the window $e \in W$ and all partitions in $p \in P$ (line 9). This edge is assigned to partition $\hat{p}$ and removed from the window (line 10). After w edge assignments, the algorithm performs the described adaptive window procedure (lines 11-17) using the two conditions (**C1**) and (**C2**).

4.2.2 Lazy Window Traversal

Clearly, the algorithm presented in the last section requires $w \times |P|$ score computations for each edge assignment resulting in large overhead for large window sizes w. In the following, we develop the idea of reducing runtime complexity by traversing only the set of high-potential edges in the window (denoted as *candidate set* C). Conversely, the *secondary set* Q contains the rest of the edges in the window. As the high-score edge is probably among the candidates, we focus on computing scores mainly for the candidates to decide which edge to assign next. If we select the candidate edges right, we will perform exactly the same assignment decisions while having a much lower runtime complexity (for $|C| << |Q|$).

But how to decide which edges to include into the candidate set? First, if we load a new edge into the window, we calculate the maximal score $\hat{g}$ for assigning edge e to any of the partitions $p \in P$. If this score is higher than a certain threshold Θ (see below), we add e to the candidate set C, otherwise we add edge e to the secondary set Q. Second, if the candidate set is empty, we calculate scores for all edges in the secondary set and add all edges whose maximal score is larger than Θ. Third, if assigning an edge leads to the creation of a new replica, the replica set of a vertex changes. In this case, edges in the secondary set that are incident to the vertex with changed replica set are reassessed whether they can be added to the candidate set. We dynamically adjust the threshold Θ to the average score g_{avg} of window edges: $\Theta = g_{avg} + \varepsilon$ for a small $\varepsilon \in [0,1]$ with the idea of including only edges in the candidate set that have better than average score.

4.2.3 Scoring Window Edges

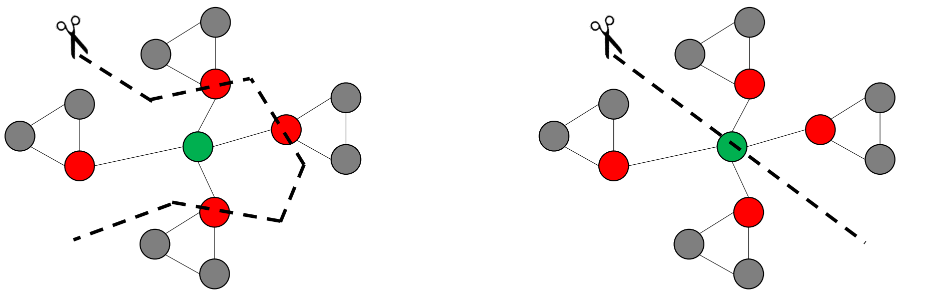
The scoring function quantifies how *good* edge e fits to partition p. However, existing single-edge scoring functions have two drawbacks. (1) They assume fixed parameter values that are chosen by domain experts. (2) They only address the problem of finding the best partition given an edge, but not the problem of finding the best edge in the window. Our scoring function extends the state-of-the-art by three optimizations to address these concerns: *adaptive balancing score*, *degree-aware window score*, and the *clustering score*.

Adaptive Balancing: The optimization constraint in Equation 4.2 requires balanced partitions. Therefore, single-edge scoring functions reinforce edge assignments towards partitions with less workload (i.e., number of edges) considering a balancing score $B(p)$ that measures the difference between partition p's and the maximal workload (see Equation 4.3).

$$B(p) = \frac{maxsize - |p|}{maxsize - minsize + \varepsilon}. \tag{4.3}$$

State-of-the-art single-edge partitioning approaches use a parameter λ to regulate how much the balancing score influences the scoring function [101]. This parameter is defined by users or domain experts [34, 101, 113]. However, selecting this parameter is a challenging problem, because different graphs require different choices.

To address this problem, we introduce an *adaptive balancing parameter* – releasing the user from the burden of choosing a suitable parameter in advance. The adaptive balancing parameter



(a) Cut vertices with lower degrees

(b) Cut vertex with high degree

Figure 4.4: Degree-aware vertex-cut partitioning.

is based on two ideas: (i) the balancing constraint can be relaxed in the beginning, i.e., λ can be set to a small value, as long as there are still enough edges to compensate imbalanced partitions; (ii) if partitions are sufficiently balanced, a high parameter value for λ distracts the scoring function from the main objective: minimize replication degree. Hence, our adaptive balancing parameter automatically adjusts to the current imbalance and progress of the partitioning algorithm. More precisely, we define the balancing parameter as a function $\lambda(\iota,\alpha)$ of the current imbalance $\iota = \frac{maxsize-minsize}{maxsize}$ and the fraction of already assigned edges $\alpha = min(1, \frac{|E'|}{m})$, where E' is the set of already assigned edges and m is the number of edges in the graph. Intuitively, the value of $\lambda(\iota,\alpha)$ should be low, i.e., tolerates high imbalance, if most edges are still unassigned. The highest acceptable imbalance, denoted as *tolerance*, should linearly decrease over time α as the end of the stream approaches, hence we define $tolerance(\alpha) = max(0, 1-\alpha)$. If the current imbalance ι exceeds the tolerated imbalance (i.e., $\iota > tolerance(\alpha)$), balancing becomes more important and $\lambda(\iota,\alpha)$ should increase. Otherwise, balancing is currently not as important and $\lambda(\iota,\alpha)$ should decrease. In Equation 4.4, we specify our formula to set $\lambda(\iota,\alpha)$ adaptively after each edge assignment. To prevent extreme values, we keep $\lambda(\iota,\alpha)$ in the fixed interval $[0.4, 5]$.

$$\lambda_{new}(\iota,\alpha) = \lambda_{old}(\iota,\alpha) + (\iota - tolerance(\alpha)). \tag{4.4}$$

Degree-aware Window Scoring: The major objective is to minimize the replication degree (see Equation 4.1). Single-edge scoring functions use a replication score $R(e = (u,v), p)$ to quantify whether vertices u and v are already replicated on partition p [34, 101, 113].

It is well-established that real-world graphs with skewed degree distributions can be divided well by preferably replicating high-degree vertices [3]. In Figure 4.4, we exemplify a stereotypical social network graph with high clustering coefficient for low-degree vertices and few high-degree vertices connecting the clusters. In Figure 4.4a, we cut the graph through vertices with median degree (red) leading to three replicated vertices. In Figure 4.4b, we cut the graph through the high-degree vertex (green) leading to only one replicated vertex.

Several approaches modify the replication score to consider *the relative vertex degree* of vertices u and v – in order to replicate high-degree before low-degree vertices [101, 140]. For instance, HDRF [101] maintains a degree table deg with the current vertex degrees to calculate the *relative degree* of vertices u and v, i.e., $\Psi'_u = \frac{deg(u)}{deg(u)+deg(v)} = 1 - \Psi'_v$. However, the *relative degree* of vertices incident to edge $e \in W$ lacks information about the *absolute degree* needed to differentiate window edges $e' \neq e \in W$. To resolve this, we introduce a truly degree-aware replication score by normalizing with respect to the vertex with maximal degree, i.e., $\Psi_u = \frac{deg(u)}{2maxDegree}$. With this modification, Ψ returns low values for low-degree vertices *in the window*. To define the replication score, we use the *indicator function* $\mathbb{1}\{p \in R_u\}$ that returns 1 (or 0) if vertex u is (or is not) replicated on partition p (i.e., $p \in R_u$).

$$R((u,v),p) = \mathbb{1}\{p \in R_u\}(2 - \Psi_u) + \mathbb{1}\{p \in R_v\}(2 - \Psi_v). \tag{4.5}$$

Clustering Score: Many real-world graphs have a high local clustering coefficient (see *small-world networks*) [136]. Graph clustering algorithms that are able to identify the dense graph regions (i.e., *clusters*) can significantly increase locality and ultimately result in better partitioning quality [73].

How can we include this prior knowledge about strong local clusters into the scoring function? In Figure 4.5, we exemplify a simple scenario, where we have to decide whether edge (u,v) should be assigned to partition p_1 or p_2. Vertex u is already replicated on both partitions so the replication score does not help here and the partitions are balanced in the number of edges. However, vertex u is already embedded into a strong local cluster on partition p_1, i.e., has three local neighbors $N_1(u) = \{u_1, u_2, u_3\}$, while it has only one local neighbor on partition p_2, i.e., $N_2(u) = \{u_4\}$. Intuitively, edge (u,v) should be assigned to partition p_1 because edges $(v,x), x \in N_1(u)$ are likely to follow in the stream (*two friends of yours are more likely to be friends as well*).

In Equation 4.6, we define the clustering score $CS(e,p)$ for edge $e = (u,v)$ and partition p as the number of times a neighboring vertex of u or v is already replicated on partition p, normalized to the interval $[0,1]$. In the example, three neighbors of (u,v), i.e., vertices u_1, u_2, u_3, are already replicated on partition p_1 compared to only one vertex u_4 on partition p_2 – leading to a higher clustering score for partition p_1. Note that for scalability reasons we calculate the neighboring function $N(u)$ for vertex u only based on the vertices in the window, i.e., the larger the window, the more accurate is the clustering score.

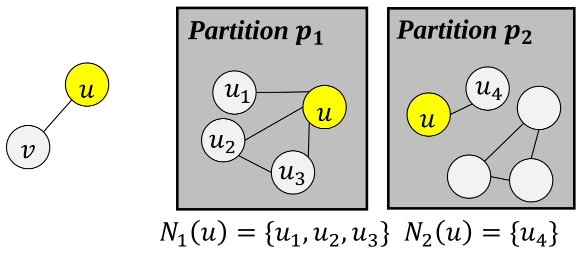


Figure 4.5: Clustering Score Example.

$$CS(e,p) = \frac{\sum_{u' \in N(u) \cup N(v)} \mathbb{1}\{p \in R_{u'}\}}{|N(u) \cup N(v)|} \tag{4.6}$$

Finally, we define the total scoring function of ADWISE in Equation 4.7.

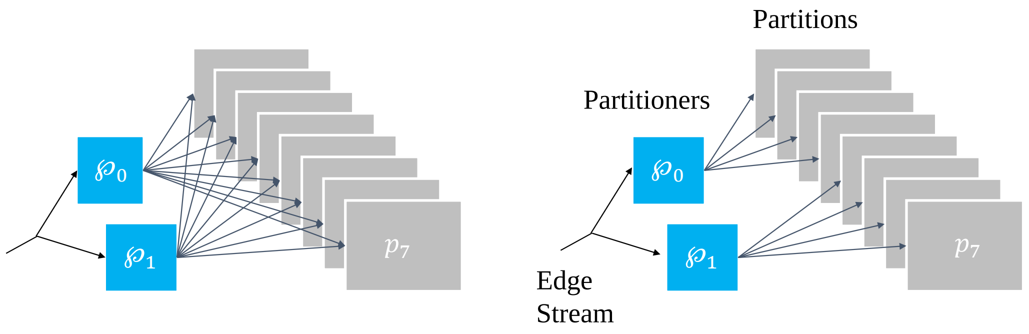
$$g(e,p) = \lambda(\iota,\alpha)B(p) + R(e,p) + CS(e,p) \tag{4.7}$$

4.2.4 Spotlight Partitioning

To speedup partitioning, graph processing systems usually employ a parallel loading model, where each worker machine uses a separate, independent streaming graph partitioner [34, 133] – each processing a disjoint portion of the global graph (i.e., *chunk*) and filling its own vertex cache. In other words, there is no vertex cache synchronization mechanism in this model and the partitioners operate independently. Due to the limited information in each vertex cache, this leads to suboptimal partitioning decisions [34].

However, we identified a second factor that negatively influences the replication degree when using parallel loading. We denote this factor as the *spread of the partitioner* which we define as the number of partitions each independent partitioner has to fill (see Figure 4.6). State-of-the-art partitioning algorithms assume a maximal spread of k, i.e., each partitioner assigns edges to each of the k partitions [34, 101].

We argue that because of this large spread, the partitioner is forced to perform partitioning decisions more often based on balancing considerations because the partitioner must equally divide the edges of its graph chunk among all partitions. Even if the incident vertices of a given edge are already replicated on partition p_0, the partitioner may be forced to assign the edge to partition p_1 to keep all partitions balanced. The more partitions there are, the higher the likelihood of such a partitioning decision. Roughly speaking, a large spread unnecessarily



(a) Partitioners with large spread (b) Partitioners with low spread

Figure 4.6: Spotlight partitioning reduces spread of partitioners to reduce the impact of balancing considerations on the partitioning decisions.

breaks up existing locality of edges in the edge stream. But many publicly available graph edge files store the edges in a breadth-first search manner[3]. In this work, we have not modified or pre-processed these original graphs in any way because this would cause additional processing overhead. In these cases, there is already a significant locality of edges in the graph file. This ultimately leads to increased replication degree as we show in Chapter 4.3.

Therefore, we propose the spotlight optimization: each of the z partitioner $\mathcal{P}_{\rangle}, i \in \{0, ..., z-1\}$ divides its graph data among $\frac{k}{z}$ partitions such that the sets of partitions of each two partitioners are disjoint. This is in contrast to the common practice, i.e., each of the z partitioners has spread k. This simple optimization is extremely effective: it reduces replication degree by up to 80 percent (see Section 4.3) for all tested strategies while *reducing* computational overhead as well due to fewer score computations. Note that the resulting partitioning is still balanced (assuming equal-sized input chunks). Although this optimization seems straightforward, it has not been applied by previous partitioning algorithms.

4.3 Evaluation

In this section, we evaluate different aspects of the ADWISE algorithm. First, we explore the trade-off between graph *partitioning* latency and *processing* latency. We show that ADWISE reduces the total graph latency (i.e., the sum of partitioning and processing latency) when computing standard graph processing algorithms on large real-world graphs. Then, we take a deeper

[3] https://snap.stanford.edu/data/index.html

Name	$\lvert V\rvert$	$\lvert E\rvert$	$\hat{c}$	Type
Orkut	3,072,441	117,184,899	0.0413	Social
Brain	734,600	165,900,000	0.509766	Biological
Web	41,291,594	1,150,725,436	0.816026	Web

Table 4.2: Real-world graphs for evaluations.

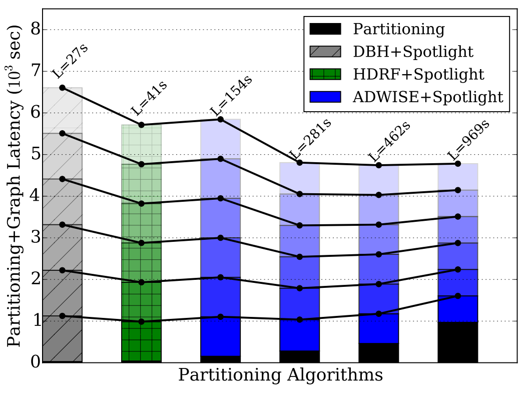
look into parallel graph loading by analyzing the effects of the spotlight optimization on partitioning quality.

Experimental Setup: In our evaluations, we used three large real-world graphs Orkut [69], Brain [110], and Web [11] with up to 1.15 billion edges (see Table 4.2). These graphs differ fundamentally with respect to the clustering coefficient $\hat{c}$: the social network Orkut has a rather weak clustering of $\hat{c} = 0.04$, the biological network Brain has moderate clustering of $\hat{c} = 0.51$, and Web has very strong clustering of $\hat{c} = 0.82$ (based on a graph sample [110]).

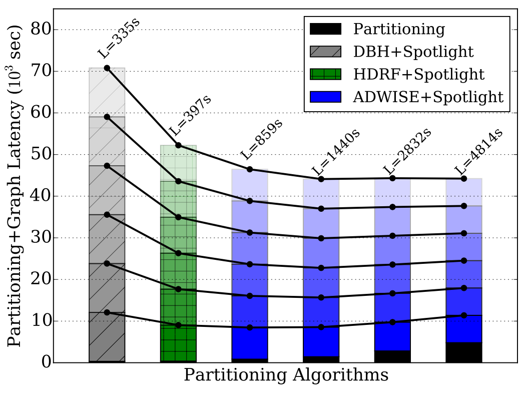
As evaluation platform, we used an in-house computing cluster with 8 nodes $\times$ 8 Intel(R) Xeon(R) CPU cores (3.0GHz, 6144 KB cache size) and 32GB RAM per node, connected via 1-Gigabit Ethernet. As benchmarks to compete against ADWISE, we evaluate Degree-based Hashing (DBH) [140] and High-Degree Replicated First (HDRF) [101] – two of the best-performing strategies w.r.t. to partitioning latency and quality [101,133,140]. For HDRF, unless stated otherwise, we set the balancing factor $\lambda = 1.1$ as recommended by the authors [101]. We integrated ADWISE as well as DBH and HDRF into the `GrapH` graph processing engine [87] to execute the graph algorithms on the partitioned graphs. Unless stated otherwise, on each of the 8 machines of the compute cluster, each instance of a partitioner (ADWISE, DBH, or HDRF) is loading a disjunct chunk of $1/8$ of the complete graph with a partitioning spread of 4; this makes a total of 32 partitions of the graph.

4.3.1 Efficacy of ADWISE to Minimize Total Graph Latency

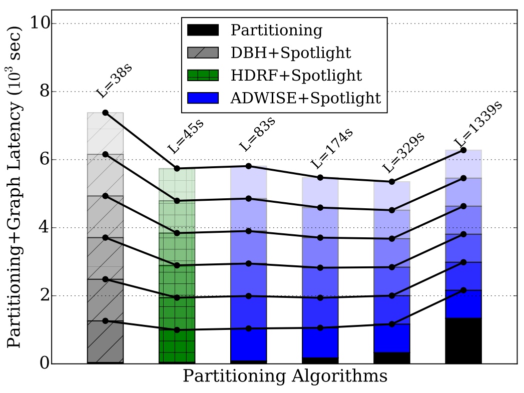
The main idea of ADWISE is to invest more time into graph partitioning in order to improve the partitioning quality, such that the sum of partitioning and processing latency, denoted as the *total graph latency*, is reduced. In the following, we show experimentally that making the trade-off between partitioning latency and quality controllable—via the partitioning latency preference L—yields a reduction of total graph latency by up to 23% compared to HDRF and by up to 47% compared to DBH when computing standard graph algorithms on large real-world graphs.



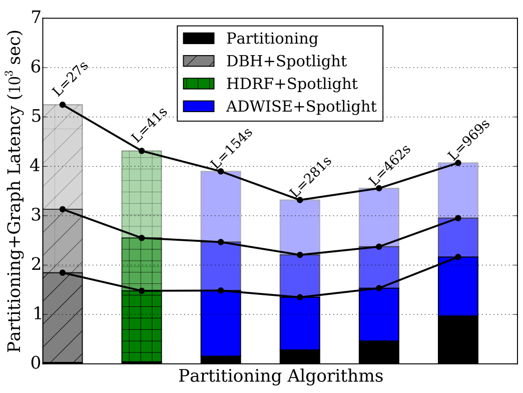
(a) PageRank on Brain



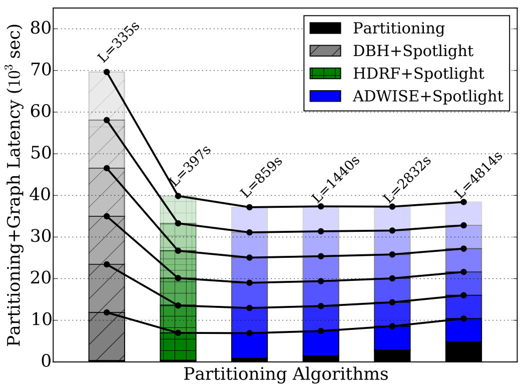
(b) PageRank on Web



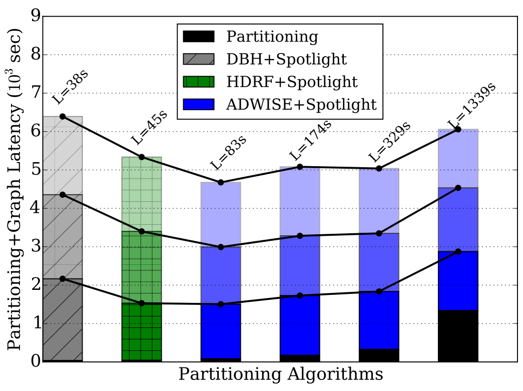
(c) PageRank on Orkut



(d) Subgraph Isomorphism on Brain

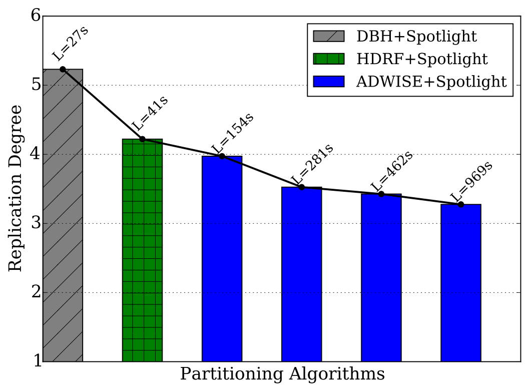


(e) Graph Coloring on Web

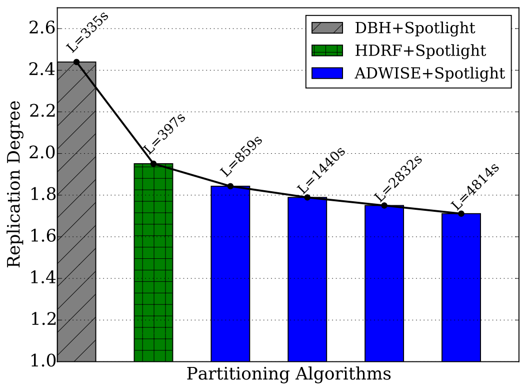


(f) Clique on Orkut

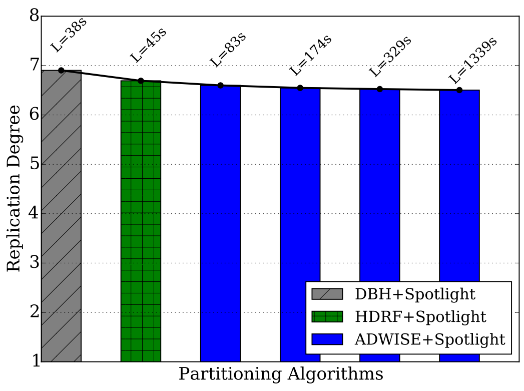
Figure 4.7: Trade-off graph partitioning latency against processing latency.



(a) Replication Degree Brain



(b) Replication Degree Web



(c) Replication Degree Orkut

Figure 4.8: Replication degree for different partitioning strategies and settings. For all presented results, the partitions are balanced, i.e., $\frac{maxsize - minsize}{maxsize} < 0.05$ (see Section 4.2.3).

Brain Graph

For our first set of experiments, we use the Brain graph that has a moderate clustering coefficient, i.e., there are relatively strong communities in the graph. As the first experiment, we executed the PageRank algorithm [34] on the Brain graph after partitioning it using DBH, HDRF and ADWISE with different (increasing) latency preferences. To evaluate the impact of partitioning quality on graph processing latency, we measured graph processing latency for blocks of 100 iterations of PageRank execution and stacked these blocks on top of the graph partitioning latency to visualize the composition of total graph latency (see Figure 4.7a). This way, the trade-off between partitioning latency and processing latency in ADWISE can be clearly seen. The most prominent observation is that ADWISE reduces total graph latency by up to 18% compared to HDRF and by up to 39% compared to DBH. Clearly, higher graph processing run-time makes it increasingly beneficial to invest more time into partitioning.

The PageRank algorithm is lightweight in terms of communication and computation: vertices exchange numerical values and perform simple arithmetic calculations in each iteration. To test communication- and computation-heavy graph processing algorithms, we execute an algorithm that solves the NP-complete subgraph isomorphism (SI) problem [86]. We searched the Brain graph consecutively for three subgraphs: circles of different lengths (i.e., path lengths of 19, 15, and 21). In Figure 4.7d, the resulting processing latencies are again visualized as stacked processing latencies on top of the partitioning latency. The sweet spot of minimal total graph latency is clearly visible for ADWISE with $L = 281s$. ADWISE reduces total graph latency by 23% compared to HDRF and by 37% compared to DBH partitioning algorithms. Even higher settings of L in ADWISE reduce the graph processing latency of the SI algorithm further, but it does not pay off in terms of total latency in the tested workloads.

The reason for reduced graph processing latency when investing more partitioning latency in ADWISE is the improved partitioning quality of the graph. To show this, in Figure 4.8a, we plotted the replication degree for the partitioning of the Brain graph and annotated each experiment with the respective partitioning latency. By increasing the partitioning latency, ADWISE reduces the replication degree subsequently by up to 29% compared to HDRF and by up to 46% compared to DBH. The reduced replication degree leads to reduced communication overhead (i.e., replica synchronization messages) and reduced computational overhead (i.e., replica processing) and therefore directly reduces graph processing latency.

The benefits of reduced graph processing latency outweighed the cost of investing more partitioning latency in the tested real-world workloads on the Brain graph, which experimentally supports our main hypothesis in this chapter. To show that this finding generalizes to other types of graphs and other graph processing algorithms, we provide further evaluations in the following.

Web Graph

The second set of experiments was performed on the Web graph that exhibits a high clustering coefficient. We measure the impact of different latency preferences in ADWISE on the total graph latency in Figure 4.7b for the PageRank algorithm. ADWISE reduces total graph latency by 16% compared to HDRF and by 38% compared to DBH. Moreover, it is already beneficial to use ADWISE with latency preference $L = 859s$ even for the first 100 iterations. When the graph processing runtime increases (i.e., more iterations are performed), it becomes more and more beneficial to invest more latency into partitioning.

To test efficacy of ADWISE on other graph processing algorithms, we also executed the graph coloring algorithm presented in [34] (see Figure 4.7e); the graph processing latency was measured after each block of 50 iterations of the graph coloring algorithm. The results show that ADWISE reduces total graph latency at partitioning latency preference $L = 859s$ by 9% compared to HDRF and by 47% compared to DBH after 300 iterations of the graph coloring algorithm. Even when executing only a single block of 50 iterations, ADWISE with latency preference $L = 859s$ already reduces total graph latency slightly compared to HDRF and significantly compared to DBH.

The partitioning quality for the different algorithms and settings is depicted in Figure 4.8b. Investing more partitioning latency in ADWISE reduces replication degree compared to HDRF by 12% (compared to DBH by 41%) for latency preference $L = 397s$ and by 25% (compared to DBH by 51%) for latency preference $L = 4814s$. As expected, allowing for larger partitioning latency in ADWISE leads to larger window sizes which lead to more informed partitioning decisions.

The evaluations on the billion-scale Web graph support our initial hypothesis that the trade-off between partitioning and graph processing latency is not addressed optimally by existing single-edge streaming algorithms. ADWISE proved its efficacy to reduce total graph latency when applied on the Web graph for both the PageRank and the graph coloring workload.

Orkut Graph

We performed a third set of experiments on the Orkut social network graph. Orkut has a low clustering coefficient, so that the clustering score in ADWISE is not effective and, hence, was switched off on this graph. For the PageRank algorithm, improving partitioning quality with ADWISE leads to decreased total graph latency by up to 11% compared to HDRF and up to 29% compared to DBH (see Figure 4.7c). Clearly, investing more partitioning latency in ADWISE pays off in comparison to the single-edge streaming partitioning algorithms.

To test generality of our findings, we also performed experiments for the graph processing problem of finding cliques of fixed size in the graph (see Figure 4.7f). We searched for cliques of sizes three, four, and five with a random walker based clique algorithm: vertices exchange messages of partially found cliques and probabilistically (P = 0.5) forward these messages if they are connected to all vertices in the partial clique message (probabilistic flooding). We performed the computation ten times for each clique size, starting the random-walk algorithm at ten different randomly chosen vertices. As the results show, the minimal total graph latency is achieved with ADWISE at partitioning latency $L = 83s$, which is 13% lower compared to HDRF. The larger partitioning latency settings $L = 174s$ or $L = 329s$ still reduce end-to-end latency slightly compared to HDRF. For even larger partitioning latencies, total graph latency increases due to the more and more prominent effect of the partitioning latency.

In comparison to the other graphs, Brain and Web, the replication degree is on a relatively high level for all partitioning algorithms (see Figure 4.8c). The reason is that the Orkut graph has a very low clustering coefficient, i.e., there is little locality in the edge stream that can be exploited by streaming partitioning algorithms. Still, ADWISE reduces replication degree by up to 4% compared to HDRF and by up to 7% compared to DBH. As observed on the execution of PageRank and clique algorithms, this small reduction of replication degree can still already lead to significant reductions of graph processing latency. We attribute this effect to the observation that some replicas contribute to overall communication overhead much more than others [86]. Improving locality of a few of those replicas can result in super-linear reductions of graph processing latency.

Result discussion: Our experiments on three real-world graphs from different domains and using four different graph processing algorithms validate that single-edge streaming partitioning algorithms are not able to solve the trade-off between partitioning latency and graph processing latency optimally. ADWISE fills this gap by allowing to invest more time into partitioning in order to improve the replication degree. This investment pays off in practical use cases – such that the total graph latency can be reduced significantly in our experiments. On the other hand, larger partitioning latencies, e.g. 10 times the single-edge latency, can lead to higher total latency due to the increasing impact of the partitioning latency.

As a practical guideline for users of ADWISE, we propose to invest about three times the latency of single-edge streaming algorithms for graph algorithms with equal or more communication volume as PageRank. If the single-edge streaming latency is not known or can not be estimated, it would even pay off to run a single-edge streaming algorithm once to measure the latency and then invest twice this latency into ADWISE.

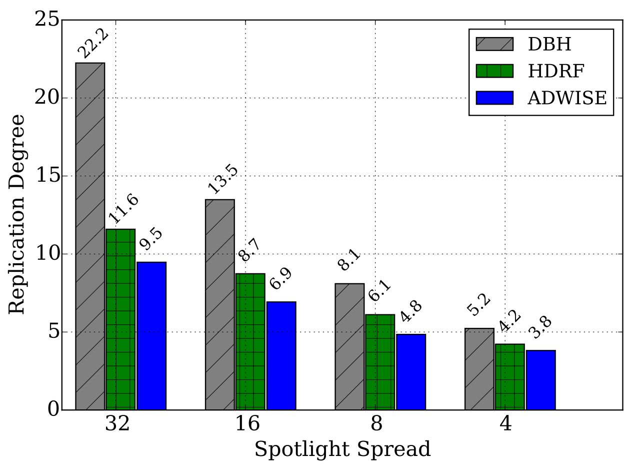


Figure 4.9: Efficacy of spotlight optimization on Brain.

4.3.2 Spotlight

Finally, we experimentally validated efficacy of the spotlight optimization (see Figure 4.9). We measured replication degree using the same computing infrastructure for all three partitioning strategies, i.e., DBH, HDRF, and ADWISE. We varied the spread of the spotlight optimization, i.e., the number of disjoint out-partitions of the $z = 8$ partitioners (see Section 4.2.4). Note that we kept the number of partitions $k = 32$ constant for all experiments, regardless of the used spread size. Instead, we only vary the number of partitions of each partitioner. In this way, we solve the same k-way vertex-cut partitioning problem but with much better results: smaller spread values lead to greatly reduced replication degree by up to 76%. The spotlight optimization is extremely effective for *all* initial partitioning strategies: it reduces replication degree significantly. Existing systems [101, 140] use a maximal spread size (e.g. spread of 32 for $k = 32$ partitions) which makes parallel graph loading less effective.

4.3.3 Summary of Evaluation Results

In summary, the evaluation results show that ADWISE reduces total end-to-end latency by up to $23 - 47\%$ compared to the partitioners HDRF and DBH. On top of that, the spotlight optimization for parallel loading reduces the replication degree of all evaluated partitioning strategies by $3 - 4\times$. Finally, we validate that these findings are reproducible for three different graphs emerging in three different areas biological networks (Brain), social network (Orkut), and web graphs (Web).

4.4 Related Work

The idea of solving large-scale graph problems in a streaming fashion is well-documented in literature [55, 116]. Stanton and Kliot [123] firstly proposed several **edge-cut partitioning** heuristics working in one pass over the graph vertices. FENNEL [129] places the vertex on a partition with many neighbors and few non-neighbors. Nishimura and Ugander [95] proposed a restreaming model that improves the partitioning in each pass using information from the previous pass. METIS [59] is an iterative offline algorithm that performs multi-level partitioning on a coarse-grained meta-graph followed by iterative adaptations on more and more fine-grained graph representations. METIS was used to produce high-quality edge-cuts for graphs with a few million edges [122] but does not scale to massive graphs [129]. Wang et al. [135] proposed a distributed partitioning algorithm based on multi-level label propagation. Zheng et al. [152] consider heterogeneous infrastructures, and Shang et al. [120] heterogeneous workloads. Martella et al. [81] proposed Spinner, a distributed edge-cut partitioning algorithm on top of the Pregel API [80] that utilizes vertex migration to adapt the partitioning dynamically. However, all of these algorithms perform edge-cut partitioning which can not be converted to a good vertex-cut partitioning [30]. For example, the number of edges to be cut in a star-like graph with $|E|$ edges is in $\Omega(|E|)$ – while the number of vertices to be cut is in $O(1)$. ADWISE employs vertex-cut partitioning.

Several streaming **vertex-cut partitioning** algorithms have been proposed. Many graph processing systems use *hashing* [34, 35] which is fast and leads to good workload balancing, but also to high replication degree, graph processing latency and communication overhead. GraphBuilder [50] is a grid-based hashing solution restricting replication of each vertex to a subset of the partitions. Degree-based hashing (DBH) [140] assigns edges to partitions by hashing only the low-degree vertex of an edge leading to better clustering properties. GraphA [75] proposes the use of an incremental number of vertex hash functions to ensure that low-degree vertices are assigned to the same partition and no large imbalances arise. The idea of 1D (and 2D) partitioning [35] is to perform edge assignments based on the adjacency matrix, i.e., assigning all edges based on the row (and column) of their source (and destination) vertex. In contrast to the previous algorithms, Greedy [34] assigns edges to partitions by considering locality explicitly, i.e., where incident vertices are already replicated. The degree-aware algorithm HDRF [101] (i.e., high-degree vertices are replicated first) is one of the best streaming vertex-cut algorithms outperforming even offline algorithms. PowerLyra [18] extends Greedy to hybrid-cuts by cutting high-degree vertices and edges incident to low-degree vertices. HoVerCut [113] enables multi-threaded processing of the graph stream by granting batch-wise, parallel access to the shared state of the partitioning algorithm. H-load [86] and G-cut [154] consider heterogeneous environments to minimize overall graph processing costs. These vertex-cut algorithms perform a single pass over the edge stream. We have shown that this extreme choice in the search space between low partitioning and low graph processing latency is not optimal for many real-world graph processing tasks. However, as a benchmark we selected the best partitioning algorithm for many graphs, i.e., HDRF, based on an experimental comparison of a wide range of single-edge streaming partitioning algorithms [133]. Note that we did not consider algorithm-specific partitioning strategies using domain knowledge to optimally partition the data for a specific graph algorithm [84]. These methods are not generally applicable to a wide range of graphs and graph applications.

H-move [86], introduced in Chapter 3, is an iterative communication-aware algorithm that repartitions the graph during graph processing. Rahmanian et al. [106] performed distributed partitioning using an iterative swap heuristic. Huang and Abadi [48] perform dynamic edge-cut partitioning with the possibility of replication, i.e., a hybrid dynamic partitioning algorithm combining edge-cut and vertex-cut. As reassignment of vertices is allowed, the proposed algorithms have super-linear runtime and, hence, do not fit into the streaming edge-cut partitioning model. Zhang et al. [150] developed an interesting all-edge neighborhood expansion (NE) heuristic with polynomial runtime that grows each partition separately using a proximity function. The authors proposed to apply NE iteratively on a random graph sample to reduce memory consumption, but there is no examination of how varying the graph sample size impacts partitioning latency and quality. Studying this trade-off is the main goal in this chapter. To summarize, all of these algorithms are computationally more intensive with **super-linear runtime** and therefore not suitable for an initial loading of the graph.

4.5 Chapter Summary

Distributed graph systems rely on fast and effective partitioning algorithms. In recent years, single-edge streaming partitioning dominated the landscape of partitioning algorithms due to the linear runtime complexity. This chapter describes the window-based streaming partitioning algorithm ADWISE that allows for investing more partitioning latency to improve partitioning quality and thus, reduce graph processing latency. The evaluation results show that ADWISE reduces total end-to-end latency by up to $23-47\%$ compared to single-edge streaming in different realistic scenarios. Moreover, the novel spotlight optimization — a simple tweak that can be applied to any partitioning algorithm with parallel loading — reduces the replication degree of all evaluated partitioning strategies by $3-4\times$ without introducing computational overhead.

5

Query-aware Multi-query Graph Processing

In Chapters 3 and 4, we focus on (static and dynamic) vertex-cut partitioning algorithms tailored to distributed graph processing. For these chapters, we assume a batch-like computation model: we load the data graph into distributed memory and execute a global graph algorithm on the distributed data graph. In this chapter, we examine a different model of computation that becomes more and more important: executing multiple graph queries with a limited scope on the distributed data graph that is shared among the queries. In particular, we focus on the placement, partitioning, and synchronization aspect of this computation model. This chapter is based on previously published work [84].

Distributed graph processing systems such as Pregel [80], PowerGraph [34], and PowerLyra [18] have emerged as the de facto standard for batch (offline) graph processing tasks due to

	Pregel [80] Kineograph [20]	PowerLyra [18] GraphX [141]	GraphH [87] Mizan [66]	Weaver [27] Seraph [143]	Q-Graph
Locality	✗	✓	✓	✗	✓
Multi-query	✗	✗	✗	✓	✓
Adaptivity	✗	✗	✓	✗	✓

Table 5.1: Research gap: multi-query graph processing with adaptive partitioning maximizing query locality.

their superior performance of data analytics on graph-structured data. However, novel graph applications have given rise to a shift of paradigms towards interactive (online) graph queries on a shared graph structure [27, 143]. These applications share three properties to which existing graph partitioning algorithms are not tailored. In the following, we quickly describe these three properties in this paragraph and show why and how these properties emerge in real-world applications in the next paragraphs. First, the scope of the queries is limited, variable, and overlapping (i.e., the query workload is *clustered* with computational hotspots in the graph). Second, multiple iterative and long-running graph queries run in parallel on a shared graph. Third, there are variations of query workload and hotspots. We denote these applications as **c**oncurrent **g**raph query **a**nalytics (**CGA**). In this chapter, we address the question: how to tailor graph partitioning and management to CGA applications in order to reduce query latency? To answer this, let us first examine three typical CGA applications.

Application 1: Today, many users request mapping services such as {Google, Apple, Yahoo, OpenStreet}-Maps to perform route planning computations that can be modeled as shortest path queries with start and end vertices on a huge road network [151]. These queries are inherently limited in scope (*what is the shortest path from home to my doctor?*) – more than 50% of search queries on mobile devices have local intent [13]. This leads to computational hotspots in certain graph areas (e.g. urban centers) that are subject to short-term changes (e.g. festivals), long-term changes (e.g. growing cities), and regular fluctuations of query workload with the time of the day or the day of the week. Hence, these services must serve multiple graph queries in parallel, with minimal query latency to yield high customer satisfaction.

Application 2: Digitized social networks allow users to measure how their online activity impacts people in their environment [31]. In general, each user accesses only her personal social network (i.e., *social circle*) – either to maintain privacy (e.g. Facebook restricts visibility of posts to defined social circles), to protect a person's integrity and trust, or simply to personalize user queries [67]. The social circles overlap because of the high clustering coefficient of social networks [136]. This leads to multiple, overlapping graph queries on shared graph data and with limited scope for social network analysis such as influence propagation [7, 63], information dissemination [17], community detection [103], and friendship recommendations [38] *on a restricted subgraph*. This type of social network analysis causes computational hotspots around hubs [87] that change over time (e.g. changing popularity of a star).

Application 3: Large-scale knowledge graphs store structured knowledge to enable smart devices to retrieve information [62]. For instance, smart cars use SPARQL [100] queries on knowledge graphs to match control rules [28]. These types of graph queries access only a small portion of the graph, but there are multiple parallel queries (e.g. from multiple smart cars) leading to computational hotspots around content with variable and dynamic popularity in the knowledge graph.

5.1 Research Gap and Contributions

We identified three challenges for CGA applications. (i) *Locality:* To parallelize query execution, graph systems partition the graph across k workers leading to *queries that are distributed across workers* and ultimately to slow query execution due to network communication and synchronization overhead between workers. Because local communication is faster than remote communication, we expect decreased average query latency when partitioning the graph in a way that increases the chance that a single query is executed on a single worker. In other words, a major challenge is to find a scalable partitioning algorithm that considers query workload to increase *query locality*, i.e., the percentage of queries that are executed on a single worker, and ultimately improving scalability and latency of the graph system. (ii) *Multi-query:* how to manage the execution of multiple queries in parallel on a shared graph? Single-query systems use a global barrier synchronization to execute the graph query in an iterative manner. This ensures that after each iteration all vertices have finished execution before proceeding with the next iteration [80]. However, we must carefully design synchronization mechanisms of multi-query systems to decouple execution of different queries such that the query locality on the partitioned graph can be fully exploited. (iii) *Adaptivity:* how to adapt the graph partitioning to dynamic query hotspots and query workloads? A key challenge is to repartition the graph at runtime to address changing query hotspots.

In Table 5.1, we categorize related work with respect to these challenges. We identified a research gap with respect to a *multi-query graph system with adaptive partitioning*. Existing graph systems such as Pregel [80], Kineograph [20], PowerLyra [18], and GraphX [141] do not support execution of multiple queries in parallel and are not adaptive to changing graph workload. Although GrapH [87] and Mizan [66] are adaptive, they are not query-aware and do not support multiple queries in parallel. Existing online multi-query systems such as Seraph [143] and Weaver [27] do not partition the graph in a locality-preserving manner.

This chapter introduces the open-source system Q-Graph[1] for CGA applications that fills this research gap. Q-Graph uses a centralized controller to maintain global knowledge about the queries running on each worker to perform *query-aware partitioning*. Q-Graph considers the dynamic query workload rather than solely the graph structure as done by state-of-the-art partitioning algorithms. We show that query-awareness of partitioning and synchronization speeds up CGA applications – as a result of improved query locality and workload balancing compared to query-agnostic static edge-cut partitioning algorithms. In particular, this chapter provides the following contributions: (i) A novel concept of query-aware partitioning to improve query locality: the Q-cut algorithm partitions the graph based on a history of queries reaching high query locality of up to 80%. Query-aware partitioning is more scalable because it operates on a small number of queries rather than a large number of vertices (see Section 5.3.2). (ii)

[1] https://gitlab.com/qgraph/GRADES2018

A hybrid-barrier synchronization model with local *and* global barriers. Local barriers reduce query latency for localized queries due to minimal synchronization overhead, while global barriers enable graph management for repartitioning and adaptivity (see Section 5.3.3). (iii) An adaptation method that optimizes the graph partitioning at runtime to dynamic query hotspots using the centralized global knowledge regarding query locality (see Section 5.3.4). (iv) A thorough evaluation of our system Q-Graph showing reduced query latency by up to 57% compared to state-of-the-art partitioning and by up to 47% compared to the state-of-the-art barrier synchronization method (see Section 5.4).

5.2 Problem Description

In this section, we introduce the background, notations, and the partitioning problem.

Background and Notations: The graph structure is given by $G = (V, E)$ with the set of vertices V and the set of directed edges $E \in V \times V$. Each vertex v maintains vertex data D_v (edge data is stored within the vertex data). We follow the predominant *vertex-centric programming model* introduced in Chapter 2 where each vertex iteratively recomputes its own vertex data based on messages from neighboring vertices. Vertex v exchanges data with a neighboring vertex v' by sending a message $m_{v \to v'}$. The application programmer specifies the vertex function $f(D_v, m_{* \to v})$ for vertex v and the set of incoming messages $m_{* \to v}$. We define a query q as a tuple (f, V_{sub}) of a vertex function f and an initial subset of active vertices $V_{sub} \subseteq V$. An example is the problem of finding the shortest path between the start vertex v_0 and the sink vertex v_{end}: The initial subset of vertices contains only the start vertex, i.e., $V_{sub} = \{v_0\}$, and the query function for a given vertex v iteratively recalculates the shortest distance from the start vertex v_0 to vertex v by updating the distance based on the distance of neighboring vertices (see Single-Source-Shortest-path (SSSP) algorithm [143]).

Multi-query graph systems support two types of requests: read-only graph analytics queries and write-enabled graph updates [24, 27, 143]. Graph analytics queries (i.e., *queries*) read the whole vertex data but write only on separate query-specific vertex data to prevent a write conflict between any pair of queries. We focus on the efficient parallel execution of multiple graph analytics queries.

To enable a consistent view on the vertex data, offline graph processing systems execute graph vertices using the synchronous and iterative bulk synchronous processing (BSP) model [80, 87, 115] consisting of three phases: i) computation, ii) communication, and iii) barrier synchronization. In the computation phase, active vertices execute the vertex function in parallel. In the communication phase, active vertices asynchronously send messages to neighboring vertices that may reside on different workers. In the synchronization phase, the system waits for all vertices to finish phases i) and ii) (*barrier synchronization*). A vertex v is considered as

$G = (V,E)$	Graph with set of vertices V and edges E.
D_v	Vertex data of vertex $v \in V$.
$m_{v \rightarrow v'}$	Message from vertex v to vertex v'.
$m_{* \rightarrow v}$	Set of incoming messages for vertex $v \in V$.
$f(D_v, \mathbb{M})$	Vertex function of v, given vertex data D_v and set of messages $\mathbb{M}$.
q	Query consisting of vertex function and set of active vertices.
$\mathbb{Q}$	Set of queries $q_i \in \mathbb{Q}$.
W	Set of worker machines $w_i \in W$.
k	Number of worker machines, i.e., $k = \|W\|$.
p	Number of queries, i.e., $p = \|\mathbb{Q}\|$.
$\mathcal{A}$	Dynamic assignment function that maps vertices to workers.
$GS(q)$	Global query scope of query q, i.e., set of active vertices within μ seconds.
μ	Time interval (sec) to capture active vertices.
$LS(q,w)$	Local query scope of query q on worker w.
L_w	Workload of worker w.
I_w	Intersection function of local query scopes on worker w.
Φ	Threshold of average query locality to decide about repartitioning.
δ	Maximally allowed imbalance.

Table 5.2: Notation overview.

active in iteration i if any other vertex has sent a message to vertex v in iteration $i-1$. These three phases, denoted as one *iteration*, are repeated until there is no active vertex.

Dynamic Partitioning Problem: A major reason for the superior performance of graph processing systems is that the graph structure provides information about data dependencies: graph vertices exchange messages only with neighboring vertices. Hence, sophisticated graph partitioning algorithms exploit the graph structure to minimize the number of messages that are sent across partition boundaries. Higher data locality improves scalability, latency, and communication overhead [87].

Given is the graph G, a set of queries $\mathbb{Q} = \{q_1, ..., q_p\}$, and a pool of workers $W = \{w_1, ..., w_k\}$. Roughly speaking, the goal is to find an assignment of vertices to workers at each point in time such that the average query latency is minimal. Note that each worker $w \in W$ executes a subset of queries $\mathbb{Q}' \subseteq \mathbb{Q}$ in parallel, depending on the vertices assigned to each worker.

The average query latency is influenced by the partitioning of the active query vertices – high locality of query vertices leads to reduced network communication and reduced query latency. The dynamic partitioning is given by an assignment function $\mathcal{A}$ that assigns vertices to workers at different points in time, i.e, $\mathcal{A} : V \times T \rightarrow W$ for the set of vertices V, a time interval $T = [T_{min}, T_{max}] \subset \mathbb{R}$, and the set of workers W. Let a function $\mathcal{X} : \mathbb{Q} \times T \rightarrow \mathcal{P}(V)$ track the set of active vertices for a query and a point in time. The function $\mathcal{X}$ incorporates the impact of partitioning decisions on the query latency – better locality leads to earlier availability of messages which leads to earlier activation times of vertices. Then, the average query latency is given by the (averaged) difference between the last and the first time instance in which a query has at least one active vertex, i.e., $\frac{1}{|\mathbb{Q}|} \sum_{q \in \mathbb{Q}} max\{t \in T | \mathcal{X}(q,t) \neq \emptyset\} - min\{t \in T | \mathcal{X}(q,t) \neq \emptyset\}$.

To quantify *partitioning quality* for CGA applications, we define the *global query scope* $GS(q) \subseteq V$ as the set of vertices that are activated during execution of query q. Likewise, we define the *local query scope* $LS(q,w) \subseteq GS(q)$ as a subset of vertices in the global scope of query q that are assigned to worker w. If $LS(q,w) = \emptyset$, query q has not activated any vertex on worker w. Furthermore, if $LS(q,w) = GS(q)$, query q is completely local on worker w. Using these definitions, we define the *query-cut* metric quantifying the quality of a given partitioning, i.e., the *locality of query execution*. More formally, query-cut is the number of non-empty local query scopes $\sum_{q \in \mathbb{Q}} |\{w \in W | LS(q,w) \neq \emptyset\}|$.

We propose to solve the problem of *balanced k-way query-aware partitioning* that minimizes the query-cut as defined previously while ensuring balanced partitions with respect to the number of active vertices (see Section 5.3.2). A smaller query-cut increases query locality and decreases query latency due to the reduced communication and synchronization overhead. Intuitively, executing a graph query on a single worker is fast (see Section 5.4.2) – a worker iterates over local vertices and executes the vertex function without waiting for remote workers, without overhead for serializing and deserializing messages, without passing the multi-layered TCP/IP stack through the operating system, and without the delay of transferring data over the network.

Research Gap: Existing partitioning methods, such as *balanced k-way partitioning* for k workers [123], are agnostic to the query workload at runtime. Graph partitioning is either based on edge-cut [123, 131] or vertex-cut [34, 87], i.e., minimizes the number of adjacent vertices or edges that are assigned to different workers. However, for CGA applications, data dependencies are not only defined by the graph structure but to a large extend by the *localization of the graph queries*. In Figure 5.1, we give an example on the neighborhood graph of New York districts with two localized queries q_1 and q_2 running on the same graph. A minimal 2-way query-aware partitioning would partition the graph such that no query is divided into multiple parts. Hence, any cut that separates queries q_1 and q_2 is minimal and leads to zero traffic between the partitions at a certain time instance. However, the *query-agnostic* edge-cut partitioning algorithm would prefer *cut 1* with edge-cut size six over *cut 2* with edge-

[3]Source of the NY map (upper right corner): Vmanjr (https://commons.wikimedia.org/wiki/File:Map_of_New_York_Economic_Regions.svg), "Map of New York Economic Regions", https://creativecommons.org/licenses/by-sa/3.0/legalcode

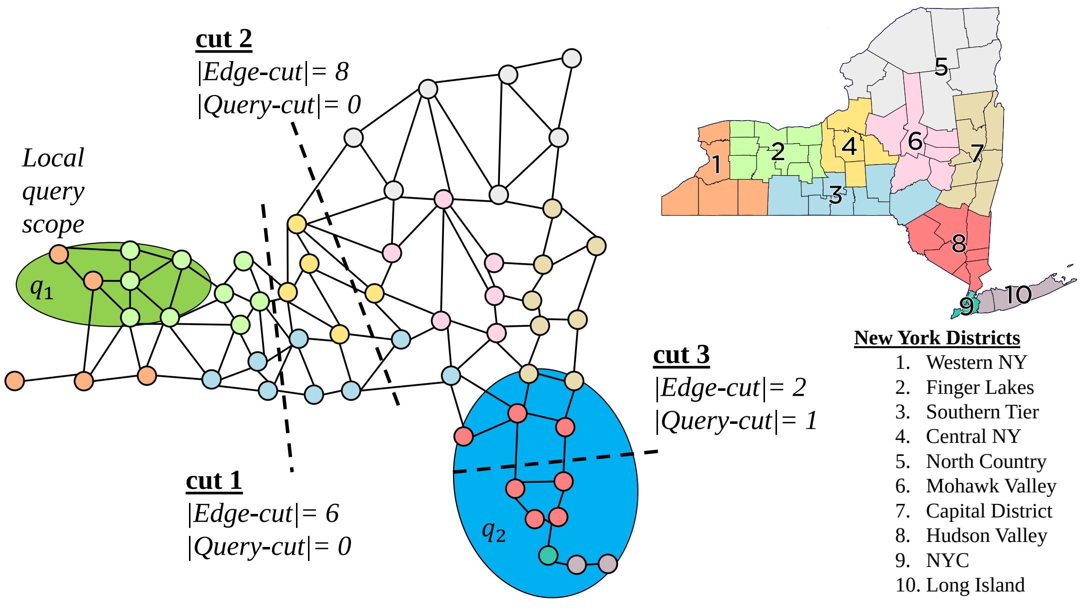


Figure 5.1: Query-agnostic partitioning optimizes edge-cut and query-aware partitioning optimizes query-cut.[3]

cut size eight – and invest scarce computational resources to calculate this cut. Therefore, query-agnostic partitioning algorithms traverse larger search spaces without producing better partitioning quality for CGA applications. Even worse, a query-agnostic edge-cut algorithm prefers suboptimal *cut 3* with edge-cut size two over *cut 1 or 2* leading to a more expensive distributed query execution.

5.3 Q-Graph System

In this section, we provide an overview of our distributed graph system Q-Graph for parallel graph query processing and describe three optimizations: query-aware partitioning, hybrid barrier synchronization, and query-aware adaptivity.

5.3.1 System Overview

Q-Graph consists of a two-layered architecture: the worker and the controller layer (see Figure 5.2). The workers perform graph processing by executing graph queries in a distributed fashion, i.e., they execute the vertex functions on the active vertices and handle message exchanges between neighboring vertices residing on different workers. The centralized controller

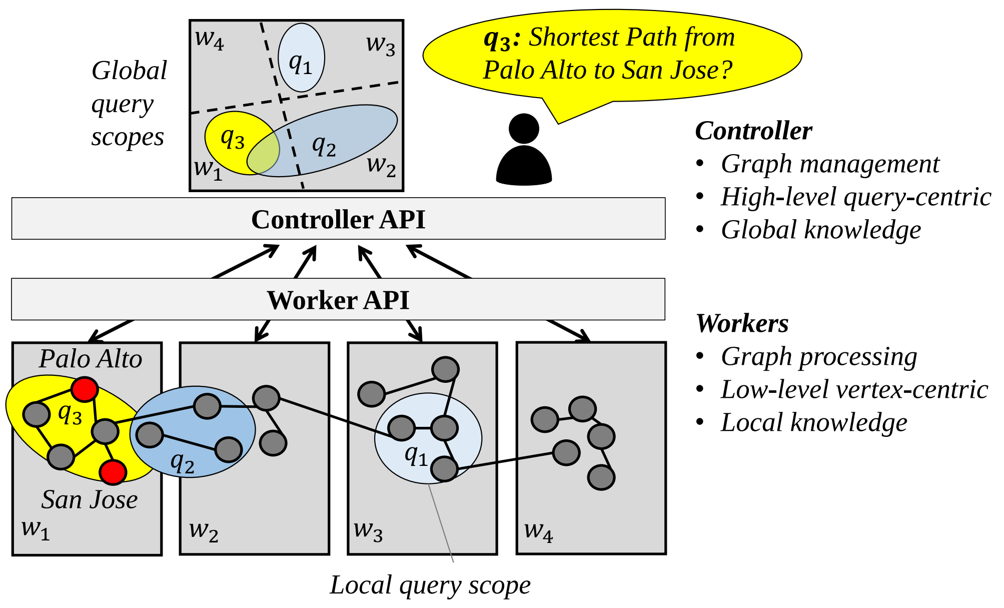


Figure 5.2: System Architecture.

manages execution of the graph queries on the distributed graph using a scalable representation of global knowledge about the graph workload – to dynamically adapt the partitioning at runtime and to ensure efficient (barrier) synchronization of the graph queries. We developed a general-purpose API for both the worker and the controller layer (see Table 5.3). The controller provides a front-end for users to access graph processing resources with the request *scheduleQuery(q)* (e.g. the user schedules query q_3 in Figure 5.2). The controller handles execution of this query by forwarding the scheduling request to the workers (i.e., calling *executeQuery(q)*). The next three sections describe the query-aware algorithm Q-cut (see Section 5.3.2), the hybrid barrier synchronization (see Section 5.3.3), and the adaptivity method (see Section 5.3.4).

5.3.2 Q-cut: Centralized Query-aware Partitioning

Q-Graph pushes partitioning decisions to the controller to benefit from the global knowledge about the query scopes and workload. The problem is that maintaining low-level global knowledge about the complete graph (vertices and edges) is not scalable. To this end, one of our central ideas is to use a more scalable representation of global knowledge by focusing on high-level query scopes rather than low-level vertex information. *As the number of queries is much smaller than the number of vertices, the controller can utilize powerful partitioning algorithms on small data to improve query locality and workload balancing*.

Command	Description
Controller API	
stats($q, \|LS(q,w)\|, I_w, w$)	Worker w updates controller with statistics about q's local query scope and intersection I_w.
barrierSynch(q,w)	Worker w indicates termination of current iteration of query q.
scheduleQuery(q)	User schedules query q.
Worker API	
move($LS(q,w), w, w'$)	Controller requests worker w to move q's local scope to w'.
barrierReady(q)	Controller releases worker waiting for q barrier.
executeQuery(q)	Controller requests worker to start query q.

Table 5.3: Q-Graph API.

Partitioning Strategy

Our strategy for query-aware partitioning involves three steps (see Figure 5.3). First, workers update the controller with query statistics after every iteration: for each active query, each worker sends the size of its local scope and the size of the *overlap* with other local query scopes. Hereby workers transform their low-level knowledge (i.e., *which* vertices each query executes) into high-level knowledge (i.e., *how many* vertices each query executes). In the example, there are two workers w_1 and w_2 and three queries q_1, q_2, q_3 on the New York districts graph from Section 5.2. Queries q_1 and q_3 span both workers leading to communication in each iteration of the query. Queries q_2 and q_3 overlap on worker w_2. Second, the controller performs query-aware partitioning (Q-cut) on the high-level representation with reduced problem size to find high-quality query-cuts. Third, the controller transforms the resulting query-cut back to the low-level representation into a proper graph partitioning by sending *move* requests to workers. In the example, local query scope $LS(q_1, w_2)$ moves to worker w_1 and local query scope $LS(q_3, w_1)$ moves to worker w_2. The new partitioning contains no queries spanning multiple workers and, therefore, has perfect query locality.

Q-cut Algorithm

The goal of query-aware partitioning is to maximize query locality. To this end, we define a cost function $c : \mathbb{S} \rightarrow \mathbb{R}$ in the space of potential solutions $\mathbb{S}$ of valid Q-cuts. We set the cost function c_s for solution s to the following query-cut metric: we sum over all queries $q_i \in \mathbb{Q}$ the number of vertices that are not assigned to the worker with the largest query scope for query q_i,

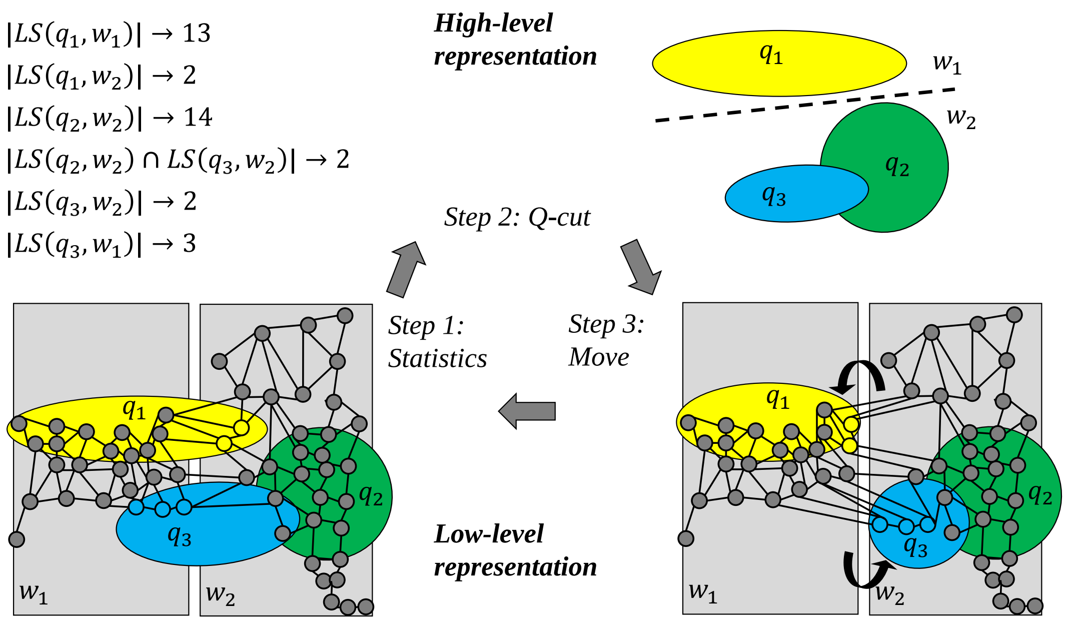


Figure 5.3: The Q-cut algorithm operates on a high-level query representation to improve low-level graph partitioning.

i.e., $\sum_{q_i \in \mathbb{Q}} \sum_{w \in W, w \neq max_{w' \in W}|LS(q_i,w')|} |LS(q_i, w)|$. For example if two workers execute two queries completely independently, the costs would be zero.

The algorithm should (a) determine low-cost solutions effectively when the controller has enough time (i.e., the problem size is small enough to be solvable within the time limit), (b) provide the best found solution when interrupted, and (c) find low-cost solutions for diverse query workloads (i.e., does not overfit to specific problems).

Algorithm 7 Iterated local search algorithm for Q-cut partitioning.

```
1: state $\hat{s} \leftarrow$ INITIALSOLUTION()
2: while not TERMINATED() do
3:     $s \leftarrow$ PERTURBATION($\hat{s}$)
4:     $s \leftarrow$ LOCALSEARCH($s$)
5:     if $c_s < c_{\hat{s}}$ then
6:         $\hat{s} \leftarrow s$
```

A well-established algorithmic framework for optimization problems meeting these requirements is *iterated local search* (ILS), a meta-heuristic that generates a sequence of solutions – each building on top of the previous solution [76]. For ILS, we must define the cost function c_s for each solution s and provide four subroutines: (i) the initial solution, (ii) a local search heuristic to find a local minimum given an initial solution, (iii) a perturbation method to overcome local minima, and (iv) a termination criterion (see Algorithm 7). After generating an initial solution, we iteratively perturb the current solution to avoid getting stuck in local minima and perform the local search method until the next local minimum is found. We store the solution with minimal costs in the variable $\hat{s}$. Selecting suitable subroutines is crucial for effectiveness and efficiency [76]. In the following, we describe the four subroutines of the Q-cut algorithm in detail, i.e., local search, perturbation, initial solution, and termination.

Local Search Subroutine We present our local search heuristic in Algorithm 8. The main idea is to simulate local query scope movements between workers as long as this leads to cost reductions. If no such move can be found, we return the last solution (denoted as *state*) which is a local minimum. More precisely, in lines 3-9, the algorithm iteratively determines the successor state s' with minimal costs $c_{s'}$ and takes this state as a starting point for the next iteration until there is no successor state with smaller costs. In lines 13-17, the algorithm determines all possible successor states that would arise by moving any local query scope from worker w to worker w' – for all respective combinations of workers and local query scopes. More precisely, as the number of these combinations can be very high, we *clustered* the queries as a preprocessing step into $4k$ clusters using a variant of the well-known Karger's algorithm with linear runtime complexity [56] and moved whole clusters between workers.

Algorithm 8 Local search heuristic to find local minimum.

```
1:  function LOCALSEARCH(State $s$)
2:      $terminated \leftarrow False$
3:      while not $terminated$ do
4:          $l \leftarrow$ SUCCESSORS($s$)
5:          $s' \leftarrow argmin_{s'' \in l} c_{s''}$
6:          if $c_{s'} < c_s$ then
7:              $s \leftarrow s'$
8:          else
9:              $terminated \leftarrow True$
10:     return $s$
11: function SUCCESSORS($s$)
12:     $l \leftarrow \emptyset$
13:     for $w, w' \in W, q \in \mathbb{Q}$ do
14:         $x \leftarrow |LS(q,w)|$
15:         if $w \neq w' \wedge x > 0 \wedge \frac{|(L_w - x) - (L_{w'} + x)|}{max(L_w - x, L_{w'} + x)} < \delta$ then
16:             $s' \leftarrow$ State after $move(LS(q,w), w, w')$
17:             $l \leftarrow l \cup \{s'\}$
18:     return $l$
```

Balancing the workload is of major importance to efficiently utilize the system resources. We define workload as a combination of two metrics: the number of vertices assigned to worker w, i.e., $|V(w)|$, and the size of the local query scopes on worker w, i.e., $\sum_{q \in \mathbb{Q}} |LS(q,w)|$. We define workload L_w as the combination of these scores, i.e., $L_w = \frac{|V(w)| + \sum_{q \in \mathbb{Q}} |LS(q,w)|}{2}$ and require that the workload of all pairs of workers is balanced, i.e., $\forall w, w' \in W : \frac{|L_w - L_{w'}|}{max(L_w, L_{w'})} < \delta$. We restrict the

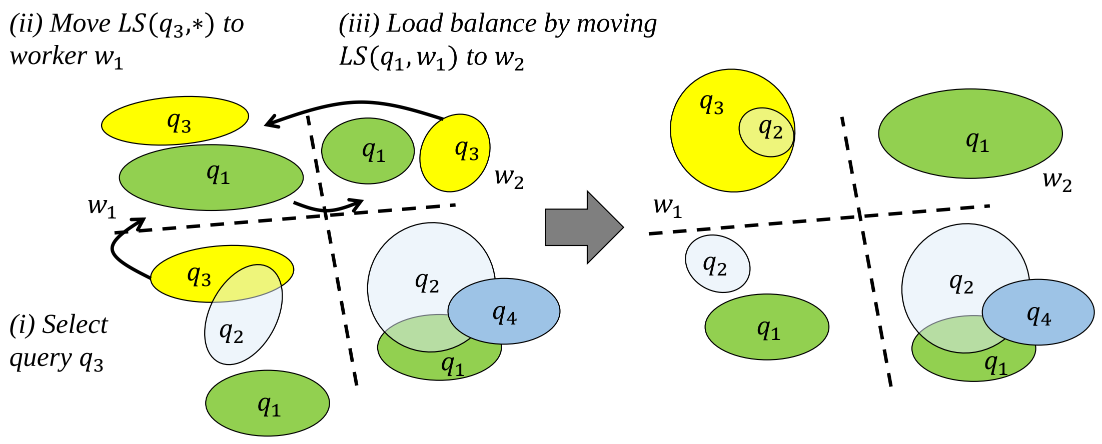

(ii) Move $LS(q_3,*)$ to worker w_1
(iii) Load balance by moving $LS(q_1, w_1)$ to w_2
(i) Select query q_3
w_1
w_2
q_1
q_2
q_3
q_4

Figure 5.4: Perturbation example.

solution space to solutions with balanced workload by excluding successor states that would result in larger workload differences due to the movement of local query scopes in Algorithm 8 line 15. We control this via the balancing threshold δ (see Section 5.4.2 for the concrete value).

Perturbation Subroutine An important subroutine of ILS is the method of perturbing a locally optimal solution state s. The perturbation transforms the converged state s into a new state s' that serves as a starting point for the next local search. A good perturbation is neither too small (i.e., the algorithm gets stuck in local minima), nor too large (i.e., the algorithm becomes uninformed). An effective perturbation strategy is to consider a meta representation of the solution (e.g. merging sets of vertices to larger clusters) and to perform a random perturbation in this representation (e.g. swapping larger clusters randomly between partitions) [76]. We used the following strategy with the idea of bringing together all local query scopes of a query to a single worker (see Figure 5.4).

I Randomly select query q being spread across at least two workers.

II Move local query scopes $LS(q,w'), w' \in W$ of query q to the worker w with the largest local query scope of query q.

III Balance workload by randomly moving local query scopes from the maximally to the least loaded worker until workload balancing is established.

This perturbation injects a certain amount of randomness (by merging local query scopes of a single query and rebalancing) but does not lead to a completely chaotic state (such as random restart).

Initial Solution and Termination Subroutine As initial solution, we take the current partitioning of the queries as received by the workers. For the termination criterion, we identified two requirements: (i) minimize the costs as much as possible to enable Q-Graph to fully exploit query locality, and (ii) prevent that the Q-cut algorithm becomes the bottleneck of the whole system. To this end, we define the termination criterion *outside of the ILS framework* by interrupting the computation as soon as a result is needed, i.e., when the adaptivity module decides to initiate repartitioning (see Section 5.3.4). The iterative nature of the Q-cut algorithm enables this early termination – even if the optimal solution is not yet reached. A major performance benefit comes from the decoupling of partitioning and computation logic: the controller can execute the Q-cut algorithm asynchronously at graph processing runtime using the latest *stats*-messages from the workers. Hence, if the controller initiates repartitioning, it can already propose a better Q-cut to the workers causing minimal latency overhead for Q-cut partitioning.

5.3.3 Hybrid Barrier Synchronization for Multi-Query Graph Processing

To perform synchronous query execution of query q, each worker w informs the controller via the API call *barrierSynch*(q,w) about the termination of a single iteration of query execution on all active query vertices. After sending this message, the worker waits for the *barrierReady*(q) message from the controller before starting with the next iteration. Thus, barriers enable iterative execution of graph algorithms and provide clear semantics for the application: in iteration i, vertex v has received all messages sent to v in the last iteration $i-1$. When the controller has received all barrierSynch-messages, it sends the *barrierReady*(q) message to all waiting workers. The barrier ensures that all vertices execute their vertex functions in lock-step going from one iteration to the next (cf. Chapter 2.1.3).

The state-of-the-art synchronization model [143] introduces an *independent global barrier* for each query. The barrier is global because all workers are involved in synchronization and independent because each query has its own barrier. However, assuming an independent global barrier causes two problems: (1) redundant *global* barriers cause communication overhead, (2) local queries, which could be executed on a single worker, face unnecessary global synchronization.

For example, suppose there are two workers w_1 and w_2 and two queries q_1 and q_2. All vertices of query q_1 reside on worker w_1 and all vertices of query q_2 reside on worker w_2. In theory, we could execute query q_1 on worker w_1 and query q_2 on worker w_2 which would lead to fast local execution of both queries. However, assuming a *global barrier* would cause both worker to synchronize *in each single iteration* of both query executions.

Why using an independent barrier? As mentioned in the last paragraph, Xue et al. [143] decoupled execution of multiple queries using an independent barrier for each query (denoted as *query barrier*). The reason is that the alternative, i.e., a barrier that is shared by all queries,

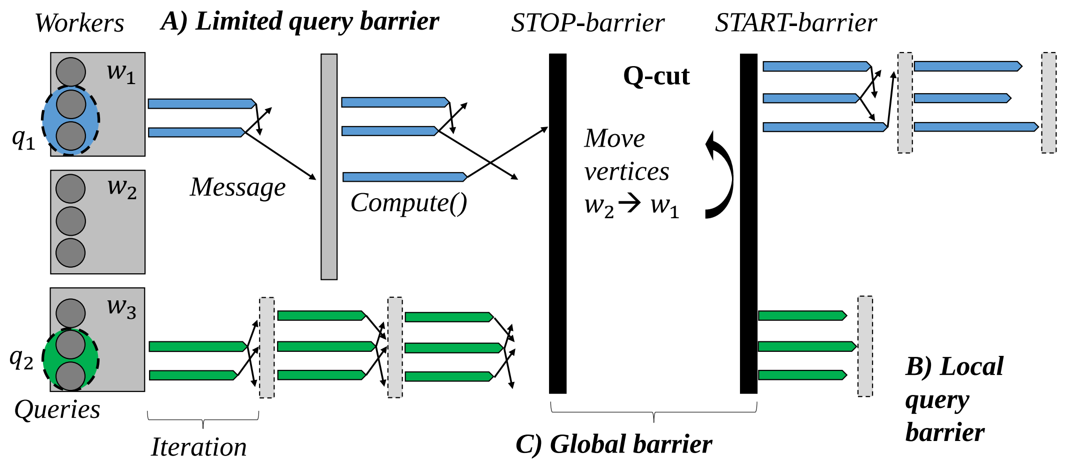


Figure 5.5: The hybrid barrier synchronization integrates: A) limited and B) local query barriers, and C) global barriers.

would result in a straggler problem because all queries have to wait for the slowest query *after each iteration*. However, to avoid unnecessary global synchronization for localized queries, we introduce *local query barriers* and *limited query barriers*, which synchronize only between workers that are currently executing the query.

Thus, we developed a three-phase synchronization mechanism, denoted as *hybrid barrier synchronization*. 1) We decouple query barriers to mitigate the straggler problem and enable independent query execution. 2) We introduce the *limited query barrier* to prevent synchronization between workers that do not execute the same query. The maximally limited barrier is the *local query barrier*, which allows communication-free execution as long as queries remain local, i.e., no distant vertices get activated via message passing. 3) In regular intervals, we initiate a global barrier that is shared across all workers. The global barrier consists of a *STOP-barrier* that halts the whole system and enables global optimization of the partitioning and a *START-barrier* that resumes normal query execution on the optimized system after all optimizations are implemented.

In Figure 5.5, we exemplify this by executing two queries q_1 and q_2 on a graph that is partitioned across three workers w_1, w_2, w_3. We indicate the computation time of a single vertex as a horizontal bar in the respective query color. Initially, both queries are local. But after the first iteration, query q_1 activates a neighboring vertex on worker w_2, which requires a limited barrier between workers w_1 and w_2 leading to the exchange of barrier messages between the workers and the controller. After three iterations of query q_2, the controller (not shown) decides to initiate a global barrier to perform repartitioning. As a result, query q_1 switches to local execution mode on worker w_1.

The hybrid barrier protocol for a single query q_1 is described in Figure 5.6. The protocol per-

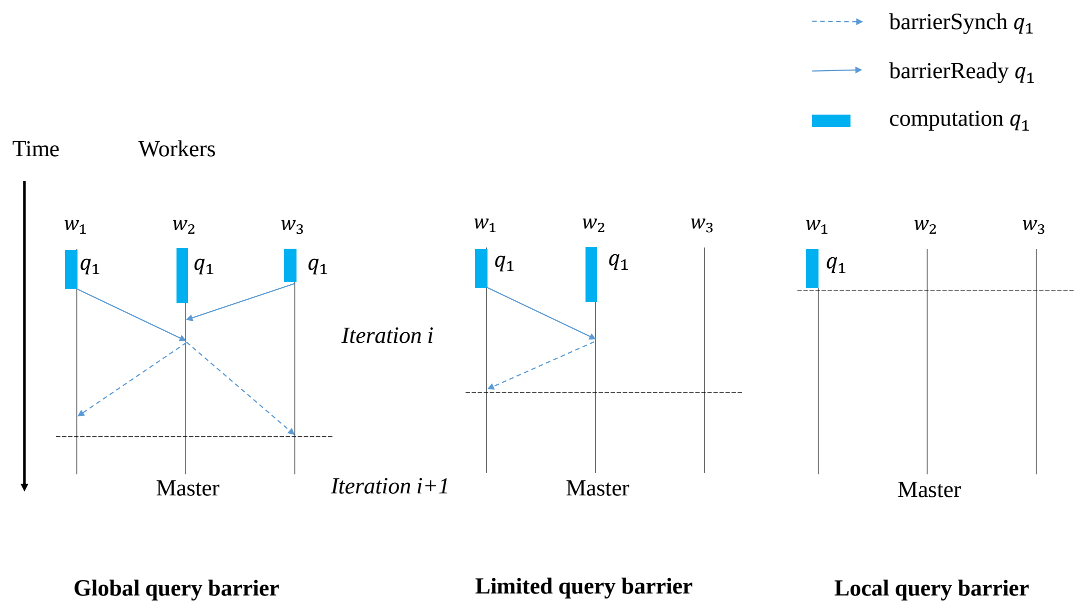


Figure 5.6: Hybrid Barrier Protocol for a single query q_1. The multi-query approach uses this protocol for each query independently.

forms a global query barrier if all workers are involved in the query. One dedicated worker is the *master* of the query (i.e., the worker with id i **modulo** k for query q_i). In each iteration, workers execute the vertex function and send a `barrierSynch` message to the master on completion of the vertex function. The master waits for all `barrierSynch` messages. If all `barrierSynch` messages are received, it sends the `barrierReady` message to each worker. Worker w_3 may leave the protocol dynamically by notifying the master that it has no active vertex for query q_1 anymore. When only one worker is left, there is no need to exchange synchronization messages. However, if the local query scope grows again to involve another worker, the query master is notified and the limited query barrier protocol is performed again.

5.3.4 Adapting to Dynamic Query Workload

`CGA` graph systems are subject to changing query workload patterns and query hotspots are not known in advance. To enable adaptivity of Q-Graph, each worker w keeps the controller updated about the sizes of the local query scopes and their intersections by regularly sending *stats* messages. The controller combines the size of the local query scopes of each query into the global query scope size and initiates repartitioning decisions by calling *move(LS*$(q,w),w,w'$*)* to move the local scope of query q from worker w to w' (i.e., fuse both local scopes of query q on worker w'). In doing so, the controller uses the global barrier. Next, we describe how query-aware partitioning adapts over time.

Dynamic Updates: The global view about the query distribution is highly dynamic, and workers have to keep the query information on the controller up-to-date. To this end, worker w sends a *stats*$(q, |LS(q,w)|, I_w, w)$ message after each iteration to indicate that query q has $|LS(q,w)|$ active vertices on worker w and intersects with other queries according to the function $I_w : 2^{\mathbb{Q}} \rightarrow \mathbb{N}$. The intersection function returns the number of shared vertices between any combination of local query scopes. For example, if three queries q_1, q_2, and q_3 involve ten vertices each and share three common vertices, the intersection function returns $I_w(\{q_1, q_2, q_3\}) = 3$. To increase communication efficiency, we piggyback statistics messages with barrier synchronization messages in our implementation.

The controller aggregates this information by calculating the global intersection functions of global query scopes based on the local intersections of local query scopes. It maintains all query statistics for a fixed duration, denoted as the (tumbling) *monitoring window*, given by the window parameter μ. The window parameter determines the degree of adaptivity of the system, i.e., how timely the partitioning reacts to dynamic query patterns. For instance, a larger window parameter allows older query statistics in the global view of the controller and leads to more long-term query-aware partitioning decisions. Furthermore, the controller initiates global barriers on demand, i.e., if the statistics indicate that the current partitioning is suboptimal. To this end, we use the *query locality*, i.e., the percentage of iterations which a query executes completely locally on a single worker, as a metric for the current quality of the partitioning. If the average query locality of all active queries is less than a threshold Φ, the controller initiates a new partitioning. In Section 5.4, we discuss the exact parameter choices. Note that the controller calculates the Q-cut algorithm (see Section 5.3.2) in parallel to the graph processing on the worker. Therefore, the partitioning latency does not affect the latency of the global barrier (see Section 5.4).

5.4 Evaluations

In this section, we show that adaptive Q-cut partitioning reduces query latency by up to 57% compared to the benchmarks. Moreover, we evaluate scalability and the hybrid barrier optimization.

5.4.1 Experimental Setup

To validate the presented concepts, we implemented Q-Graph (25k lines of Java code) and made both source code and data publicly available.

Graph Data and Query Generation: For realistic graph data, we converted the OpenStreetMap road networks (i) *Germany* (GY) to a graph with 11,805,883 vertices (i.e., junctions) and 30,804,741 edges (i.e., road segments), and (ii) the region *Baden-Wuerttemberg (BW)* to a graph with 1,802,728 vertices and 4,770,566 edges. Note that these are medium sized graphs — as our main goal is to reduce query latency in the presence of *multiple localized and iterative queries*. We set the edge weights to the length of the respective road segments, divided by the speed limit to estimate the travel time over this road segment.

We evaluated two queries: Single-source shortest path (SSSP) and Point-of-interest (POI). SSSP calculates the shortest path between a given start and end vertex. POI retrieves the closest vertex with a specified tag (e.g. gas station) to a given start vertex. We assign a tag to each vertex with probability $\frac{1}{12500}$ which is approximately the ratio of gas stations to road segments. To get realistic query workload, we determined the 64 biggest cities in GY and 16 biggest cities in BW and generated for each query a random start vertex around these *hotspots* – keeping the number of queries per city proportional to their populations. For SSSP, we also generated an end vertex with variable euclidean distance to the start vertex to account for intra- and inter-urban mapping queries.

Initial Partitioning: We used two algorithms with different strengths to initially partition the graph: *Hash* leads to ideal workload balancing, and *Domain* leads to ideal locality of up to 98% local execution per query (we validate both claims in Section 5.4.2). Domain serves as a best-case static partitioning algorithm: a domain expert, who already knows the hotspots of the query distribution in advance, manually partitions the graph such that each hotspot is assigned to a single partition. We also tested a state-of-the-art partitioning algorithm *linear deterministic greedy* (LDG) [123]. But LDG resulted in partitions with zero workload due to the skewness of the query distribution such that we excluded it from the experiments[4]. Then, we applied Q-cut on top of this initial partitioning as queries are processed, and measured how query latency and performance changes over time.

Computing Infrastructures: To evaluate both scale-up and scale-out performance, we used the following computing infrastructures: (i) *M1*, a multi-core machine with 8 CPU cores (Intel(R) Core i7-2630QM, 2.90GHz, 6MB cache) and 8GB RAM, (ii) *M2*, a multi-core machine (AWS *m4.2xlarge*) with 8 CPU cores (Intel(R) Xeon E5-2676 v3, 2.4GHz, 30MB L3 cache) and 32GB RAM, and (iii) *C1*, a cluster with 8 nodes $\times$ 8 cores (Intel(R) Xeon(R), 3.0GHz, 6MB cache) and 32GB RAM per node, connected via 1-Gigabit Ethernet. For the scale-up infrastructures, we followed a well-established design choice to exploit k cores by executing k partitions in parallel on a single machine [138] and relied on loopback TCP for communication between partitions. To evaluate efficacy of query-cut partitioning, the scale-up infrastructure is a more challenging scenario than the scale-out infrastructure because the benefits of improved partitioning (i.e., reduced network traffic) are less pronounced on a single multi-core machine.

[4]The initial experiments with this imbalanced, LDG-partitioned graph suggest an increased average query latency by factor two to six compared to the other methods.

Therefore, we first test QGraph in a scale-up (see Section 5.4.2) and then in a scale-out environment (see Section 5.4.3).

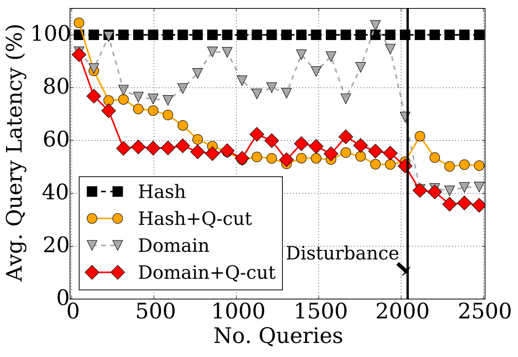
System Settings: QGraph has several system parameters that impact overall system performance. We list the most important ones in the following and published the full configuration to our open-source web repository[5] for full reproducibility: (i) we set the time to calculate a query-cut on the controller to 2 seconds. (ii) If the controller detects that the average query locality is less than threshold $\Phi = 0.7$, it starts the Q-cut algorithm (see Section 5.3.4). Although we did not observe that query latency is sensitive to the exact parameter choice, we recommend a value between the locality of Hash and Domain (see Figure 5.9b), i.e., $\Phi \in [0.3, 0.99]$. For more global queries, a modest locality level of $\Phi < 0.5$ might be preferable. However, this chapter addresses *localized* queries for which the parameter choice of $\Phi = 0.7$ is a robust decision for all performed evaluations. (iii) We set the monitoring window μ defining how long old queries will be kept (see Section 5.3.4) to 240 seconds in order to accumulate a few dozen queries for the Q-cut algorithm and restrict the maximal number of queries in the Q-cut algorithm to 128. (iv) To reduce TCP overhead, the sender thread batches vertex messages with a maximum of 32 vertex messages per batch and 32 kilobytes batch size. Increasing the batch size beyond these limits has not reduced average query latency further due to the increasing waiting time on both the sender and receiver side.

5.4.2 Adaptive Q-cut Partitioning:

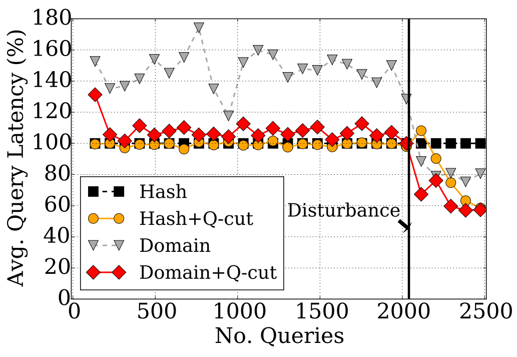
To show both static and dynamic behavior of the Q-cut algorithm, we executed 2048 hotspot SSSP queries on the BW graph in batches of 16 parallel queries ($k = 8$ workers on $M2$) – followed by a *disturbance* to test how Q-Graph adapts to changing query workloads (see Figure 5.7a). For the disturbance, we abruptly changed the query workload from intra-urban queries to random inter-urban queries between neighboring cities for further 496 queries. We measured average query latency and normalized by the query latency of Q-Graph using the static Hash partitioned graph (Q-cut switched off). In the first phase of the experiment, Q-Graph with Q-cut switched on reduces average query latency continuously by up to 49% compared to static Hash and by up to 40% compared to static Domain. The large fluctuation of Domain is a result of the increased imbalance of query workload compared to Hash and Q-cut.

In the second phase, the query workload changes: inter-urban queries become more complex with larger query scope. The bad locality of static Hash harms scalability due to high communication overhead – the relative improvement of all other approaches compared to static Hash becomes more prominent. Nevertheless, the Q-cut algorithm reduces average query latency on top of both static partitioning strategies Hash and Domain.

[5] https://gitlab.com/qgraph/GRADES2018



(a) SSSP on BW.



(b) SSSP on GY.

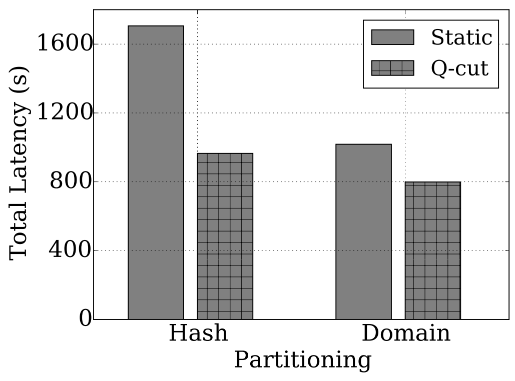
Figure 5.7: (a)-(b) Adaptive query-aware partitioning reduces query latency over time.

We performed a similar experiment on *M2* using the GY graph in Figure 5.7b. In this setting, Q-cut reduces query latency by up to 45% compared to static Hash and 30% compared to static Domain partitioning. Note that for the larger GY graph, workload balancing is a more important objective than query locality as can be seen by the relatively improved performance of Hash. The reason is that the straggler problem becomes more prominent due to the higher complexity of the road network and higher number of queries processed by the worker responsible for the largest German city Berlin. However, Q-cut reduces average query latency again on top of both partitioning strategies Hash and Domain.

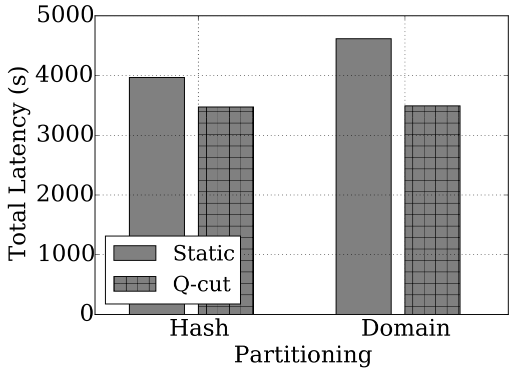
The summed latency over all 2048 queries is aggregated in Figure 5.8a for the SSSP query on the BW graph (reduced total latency by 43% compared to Hash and 22% to Domain), in Figure 5.8b for the SSSP query on the GY graph (reduced total latency by 13% compared to Hash and 25% to Domain). For the last experiment, we executed 2048 POI queries on the BW graph on *M2* to validate efficacy of Q-cut for different *types* of graph queries (see Figure 5.8c). The summed latency was reduced by 50% compared to static Hash and by 28% compared to static Domain.

Balancing the workload across machines mitigates the straggler problem and enables good resource utilization. We compare workload balancing of the four partitioning strategies in Figure 5.9a by executing 2048 SSSP queries on the BW graph (as previously described).We measured workload as the number of active vertices on a worker in a time window of 60 seconds and workload imbalance as a worker's deviation from the average workload (averaged over a sliding window of 10 seconds). Clearly, Domain leads to relative high workload imbalance because of the diverse query hotspots while Hash results in balanced workload. Q-cut converges to an imbalance of 20% because we set the maximally allowed imbalance to 25 percent, i.e., $\delta = 0.25$.

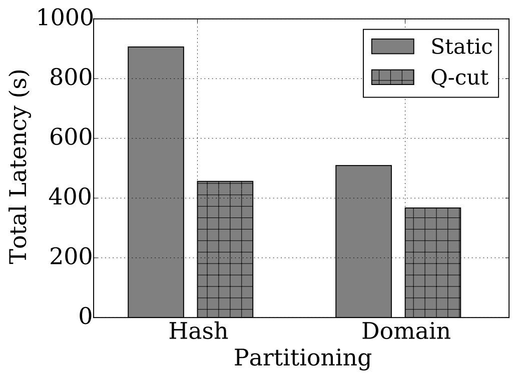
But how good is partitioning quality for the different partitioning algorithms? In Figure 5.9b,



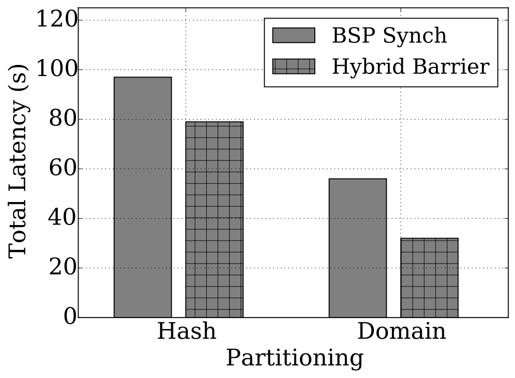
(a) Q-cut on BW for SSSP.



(b) Q-cut on GY for SSSP.

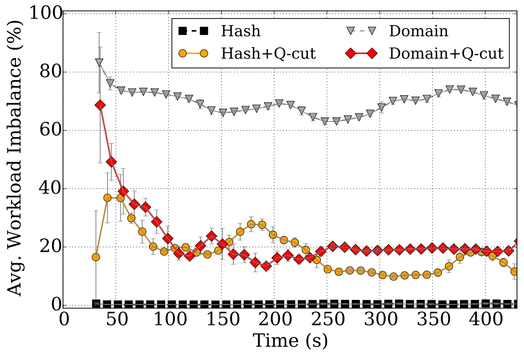


(c) Q-cut on BW for POI.

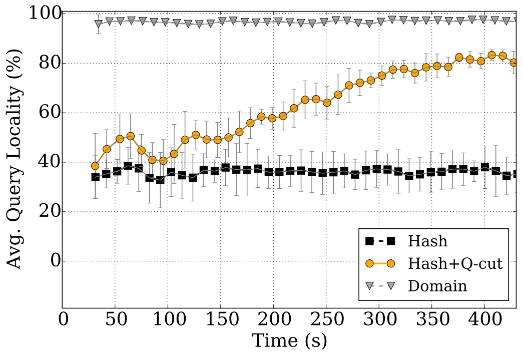


(d) Hybrid Barrier on BW for SSSP.

Figure 5.8: (a)-(b) Q-cut reduces query latency for different partitioning strategies for SSSP and (c) for POI compared to static partitioning Hash and Domain. (d) The query latency decreases with better partitioning and hybrid barrier synchronization.



(a) SSSP on BW.



(b) SSSP on BW.

Figure 5.9: (a) Workload Balancing. (b) Percentage of local query executions.

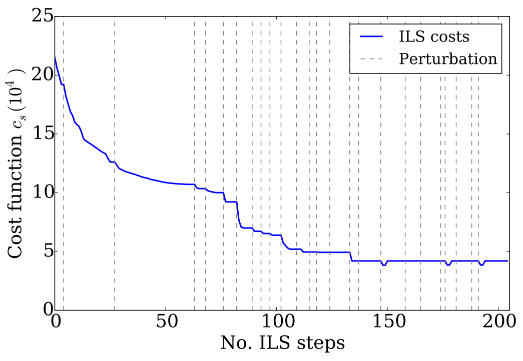


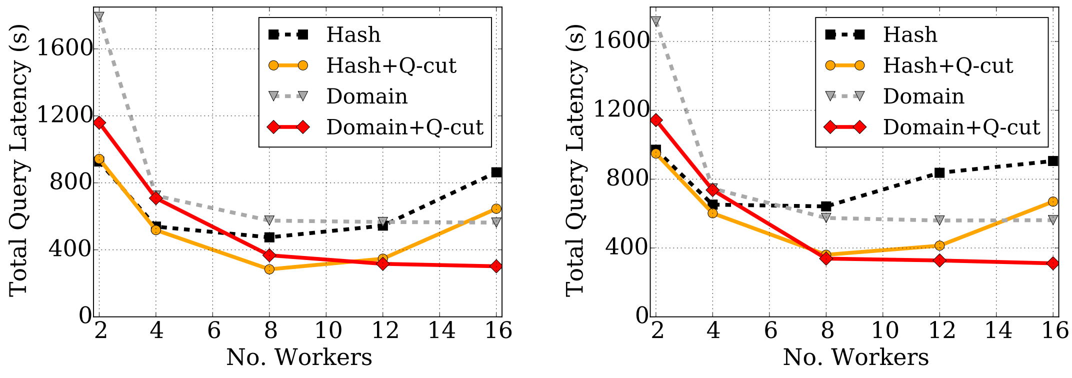
Figure 5.10: Perturbation overcomes local minima (SSSP on BW).

we measured the *query locality*, defined as the percentage of iterations that are executed completely locally. We calculated the running average and standard deviation over all queries with window size 20 seconds. The result shows that Domain leads to almost optimal locality of more than 95% while Hash reaches only 38% locality. However, even when Q-cut starts on top of the suboptimal Hash prepartitioning, locality increases over time and converges against a locality level of 80%. But, in contrast to Domain, Q-cut always ensures workload balance under dynamic query workload – higher query locality would result in higher workload imbalance which we do not allow.

Next, we validate the efficacy of the iterated local search algorithm in the controller. To this end, we plot the costs c_s of the currently found best solution state s during a single run of the algorithm. We monitored the first execution of the ILS on the controller with the Hash-partitioned BW graph. The results show that costs c_s are reduced by more than 75%. At the same time, the whole execution of the algorithm takes only two seconds – executed asynchronously to the graph query computation on the workers and hence introducing no latency penalty. Note that each ILS execution retrieves a high-quality partitioning for a graph with millions of vertices in minimal time which is a consequence of our high-level query-centric representation. We also highlighted the points of perturbation (after ILS got stuck in local minima). The combination of local searches until convergence and perturbations provides an effective method to overcome local minima – which experimentally verifies the design decisions for the perturbation routine.

5.4.3 Scalability and Hybrid Barrier

We evaluate how adding more workers impacts the total query latency of Q-Graph in Figure 5.11 for SSSP and POI on the BW graph by executing 1024 respective queries (16 parallel queries) on *C1*. For Hash prepartitioning on SSSP, the total query latency decreases from 927 to 474 seconds when increasing the number of workers from two to eight, and increases to 863 seconds when increasing the number of workers further. The reason is that Hash leads to high



(a) BW graph SSSP query.

(b) BW graph POI query.

Figure 5.11: Q-Graph scalability.

communication overhead in the distributed setting of *C1* and thus harms scalability for a large number of workers. This effect is alleviated by using Q-cut which reduces total query latency to 283 seconds for eight workers. For Domain prepartitioning on SSSP, the overall communication overhead is minimized which improves scalability – increasing the number of workers reduces latency from 1790 with two workers to 562 seconds with 16 workers. Q-cut leads to even better scalability reducing latency from 1150 with two workers to 301 seconds with 16 workers. The reason for Domain's worse runtime for a lower number of workers (e.g. $k = 2$ workers) is the suboptimal workload balancing leading to massive straggler problems. Similar results were obtained for POI.

The hybrid barrier optimization enables full exploitation of locality. To show this, we measured total query latency of our Q-Graph system for 64 shortest path queries on the BW graph ($k = 8$ workers) on *M1* (see Figure 5.8d). We compared total query latency between traditional BSP-like barrier synchronization, i.e., all queries perform global synchronization after all iterations, and our hybrid barrier synchronization. Clearly, better partitioning (Domain) leads to $1.7-2.4\times$ reduced total query latency. But importantly, the hybrid barrier optimization reduces total query latency by $1.2-1.7\times$ for *both* methods: Domain and Hash.

5.4.4 Summary of Evaluation Results

In summary, we validated that our *query-cut* partitioning algorithm dynamically improves the graph partitioning such that over time, the percentage of locally executed queries increases by 40% while keeping the workload of the partitions balanced. In combination with the hybrid barrier optimization, the resulting high locality and low workload imbalance speeds up—by factor $2.2\times$—the latency of both queries single-source shortest path (SSSP) and point of interest search (POI) on both evaluated graphs (GY and BW) with up to 4.7 million vertices.

5.5 Related Work

Several graph processing systems have influenced and inspired our work. Pregel [80] was the first iterative graph system using the general-purpose, vertex-centric programming model on a message passing implementation. Follow-up systems are PowerGraph [34], PowerLyra [18], GraphX [141], GrapH [87], Mizan [66], and GPS [115]. All of these systems support only a single, batch processing query rather than multiple, interactive graph queries. Hauck et al. [43] provided experimental evidence that single-query graph systems do not scale well in a multi-query environment.

Sedge [144] uses a complementary partitioning scheme based on replication of graph data to cope with localized queries and dynamic query hotspots. Parallelism is achieved by maintaining multiple independent Pregel instances. However, it is not possible to execute multiple queries in parallel on a single Pregel instance. Therefore, these concepts are orthogonal to our methods of Q-cut partitioning, hybrid barrier synchronization, and adaptivity.

Several *query-agnostic* partitioning algorithms optimizing the number of cut vertices or edges have been proposed [23, 54, 81, 85, 101, 113, 123, 129, 140]. As shown in Section 5.4, even best-case edge-cut partitioning algorithms lead to suboptimal locality, workload balancing, and query latency. This is a result of being agnostic to the actual query workload on the graph.

Two graph systems support concurrent localized graph analytics queries. Weaver [27] focuses on concurrency control, i.e., how to enable transactional graph updates during query execution. Their interesting concept of refinable timestamps enables efficient ACID transactions on dynamic graphs. Seraph [143] decouples the data that is accessed by the individual queries and the computational logic such as the shared graph structure. Both systems utilize a multi-version graph data structure that allows graph queries to run on a consistent snapshot. However, Weaver and Seraph do not support adaptive partitioning which is the focus of our work. Moreover, the existing body of research about graph database systems (see [6]) does not specifically optimize query localization and synchronization in vertex-centric graph processing systems.

NScale [105] is a subgraph-centric graph processing system where queries run in a fixed neighborhood around a specified vertex. They consider replication to ensure that each subgraph is "completely contained within [...] one partition". In contrast, Q-Graph allows queries to dynamically grow and shrink on a shared graph structure. Hence, Q-Graph supports a superset of more complex graph applications.

5.6 Chapter Summary

Emerging `CGA` applications with localized graph queries running in parallel on a shared graph structure require novel types of partitioning and synchronization methods. We developed and evaluated the idea of scalable management of centralized knowledge about query workload to perform *query-aware adaptive partitioning*. Combined with a novel synchronization mechanism for `CGA` applications (*hybrid-barrier*), we observed a speedup of average query latency by up to 2.2$\times$.

6

Massive Hypergraph Partitioning with Neighborhood Expansion

In the previous Chapters 3-5, we introduce specialized graph partitioning algorithms for distributed graph processing. But graphs can only capture binary relations between vertices, while important application domains require a more generalized abstraction that is powerful enough to express n-ary relations. Hence, we address the growing demand of partitioning billion-scale *real-world hypergraphs* in this chapter.

6.1 Research Gap and Contributions

Many practical applications model problems as hypergraphs where connections between vertices are multi-dimensional, i.e., each vertex can directly communicate to n vertices in a group (called “hyperedge”) and each vertex can be in multiple groups. Hypergraph partitioning [4, 14, 26, 47, 54, 61, 128, 132] deals with the problem of optimally dividing a hypergraph into a set of equally-sized components. Applications of hypergraph partitioning arise in diverse areas such as VLSI design placement [58], optimizing the task and data placement in workflows [15], minimizing the number of transactions in distributed data placement [23], optimizing storage sharding in distributed data bases [54], and as a necessary preprocessing step in distributed hypergraph processing [49].

Formally, dividing the hypergraph in k parts is denoted as the *balanced k-way hypergraph partitioning problem*. The goal is to divide the hypergraph into k equally-sized partitions such that the number of times neighboring vertices are assigned to different partitions is minimal (this is denoted as the (k-1) metric, as introduced in Section 6.2). The balanced k-way hypergraph partitioning problem is NP-hard. Hence, a heuristic approach is needed to solve the problem for massive hypergraphs.

In literature, a couple of heuristic hypergraph partitioning algorithms have been proposed, but they have shortcomings. Streaming hypergraph partitioning [4] considers one vertex at a time from a stream of hypergraph vertices. Based on a scoring function, it greedily assigns each vertex from the stream to the partition that yields the best scoring. While this algorithm has low run-time, it does not consider all relationships between all vertices when deciding on the partitioning, so that partitioning quality suffers. A recent hypergraph partitioning algorithm, Social Hash Partitioner [54], considers the complete hypergraph at once. It iteratively performs random permutations of the current partitioning followed by a greedy optimization to choose a better permutation over a worse one. While this approach generally converges to some form of an improved partitioning and is highly scalable, we argue that random permutations may not be the most effective choice for the partitioning heuristic.

In the related field of *graph* partitioning (cf. Chapters 3-5), a recently proposed algorithm uses *neighborhood expansion* to exploit structural properties of natural graphs [150]. Graphs can be regarded as special cases of hypergraphs, where each hyperedge contains only a single vertex. However, the original neighborhood expansion algorithm for graphs cannot be directly applied to hypergraphs. As hyperedges may contain a very large number of vertices, the neighborhood of a single vertex can be huge, rendering neighborhood expansion infeasible.

In this chapter, we present the first algorithm which successfully incorporates neighborhood expansion to hypergraph partitioning. We propose HYPE, a hypergraph partitioning algorithm specifically tailored to real-world hypergraphs with skewed degree distribution. HYPE grows k partitions based on the neighborhood relations in the hypergraph. We evaluate the performance of HYPE on a set of real-world hypergraphs, including a novel hypergraph data set consisting of authors and subreddits from the online board Reddit. Reddit is, to the best of our knowledge, the largest *real-world hypergraph* that has been considered in evaluating hypergraph partitioning algorithms up to now. In our evaluations, we show that HYPE can partition very large hypergraphs efficiently with high quality. HYPE is 39% faster and yields 95% better partitioning quality than *streaming partitioning* [4]. We released the source code of HYPE as an open source project: `https://github.com/mayerrn/HYPE`.

6.2 Problem Formulation

In this section, we formulate the hypergraph partitioning problem addressed in this chapter.

Problem Formulation: The hypergraph is given as $G = (V,E)$ with the set of vertices V and the set of hyperedges $E \subset 2^V$. Given vertex $v \in V$, we denote the set of adjacent vertices, i.e.,

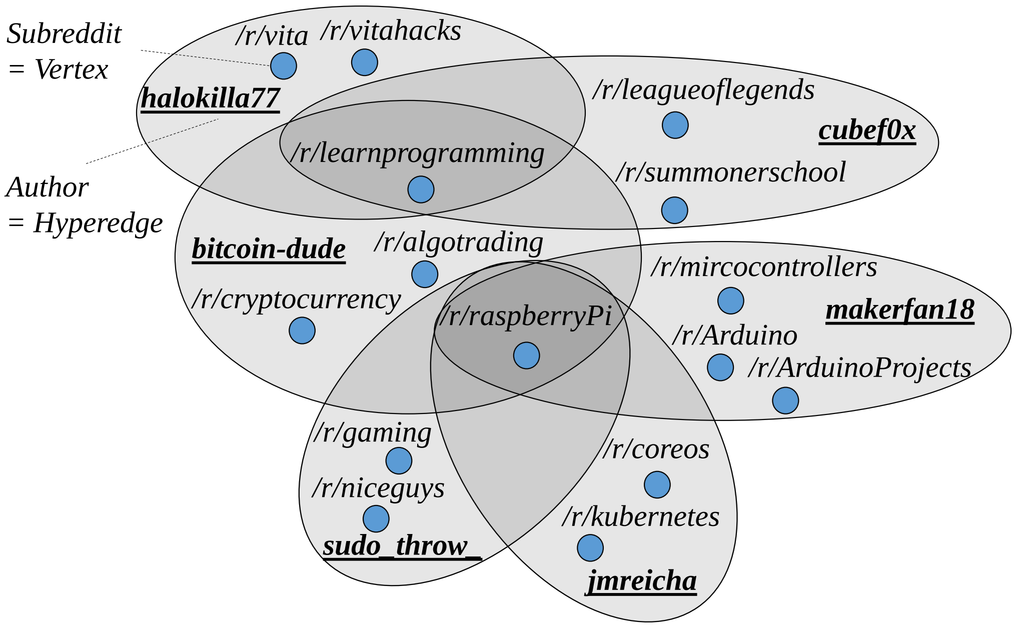


Figure 6.1: A small extract of the global Reddit graph.

the set of neighbors of v, as $N(v) \subseteq V$. The goal is to partition the hypergraph into k partitions $P = \{p_0, p_1, ..., p_{k-1}\}$ by assigning vertices to partitions. The assignment function $A : V \to P$ defines for each vertex in V to which partition it is assigned. We write $A(v) = p_i$ if vertex v is assigned to partition p_i. Each hyperedge spans between 1 and k partitions. The optimization objective is the $(k-1)$-cut that sums over each hyperedge the number of times it is assigned to more than one partition, i.e., $\sum_{e \in E} |\{p \in P | \exists v \in E : A(v) = p\}| - 1$. We require that the assignment of vertices to partitions is balanced in the number of vertices assigned to a single partition, i.e., $\forall p_0, p_1 \in P : |\{v \in V | A(v) = p_0\}| < \lambda |\{v \in V | A(v) = p_1\}|$ for a small balancing factor $\lambda \in \mathbb{R}$. This problem is denoted as *balanced k-way hypergraph partitioning problem* and it is NP-hard [5].

Reddit Hypergraph Example: We give an example of the Reddit hypergraph in Figure 6.1. The Reddit social network consists of a large number of subreddits where each subreddit concerns a certain topic. For example, in the subreddit `/r/learnprogramming`, authors write comments related to the topic of learning to program. Each author writes comments in an arbitrary number of subreddits and each subreddit is authored by an arbitrary number of users. One way to build a hypergraph out of the Reddit data set (see Section 6.4) is to use subreddits as vertices and authors as hyperedges that connect these vertices. This hypergraph representation provides valuable information about the similarity of subreddits. For instance, if many authors are active in two subreddits (e.g. `/r/Arduino` and `/r/ArduinoProjects`), i.e., many hyperedges overlap significantly in two vertices, it is likely that the two subreddits concern similar

G	Hypergraph $G = (V, E)$
V	Set of vertices $V \subset \mathcal{N} \times \mathcal{N}$
E	Set of hyperedges $E \subset 2^V$
$N(v)$	Set of adjacent vertices of vertex $v \in V$
$N(X)$	Union of all sets $N(x)$ for $x \in X$
P	Set of partitions $p_i \in P$ with $\|P\| = k$
A	Function assigning vertices V to partitions P
λ	Balancing factor
C	Current core set of vertices
F	Current fringe
$d_{ext}(u, S)$	External neighbors score of vertex u
s	Maximal size of the fringe
r	Number of fringe candidate vertices

Table 6.1: Notation overview.

content.

The size of the hyperedges and the degree of the vertices resemble a power law degree distribution for real-world graphs such as Reddit, Stackoverflow, and Github. Hence, most vertices have a small degree and most hyperedges have a small size. These parts of the graph with small degrees build relatively independent communities. In the reddit graph, there are local communities such as people who write in the `/r/Arduino` and `/r/ArduinoProjects` subreddits but not in, say the `/r/Baby` subreddit. This property of strong local communities is well-observed for graphs [136]—and it holds for real-world hypergraphs as well. Power law distributions are long-tailed, i.e., there are some hub vertices or edges with substantial sizes or degrees. In graph literature, it has been shown that focusing on optimal partitioning of the local communities at the expense of optimal placement of the hubs is a robust and effective partitioning strategy [3, 34, 101]. In Section 6.3, we show how we exploit this observation in the HYPE partitioner.

6.3 Hypergraph Partitioning with Neighborhood Expansion

In the following, we first explain the idea of neighborhood expansion in Section 6.3.1. We introduce our novel hypergraph partitioning algorithm HYPE in Section 6.3.2 and discuss the balancing of hypergraph partitions in Section 6.3.3. Finally, we present a more formal pseudocode notation of the HYPE algorithm in Section 6.3.4 and perform a complexity analysis in Section 6.3.5.

6.3.1 Neighborhood Expansion Idea

A practical method for high-quality *graph* partitioning is *neighborhood expansion* [150]. The idea is to grow a core set of vertices via the neighborhood relation given by the graph structure. By exploiting the graph structure, the locality of vertices in the partition is maximized, i.e., neighboring vertices in the graph tend to reside on the same partition. The algorithm grows the core set one vertex at a time until the desired partition size is achieved. In order to partition the graph into k partitions, the procedure of growing a core set C_i is repeated k times for $i \in \{0, 1, ..., k-1\}$.

We aim to adopt neighborhood expansion to hypergraph partitioning. To this end, we have to overcome a set of challenges which are related to the different structure of hypergraphs when compared to normal graphs. In particular, the number of neighbors of a vertex quickly explodes as the hyperedges contain multiple neighboring vertices at once. Before we explain in detail those challenges and our approach to tackle them, we sketch the basic idea of neighborhood expansion in the following.

Figure 6.2 sketches the general framework for growing the core set C_i of partition $p_i \in P$. There are three overlapping sets: the vertex universe, the fringe, and the core set. The vertex universe $V' \subseteq V$ is the set of *remaining vertices* that can potentially be added to the fringe F_i, i.e., $V' = V \setminus C_0 ... \setminus C_i \setminus F_i$. The fringe F_i is the set of vertices that are currently considered for the core set. The core C_i is the set of vertices that are assigned to partition $p_i \in P$. The three sets are non-overlapping, i.e., $V' \cap F_i = F_i \cap C_i = C_i \cap V' = \emptyset$.

Initially, the core consists of seed vertices that are taken as a starting point for growing the partition. Based on these seed vertices, the fringe contains a subset of all neighboring vertices. In graph partitioning [150], the fringe contains not a subset but all vertices that are in a neighborhood relation to one of the vertices in the core set. In the Figure 6.2, a fringe candidate vertex, say vertex v, is moved from the vertex universe to the fringe and then to the core set. In other words, any strategy based on neighborhood expansion must define the two functions `upd8_fringe()` and `upd8_core()`.

As we develop a hypergraph partitioning algorithm based on the neighborhood expansion framework, we define the neighborhood relation and the three vertex sets accordingly. However, migrating the idea of neighborhood expansion from graph to hypergraph partitioning is challenging. The number of neighbors of a specific vertex in a typical hypergraph is much larger than in a typical graph. The reason is that the number of neighbors is not only proportional to the number of incident hyperedges but also to the size of these hyperedges. For example, sup-

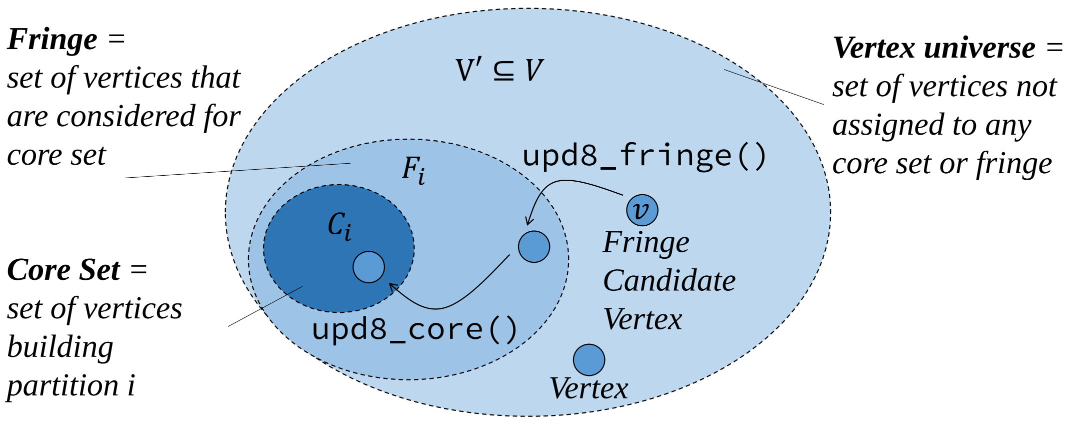


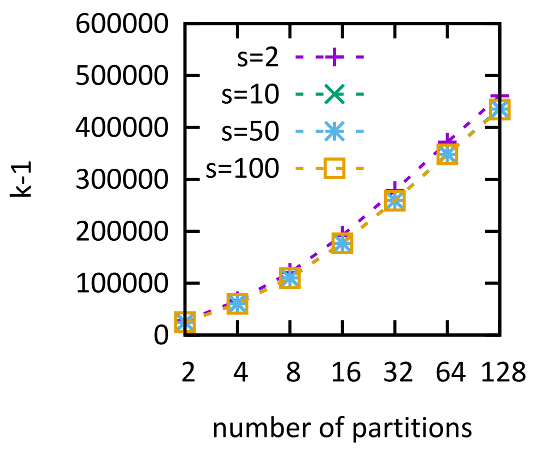
Figure 6.2: High-level idea of neighborhood expansion.

pose you are writing a comment in the `/r/Python` subreddit. Suddenly, hundreds of thousands other authors in `/r/Python` are your direct neighbors. In other words, the neighborhood relation is *group based* rather than *bidirectional* which leads to massive neighborhoods. The large number of neighbors changes the runtime behavior and efficiency of neighborhood expansion. For instance, in such a hypergraph, the fringe would suddenly contain a large fraction of the vertices in the hypergraph. But selecting a vertex from the fringe requires $O(|V|)$ comparisons. This leads to high runtime overhead and does not scale to massive graphs. HYPE alleviates this problem by reducing the search space significantly as described next.

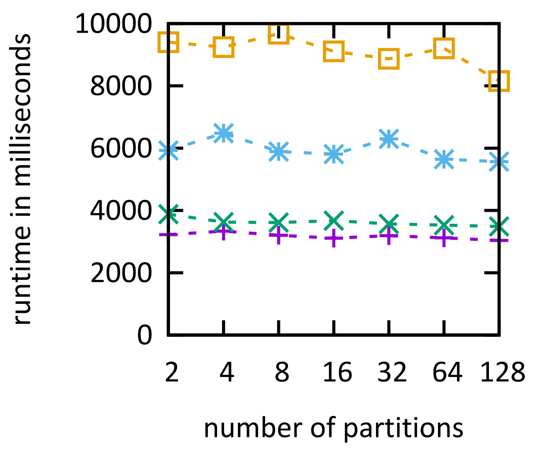
6.3.2 HYPE Algorithm

The HYPE algorithm grows the core set for partition $p_i \in P$ one vertex at a time. We load one vertex from the fringe to the core set and update the fringe with fresh vertices from the vertex universe. The decision of which vertex to include into the core and fringe sets is a critical design choice that has impact on the algorithm runtime and partitioning quality.

In this section, we explain the HYPE algorithm in detail, including a discussion of the design choices. In doing so, we first provide the basic algorithm in Section 6.3.2 and then discuss optimizations of the algorithm in Section 6.3.2. These optimizations tremendously reduce the algorithm runtime without compromising on partitioning quality.



(a) Partitioning quality.



(b) Runtime.

Figure 6.3: Limiting the fringe size s to a small value (e.g. $s = 10$) keeps partitioning quality intact while reducing runtime significantly (StackOverflow hypergraph).

Basic Algorithm

The approach of growing a core set C_i is repeated in a sequential manner for each partition p_i for $i \in \{1, 2, ..., k\}$. It consists of a four step process. We initialize the computation with step 1., and iterate steps 2. and 3., until the algorithm terminates as defined in step 4.

1. Initialize the core set.
2. Move vertex from vertex universe into fringe.
3. Move vertex from fringe into core set.
4. Terminate the expansion.

We now examine these steps in detail. A formal algorithmic description is given in Section 6.3.4.

1. Initialize the core set The core set C_i must initially contain at least one vertex in order to grow via the neighborhood expansion. In general, there are many different ways to initialize the core set. This problem is similar to the problem of initializing a cluster center for iterative clustering algorithms such as `K-Means` [42]. Here, the defacto standard is to select random points as cluster centers [12, 65]. In fact, a comparison of several initialization methods for `K-Means` shows that the random method leads to robust clustering quality [98]. As the problem of selecting an initial "seed" vertex from which the core set grows is similar to this problem, we perform random initialization.

2. Move vertex to fringe The function `upd8_fringe()` determines which vertices move from the *vertex universe* to the fringe F_i. The vertex universe consists of all vertices that are neither in the fringe F_i, nor in the core set C_i of any previous execution of the algorithm for any partition $p_j \in P$ with $j < i$.

The standard strategy of neighborhood expansion is to fill the fringe F_i with all vertices that are neighbors to any core vertex, i.e., $F_i \leftarrow N(C_i) \setminus C_0 \setminus ... \setminus C_i$. But for real-world hypergraphs, this quickly overloads the fringe with a large number of vertices from the vertex universe. To prevent this, we restrict the fringe to contain only s vertices, i.e., $|F_i| \leq s$. In Figure 6.3, we validate experimentally that setting s to a small constant value, i.e., $s = 10$, keeps partitioning quality high but reduces runtime by a large factor. For brevity, we omit the discussion of similar results observed for other hypergraphs.

But how do we select the next vertex to be loaded into the fringe? Out of the vertex universe V', we select a vertex to be included to the fringe using a scoring metric as described in the next paragraphs. The intuition behind this scoring metric is to find the vertex that preserves the highest *locality* when assigned to the core set.

To this end, we define *locality* as the frequency that for a given vertex $v \in V'$, a neighboring vertex $v' \in N(v)$ resides on the same partition. High locality leads to low cut sizes and good partitioning quality. To improve locality, our goal is to grow the core set into the smaller local communities and assign all vertices of these smaller communities to the same partition. If a high proportion of neighbors of vertex $v \in V'$ is already assigned to the core set, assigning vertex v to the core set as well will improve locality.

In Figure 6.4, we see an example hypergraph that has the typical properties of real-world hypergraphs: the size of the hyperedges follows a power law distribution. To grow the fringe, there are three options: include vertex v_1, v_2, or v_3. Intuitively, we want to grow the fringe towards the local communities to preserve locality. We achieve this by selecting vertices based on the *external neighbors* with respect to the fringe F. In other words, we want to add vertices to the fringe that have a *high* number of neighbors in the fringe or the core set, and a *low* number of neighbors in the remaining vertex universe. A vertex with low external neighbors score tends to have high locality in the fringe and the core set. Formally, we write $d_{ext}(v, F_i)$ to denote the number of neighboring vertices of v that are not already contained in the fringe as defined in Equation 6.1.

$$d_{ext}(v, F_i) = |N(v) \setminus F_i| \tag{6.1}$$

We denote the vertices for which we calculate the external neighbors score as *fringe candidate vertices*. For each execution of `upd8_fringe()`, we select r fringe candidate vertices. The

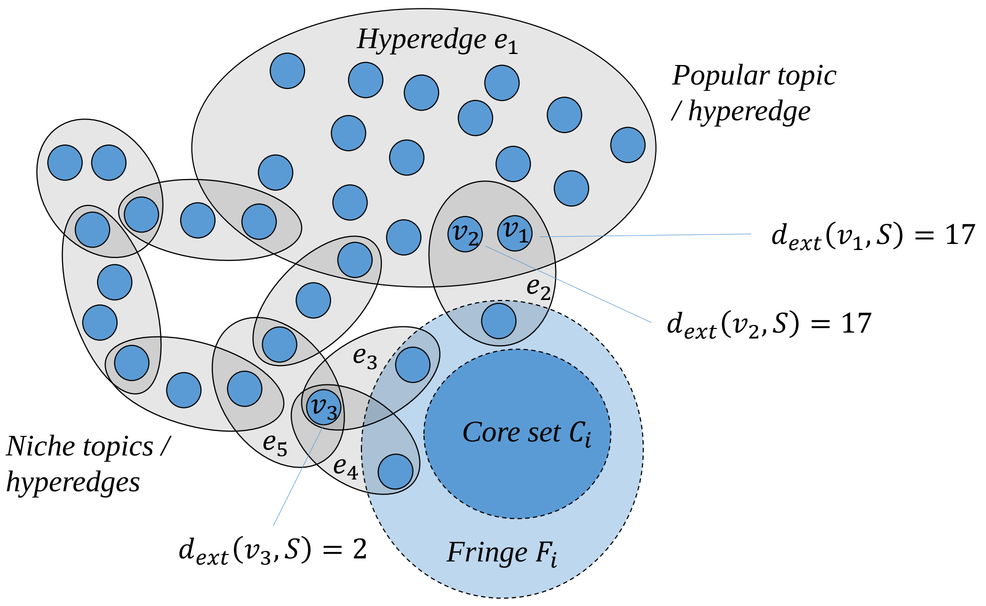


Figure 6.4: The external neighbors metric determines which vertex to move into the fringe.

fringe F_i contains up to s vertices. Hence, after assigning one vertex to the core set, we take the top s vertices out of the $s-1+r$ fringe candidate vertices as the new fringe.

In Section 6.3.2, we describe three optimizations on the `upd8_fringe()` function that reduce the runtime while keeping the partitioning quality intact.

3. Move vertex to core set Next, we describe the function `upd8_core()` that moves a vertex from the fringe F_i to the core set C_i. The function simply selects the vertex with smallest (cached) external neighbors score. This vertex v is then moved to the core, i.e., $C_i \leftarrow C_i \cup \{v\}$. This decision is final. Once assigned to the core C_i, vertex v will never be assigned to any other core C_j when considering a partition $j > i$. Hence, we remove the vertex from the remaining set of vertices in the vertex universe, i.e., $V' \leftarrow V' \setminus \{v\}$. In case the fringe can not be filled with enough neighbors, we add a random vertex from the vertex universe to the fringe and proceed with the given algorithm.

4. Terminate the expansion We terminate the algorithm as soon as the core set contains $\frac{|V|}{k}$ vertices. This leads to perfectly balanced partitions with respect to the number of vertices. Upon termination, we release the vertices in the fringe F_i and store the vertices in the core set C_i in a separate partitioning file for partition i. After this, we restart expansion for the next partition $i \leftarrow i+1$ or terminate if all vertices have been assigned to partitions. In Section 6.3.3, we discuss other possible balancing schemes.

Dataset	Vertices	Hyperedges	#Vertices	#Hyperedges	#Edges
Github [69]	Users	Projects	177,386	56,519	440,237
StackOverflow [69]	Users	Posts	641,876	545,196	1,301,942
Reddit	Subreddits	Authors	430,156	21,169,586	179,686,265
Reddit-L	Comments	Authors & Subreddits	2,862,748,675	21,599,742	5,725,497,350

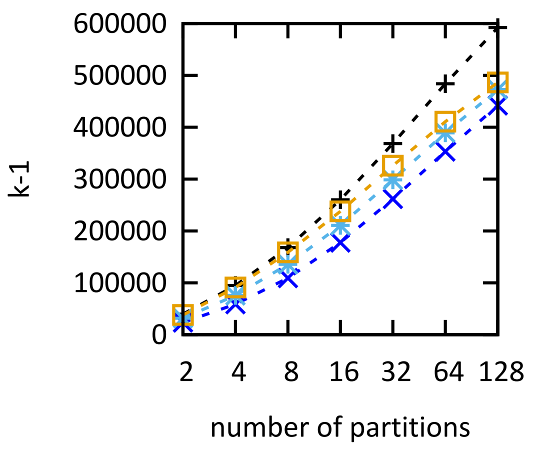
Table 6.2: Real-world hypergraphs used in evaluations.

Optimization of Fringe Updates

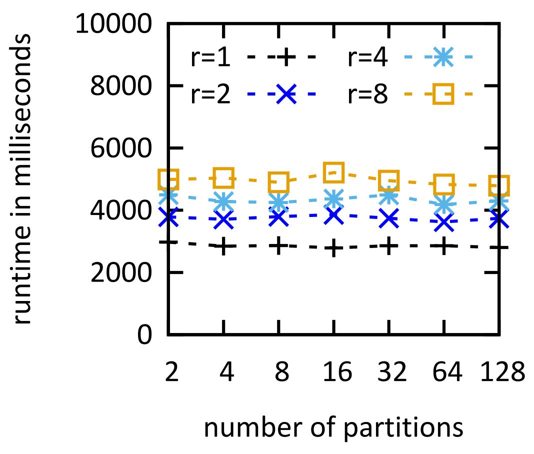
When moving a vertex from the vertex universe to the fringe, we have to be careful in order to *efficiently* select a good vertex. Calculating a score for all vertices in the vertex universe would be much too expensive. For example, suppose we add vertex v_2 to the fringe in the example in Figure 6.4. Suddenly, all vertices in the huge hyperedge e_1 could become fringe candidate vertices for which we would have to calculate the external neighbors score. Note that to calculate the external neighbors score, we must perform set operations that may touch an arbitrary large portion of the global hypergraph.

Our strategy to address this issue involves three steps: (a) select the best fringe candidate vertices from the vertex universe in an *efficient* manner by traversing small hyperedges first, (b) reduce the number of fringe candidate vertices r to $r = 2$, and (c) reduce the computational overhead to calculate the score for a fringe candidate vertex by employing a scoring cache. These three optimizations help us to limit the overhead per decision of which vertex to include into the fringe. Next, we describe the optimizations in detail.

Maximize the chance to select r good fringe candidate vertices First, we describe how we maximize the chance to select good fringe candidate vertices from the vertex universe. The optimal vertex to add to the fringe has minimal external neighbors score, i.e., $argmin_{v \in V'} d_{ext}(v, F_i)$. Vertices that reside in large hyperedges (e.g. e_1 in Figure 6.4) have a high number of neighbors. Hence, it is unlikely that these vertices have a low external neighbors score with respect to the fringe F. For example, in Figure 6.4, vertices v_1 and v_2 have 18 neighbors, whereas vertex v_3 has only 4 neighbors. Thus, vertex v_3 has a much higher chance of being the vertex with the minimal external neighbors score. Based on this observation, we optimize the selection of fringe candidate vertices by ordering all hyperedges that are incident to the fringe F_i with respect to their size and consider only vertices in the smallest hyperedge for inclusion into the fringe (e.g., hyperedges e_4, e_3, e_2 with $|e_4| \leq |e_3| \leq |e_2|$ results in the initial selection of hyperedge e_4). When we cannot retrieve r vertices from the smallest hyperedge (because it does



(a) Quality: (K-1) Metric.



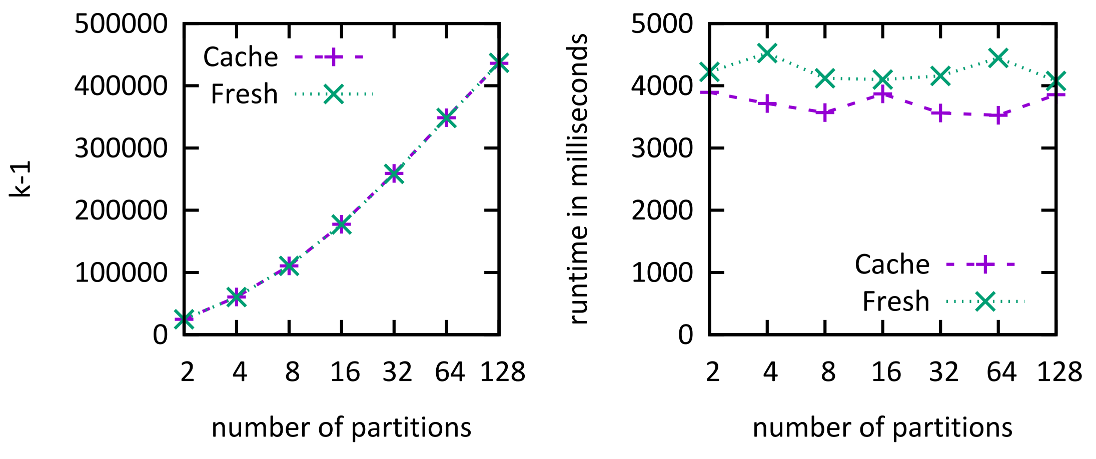
(b) Runtime.

Figure 6.5: Limiting the number of fringe candidate vertices r to $r = 2$ leads to the best partitioning quality (StackOverflow hypergraph).

not contain enough vertices that are not already in C or F), we proceed with the next larger hyperedge.

Reduce the number of fringe candidate vertices to be selected Next, we limit the number of fringe candidate vertices to $r = 2$ vertices. From these r vertices, we select the vertex with the smaller external neighbors score. The basic principle of selecting the best out of two random choices is known in literature as *"the power of two random choices"* [108] and has inspired our design. We experimentally validated that considering more than two options, i.e., $r > 2$, does not significantly improve the decision quality, cf. Figure 6.5. Clearly, the lower the number of fringe candidate vertices r is, the lower is the runtime of the algorithm. Interestingly, using two choices, i.e., $r = 2$ leads to *better* partitioning quality than all other settings of r. Apparently, a higher value for r forces the algorithm to consider fringe candidate vertices from larger hyperedges which distracts the algorithm from the smaller hyperedges.

Reduce the overhead to compute an external neighbors score for fringe candidate vertices The external neighbors score requires calculation of the set intersection between two potentially large sets (see Equation 6.1). This calculation must be done for all fringe candidate vertices. To prevent recomputation, we use a caching mechanism. The score is calculated only when the vertex is accessed for the first time (lazy caching policy). While this means that the cached score may change when including more and more vertices into the fringe, our evaluation results show that partitioning quality stays the same when using caching (see Figure 6.6). But the benefit of reducing score computations improves runtime by up to 20%.



(a) Quality: (K-1) Metric. (b) Runtime.

Figure 6.6: The caching optimization for external neighbors score computation keeps partitioning quality intact while reducing runtime by up to 20% on the Stackoverflow hypergraph.

6.3.3 Balancing Considerations

The default balancing objective of the HYPE algorithm leads to a balanced number of vertices on each partition. For n vertices and k partitions, the algorithm repeats the neighborhood expansion, one vertex at a time, until there are exactly $max = n/k$ vertices per partition. Vertex balancing is the standard method for distributed graph processing systems such as Pregel [80]—considering that the workload per partition is roughly proportional to the number of vertices per partition. Therefore, many practical algorithms such as the popular multilevel k-way hypergraph partitioning algorithm [61] focus on vertex balancing.

However, some applications of hypergraph partitioning may benefit from balancing the sum of vertices *and* hyperedges [4]. More precisely, for n vertices and m hyperedges, the algorithm should partition the hypergraph in a way such that each partition is responsible for $\frac{n+m}{k}$ vertices or hyperedges. In the following, we discuss two ideas how HYPE can achieve this. First, we assign a weight $w(v)$ to each vertex v, i.e., the weight $w(v) = 1 + |E_v|$ with E_v being the set of incident hyperedges of vertex v. Then, we repeat the neighborhood expansion algorithm by assigning vertices until each partition has $max = \frac{n+m}{k}$ total weight (or less). The rationale behind this method is the law of large numbers: it is not likely that a single vertex assignment will suddenly introduce a huge imbalance in relation to the already assigned vertices. Second, to achieve perfect edge balancing, we can flip the hypergraph, i.e., viewing each original vertex as a hyperedge and each original hyperedge as a vertex. When balancing the number of vertices in the flipped graph, we actually balance the number of hyperedges in the original graph. After termination of the algorithm, we flip the hypergraph back to the original representation. We leave an investigation of other balancing constraints as future work.

6.3.4 HYPE Pseudocode

Algorithm 9 HYPE algorithm for hypergraph $G = (V, E)$.

```
1: $V' \leftarrow V$
2: for $i \in [0..k-1]$ do
3:     $C_i \leftarrow \{V'.random()\}$
4:     $V' \leftarrow V' \setminus C_i$
5:     $F_i \leftarrow \{\}$
6:     $c = < key, val > \{\}$ // clear cache
7:     while $|C_i| < \frac{|V|}{k}$ do
8:         UPD8_FRINGE()
9:         UPD8_CORE()
```

Algorithm 9 lists the main loop. We repeat the following method for all partitions $p_i \in P$. After some housekeeping tasks such as filling the core set C_i of partition p_i with a random vertex (line 3), initializing the fringe (line 5), and clearing the $(key, value)$ cache (line 6), we repeat the main loop (lines 7-8) until the core set has exactly $\frac{|V|}{k}$ vertices. The loop body consists of the two functions `upd8_fringe()` and `upd8_core()` that are described next.

Algorithm 10 The function `upd8_fringe()` updates the fringe F_i with vertices from the vertex universe V'.

```
1: function UPD8_FRINGE()
2:
3:     #Determine r fringe candidate vertices
4:     $F_{cand} \leftarrow \{\}$
5:     $X \leftarrow \{e \in E | C_i \cap e \neq \emptyset\}$
6:     $X' \leftarrow [e_0, e_1, ... | e_j \in X \wedge |e_j| > |e_{j-1}|]$
7:     for $e \in X'$ do
8:         for $v \in e \wedge v \notin F_i \wedge \forall j \leq i : v \notin C_j$ do
9:             if $|F_{cand}| < r$ then
10:                $F_{cand} \leftarrow F_{cand} \cup \{v\}$
11:            else
12:                break loop line 7
13:
14:    #Update cache
15:    for $v \in F_{cand} \wedge v \notin c.keys()$ do
16:        $c(v) \leftarrow d_{ext}(v, F_i)$
17:
18:    #Update fringe
19:    $F_i' \leftarrow [v_0, v_1, ... | v_j \in F_i \cup F_{cand} \wedge c(v_{j-1}) < c(v_j)]$
20:    $F_i \leftarrow \{v | v \in F_i'.subsequence(0..s-1)\}$
21:    if $F_i = \emptyset$ then
22:        $F_i \leftarrow \{V'.random()\}$
```

Algorithm 10 lists the `upd8_fringe()` function. The function consists of three steps: determine the r fringe candidate vertices (lines 3-10), update the cache (lines 12-14), and update the fringe (lines 16-20). The first step calculates the fringe candidate vertices F_{cand} by first sorting the hyperedges that are incident to the core set C_i by size (ascending) and then traversing these hyperedges for vertices that can still be added to the fringe (i.e., are not already assigned to the fringe or any core set). The second step updates the cache with the current external neighbors score with respect to the current fringe F_i. The third step sets the fringe to the set of top s vertices with respect to the external neighbors score. If the fringe is still empty after these steps, we initialize it with a random vertex.

Algorithm 11 The function `upd8_core()` updates the core C_i with vertices from the fringe F_i.

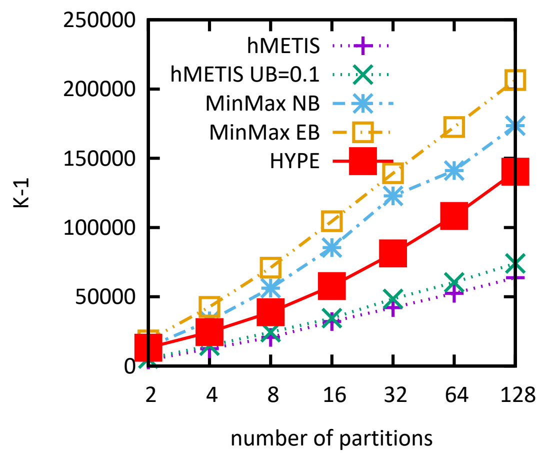
```
function UPD8_CORE()
    $v \leftarrow argmin_{v \in F_i} c(v)$
    $C_i \leftarrow C_i \cup \{v\}$
    $F_i \leftarrow F_i \setminus \{v\}$
    $V' \leftarrow V' \setminus \{v\}$
```

Algorithm 11 lists the `upd8_core()` function. We load the vertex with the minimal cached external neighbors score into the core C_i and remove this vertex from the fringe F_i and the vertex universe V'.

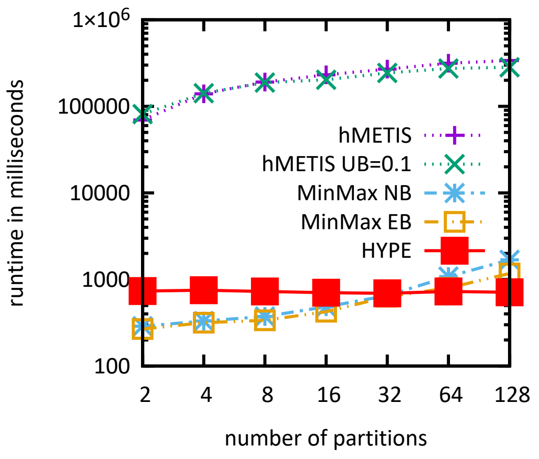
6.3.5 Complexity Analysis

For the following analysis, we denote the number of vertices as $n = |V|$ and the number of hyperedges as $m = |E|$. Algorithm 9 repeats for k partitions the procedure of moving $\frac{n}{k}$ vertices from the vertex universe to the fringe and from the fringe to the core. Next, we analyze the runtime for those procedures `upd8_fringe()` and `upd8_core()`.

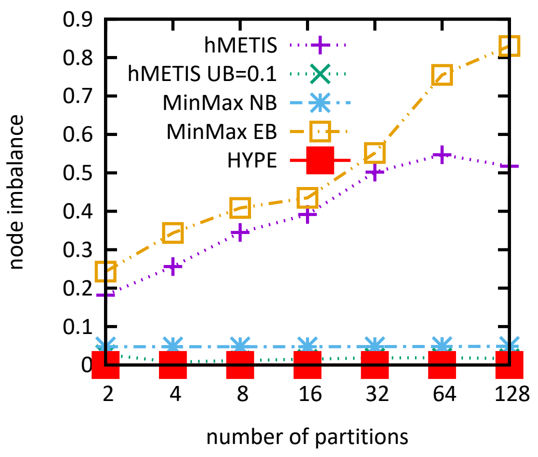
The function `upd8_fringe()` in Algorithm 10 consists of three steps. First, it determines r fringe candidate vertices from the vertex universe (lines 3-12). As we set r to a small constant ($r = 2$), this step is very fast in practice with the following caveat: The algorithm needs to sort the incident hyperedges with respect to the hyperedge size (line 6). The computational complexity of sorting all hyperedges is $O(m * log(m))$. It is sufficient to sort the hyperedges only once in the beginning of the HYPE algorithm. Recap that the algorithm fills the fringe with $r = 2$ new candidate vertices. In the worst case, the fringe is incident to all hyperedges in the hypergraph. Therefore, selecting the fringe candidate vertices is in $O(m)$ to iterate over all hyperedges. ***This is a pessimistic estimation—in practical cases it is sufficient to check the first few smallest hyperedges to find the $r = 2$ candidates***. Second, the algorithm updates the cache with fresh external degree scores for new candidates vertices (lines 14-16). It calculates the external degree score at most once for every candidate vertex (it is just read from cache if needed again later). As there are only $r = 2$ fringe candidate vertices in each execution of `upd8_fringe()`, the total number of external degree score calculations is limited to $2 * n$. The overhead of calculating the external degree of a vertex with respect to a set of s fringe vertices is



(a) Quality: (k-1) Metric.



(b) Runtime.



(c) Balancing.

Figure 6.7: Evaluations on the Github hypergraph (lower is better).

$O(s)$ (cf. Equation 6.1), but s is a constant ($s = 10$). Hence, the total computational complexity of updating the cache is $O(n)$. Third, the algorithm updates the fringe with vertices from the fringe candidates (lines 18-22). As both the fringe and the fringe candidates have constant sizes $r = 2$ and $s = 10$, the complexity of the third step is $O(1)$.

The function `upd8_core()` in Algorithm 11 selects the vertex with minimal cached external neighbors score from the constant-sized fringe. Thus, the complexity is $O(1)$ including the housekeeping tasks in lines 3-5.

In total, the worst-case computational complexity of the HYPE algorithm is $O(m * log(m) + k * \frac{n}{k} * m + n)) = O(m * log(m) + n * m)$. As highlighted above, in practice we observe that only a small, constant number of hyperedges is checked in order to find the r candidate vertices, so that we observe a complexity of $O(m * log(m) + n)$.

6.4 Evaluations

In this section, we evaluate the performance of HYPE on several real-world hypergraphs.

Experimental Setup All experiments were performed on a shared memory machine with 4 x Intel(R) Xeon(R) CPU E7-4850 v4 @ 2.10GHz (4 x 16 cores) with 1 TB RAM. The source code of our HYPE partitioner is written in C++.

Hypergraph Data Sets We perform the experiments on different real-world hypergraphs, i.e., Github [69], StackOverflow [69], Reddit and Reddit-L[1], as listed in Table 6.2. All of the hypergraphs show a power law distribution of vertex and hyperedge degrees. In addition to the number of vertices and hyperedges, we report the number of edges. An edge represents an assignment of a vertex to a hyperedge.

We highlight that for this work, we crawled two large real-world hypergraphs from the Reddit dataset using the relations between authors, subreddits and comments.

Benchmarks We choose our benchmarks for evaluating HYPE based on 3 categories.

Group (I) consists of a wide range of hierarchical partitioners such as hMetis [61], Mondriaan [132], Parkway [128], PaToH [14], Zoltan [26], and KaHyPar [47]. As no partitioner in group (I) consistently outperforms the other partitioners in terms of partitioning quality, scalability and partitioning time, we decided for the well-established and widely used hypergraph partitioner hMETIS. Several recent papers show that hMETIS leads to competitive partitioning performance with respect to the $(k-1)$ metric [4, 54, 145]. Hence, we chose hMETIS as the representative partitioner from group (I) in this work. We run hMETIS in two different settings: with and without enforced vertex balancing. Due to the high partitioning quality, hMETIS serves as the benchmark for partitioning quality on small to medium hypergraphs. We used hMETIS in version 2.0pre1 with the parameters `-ptype=rb -otype=soed -reconst`. To enforce vertex balancing, we set[2] the parameter `UB=0.1`.

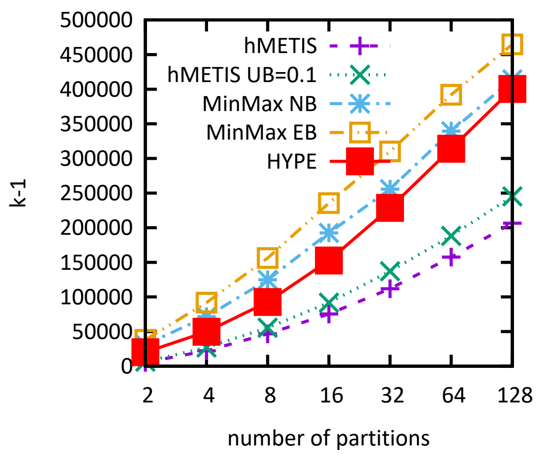
Group (II) consists of the recently proposed Social Hash Partitioner (SHP) [54]. The authors released the raw source code of SHP. Yet, we could not reproduce their results as neither configuration files and parameters, nor scripts, execution instructions, or documentations were

[1]https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/

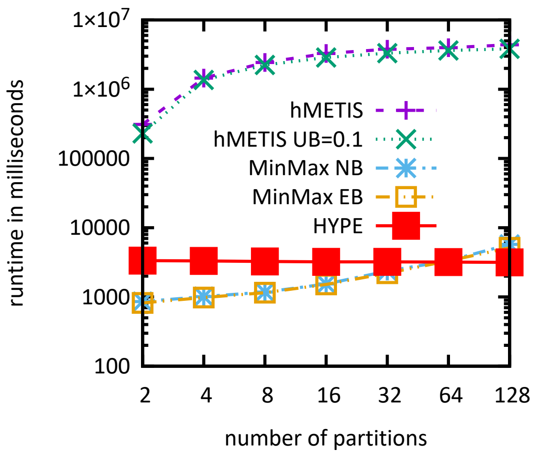
[2]We determined the UB parameter experimentally such that the measured imbalance is comparable to HYPE and MinMax. Schlag et al. [119] provide an equation to set UB in order to enforce a specific imbalance constraint at a given k.

provided. However, in our evaluations we partitioned hypergraphs of similar size as SHP in a similar runtime, even though HYPE uses a purely sequential partitioning algorithm.

Group (III) comprises multiple streaming partitioning strategies proposed by Alistarh et al. [4]. The greedy MinMax strategy constantly outperformed all other strategies according to their paper[3]. Moreover, we designed a novel vertex-balanced variant of MinMax that outperforms the original approach with respect to the (k-1) metric[4]. We denote this variant as *MinMax NB* (**n**ode **b**alanced) in contrast to the standard *MinMax EB* (**e**dge **b**alanced).



(a) Quality: (k-1) Metric.



(b) Runtime.

Figure 6.8: Evaluations on the StackOverflow hypergraph (lower is better).

Experiments In all experiments, we capture the following metrics. (1) The (k-1) metric to evaluate the *quality* of the hypergraph partitioning. This is the default metric for partitioning quality [26, 61]. Other partitioning quality metrics such as the *hyperedge-cut* and the *sum of external degree* performed similar in our experiments[5]. (2) The runtime of the algorithm to partition the whole input hypergraph in order to evaluate the *speed* of the partitioning algorithms. (3) The vertex imbalance as a metric to capture the *fairness* of the hypergraph partitioning. We compute vertex imbalance as the normalized deviation between the maximal and the minimal number of vertices assigned to any partition, i.e., $\frac{maxsize - minsize}{maxsize}$.

For each experiment, we increase the number of partitions from 2 up to 128 in exponential steps.

[3]The optimization goal of MinMax is to minimize the maximum number of hyperedges associated with a partition via the partition's vertices. However, both metrics MinMax and (k-1) are closely related: They measure the spread of hyperedges across partitions.

[4]We allowed a slack parameter of up to 100 vertices, cf. [4].

[5]The close relationship between these metrics is described in literature [54].

6.4.1 Performance Evaluations

The performance evaluations show the benefits of HYPE when partitioning large hypergraphs. Its runtime is independent of the number of partitions, so that it is faster than streaming partitioning when the number of partitions is large. Further, the hierarchical partitioning algorithm hMETIS does not scale to very large hypergraphs, i.e., it cannot partition the Reddit hypergraph, and takes orders of magnitude longer for the smaller hypergraphs. We discuss the detailed results next.

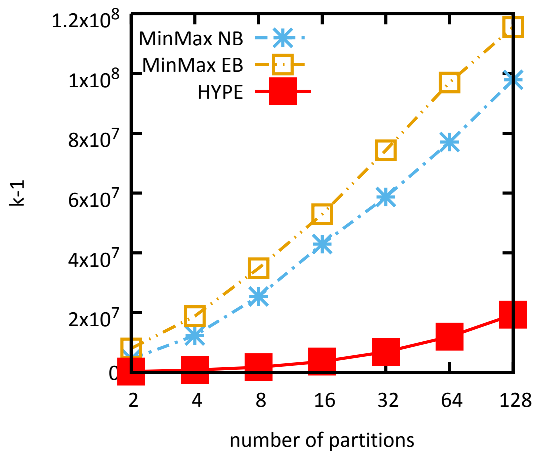
Github Figure 6.7a shows the partitioning quality on the Github hypergraph. In the (k-1) metric, hMETIS performs best (e.g., 47 % better than HYPE at k = 128). HYPE performs up to 45% better than MinMax hyperedge-balanced, and up to 34 % better than MinMax vertex-balanced.

Partitioning runtime is depicted in Figure 6.7b. hMETIS took 70 to 338 seconds to partition the Github hypergraph, which is orders of magnitude slower than MinMax and HYPE (up to 476 $\times$ slower than HYPE). The partitioning runtime of HYPE is independent of the number of partitions, as each partition is filled with vertices sequentially until it is full. Different from that, in MinMax, the partitioning runtime depends on the number of partitions, as MinMax works with a scoring function that computes for each vertex a score for each partition and then assigns the vertex to the partition where its score is best. Hence, for up to 32 partitions, HYPE has a higher runtime than MinMax, up to 2.7 $\times$ higher, whereas for 64 and 128 partitions, the runtime of HYPE is lower (up to 2.4 $\times$ lower).

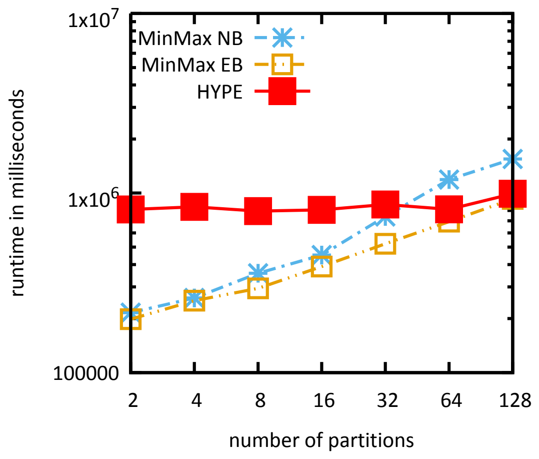
In terms of balancing, cf. Figure 6.7c, HYPE shows perfect vertex balancing, while the MinMax vertex-balanced has a slight imbalance of up to 5%. Unsurprisingly, MinMax hyperedge-balanced has a poor vertex balancing. In hMETIS, when vertex balancing is turned on, the maximum imbalance was 3%, whereas without that flag, vertex imbalance was tremedously higher (up to 55% imbalance). The balancing results are similar for all other hypergraphs, so we will not discuss balancing in the following results any more.

StackOverflow Figure 6.8a shows the partitioning quality on the StackOverflow hypergraph. In the (k-1) metric, hMETIS performs best (e.g., 39 % better than HYPE at k = 128). HYPE performs up to 47% better than MinMax hyperedge-balanced, and up to 35% better than MinMax vertex-balanced.

Partitioning runtime is depicted in Figure 6.8b. hMETIS took between 206 and 4374 seconds (i.e., 73 minutes) to partition the StackOverflow hypergraph, which is orders of magnitude slower than MinMax and HYPE (up to 1,384 $\times$ slower than HYPE). In numbers, HYPE takes 3 seconds to build 128 partitions, while hMETIS takes more than 1 hour! The comparison



(a) Quality: (K-1) Metric.



(b) Runtime.

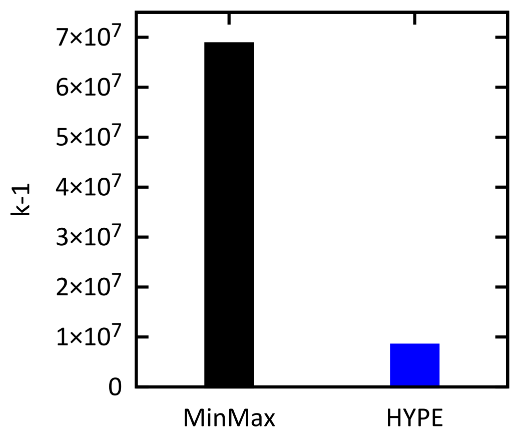
Figure 6.9: Evaluations on the Reddit hypergraph (lower is better).

between HYPE and MinMax on StackOverflow leads to similar results as on the Github hypergraph: With up to 32 partitions, MinMax is faster (up to 4.1 $\times$ faster), but with 64 and 128 partitions, HYPE is faster (up to 1.8 $\times$ faster). Again, the reason for this is that the HYPE runtime is independent of the number of partitions.

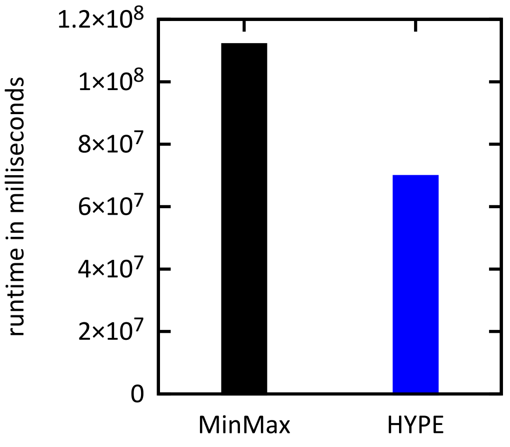
Reddit Figure 6.9a shows the partitioning quality on the Reddit hypergraph. For this hypergraph, we could not produce results with the hMETIS partitioner, as it crashed or was running for days without returning a result. Consistent to the experiments reported in [54], many partitioners from group (I) are not able to partition such large hypergraphs. Hence, in the following, we only report results for MinMax and HYPE.

On the Reddit hypergraph, the advantage of exploiting local communities in HYPE pays off to the full extent: HYPE outperforms the streaming partitioner MinMax, that ignores the overall hypergraph structure, by orders of magnitude. For 2, 4 and 8 partitions, HYPE achieved an improvement of 95% compared to MinMax hyperedge-balanced, and 93% compared to MinMax vertex-balanced in the (k-1) metric. Thus, HYPE leads to a partitioning quality that is up to **20** $\times$ **better** than when using MinMax. For 16 partitions, HYPE performs 93% and 91% better, for 32 partitions 91% and 88%, for 64 partitions 88% and 84%, and for 128 partitions 83% and 80% better than MinMax hyperedge-balanced and MinMax vertex-balanced partitioners, respectively.

Comparing the partitioning runtime of HYPE and MinMax in Figure 6.9b, we see again that the runtime of HYPE is independent of the number of partitions, whereas MinMax has a higher runtime with growing number of partitions because of its scoring scheme. While at 2 partitions, MinMax is up to 4 $\times$ faster than HYPE, at 64 partitions HYPE becomes faster than MinMax, by



(a) Quality: (k-1) Metric.



(b) Runtime.

Figure 6.10: Evaluations on the Reddit-L hypergraph (lower is better).

up to 36% at 128 partitions. As with the other hypergraphs, MinMax vertex-balanced is slightly slower than MinMax hyperedge-balanced as the hyperedge balance can change significantly after assigning a single vertex. This often forces assignment to a single partition (the least loaded) such that the partitions remain balanced. In such cases, the forced partitioning decisions for hyperedge-balanced partitioning can be performed very quickly.

Reddit-L In Figure 6.10, we compare partitioning quality and runtime of HYPE against MinMax on the large Reddit-L hypergraph with $k = 128$ partitions. MinMax requires more than 31 hours to partition Reddit-L compared to the 19 hours of HYPE. Although being 39% faster than MinMax, HYPE still outperforms MinMax in partitioning quality by 88%: MinMax has a (k-1) score of 68,709,969 compared to HYPE's 8,357,200. Note that MinMax already belongs to the fastest high-quality partitioners. However, HYPE is able to outperform MinMax because its runtime does not depend on the number of partitions.

6.4.2 Discussion of the Results

We conclude that HYPE shows very promising performance in hypergraph partitioning. First, it is able to partition very large hypergraphs, which cannot be partitioned by algorithms from group (I). Second, it consistently provides better partitioning quality than streaming MinMax. On top of that, the HYPE algorithm is comparably easy to implement and to manage because all system parameters are fixed.

6.5 Related Work

In the last decades, research on hypergraph partitioning was driven by the need to place transistors on chips in Very-large-scale integration (VLSI) design [58], as logic circuits can be modeled as large hypergraphs that are divided among chips. The most popular hypergraph partitioning algorithm from that area is hMETIS [61] which is based on a multilevel contraction algorithm and produces good partitioning quality for medium-sized hypergraphs in the magnitude of up to 100,000 edges.

However, multilevel partitioning algorithms do not scale to large hypergraphs, as shown in our evaluations. Parallel implementations of multilevel partitioning have been proposed [26], but the problem of high computational complexity and memory consumption remains. For instance, *Zoltan* [26] is a parallel multilevel hypergraph partitioning algorithm. The evaluated graphs on Zoltan are relatively small—within a magnitude of up to 30 million edges or less—while using up to 64 parallel machines to process them. Other algorithms of that group are Mondriaan [132], Parkway [128], PaToH [14], and KaHyPar [47]. For hypergraphs with hundreds of millions of edges, these algorithms are not practical as they take hours or even days to complete, if they terminate at all.

The bad scalability of multilevel partitioning algorithms led to the development of more scalable partitioners. *Social Hash Partitioner (SHP)* achieves scalability to very large hypergraphs (up to 10 billion edges) by means of massive parallelization [54]. SHP performs random swaps of vertices between partitions and greedily chooses the best swaps. Random swaps fit well with the objective of parallelization and distribution in SHP, but may not be the most efficient heuristic. Investigating on the phenomenon of scalability versus efficiency [91], we conceived the idea to devise an efficient hypergraph partitioning algorithm.

Another approach to partition very large hypergraphs are *streaming* algorithms. Streaming hypergraph partitioning takes one vertex at a time from a stream of vertices, and calculates a score for each possible placement of that vertex on each of the partitions. The vertex is then placed on the partition where its placement score is best, and cannot be removed any more. Alistarh et al. [4] proposed different heuristic scoring functions, where greedily assigning vertices to the partition with the largest overlap of incident hyperedges is considered best. There are two issues with the streaming approach. First, by only taking into account a single vertex at a time and placing it, information about the neighborhood of that vertex is not exploited although available in the hypergraph. Second, the complexity of the algorithm depends on the number of partitions, as the scoring function is computed for each vertex on each partition. For a large number of partitions, streaming partitioning becomes slow.

A closely related problem is *balanced k-way graph partitioning* which faces similar challenges such as billion-scale graph data and the need for fast algorithms. Multilevel graph partitioners such as METIS [59] and ParMETIS [60] do not scale very well. Spinner [81] is a highly scalable graph partitioner that, like SHP, performs iterative random permutations and greedy selection of the best permutation. There is a large number of streaming graph partitioning algorithms, such as HDRF [101], H-load [87], and ADWISE [85]. The "neighborhood heuristic" by Zhang et al. [150] follows a completely different approach by exploiting the graph structure when performing partitioning decisions. The algorithm grows a core set by successively adding neighbors of the core set to a fringe set. However, the given heuristic can not be applied directly to hypergraph partitioning as the calculation of scores is way too expensive in hypergraphs (see Section 6.3). While hypergraphs can be transformed into bipartite graphs, graph partitioning algorithms cannot be used to perform hypergraph partitioning. First, the bipartite graph representations contain one artificial vertex per hyperedge that destroys the vertex balancing requirement of hypergraph partitioning. Second, the (k-1) metric is ignored by graph partitioning algorithms.

In recent years, several distributed hypergraph systems emerged that fueled the need for efficient massive hypergraph partitioning. These systems are inspired from the area of distributed graph processing systems and apply the vertex-centric programming model from graph processing to hypergraph processing. For instance, HyperX [49] allows applications to specify vertex and hyperedge programs which are then executed iteratively by the system. Also, Mesh [44] builds upon the popular GraphX system [35] and shows promising performance. These systems show significant reduction of processing latency with improved partitioning quality.

6.6 Chapter Summary

In this chapter, we propose HYPE, an effective and efficient partitioner for real-world hypergraphs. The partitioner grows k core sets in a sequential manner using a neighborhood expansion algorithm with several optimizations to reduce the search space. Due to the simplicity of the design and focus on the hypergraph structure, HYPE is able to partition the large Reddit hypergraph with billions of edges in less than a day. This is the partitioning of one of the largest real-world hypergraph reported in literature. HYPE not only improves partitioning quality by up to 95% compared to streaming hypergraph partitioning, but *reduces* runtime as well by 39%.

A promising line of future research on HYPE is to explore how to grow the k core sets in parallel. In this scenario, several core sets *"compete"* for inclusion of attractive vertices, so the crucial questions are how to minimize the number of "collisions" and how to deal with collisions when they happen.

7

Summary and Future Work

In this chapter, we provide a summary of contributions, draw conclusions, and propose future work in the area of graph partitioning for distributed graph processing.

7.1 Thesis Summary

Modern graph processing systems have used general partitioning algorithms to minimize communication and synchronization overhead in a distributed environment. In this thesis, we show that—by tailoring the partitioning algorithms to the specific domain of distributed graph processing on real-world data sets—we can significantly improve partitioning quality which reduces costs, communication volume, and processing latency. In particular, we summarize our contributions in the following.

- Existing vertex-cut partitioning algorithms minimize the replication degree in order to minimize communication overhead. However, in Chapter 3, we provide evidence that the implicit *homogeneity assumptions* of vertex traffic and network costs are not valid for modern graph processing. Instead, vertex traffic and network costs are heterogeneous. In fact, following a power-law distribution, heterogeneity of vertex traffic is so prominent that the reader might be surprised about the homogeneity assumption. The vertices with top 20% of vertex traffic contribute 70-99% of total traffic for many standard graph algorithms such as PageRank, subgraph isomorphism, and cellular automaton. To address this issue, we proposed the graph processing system GrapH that takes heterogeneous vertex traffic and network costs into account. The system dynamically tracks past and predicts future vertex traffic and adapts the partitioning at runtime to minimize future *expected communication costs*. To calculate these expected communication costs, it considers the *investment costs* of migrating graph data between partitions. In our evaluations, we show that this integrated approach of graph processing, vertex traffic monitoring and prediction, and repartitioning, outperforms state-of-the-art by up to 60% with respect to communication costs, while improving end-to-end latency of graph computation by more than 10%.

- A multitude of single-edge streaming partitioning emerged in recent years. The reason for their popularity is their fast partitioning speed—they are able to solve the NP-hard partitioning problem heuristically in linear runtime (in the graph size) while providing reasonable quality. However, in Chapter 4, we show that this practice of investing minimal time into graph partitioning harms end-to-end latency of partitioning plus graph processing. With the ADWISE algorithm, we made the trade-off between partitioning latency and graph processing latency controllable. This allows us to invest more time into partitioning to improve partitioning quality and ultimately reduce graph processing latency. The evaluation results show that ADWISE reduces total end-to-end latency by up to $23-47\%$ compared to single-edge streaming in different realistic scenarios. On top of that, we provide a small but highly effective optimization that can be applied to any single-edge streaming partitioning algorithm which reduces replication degree by $3-4\times$ without introducing additional computational overhead.

- An emerging class of applications, which we denote as *concurrent graph query analytics* (`CGA`), demands support of localized query access patterns. Here, the graph queries do not access the global graph but only a local area of the graph. At the same time, many queries run in parallel on the shared graph structure. Yet, traditional graph partitioning algorithms are oblivious to the graph queries which leads to suboptimal partitioning performance. In Chapter 5, we present novel algorithms tailored to `CGA` applications for more efficient communication, synchronization, and workload balancing. In particular, we developed a scalable method of managing centralized knowledge about query workload to perform query-aware adaptive partitioning. Moreover, we provide a novel concept of hybrid barrier synchronization observing a speedup of average query latency by up to $2.2\times$.

- Many important real-world applications—such as social networks or distributed data bases—can be modeled as hypergraphs. In such a model, vertices represent entities—such as users or data records—whereas hyperedges model a group membership of the vertices—such as the authorship in a specific topic or the membership of a data record in a specific replicated shard. To optimize such applications, we need an efficient and effective solution to the NP-hard balanced k-way hypergraph partitioning problem. However, existing hypergraph partitioners that scale to very large graphs do not effectively exploit the hypergraph structure when performing the partitioning decisions. In Chapter 6, we propose HYPE, a hypergraph partitioner that exploits the neighborhood relations between vertices in the hypergraph using an efficient implementation of neighborhood expansion. HYPE improves partitioning quality by up to 95% and reduces runtime by up to 39% compared to streaming partitioning.

7.2 Conclusions

In this thesis, we have examined four independent research directions that all support our main hypothesis: tailoring graph partitioning to the specific area of distributed graph processing—and hypergraph partitioning to the specific properties of real-world graphs—opens the road for many improvements of key performance metrics such as communication costs, processing latency, and workload balancing. In specific, we show this in four dimensions: our algorithms consider (i) heterogeneities of vertex traffic and network communication (Chapter 3), (ii) the trade-off between partitioning latency and graph processing latency (Chapter 4), (iii) dynamic workload of localized graph queries (Chapter 5), and (iv) the skeweness of large real-world hypergraphs (Chapter 6).

Clearly, these *considerations of application specifics* come at a cost of more computational work or memory footprint. In particular, (i) considering vertex traffic requires to maintain a vertex traffic value for each vertex, (ii) investing more partitioning latency does not guarantee to reduce later graph processing latency, (iii) repartitioning the graph reactively based on the query workload can lead to oscillations and migration overhead when query workload can not predicted adequately, and (iv) considering the hyperedge and hypervertex degree causes additional maintenance overhead for the algorithm.

However, we have shown in this thesis that the benefits outweigh the costs significantly which leads to much more accurate partitioning decisions. In our experience, workload patterns in the area of distributed graph processing tend to repeat which offers a reasonable amount of predictability. For example, (i) vertex traffic remains surprisingly stable—we have shown that even simple prediction methods are able to catch these patterns well. Moreover, (ii) even for modestly complex graph algorithms such as PageRank, it pays off to invest three times the minimal partitioning latency. Additionally, (iii) query workload hotspots change slowly and query-centric partitioning is an excellent proxy for fine-grained vertex-centric partitioning leading to improved partitioning quality at much smaller runtime complexity. Finally, (iv) considering hypervertex degree and hyperedge degree *reduces* runtime overhead for many practical hypergraphs because the neighborhood heuristic avoids expansion in the computationally expensive regions of the hypergraph (high-degree hypervertices and hyperedges). In conclusion, the algorithms presented in this thesis improved efficiency of distributed graph processing, graph partitioning, and hypergraph partitioning by tailoring the partitioning towards their specific application domains. In particular, we show that (i) considering dynamic traffic patterns lead to much better partitioning quality, (ii) investing more time into partitioning reduces integrated latency of partitioning plus graph processing, (iii) lifting the partitioning algorithms from the vertex level to the query level reduces partitioning overhead and improves partitioning quality, and (iv) considering the skewed degree distribution improves efficiency of hypergraph partitioning.

7.3 Future Work

The presented algorithms constitute only the first few steps towards efficient distributed graph processing and hypergraph partitioning—when considering the wide range of application domains in these areas. Novel application areas arise frequently such as graph processing on GPUs or modeling complex machine learning applications as distributed data flow graphs. An example for the latter is the popular TensorFlow system [1] proposed by Google for distributed (deep) machine learning. For TensorFlow, we propose several fast heuristics for partitioning and scheduling of distributed data flow graphs [88]. However, a detailed analysis of this problem is still an open research question.

Many different paths for optimization are still unexplored (or under-explored) in the area of graph partitioning for distributed graph processing. One such optimization is to perform *partial replication* to increase access locality and scalability at the cost of a larger memory footprint and synchronization overhead. Other ideas include highly-parallelized graph partitioning on a GPU, the use of meta-heuristics and automatic heuristics selection, and using machine learning to learn the scoring function of streaming partitioning algorithms (when partitioning large graphs, we can have billions of training data items to learn a prediction model).

To wrap up, this thesis provides a first step towards efficient processing of graph-structured data sets by proposing new domain-specific partitioning algorithms.

Bibliography

[1] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[2] T. Aittokallio and B. Schwikowski. Graph-based methods for analysing networks in cell biology. *Briefings in Bioinformatics*, 7(3):243–255, 2006.

[3] R. Albert, H. Jeong, and A.-L. Barabási. Error and attack tolerance of complex networks. *Nature*, 406(6794):378, 2000.

[4] D. Alistarh, J. Iglesias, and M. Vojnovic. Streaming min-max hypergraph partitioning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS) - Volume 2*, 2015.

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

[6] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[7] A. Arora, S. Galhotra, and S. Ranu. Debunking the myths of influence maximization: An in-depth benchmarking study. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 651–666. ACM, 2017.

[8] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[9] C. Bahnmüller. Improving google's open-source machine learning system tensorflow. Master's thesis, 2018.

[10] T. Y. Berger-Wolf and J. Saia. A framework for analysis of dynamic social networks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528. ACM, 2006.

[11] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web (WWW)*, pages 587–596. ACM, 2011.

[12] P. S. Bradley and U. M. Fayyad. Refining initial points for k-means clustering. In *ICML*, volume 98, pages 91–99. Citeseer, 1998.

[13] F. Cacheda, N. Barbieri, and R. Blanco. Click through rate prediction for local search results. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM)*, pages 171–180. ACM, 2017.

[14] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 10(7):673–693, Jul 1999.

[15] U. V. Çatalyürek, K. Kaya, and B. Uçar. Integrated data placement and task assignment for scientific workflows in clouds. In *Proceedings of the 4th International Workshop on Data-intensive Distributed Computing*, 2011.

[16] Ò. Celma and P. Lamere. If you like the beatles you might like...: a tutorial on music recommendation. In *Proceedings of the 16th ACM International Conference on Multimedia*, pages 1157–1158. ACM, 2008.

[17] M. Cha, A. Mislove, and K. P. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pages 721–730. ACM, 2009.

[18] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*, page 1. ACM, 2015.

[19] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 3. ACM, 2012.

[20] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, pages 85–98. ACM, 2012.

[21] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[22] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web (TWEB)*, 4(4):16, 2010.

[23] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[24] M. Das, A. Simitsis, and K. Wilkinson. A hybrid solution for mixed workloads on dynamic graphs. In *Fourth International Workshop on Graph Data Management Experiences and Systems (GRADES)*, page 1, 2016.

[25] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1223–1231, 2012.

[26] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th International Parallel Distributed Processing Symposium*, 2006.

[27] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment*, 9(11), 2016.

[28] R. Eigner and G. Lutz. Collision avoidance in vanets-an application for ontological context models. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 412–416. IEEE, 2008.

[29] L. Epple. Billion-scale hypergraph partitioning. Master's thesis, University of Stuttgart, 2018.

[30] U. Feige, M. Hajiaghayi, and J. R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM Journal on Computing*, 2008.

[31] S. Galhotra, A. Arora, S. Virinchi, and S. Roy. Asim: A scalable algorithm for influence maximization under the independent cascade model. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, pages 35–36. ACM, 2015.

[32] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[33] H. Geppert. Scalable hypergraph partitioning. B.S. thesis, 2017.

[34] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[35] 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 *OSDI*, 2014.

[36] J. Grunert. Concurrent query analytics on distributed graph systems. Master's thesis, 2017.

[37] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM Computer Communication Review (CCR)*, 2015.

[38] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, pages 505–514. ACM, 2013.

[39] G. Hamerly and C. Elkan. Learning the k in k-means. In *Advances in Neural Information Processing Systems (NIPS)*, pages 281–288, 2004.

[40] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[41] P. B. Hansen. Parallel cellular automata: A model program for computational science. *Concurrency: Practice and Experience*, 1993.

[42] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[43] M. Hauck, M. Paradies, and H. Fröning. Can modern graph processing engines run concurrent queries efficiently? In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, page 5. ACM, 2017.

[44] B. Heintz, S. Singh, C. Tesdahl, and A. Chandra. Mesh: A flexible distributed hypergraph processing system. 2016.

[45] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[46] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Exper.*, 2014.

[47] T. Heuer and S. Schlag. Improving coarsening schemes for hypergraph partitioning by exploiting community structure. In *16th International Symposium on Experimental Algorithms, (SEA 2017)*, 2017.

[48] J. Huang and D. J. Abadi. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment*, 9(7):540–551, 2016.

[49] J. Huang, R. Zhang, and J. X. Yu. Scalable hypergraph learning and processing. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2015.

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

[51] C. Jayalath, J. Stephen, and P. Eugster. From the cloud to the atmosphere: Running mapreduce across data centers. *IEEE Transactions on Computers (ToC)*, 2014.

[52] Y. Jing and S. Baluja. Visualrank: Applying pagerank to large-scale image search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1877–1890, 2008.

[53] S. Jo, J. Yoo, and U. Kang. Fast and scalable distributed loopy belief propagation on real-world graphs. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM)*, pages 297–305. ACM, 2018.

[54] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, and A. Shalita. Social hash partitioner: a scalable distributed hypergraph partitioner. *Proceedings of the VLDB Endowment*, 10(11):1418–1429, 2017.

[55] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, et al. Streaming graph challenge: Stochastic block partition. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–12. IEEE, 2017.

[56] D. R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[57] J. Kari. Theory of cellular automata: A survey. *Theoretical computer science*, 334(1-3):3–33, 2005.

[58] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[59] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[60] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[61] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, 1999.

[62] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *Data Engineering (ICDE), 2008 IEEE 24th International Conference on*, pages 953–962, April 2008.

[63] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 137–146. ACM, 2003.

[64] A. Khan and S. Elnikety. Systems for big-graphs. *Proceedings of the VLDB Endowment*, 7(13), 2014.

[65] S. S. Khan and A. Ahmad. Cluster center initialization algorithm for k-means clustering. *Pattern recognition letters*, 25(11):1293–1302, 2004.

[66] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[67] G. Koutrika and Y. Ioannidis. Personalization of queries in database systems. In *Data Engineering (ICDE), 2004 IEEE International Conference on*, pages 597–608. IEEE, 2004.

[68] D. Kumar, A. Raj, D. Patra, and D. Janakiram. Graphive: Heterogeneity-aware adaptive graph partitioning in graphlab. In *ICCPW*, 2014.

[69] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, pages 1343–1350. ACM, 2013.

[70] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, 2013.

[71] L. Laich. Graph partitioning and scheduling for distributed dataflow computation. B.S. thesis, 2017.

[72] A. Lenharth, D. Nguyen, and K. Pingali. Parallel graph analytics. *Communications of the ACM*, 59(5):78–87, 2016.

[73] 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*, 2009.

[74] C. Li. Distributed data analytics using graph processing frameworks. Master's thesis, University of Stuttgart, 2015.

[75] D. Li, C. Zhang, J. Wang, Z. Zhang, and Y. Zhang. Grapha: Adaptive partitioning for natural graphs. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2358–2365. IEEE, 2017.

[76] H. R. Lourenço, O. C. Martin, and T. Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[77] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[78] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 2014.

[79] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *Proceedings of the VLDB Endowment*, 5(4):310–321, 2011.

[80] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[81] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 1083–1094. Ieee, 2017.

[82] C. Mayer, R. Mayer, and M. Abdo. Stream learner: Distributed incremental machine learning on event streams: Grand challenge. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS)*, DEBS '17, pages 298–303, New York, NY, USA, 2017. ACM.

[83] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel. Hype: Massive hypergraph partitioning with neighborhood expansion. In *Big Data (Big Data), 2018 IEEE International Conference on*, 2018.

[84] C. Mayer, R. Mayer, J. Grunert, A. Tariq, and K. Rothermel. Q-graph: Preserving query locality in multitenant graph processing. In *1st Joint International Workshop on*

Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) 2018, page 7. ACM, 2018.

[85] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, and K. Rothermel. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *Distributed Computing Systems (ICDCS), 2018 IEEE 38th International Conference on*, 2018.

[86] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. Graph: Heterogeneity-aware graph computation with adaptive partitioning. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 118–128. IEEE, 2016.

[87] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel. Graph: Traffic-aware graph processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2018.

[88] R. Mayer, C. Mayer, and L. Laich. The tensorflow partitioning and scheduling problem: it's the critical path! In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*, pages 1–6. ACM, 2017.

[89] R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel. Graphcep: Real-time data analytics using parallel complex event and graph processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 309–316. ACM, 2016.

[90] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 2015.

[91] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, 2015.

[92] S. Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):62:1–62:33, Mar. 2016.

[93] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *International Workshop on Experimental and Efficient Algorithms*, pages 189–202. Springer, 2005.

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

[95] J. Nishimura and J. Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference Knowledge discovery and data mining*, 2013.

[96] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[97] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24(3):515–554, 2012.

[98] J. M. Pena, J. A. Lozano, and P. Larranaga. An empirical comparison of four initialization methods for the k-means algorithm. *Pattern recognition letters*, 20(10):1027–1040, 1999.

[99] C. Peng, M. Kim, Z. Zhang, and H. Lei. Vdn: Virtual machine image distribution network for cloud data centers. In *INFOCOM*, 2012.

[100] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[101] 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 (CIKM)*, pages 243–252. ACM, 2015.

[102] C. Pizzuti. Ga-net: A genetic algorithm for community detection in social networks. In *International Conference on Parallel Problem Solving from Nature*, pages 1081–1090. Springer, 2008.

[103] A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey. High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd International Conference on World Wide Web (WWW)*, pages 225–236. ACM, 2014.

[104] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[105] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, 2016.

[106] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi. Distributed vertex-cut partitioning. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 186–200. Springer, 2014.

[107] Z. Riaz, F. Dürr, and K. Rothermel. Optimized location update protocols for secure and efficient position sharing. In *International Conference and Workshops on Networked Systems*, pages 1–8, 2015.

[108] A. W. Richa, M. Mitzenmacher, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[109] L. Rieger. Distributed graph partitioning for large-scale graph analytics. Master's thesis, University of Stuttgart, 2016.

[110] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[111] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424. ACM, 2015.

[112] J. Ruan and W. Zhang. An efficient spectral algorithm for network community discovery and its applications to biological and social networks. In *Data Mining (ICDM 2007), Seventh IEEE International Conference on*, pages 643–648. IEEE, 2007.

[113] H. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi. Boosting vertex-cut partitioning for streaming graphs. In *BigData Congress*, 2016.

[114] S. Sakr and G. Al-Naymat. Relational processing of rdf queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, 2010.

[115] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013.

[116] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proceedings of the VLDB Endowment*, 2013.

[117] V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: applications to community discovery. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 737–746. ACM, 2009.

[118] P. Schäfer. Finding relevant videos in big data environments-how to utilize graph processing systems for video retrieval. Master's thesis, 2017.

[119] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67. SIAM, 2016.

[120] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *Data Engineering (ICDE), 2013 IEEE International Conference on*, 2013.

[121] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[122] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[123] 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*, pages 1222–1230. ACM, 2012.

[124] T. Stützle. Iterated local search for the quadratic assignment problem. *European Journal of Operational Research*, 174(3):1519–1539, 2006.

[125] T. Suzumura, C. Houngkaew, and H. Kanezashi. Towards billion-scale social simulations. In *Simulation Conference (WSC), 2014 Winter*, pages 781–792. IEEE, 2014.

[126] P. Symeonidis, E. Tiakas, and Y. Manolopoulos. Product recommendation and rating prediction based on multi-modal social networks. In *Proceedings of the Fifth ACM conference on Recommender systems*, pages 61–68. ACM, 2011.

[127] K. Ten Tusscher, D. Noble, P. Noble, and A. Panfilov. A model for human ventricular tissue. *American Journal of Physiology-Heart and Circulatory Physiology*, 2004.

[128] A. Trifunovic and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563 – 581, 2008.

[129] 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 (WSDM)*, 2014.

[130] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[131] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 144–153. IEEE, 2014.

[132] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47(1):67–95, 2005.

[133] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proceedings of the VLDB Endowment*, 10(5):493–504, 2017.

[134] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. Wanalytics: Analytics for a geo-distributed data-intensive world. *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR) 2015*, 2015.

[135] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *Data Engineering (ICDE), 2014 IEEE International Conference on*, 2014.

[136] D. J. Watts and S. H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.

[137] T. Weiss. Experimental comparison of distributed graph processing systems. B.S. thesis, 2018.

[138] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, SoCC '15, pages 408–421, New York, NY, USA, 2015. ACM.

[139] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 74–85. IEEE, 2013.

[140] 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 (NIPS)*, 2014.

[141] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES)*, page 2. ACM, 2013.

[142] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao. Heterogeneous environment aware streaming graph partitioning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2015.

[143] J. Xue, Z. Yang, S. Hou, and Y. Dai. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers (ToC)*, 66(5):876–890, 2017.

[144] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM, 2012.

[145] W. Yang, G. Wang, K.-K. R. Choo, and S. Chen. Hepart: A balanced hypergraph partitioning algorithm for big data applications. *Future Generation Computer Systems*, 83:250 – 268, 2018.

[146] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Generalized belief propagation. In *Advances in Neural Information Processing Systems (NIPS)*, pages 689–695, 2001.

[147] Y. Yuan, G. Wang, J. Y. Xu, and L. Chen. Efficient distributed subgraph similarity matching. *The VLDB Journal*, 2015.

[148] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale rdf data. In *Proceedings of the VLDB Endowment*, volume 6, pages 265–276. VLDB Endowment, 2013.

[149] F. B. Zhan and C. E. Noon. Shortest path algorithms: an evaluation using real road networks. *Transportation science*, 32(1):65–73, 1998.

[150] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 605–614. ACM, 2017.

[151] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, and H. T. Shen. Distributed shortest path query processing on dynamic road networks. *The VLDB Journal*, 26(3):399–419, 2017.

[152] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. In *Data Engineering (ICDE), 2016 IEEE International Conference on*, 2016.

[153] A. Zheng, A. Labrinidis, P. K. Chrysanthis, and J. Lange. Argo: Architecture-aware graph partitioning. In *Big Data (Big Data), 2016 IEEE International Conference on*, 2016.

[154] A. C. Zhou, S. Ibrahim, and B. He. On achieving efficient data transfer for graph processing in geo-distributed datacenters. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 1397–1407. IEEE, 2017.

[155] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference (ATC)*, pages 375–386, 2015.

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(Christian Mayer)