Optimizing Differential Computation for Large-Scale Graph Processing

by

Siddhartha Sahu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Keval Vora
Associate Professor,
School of Computing Science,
Simon Fraser University

Supervisor: Semih Salihoğlu
Associate Professor,
School of Computer Science,
University of Waterloo

Internal Member: M. Tamer Özsu
University Professor,
School of Computer Science,
University of Waterloo

Internal Member: Khuzaima Daudjee
Research Associate Professor,
School of Computer Science,
University of Waterloo

Internal-External Member: Patrick Lam
Associate Professor,
Electrical and Computer Engineering,
University of Waterloo

Author’s Declaration

Statement of Contributions

Some portions of this thesis are based on peer-reviewed joint work with Semih Salihoglu, Amine Mhedhbi, Jimmy Lin, and M. Tamer Özsu, in which I am the first author and the primary contributor [140, 141, 142].

Abstract

Diverse applications spanning areas such as fraud detection, risk assessment, recommendations, and telecommunications process datasets characterized by entities and their relationships. Graphs naturally emerge as the most intuitive abstraction for modeling these datasets. Many practical applications seek the ability to share computations across multiple snapshots of evolving graphs to efficiently perform analyses, such as evaluating changing road conditions in transportation networks or performing contingency analysis on infrastructure networks. The research in this thesis is motivated by the challenge of efficiently supporting such applications on large datasets.

Differential computation (DC) has emerged as a powerful general technique for incrementally maintaining computations over evolving datasets, even those containing arbitrarily nested loops. It is thus a promising technique that can be used to build the kinds of applications that motivate this thesis. We present a study of DC that explores how it can be used to build practical data systems. In particular, this thesis addresses two challenges that impede the adoption of DC: (i) the lack of high-level interfaces that can be used to develop graph-specific applications; and (ii) scalability challenges that arise due to the general maintenance technique used by DC, making it less efficient for application-specific workloads.

The main contribution of this thesis is to show how DC can be made more practical for graph processing systems. To address the lack of high-level interfaces, we built GraphSurge, a system that can be used to create and analyze multiple views over static graphs using a declarative programming interface. When users perform graph computations on a collection of views, GraphSurge internally uses DC to share the computation across all views. We develop several optimizations that improve the scalability of DC. Within GraphSurge, we identify two optimization problems, which we call collection ordering and collection splitting, and present algorithms to solve these problems. These optimizations improve the runtime of GraphSurge applications by up to an order of magnitude. In the reference implementation of DC, we identify two design bottlenecks in how data is indexed and processed within operators. To address the bottlenecks, we implement a new index and an optimization called Fast Empty Difference Verification, which improves the runtime of graph processing workloads by up to 14×. Our work was informed by insights from a non-technical user survey we conducted to understand how graphs are used in practice.

Acknowledgements

This thesis has been a long time in the making. As far back as I can remember, I have always wanted to do a Ph.D. Now that I am at the end of the journey, I can only take a small slice of the credit. I am here mainly because of the generosity and help of so many wonderful people.

I will start by mentioning my eternal gratitude for my family. My grandma and grandpa always showered me with unwavering love, spoiling me to the core. My mom and dad kept me more grounded, at the same time encouraging me to follow my own path and unhesitatingly providing me with everything I needed to do so. They are the ones who instilled in me a love for books and learning. Despite our sibling rivalry, my sister has always been there for me when I needed her. I would not have been able to come halfway around the world to Canada from a small city in India without all the hard work and sacrifices that my family has put in.

I owe everything I know about research and academic writing to my advisor Semih Salihoğlu. Semih has been a strong guiding light throughout my Ph.D. journey with all its ups and downs. Semih's ability to drill down to the core of a problem and ask the right questions has never failed to amaze me. I was fortunate enough to be one of his first students. I vividly remember our early days of building the first version of GraphflowDB, with many rounds of meticulous code reviews. We managed to quickly publish what was my first academic paper and drive down to Chicago together to present it, giving me a great first taste of what research is all about. The projects I worked on from there were more challenging, but Semih always managed to point me in the right direction. Outside of research, Semih constantly encouraged us to socialize and have fun as a group, including organizing many social meets, gaming sessions, and barbecue outings. I will always look up to him as a great researcher and a great human being.

I am grateful to the Naiad team for developing Differential Computation, and a special shout-out to Frank McSherry for putting in the effort to implement Differential Dataflow (DD), an open-source Rust project that serves as the foundation of my thesis. Rust is an exceptional systems programming language, and I am glad to have been able to use it throughout my research. Especially when working with a complex codebase like DD, Rust ensures that memory safety and concurrency bugs have not crept in while modifying and refactoring code, leaving more room to focus on the actual logic. I appreciate the Rust community for having been able to create a delightful ecosystem around the core language.

I am grateful to Bogdan Arsintescu, Juan Colmenares, and the rest of the LIquid graph team at LinkedIn for hosting me as an intern and answering all my incessant questions. It gave me a great perspective on what it took to run a practical graph system at scale.

I am constantly amazed by the thoughtfulness and vivacity of the many friends and colleagues I have met through the years. In no particular order, I thank: Amine Mhedhbi and Chathura

Kankanamge, with whom I have fond memories of spending countless days and nights doing research and discussing life; Angshuman Ghosh and Khushbu Patel, who are my goto partners-in-crime for everything food and fun; Mayumi, for giving me a place to stay and teaching me about Japanese culture, Arjun Singh, whom I can trust to travel with anywhere in the world; Pranjal Gupta, who spent a lot of time with me biking and driving around Ontario; Anurag Dubey, with whom I shared many great hiking experiences in Mumbai and Denver; Olivier Nguyen, my first housemate in Canada who graciously tolerated my enthusiasm at the new Canadian life; Babar Memon, who led me on several adventures exploring the Waterloo social scene; Abeer Khan, Hemant Saxena, Vineet John, Kshitij Jain, Brad Glasbergen, Michael Abebe, Anil Pacaci, Royal Sequiera, Shahid Khaliq, and Nafisa Anzum, who were always fun to be around in and outside the DSG lab; Aaditya Tripathy, who is always up for trying new experiences; Ravi Hemnani, Bharath MS, Abhinav Sethi, Shyam Sunder, and the rest of the *gmond* gang, for making life at work enjoyable and for getting me started on my Ph.D. applications; Aamir Jamal, Sunny Lalwani, Vivek Mangal, Rahul Prakash, and Indrajit Sarkar, who showed me how to have fun in life; Pankaj Kumar, who got me out of my comfort zone; Vishal Kumar and Samir Pramanik, for our intense gaming sessions; and Margaret Hancock, who I was lucky to have as a landlady, taught me how to make a pizza, and made life during the pandemic more bearable with puzzles and socially distant outings. Most importantly, I thank my girlfriend, Meenakshi Gupta, for always pushing me to be my better self and for her unwavering support that made the hours spent on my thesis less lonely.

I am truly grateful to the UWaterloo staff who take care of all the work necessary to keep academia functioning. I especially thank Paula Roser and Nadine Zinger for being readily accessible, Jo-Ann Hardy and Joe Patrick with whom I had many fun interactions, and Lori Paniak for his constant vigilance of our server farms.

I had many enthusiastic conversations with Prof. Khuzaima Daudjee. He always had a welcoming presence in the DSG Lab, often dropping in to check up on my research. I also had a lot of fun TAing his distributed systems course, helping design and evaluate a prototype distributed file system assignment. I thank Prof. Bernard Wong and Prof. Samer Al-Kiswany, whom I interacted with through their graduate courses and who gave me great advice on how to navigate academic life. I also thank Prof. Sujan Kumar Saha and Prof. Abhijit Mustafi from my undergraduate years, who gave me encouragement and support in pursuing my Ph.D.

Finally, I thank my committee members: Prof. Keval Vora, Prof. Tamer Özsu, Prof. Khuzaima Daudjee, and Prof. Patrick Lam. Everyone asked great questions, both at a high level as well as on the nitty-gritty details, and their suggestions for improvements were spot on. I sincerely appreciate the time that they have taken to read and engage with my work. The moment when they announced my thesis had been approved made me feel that it was all worth it.

Dedication

To

Thakuma & *Dadu*,
for their everlasting love and care,

and

Ma, *Bapi*, & *Munu*,
for always having my back.

Table of Contents

List of Figures

List of Tables

"The world is full of fascinating problems waiting to be solved."

— How To Become A Hacker

"And when all the wars are done, a butterfly will still be beautiful."

— Ruskin Bond

1 Introduction

A variety of applications, such as fraud detection, risk assessment, and recommendations from telecommunications, transportation, financial, social, and biological networks process large datasets that consist of a set of entities and the relationships between these entities. Graphs are perhaps the most natural abstraction to model the records of these datasets in data systems. Many applications require the core capability of sharing computation across multiple similar snapshots of these graphs to perform their computations efficiently. An example of a popular application is analyzing dynamic graphs which evolve by ingesting small-size updates. For example, finding the shortest paths on a transportation network where road conditions change over time can be modeled as a dynamic graph. The states of the graph after each set of updates can be seen as multiple snapshots, over which the same computation of finding the shortest paths needs to be performed. Similarly, contingency or perturbation analysis is a type of "what if" analysis performed on infrastructure networks. In this analysis, the network is modified in small ways to study its resilience to perturbations. Different perturbation scenarios can be modeled as multiple snapshots of graphs over which the same set of computations analyze the structures of these snapshots. In order to efficiently support these applications on large-scale datasets, data systems need the ability to sharing computations across snapshots instead of re-executing these computations from scratch. These applications and this core problem are the primary motivation of the research conducted in this thesis.

Computation sharing across different snapshots of datasets is the key problem behind two fundamental optimization problems in data management, namely, incremental view maintenance [150] and multi-query optimization [147]. There is a long line of research on these problems for supporting data computations that can be expressed in relational algebra [77].

However, these approaches do not efficiently handle recursive computations and thus often fall short on many graph computations, which are recursive in nature. *Differential computation* (DC) [47, 124] is a recent technique that has emerged as a general technique for sharing and maintaining computation across evolving datasets—or those that can be modeled as such—for arbitrary dataflow computations, including those that contain nested recursive loops. As such, it is a promising technique that could serve as a foundation for building data systems that support the applications motivating this thesis.

Briefly, DC is based on a simple principle. Given a dataflow that consists of operators whose inputs and outputs are sets of records, DC maintains and indexes all input and output records of each operator at every "timestamp" in the computation. Timestamps are multi-dimensional and represent points in time in the computation, such as the version of an input dataset or the iteration number within a recursive loop. DC stores records as a set of timestamped *differences*, such that the set of records can be reproduced for each timestamp. Upon changes to base datasets, DC automatically activates the operators whose inputs and outputs may have changed, so that they rerun to "fix" the previously indexed state of computation. As shown in the original publications on DC [47, 124] and several follow-up work [52, 142], DC is very general and can maintain complex graph computations, such as finding weakly connected components, finding shortest paths, and some regular path queries. The popular algorithms of these computational problems have single loops. DC can even maintain and share computation when running algorithms that contain nested loops, such as the Coloring algorithm [132] for finding strongly connected components (SCC). Indeed, maintenance of SCC was the motivating example used in the original publication of DC [124].

The study of DC in the literature is in its infancy and the principles for making it more practical in actual data systems are still not well understood. The premise of this thesis is that two challenges, among others, hinder the adoption of DC in practical graph systems:

- The *Differential Dataflow* (DD) system [46], which is the reference implementation of DC, is a general and not graph-specific data processing system. As such, it lacks the interfaces needed for developing graph computation applications.
- As a general computation maintenance and sharing technique, DC is oblivious to the properties of the graph-specific computations, which can make it less efficient than graph-specific maintenance techniques.

This thesis argues that we can make DC more practical for application developers by designing systems with graph-specific interfaces and graph-specific optimizations, i.e., optimizations that are specialized to the common graph computations. In the rest of this introductory chapter, we first give an overview of the contributions and then an outline of the thesis.

1.1 Contributions

1.1.1 Survey of Graph Technology Users

In Chapter 2, we present a user study that aims to understand what type of graph data users have, what graph computations they perform, what applications use these computations, which software they use, and what their main challenges are. The chapter presents the results of an online survey that was filled by 89 users of different graph technologies and 8 in-person interviews, as well as a study of the white-papers of existing open-source and commercial graph technologies, including graph database management systems, graph computations, and graph visualization systems. The contributions of this chapter are orthogonal to the technical contributions of this thesis in other chapters and may be of independent interest to readers. The rationale for including it as part of the thesis is that some of our findings in this study serve as motivation for the technical problems that will be addressed in the rest of the thesis. Specifically, the survey identifies performance and scalability problems in existing graph technologies, the importance of graph computations over dynamic graphs, and the description of an actual contingency analysis application on an electric grid network. This chapter can be read in isolation and in any order, while the rest of the chapters should be read in order.

1.1.2 The Graphsurge System

In Chapter 4, we present the Graphsurge system. Graphsurge is a graph computation system for analyzing multiple snapshots of a static graph. It is designed to support applications similar to contingency analysis, where a common analysis, such as a path or connectivity analysis, needs to be performed on many snapshots of the same graph. Graphsurge has a high-level programming interface to define individual graph snapshots called views, and view collections, which consist of multiple views. Users program Graphsurge by writing batch graph computations over a single snapshot using a dataflow-based API. Graphsurge then runs the computation across all views and uses DC to automatically share computation. Within the context of Graphsurge, we introduce two optimization problems:

- Collection ordering is the problem of giving an order to the views in a view collection so that DC can maximize computation sharing across the views. We show that this problem is NP-hard and provide an efficient 3×-approximation algorithm.
- Collection splitting is the problem of choosing the graph views in an ordered collection that do not share enough computation with their previous view so that re-executing a computation from scratch is cheaper than trying to share computation. We provide an adaptive algorithm that can decide when to "split" a view during runtime, i.e., when Graphsurge is performing the computation on a view collection.

We present extensive experiments demonstrating that our algorithms improve the runtimes of contingency analysis-like applications by up to an order of magnitude compared to using vanilla DC, as well as several specialized systems that can be repurposed to run the same workloads. The research presented in this chapter demonstrates that DC can be the foundation for developing graph computation systems for contingency analysis-like applications by developing graph-specific interfaces. This is also the first demonstration that DC can be applied to workloads other than dynamic graph computations, which was the motivating application of DC in prior literature. Further, this research also demonstrates that DC's performance can be improved by designing application-specific optimizations.

1.1.3 Optimizations to DC and DD

In Chapter 5, we study the problem of optimizing DC and its reference implementation DD. We first study the problem of how the differences should be stored in DD. We present two design alternatives, either by indexing the differences by the values or by the timestamps. We show that for a suite of common graph computations, the second option is often more efficient, especially when the system memory is limited. Second, we observe that the main computational bottleneck of DC is to rerun operators on different dataset snapshots to determine whether the outputs of these operators may have changed or not. Often, this expensive computation of rerunning the operators only verifies that there are no new differences in the output of the operator. Based on this observation, we ask whether it is possible to detect that there are no output differences, without rerunning DC's default operator logic. We answer this question in the affirmative by providing an optimization that is designed for a dataflow sub-routine called *iterative frontier expansion* [52] (IFE), a common subroutine across many graph computations. Our optimization is called *Fast Empty Difference Verification (FEDiV)*. FEDiV can only be applied for IFE and as such is a graph-specific optimization, as stated in the central argument of this thesis. We leave open whether FEDiV-like optimizations can be applied to DC when maintaining arbitrary dataflow computations. We provide proof that FEDiV is correct. Finally, we provide experiments demonstrating that using timestamp-based indexing of differences and FEDiV on a suite of graph computation workloads can improve DD's runtime by up to 19×. The workloads in this suite include both dynamic graph workloads as well as workloads that are used to evaluate Graphsurge in Chapter 4.

1.2 Thesis Outline

The rest of this thesis is organized as follows. We first present the methodology and analysis of a non-technical user survey in Chapter 2. We then give a background on differential computation in

Chapter 3. Chapter 4 presents our work on Graphsurge and Chapter 5 shows how we optimized DC for scalability. We then present related work in Chapter 6 and end with conclusions and directions for future work in Chapter 7.

2

User Survey of Graph Technology[1]

In this chapter, we present a survey of users of several classes of graph technology. Graph technology used for managing or processing graphs is highly varied and includes graph database management systems (DBMSs) [12, 14, 18, 27, 30, 36, 41] and RDF systems [35]. These are the two classes of DBMSs in the market that offer a graph-based data model. Graph DBMSs adopt the *property graph* data model [64], while RDF systems adopt the *resource description framework* data model [64]. Additional graph technologies include linear algebra software [13,28], graph visualization software [16, 19], graph query languages [11, 34, 160], and distributed graph processing systems [5, 7, 9]. Despite the prevalence of these software, there is little research on how graph data is actually used in practice and the major challenges facing users of graph data, both in industry and research. In April 2017, we conducted an online survey across 89 users of 22 different software products, to answer 4 high-level questions:

(i) What types of graph data do users have?

(ii) What computations do users run on their graphs?

(iii) Which software do users use to perform their computations?

(iv) What are the major challenges users face when processing their graph data?

Our major findings that are specifically relevant for this thesis are as follows:

[1]Contents of this chapter have appeared in VLDB 2018 Conference [140] and a special issue of the VLDB Journal [141]. The survey was reviewed and received ethics clearance through the University of Waterloo Research Ethics Committee (ORE #22102).

- *Variety:* Graphs in practice represent a wide variety of entities, many of which are not naturally thought of as vertices and edges, such as traditional products, orders, and transactions.
- *Ubiquity of Very Large Graphs:* Many graphs in practice are large, often containing over a billion edges. Many of these graphs are highly dynamic and change frequently.
- *Challenge of Scalability:* Scalability is unequivocally the most pressing challenge faced by participants. The ability to process very large graphs efficiently seems to be the biggest limitation of existing software.

Many users indicated that they perform incremental or dynamic computations, but using custom solutions. This indicates the need for easy-to-use solutions for performing incremental computations on large dynamic graphs. Our survey has also highlighted other interesting facts, such as the importance of graph visualization for users, the prevalence of RDBMSes to manage and process graphs, the prevalence of machine learning on graph data, e.g., for clustering vertices, predicting links, and finding influential vertices.

We further reviewed user feedback in the mailing lists, bug reports, and feature requests in the source code repositories of 22 software products between January and September of 2017 with two goals: (i) to answer several new questions that the participants' responses raised; and (ii) to identify more specific challenges in different classes of graph technologies than the ones we could identify in participants' responses.

To better understand the applications supported by graphs, we reviewed the whitepapers posted on the websites of 8 graph software products. We also interviewed 6 users and 2 developers of graph processing systems. Our reviews and interviews corroborated our findings that graphs have a wide range of applications but also highlighted several new applications, such as contingency analysis, that we had not identified in our online survey. Contingency analysis will be one of the motivating applications for the Graphsurge system covered in Chapter 4.

2.1 Methodology of Online Survey, Mailing Lists, and Source Repositories

In this section, we first describe the format of our survey and then how we recruited the participants. Next, we describe the demographic information of the participants, including the organizations they come from and their roles in their organizations. Then we describe our methodology for reviewing user feedback in the mailing lists, bug reports, and feature requests in the code repositories of the software products. Finally, we describe our methodology of reviewing whitepapers and our interviews in Sections 2.3.1 and 2.4.1, respectively.

2.1.1 Online Survey Format and Participants

Format

The survey was in the format of an online form. All of the questions were optional. There were 2 types of questions:

(i) *Multiple Choice*: There were 3 types of multiple choice questions: (a) yes-no questions; (b) questions that allowed only a single choice as a response; and (c) questions that allowed multiple choices as a response. The participants could use an *Other* option when their answers required further explanation or did not match any of the provided choices. We randomized the order of choices in the questions about the computations that participants run and the challenges they face.

(ii) *Short Answer*: For these questions, the participants entered their responses in a text box.

There were 34 questions grouped into six categories: (i) demographic questions; (ii) graph datasets; (iii) graph and machine learning computations; (iv) graph software; (v) major challenges; and (vi) workload breakdown.

Participant Recruitment

We prepared a list of 22 popular software products for processing graphs (see Table 2.1) that had public user mailing lists covering 6 types of technologies: graph database systems, RDF engines, distributed graph processing systems (DGPSes), graph libraries to run and compose graph algorithms, visualization software, and graph query languages.[2] Our goal was to be as comprehensive as possible in recruiting participants from the users of different graph technologies. However, we acknowledge that this list is incomplete and does not cover all of the graph software used in practice.

We conducted the survey in April 2017, and used 4 methods to recruit participants from the users of these 22 software products:

- *Mailing Lists*: We posted the survey to the user mailing lists of the software in our list.
- *Private Emails*: Five mailing lists: (i) Neo4j; (ii) OrientDB; (iii) ArangoDB; (iv) JanusGraph; and (v) NetworkX, allowed us to send private emails to the users. We sent private emails to 171 users who were active on these mailing lists between February and April of 2017.

[2]The linear algebra software we considered, e.g., BLAS [13] and MATLAB [28], either did not have a public mailing list or their lists were inactive.

Technology	Software	# Users	
Graph Database System	ArangoDB [12]	40	238
	Caley [14]	14	
	DGraph [18]	33	
	JanusGraph [27]	32	
	Neo4j [30]	69	
	OrientDB [36]	45	
	Sparksee [41]	5	
RDF Engine	Apache Jena [8]	87	110
	Virtuoso [35]	23	
Distributed Graph Processing Engine	Apache Flink (Gelly) [5]	24	39
	Apache Giraph [7]	8	
	Apache Spark (GraphX) [9]	7	
Query Language	Gremlin [11]	82	82
Graph Library	Graph for Scala [22]	4	97
	GraphStream [26]	8	
	Graphtool [23]	28	
	NetworKit [31]	10	
	NetworkX [32]	27	
	SNAP [40]	20	
Graph Visualization	Cytoscape [16]	93	116
	Elasticsearch X-Pack Graph [19]	23	
Graph Representation	Conceptual Graphs [15]	6	6

Table 2.1: Software products used for recruiting participants and the count of active users on their mailing list in Feb-Apr 2017. The last column is the total user count for each technology.

- *Slack Channels*: Two of the software products on our list, Neo4j and Cayley, had Slack channels for their users. We posted the survey to these channels.
- *Twitter*: A week after posting our survey to the mailing lists and Slack channels and sending private emails, we posted a tweet with a link to our survey to 7 of the 22 software products that had an official Twitter account. Neo4j retweeted our tweet.

Participants that we recruited through different methods shared the same online link and we could not tell the number of participants recruited from each method. We suspect that there were more users from graph database systems because their mailing lists contained more active users, as can be seen in Table 2.1. Moreover, 4 of the 5 mailing lists that allowed us to send private emails and the Slack and Twitter channels belonged to graph database systems. Within one week of posting the survey on Twitter, we received 12 responses.

In the end, there were 89 participants. Below, we give an overview of the organizations these participants work in and the role of the participants in their organizations.

Field	Total	R	P
Information Technology	48	12	36
Research in Academia	31	31	0
Finance	12	2	10
Research in Industry Lab	11	11	0
Government	7	3	4
Healthcare	5	3	2
Defense & Space	4	3	1
Pharmaceutical	3	0	3
Retail & E-Commerce	3	0	3
Transportation	2	0	2
Telecommunications	1	1	0
Insurance	0	0	0
Other	5	2	3

(a) The participants' fields of work.

Size	Total	R	P
1–10	27	17	10
10–100	23	6	17
100–1000	14	4	10
1000–10000	6	4	2
> 10000	15	4	11

(b) Sizes of the participants' organizations.

Table 2.2: The demographics of the participants.

Field of Organizations: We asked the participants which field they work in. Participants could select multiple options. Table 2.2a shows the 12 choices and participants' responses. In the table, "R" and "P" indicate researchers and practitioners (defined momentarily), respectively. In addition to the given choices, using the *Other* option, participants indicated 5 other fields: education, energy market, games and entertainment, investigations and audits, and grassland management. In total, participants indicated 17 different fields, demonstrating that graphs are being used in a wide variety of fields. Throughout the chapter, we group the participants into 2 categories:

- *Researchers* are the 36 participants who indicated at least one of their fields as research in academia or research in an industry lab.
- *Practitioners* are the remaining 53 participants who did not select research in academia or an industry lab.

Size of Organizations: Table 2.2b shows the sizes of the organizations that the participants work in, which ranged from very small organizations with fewer than 10 employees to very large ones with more than 10,000 employees.

Role at Work: We asked the participants their roles in their organizations and gave them the following 4 choices: (i) researcher; (ii) engineer; (iii) manager; and (iv) data analyst. Participants could select multiple options. The top 4 roles were engineers, selected by 54, researchers, selected by 48, data analysts, selected by 18, and managers, selected by 16. The other roles participants indicated were architect, devops, and student.

2.2 Online Survey

In this section, we describe the questions we asked in the survey and report the responses of the participants.

2.2.1 Graph Datasets

In this section, we describe the properties of the graph datasets that the participants work with.

Real-World Entities Represented

We asked the participants about the real-world entities that their graphs represent. We provided them with 4 choices and the participants could select multiple choices.

(i) *Humans*: e.g., employees, customers, and their interactions.

(ii) *Non-Human Entities*: e.g., products, transactions, or web pages.

(iii) *RDF or Semantic Web*.

(iv) *Scientific*: e.g., chemical molecules or biological proteins.

For the participants who selected non-human entities, we followed up with a short-answer question asking them to describe what these are. Participants indicated 52 different kinds of non-human entities, which we group into 7 broad categories.[3] We indicate the acronyms we use in our tables for each category in parentheses:

(i) *Products* (NH-P): e.g., products, orders, and transactions.

(ii) *Business and Financial Data* (NH-B): e.g., business assets, funds, or bitcoin transfers.

(iii) *World Wide Web Data* (NH-W).

(iv) *Geographic Maps* (NH-G): e.g., roads, bicycle sharing stations, or scenic spots.

(v) *Digital Data* (NH-D): e.g., files and folders or videos and captions.

(vi) *Infrastructure Networks* (NH-I): e.g., oil wells and pipes or wireless sensor networks.

(vii) *Knowledge and Textual Data* (NH-K): e.g., keywords, lexicon terms, words, and definitions.

Table 2.3 shows the responses. In the table, the number of academic publications that use each type of graph is listed in the *A* row. We highlight two interesting observations:

[3]Six entities that the participants mentioned did not fall under any of our 7 categories, which we list for completeness: call records, computers, cars, houses, time slots, and specialties.

Category	Human	RDF	Scientific	Non-Human	NH-P	NH-B	NH-W	NH-G	NH-D	NH-I	NH-K
Total	45	23	15	60	13	11	4	7	5	9	11
R	18	11	9	22	1	6	2	4	1	7	6
P	27	12	6	38	12	5	2	3	4	2	5
A	165	20	45	169	7	28	77	33	0	17	11

Table 2.3: Real-world entities represented by the participants' graphs and studied in publications. Legend for non-human entities: *Products* (NH-P), *Business and Financial Data* (NH-B), *Web Data* (NH-W), *Geographic Maps* (NH-G), *Digital Data* (NH-D), *Infrastructure Networks* (NH-I), *Knowledge and Textual Data* (NH-K).

- *Variety*: Real graphs capture a very wide variety of entities. Readers may be familiar with entities such as social connections, infrastructure networks, and geographic maps. However, many other entities in the participants' graphs may be less natural to think of as graphs. These include malware samples and their relationships, videos and their captions, or scenic spots, among others. This lends credence to the cliché that graphs are everywhere.
- *Product Graphs*: Products, orders, and transactions were the most popular non-human entities represented in practitioners' graphs, indicated by 12 practitioners. This contrasts with their relative unpopularity among researchers and academics. Such product-order-transaction data is traditionally the classic example of enterprise data that perfectly fits the relational data model. Interestingly, enterprises represent similar product data as graphs, possibly because they find value in analyzing connections in such data.

Size

We asked the participants the number of vertices, number of edges, and total uncompressed size of their graphs. They could select multiple options. Tables 2.4a, 2.4b, and 2.4c show the responses. As shown in the tables, graphs of every size, from very small ones with fewer than 10K edges to very large ones with more than 1B edges, are prevalent across both researchers and practitioners. We make one interesting observation:

- *The Ubiquity of Very Large Graphs:* A significant number of participants work with very large graphs. Specifically, 20 participants (8 researchers and 12 practitioners) indicated using graphs with more than a billion edges. These large graphs represent a variety of entities, including social, scientific, RDF, product, and digital data,[4] indicating that very large graphs appear in a wide range of domains.

[4]Some participants selected multiple graph sizes and multiple entities, so we cannot perform a direct match of which graph size corresponds to which entity. The entities we list here are taken from the participants who selected a single graph size and entity, so we can directly match the size of the graph to the entity.

Vertices	Total	R	P
< 10K	22	11	11
10K–100K	22	9	13
100K–1M	19	7	12
1M–10M	17	6	11
10M–100M	20	10	10
> 100M	27	10	17

(a) Number of vertices.

Edges	Total	R	P
< 10K	23	11	12
10K–100K	22	9	13
100K–1M	13	3	10
1M–10M	9	5	4
10M–100M	21	8	13
100M–1B	21	8	13
> 1B	20	8	12

(b) Number of edges.

Size	Total	R	P
< 100MB	23	12	11
100MB–1GB	19	9	10
1GB–10GB	25	9	16
10GB–100GB	17	5	12
100GB–1TB	20	8	12
> 1TB	17	5	12

(c) Total uncompressed bytes.

Table 2.4: The sizes of the participants' graphs.

One thing that is not clear from our survey is how much larger the participants' graphs are beyond the maximum limits we inquired about (100 million vertices, 1 billion edges, and 1TB uncompressed data). In order to answer this question, we categorized the graph sizes mentioned in the user emails we reviewed that were beyond these sizes. Focusing on the number of edges, we found 42 users with 1–10B-edge graphs, 17 with 10B–100B-edge graphs, and 7 users processing graphs over 100B edges. Two participants also clarified through an email exchange that their graphs contained 4B and 30B edges. As in our survey results, these large graphs represented a wide range of entities, such as product-order-transaction data, or entities from agriculture and finance.

Other Questions on Graph Datasets

Topology: We asked the participants whether their graphs were: (i) *directed or undirected*; and (ii) *simple graphs or multigraphs*. We clarified that multigraphs are those with possibly multiple edges between two vertices, while simple graphs do not allow multiple edges between two vertices. Tables 2.5a and 2.5b show the responses.

Types of Data Stored on Vertices and Edges: We asked the participants whether they stored data on the vertices and edges of their graphs. All participants except 3 indicated that they do. We asked the types of data they store and gave them 4 choices: (i) string; (ii) numeric; (iii) date or timestamp; and (iv) binary. Table 2.5c shows participants' responses. Five participants also indicated storing JSON, lists, and geographic coordinates using the *Other* option.

Dynamism: We asked the participants how frequently the vertices and edges of their graphs change, i.e., are added, deleted, or updated. We provided 3 choices with the following explanations: (i) *static*: there are no or very infrequent changes; (ii) *dynamic*: there are frequent changes, and all changes are stored permanently; and (iii) *streaming*: there are very frequent changes

Topology	Total	R	P
Only Directed	63	23	40
Only Undirected	11	6	5
Both	15	7	8

(a) Directed vs. Undirected

Topology	Total	R	P
Only Simple Graphs	26	9	17
Only Multigraphs	50	20	30
Both	13	7	6

(b) Simple vs. Multigraphs

Type	Vertices			Edges		
	Total	R	P	Total	R	P
String	79	31	48	66	24	42
Numeric	63	23	40	59	23	36
Date/Timestamp	56	19	37	49	18	31
Binary	15	8	7	8	4	4

(c) Data types stored on vertices and edges.

Table 2.5: The topology and stored data types of the participants' graphs.

Frequency	Total	R	P
Static	40	21	19
Dynamic	55	22	33
Streaming	18	9	9

Table 2.6: Frequency of changes.

and the participants' software discards some of the graph after some time. Table 2.6 shows the responses. 55 users (22 researchers and 33 practitioners) indicated they had dynamic graphs and 18 (9 researchers and 9 practitioners) indicated they had streaming graphs. This is strong evidence that many graphs used in practice are changing frequently.

2.2.2 Computations

In this section, we describe the computations that the participants perform on their graphs.

Graph Computations

Our goal in this question was to understand what types of graph queries and computations, not including machine learning computations, participants perform on their graphs. We asked a multiple choice question that contained as choices a list of queries and computations followed by a short answer question that asked for computations that may not have appeared in the first question as a choice. In the multiple choice question, instead of asking for a set of ad-hoc queries and computations, we selected a list of graph queries and computations that appeared in the publications of 6 conferences. The conferences we reviewed are shown in Table 2.7.

Conference	Years reviewed
VLDB	2014 [105], 2017 [63], 2018 [51]
KDD	2015 [119], 2017 [111], 2018 [112]
SOCC	2015 [148], 2017 [153], 2018 [154]
OSDI / SOSP	2016 [113], 2017 [155], 2018 [55]
ICML	2016 [56], 2017 [136], 2018 [84]
SC	2016 [107], 2017 [58], 2018 [145]

Table 2.7: Academic conferences and surveyed years.

Computation	Total	R	P	A
Finding Connected Components	55	18	37	31
Neighborhood Queries (e.g., finding 2-degree neighbors of a vertex)	51	19	32	9
Finding Short / Shortest Paths	43	18	25	28
Subgraph Matching (e.g., finding all diamond patterns, SPARQL)	33	14	19	52
Ranking & Centrality Scores (e.g., PageRank, Betweenness Centrality)	32	17	15	45
Aggregations (e.g., counting the number of triangles)	30	10	20	24
Reachability Queries (e.g.,checking if u is reachable from v)	27	7	20	8
Graph Partitioning	25	13	12	12
Node-similarity (e.g., SimRank)	18	7	11	11
Finding Frequent or Densest Subgraphs	11	7	4	4
Computing Minimum Spanning Tree	9	5	4	4
Graph Coloring	7	3	4	8
Diameter Estimation	5	2	3	2

Table 2.8: Graph computations performed by the participants and studied in publications.

Table 2.8 shows the 13 choices we provided in the multiple choice question, the responses we got, and the number of academic publications that use or study each computation. As shown in the table, all of the 13 computations are used by both researchers and practitioners. Except for 2 computations, the popularity of these computations is similar among participants' responses and academic publications. The exceptions are neighborhood and reachability queries, which are respectively used by 51 and 27 participants, but studied respectively in 10 and 8 publications. Finding connected components appears to be a very popular and fundamental graph computation—it is the most popular graph computation overall and also among practitioners. We will use algorithms to solve this problem extensively throughout the thesis.

Computation	**Total**	**R**	**P**	**A**
Clustering	42	22	20	22
Classification	28	10	18	34
Regression (Linear / Logistic)	11	5	6	2
Graphical Model Inference	10	5	5	5
Collaborative Filtering	9	4	5	5
Stochastic Gradient Descent	4	2	2	9
Alternating Least Squares	0	0	0	1

(a) Machine learning computations.

Computation	**Total**	**R**	**P**	**A**
Community Detection	31	15	16	15
Recommendation System	26	10	16	5
Link Prediction	25	10	15	11
Influence Maximization	14	5	9	6

(b) Problems solved by machine learning algorithms.

Table 2.9: Machine learning computations and problems performed by the participants and studied in publications.

Machine Learning Computations

We next asked participants what kind of machine learning computations they perform on their graphs. Similar to the previous question, these questions were formulated to identify the machine learning computations that appeared in the academic publications we reviewed. We asked the following 2 questions:

- *Which machine learning computations do you run on your graphs?* The choices were: clustering, classification, regression (linear or logistic), graphical model inference, collaborative filtering, stochastic gradient descent, and alternating least squares.
- *Which problems that are commonly solved with machine learning do you solve using graphs?* The choices were: community detection, recommendation system, link prediction, and influence maximization.

Tables 2.9a and 2.9b show the responses and the number of academic publications that use or study each computation. It is clear that machine learning is used very widely in graph processing. Specifically, 61 participants indicated that they either perform a machine learning computation or solve a problem using machine learning on their graphs. Clustering is the most popular computation performed, while community detection is the most popular problem solved using machine learning.

Other Questions on Computations

Streaming Computations: We asked the participants if they performed incremental or streaming computations on their graphs: 32 participants (16 researchers and 16 practitioners) indicated that they do. We followed up with a question asking them to describe the incremental or streaming computations that they perform. A total of 4 participants indicated computing graph

Software	**Total**	**R**	**P**	**A**
Graph Database System (e.g., Neo4j, OrientDB, TitanDB)	59	20	39	6
Apache Hadoop, Spark, Pig, Hive	29	11	18	10
Apache Tinkerpop (Gremlin)	23	9	14	1
Relational Database Management System (e.g., MySQL, PostgreSQL)	21	6	15	7
RDF Engine (e.g., Jena, Virtuoso)	16	8	8	12
Distributed Graph Processing Systems (e.g., Giraph, GraphX)	14	8	6	36
Linear Algebra Library / Software (e.g., MATLAB, Maple, BLAS)	8	6	2	6
In-Memory Graph Processing Library (e.g., SNAP, GraphStream)	7	5	2	4

Table 2.10: Software for graph queries and computations.

or vertex-level statistics and aggregations; A total of 3 participants indicated incremental or streaming computation of the following algorithms: approximate connected components, k-core, and hill climbing. The incremental connected components computation will be one of the core of computations we will use throughout this thesis.

We note that the 22 software products in Table 2.1 have limited or no support for incremental and streaming computations. We believe this is due to the lack of user-friendly off-the-shelf incremental and streaming systems for graphs. Although this thesis does not produce such a system, it argues that DD is a good foundation on which to develop such systems.

2.2.3 Graph Software

We next review the properties of the different graph software that the participants use.

Software Types

Software for Querying and Performing Computations: We asked the participants which types of graph software they use to query and perform computations on their graphs. The choices included 5 types of software from Table 2.1 as well as distributed data processing systems (DDPSes), such as Apache Hadoop and Spark, relational database management systems (RDBMSes), and linear algebra libraries and software, such as BLAS and MATLAB. Table 2.10 shows the exact choices and responses: 84 participants answered this question and each selected 2 or more types of software. We highlight 2 interesting observations:

- *Popularity of Graph Database Systems*: The most popular choice was graph database systems. We suspect this is partly due to their increasing popularity and partly due to the inherent

Software	Total	R	P	A
Graph Visualization	55	22	33	15
Build / Extract / Transform	14	8	6	0
Graph Cleaning	5	1	4	2
Synthetic Graph Generator	4	3	1	49
Specialized Debugger	2	0	2	0

Table 2.11: Software used for non-querying tasks.

bias in the participants we recruited—as explained in Section 2.1.1, more of them came from users of graph database systems.

- *Popularity of RDBMSes*: 21 participants (6 researchers and 15 practitioners) chose RDBMSes. We consider this number high given that we did not recruit participants from the mailing lists of any RDBMS. Interestingly, 16 of these 20 participants also indicated using graph database systems. From our survey, we cannot answer what the participants used RDBMSes for. It is possible that they use an RDBMS as the main transactional storage and a graph database system for graph-specific tasks such as traversals.

Software for Non-Querying Tasks: We asked the participants which types of graph software, possibly an in-house one, they use for tasks other than querying graphs. Table 2.11 shows the choices and the responses. We highlight one interesting observation:

- *Importance of Visualization:* Visualization software is, by a large margin, the most popular type of software participants use among the 5 choices. This clearly shows that graph visualization is a very common and important task. As we discuss in Section 2.2.4, participants also indicated graph visualization as one of their most important challenges.

2.2.4 Practical Challenges

We asked the participants 2 questions about the challenges they face when processing their graphs. First, we asked them to indicate their top 3 challenges out of 10 choices we provided. Table 2.12 shows the choices and the participants' responses. Second, we asked them to state their biggest challenge in a short-answer question. Three major challenges stand out unequivocally from the responses:

- *Scalability*: The ability to process large graphs is the most pressing challenge participants face. Scalability was the most popular choice in the first question for both researchers and practitioners. Moreover, it was the most popular answer in the second question where 13 participants reiterated that scalability is their biggest challenge. The specific scalability

Challenge	Total	R	P
Scalability (i.e., software that can process larger graphs)	45	20	25
Visualization	39	17	22
Query Languages / Programming APIs	39	18	21
Faster graph or machine learning algorithms	35	19	16
Usability (i.e., easier to configure and use)	25	10	15
Benchmarks	22	12	10
More general purpose graph software (e.g., that can process offline, online, and streaming computations)	20	9	11
Graph Cleaning	17	7	10
Debugging & Testing	10	2	8

Table 2.12: Graph processing challenges faced by participants.

challenges that the participants mentioned include inefficiencies in loading, updating, and performing computations, such as traversals, on large graphs.

- *Visualization*: Perhaps more surprisingly, graph visualization emerges as one of the top 3 graph processing challenges, as indicated by 39 participants in the first question and 1 participant in the short-answer question.
- *Query Languages and APIs*: Query languages and APIs present another common graph processing challenge, as indicated by 39 participants in the first question and 5 participants in the short-answer question. The specific challenges mentioned in the short-answer responses include expressivity of query languages, compliance with standards, and integration of APIs with existing systems.

2.3 Applications from Whitepapers

2.3.1 Methodology

In order to understand the popular application areas and fields using graph software, we surveyed the whitepapers of software vendors. Whitepapers are documents that software vendors provide, often for marketing purposes, to give information about the use cases of their products. In our case, we consider whitepapers to be any document found on a software vendor's official website categorized as a whitepaper, a use case, a case study, or a scenario. From the initial software products in Table 2.1, only four graph database systems, specifically ArangoDB, Neo4j, OrientDB, and Sparksee had whitepapers. To extend our review, we add the whitepapers of four

RDF engines that were not in our initial list: AllegroGraph [3], AnzoGraph [4], GraphDB by Ontotext [25], and Stardog [42]. We note that we only found whitepapers for graph database systems and RDF engines. In the end, we reviewed 89 whitepapers.

2.3.2 Applications

We labeled each whitepaper with a high-level application category and the field of industry of the customer in the case study. Table 2.13 shows the different applications, the fields of industry in which the application was covered, and the number of whitepapers from graph databases and RDF systems that discussed the project. We found a total of 12 applications described in the whitepapers of graph databases and 5 applications in the whitepapers of RDF systems. As seen in the table, there is an overlap of the applications across both types of systems.

The three most popular applications were as follows:

- *Data Integration*: 44 whitepapers discussed primarily some data integration task that constructs a central, highly heterogeneous graph from multiple sources. Data integration was also referred to as *master data management* or *knowledge graph creation* by some whitepapers. Many of these 44 whitepapers, as well as many whitepapers that discussed an initial data integration step, emphasized that customers found data integration easier in the semi-structured graph models than in structured relational tables.
- *Personalization & Recommendations*: The second most popular application was the use of graph-based application data to personalize user interactions and provide better recommendations for customers. For example, one whitepaper described an e-commerce website that created a graph representation of the behavior of online shoppers and the interactions between customers and products to help make new product recommendations [2].
- *Fraud & Threat Detection*: The third most popular application was the detection of fraud and threats in various businesses. For example, one whitepaper described the use of graphs to detect financial fraud in banks by looking for cycles in the graph formed after linking bank accounts, personal details, and financial transactions [17].

2.4 Applications from Interviews

2.4.1 Methodology

Whitepapers give an overview of the important applications using different software, but often contain very high-level and non-technical marketing language. To understand some of the applications using graphs in more depth, we invited the participants of our online survey for an

Application	Example	Fields	GDB	RDF	Total
Data Integration	Building an ontology by integrating multiple heterogeneous biomedical data sources	Aerospace, Art & Culture & Heritage, Education, Entertainment, Finance, Food & Cooking, Government, Health & Life Sciences, Intelligence & Law Enforcement, IT, Journalism, Marketing, Retail, Social Media, Toys & Figurines	23	21	44
Personalization & Recommendation	Recommending products on an e-commerce platform	Entertainment, Finance, Health & Life Sciences, Hospitality & Travel, IT, Manufacturing, Marketing, Media, Music, Retail, Social Media, Telecommunication	19	5	24
Fraud & Threat Detection	Detect cybercrime by searching for anomalous patterns	Finance, Government, Insurance, Media, Retail	9	1	10
Risk Analysis & Compliance	Risk reporting by banks to comply with government regulations.	Finance, Health & Life Sciences, IT, Supply Chain & Logistics	2	3	5
Identity & Access Management	Monitor direct and indirect owners of businesses for financial analysis	Insurance, IT, Telecommunication	4	0	4
Infrastructure Management & Monitoring	Manage cascading failures by tracking server interdependencies	Intelligence & Law Enforcement, IT	3	0	3
Delivery & Logistics	Routing and tracking delivery parcels	Retail, Supply Chain & Logistics	2	0	2
Social Network Analysis	Find the most viral users with maximum reach to other users	Social Media	2	0	2
Other Applications	Natural language question answering, Call graph analysis, Code analysis, Drug discovery, Traffic route recommendation	IT, Telecommunication, Traffic Management	3	2	5

Table 2.13: Application areas and example uses of graphs in various fields described in graph software whitepapers.

in-person interview. 33 participants had provided us with their email addresses and 4 of them agreed. To extend our interviews, we reached out to several of our contacts in major software companies and graph vendors. We did 4 additional in-person interviews; 2 developers and 2 users of graph processing software in major enterprises.

The occupations of our interviewees were as follows:

- Two IT consultants to several large enterprises on graph technologies.
- A developer of graph processing systems at Alibaba.
- A developer of graph processing systems at Siemens.
- A principal scientist at Amazon working on knowledge graphs.
- Engineers from a contact management company called FullContact [20], an electric utility company called State Grid [43], and a startup called OpenBEL [33] that develops data publishing tools for biologists.

We led the interviews with an open-ended question asking the interviewee to walk us through a concrete business application that uses graph data. The developers explained the applications of their customers. We asked questions about the details of the graph data, the computations they run, and the processing software they use in their applications. In addition, we asked if the interviewees used graph visualization and performed incremental computations on their graphs.

2.4.2 Overall Observations

We make four observations:

- None of the applications that used graphs representing transactional business data used a graph database or an RDF store as the main system of record. In each case, a relational system was the main system of record and the transactional data was replicated to a graph software for the application to use. This gives a sense of where graph databases and RDF systems are in the IT ecosystem of our interviewees' enterprises.
- Interviewees mentioned visualizing graphs in data exploration, debugging, query formulation, and as a presentation tool within the enterprise, e.g., to show a manager the benefits of modeling an application data as a graph.
- Several interviewees mentioned processing highly dynamic graphs and buffering a window of several hours or days of these graphs. However, the computations in those applications were batch computations. For example, in one case, 3 days of business data would be copied over into a graph software to search for subgraph patterns. None of the interviewees mentioned using a readily available system with incremental computation capabilities for these computations. All of the solutions were custom-made solutions.

Overall there were similarities between the applications described by our interviewees and those from the whitepapers but we also discovered some new applications. We cover one such new application called contingency or perturbation analysis in the next section. This application will be one of the motivating applications for the research presented in Chapter 4.

2.4.3 Contingency Analysis of Power Failures at StateGrid

Contingency analysis is a preemptive analysis done on an electric power grid to check the severity of different possible failures. Our interviewee from StateGrid described a contingency analysis system designed for the grid in one Chinese province. In contrast to other applications which often use one large graph, this application, logically, uses a very large number of small graphs. Interestingly, these graphs are very similar to each other and the application repeats the same computation on each graph in parallel. We describe the input graph, the computation, and the software used by the application.

Input Graph: The application has a *base graph* that represents the components of the power grid using the abstract *bus-branch* standardized model [92]:

- Vertices correspond to buses that represent electrical nodes, which can include power system elements like substations, loads, and generators. Operational parameters such as bus ID, load power, voltage magnitude, voltage angle, self-impedance, and power injection are stored as vertex attributes. There are approximately 2.5K vertices.
- Edges correspond to branches that represent electrical paths for current flows, such as transmission lines and transformers. Operational parameters such as power flow, line impedance, and transformer turn ratio are stored as edge attributes. There are approximately 3K edges.

This is a dynamic graph. The attributes on the edges change every few seconds and the topology changes every few minutes, e.g., when a new node is added or removed.

Computation and Software Used: To determine how the failure of a component affects the flow of power in the grid, the application generates a few thousand logical *derived graphs* from the base graph. Each derived graph perturbs the base graph slightly, say by removing a single edge, to simulate a potential failure. For each derived graph G', the application formulates some power equations. We do not provide the details of these equations, but an overview can be found in reference [170]. In a simplified form, readers can think of these as equations of the form $Ax = B$, where A and B are power-related matrices, each row of which represents information about the neighborhood of a vertex in the derived graph. These equations are solved in parallel using matrix factorization. We note that there is a significant potential to reuse the computation results across the derived graphs, as the graphs are very similar. However, the solution our interviewee described does not share any computation and runs the same computation over different snapshots separately. The base graph is stored in Tigergraph [44]. Derived graphs are logical and not explicitly stored but their corresponding matrices A and B are read from Tigergraph in parallel and moved to a custom code that solves the power flow linear algebra equations. The solution x's are analyzed to assign severity values to each derived graph and an alert is raised for abnormally high severity values, indicating the system has found a potential failure case, which could have severe outcomes.

2.5 Related Work

To the best of our knowledge, our survey is the first study that has been conducted across users, and of a wide spectrum of graph technologies and various public information about these technologies, to understand graph datasets, computations, and software that is in use, the business applications that use graphs, and the challenges users face.

Several surveys in the literature have conducted user studies to compare the effectiveness of different techniques used to perform a particular graph processing task, primarily in visualization [65, 101] and query languages [106, 135, 137]. Additionally, several software vendors have conducted surveys of their users to understand how their software is used to process graphs. Some of these surveys are publicly available [1, 6, 10]. However, these surveys are limited to studying a specific software product.

There are also numerous surveys in the literature studying different topics related to graph processing. Examples include surveys on query languages for graph database systems and RDF engines [54, 95, 102], graph algorithms [49, 100, 110, 164], graph processing systems [57, 120], and visualization [79, 161]. These surveys do not study how users use the technologies in practice.

2.6 Summary

Managing and processing graph data is prevalent across a wide range of fields in research and industry. We surveyed 89 users, interviewed 8 users, and reviewed user emails, code repositories, and whitepapers of a large suite of software products. The participants' responses and our review provide useful insights into the types of graphs users have, the software and computations users use, the business applications they develop, and the major challenges users face when processing their graphs. Some of our findings have motivated the research that will be presented in the following chapters. Specifically, the survey results indicate that many users have applications that perform incremental computations on very large and dynamic graphs. Yet, there are also no readily available systems that can be easily used for developing these applications. The core technical capability of systems that can perform incremental computations is the ability to share computation across different snapshots of a graph. The contingency analysis application that we reviewed, although not an application on a dynamic graph, can also benefit from the same capability of sharing computations across snapshots. The use of DC and DD systems to provide this core capability to support such applications efficiently and at scale is the motivation for the technical research conducted in this thesis.

3

Differential Computation

In this chapter, we cover the background of the following topics:

- The dataflow abstraction we assume. Here we also introduce the multi-dimensional timestamps that identify different points in time in dataflows and how recursive loops that run until a fixed point are constructed.
- The DC incremental maintenance technique.
- The iterative frontier expansion subroutine, which is the common dataflow computation that is used in several popular graph computations, including those used in this thesis.
- The Timely Dataflow and Differential Dataflow systems, on top of which the reference implementation of DC is based.

For reference, the details of the background we give on dataflows, timestamps, and DC can be found on the original publications on DC [47, 124].

3.1 Dataflows and Timestamps

We assume that computations are expressed as dataflows that consist of operators that consume one or more input *collections*, and output a collection that in turn can be an input to other operators. Collections can contain any generic data d, where usually $d = (\texttt{key}, \texttt{value})$. But as explained momentarily, they are modeled in an extended manner in DC. We assume that operators work on partitions of collections, execute independently on each key, and produce an output, as done in standard dataflow-based data processing systems, such as Spark [166] or

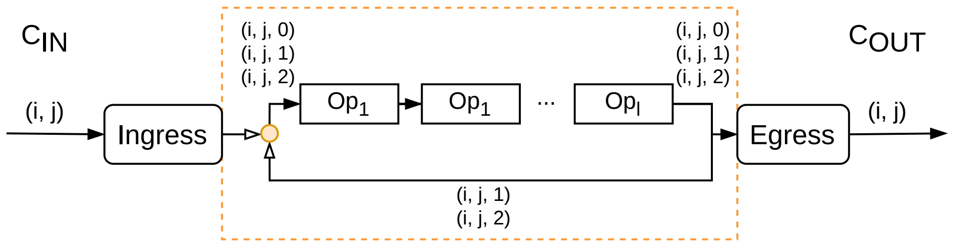


Figure 3.1: Example loop within a dataflow. Ingress and egress operators respectively add and remove the last dimension of the timestamps inside the loop.

MapReduce [80]. Some operators consume "external" collections that are input from outside of the dataflow and can change over time. Different points in time in the computation are identified by timestamps. We assume that a central scheduler analyzes the input and output connections in the dataflow and ensures that all operators in a dataflow make progress by processing their inputs. Once there are no operators to execute, the dataflow has finished its computation.

Initially, the timestamps for the source operators that consume data from external collections start at $t = (0)$. This first dimension of the timestamps is incremented by one each time the user updates external collections. New dimensions to the timestamps are automatically added within *loops*, the constructs within which recursive computations execute. The outputs of loops are fed back into the loop as inputs until a fixed point is reached. Figure 3.1 shows an example loop construct of a dataflow. The loop takes collection C_{IN} as input and outputs C_{OUT}. Loops start with an *Ingress* operator and ends with an *Egress* operator. Ingress extends the time (i, j) by adding a new dimension with its value initialized to 0, i.e., the time in the inner loop starts at (i, j, 0). In the loop, the output of the loop is fed back into the loop. The actual implementation has two other operators that we omitted from the figure for simplicity: a Feedback operator that duplicates the output of the last operator (the one before Egress) and a Concat operator that combines Feedback's output with the original input from Ingress to give to the first operator of the loop. In Figure 3.1, the loop runs 3 times, at timestamps (i, j, 0), (i, j, 1), and (i, j, 2), before a fixed point is reached. For each output, Egress removes the last dimension, such that the timestamp for the operator after Egress, i.e., to the right of Egress outside of the loop, executes at (i, j) again. Note that the fact that time is at (i, j) indicates that the loop shown in Figure 3.1 is a nested loop within another loop that is running at iteration j and the external input collections have received i updates.

To summarize, time "moves forward" in two ways. First, time moves within loop bodies as explained above. Second, the application developer can notify the scheduler that some new inputs have arrived at the dataflow. This increments the very first dimension of the timestamps by 1. For example, the next time external collections are updated, time would move to $(i{+}1, 0)$. In practice, this operation is done by the application managing the dataflow by externally signaling the dataflow scheduler that new inputs are available at $(i{+}1)$.

The dataflow abstraction we covered is very general and can model both batch computations with arbitrarily nested loops, as well as streaming computations. That is, as long as new data arrives at the system, the first dimension can continue incrementing in an unbounded manner.

3.2 Differential Computation (DC)

DC is a very general incremental maintenance technique that can maintain the outputs of any arbitrary computation that can be modeled in this dataflow model. There are two components to how DC works. First, DC stores every input and output collection for each operator in a dataflow in each timestamp that has been executed in the form of *differences*. Second is the *DC maintenance procedure* which consists of two rules that decide when an operator `Op` in the dataflow should re-execute to maintain `Op`'s output collection when external inputs to the dataflow change. We cover each component in turn following the overview of DC from reference [52].

Differences: Instead of keeping a copy of each collection C_t at each timestamp t in the computation, DC stores these values compactly as a set of differences δC_t, such that the version of a collection at any timestamp can be reconstructed. Differences extend data tuples d = (`key`, `value`) to triples as ((`key`, `value`), t^*, Δ), where T_i is the timestamp and Δ is the multiplicity of d at time T_i, normally represented by signed integers. For external collections, +Δ and -Δ represent insertions or deletions to the base data collections. For intermediate collections that are generated by the operators, the multiplicities may not have a clear interpretation. We will give examples of these non-obvious differences later in this thesis.

Collections support multiset operations. Specifically, the *union* of two collections $C_1 + C_2$, also called the CONSOLIDATE operation, adds the multiplicities of each unique ((`key`, `value`), T_i) tuple present in both C_1 and C_2. Tuples with 0 multiplicity are removed. $C_1 - C_2$ is also a union operation, except the multiplicity of the differences in C_2 is first flipped to the opposite sign.

The +Δ or -Δ assigned to differences ensure that the union of all collections δC_i prior to and including timestamp t gives exactly C_t. Consider an operator with one input and one output collection, I and O, respectively. DC ensures that the following equations hold for the operator:

$$I_t = \sum_{s \le t} \delta I_s \;\Rightarrow\; \delta I_t = I_t - \sum_{s<t} \delta I_s \tag{3.1}$$

$$O_t = \mathtt{Op}(\sum_{s \le t} \delta I_s) \;\Rightarrow\; \delta O_t = \mathtt{Op}(\sum_{s \le t} \delta I_s) - \sum_{s<t} \delta O_s \tag{3.2}$$

DC uses Equations 3.1 and 3.2 to compute which differences should be stored in δI_t and δO_t for each timestamp. These differences are logically kept for each timestamp that the dataflow has executed at. Note that for many timestamps, many (`key`, `value`) pairs will have empty differences and do not need to be stored. These differences are stored in an index and different versions of a collection at different timestamps t can be constructed by DC's maintenance procedure when needed, as explained next.

DC's Maintenance Procedure Let C_t^k indicate the differences in C_t's partition for key k. DC reruns an operator `Op` using Equation 3.2 according to two rules:

(i) *Direct rerunning rule:* If `Op`'s input I has differences at timestamp τ for a key k, i.e., δI_τ^k is non-empty, DC first reruns `Op` on key k at τ, i.e., DC reassembles $I_\tau^k = \sum_{t \le \tau} \delta I_t^k$, then executes `Op`$(I_\tau^k)$, which computes a new O_τ^k. DC then computes the difference set δO_τ^k as $\delta O_\tau^k = O_\tau^k - \sum_{t<\tau} \delta O_t^k$.

(ii) *Upper bound rule:* DC may also rerun `Op` at particular timestamps ω, even if `Op`'s inputs have no changes at ω. Specifically, DC finds all timestamps $t_i \nless \tau$ in which `Op`'s input has differences for key k and reruns `Op` on timestamps ω that are in the "join", i.e., least upper bound, of t_i and τ. For example, if DC reran `Op` at τ=$(1,2)$ and k has input differences at t_i=$(0,5)$, then DC will also rerun `Op` at ω=$(1,5)$. Note that because timestamps may be multi-dimensional, we use the symbol $\nless$ instead of $>$. Multi-dimensional timestamps are only partially, and not fully, ordered and DC uses this property to correctly operate on nested and iterative computations.

Reference [47] formally proves that applying these two rules, to decide which operators to rerun and compute differences for, can maintain any dataflow computation correctly.

Although based on very simple principles, DC is a general incremental maintenance technique. Specifically, since it can maintain arbitrary dataflows with arbitrary loop structures, it is suitable for maintaining complex dynamic graph computations, which can be singly and even doubly nested. That is why it is a promising technique to be the algorithmic foundation of systems built to support applications that operate on dynamic graphs. DC can be efficient because often, upon updates to a graph, no differences will be computed for many keys' inputs. For example, consider running a connected components algorithm on a dynamic graph. When a small set of updates is applied to a graph, it may affect only a small set of vertices, which would be the keys in a dataflow, and DC can automatically detect and localize the changes, without rerunning the entire computation. In the next section, we give several example dataflow computations and how DC would maintain these computations upon updates, demonstrating the effectiveness of DC. At the same time, if updates to a graph lead to many changes, then DC may need to rerun the operators in a dataflow for many keys, which would be very inefficient.

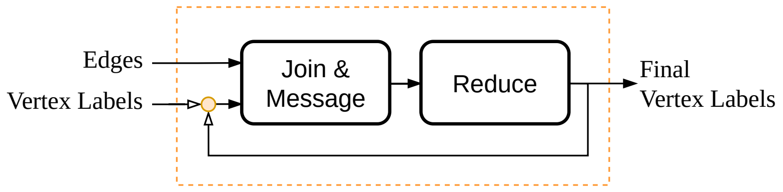


Figure 3.2: The generic IFE dataflow that appears as a subroutine in the graph computations covered in this thesis.

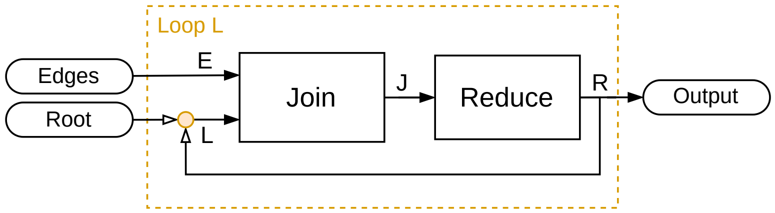
If we model the updates to a graph as snapshots, intuitively DC is an efficient technique if the input and output collections that would be generated across different snapshots are similar. This can be equivalently stated as follows: DC can be an efficient technique if the computational footprint of a dataflow, as captured by the state of the collections that it generates, is similar across different snapshots of its external input collections.

3.3 Iterative Frontier Expansion Dataflow (IFE)

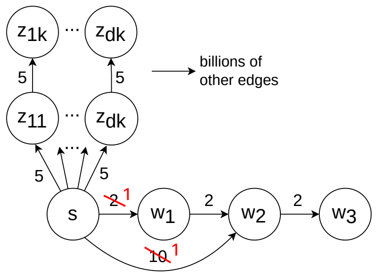
We next describe the IFE dataflow, which is a common subroutine in the graph computations that will be used in this thesis. IFE is the dataflow expression of the common vertex-centric paradigm that is commonly used to parallelize graph computations. The vertex-centric paradigm works as follows. Initially, each vertex in an input graph starts with a label. Then, iteratively, vertices aggregate their neighbors' labels and compute a new label for themselves until a fixed point of vertex labels is reached. Depending on the computation, sometimes incoming and sometimes outgoing neighbors could be used to aggregate vertex labels. Figure 3.2 shows this paradigm as a dataflow. In the dataflow, a `Join and Message` (henceforth `Join`) and a `Reduce` operator are in a loop. In this figure and the rest of the dataflow figures we use in this thesis, we omit the Egress and Ingress operators for simplicity. `Join` implements the logic of each vertex v "messaging" its neighbors with v's label. `Reduce` implements the logic of aggregating. IFE is a singly-iterative computation, i.e., contains a single loop. Therefore, the timestamps within the IFE loop are 2-dimensional, where the first dimension identifies the number of updates that happen to the graph and the second dimension identifies the iterations in the loop. We next give several examples of IFE dataflows implementing specific graph computations.

3.3.1 Single-source Shortest Paths (SSSP)

We begin with the Bellman-Ford [78] algorithm to compute SSSP on a possibly weighted directed graph implemented as a dataflow computation. We also use this example to simulate DC. In this computation, a source s is identified and starts labeling itself with value 0, while all other vertices



(a) Dataflow of the Bellman-Ford SSSP algorithm.



(b) `Edges` (`E`).

Figure 3.3: Example dataflow for computing SSSP and an input graph.

label themselves with ∞. These labels indicate the latest distances found to each vertex. Then iteratively until a fixed point, each vertex v takes the minimum of each of its in-neighbor w's latest distance plus the weight on the $w{\to}v$ edge, and updates its own latest distance. The SSSP dataflow is shown in Figure 3.3a. `Join` takes two input collections, `Edges` (`E`) and `Distances` (`L`), and outputs the messages between vertices as collection `J`. `Reduce` aggregates the join output and produces collection `R`, which is fed back into the loop as `L`.

Now consider running this dataflow on the graph shown in Figure 3.3b. Vertex s in the graph is the source vertex. Consider further the graph being updated first by changing (s, w_1)'s cost from 2 to 1 and then (s, w_2)'s cost from 10 to 1. Table 3.1 shows the differences that would be generated by DC. The columns represent graph snapshots and the rows represent IFE iterations. The input graph contains billions of edges among the z_{jk} vertices, and we summarize the difference sets relevant to them as δZ_E, δZ_D, and δZ_M, without showing the individual differences between them in Table 3.1. Moreover, we only start with $\delta L_{0,0} = [(s, 0, +1)]$ and omit all other vertices—the absence of a label implies ∞. Now consider that $\delta L_{(0,1)}$ contains a $(w_1, 2, +1)$ capturing the fact that after the first messaging between vertices, w_1's latest known distance improves from (an implicit) ∞ to 2. Similarly, consider $\delta J_{(1,0)}$, i.e., at the first iteration after the graph has been updated to G_1. There are differences $(w_1, 2, -1)$ and $(w_1, 1, +1)$, which capture the change in the messages "sent" from s to w_1 in the very first iteration, i.e., iteration 0, of the SSSP computation, reflecting the change in edge weight between s and w_1 in the graph.

Importantly, observe that maintaining computations differentially can be very efficient. Indeed, for graph snapshots G_1 and G_2, Table 3.1 shows all of the merely 30 updates that DC generates, even though the graph contains billions of edges. This is because DC automatically notices that the results of computations working on partitions related to vertices z_{ij} effectively could not have changed after the updates. As a result, DC does not rerun any computation for those vertices. Therefore, DC effectively shares virtually the entire SSSP computation, except for generating and keeping track of minor differences in this example billion-scale graph.

		G_0	G_1	G_2
0	δE	$(s, (w_1, 2), +1)$, $(s, (w_2, 10), +1)$, $(w_1, (w_2, 2), +1)$, dZ_E	$(s, (w_1, 2), -1)$, $(s, (w_1, 1), +1)$	$(s, (w_2, 10), -1)$, $(s, (w_2, 1), +1)$
	δL	$(s, 0, +1)$	$\varnothing$	$\varnothing$
	δJ	$(w_1, 2, +1)$, $(w_2, 10, +1)$, dZ_M	$(w_1, 2, -1)$, $(w_1, 1, +1)$	$(w_2, 10, -1)$, $(w_2, 1, +1)$
1	δE	$\varnothing$	$\varnothing$	$\varnothing$
	δL	$(w_1, 2, +1)$, $(w_2, 10, +1)$, dZ_D	$(w_1, 2, -1)$, $(w_1, 1, +1)$	$(w_2, 10, -1)$, $(w_2, 1, +1)$
	δJ	$(w_2, 4, +1)$, $(w_3, 12, +1)$, dZ_M	$(w_2, 4, -1)$, $(w_2, 3, +1)$	$(w3, 12, -1)$, $(w_3, 3, +1)$
2	δL	$(w_2, 10, -1)$, $(w_2, 4, +1)$, $(w_3, 12, +1)$, dZ_D	$(w_2, 4, -1)$, $(w_2, 3, +1)$	$(w_2, 10, +1)$, $(w_2, 3, -1)$, $(w_3, 12, -1)$, $(w_3, 3, +1)$
	δJ	$(w_3, 12, -1)$, $(w_3, 6, +1)$, dZ_M	$(w_3, 6, -1)$, $(w_3, 5, +1)$	$(w_3, 5, -1)$, $(w_3, 12, +1)$
3	δL	$(w_3, 12, -1)$, $(w_3, 6, +1)$, dZ_D	$(w_3, 6, -1)$, $(w_3, 5, +1)$	$(w_3, 5, -1)$, $(w_3, 12, +1)$
	δJ	dZ_M	$\varnothing$	$\varnothing$
...		rest contains dZ_D, dZ_M	rest is $\varnothing$	rest is $\varnothing$
k		...	...	...

Table 3.1: SSSP differences for the example graph. $\delta E = \varnothing$ and is omitted after iteration 0.

3.3.2 Multiple Pair Shortest Path (MPSP)

The MPSP algorithm finds the shortest paths between multiple (src, dst) pairs. Figure 3.4 shows the dataflow for MPSP implementing the bidirectional variant of the Bellman-Ford algorithm. The *src* and *dst* vertices from all pairs independently start with a label 0. Iteratively, the minimum distance is propagated to the neighbors from both the forward and background directions, along with metadata identifying the query pair to which a propagated distance belongs. Pairs whose *src* and *dst* meet are filtered out as output, and the iterations continue until either all pairs have been matched or there are no more edges left to process, indicating that unmatched pairs do not have a path between them. Note that the dataflow for MPSP contains 3 pairs of `Join` and `Reduce` operators instead of the single pair in IFE. However, the semantics of the `Join` and `Reduce` pair in the IFE that we rely on in this thesis remain the same for the 3 pairs. Similar to SSSP, this computation also uses 2-dimensional timestamps.

3.3.3 Weakly Connected Components (WCC)

The WCC algorithm finds all subgraphs of a given graph, such that each vertex in a subgraph is reachable from every other vertex by some undirected path. The WCC IFE is very similar to the SSSP IFE except the graph is undirected and edges are not weighted. We omit its dataflow figure. Briefly, vertices start by labeling themselves with their own IDs and iteratively update their

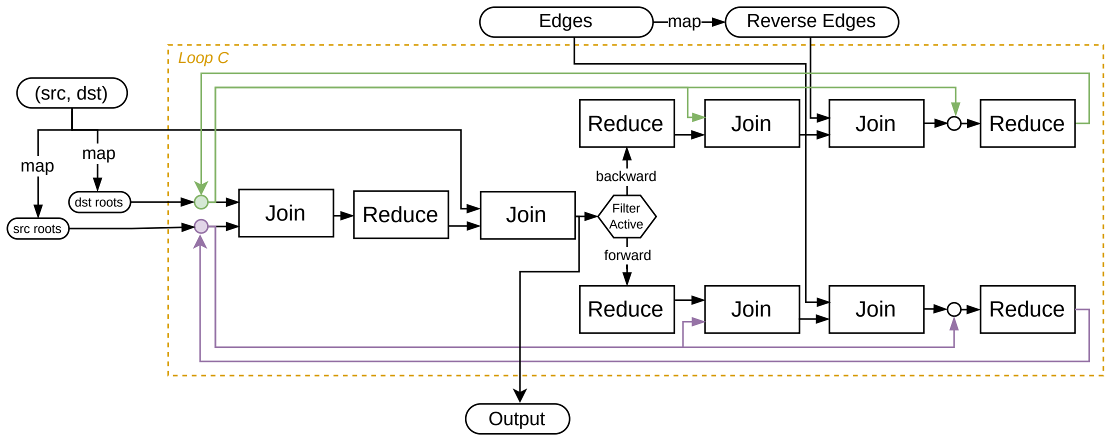


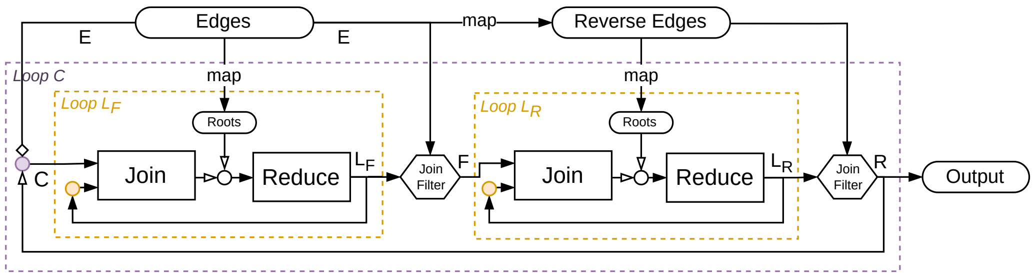
Figure 3.4: Example dataflow for computing MPSP.

labels to the minimum (Reduce operator) of their labels and the latest labels of their neighbors. When the computation reaches a fixed point, each vertex v is identified with the minimum vertex ID in v's weakly connected components. Similar to SSSP and MPSP, this computation also uses 2-dimensional timestamps.

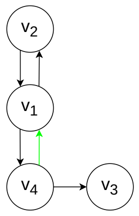
3.3.4 Strongly Connected Components (SCC)

The SCC algorithm finds the subgraphs of a given graph, such that each vertex in a subgraph is reachable from every other vertex by a directed path. We describe a variant of the *Coloring* algorithm by Orzan [132] to compute the SCCs of a directed graph G. Our goal is to present an example computation that requires 3-dimensional timestamps. In this variant of Coloring, the algorithm removes the cross-SCC edges from the graph iteratively until no edges can be removed. When the computation ends, only the edges that are local to an SCC remain. This will be based on the forward and backward "coloring" iterations. First, using only the forward edges of G, a WCC-like label propagation computation assigns labels to each vertex. If two adjacent vertices $u{\longrightarrow}v$ get different labels, then the $u{\longrightarrow}v$ is removed. Then using only the transpose of the edges in G, the same computation is repeated. These forward-backward coloring phases are repeated until we can no longer remove any edges. If one prefers to assign each vertex a label that identifies its SCC, a final label propagation, similar to the WCC computation, can be applied after the cross-SCC edge removal.

The dataflow of this computation is shown in Figure 3.5a. Recall that we identify loops with a box. In the figure, the large box corresponds to the outer loop that continues the forward-backward label propagations until there are no edge removals. Timestamps at the operators



(a) Dataflow of the SCC Coloring algorithm.



(b) Edges (E).

Figure 3.5: Example dataflow for computing SCCs and an input graph.

outside the large box are 1-dimensional, corresponding only to graph versions. Timestamps at the operators inside the large box but outside of the nested inner boxes are 2-dimensional. Finally, the operators inside the nested inner boxes are 3-dimensional (i, j, k), where i is the graph version G_i, j is the current forward-backward pass P_j, and k is the IFE iteration I_k. The inner boxes correspond to the loops of the forward and backward label propagations. Consider running this dataflow on the input graph shown in Figure 3.5b. In the graph, the black edges are the original edges for graph G_0 and the green $v_4 \rightarrow v_1$ edge is an insertion that is performed to obtain graph G_1. For reference, Tables 3.2a and 3.2b show the differences generated when executing the SCC dataflow on the G_0 and G_1, respectively. The columns in each table now represent the forward-backward passes. The differences for the IFE iterations are not shown. Observe that the 4 output differences δR for G_0 are present in two passes P_0 and P_1. The final output for G_0 can be computed as $R_0 = \textsc{consolidate}([(v_1, v_2, +1), (v_2, v_1, +1), (v_4, v_3, +1), (v_4, v_3, -1)]) = [(v_1, v_2, +1), (v_2, v_1, +1)]$, which are the two edges belonging to the single SCC in G_0. Similarly, the output for G_1 is the union of all 8 differences in δR for both G_0 and G_1, resulting in the 4 edges belonging to the two SCCs in G_1.

3.4 A Note on the Differential Implementation of Operators

We next discuss how certain differential operators are implemented. One can imagine a DC implementation that takes operators as a black box, simply monitors the inputs and outputs of the operators, and applies the maintenance procedure of DC using Equation 3.2. For instance, the Reduce operator operates on a single input δJ_t and produces output δR_t by following the equation. However, this can be inefficient, as the output differences of some operators can be computed more efficiently. In particular, we note the following two specializations:

	P_0	P_1	P_2
δC	$(v_1, v_2, +1)$, $(v_1, v_4, +1)$, $(v_2, v_1, +1)$, $(v_4, v_3, +1)$	$(v_1, v_2, +1)$, $(v_1, v_2, -1)$, $(v_1, v_4, -1)$, $(v_2, v_1, +1)$, $(v_2, v_1, -1)$, $(v_4, v_3, +1)$, $(v_4, v_3, -1)$	$(v_4, v_3, -1)$
<*Loop* L_F>	3 inner IFE iterations with timestamps $t = (G_0, P_{j\in[0,3)}, I^1_{k\in[0,3)})$ to compute *forward* edges labels.		
δL_F	$(v_1, 1, +1)$, $(v_2, 1, +1)$, $(v_3, 1, +1)$, $(v_4, 1, +1)$	$(v_4, 1, -1)$, $(v_4, 4, +1)$, $(v_3, 1, -1)$, $(v_3, 3, +1)$	$\varnothing$
δF	$(v_1, v_2, +1)$, $(v_2, v_1, +1)$, $(v_1, v_4, +1)$, $(v_4, v_3, +1)$	$(v_1, v_4, -1)$, $(v_4, v_3, -1)$	$\varnothing$
<*Loop* L_R>	3 inner IFE iterations with timestamps $t = (G_0, P_{j\in[0,3)}, I^2_{k\in[0,3)})$ to compute *reverse* edges labels.		
δL_R	$(v_1, 1, +1)$, $(v_2, 1, +1)$, $(v_3, 3, +1)$, $(v_4, 3, +1)$	$(v_4, 3, -1)$, $(v_4, 4, +1)$	$\varnothing$
δR	$(v_1, v_2, +1)$, $(v_2, v_1, +1)$, $(v_4, v_3, +1)$	$(v_4, v_3, -1)$	$\varnothing$

(a) Differences for G_0.

	P_0	P_1	P_2
δC	$(v_4, v_1, +1)$	$(v_1, v_4, +1)$, $(v_4, v_1, +1)$, $(v_4, v_1, -1)$, $(v_4, v_3, -1)$	$(v_4, v_3, +1)$
<*Loop* L_F>	Elided inner iterations with timestamps $t = (G_1, P_{j\in[0,3)}, I^1_{k\in[0,3)})$ to compute *forward* edges labels.		
δL_F	$\varnothing$	$(v_4, 1, +1)$, $(v_4, 4, -1)$	$\varnothing$
δF	$(v_4, v_1, +1)$	$(v_1, v_4, +1)$	$\varnothing$
<*Loop* L_R>	Elided inner iterations with timestamps $t = (G_1, P_{j\in[0,3)}, I^2_{k\in[0,3)})$ to compute *reverse* edges labels.		
δL_R	$(v_4, 1, +1)$, $(v_4, 3, -1)$	$(v_4, 3, +1)$, $(v_4, 4, -1)$	$\varnothing$
δR	$(v_1, v_4, +1)$, $(v_4, v_1, +1)$, $(v_4, v_3, -1)$	$(v_4, v_3, +1)$	$\varnothing$

(b) Differences for G_1 after $v_4 \longrightarrow v_1$ insertion.

Table 3.2: SCC differences for the example graph. Differences in the IFE iterations are not shown.

Linear Operators

These operators have the property that $\texttt{Op}(\sum_{s\leq t}\delta I_s) = \texttt{Op}(\delta I_t) + \texttt{Op}(\sum_{s<t}\delta I_s)$. Since $\texttt{Op}(\sum_{s<t}\delta I_s)$ $= \sum_{s<t}\delta O_s$ and cancel out, Equation 3.2 simplifies to:

$$\delta O_t = \texttt{Op}(\delta I_t)$$

Thus, linear operators can directly compute δO_t from δI_t, completely avoiding the rerunning logic of Equation 3.2. Examples of such operators include: (i) `Map`, which transforms input differences by applying a user-defined function $f(data)$; (ii) `Filter`, which only outputs input differences that pass a predicate $p(data)$; and (iii) `Negate`, a DD specific operator used to flip multiplicities from +Δ to -Δ and vice versa.

The `Join` Operator

`Join` operates on two inputs `Edges` δE_t and `Labels` δL_t at timestamp t to produce output δJ_t. According to Equation 3.2, the differences in the output of `Join` is computed as:

$$\delta J_t = \text{Join}((\delta E_t + \sum_{s<t}\delta E_s), (\delta L_t + \sum_{s<t}\delta L_s)) - \sum_{s<t}\delta J_s$$

The `Join` operation can be distributed over the collections as:

$$\delta J_t = \text{Join}(\delta E_t, \sum_{s<t}\delta L_s) + \text{Join}(\sum_{s<t}\delta E_s, \delta L_t) + \text{Join}(\delta E_t, \delta L_t) + \text{Join}(\sum_{s<t}\delta E_s, \sum_{s<t}\delta L_s) - \sum_{s<t}\delta J_s$$

We can also show that $\text{Join}(\sum_{s<t}\delta E_s, \sum_{s<t}\delta L_s)$ is equal to $\sum_{s<t}\delta J_s$ and they cancel out, as follows. First, recall that in IFE, timestamps are 2-dimensional (i, j), where i corresponds to the graph updates and j corresponds to the IFE iterations. We consider the two cases, when j=0 and j>0, respectively. First, δE_t is only non-empty at j=0. $\text{Join}(\sum_{s<(i,0)}\delta E_s, \sum_{s<(i,0)}\delta L_s)$ is then exactly equal to $\text{Join}(\sum_{s\leq(i-1,0)}\delta E_s, \sum_{s\leq(i-1,0)}\delta L_s)$. This is the result of the join for the previous update $i-1$ and corresponds to output $\sum_{s\leq(i-1,0)}\delta J_s$. The importance of being able to convert the $<$ to $\leq$ in the timestamps is that, DC maintains the differences such that at any timestamp t, the outputs of operators are exactly equal to running the operator on the union of the differences up to and including time t ($O_t = \texttt{Op}(\sum_{s\leq t}\delta I_s)$ from Equation 3.2). This guarantees that $\text{Join}(\sum_{s<t}\delta E_s, \sum_{s<t}\delta L_s)$ is equal to $\sum_{s<t}\delta O_s$ at timestamps $(i, 0)$. Second, when $j > 0$, δE_t is empty because edges only change before the IFE iterations start. Therefore, $\text{Join}(\sum_{s<(i,j)}\delta E_s, \sum_{s<(i,j)}\delta L_s) = \text{Join}(\sum_{s\leq(i,j-1)}\delta E_s, \sum_{s\leq(i,j-1)}\delta L_s)$. Therefore, by the same argument as in the previous case, we can argue that Join($\sum_{s<t} \delta E_s$, $\sum_{s<t} \delta L_s$) is equal to $\sum_{s<t} \delta J_s$. Thus, Equation 3.2 can be simplified as:

$$\delta J_t = \text{Join}(\delta E_t, \sum_{s<t} \delta L_s) + \text{Join}(\sum_{s<t} \delta E_s, \delta L_t) + \text{Join}(\delta E_t, \delta L_t)$$

This simplification avoids the more expensive recomputation logic of Equation 3.2, by not having to recompute `Join` twice on two sets of combined inputs or process the previous outputs. Indeed, this is the `Join` operator implementation available in the Differential Dataflow system that we will review next. Another key takeaway from this observation is that in IFE computation `Join` is computationally cheap while `Reduce` is expensive.

3.5 Timely Dataflow (TD) and Differential Dataflow (DD)

DD [46] is a reference implementation of the DC algorithm to maintain dataflow computations. DD is implemented on top of TD [29, 127]. TD is a system for general, possibly iterative, data-parallel computations that are expressed as a combination of *timely operators*, such as `Map`, `Reduce`, and `Iterate`, that transform one or more input data collection streams to an output collection stream. TD is an inherently streaming system, similar to systems like Apache Flink [69], and be used to develop event-driven applications. Operators process their streams at different timestamps and get notified when certain timestamps have been processed by upstream operators, so they can proceed with any pending work they need to do before times move forward. Despite being a streaming system, TD can be used to implement bulk synchronous computations, since developers have the ability to notify TD when certain times have finished. These notifications can be used by operators to synchronize at these timestamps.

Dataflows in TD can contain loops as in the abstraction we described in Section 3.1. In dataflows with loops, timestamps become multi-dimensional vectors of integers, $<i_1, i_2, \ldots, i_k>$. i_j represents different nested iterations of the computation. Finally, similar to systems such as MapReduce and Spark, TD takes in a dataflow computation and can automatically scale the computation to multiple workers, within or across compute nodes, where each worker processes only a partition of input collections and produces a partition of an output collection stream.

DD is built on top of TD. The DD layer has two main components. First is the library of differential operators implemented in DD. These operators include `Map`, `Filter`, `Join`, `Concat`, and different versions of `Reduce`. The differential operators implement DC's maintenance procedure of Equation 3.2 or some optimized version of these operators. Using TD's default scheduler and these operators, users can compose programs that will automatically be maintained by DD. The second component of DD is an index to store the input and output differences of operators. We delay presenting this structure, called *Spine* in the DD codebase, to Chapter 5, where we will discuss optimizations to how this index can be architected.

DD is available as an open-source library built using the Rust programming language. The systems and optimizations we present in this thesis are implemented on top of DD. However, we note that DC is a general algorithm that can be implemented using different techniques in different languages. For instance, in a separate work that is not part of this thesis, we implemented DC in a single-threaded Java system as part of our work in reference [52]. All of the optimizations we present in this thesis, except for the one that targets DD's index, can be applied to these other implementations as well.

4
Graphsurge: Graph Computations on View Collections Using Differential Computation

In this chapter, we describe the Graphsurge system. Although the motivating applications for DC in the publications that introduced it were on incremental maintenance of computations on dynamic inputs, Graphsurge is a system that uses DC for computations over multiple views of static graphs. Many applications require the ability to analyze different snapshots or *views* of a large-scale static graph, often based on selecting subsets of nodes or edges that satisfy different predicates. In Chapter 2 (Section 2.4.3), we reviewed the contingency analysis application from the StateGrid company, which performs a "what if" analysis on an electric grid network. Here, multiple and highly similar views of the electric grid, modeled as a graph, are constructed and the same computations are performed on each of these views. Each view models a possible perturbation scenario to the current state of the grid. Similar contingency analysis applications have also been described in the literature from many other fields, such as communication [156], transportation [103], or biological networks [165]. For example, in network analyses in neuroscience, scientists "lesion" anatomical or functional brain networks by deleting nodes or edges randomly or in a targeted way [68], e.g., by deleting subsets of the highest degree nodes, and study the effects of these lesions on the average path lengths between different nodes in these graphs.

Another popular example application that analyzes multiple views of static graphs is historical analyses of graphs where nodes or edges have some time property. A network scientist

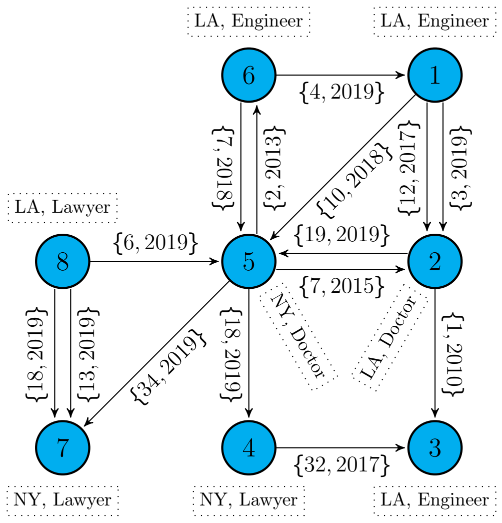


Figure 4.1: Example phone calls graph.

might study the history of the connectivity of the call graph from Figure 4.1 and compute one view of the graph for each year between 2010 and 2020. Similarly, the analyst can study the history of more complex views, where each view contains only the calls up to a certain duration, say for ≤ 1, ≤ 5, or ≤ 10 minutes. A classic example of such analyses from the literature is reference [118] that studied the component size, vertex degrees, and diameters of different time-windows in time-stamped citation and web graphs, and under different selection criteria of vertices, e.g., those belonging to a particular component or without incoming edges. In other settings, applications may study the history of social or e-commerce networks to find the trends in the centralities or importance rankings of nodes across different snapshots.

These and many other applications require constructing multiple, sometimes hundreds of, views of a static input graph, and running the same computation across each view. Without system support, users would need to resort to running these computations from scratch on each view, which can be very inefficient. A system that can share computation across views would be of immense use in enabling the efficient development of these applications. This chapter presents the GRAPHSURGE system, which is built for this purpose.[1] GRAPHSURGE is an computation system developed on top of Timely Dataflow (TD) and Differential Dataflow (DD) that treats graph views as first-class citizens. The system has a declarative view definition language called *GVDL*, using which users can define: (1) individual graph views; or (2) collections of graph views, called *view collections*. Users program GRAPHSURGE by writing batch computations using a dataflow-based API. When users execute their programs on view collections, GRAPHSURGE

[1]Code is available at https://github.com/dsg-uwaterloo/graphsurge.

automatically shares computation across the views to improve performance by leveraging *differential computation*, which can have significant performance benefits. For example, in a historical analysis application that analyzes the evolution of a Stack Overflow dataset over 5 years, running a strongly connected components algorithm from scratch takes 431s, while the same analysis takes merely 43s using Graphsurge (see Section 4.4.2).

Our approach is based on the observation that although Graphsurge processes static graphs, one can organize view collections as *edge difference sets* that represent them as an evolving graph. Specifically, Graphsurge first orders a view collection C with k views and gives each view an index $GV_1, \ldots, GV_k$ (discussed momentarily). Then, the system runs a TD program that first materializes GV_1 and for each remaining view GV_i, materializes only GV_i's *edge differences*, i.e., edge additions and deletions, compared to GV_{i-1}. This edge difference-based storage compactly materializes all of the views in the C and represents C as an evolving graph over k time steps. Finally, when running the same computation across the views of C, Graphsurge feeds the user program and the computed difference sets for each GV_i to DD, which shares computation across views internally by running the program differentially across the views.

Unlike streaming applications on DD or specialized graph streaming systems, such as GraphBolt [122], the static nature of the views defined in Graphsurge gives the system several interesting optimization opportunities, which we next discuss.

Collection Ordering Problem: Intuitively, once a view collection is ordered as consecutive edge difference sets, neighboring views that are more similar allows differential computation to share more computation across the views. In streaming or continuous query processing systems, a system has no choice over the order of the updates that come in, so effectively has no choice as to the order of the snapshots on which a computation has to be performed. Instead, the static nature of the views gives Graphsurge an opportunity to order the views as a preprocessing step and put similar views close to each other. We show that this problem is NP-hard, but show a constant-factor approximation algorithm that we have integrated into Graphsurge. In our evaluations, we show that our collection ordering optimization can lead to up to 10× runtime improvements when good orderings are unclear.

Collection Splitting Problem: Even after a system has found a good ordering that minimizes the differences between views and maximizes computation sharing, there are cases when differentially maintaining the computation for a view GV_j, given the differential output for $GV_0, \ldots, GV_{j-1}$, might be slower than rerunning GV_j from scratch. We call this the collection splitting problem, as rerunning the computation from scratch at GV_j effectively splits the view collection into 2 sub-collections, each of which would be run differentially (in the absence of further splittings). Several factors can trigger this behavior, such as the computation being executed may be unstable or the views may not be similar enough to benefit from differential

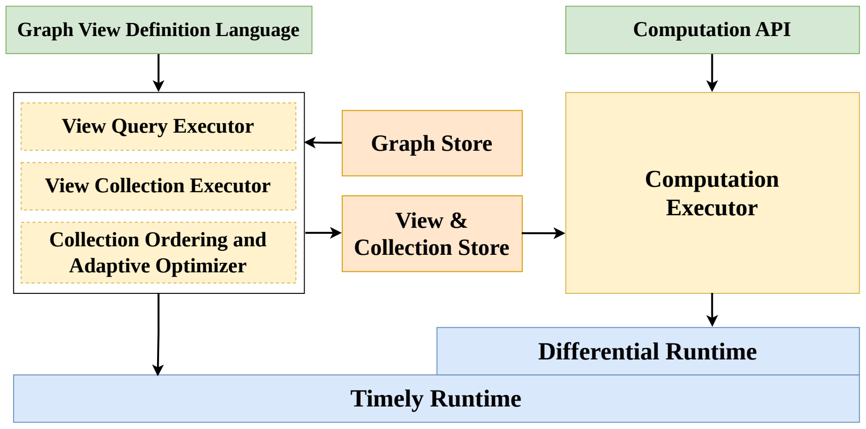


Figure 4.2: Graphsurge architecture.

computation sharing. We show that Graphsurge can monitor the performance characteristics of computations at runtime and make effective decisions about whether to run each view differentially or from scratch. In our evaluations, we show that our collection splitting optimizer can detect whether running views differentially or from scratch is optimal, leading to up to 1.9× performance improvements over the better of these baselines when neither is optimal.

In the rest of the chapter, we first present the system architecture of Graphsurge in Section 4.1, and then present the two optimization problems—collection ordering in Section 4.2 and collection splitting in Section 4.3. Finally, we present extensive experiments that evaluate Graphsurge in Section 4.4 and then summarize our contributions in Section 4.5.

4.1 The Graphsurge System

Figure 4.2 shows the architecture of Graphsurge. Users program Graphsurge through two interfaces: (i) a declarative *graph view definition language* (GVDL) to define individual views and view collections over base graphs; and (ii) a DD-based API to write dataflow programs for graph computations that consume the difference stream of a view or view collection. Graphsurge uses TD and DD as its execution layer for both creating views as well as for running user-specified computation programs on views. As such, dataflows written in TD and DD can be automatically parallelized both in a single multi-core machine and in a distributed cluster.

Users import *base input graphs* to Graphsurge through CSV files that contain the nodes and edges of the graph and their properties. Upon loading, nodes and edges are given unique 32-bit IDs and stored as a *node stream* and *edge stream* in the *Graph Store* (GStore), respectively. Each edge in an edge stream is a (eID, sID, dID, $\texttt{key}_1$, $\texttt{val}_1$, ...) tuple, where eID is the edge ID,

```
create view CA–Long–Calls on Calls
edges where src.state = 'CA' and dst.state = 'CA'
    and duration > 10 and year = 2019
```

Listing 4.1: Example GVDL view query.

`sID` and `dID` are source and destination node IDs that point to offsets in the node stream, and $\texttt{key}_i$ and $\texttt{val}_i$ are the key-value properties of the edge.

When a user runs a GVDL query, a TD program is executed that reads the edge stream from the GStore, applies the filter predicates to generate the difference sets for the corresponding view or view collection, and stores them in the *View and Collection Store* (VCStore). When the user runs a computation on a view or view collection, a DD program reads the difference sets from the VCStore to run the user-defined dataflow. In a distributed cluster, both GStore and VCStore are replicated across all the machines. Each thread of a running TD and DD dataflow operates on a logical partition of the edge stream when creating views or view collections, and of the difference sets when running a computation. The parallel running operators in the TD and DD dataflows perform read-only queries to GStore and VCStore and do not require any locks or coordination.

4.1.1 Individual Views

Individual View Definition

GVDL is a language to define views over base graphs. A GVDL query to create a view has a single `where` clause, applied as a predicate on an input graph (or another materialized view), that specifies the edges of the output view. Predicates can be arbitrary conjunctions or disjunctions and can access the properties of the source and destination nodes as well as the edges.

Example 1. Listing 4.1 shows a view an analyst can construct on our running example `Calls` graph. The view consists of calls made in California in 2019 with a duration > 10 minutes.

GVDL queries that define individual views are straightforwardly compiled into TD dataflow programs. The dataflow consists of a filter operator to apply the user-specified predicates to the edge stream and compute the difference sets. The output of the program is materialized as a stream in the VCStore.

The views that users can express in GVDL are noticeably simple. For instance, GVDL allows filtering nodes and edges but does not allow their aggregations. This is because we designed GVDL to be able to express the use cases we have explored in this chapter. In Section 4.1.2, we will discuss a second advantage of keeping GVDL simple when we discuss how to compactly store multiple views in a view collection.

```
pub trait GraphSurgeComputation {
   type Results;
   fn graph_computation(input_stream: &InputStream) -> Collection<Self::Results>
}
```

Listing 4.2: Differential Computation API.

Computations on Individual Views

Users write arbitrary DD dataflow programs to run computations on their views with the constraint that one of the inputs to the dataflow is the Graphsurge-specific difference stream for the view. Graphsurge exposes a Rust interface to users containing a `graph_computation` function, using which users can write arbitrary DD programs that are expected to return output associated with the vertices of a view, such as the connected component ID of each vertex in a connected components computation. Listing 4.2 shows the interface of the `graph_computation` function. Users invoke their programs through a separate command line and specify their `graph_computation` function and the view on which to run this function. Graphsurge's *computation executor* calls the users' `graph_computation` function to obtain a computation dataflow, and feeds the edge stream corresponding to the view into it. When the computation is executed on a single view, the entire edge stream is fed into the dataflow at once. Computations executed on a view collection are more involved and are described in the next section.

4.1.2 View Collections

View Collection Definition

To share computations across multiple views of the same graph, Graphsurge allows users to organize views in a *view collection*. A view collection organizes a set of views as a single timestamped[2] *edge difference stream* C, where each view corresponds to a state of the stream at a particular timestamp t.

Example 2. Listing 4.3 shows a simple demonstrative GVDL query defining a view collection with four views on our `Calls` graph. Each view including all calls within a range of edge IDs.

Graphsurge materializes the view collection described by a GVDL query in three steps. Below, we let p_j denote the predicate defining GV_j in a given view collection.

[2]We use the term timestamp to follow differential computations' terminology. This should not be confused with any application-specific "time" property, such as the `year` property we use in our running example.

```
create view collection call-analysis on Calls
  [GV₁: ID < 100],
  [GV₂: ID ≥ 50 and ID < 199],
  [GV₃: ID ≥ 10 and ID < 100],
  [GV₄: ID ≥ 60 and ID < 199]
```

Listing 4.3: Example GVDL view collection query.

Step 1. ***Edge Boolean Matrix Computation*:** For each edge e_i in the base graph and each view GV_j in the collection, Graphsurge runs the predicate p_j on e and outputs an *edge boolean matrix* (EBM) that specifies whether e_i satisfies p_j. This is an embarrassingly parallelizable computation and is performed by a TD dataflow.

Step 2. ***Collection Ordering*:** The goal of this step is to put views with a higher overlap of their edges next to each other so that the differences between neighboring views are smaller, and running a computation on the collection results in higher computation sharing. To achieve this, Graphsurge reorders the views in EBM so that views whose predicates satisfy highly overlapping sets of edges are adjacent to each other. As we will discuss momentarily, the goal of this optimization is to store the views in the collection in a more compact edge difference stream, i.e., using fewer edge differences. As we demonstrate in our evaluations, this optimization step can lead to significant performance benefits. The output of this step is the same EBM but possibly with a different column ordering. We defer the details of how collections are ordered to Section 4.2.

Step 3. ***Edge Difference Stream (EDS) Computation*:** Finally, Graphsurge takes the reordered EBMs and materializes the views in the view collection as an edge difference stream that is consistent with the semantics of difference sets of DC. Specifically, we treat the entire view collection C as an evolving input stream according to the order obtained in step 2. For simplicity, let $GV_1, \ldots, GV_k$ be the order of the views after step 2, so $C_t = GV_t$. Recall Equation 3.1 from Section 3.2, that according to DC semantics, the differences of a stream S at timestamp t is $\delta S_t = S_t - \sum_{s<t} \delta S_s$. So the edge difference of a view δC_t at time t is computed to ensure that $\delta C_t = GV_t - \sum_{s<t} \delta C_s$ equality holds. Specifically, the multiplicity of each edge e_i in δC_t is: (i) 0 if GV_{t-1} and GV_t both contain or both do not contain e_i; (ii) 1 if GV_{t-1} does not contain e_i and GV_t does; or (iii) -1 if GV_{t-1} contains e_i and GV_t does not. The contribution of each edge e_i to δC_t can be computed independently, so this is another embarrassingly parallelizable step.

Example 3. *Figure 4.3 shows an example EBM for the view collection from Listing 4.3. For example, GV_1 has 1 for all the edges e_0 to e_{99} and 0 for others since its predicate is $ID < 100$. Ignore the right side of the figure for now. On the left side, the figure also shows the EDS_{def} that corresponds to the default order of GV_1,GV_2,GV_3,GV_4. The first row of EDS_{def} for e_0-e_9 contains: (i) +1 for GV_1 because*

	EBM				$EDS_{\texttt{def}}$ (# = 540)				$EDS_{\texttt{opt}}$ (# = 260)			
edges	GV_1	GV_2	GV_3	GV_4	GV_1	GV_2	GV_3	GV_4	GV_3	GV_1	GV_2	GV_4
e_0-e_9	1	0	0	0	+1	-1	∅	∅	∅	+1	-1	∅
e_{10}-e_{49}	1	0	1	0	+1	-1	+1	-1	+1	∅	∅	-1
e_{50}-e_{59}	1	1	1	0	+1	∅	∅	-1	+1	∅	∅	∅
e_{60}-e_{99}	1	1	1	1	+1	∅	∅	∅	∅	∅	+1	∅
e_{100}-e_{199}	0	1	0	1	∅	+1	-1	+1	∅	∅	+1	∅

Figure 4.3: An example EBM for the view collection in Listing 4.3 and 2 EDS's for 2 different collection orders (Section 4.2).

GV_1 contains all of these edges; (ii) -1 for GV_2 because GV_2 does not contain any edges (so that the union of these differences with GV_1 gives the empty set); and (iii) ∅ for the rest of the views because they also do not contain these edges.

We end this section with a note on GVDL. Recall from Section 4.1.1 that we have limited the view queries users can express in GVDL to simple node and edge filter predicates. This ensures that each view over the same base graph contains a subset of a larger "ground truth" set of edges and that each view has the same set of node IDs, i.e., a node with ID u in a view GV_i maps to the node with ID u in a view GV_j. This allows GRAPHSURGE to easily compute an EBM for a collection, and the edge differences between the views from the EBM. If we allowed views that created new nodes, and we could not assume a node ID mapping between the views, the system could not easily compute an edge boolean matrix or difference stream, which is critical for us to store the views compactly and use differential computation when we run computations over view collections (discussed in the next section). One can extend GVDL to support more general *individual views* that can create new nodes and edges, e.g., those that form super nodes and edges, and run computations over these views. However, it would be challenging to store multiple such views compactly in a view collection as an edge difference stream, if the system cannot infer a mapping between the nodes across views.

Computations on View Collections

Given a computation program P that a user wants to run on all views of a view collection C, in the absence of any collection splitting, which is an optimization we describe in Section 4.3, the computation executor runs P as follows. First, the system runs P on C_0, i.e., the "first" view in C, and when this computation finishes, in an outside loop *advances* (in DD terminology) C to C_1 by feeding δC_1 to DD. Then the system feeds δC_2 to DD, and so on and so forth, until all views are evaluated. When computing P at each time t, DD will automatically share computation from the "prior" views on which P has been computed, in some cases leading to significant performance gains as compared to running P on each view from scratch. The output of the DD program is a set of *output differences* in the form of a `(VID, Results)` output stream. The output difference stream can then be stored or processed further by the user.

Support for dynamic graphs

While Graphsurge is built for applications that work with views of a static graph, it can also support analysis of dynamic graphs by ingesting a timestamped stream of updates to a graph and creating a view collection where the views represent batches of updates for different time windows given by a filter predicate on the edge timestamps. However, in our current implementation, the computations are not performed in a traditional streaming fashion and all of the data that a user wants to analyze needs to be fully ingested into Graphsurge before creating view collections and running computations on them.

4.2 Collection Ordering

Given a set of k views in a view collection C defined by an application, there are $k!$ different ways Graphsurge can order the views before running computations differentially on the collection. This is important because the number of edge differences that are generated in the final collection is solely determined by the order of the views. Recall from Section 4.1.2 that when running computations on a view collection C, Graphsurge iterates over neighboring views and for view t feeds in the difference set δC_t to DD (in absence of collection splitting). The smaller the size of the differences, the larger the structural overlap between view C_t and the union of the views prior to C_t, which we expect to lead to larger computation sharing. As we present in our evaluations, by picking orderings that minimize the set of differences, Graphsurge can improve performance significantly in certain applications. We can formulate this problem as a concrete optimization problem as follows:

Definition 1. *Collection Ordering Problem (COP):* Given a view collection C, find the collection ordering that minimizes the sum of the sizes of difference sets δC_t.

We next show that COP is NP-hard. Our proof is through a reduction from the *consecutive block minimization problem (CBMP)* for boolean matrices. In a boolean matrix B, such as the edge boolean matrix (EBM) in Figure 4.3, a *consecutive block* is a maximal consecutive run of 1-cells in a single row of B, which is bounded on the left by either the beginning of the row or a 0-cell, and bounded on the right by either the end of the row or a 0-cell. Given a column ordering σ for B, let $cb(B, \sigma)$ denote the total number of consecutive blocks in B over all rows. CBMP is the problem of finding the ordering σ^* of the columns of B that minimizes $\min_\sigma cb(B, \sigma)$. CBMP is known to be NP-hard [115].

Theorem 4.2.1. *COP is NP-hard.*

Proof. For an input graph G, let C be a view collection over G, and let the EBM of C be C_{EBM}. For a fixed column ordering σ of C_{EBM}, let the size of the difference sets in σ be $ds(C_{\text{EBM}}, \sigma)$. Therefore, COP is equivalent to the following problem on boolean matrices: given a boolean matrix C_{EBM}, find the σ^* that minimizes $\min_\sigma ds(C_{\text{EBM}}, \sigma)$. Recall from step 3 of view collection materialization (Section 4.1.2) that the difference set for an edge e, which is represented by a row r in C_{EBM}, is calculated as follows: for the first appearance of e from left to right, i.e., for the first 1 in r, we count one difference. Then for each subsequent alternating appearance of a 0, then 1, then 0, etc., we count one additional difference. Note that this is different from the definition of a consecutive block. For example, a row `1110` has 1 consecutive block but 2 diffs: one diff for the first view, and one diff for the last view.

Our reduction is from CBMP. Given a $k_1 \times k_2$ matrix B to CBMP and its complement B^C, consider (in poly-time) constructing a $2k_1 \times k_2$ matrix B_{EBM} that contains B and B^C under B. B^C contains 1s where B contains 0s and vice versa. Note that for each row r of B, both r and r^C appear in B_{EBM} exactly once. Let B_0, B_1, and B_{01} be the set of rows in B that contain only 0s, only 1s, and both a 0 and a 1, respectively, and let $|B_0| = m_0$, $|B_1| = m_1$, and $|B_{01}| = m_{01}$. Given an arbitrary column ordering σ, each row in B induces the following differences in B_{EBM}:

- Row r in B_0 yields 0 but r^C yields 1 difference.
- Row r in B_1 yields 1 difference but r^C yields 0 difference.
- Row r in B_{01} requires analyzing two cases. Let $cb(r, \sigma)$ denote the number of consecutive blocks only in row r. (i) If r's last cell is a 0, then r yields $2cb(r, \sigma)$ and r^C yields $2cb(r, \sigma) - 1$ differences; and (ii) otherwise r yields $2cb(r, \sigma) - 1$ and r^C yields $2cb(r, \sigma)$ differences. Therefore, in either case, r yields $4cb(r, \sigma) - 1$ differences.

Therefore, $ds(B_{\text{EBM}}, \sigma)$ is: $(\sum_{j \in B_{01}} 4cb(r, \sigma) - 1) + m_0 + m_1$, which is equal to $4cb(B, \sigma) - m_{01} + m_0 + m_1$. This establishes a one-to-one connection between the sizes of the difference sets in B_{EBM} and the number of consecutive blocks in B under any ordering σ. Since for any B, m_0, m_1, and m_{01} are fixed, finding the optimal ordering σ^* that minimizes $ds(B_{\text{EBM}})$ also minimizes $cb(B)$, completing the proof that solving COP is NP-hard. □

We next describe a 3-approximation to COP, which uses a 1.5-approximation algorithm for CBMP from reference [96], $\text{CBMP}_{1.5}$, which we next review. $\text{CBMP}_{1.5}$ takes as input an $m \times k$ boolean matrix B, creates the matrix $0B$ by padding a 0-column and then transforms $0B$ into a $(k+1)$ clique G^{0B}, where each column (so each view in our case) is a node, and the weight between the nodes is the Hamming distance of the columns they represent. Reference [96] shows that G^{0B} satisfies the triangle inequality and the entire transformation from B to G^{0B}

Algorithm 1: Collection Ordering Optimizer

```
input: Edge Boolean Matrix $B_{m\times k}$, W workers
output: A column ordering $\sigma^*$
begin
    Partition $B_{m\times k} \longrightarrow \bigcup_{i=0}^{W-1} B_i$;
    begin At each worker $w_i$, $0 \leq i < W$:
        $C_i \longleftarrow [0|B_i]$;
        $U \longleftarrow$ unit matrix;
        $D_i = C_i^T(U - C_i) + (U - C_i)^T C_i$;
        Shuffle $D_i$ to worker $w_0$
    begin At worker $w_0$:
        Receive $D_i$ from all workers $w_i$;
        $D \longleftarrow \sum_{i=0}^{W-1} D_i$;
        $G^{0B} \longleftarrow$ complete graph $(|V| = (k+1))$ induced from $D$;
        $\sigma^* \longleftarrow$ tsp_christofides($G^{0B}$);
        Broadcast $\sigma^*$ to all workers $w_i$;
```

is approximation preserving. Therefore, solving the Traveling Salesman Problem, with the well-known Christofides algorithm [76] yields a 1.5-approximation to CBMP after removing the 0-column from the tour, which gives a chain between the remaining columns to get an ordering.

Corollary 4.2.1.1. *Running CBMP$_{1.5}$ on the EBM of a view collection gives a 3-approximation algorithm for COP.*

Proof. To see this, consider any input C_{EBM} to COP and any ordering σ for C_{EBM}. Because each row r contains either $2cb(r,\sigma)-1$ or $2cb(r,\sigma)$ differences, $cb(C_{\text{EBM}},\sigma) \leq ds(C_{\text{EBM}},\sigma) \leq 2cb(r,\sigma)$. Therefore, since Christofides algorithm returns a 1.5-approximation algorithm for CBMP, it returns a 3-approximation for COP. □

Algorithm 1 shows our collection ordering optimizer. Given EBM $B = C_{\text{EBM}}$ as input, we construct G^{0B} using a TD program that performs the padding and then in an embarrassingly parallel way finds the Hamming distances between each view. Then we collect the G^{0CB} in a single TD worker and run the Christofides algorithm. The output of this algorithm gives 2 possible orders, one for each direction of the chain and either is a 3-approximation. However,

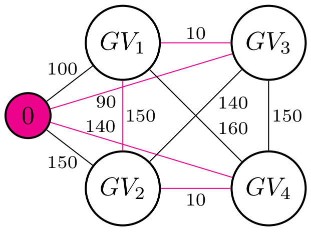


Figure 4.4: G^{0B} of the collection from Listing 4.3. Purple lines are the output tour/order of the Christofides algorithm.

these orders do not necessarily yield the same number of differences, and we pick the order with the smaller differences[3].

Example 4. *Figure 4.4 shows the example G^{0B} corresponding to the view collection from Listing 4.3, whose EBM was shown in Figure 4.3. For example, the weight of the edge between GV_1 and GV_3 is 10 because there are only 10 edge differences between these two views, specifically GV_1 contains edges e_0 to e_9, while GV_3 does not and the views overlap on other edges. The red lines in the figure show the TSP tour that the Christofides algorithm outputs. Taking the 0 node out of the tour gives us a chain, where the GV_3,GV_1,GV_2,GV_4 order is the better of the two possible orders. The EDS EDS_{opt} that corresponds to this optimized order is shown on the right side of Figure 4.3, which reduces the number of differences of the input order from 540 to 260.*

Our collection ordering technique materializes each view in a collection. An interesting question is whether a good ordering can be obtained through only inspecting the definitions of the views and without inspecting and materializing the views. This can be possible for example when there is a containment relationship between the predicates defining the views, in which case the best order follows the containment order. For example, a system can infer that a view defined by the predicate "year < 2010" is contained within the view defined by "year < 2011". Although general query or view containment [74] is a hard problem [71, 151], prior literature has identified cases when it can be determined, e.g., when the predicates are certain conjunctive queries [48, 72]. In cases when views are arbitrary, we do not know of any technique to find a good ordering without inspecting the data in the views.

4.3 Collection Splitting

Even after we find a good ordering that minimizes the sizes of the difference sets generated, running computations on each view differentially may not be ideal. If the computation footprint

[3]We note that these orders would return the same value for CBMP.

|Difference Sets|	Algorithm	`diff-only`	`scratch`
1K	**BFS**	**1.4s**	13.5s
	PR	**66.5s**	136.2s
3.5M	**BFS**	**13.0s**	25.7s
	PR	281.9s	**193.2s**

Table 4.1: Runtimes of BFS and PR for two view collections on the Orkut graph, containing 1K- and 3.5M-size difference sets, in two ways: (i) `diff-only`; and (ii) `scratch`.

of P on GV_i and GV_{i+1} are very different, differentially fixing P on GV_{i+1} might be slower than running P on GV_{i+1} from scratch.

Several factors determine how big the difference is between P's footprint on two consecutive views GV_i and GV_{i+1}, which determines how expensive it is to compute GV_{i+1} differentially. Two of these factors can be observed by Graphsurge: (1) how stable is P? (2) how large are the difference sets? We use the term unstable to refer to computations that may generate a lot of output differences even with small differences in input datasets. We next demonstrate these factors through a controlled experiment. We will also demonstrate this on a more realistic application in Section 4.4. We take 10M edges from the Orkut social network graph and construct an initial view GV_1 and then construct two artificial view collections each containing 20 views: (i) C_{1K}, in which we randomly add 500 edges and remove 500 edges to each GV_{i-1}; (ii) $C_{3.5M}$, in which we add 2M edges and remove 1.5M edges randomly to each GV_{i-1}. The sizes of the difference sets are picked to obtain a collection with highly similar and highly different views, respectively. We then run Breadth First Search (BFS) and PageRank (PR) on both the collections in two ways: (i) `diff-only`: runs the collection only differentially; and (ii) `scratch`: runs each view in the collection from scratch.

Table 4.1 shows the runtimes. First, notice that on $C_{3.5M}$, while it is better to run BFS differentially, it is better to run PageRank from scratch. This is because PageRank is a less stable algorithm than BFS. For example, assume $GV_{i+1} = GV_i \cup \{u \rightarrow v\}$, so the views differ by a single edge addition, and consider differentially updating the first iteration of BFS. This addition results in 1 difference in the `Join` operator. In vertex-centric terms, it will result in u sending 1 more extra message to v containing u's current distance. In contrast, in PageRank, u sends a message of $1/deg(u)$ to its neighbors, so all the messages that u sends might change after the update. Second, observe that when the views are sufficiently similar, specifically in C_{1K}, running PageRank differentially also starts to be the better option. That is, the size of the differences also determines whether running views differentially vs from scratch is the better option.

We have implemented an adaptive optimizer that decides whether to run each view GV_i in a view collection differentially or from scratch. Our optimizer observes two simple runtime metrics to make its splitting decisions: (i) Each time the system decides to split the collection at GV_i and run GV_i from scratch, we measure how long it took to compute P on GV_i from scratch and what was the size of G_i; and (ii) Each time the system decides to run GV_i differentially, we keep track of how long it took to run P and what was the size of δC_i. Then, using two linear models, the optimizer estimates how long it would take to rerun GV_i from scratch and differentially, respectively, given the sizes of GV_i and δC_i, and picks the faster predicted option. Specifically:

1. Run GV_1 from scratch and GV_2 differentially and keep track of ($|GV_1|$, st_1), for **s**cratch **t**ime, and (δC_2, dt_2), for **d**ifferential **t**ime.
2. For each view GV_i for $i = 3, \ldots, k$, estimate the run time of running GV_i from scratch or differentially using the previously collected data.

In our actual implementation, we make splitting decisions for ℓ views at a time (10 by default) as feeding multiple views to DD makes DD's data indexing code run faster. We will demonstrate that our optimizer can adapt to running computations differentially or from scratch, depending on which option is superior, and can even outperform both options in some cases by selectively splitting collections in a subset of the views.

We next discuss an important question: how much faster can an algorithm P running differentially on a view collection C be compared to running P on each view from scratch (and vice versa)? A high-level answer should instruct the benefits we can expect from adaptive splitting. Consider a k-view collection C, where each view is identical. This is conceptually the best case for running P on C differentially, where after the first view, the rest of the views are computed instantaneously. Therefore, differentially computing P can be *k factor better* than running P from scratch. Interestingly, the situation is not similar in the reverse direction. The worst case for running P on C differentially is if each view was completely disjoint, i.e., $\delta C_i = -GV_{i-1} + GV_i$. We effectively completely remove GV_{i-1} and add GV_i. Therefore, when running P on GV_i differentially, DD, to the first approximation, will "undo" computation for GV_{i-1} and then run P from scratch differentially. We effectively compute P on each view twice, and should expect a bounded, around 2x, slow down to running computations differentially even in this worst case. This is an important robustness property of running computations differentially. It is still important to perform our splitting optimization because: (i) there can still be a significant performance gain over pure differential computation (we will report up to 1.9× improvements); and (ii) some unstable computations consistently perform better when computed from scratch and our splitting optimization automatically detects those cases.

4.4 Evaluation

We next present our experiments. Section 4.4.2 starts by empirically demonstrating the possible performance gains of running computations differentially across views vs running them from scratch. Section 4.4.3 and 4.4.4, respectively, evaluate the benefits of our collection splitting and ordering optimizations. Section 4.4.5 presents baseline comparisons between DD and the GraphBolt [122]. Finally, Section 4.4.6 presents that Graphsurge obtains good scalability across compute nodes in a cluster.

4.4.1 Experimental Setup

Datasets: We evaluate Graphsurge on 5 real-world graphs. `size` below indicates the size of each dataset on disk.

- **Stack Overflow** [38] (SO, |V| = 2M, |E| = 63M, size = 1.6GB) is a temporal dataset where every edge has an associated UNIX timestamp indicating its creation time.
- **Paper Citations** (PC, |V| = 172M, |E| = 605M, size = 14.8GB) is a paper-to-paper citation graph constructed from the Semantic Scholar Corpus [53] (version 2019-10-01). The vertices have 2 associated properties: the year of publication and the count of co-authors.
- **Com-Livejournal** [37] (CLJ, |V| = 4M, |E| = 34M, size = 1.1GB) is a social network graph containing a list of ground-truth communities representing social groups that a subset of the users are part of. Users can be part of multiple communities.
- **Com-Wiki-Topcats** [39] (WTC, |V| = 1M, |E| = 28M, size = 719MB) is a web graph whose vertices can belong to one or more communities representing the category of a web page.
- **Twitter** [116] (TW, |V| = 42M, |E| = 1.5B, size = 25GB) is a social network graph.

Computations: We use 6 different computations: (i) weakly connected components (WCC); (ii) strongly connected components (SCC); (iii) breadth-first search (BFS); (iv) single source shortest path (SSSP); (v) PageRank (PR); and (vi) multiple pair shortest path (MPSP). For BFS and SSSP, we set the source to a random vertex that has outgoing edges. For MPSP, we select 5 pairs of vertices (`src`, `dst`) that have a path between `src` and `dst` of length equal to the diameter of the graph. All computations are implemented using Graphsurge's DD-based computation API.

Hardware and Software: We compiled Graphsurge using `rustc` v1.46.0, `timely-dataflow` v0.11.0, and `differential-dataflow` v0.11.0 and performed our experiments on a cluster of up to 12 machines each running Ubuntu 18.04.3. Each machine has 2× Intel E5-2670 @2.6GHz CPU with 32 logical cores. Every machine has 256 GB RAM, except 2, which have 512GB RAM. Except for our scalability experiments, all experiments were performed on a single machine.

4.4.2 Comparison of Differential Computing vs Rerunning from Scratch

Recall our observation from Section 4.3 that while differentially computing P can be unboundedly faster than running from scratch, the reverse comparison is bounded. We start by demonstrating this intuition empirically. We model a historical analysis application, where we build two sets of view collections on the SO dataset:

(i) C_{sim}: are a set of **s**imilar view collections that each start with a 5-year window of the graph, from May 2008 to May 2013, which forms the first view. Then we set a time window of size w of 1 day, 1 month, 6 months, 1 year, and 2 years, and *expand* the initial window by w, so each view GV_i includes GV_{i-1} plus an additional number of edges for a larger w-size window. This generates 5 collections. $\text{C}_{\text{sim:1d}}$ contains the most similar and largest number of views. $\text{C}_{\text{sim:2y}}$ is the least similar and contains the fewest number of views.

(ii) C_{no}: are a set of **n**on-**o**verlapping, so highly different views, where we start with a window of the graph from May 2008 till December 2008, then we completely *slide* the window by a window of size w of 6 months, 1, 2, 3, and 4 years. This generates 5 collections, all of which are completely non-overlapping. The window size w allows us to create collections with increasingly more views.

We evaluated the performance of 6 computations on each collection, turning our splitting and ordering optimizers off, in two ways: `diff-only` and `scratch`, which we described in Section 4.3. We expect `diff-only` to be more performant than `scratch` in each C_{sim} collection, but increasingly more as w gets smaller and there are a larger number of views. We expect `scratch` to be more performant in each C_{no} collection, but we do not expect to see increasingly more gains as the number of views increases. Figures 4.5 show our results for the C_{sim} and C_{no} collections, respectively. Observe that in C_{sim} collections, indeed as w gets smaller, we see an increasing factor on benefits for `diff-only` varying from 1.5× to 13.9×. The only exception is PR, which we observed is not as stable as the rest of our computations. In contrast, in the C_{no} collection, we see up to 2.5× performance improvements for `scratch`, but we do not observe improved factors with an increasing number of views.

4.4.3 Benefits of Collection Splitting

We next evaluated Graphsurge's adaptive splitting optimizer, continuing our previous setup. We kept our ordering optimizer off to only study the behavior of our adaptive optimizer, which we refer to as `adaptive`. We reran the previous experiment with `adaptive`. The `adaptive` bar in Figure 4.5 shows our results. Except for two experiments, running BFS and SSSP on C_{no} with 6-month slides, `adaptive` can perform as well or almost as well as the better of `diff-only` or

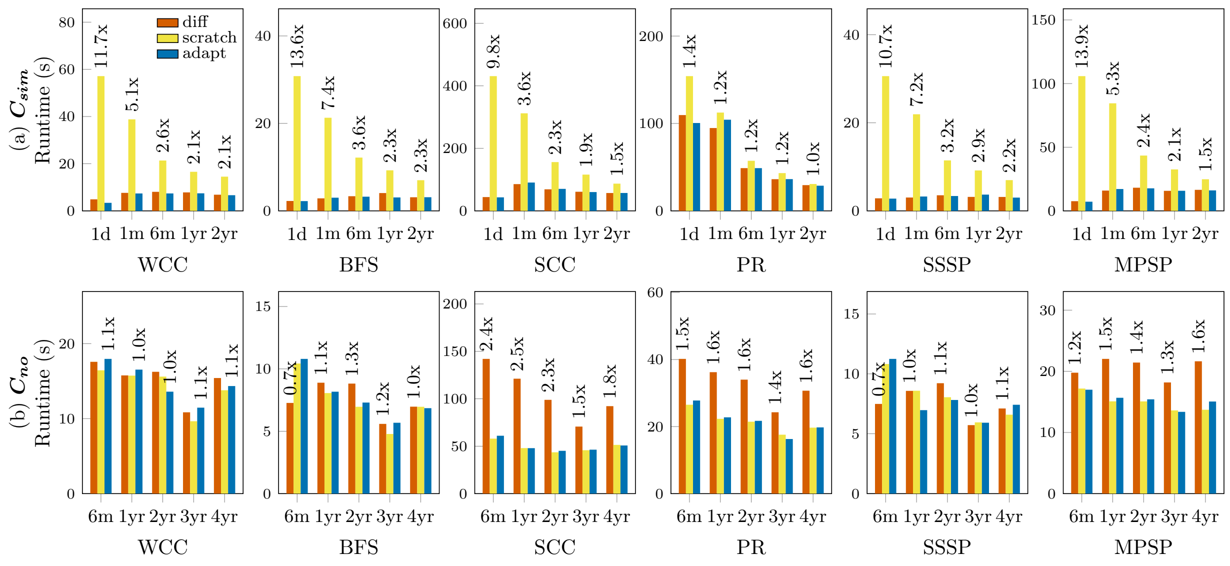


Figure 4.5: Runtimes of computations running on views differentially, from scratch, or adaptively.

scratch. Note that in these experiments, it is always better to either run the computations with one of diff-only or scratch, so we do not expect adaptive to outperform both of these strategies. Importantly, in almost all cases, we adapt to the better strategy.

Next, we created a view collection in which adaptive can outperform both diff-only and scratch. Specifically, we created a view collection C_{aut} on the PC citation dataset, which contains the Cartesian product of two sets of windows on two properties. First is a 5-year non-overlapping window from $[1996, 2000]$ to $[2016, 2020]$. The other is a window for the number of **aut**hors on the papers, that expands from $[0, 5]$ to $[0, 25]$ in windows of size 5. For example, the view $[1996, 2000] \times [0, 5]$ is the view that contains all papers written between 1996 and 2000 containing at most 5 authors and their citations. This collection contains views that generate a sequence of addition-only differences as the number of authors window expands, and then a non-overlapping view, when the year window slides, creating a potential splitting point. Table 4.2 shows the runtimes of 6 computations on C_{aut}. Observe that adaptive matches or outperforms, by up to 1.9x, the better of diff-only and scratch. adaptive is able to appropriately split the computation when the year window slides and consistently outperforms diff-only and scratch when running all the computations.

	WCC		BFS		SCC	
diff	117.26	(1.9×)	19.29	(1.4×)	314.78	(1.8×)
scratch	120.53	(1.9×)	44.20	(3.3×)	351.83	(2.0×)
adapt	61.88		13.54		179.27	
	PR		**SSSP**		**MPSP**	
diff	79.07	(1.6×)	18.6607	(1.3×)	20.4278	(1.2×)
scratch	114.74	(2.3×)	42.3927	(3.0×)	43.7398	(2.6×)
adapt	50.13		14.347		16.6355	

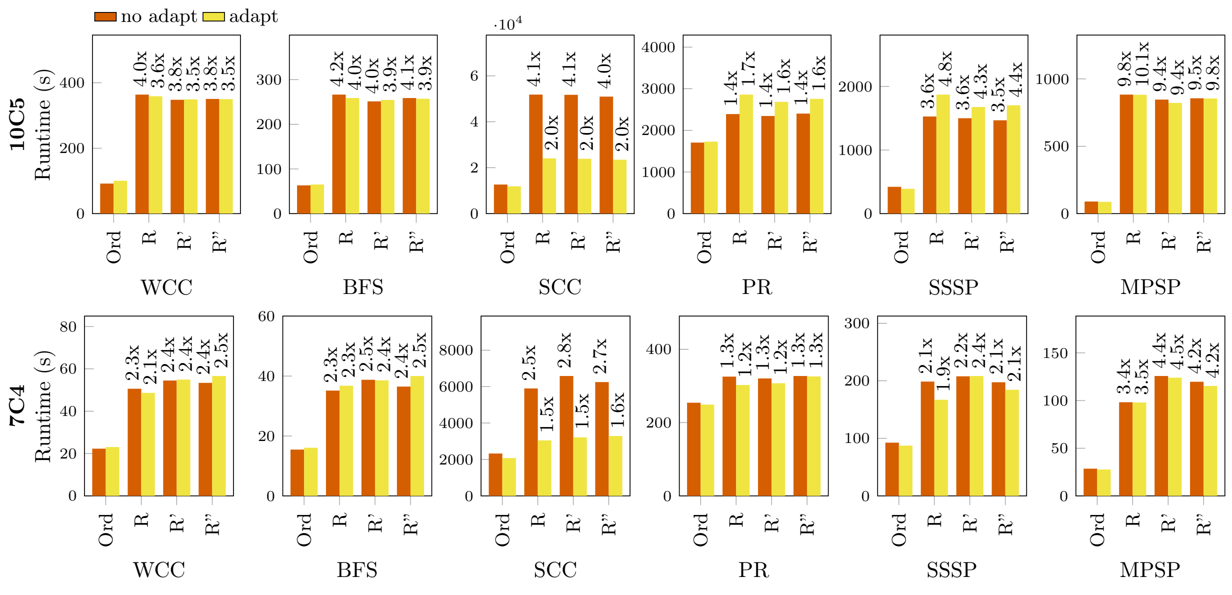
Table 4.2: Runtimes (s) of computations for the C_{aut} view collection showing that the adaptive optimizer can outperform both running differentially and from scratch.

4.4.4 Benefits of Collection Ordering

The goal of our next experiment is to study the performance gains of our collection ordering optimization. We developed a perturbation analysis application on CLJ, which contains ground-truth communities. We constructed view collections by taking the N largest communities and removing k combinations of these N communities to perturb the graphs in a variety of ways. Specifically, we constructed collections for two (N, k) combinations: $C_{10,5}$ sets N=10 and k=5 and contains 252 views, and $C_{7,4}$ sets N=7 and k=4 and contains 35 views. Note that this is an application where finding a good manual order is difficult, as each view removes possibly millions of edges, and there are hundreds of views in the collection. Therefore as a baseline, we will use random collection orderings.

We first turned our adaptive splitting optimizer off to isolate the benefits due to collection ordering only and compared the performance of the order that Graphsurge picks, which we call `Ord`, with 3 random orderings, denoted by `R`, `R'`, and `R"`, respectively. We then executed 6 computations on the view collections. The `no adapt` bars in Figure 4.6 show our results. Table 4.3 presents the amount of total edge differences in our edge difference sets. Observe that: (i) our optimizer's order generates between 3.4× to 16.8× fewer differences than the random orders; and (ii) our ordering optimization improves performance consistently and between 1.3× to 9.8× across our experiments. For reference, Table 4.3 also reports the times it takes Graphsurge to compute the collections, with and without ordering, in row CCT (collection creation time). The difference between the random order's CCT and `Ord`'s CCT is the overhead of ordering, which ranged between 1.3× and 1.8×.

We next turned the adaptive splitting optimization on to measure the performance benefits in the full system. This forms a full end-to-end experiment as both of our optimizations are turned on. We expect the benefits of ordering to decrease when adaptive splitting splits and improves the performance of the random ordering. If adapting defaults to running only differentially, we


(a) CLJ dataset.

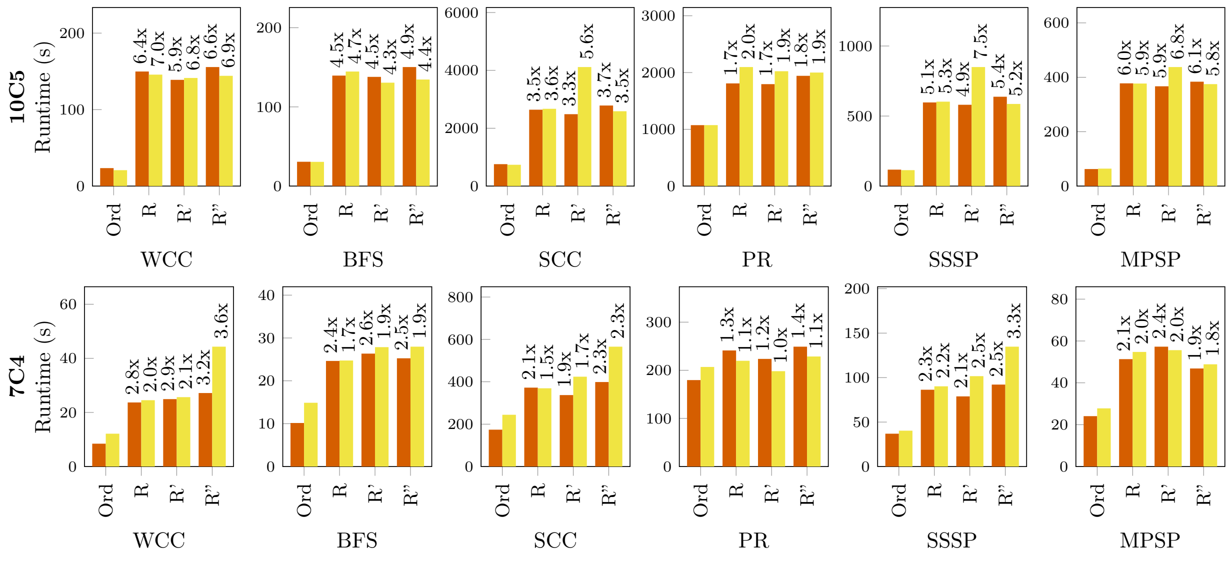

(b) WTC dataset.

Figure 4.6: Runtime of algorithms showing the benefits of collection ordering with adaptive splitting turned on and off.

			Ord		R1		R2		R3	
CLJ	10C5	# Diffs	**158M**		**1.5B**	(9.6×)	**1.4B**	(9.6×)	**1.5B**	(9.6×)
		CCT	355.0	(+151.5)	203.5	(1.7×)	207.8	(1.7×)	236.0	(1.7×)
	7C4	# Diffs	**54M**		**191M**	(3.6×)	**211M**	(3.6×)	**194M**	(3.6×)
		CCT	38.8	(+8.7)	30.2	(1.3×)	31.4	(1.3×)	34.7	(1.3×)
WTC	10C5	# Diffs	**73M**		**1.2B**	(16.8×)	**1.1B**	(16.8×)	**1.1B**	(16.8×)
		CCT	299.0	(+128.5)	170.5	(1.8×)	168.9	(1.8×)	192.3	(1.8×)
	7C4	# Diffs	**44M**		**149M**	(3.4×)	**149M**	(3.4×)	**192M**	(3.4×)
		CCT	35.1	(+8.5)	26.6	(1.3×)	25.6	(1.3×)	30.0	(1.3×)

Table 4.3: The number of diffs and collection creation time (CCT) in seconds for $C_{10,5}$ and $C_{7,4}$ on CLJ and WTC for three random orders R1, R2, and R3, and our optimizer's order.

expect the results to be similar to our previous results. The `adapt` bars in Figure 4.6 show our results. There are 3 experiments in which adapting improves the random order's performance by splitting: when running SCC on CLJ with both $C_{10,5}$ and $C_{7,4}$ collections and running SSSP on CLJ with $C_{7,4}$ experiment. In these cases, the benefits of ordering decrease compared to when adapting optimization was off. For example, when running SCC on CLJ with $C_{10,5}$ collection, the benefits of ordering decrease from 4.1× to 2.0×. In other experiments, our adaptive optimization defaults to running all computations differentially (or performs slightly worse than running only differentially). In these cases, the ordering optimization improves the performance similar to when adaptive optimization was off (between 1.1× to 10.1×).

4.4.5 Baseline Temporal Systems

In this section, we provide baseline comparisons against GraphBolt [122] (GB). GB is a shared-memory streaming system that is developed on top of Ligra [152] and designed to maintain computation results over a stream of updates. As such, we can develop a Graphsurge-like system on top of GB by feeding our view collections as an evolving graph to GB instead of DD. The primary difference between DD and GB, and the reason we chose DD, is that GB requires users to write explicit maintenance code in functions such as `retract` or `propagatedelta`, which is challenging for some algorithms, such as the doubly-iterative SCC algorithm.

The GB reference implementation [24] includes two incremental computation engines. The first is the Kickstarter engine [163], which is limited to monotonic graph computations and is used to implement SSSP. The second is the Graphbolt engine, which is more general and is used to implement PR.

We evaluate the performance of Graphsurge and GB for two computations, SSSP and PR, on the TW dataset. We simulate a temporal computation application in Graphsurge by

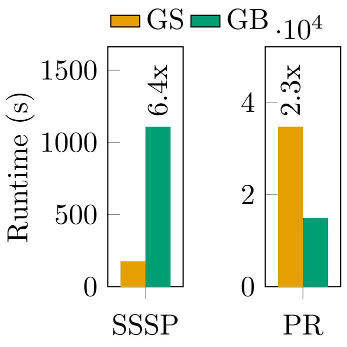


Figure 4.7: Graphsurge vs GraphBolt.

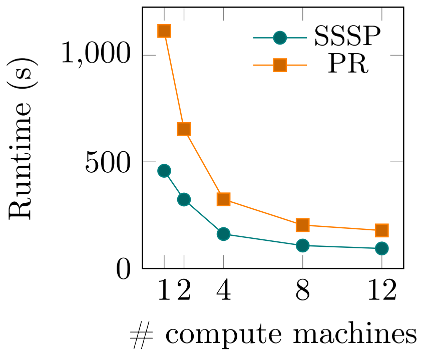


Figure 4.8: Runtimes in a distributed setting.

constructing a view collection with 1001 views, where the first view contains 50% of the total edges in the original graph selected randomly, and each of the remaining views contains 500 additions and 500 deletions based on the previous view. Figure 4.7 shows the comparison results. We note that Graphsurge is up to 6.4× faster than GB for SSSP, but is up to 13.5× slower than GB for PR. These numbers are similar to the numbers in reference [122] (Figures 8 and 9) for GB and DD.

There are two primary reasons for the lower performance of Graphsurge for PR. First, DD's execution engine uses a dataflow architecture, which is based on message passing, and has a higher runtime overhead as compared to the shared memory architecture of GB, which propagates updates by directly writing to memory locations using atomic operations. Despite this advantage, as reference [122], we found DD to be more performant on SSSP. Second, Graphsurge uses DD as its computation engine, which is built to support general incremental computation. However, this generality can naturally come at a performance cost, because DD is unable to take advantage of computation-specific optimizations. For example, GB implements a PR-specific incrementalization code, which is more efficient than differential computation. However, specialized incremental versions of many algorithms are similar to differential computation. For example, GB's incremental SSSP algorithm is effectively differential computation, and for such algorithms, DD generates equally efficient incremental versions automatically.

4.4.6 Distributed Execution and Scalability

We next demonstrate the ability of Graphsurge to scale in a distributed setting. We modified the Twitter dataset by assigning artificial city, state, and country attributes to the vertices and an affinity weight to edges that indicate the level of interaction between users. We modeled a social network analysis application that studies the connected users who live within the same city, state, and country with three different affinity levels, low, medium, and high, constructing a view collection with 9 views. We measured the runtime for 2 computations: SSSP and PR, on this view collection using up to 12 compute machines, each with 32 worker threads. Figure 4.8 shows the scalability results on this large view collection. Additional machines improve the runtime for both of the computations almost linearly. This experiment demonstrates that Graphsurge can take full advantage of TD and DD for seamlessly scaling to multiple machines in a distributed environment. Since this experiment aimed to simply verify that Graphsurge is able to utilize DD's ability to scale in a distributed setting, we did not run the rest of the 4 computations and expect them to exhibit similar runtime characteristics.

4.5 Summary

We presented the design and implementation of Graphsurge, an open-source view-based graph computation system, developed on top of the TD system and its DD layer. Graphsurge allows users to define arbitrary views over their graphs, organize these views into view collections, and perform arbitrary graph computations using a DD-based computation API. Graphsurge is motivated by real-world applications, such as perturbation analysis or analysis of the evolution of large-scale networks, that require capabilities to analyze multiple, sometimes hundreds of views of static input graphs efficiently. We presented two optimization problems, the collection ordering and splitting problems, for which we described efficient algorithms and studied the performances of our optimizations. Graphsurge's approach for computation sharing is based on differential computation. Within the context of this thesis, the research presented in this chapter demonstrates that one can make DC more practical for developers of graph applications through graph-specific interfaces over the DD system. Further, the chapter showed that DC has broader applications than dynamic graph computations. Finally, the collection ordering and splitting optimizations demonstrate that one can improve the performance of DC through application-specific optimizations. We will provide another graph-specific optimization directly for DC's maintenance rules in the next chapter.

5 Scaling Differential Computation for Large-Scale Graph Processing

The DD system, which is the reference implementation of DC, and which we used in Chapter 4, is a general dataflow system that is oblivious to the underlying computation. In the Graphsurge system, we showed that several graph applications can benefit from DC by designing: (i) graph-specific interfaces over its reference implementation DD; and (ii) graph-specific optimizations. These are the core arguments of this thesis. In this chapter, we show that we can also provide graph-specific optimizations to DC and DD directly to improve the performance of graph applications that can benefit from DC. All of our optimizations are motivated by the behavior of DC on the IFE dataflow.

We identify two optimizations to DD's in-memory index, which stores operator state. First, we observe that DD's index is a write-friendly implementation. But to check if operators need to generate differences, DC's core maintenance logic, based on Equation 3.2, does significant reads of differences. Therefore, we propose a new design that makes the storage of indexes more read-friendly. Second, DD stores differences, which consist of keys, values, timestamps, and multiplicities, indexed by keys and values. We show that by indexing by keys and timestamps instead, we can decrease the amount of data that is scanned from indices to temporary buffers to implement Equation 3.2.

Our third and most important optimization is *Fast Empty DIfference Verification* (FEDiV). It is an optimization for the Reduce operator in IFE when Reduce uses a min or max aggregation, which is the case in all of the graph algorithms we considered so far in the thesis, such as

WCC, MPSP, or BFS, except PR. Reduce is computationally one of the most expensive operators to maintain for DD. We show how to verify that such Reduce operators do not produce any differences at time t without inspecting any of the input differences to Reduce at times $< t$, by only inspecting the output differences. Because input differences to Reduce consist of the output of Join, they are much larger than the output differences. Therefore, when FEDiV is successful, i.e., it verifies that the Reduce's output is empty, it avoids both expensive scans out of the index and rerunning the Reduce logic.

5.1 Background

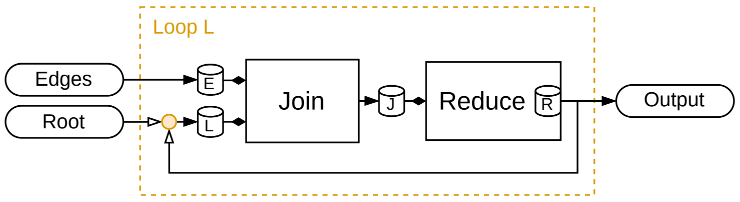
Recall from Section 3 that DD operates on collections that are a multiset of differences of the form ((key, value), t^*, Δ). DD operators process one or more input collections to produce an output collection based on Equation 3.2, which we can rewrite as:

$$\delta O_t = \mathtt{Op}(\delta I_t + \sum_{s<t} \delta I_s) - \sum_{s<t} \delta O_s \qquad (5.1)$$

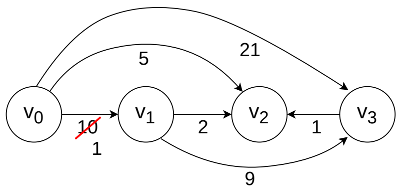
Equation 5.1 indicates that DD operators receive inputs δI_t at time t and produce output δO_t, by possibly requesting access to their input history $\sum_{s<t} \delta I_s$ and output history $\sum_{s<t} \delta O_s$, which DD indexes separately. To save memory, DD avoids automatically storing the input and output history for every operator because not every operator needs such access, as we discussed in Chapter 3 (Section 3.4). Instead, operators individually specify which of their input and output history should be indexed.

Collections are indexed in DD using the *arrange* operator [123]. Operators requiring access to their input or output history operate on the *arrangements* of the corresponding collection. For instance, Figure 5.1 shows the arrangements used by the operators in the dataflow of the SSSP algorithm from Chapter 3. In particular, Join requires arrangements of its two input collections E and L and produces an output collection J. Reduce, in turn, requires an arrangement of J as its input and maintains an arrangement of its computed outputs R.

Arrangements are not shared across workers. Instead, collections are sharded based on the key such that all differences for a particular key are indexed together in a single worker. Thus, read and write operations on an arrangement do not require any expensive coordination between workers. Within a worker, multiple operators can share an arrangement. For example, in Figure 5.1a, the arrangements R and L store the same set of differences (except for those of *Root*). We can rewrite the dataflow such that a single arrangement, say R, can be used by both Join and Reduce without affecting the computation semantics, thus avoiding a second copy of the index. Our implementation does this optimization, but we omit it in the figure for simplicity.



(a) Dataflow of SSSP with *arranged* collection indices.



(b) Edges (E).

Figure 5.1: Dataflow and input graph for SSSP.

When using arrangements, operators process input and outputs in units of a `Batch`. The history of a collection is represented using a set of Batches, which is organized in a data structure called the *Spine*. We next describe how Batches and the Spine are used to create arrangements, and then show how operators retrieve data out of the arrangements for their computations.

5.1.1 Arrangements

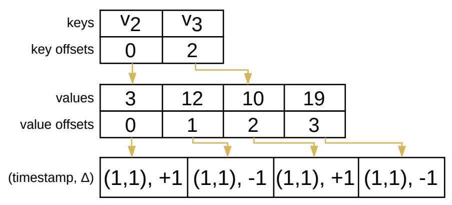
The `Arrange` operator takes a collection as its input and transforms them into a sequence of Batches, each representing a unit of work containing differences for one or more keys. Consider Table 5.2a, which shows the differences when running SSSP on the example graph in Figure 5.1b. For the input differences $\delta J_{1,1}$ at $t{=}(1,1)$, Figure 5.2b shows the corresponding Batch created by the `Arrange` operator. Each Batch is a composite two-level index, stored as a set of 5 arrays, mapping sorted keys and then sorted values to (t^*, Δ) pairs. Two offset arrays are used to indicate the associated entries in the next array.

As the computation progresses, operators create new output Batches at different timestamps. DD stores these batches inside the *Spine*, which stores an array of Batches, maintaining the invariant that an index i of the array stores a Batch of length 2^i, where the length of a Batch is the count of its (t^*, Δ) pairs. Incoming Batches are inserted at the appropriate index and existing Batches are progressively merged into larger Batches, creating a series of larger and larger Batches stored along the array indices. Figure 5.2 shows the state of the Spine for arrangement J for the differences shows in Table 5.2a.

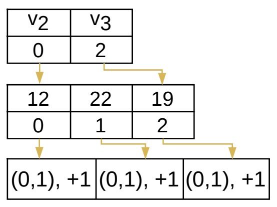
For long-running computations, the collection history can grow unboundedly across a large number of timestamps. However, as timestamps expire, the differences at those timestamps can be merged and potentially removed when they cancel out. For example, Figure 5.2d shows how Batches $B_{0,1}$ and $B_{1,1}$ can be merged into a new Batch B_{new}. Note how the differences $(12, (0,1), +1)$ and $(12, (1,1), -1)$ for v_2 cancel out and doesn't need to be stored in B_{new}. Similarly for the value 19 of v_3. It is safe to perform such cancellations after the timestamp (1,1) has expired, as DD operators can no longer seek data for timestamps $t{\leq}(1,1)$.

		G_0	G_1
0	δE	$(v_0, (v_1, 10), +1)$, $(v_0, (v_2, 5), +1)$, $(v_0, (v_3, 21), +1)$, $(v_1, (v_2, 2), +1)$, $(v_1, (v_3, 9), +1)$, $(v_3, (v_2, 1), +1)$	$(v_0, (v_1, 1), +1)$, $(v_0, (v_1, 10), -1)$
	δL	$(v_0, 0, +1)$	$\varnothing$
	δJ	$(v_1, 10, +1)$, $(v_2, 5, +1)$, $(v_3, 21, +1)$	$(v_1, 1, +1)$, $(v_1, 10, -1)$
1	δL	$(v_1, 10, +1)$, $(v_2, 5, +1)$, $(v_3, 21, +1)$	$(v_1, 1, +1)$, $(v_1, 10, -1)$
	δJ	$(v_2, 12, +1)$, $(v_2, 22, +1)$, $(v_3, 19, +1)$	$(v_2, 12, -1)$, $(v_2, 3, +1)$, $(v_3, 10, +1)$, $(v_3, 19, -1)$
2	δL	$(v_3, 19, +1)$, $(v_3, 21, -1)$	$(v_2, 3, +1)$, $(v_2, 5, -1)$, $(v_3, 10, +1)$, $(v_3, 19, -1)$
	δJ	$(v_2, 20, +1)$, $(v_2, 22, -1)$	$(v_2, 11, +1)$, $(v_2, 20, -1)$

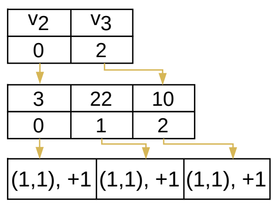
(a) SSSP differences. $\delta E = \varnothing$ and is omitted after iteration 0.



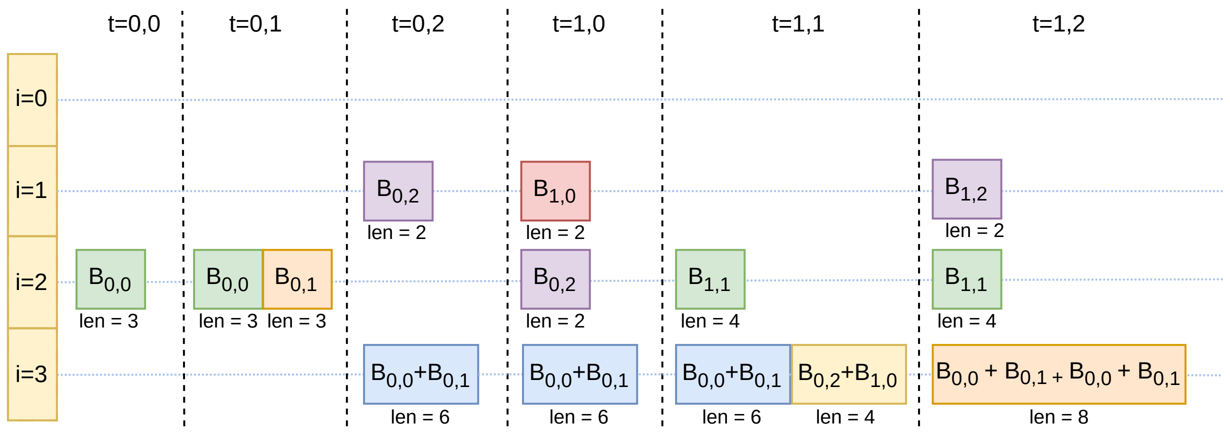
(b) δJ Batch $B_{1,1}$, len=4.



(c) δJ Batch $B_{0,1}$, len=3.



(d) Merged $B_{0,1} + B_{1,1}$, len=3.



(e) State of Spine for J at different timestamps.

Figure 5.2: Differences, Batches, and Spine for SSSP on the example graph from Figure 5.1b.

5.1.2 Operations on Arrangements

DD arrangements offer several functions to retrieve data from the index, defined for both individual Batches and the Spine. The key functions include:

- `seek_key(`k`)`: searches for Batches that contains key k using binary-search.
- `seek_value(`k, v`)`: among the Batches containing k, searches for those containing value v, using binary-search.
- `get_value(`k`)`: returns the next sorted value across all Batches containing k.
- `get_times(`k, v`)`: returns (t^*, Δ) pairs from all Batches containing (k, v).
- `consolidate(array:[(data, Δ)])`: computes the multiset union of the given array, by sorting and merging entries into (data, $\sum \Delta_i$) pairs, and dropping any (data, 0) pairs.

When accessing arrangements, an operator first searches for a key k using `seek_key(`k`)`. If found, the operator next loops through all values v for k using `get_values(`k`)`, extracts the (t^*, Δ) pairs for each value using `get_times(`k, v`)`, and stores the consolidated (`value`, t^*, Δ) tuples in a temporary buffer for further processing.

We next show an example of how `Reduce` uses arrangements to compute its output. Consider again the example in Table 5.2a, with the `Reduce` operator computing the output $R^{v_2}_{1,1}$ for key v_2 at timestamp $t = (1, 1)$. The input arrangement is `Spine`$\{B_{0,0}, B_{0,1}, B_{0,2}, B_{1,0}\}$, while the output arrangement is `Spine`$\{B_{0,1}, B_{0,2}, B_{1,0}\}$

`Reduce` does the following 8 sub-computations representing units of Equation 5.1:

(i) $\delta J^{v_2}_{1,1} = [(3, +1), (12, -1)]$, based on values for v_2 from $B_{1,1}$.

(ii) $\sum_{s<1,1} \delta J^{v_2}_s = [(5, +1), (12, +1)]$, based on values for v_2 in the input arrangement. Batch $B_{0,2}$ contains v_2 but is ignored because $t_{0,2} \nless t_{1,1}$.

(iii) $\delta J^{v_2}_{1,1} + \sum_{s<1,1} \delta J^{v_2}_s = [(3, +1), (12, -1), (5, +1), (12, +1)]$ as the union of (i) and (ii).

(iv) $\textsc{consolidate}(J^{v_2}_{1,1} + \sum_{s<1,1} \delta J^{v_2}_s) = [(3, +1), (5, +1)]$, from (iii).

(v) $min(J^{v_2}_{1,1} + \sum_{s<1,1} \delta J^{v_2}_s) = [(3, +1)]$, from (iv). For SSSP, `Reduce` computes the min.

(vi) $\sum_{s<1,1} \delta R^{v_2}_s = [(5, +1)]$, based on values of v_2 in the output arrangement. Only Batch $B_{0,1}$ contains v_2.

(vii) $\textsc{consolidate}(\sum_{s<1,1} \delta R^{v_2}_s) = [(5, +1)]$, from (vi).

(viii) Finally, $\delta R^{v_2}_{1,1} = [(3, +1)] - [(5, +1)] = [(3, +1), (5, -1)]$, from (v) and (vii).

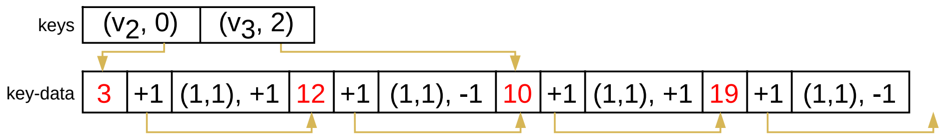


Figure 5.3: Compact Batch layout with merged (`value`, t^*, Δ).

`Reduce` repeats these steps for all keys k_t^i that are either present in δJ_t or are marked for recomputation from an earlier time $t_p < t$. δR_t is the union of the `Reduce` output for all keys processed at time t and is inserted as one or more new Batches in the output arrangement R.

5.2 Read-optimized Compact Batches

The Batch structure is designed to handle general user-defined collections of the form ((K, V), T, R), where K, V, T, and R are generic types implementing certain required interfaces (called *traits* in Rust). The Batch layout, as we showed in Figure 5.2b, handles two key operations: (i) seek keys and values within keys, using binary search on the sorted key and value arrays; and (ii) merge two batches, by performing a sorted merge on the key and value arrays, such that (t^*, Δ) pairs for each (`key`, `value`) can be consolidated from both batches, as we showed in Figure 5.2e.

For the graph computations using IFE computations that we focus on in this thesis, collections are more well-defined. In particular, all types can be represented using integers, with keys representing node IDs, values representing either neighbor nodes IDs or labels, timestamps representing graph version or IFE iterations, and Δs representing positive or negative counts.

Moreover, we observe that `Join` and `Reduce`, the two main operators in IFE, access arrangements in a well-defined pattern: (i) find Batches containing key k; (ii) retrieve and merge (`value`, t^*, Δ) entries from the matching Batches containing k. We focus on the second operation. Since `value`s and their (t^*, Δ) pairs are stored in separate arrays within a Batch, they are unlikely to be stored close together in memory, requiring seeks to different memory pages.

To optimize memory reads, we modified the Batch layout to make it more compact, by merging the five arrays into just two arrays. Consider Batch $B_{1,1}$ from Figure 5.2b. Figure 5.3 shows the equivalent compact Batch $B_{1,1}^{compact}$. We store the `key`s and key offsets in one array, and the `value`s, value offsets, and (t^*, Δ) pairs together in the second array. Since values and their corresponding (t^*, Δ) pairs are stored together, all the data for a key stored in a compact Batch can be fetched from consecutive memory locations.

Seeking random memory pages is a fast operation and we expect that Batches with or without compaction will have the same random access characteristics. However, the compact Batches have two performance benefits: (i) values and their associated (t^*, Δ) pairs are more likely to present together on the same page and thus in the same CPU cache lines; and (ii) merging Batches are more efficient since values and their associated data can copied together into the merged Batch using `memcpy`. In our evaluation in Section 5.5, we show runtime gains of up to 1.7× when DD is operating fully in memory.

We next consider datasets whose working set does not entirely fit in the available memory. DD can continue with degraded performance by letting the operating system swap excess memory pages to disk. In such a scenario, the compact Batch layout has a better advantage over the existing layout due to sequential disk accesses as compared to random access and benefits from prefetching. In our evaluation, we show a runtime gain of up to 4.2× when computations are limited to 10% of total memory required for their working set.

5.3 Time-based Indices

Batches store collection differences indexed over keys and values, which allows it to efficiently seek specific keys and values in arrangements. However, operators in IFE dataflows do not need to seek specific values. Instead, at time t, they retrieve all (`value`, t^*, Δ) tuples for a `key` for all $t^* \leq t$. Figure 5.5a outlines how operators use a temporary buffer to perform this operation. Since the same value can occur in multiple Batches, operators first copy all values and their corresponding (t^*, Δ) pairs to union them together. Entries corresponding to timestamps $t^* \nleq t$ are simply marked for future recomputation and skipped. The remaining (`value`, Δ) pairs are then collected for further processing by the operators.

Consider the case where a key is present in all N IFE iterations with timestamps $(0, j \in [0, N))$. Now suppose the graph is updated such that output for k needs to be recomputed. At t=$(1, 0)$, the entire history of k across N iterations is first copied into the temporary buffer. However, all values except those at t=$(0, 0)$ are irrelevant and will be skipped, thus utilizing only a fraction of the values copied into the temporary buffer, resulting in wasted runtime and memory.

We made a second modification to the Batch layout where the secondary index is based on the timestamps instead of values. Figure 5.4 shows an example Batch B^{time} when indexed by (key, timestamp). Figure 5.5b outlines the state of the temporary buffers with the modified layout. Operators can now directly retrieve all timestamps τ_i of a key, and only copy values when $\tau_i \leq t$, thus saving CPU time due to fewer copies and avoiding unnecessary memory allocations for irrelevant values. In our evaluation, we show up to 4.3× better runtime and up to 1.7× less memory utilization from this optimization.

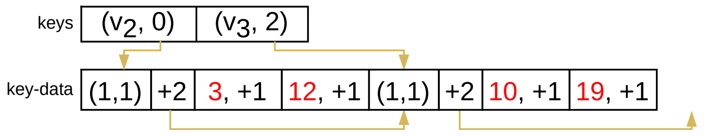
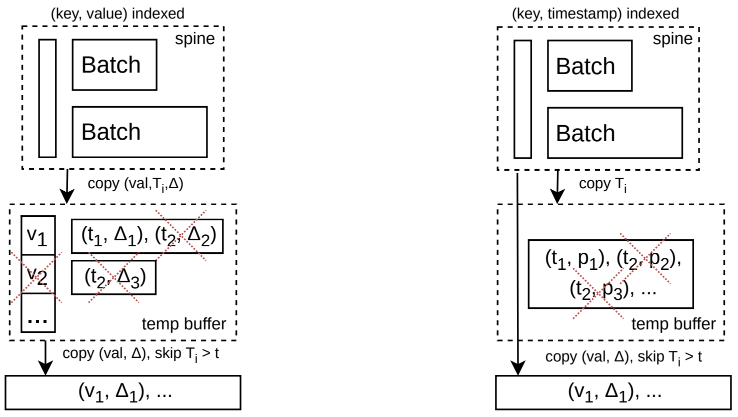


Figure 5.4: Example Batch indexed by (key, time).



(a) Using Batch indexed by (key, value). (b) Using Batch indexed by (key, time).

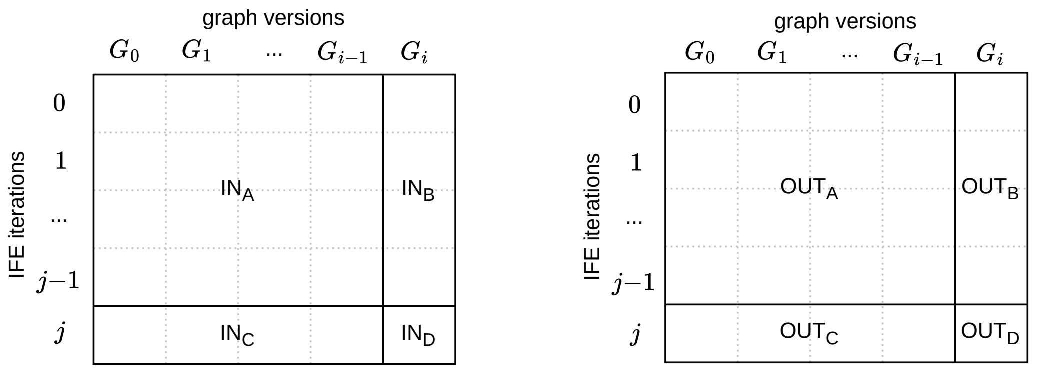
Figure 5.5: DD index operations with or without time-based indices.

5.4 FEDiV: Avoiding Difference Computations

The most expensive operator of the IFE dataflow is the Reduce operator. Vanilla DC applies the general DC rule of recomputing an operator Op. We review this rule here briefly. Consider Figure 5.6 which divides the input and output indices of an operator for a single key v into 4 regions, A, B, C, and D, respectively. Each region represents the differences in the index with timestamps covered by that region. In the figure, the x-dimension shows graph versions and the y-dimension shows IFE iterations. For simplicity, the operator is assumed to have a single input index. Recall from Chapter 3 that the recomputation logic is the following:

$$\delta\text{OUT}_{(i,j)} = \texttt{Op}\Big(\sum_{t\leq(i,j)} \delta\text{IN}_t\Big) - \sum_{t<(i,j)} \delta\text{OUT}_t \tag{5.2}$$

Let's call the computation $\texttt{Op}(\sum_{t\leq(i,j)} \delta\text{IN}_t)$ as the "new output computation". This involves scanning and consolidating the input index up to and including time (i, j) and then rerunning the Reduce operator on the consolidated input. Similarly, $\sum_{t<(i,j)} \delta\text{OUT}_t$ scans and consolidates the output index up to but excluding time (i, j).

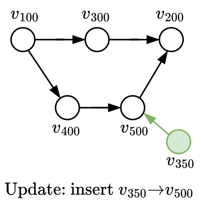


(a) Input index IN. (b) Output index OUT.

Figure 5.6: Input and output regions for a single key v during Reduce recomputation.

This rule is oblivious to the actual computation that happens inside Op and ignores any properties of the Reduce operator in the IFE dataflow. In this section, we ask: "Can we specialize the DC recomputation rule to skip certain recomputations of Reduce?" One reason to ask this question is that often the Reduce recomputations yield empty deltas, i.e., $\delta\text{OUT}_{(i,j)} = \emptyset$. Recall that workloads for which updates to inputs do not generate many output differences are the cases when DC can be effective. This is the case when the computational footprint of DC is very similar across two versions of the input, so we can "fix" these differences effectively. Yet, to verify that the deltas are empty still requires many recomputations. In this section, we seek to exploit domain-specific knowledge about the actual computation that happens inside the Reduce operators in IFE to avoid doing the full recomputation. Specifically, we will show an effective way to determine some popular scenarios when we can determine $\delta\text{OUT}_{(i,j)} = \emptyset$ without doing the "new output computation" at all and instead by only inspecting $\delta\text{IN}_{(i,j)}$, which corresponds to IN_D in Figure 5.6a, and $\sum_{t<(i,j)} \delta\text{OUT}_t$. In DD, when Reduce operator executes at (i, j), $\delta\text{IN}_{(i,j)}$ is readily available, so reading it does not require scanning the input index.

Our optimization exploits the property that Reduce operators are minimum or maximum aggregation operators in the graph computations we consider. These computations have two properties: (i) at any timestamp t, they take as input a set of inputs and produce a single output value $\delta\text{OUT}_{(i,j)} = (val, +1)$ tuple; and (ii) when running the IFE dataflow on a graph input, the Reduce outputs monotonically decrease (for minimum aggregation) or increase (for maximum aggregation). We begin by building an intuition for how we can guarantee that $\delta OUT_{(i,j)} = \emptyset$ with some simple cases and also describe where this intuition falls short. Then we describe two rules, one of which partially scans the input index and the other completely avoids scanning it (except $\delta\text{IN}_{(i,j)}$), and present rigorous proof of the correctness of these rules. We call this optimization FEDiV, for **F**ast **E**mpty **DI**fference **V**erification.



δIN for v_{200}

	G_0	G_1
0	(200, +1)	∅
1	(300, +1) (500, +1)	∅
2	(300, -1), (100, +1) (500, -1), (400, +1)	(400, -1) (350, +1)
3	(400, -1), (100, +1)	(400, +1)
4	∅	∅

δOUT for v_{200}

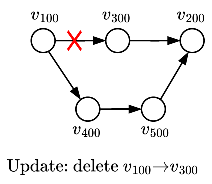
	G_0	G_1
0	(200, +1)	∅
1	∅	∅
2	(200, -1) (100, +1)	∅
3	∅	∅
4	∅	∅

Figure 5.7: WCC input and output differences for v_{200} when inserting the edge $v_{350} \longrightarrow v_{500}$.

5.4.1 Preliminary Example

The first observation we make is that since Reduce operator Op is an aggregation operator, the union of the output differences in a rectangle between the origin $(0, 0)$ and any (i, j) in Figure 5.6b is a single value m with +1 multiplicity. This is because, by Equation 5.2, the union of the output differences in the rectangle is equal to $\text{Op}(\sum_{t\leq(i,j)} \delta IN_t)$, which outputs a single aggregate value. This is "the value of key v at the j'th iteration after the i'th update to the graph". The goal of the recomputation logic is to compute the output differences OUT_D, such that OUT_D when unioned with OUT_A, OUT_B, and OUT_C, alternatively written as $\text{OUT}_{A+B+C+D}$, gives a single value $(m, +1)$. Therefore, if the union OUT_{A+B+C} is not a single value, we can be certain that OUT_D is not empty and we need to rerun the Reduce operator and follow Equation 5.2. This establishes our first rule: "If OUT_{A+B+C} is not a single value $(m, +1)$, then we cannot avoid Reduce recomputation. Moving forward, we focus on the case when OUT_{A+B+C} is a single value. We also assume that the aggregation operator in Reduce is the *minimum* function.

Intuitively one may think that the following rule is correct: "If the new batch of updates IN_D contains input differences strictly greater than m, then these new input differences cannot "retract" m from the output. Therefore m has to be the final output of $\text{Op}(\sum_{t\leq(i,j)} \delta\text{IN}_t)$." For example, consider running the WCC version of IFE in the input graph shown in Figure 5.7 and inserting a new edge from a v_{350} to v_{500}. Figure 5.7 shows the input and output differences that would be generated for key v_{200} upon this update. Consider running the Reduce computation at t=$(1, 2)$. DC would recompute Reduce at $(1, 2)$ since there are non-empty input differences at $(1, 2)$. OUT_{A+B+C}=$(100, +1)$ and IN_D=$\{(400, -1), (350, +1)\}$, which intuitively cannot retract (100, +1). So one might think that the rule articulated at the beginning of this paragraph is correct. We next show that unfortunately, this intuition is incorrect and that we must further inspect the differences in IN_C (or symmetrically in IN_B).



δIN for v_{200}

	G_0	G_1
0	(200, +1)	∅
1	(300, +1) (500, +1)	∅
2	(300, -1), (100, +1) (500, -1), (400, +1)	(100, -1) (300, +1)
3	(400, -1), (100, +1)	∅
4	∅	∅

δOUT for v_{200}

	G_0	G_1
0	(200, +1)	∅
1	∅	∅
2	(200, -1) (100, +1)	(200, +1) (100, -1)
3	∅	(200, -1) (100, +1)
4	∅	∅

Figure 5.8: WCC input and output differences for v_{200} when deleting the edge $v_{100} \longrightarrow v_{300}$.

5.4.2 Input History Scanning Rules

Now consider the example in Figure 5.8, which shows the differences generated when deleting the edge $v_{100} \longrightarrow v_{200}$. Consider running the Reduce computation at timestamp $(1, 3)$. DC would recompute Reduce at $(1, 3)$ since $(1, 3)$ is in the *join* of $(0, 3)$ and $(1, 2)$. In this case, OUT$_{A+B+C}$=$(200, +1)$ and IN$_D$=∅. However, there are still output differences, as shown in the figure. Intuitively, the reason behind this is that there were 2 paths from v_{100} to v_{200} prior to the update. One of them of length 2 and the other of length 3. Previously, the longer path was not generating any output differences, since by the time the label 100 reaches v_{200} through the longer path, the label of v_{200} was already 100, thus preventing IN$_C$, which in this example is the input differences at timestamp $(1, 3)$, from generating any differences in OUT$_C$.

To fix this, we need to inspect not only IN$_D$ but also IN$_C$ and ensure that the input differences in IN$_C$ also cannot retract m. That is, the following rule would correctly skip recomputations without doing the full computation in Equation 5.2: "If OUT$_{A+B+C}$=$(m, +1)$ and the input differences in IN_C and IN$_D$ contains differences with values greater than m, then m has to be the final output of Op$(\sum_{t \leq (i,j)} \delta IN_t)$." Symmetric to this rule, we could also have checked if IN$_B$ contains differences with values greater than m. We omit the proof of this rule as we will implement and prove another rule that avoids inspecting the input differences at all with a stricter precondition. The problem with these rules is that in addition to inspecting IN$_D$, which DD keeps in a separate buffer and makes readily available to the Reduce operator, they also require scanning these input differences and identifying the ones in region IN$_C$ or IN$_B$. This is an expensive operation for two reasons: (i) input differences are generated by the in-neighbors of vertices, and vertices in real-world graphs often have large numbers of neighbors; and (ii) differences in IN$_C$ and IN$_B$ are in the spine index and not readily available.

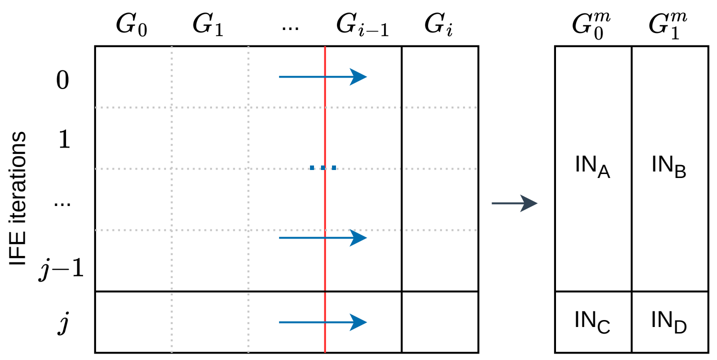


Figure 5.9: Generic description of consolidating input differences to 2 graph versions.

5.4.3 Avoiding Input History Scanning

We next describe a rule that only requires inspecting IN$_D$. Our rule is the following:

Theorem 5.4.1. *Let IN$_{A+B+C+D}$ and OUT$_{A+B+C}$=$(m, +1)$ be defined as before. Let OUT$_{A+B}$ = $(m_B, +1)$ be the union of OUT$_A$ and OUT$_B$. Let OUT$_{A+C}$=$(m_C, +1)$ be the union of OUT$_A$ and OUT$_C$. If $m = m_B = m_C$ and the differences in IN$_D$ are greater than m, then OUT$_D$ is empty.*

This is the FEDiV rule that we implemented in DD. FEDiV first computes OUT$_{A+B+C}$ and checks that it is a single value $(m, +1)$. Then it computes m_B and m_C and checks that they equal m. Finally, it checks that the differences in IN$_D$ are greater than m. We next prove Theorem 5.4.1.

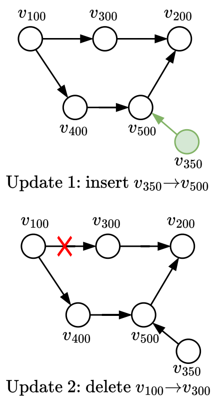
Proof. Let IN$_{A+B+C+D}$ be the union of IN$_A$, IN$_B$, IN$_C$, and IN$_D$. We prove the theorem by proving two lemmas whose proofs we will present momentarily:

Lemma 5.4.1.1. *The value m has to appear with a positive multiplicity in IN$_{A+B+C+D}$.*

Lemma 5.4.1.2. *No value smaller than m can appear with a positive value in IN$_{A+B+C+D}$.*

These two lemmas imply as a corollary that m is the smallest value in IN$_{A+B+C+D}$ that appears with a positive difference, and hence the output of `Op`$(\sum_{t\leq(i,j)} \delta \mathrm{IN}_t)$, which runs the `Reduce` operator on IN$_{A+B+C+D}$, must be $(m, +1)$. Therefore, the result of the difference computation in Equation 5.2 is empty, since OUT$_{A+B+C}$ is already $(m, +1)$. □

Proof of Lemma 5.4.1.1. Let's first prove that m has to appear with a positive multiplicity in IN$_{A+B+C+D}$. Consider merging all graph versions between 0 and i-1 (inclusive) to a single dimension. Figure 5.9 shows this operation. This is the difference-merging operation that DD



δIN for v_{200}

	G_0	G_1	G_2
0	(200, +1)	∅	∅
1	(300, +1) (500, +1)	∅	∅
2	(300, -1), (100, +1) (500, -1), (400, +1)	(400, -1) (350, +1)	(100, -1) (300, +1)
3	(400, -1), (100, +1)	(400, +1)	∅
4	∅	∅	∅

δIN for v_{200} after merging G_0 and G_1

	G_0^m	G_1^m
0	(200, +1)	∅
1	(300, +1) (500, +1)	∅
2	(300, -1), (100, +1) (500, -1), (350, +1)	(100, -1) (300, +1)
3	(100, +1)	∅
4	∅	∅

Figure 5.10: Example merging of input differences.

performs from time to time when old graph versions are no longer needed. Within each IFE iteration, this operation unions all of the differences in graph versions 0 to $i-1$ and consolidates all the differences. This does not affect the consolidated contents of IN$_A$ and IN$_C$. After merging, we have 2 graph versions—G_0^m, the first graph version *without any updates*, and G_1^m, the graph version after the "first update". Therefore, the consolidated differences in IN$_A$ and IN$_C$ are exactly the differences that are generated if we were to apply all graph updates from G_0 to G_{i-1} into a single static graph and run IFE on that. For example, Figure 5.10 shows an example of reducing graph versions from 3 to 2. The figure considers performing the two updates of inserting $v_{350} \longrightarrow v_{500}$ and deleting $v_{100} \longrightarrow v_{200}$ one after another in our running example. Readers can verify that the differences in the G_0^m column after merging are exactly equal to the differences one would get when running WCC in the bottom graph snapshot in the figure, i.e., including the $v_{350} \longrightarrow v_{500}$ edge.

We first argue that m cannot appear with a negative value in IN$_C$. Recall that the input differences correspond to the outputs of the `Join` operator. At any iteration, the `Join` operator joins any new Reduce outputs for a vertex w from the previous iteration with the outgoing edges of w, producing an output difference for each outgoing neighbor of w. Since there are no graph updates in G_0^m, the join outputs generated by vertex w monotonically decrease. Therefore, if there is any $(m, -1)$ at iteration j, i.e., in IN$_C$, it must be the case that it is followed with a lower value $(m' < m, +1)$. Since Reduce outputs the minimum value of all input differences with positive values, and since m' is one such value in IN$_C$, OUT$_{A+C}$ would have to be a value smaller than or equal to $m' < m$. But this contradicts our assertion that OUT$_{A+C}$=m_A=m. Therefore m cannot appear with a negative multiplicity in IN$_C$.

We also know that OUT$_{A+B}$=m_B=m, therefore IN$_{A+B}$ must include m with a positive difference. Finally, we verified that IN$_D$ only contains differences with values greater than m. Therefore

$IN_{A+B+C+D}$, i.e., the union of IN_{A+B}, IN_C, and IN_D, must contain m with a positive difference, completing the proof of the lemma. □

Proof of Lemma 5.4.1.1. Next, we prove that no value $m' < m$ can appear in $IN_{A+B+C+D}$ with a positive difference. We already know that IN_D contains only differences greater than m. Note that since OUT_A is a rectangle from the origin, OUT_A consolidates to a single value. Let's call the union of differences at $\text{OUT}_A = m_A$. We first show that if $m = m_B = m_C$, then it must be that m_A is also equal to m. Since $\text{OUT}_{A+C} = m_C$, we can write OUT_C as $(m_A, -1) + E_C + (m_C, +1)$ where E_C are some set of (possibly empty) differences that when consolidated results in the empty set. Similarly, we can write OUT_C as $(m_A, -1) + E_B + (m_B, +1)$ for some E_B that when consolidated results in the empty set. Since $\text{OUT}_{A+B+C} = m$, we have $m = (m_A, +1) + ((m_A, -1) + E_B + (m_B, +1)) + ((m_A, -1) + E_C + (m_C, +1))$. Simplifying this equation gives $m = (m_B, +1) + (m_A, -1) + (m_C, +1)$. Therefore, m_A must cancel out either m_B or m_C. However, since we already assumed $m = m_B = m_C$, this establishes that m_A is also equal to m. Consequently, IN_A must contain a minimum value m with a positive difference.

Let the union of differences in IN_A be S_A. IN_A represents a set of `Reduce` outputs of the incoming neighbors of v from the previous iteration, so only contains a set of values (val_1, $+\text{k}_1$), . . ., (val_z, $+\text{k}_z$), such that $\text{val}_i \geq m$ and $k_\ell > 0$. That is, S_A only contains values with positive differences. Since IN_{A+B} is also a rectangular region, its consolidation is also a set of values S_{A+B} are greater than or equal to m with positive differences. Let's represent the consolidated differences in IN_B as $S_{A^-} + \text{S}_{B^+}$, where: (i) S_{A^-} is the set of values with negative differences, which must intersect with the values in S_A (since IN_{A+B} only contains values with positive differences); and (ii) S_{B^+} are the set of values with positive differences. Note that since the minimum value in S_{A+B} is m, S_{B^+} cannot contain a value $m' < m$. Therefore, the consolidated IN_B does not contain a value $m' < m$. A symmetric argument (omitted) shows that consolidated IN_C cannot contain a value $m' < m$ with a positive difference. Therefore none of consolidated IN_A, IN_B, IN_C, or IN_D can contain a value $m' < m$ with a positive difference, completing the proof of the lemma. □

Readers might wonder if a more flexible rule that only checked that m is equal to m_B instead of checking that m is equal to both m_B and m_C could have worked. The answer is no, and the example in Figure 5.8 gives a counter-example. In this example at IFE iteration 3, m=$(200, +1)$, OUT_{A+B}=m_A=$(200, +1)$, and there are further no differences in IN_D. Yet OUT_D is non-empty. In this example, since OUT_{A+C}=m_C=$(100, +1)$, which is not equal to m_A, our optimization would not skip the DC's default recomputation logic in Equation 5.2. Finally, it is not clear, and probably unlikely, that our proof generalizes to higher-dimensional computations in DC. That is because at least our proof makes very explicit use of several properties of IFE computation, which is a 2D computation in DC. An interesting research question is whether similar rules can be designed for generic and possibly arbitrary-dimensional DC computations, possibly by assuming some properties of the operators in a dataflow.

The above proof heavily relies on the aggregation function in the IFE dataflow being a minimum, although a symmetric argument would also work if it was a maximum. However, it would not work for other common aggregation functions, such as computing the average. It is also left open whether a similar rule can be derived for dataflows that lead to timestamps with 3 or a higher number of dimensions. The above proof relied heavily on the merging operation to reduce one of the dimensions to just 2, so we could prove that m could not appear with a negative value. This argument does not seem easy to generalize to 3D, for example for the SCC dataflow from Section 3.3.4, and is left for future work.

5.5 Evaluation

We next evaluate the effectiveness of the three optimizations presented in this chapter on a suite of graph computations that use IFE. Our goals are to measure how much benefit these optimizations provide over vanilla DD in several settings. We expect that under our compact and time-based indices, DD should use less memory, due to time-based indices leading to fewer temporary memory allocations and compact indices leading to more sequential reads. We also expect that the FEDiV optimization should improve the runtime of computations as well as lead to fewer data reads, as each time the optimization skips rerunning the `Reduce` operator, the system avoids scanning data from the input index of `Reduce` to temporary buffers completely. We cover cases when all data fits in memory. as well as when it does not. We cover the case when data spills over to disk to demonstrate the performance benefits when scaling out of memory.

5.5.1 Setup

We implemented all of our optimizations on top of DD. For simplicity, we implemented each optimization on top of the previous one, as follows:

- Udd: Unmodified DD.
- Ocidx: Udd + compact indices, from Section 5.2.
- Otime: Ocidx + time-based indices, from Section 5.3.
- Ofediv: Otime + avoiding recomputations in the `Reduce` operator, from Section 5.4.

We evaluate our optimizations on 3 real-world graphs, as shown in Table 5.1, using 3 different computations: (i) weakly connected components (WCC); (ii) multiple source shortest path (MSSP), which computes the shortest weighted path from multiple sources to all other nodes in the

Dataset	|V|	|E|	Disk Size	Description
ego-Gplus (GPlus)	107K	13M	53M	Social graph from Google+
soc-LiveJournal (SocLJ)	4M	68M	301M	Social graph from LiveJournal
Twitter	42M	1.5B	5.5G	Social graph from Twitter

Table 5.1: Datasets used in the experiments.

graph; and (iii) multiple pair shortest path (MPSP), which computes the shortest path between multiple pair of nodes. For MSSP, we randomly select 3 nodes that have outgoing edges. For MPSP, we select 5 pairs of vertices (src, dst), such that there is a path between src and dst of length equal to the diameter of the graph. All computations are implemented using DD's dataflow API. Using these computations, we evaluated our optimizations on a dynamic graph workload, where we start with an initial graph and perform updates to it to incrementally maintain these computations (Section 5.5.2). We also evaluated them on the workloads we used to evaluate Graphsurge from Chapter 4 (Section 5.5.5). Sections 5.5.3 and 5.5.4 contain experiments demonstrating the effects of our optimizations when we scale the system out of memory as well as across multiple machines.

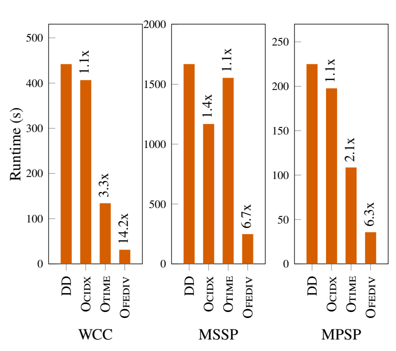
We performed our evaluation on a cluster of up to 12 machines each running Ubuntu 18.04.3. Each machine has 2× Intel E5-2670 @2.6GHz CPU with 32 logical cores and up to 512GB RAM. Except for the experiments evaluating out-of-memory performance, all experiments were performed in memory. We consider computations that require more than 500GB RAM to have failed with an Out-Of-Memory (OOM) error and that exceed 3 hours of runtime to have failed with a Time-Limit-Exceeded (TLE) error.

5.5.2 Runtime Evaluation on Dynamic Graph Workloads

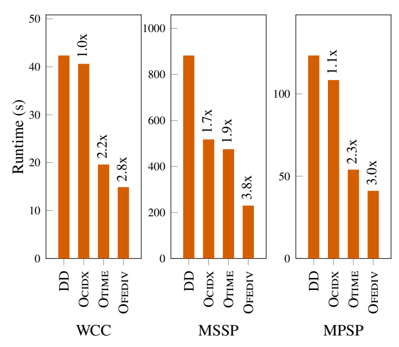
For each of the three datasets from Table 5.1, we simulate a continuous graph dataset by first randomly selecting 90% of the edges to form the first batch and then performing 5000 updates, each consisting of 25 edge additions and 25 edge deletions.

We evaluate the performance of 3 computations on each dataset, turning the optimizations on one by one. We expect that the cumulative effect of each optimization will result in better performance as compared to Udd. Figure 5.11 shows the runtime to perform 5000 updates. We exclude the time to run the computations on the initial batch.

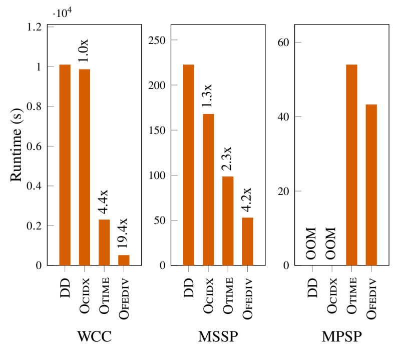
We can observe that except in two cases, MSSP on GPlus, each optimization is able to reduce the runtime consistently over the previous one. Ocidx shows the lowest runtime gains—up to 1.7× over Udd. This is expected since Ocidx adds read-optimized indices, which only have a small effect on the runtime of purely in-memory computations. In the upcoming Section 5.5.3, we



(a) GPlus

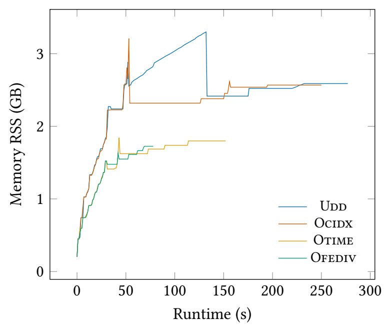


(b) SocLJ

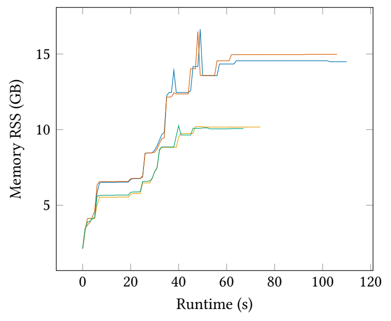


(c) Twitter

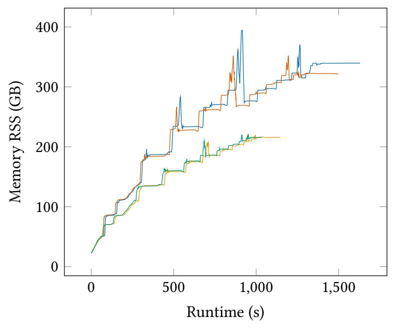
Figure 5.11: Runtimes of the optimizations for different graph computations.



(a) MPSP on GPlus.



(b) WCC on SocLJ.



(c) MSSP Twitter.

Figure 5.12: DD's total memory usage over the course of computations for each optimization.

show that the effect is magnified when the computations run in a memory-limited environment. Next, OTIME results in up to 4.4× runtime gain over UDD. Of note, MPSP on the Twitter dataset is only able to run successfully in memory after enabling OTIME. These results show that the benefit of reducing memory allocations and copies leads to visible reductions in runtime. Finally, OFEDIV results in up to 19.4× runtime gains over UDD and up to 6.3× over OTIME. As we discussed in Section 5.4, OFEDIV reduces memory allocations and avoids expensive consolidation operations, which also translates to important runtime gains, as the experiments demonstrate.

Figure 5.12 shows the actual memory consumption of DD with the optimizations turned on or off. We can observe that OCIDX does not cause any practical difference to the memory usage. However, OTIME is able to significantly reduce memory usage, by up to 1.7x, which then also benefits OFEDIV. Note that the *x*-axis is the time needed to finish the computation and DD is able to finish sooner with the optimizations turned on.

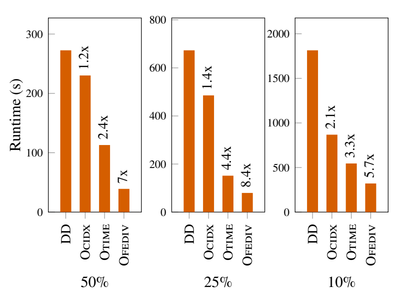
Dataset	Comp	#recomputations	#empty	#skipped	%skipped
GPlus	WCC	2712632	2705761	2358871	87.18
	MPSP	6430534	6265441	6176585	98.58
	MSSP	30317350	29870879	29823361	99.84
SocLJ	WCC	4391819	4260463	3903925	91.63
	MPSP	24464521	22805542	22296333	97.77
	MSSP	154396098	144650405	141026770	97.49
Twitter	WCC	2880159	2851472	2547692	89.35
	MSSP	21913192	21677886	21629107	99.77
	MPSP	24937244	24382549	24301488	99.67

Table 5.2: Counts of total `Reduce` recomputations, recomputations which do not produce any output, and recomputations that FEDiV was actually able to skip.

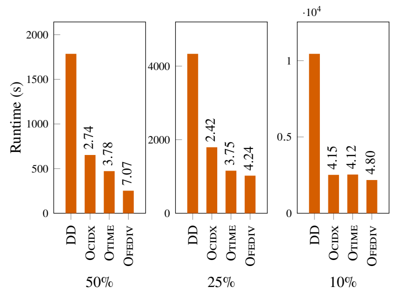
A natural next question is to ask how effective the Fediv optimization is in skipping recomputations where possible. We can answer this question by counting how many `Reduce` recomputations result in no effective change in output and how many of those calls were successfully skipped by the Ofediv optimization. Table 5.2 shows the counts for 3 computations over the 3 datasets. We can observe that Ofediv skipped 87-99% of all recomputations which resulted in an empty output. Note that although we can avoid falling back to DC's default slower recomputation logic over 90% of the time, we cannot expect commensurate gains for several reasons. First, we also do some computation to do the checks in Fediv, which partially offsets the computational gains. Second, there are other computations than `Reduce` in IFE, such as the `Join` operator as well as the work done in the arrange operators to index differences. Yet, overall we observe that Ofediv is highly effective in avoiding output difference computations when it is possible to do so.

5.5.3 Scaling Out-of-Memory

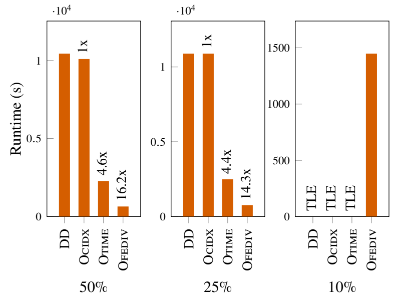
Udd normally aims to run computations purely in memory. In this section, we evaluate the runtime performance of Udd with and without optimizations, when computations do not have access to sufficient memory. To avoid OOM failures, we enable Linux `swap` on the machines and allow the OS to transparently handle swapping memory in and out of disk. Because our machines normally have sufficient RAM for most of the computations, we simulated a low memory system by running the computations in Docker [45] containers, where we manually restrict the amount of available RAM inside the container. For each computation and dataset that we ran in Section 5.5.2, we noted the maximum amount of RAM `M_max` consumed over the entire run, by recording the process Resident Set Size (RSS) as reported by the OS. Next, we created 3 containers, each restricted to 50%, 25%, and 10% of `M_max` RAM. We then reran



(a) MPSP on the GPlus.



(b) MSSP on the SocLJ.



(c) WCC on the Twitter.

Figure 5.13: Runtimes of optimizations when operating under low memory conditions.

the computations with the optimizations turned off or on in these 3 containers. Overall, we expect that our optimizations consistently yield visible improvements over vanilla DD as in the pure memory case. However, we expect the incremental benefits from FEDiV to decrease as the system is under more and more memory pressure. This is because FEDiV primarily decreases the amount of computation. Yet, as the system has less memory, more runtime will be spent on I/O, so the computation will start becoming more I/O and less computation-bound.

In this experiment, we used one computation on each dataset, specifically MPSP on GPlus, MSSP on SocLJ, and WCC on Twitter. Figure 5.13 shows the runtime results. First, from Figure 5.13 we can observe that Ocidx results in better runtimes, up to 4x, as compared to running purely in-memory in Section 5.5.2. This is mainly because, with Ocidx turned on, all data for a key is stored together. We expect this to result in better cache locality when swapping in and out of disk. Storage devices, both memory and disk, also benefit from prefetching when reading sequentially, so the sequential storage of Ocidx should also lead to better prefetching when swapping data in. Next, we observe that benefits from all the optimizations turned on first increase as memory gets more restricted, then decrease again as memory gets scarce. Initially, the benefits from lower memory utilization and avoiding recomputations dominate, as Udd needs to spend more time seeking data inside indices and recomputing results. However, as memory gets more and more restricted, the entire computation is competing for access to data in memory, and local optimizations in Reduce and indices do not retain any additional advantage. However, we can observe that overall runtime gains persist even in low memory conditions. Table 5.3 shows the amount of reads and write I/O performed by different system configurations with decreasing memory. We can verify that with each added optimization, the amount of I/O from and to disk decreases, the one exception being Otime for WCC on Twitter. For instance, DD incurs 4× more reads as compared to Ocidx and 5× more reads as compared to Ofediv when running MSSP on SOcLJ at 10% memory.

		50%	25%	10%	50%	25%	10%	50%	25%	10%
Udd	Reads (GB)	2.07	10.80	41.42	112.14	562.69	1577.46	11.90	49.59	104.09
	Writes (GB)	1.19	2.88	3.47	42.15	67.96	86.45	0.00	27.66	35.50
Ocidx	Reads (GB)	1.15	6.94	21.89	15.10	176.76	384.19	14.56	40.58	81.11
	Writes (GB)	0.62	2.15	3.12	12.58	31.95	46.71	0.00	22.90	30.85
Otime	Reads (GB)	0.10	1.61	11.09	0.88	96.20	321.14	6.82	22.21	113.77
	Writes (GB)	0.00	1.11	2.37	0.00	26.90	46.20	0.00	12.90	113.64
Ofediv	Reads (GB)	0.22	1.65	10.42	1.10	93.92	317.40	6.63	15.38	54.63
	Writes (GB)	0.00	1.05	2.32	0.00	27.18	45.91	0.00	3.53	13.22

(a) MPSP on GPLus. (b) MSSP on SocLJ. (c) WCC on Twitter.

Table 5.3: Total disk I/O incurred due to swapping.

5.5.4 Scaling Across Compute Nodes

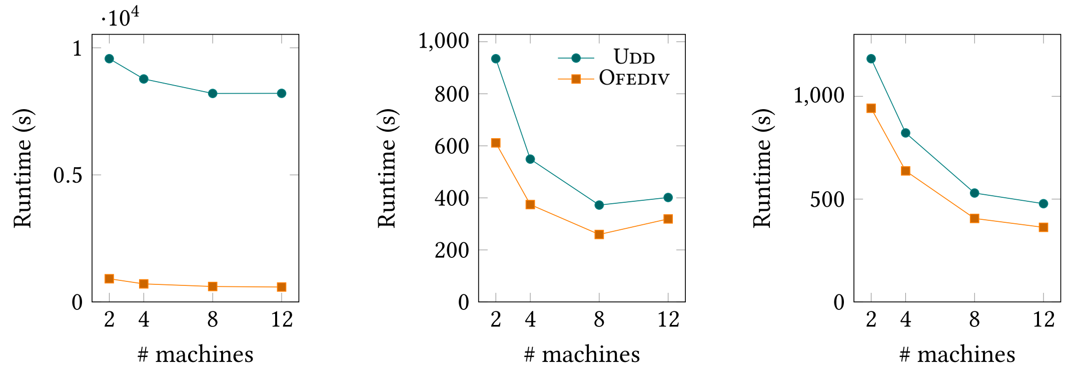
We next evaluate how our optimizations affect the ability of DD to scale in a distributed setting. We measured the runtime of 3 algorithms on the Twitter dataset, using up to 12 compute machines, each with 16 worker threads, running Udd and Ofediv only.

These experiments mainly aim to verify that the pure in-memory single-node performance gains we saw translate to a distributed setting as well. Figure 5.14 shows the scalability results. As we expected, additional machines improve the runtime for DD both with and without optimizations turned on, up to 8 machines. However, with 12 machines, the runtime for both Udd and Ofediv degrade. This is because the number of updates per batch is small and consequently the amount of work to do per batch is low. With a large number of workers in a distributed environment, more time is spent coordinating between the workers than doing useful work.

5.5.5 Graphsurge Evaluation

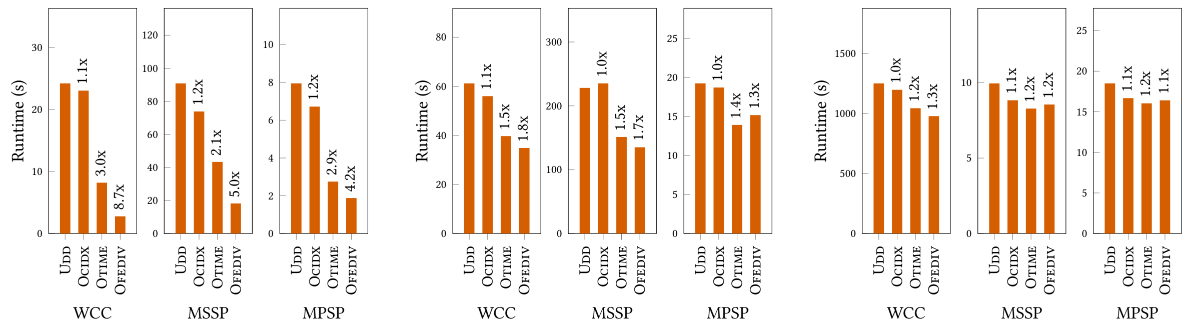
For completeness, we evaluate the effect of our optimizations on the view collections that we developed for the evaluation of Graphsurge, the DD-based view computation system that we described in Chapter 4. In particular, we prepared datasets for 2 sets of view collections: (i) C_{sim}, which is a set of 5 view collections corresponding to windows of 1 day, 1 month, 6 months, 1 year, and 2 years, respectively, and contain views similar to each other, as described in Section 4.4.2; and (ii) C_{auth}, a view collection that contains sequence of similar views, separated by completely dissimilar views, as described in Section 4.4.3. We ran 3 computations on these two sets of view collections. Figure 5.15 shows the results. We only show representative results for 2 of the 5 view collections from C_{sim}. The rest of the results are similar.

When the view collections are similar, as in C_{sim} with a window of 1 day, our optimizations



(a) WCC (b) MSSP (c) MPSP

Figure 5.14: Runtimes of optimizations in a distributed setting.



(a) 1-day C_{sim}. (b) 2-years C_{sim}. (c) C_{auth}.

Figure 5.15: Runtimes of optimizations on datasets from Graphsurge.

show a combined improvement of up to 8.7x. However, when views are dissimilar, as in C_{auth}, runtime gains are only up to 1.3x. This is expected, as when views are dissimilar, Reduce is less likely to produce empty output and we cannot avoid recomputations. Moreover, benefits from time-based indices are also minimized because when views are completely dissimilar, time spent recomputing completely new output across both Join and Reduce dominates the time saved in creating temporary buffers. The optimizations we have developed work best when recomputations due to updates result in only a small amount of new output.

5.6 Summary

DC relies on collection inputs and outputs indices to maintain computations. In this Chapter, we explored the design of these indices in DD and described how DD operators retrieve index data to run their computation. We then identified some bottlenecks in the existing index design and designed a new index that is: (i) compact; and (ii) indexed by timestamps rather than values. Both these optimizations are better suited for IFE-dataflow computations. We also identified a case where the `Reduce` operator uselessly performs a lot of work even though the outputs remain empty. To avoid this work when possible, we implement a new optimization called Fast Empty DIfference Verification, that can determine when reruns of the `Reduce` operator will be empty, without running the full operator logic. We experimentally evaluated each of our optimizations and showed significant scalability improvement in terms of both runtime and memory. We again demonstrated how application-specific optimizations can enhance the scalability of DC.

6 Related Work

Differential computation (DC) was first described as a component of the Naiad system [124, 127], with well-developed theoretical foundations [47]. This thesis studies DC in the context of graph computations, which it is especially suitable for due to its ability to incrementally maintain arbitrarily nested iterative dataflows. Several other applications in the literature make use of DC, demonstrating its generality. Examples include maintaining and verifying routing configuration in large software-defined networks [157, 168], incrementally updating machine-learning models to forget parts of the data they were trained on [146], deriving output lineage in iterative dataflow computations [75], and efficiently querying and transforming model frameworks used to design product lines, such as in aerospace and automotive industries [169].

DC is one of many approaches for computation sharing in the literature, which we can broadly classify into two categories: general incremental computation techniques and solutions specialized for specific computations. DC is an example of the first category, but we cover the related literature on both topics here. In particular, we cover the literature on incremental view maintenance from database literature, focusing on general techniques that can maintain recursive computations similar to DC. Sharing computations implies the need to store computation state and we cover the various techniques that have been developed to optimize state maintenance.

We reviewed the related work for our non-technical survey work in Chapter 2 and thus omit it here. The technical work presented in this thesis is on the Differential Dataflow (DD) system, which is built on top of the Timely Dataflow (TD) system. TD is a distributed dataflow system. There is extensive work in the literature on distributed dataflow systems such as MapReduce [80], Spark [167], and Flink [69]. Some of these systems have been extended to run iterative computations effectively. Examples include HaLoop [66], GraphX [90], CIEL [128], and Gelly Streaming [21]. However, unlike DC, these systems do not support sharing previous output when the input data changes. For this reason and that this thesis focuses primarily on the computation-sharing abilities of DC, we do not cover distributed dataflow systems in detail.

6.1 General Techniques for Computation Sharing

Overview of IVM: View materialization [94] is a classic technique in data systems used to precompute and cache the results of a query Q_V as a logical *view* V over some base data D, i.e, $V{=}Q_V(D)$. Views facilitate reusing previous results to answer repeated queries instead of rerunning them from scratch. When the base data changes (say from D to D'), V should also be updated to $V'{=}Q_V(D')$ to avoid outdated responses. View maintenance is the problem of keeping the views up to date with respect to the base data. Instead of rerunning the view definition query from scratch after every update, incremental view maintenance (IVM) is a technique that incrementally updates a view using the set of delta changes to the base data, i.e, $V'{=}V{+}\delta V$, where $\delta V{=}Q_V(\delta D)$ and $\delta D{=}D'{-}D$. Because δD is expected to be small relative to D, IVM can significantly reduce the time required to update the view results. There is extensive literature on incremental view maintenance of the outputs of relational or datalog queries [70,82,93,126,143]. Shirkova et al. [150] is a recent general survey on view materialization and maintenance techniques. Prior work in this literature, excluding those on IVM of Datalog programs and DBSP, which we cover momentarily, generally discusses computations that can be expressed in relational algebra, which does not contain recursion. A popular technique in this space is to algebraically decompose the joins in the queries into multiple smaller "delta" queries [61]. Each delta query is easy to evaluate because at least one of the relations in the query is a delta relation δR, i.e., contains only the updated tuples in R. The state-of-the-art approach here is F-IVM [130], for factorized IVM, which also compresses the outputs of the delta-queries as *tries*, using the compression technique of factorization, and maintains the delta queries on these tries [131].

Many IVM techniques for relational algebra, including many delta decomposition-based techniques, are stateless techniques, though some techniques do store state. DBToaster [50] is an example technique that keeps state in the form of results of *higher-order delta queries*. Briefly, higher-order queries generalize delta queries to deltas of delta queries, similar to taking higher-order derivatives of functions. We omit the details as these are related to non-recursive queries and do not directly relate to graph computations, which is the focus of this thesis.

IVM Algorithm for Datalog: Traditional IVM techniques for Datalog programs can be broadly classified into two approaches. The first consists of *counting* approaches that track the lineage of facts with counts, such that a zero count implies a fact can be deleted. However, counting algorithms only work for non-recursive Datalog [126].

The second group consists of the Delete/Rederive (D/Red) algorithm [94] and its derivatives, such as the Forward/Backward/Forward algorithm [126]. Unlike counting, these algorithms can maintain recursive Datalog programs. However, these techniques can be highly inefficient as input deletions require propagating the delete to all derived facts and rederiving the ones that are still valid, which can require a large amount of work, even if the final output changes very little. In contrast to DC, D/Red can be seen as a stateless but computationally less efficient maintenance algorithm. There is no open source Datalog system that we are aware of that implements D/Red. The only project we are aware of that has IVM capabilities over Datalog is the Differential Datalog project [139]. Interestingly, Differential Datalog is built on top of DD and uses DC instead of Datalog-specific algorithms such as D/Red.

DBSP: DBSP [67] is perhaps the most directly related work that has recently been published. DBSP is a mathematical theory to describe general incremental computation in databases using formalism from discrete digital signal processing. As stated by the authors, it was inspired by DC and started as an effort to provide a new abstraction for DC that is easier to understand. It models relations as timestamped streams, similar to collections in DC, and generic operators that map streams to other streams and loops. These ingredients can model the dataflows in DC. Then with higher level operators such as incrementalizing, taking derivatives, and integrating operators, they provide the same capability of being able to maintain arbitrary dataflow of computations. The authors show how one can convert recursive Datalog queries into equivalent DBSP programs and maintain them.

At the moment, the computational efficiency of the general DBSP maintenance algorithm in terms of the state it keeps is not understood. The authors primarily provide proof of DBSP's correctness. This is similar to how the original publications on DC only focused on correctness and not on the resource usage of DC. There is also no reference implementation of DBSP so far, although a closed-source version is being developed at a commercial company by the authors. It is an important direction for future work to both provide an implementation of DBSP and compare its resource consumption to DC. However, it is a recent alternative to DC that can be the foundation for maintaining arbitrary dataflow computations including iterative graph computations that we focus on in this thesis.

6.1.1 Other Computation Sharing Techniques

DC stores the computational footprint of a dataflow computation on a given input and detects and shares this stored state when the computation is executed on an updated version of the input. This is similar to work that shares computations across multiple queries that run over the same input.

Examples include work on continuous querying systems, such as NiagaraCQ [73], or systems that support running multiple queries, e.g., when incrementally maintaining multiple views [109] or running multiple queries in a batch [108, 138]. These techniques share computation by detecting common sub-expressions across queries over the same input, while DC incrementally maintains the same computation for evolving inputs.

6.2 Specialized Computation Sharing Techniques

There is a vast literature [60, 83, 88, 98, 125, 133] studying specialized implementations of graph algorithms on dynamic graphs, which can incrementally update their output when the input graph changes. Covering this entire literature is beyond the scope of this thesis. To compare the research in this thesis broadly with these specialized techniques, we cover 2 example publications.

Fan et al. [87] presents theoretical results that show that the cost of performing six specific incremental graph computations, e.g., regular path queries and strongly connected components, cannot be bounded by only the size of the changes ΔG in input and output. Then they develop and evaluate algorithms that have guarantees in a more relaxed notion of boundedness, based on the subset of the graph inspected by the batch algorithm being incrementalized.

GraphBolt [122] is a shared-memory parallel streaming system that can maintain dynamic versions of graph algorithms. GraphBolt requires users to write explicit maintenance code in functions such as `retract` or `propagateDelta` that generic systems such as DC do not require. As graph updates arrive, the system executes these functions, and if a user-provided dynamic algorithm has provable convergence guarantees, the system will correctly maintain the results.

These incremental algorithms specialized to specific algorithms can be more efficient than a DC implementation using generic operators. For instance, the authors of GraphBolt demonstrated, and we verified in Chapter 4, that their custom implementation of PageRank is faster than the corresponding implementation in DC. Nevertheless, generic solutions such as DC have the advantage that users can program arbitrary computations as a static dataflow, which DC will automatically maintain for changing inputs.

6.2.1 Streaming Graph Computation Systems

Many systems in the literature are built to support computation on streaming graphs, i.e., inputs represent a sequence of edge additions or deletions, as extensively surveyed in Besta et al. [59]. These systems, such as the S-Graffito streaming graph system [134], support windowed queries over streams. The primary difference between streaming graphs and dynamic graphs, which this thesis is concerned with, is that streaming systems assume that the data is too large and cannot be stored in its entirety, so computations need to be done on sliding windows of tuples as time moves on. Instead, systems with incremental computation capabilities primarily assume that the data is changing but can be kept in its entirety. In a sense, systems for dynamic graphs evaluate growing windows from time 0.

Tegra [104] is a system developed on top of Apache Spark [167], that is designed to perform ad-hoc window-based computations on a dynamic graph. Specifically, Tegra allows users to tag arbitrary snapshots of their graphs with timestamps. The system has a technique for sharing arbitrary computation across snapshots through a differential computation-like computation maintenance logic. However, the system is optimized for retrieving arbitrary snapshots quickly instead of sharing computation across snapshots efficiently.

Kickstarter [163] is a technique built to improve stream processing systems that maintain approximate results of iterative computations but do not support edge deletions. Kickstarter enables support for edge deletions in such systems by carefully tracking dependencies between edges and computation outputs and *trimming* approximate values affected by edge deletions. This technique is suitable only for monotonic graph algorithms, such as shortest path queries. **DZiG** [121] is a related streaming graph processing system that is designed to exploit the *computation sparsity* of iterative computations, where vertices often converge to a value that remains stable across evolving inputs. DZiG incorporated sparsity tracking into its incremental processing engine, allowing computations to track and incrementally refine computed values while avoiding unnecessary propagations when the values stabilize.

Chronos [97] optimizes sharing computation across multiple temporal graph snapshots by designing a specialized graph layout that enhances the data locality of the snapshots when processed together. The Chronos execution engine is designed to exploit the special layout, for instance by batching computations for a vertex across multiple snapshots and reusing computed values across snapshots. Vora et al. [162] develops two general optimizations for temporal graph processing. First, they reduce the cost of retrieving values of a vertex for multiple snapshots by reordering computations such that overlapping vertices are processed together. Second, they reduce the amount of recomputation done for a snapshot by sharing computed values from earlier snapshots.

6.3 State Management

Differential Dataflow (DD) operators can require indexing the state of previous inputs and outputs. DD stores these indices in memory, enabling fast reads and writes. However, computations can often result in a prohibitively large amount of state, for instance when operating on big datasets, even exceeding the amount of available memory on a machine. This is the general problem of managing the state of computations in processing systems [159]. DD can handle such a scenario in two ways. First, it can scale out [144] the computation across multiple machines, using its Timely Dataflow runtime. Second, DD indices can be shared between operators where possible [123], lowering the memory required for the computation. However, these techniques may not be sufficient, for instance when a distributed environment is unavailable or cost-prohibitive. In the next two sections, we review techniques that can generally be used by systems to operate gracefully in a limited memory environment.

6.3.1 Spilling to Disk

Main-memory systems can combine the fast performance of in-memory data with the scalability of disk-based systems. Traditional disk-based database systems use a buffer pool manager [91] (BPM) to transparently switch data between disk and main memory. However, this classic implementation of BPM assumes that the data is primarily stored on disk, which adds a significant overhead [99] if used directly in main-memory systems [86]. Instead, recent systems invert the BPM model, assuming that the data is primarily stored in memory, and moving a subset of *cold*, i.e., least recently used, to disk when necessary. We next review multiple approaches in the literature that support the inverted BPM model.

Anti-Caching [81] tracks accesses to the tuples of a table by storing additional metadata per tuple and moves the least recently used tuples to disk. Indexes point to tuples existing both in memory and on disk. The system aborts transactions that need to access data residing on disk and restarts them after loading the required data into memory. Because the entire index in Anti-Caching is kept in memory, the system can face significant memory overhead as the data size and number of indexes increase. **Siberia** [85] only tracks in-memory data instead. For the remaining data on disk, Siberia uses a Bloom filter to test for the existence of potential records before accessing the data on disk. However, the need to track the data movement between memory and disk, and maintain the indexes and the Bloom filters, leads to implementation complexity. **LeanStore** [117] developed a simpler technique by removing indirect *logical page identifiers* used in traditional BPMs and using direct virtual memory pointers to address data pages. To distinguish between pages residing in memory and on disk, LeanStore uses *pointer swizzling* [114] that stores the page status in one bit of the address pointer. Unlike prior systems, LeanStore does not directly write cold pages to disk, but instead moves them to an intermediate *cooling* state, which gives the system a chance to prevent pages that need to be immediately accessed again from being dropped. These techniques allow LeanStore to get close to optimal performance when the data resides completely in memory, which gracefully degrades as the size of the data grows beyond the available memory. **Umbra** [129] extended the LeanStore implementation to work with variable-sized pages, which allows better handling of large data tuples that might not fit in pages of a fixed size.

6.3.2 Recreating State on Demand

Instead of keeping the entire state of an incremental computation, which enables fast computations, systems can drop a subset of the state and recompute them when next required by the computation. We review three such techniques.

The work by Ammar et al. [52], which the author of this thesis also co-authored, demonstrated how a single node implementation of DC can operate with dropped state. Specifically, for IFE dataflows, output differences from the `Join` operator can be partially dropped to recover memory and recreated on demand. However, the system still needs to maintain some additional state for each dropped difference, so that it can correctly determine if a recomputation is needed, but limits how much memory is recovered by dropping differences. This limitation can be overcome by tracking state in a probabilistic data structure, such as a Bloom filter [62], which uses much less memory at the expense of more recomputations. DC can thus continue operating with a bounded memory by trading off increased computation time. As part of the work we did for this thesis, we implemented the easiest of these state recreation techniques in DD called *join on demand* (JOD). JOD does not save the output of the `Join` operator in the IFE dataflow. Instead, when a key u in the `Reduce` operator needs its input, which is the "dropped" `Join` output, it would create it on demand by reading the outputs of `Reduce` of its neighbors. In the shared-memory implementation of DC implemented by Ammar et al. [52], since all indices are in memory and accessible, these lookups are cheap. However, in the distributed dataflow implementation, which we implemented in DD, all of these correspond to network communications and round trips between TD workers, which showed us that at least JOD can be prohibitively expensive when implemented on DD. Specifically, we saw two orders of magnitude slow down in the performance of vanilla DD (though we used less memory).

Noria [89] uses *partially-stateful dataflows* to incrementally maintain the results of relational views and answer read-heavy workload queries. SQL views in Noria are internally computed using a parallel and distributed directed acyclic dataflow graph, consisting of relational operators such as `Join`, `Reduce`, and `Filter`. The operators store only a partial recently-accessed subset of their results and mark the rest as *evicted*. Noria ensures that these partial results are kept synchronized with updates to the base table. When an evicted key is needed, the operators recompute it by querying its upstream operators, recursively going back to the base tables in the worst case. Noria ensures that materialized views are eventually consistent and correct in the presence of concurrent updates and recomputations, thus allowing it to operate with a bounded amount of operator state.

CrocodileDB [149] supports view materialization using Intermittent Query Processing [158]. When users allow for a delay in materializing results, CrocodileDB intelligently defers output materialization until strictly required. When deferring executions, the system can deactivate query execution and release resources. However, instead of dropping the entire state, CrocodileDB uses a predictive query planner to dynamically select a subset of the intermediate state that fits within available memory and which is most likely to be reused, thus reducing the amount of useful state that needs to be recomputed.

7 Conclusion and Future Work

Graphs are the most intuitive abstraction to represent data characterized by entities and their relationships. Many practical applications that process graphs can benefit from sharing computations across multiple snapshots of evolving graphs, for instance when evaluating changing road conditions in transportation networks or conducting contingency analysis on infrastructure networks. The research in this thesis was motivated by the challenge to efficiently support such applications on large-scale datasets.

DC is a general technique for incrementally maintaining computations across evolving datasets, even those containing arbitrarily nested loops. It is thus well suited for supporting the kind of applications that motivated this thesis. We presented a study of DC that showed how it can used to build practical data systems. In particular, this thesis identified and addressed two key challenges that impede the adoption of DC. The first is the lack of high-level interfaces that can be used to develop graph-specific applications. The second is scalability challenges that arise due to the general maintenance technique used by DC, making it less efficient for graph-specific workloads.

7.1 Contributions

This thesis has studied the application of DC to large-scale graph computations. We built the Graphsurge system that shows a novel application of DC beyond dynamic datasets. We also developed several optimizations that leverage application-specific characteristics. Our work was in part led by insights we gleaned from a user survey.

This thesis first presented results from a non-technical user survey that we conducted to understand how graphs are used in practice in both academia and industry. To the best of our knowledge, this survey was the first study of its kind to understand the data, computations, applications, software, and main challenges users face when working with graphs. We analyzed the findings from 89 surveyed users, 8 in-person interviews, and a study of the white papers of several graph technologies. The survey identified the need to address scalability problems in existing graph software, the importance of graph computations over dynamic graphs, and the description of an actual contingency analysis application on an electric grid network. These insights informed our work in the rest of the thesis.

The next contribution of this thesis was the Graphsurge system that we built to support a view-based analysis of static graphs. It allows users to create different views of a static graph using a high-level programming interface. Users can run arbitrary graph computations on collections of such views, which internally uses DC to automatically share computation across similar views. We identified two optimization problems within Graphsurge. The first is the collection ordering problem of determining the best order of views in a view collection such that DC can maximize computation sharing. We showed that this problem is NP-hard and provided an efficient 3x-approximation algorithm. The second was the collection splitting problem of detecting when views in an ordered collection do not share enough computation with the previous view, such that re-executing a computation from scratch is cheaper than trying to share computation. We developed an adaptive algorithm that can decide when to *split* a collection dynamically at runtime, to prevent sharing computation across dissimilar views. We presented extensive experiments demonstrating that our algorithms improve the runtime of view-based computations in Graphsurge by up to an order of magnitude compared to the baselines.

The final research contribution of this thesis was to improve the scalability of DC and its reference implementation DD. We developed two key optimizations that are suitable for a common dataflow subroutine called iterative frontier expansion (IFE). The first optimization was based on the observation that it is more optimal to index collection differences in operators by timestamps rather than values. The second optimization was based on the observation that rerunning DC operators repeatedly performs a lot of work only to verify that the operator output is empty. We presented a technique called *Fast Empty Difference Verification (FEDiV)* that can detect empty operator output without actually running the full operator logic. Experimental evaluation demonstrated that our optimizations are effective in improving the scalability of DD.

The research presented in this thesis demonstrated the suitability of DC for building graph computation systems. We showed how DC can be made more practical for users by presenting high-level interfaces to express graph computations. We also showed how DC can be used in novel workloads beyond dynamic datasets. Through experimental evaluation, we showed how application-specific optimizations can effectively improve DC's performance.

7.2 Future Work

We now discuss some possible directions for future work that can extend the work done in this thesis and study the many questions that it raises.

7.2.1 FEDiV for higher dimensional timestamps

In Section 5.4, we presented a proof for FEDiV, which can safely avoid differences recomputation in the Reduce operator when we can guarantee that the output will be empty. This technique only works for an IFE sub-dataflow in singly nested computations, i.e., those with 2-dimensional timestamps, such as WCC and SSSP.

Although the proof we provided in Section 5.4 does not seem to be directly applicable to higher dimensions, a different proof could extend the idea of using common properties such as min and max aggregations to computations with 3 or even higher dimensional timestamps, such as the SCC computation shown in Figure 3.5a. SCC especially should be the first step of this generalization because it uses IFE and our proof may in fact generalize to SCC with a similar approach. More broadly, our work leaves open the following important question: what other alternative rules to DC's maintenance logic rules from Equation 5.2 can be used to verify that the outputs of DC operators are empty? Positive answers to this more general question could yield improvements for any applications running on DD.

7.2.2 Explicit State Management

Differential Dataflow (DD) at present relies on operating system (OS) swap to deal with low memory environments when its working set cannot fit in the available memory. DD could instead adapt well-known techniques for buffer pool management to manually handle the movement of data between memory and disk, possibly resulting in a better runtime performance compared to simply swapping. For instance, DD can use its internal knowledge of computations to predict the set of hot and cold memory pages as the computation progresses. In contrast, the OS can only employ black box cache eviction policies such as tracking Least Recently Used pages. We did not pursue this research direction in the context of this thesis.

Furthermore, although our initial implementation of some of the state-dropping and reconstruction optimizations proposed by Ammar et al. [52] for DC yielded poor memory performance tradeoffs, we did not fully cover all of the optimizations proposed by them. Dropping some of the differences selectively can still provide more reasonable tradeoffs. Our intuition is that these optimizations are still likely to yield poorer tradeoffs compared to techniques that don't drop differences but simply spill them to disk. This hypothesis should however be tested.

7.2.3 Theories for IVM Comparing DC to Other IVM Techniques

Perhaps the most foundational question that is not answered in this thesis or any prior publication on DC is: how does DC fare in terms of its resource usage compared to other IVM techniques? For example, no theory tries to capture DC, delta query compositions, and DRed algorithms together to be able to compare whether or not they are optimal. We know for example that delta query compositions and DRed do not store any state and perform expensive computations to maintain queries. In contrast, DC maintains a large amount of state, in fact, the entire set of input and output differences. There are also other IVM techniques, such as DBToaster that also stores state and DBSP which is similar to DC. An understanding of how to compare these approaches would be of immense interest to the research community. Even a theory that can help analyze the complexity of the work done by DC to maintain queries would be a good step in better understanding the foundations of DC.

References

[1] The 2016 State of the Graph Report. https://neo4j.com/resources/2016-state-of-the-graph.

[2] AboutYou Data-Driven Personalization with ArangoDB. https://www.arangodb.com/why-arangodb/case-studies/aboutyou-data-driven-personalization-with-arangodb.

[3] AllegroGraph. https://franz.com/agraph/allegrograph.

[4] AnzoGraph. https://www.cambridgesemantics.com/product/anzograph.

[5] Apache Flink. https://flink.apache.org.

[6] Apache Flink User Survey 2016. https://github.com/dataArtisans/flink-user-survey-2016.

[7] Apache Giraph. https://giraph.apache.org.

[8] Apache Jena. https://jena.apache.org.

[9] Apache Spark GraphX. https://spark.apache.org/graphx.

[10] Apache Spark. Preparing for the Wave of Reactive Big Data. https://info.lightbend.com/white-paper-spark-survey-trends-adoption-report-register.html.

[11] Apache TinkerPop. https://tinkerpop.apache.org.

[12] ArrangoDB. https://www.arangodb.com.

[13] Basic Linear Algebra Subprograms. http://www.netlib.org/blas.

[14] Caley. https://cayley.io.

[15] Conceptual Graphs. http://conceptualgraphs.org.

[16] Cytoscape. http://www.cytoscape.org.

[17] Detect Fraud in Real Time with Graph Databases. https://neo4j.com/whitepapers/fraud-detection-graph-databases.

[18] DGraph. https://dgraph.io.

[19] Elasticsearch X-Pack Graph. https://www.elastic.co/products/x-pack/graph.

[20] FullContact. https://www.fullcontact.com.

[21] Gelly Streaming. https://github.com/vasia/gelly-streaming.

[22] Graph for Scala. http://www.scala-graph.org.

[23] Graph-tool. https://graph-tool.skewed.de.

[24] GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. https://github.com/pdclab/graphbolt.

[25] GraphDB by Ontotext. https://www.ontotext.com/products/graphdb.

[26] GraphStream. http://graphstream-project.org.

[27] JanusGraph. http://janusgraph.org.

[28] MATLAB. https://www.mathworks.com.

[29] A Modular Implementation of Timely Dataflow in Rust. https://github.com/TimelyDataflow/timely-dataflow.

[30] Neo4j. https://neo4j.com/download-center/#community.

[31] NetworKit. https://networkit.iti.kit.edu.

[32] NetworkX. https://networkx.github.io.

[33] OpenBEL. http://openbel.org.

[34] openCypher. https://www.opencypher.org.

[35] OpenLink Virtuoso. https://virtuoso.openlinksw.com.

[36] OrientDB. https://orientdb.com.

[37] SNAP: Network datasets: Com-livejournal social network. https://snap.stanford.edu/data/com-LiveJournal.html.

[38] SNAP: Network datasets: Stackoverflow temporal network. https://snap.stanford.edu/data/sx-stackoverflow.html.

[39] SNAP: Network datasets: Wiki-topcats network. https://snap.stanford.edu/data/wiki-topcats.html.

[40] SNAP: Standford Network Analysis Project. https://snap.stanford.edu.

[41] Sparksee. http://www.sparsity-technologies.com.

[42] Stardog. https://www.stardog.com.

[43] State Grid. http://www.sgcc.com.cn/ywlm/index.shtml.

[44] TigerGraph. https://www.tigergraph.com.

[45] Docker, 2024. https://www.docker.com.

[46] An Implementation of Differential Dataflow Using Timely Dataflow on Rust, 2024. https://github.com/TimelyDataflow/differential-dataflow.

[47] M. Abadi, F. McSherry, and G. D. Plotkin. Foundations of Differential Dataflow. In *Foundations of Software Science and Computation Structures*, 2015. https://doi.org/10.1007/978-3-662-46678-0_5.

[48] F. Afrati, C. Li, and P. Mitra. Rewriting Queries Using Views in the Presence of Arithmetic Comparisons. *Theoretical Computer Science*, 368(1-2), 2006, https://doi.org/10.1016/J.TCS.2006.08.020.

[49] C. C. Aggarwal and H. Wang. Graph Data Management and Mining: A Survey of Algorithms and Applications. In *Managing and Mining Graph Data*. 2010. https://doi.org/10.1007/978-1-4419-6045-0_2.

[50] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proceedings of the VLDB Endowment*, 2(2), 2009, https://doi.org/10.14778/1687553.1687592.

[51] S. Amer-Yahia and J. Pei, editors. *Proceedings of the VLDB Endowment, Volume 11*. 2017/2018. http://www.vldb.org/pvldb/volumes/11.

[52] K. Ammar, S. Sahu, S. Salihoglu, and M. T. Özsu. Optimizing differentially-maintained recursive queries on dynamic graphs. *Proceedings of the VLDB Endowment*, 15(11), 2022, https://doi.org/10.14778/3551793.3551862.

[53] W. Ammar, D. Groeneveld, C. Bhagavatula, I. Beltagy, M. Crawford, D. Downey, J. Dunkelberger, A. Elgohary, S. Feldman, V. Ha, R. M. Kinney, S. Kohlmeier, K. Lo, T. C. Murray, H.-H. Ooi, M. E. Peters, J. L. Power, S. Skjonsberg, L. L. Wang, C. Wilhelm, Z. Yuan, M. van Zuylen, and O. Etzioni. Construction of the literature graph in semantic scholar. In *Human Language Technology: Conference of the North American Chapter of the Association of Computational Linguistics*, 2018. https://doi.org/10.18653/v1/n18-3011.

[54] R. Angles, M. Arenas, P. Barcelúndefined, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5), 2017, https://doi.org/10.1145/3104031.

[55] A. C. Arpaci-Dusseau and G. Voelker, editors. *Proceedings of the Symposium on Operating Systems Design and Implementation*. 2018. https://www.usenix.org/conference/osdi18.

[56] M.-F. Balcan and K. Q. Weinberger, editors. *Proceedings of the International Conference on Machine Learning*. 2016. http://jmlr.org/proceedings/papers/v48.

[57] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr. Large Scale Graph Processing Systems: Survey and an Experimental Evaluation. *Cluster Computing*, 18(3), 2015, https://doi.org/10.1007/s10586-015-0472-6.

[58] Bernd Mohr and Padma Raghavan, editors. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017. https://doi.org/10.1145/3126908.

[59] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems*, 34(6), 2023, https://doi.org/10.1109/TPDS.2021.3131677.

[60] S. Bhattacharya, M. Henzinger, and D. Nanongkai. A New Deterministic Algorithm for Dynamic Set Cover. In *Foundations of Computer Science*, 2019. https://doi.org/10.1109/FOCS.2019.00033.

[61] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *ACM International Conference on Management of Data*, 1986. https://doi.org/10.1145/16894.16861.

[62] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), 1970, https://doi.org/10.1145/362686.362692.

[63] P. Boncz and K. Salem, editors. *Proceedings of the VLDB Endowment, Volume 10*. 2016/2017. http://www.vldb.org/pvldb/volumes/10/.

[64] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. Querying graphs. In *Synthesis Lectures on Data Management*. 2018. https://doi.org/10.2200/S00873ED1V01Y201808DTM051.

[65] S. Bridgeman and R. Tamassia. A User Study in Similarity Measures for Graph Drawing. In *Graph Drawing Symposium*. 2001. https://doi.org/10.1007/3-540-44541-2_3.

[66] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3(1-2), 2010, https://doi.org/10.14778/1920841.1920881.

[67] M. Budiu, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proceedings of the VLDB Endowment*, 16(7), 2022, https://doi.org/10.14778/3587136.3587137.

[68] E. Bullmore and O. Sporns. Complex Brain Networks: Graph Theoretical Analysis of Structural and Functional Systems. *Nature Reviews Neuroscience*, 10(3), 2009, https://doi.org/10.1038/nrn2575.

[69] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015. http://sites.computer.org/debull/A15dec/p28.pdf.

[70] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Very Large Data Bases Conference*, 1991. http://www.vldb.org/conf/1991/P577.PDF.

[71] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Symposium on the Theory of Computing*, 1977. https://doi.org/10.1145/800105.803397.

[72] C. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. In *International Conference on Database Theory*, 1997. https://doi.org/10.1007/3-540-62222-5_36.

[73] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *ACM International Conference on Management of Data*, 2000. https://doi.org/10.1145/342009.335432.

[74] R. Chirkova. Query containment. In *Encyclopedia of Database Systems*. 2018. https://doi.org/10.1007/978-1-4614-8265-9_1269.

[75] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining Outputs in Modern Data Analytics. *Proceedings of the VLDB Endowment*, 9(12), 2016, https://doi.org/10.14778/2994509.2994530.

[76] N. Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. *Operations Research Forum*, 3(1), 2022, https://doi.org/10.1007/S43069-021-00101-Z.

[77] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 26(1), 1983, https://doi.org/10.1145/357980.358007.

[78] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd edition edition, 2009. http://mitpress.mit.edu/books/introduction-algorithms.

[79] W. Cui. *A Survey on Graph Visualization*. PhD thesis, Hong Kong University of Science and Technology, 2007. https://cse.hkust.edu.hk/~huamin/MSBD5005/graph.pdf.

[80] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, 2004. http://www.usenix.org/events/osdi04/tech/dean.html.

[81] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-Caching: A New Approach to Database Management System Architecture. *Proceedings of the VLDB Endowment*, 6(14), 2013, https://doi.org/10.14778/2556549.2556575.

[82] A. Deligiannakis. View maintenance aspects. In *Encyclopedia of Database Systems*. 2009. https://doi.org/10.1007/978-0-387-39940-9_838.

[83] R. Duan, H. He, and T. Zhang. Dynamic Edge Coloring with Improved Approximation. In *ACM-SIAM Symposium on Discrete Algorithms*, 2019. https://doi.org/10.1137/1.9781611975482.117.

[84] J. G. Dy and A. Krause, editors. *Proceedings of the International Conference on Machine Learning*. 2018. http://jmlr.org/proceedings/papers/v80/.

[85] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proceedings of the VLDB Endowment*, 7(11), 2014, https://doi.org/10.14778/2732967.2732968.

[86] F. Faerber, A. Kemper, P.-Å. Larson, J. Levandoski, T. Neumann, and A. Pavlo. Main Memory Database Systems. *Foundations and Trends in Databases*, 8(1-2), 2017, https://doi.org/10.1561/1900000058.

[87] W. Fan, C. Hu, and C. Tian. Incremental Graph Computations: Doable and Undoable. In *ACM International Conference on Management of Data*, 2017. https://doi.org/10.1145/3035918.3035944.

[88] W. Fan, X. Wang, and Y. Wu. Incremental Graph Pattern Matching. *ACM Transactions on Database Systems*, 38(3), 2013, https://doi.org/10.1145/2489791.

[89] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. T. Morris. Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018. https://www.usenix.org/conference/osdi18/presentation/gjengset.

[90] 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 *USENIX Symposium on Operating Systems Design and Implementation*, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[91] G. Graefe. Buffer Pool. In *Encyclopedia of Database Systems*. 2018. https://doi.org/10.1007/978-1-4614-8265-9_682.

[92] W. Group. Common Format For Exchange of Solved Load Flow Data. *IEEE Transactions on Power Apparatus and Systems*, PAS-92(6), 1973, https://doi.org/10.1109/TPAS.1973.293571.

[93] A. Gupta, I. S. Mumick, et al. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin*, 18(2), 1995. http://sites.computer.org/debull/95JUN-CD.pdf.

[94] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *Sigmod Record*, 22(2), 1993, https://doi.org/10.1145/170036.170066.

[95] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *International Semantic Web Conference*. 2004. https://doi.org/10.1007/978-3-540-30475-3_35.

[96] S. Haddadi and Z. Layouni. Consecutive Block Minimization is 1.5-Approximable. *Information Processing Letters*, 108(3), 2008, https://doi.org/10.1016/j.ipl.2008.04.009.

[97] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *European Conference on Computer Systems*, 2014. https://doi.org/10.1145/2592798.2592799.

[98] K. Hanauer, M. Henzinger, and C. Schulz. Recent Advances in Fully Dynamic Graph Algorithms – a Quick Reference Guide. *ACM Journal of Experimental Algorithmics*, 27(1.11), 2022, https://doi.org/10.1145/3555806.

[99] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *ACM International Conference on Management of Data*, 2008. https://doi.org/10.1145/1376616.1376713.

[100] I. Herman, G. Melançon, and M. S. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1), 2000, https://doi.org/10.1109/2945.841119.

[101] D. Holten and J. J. van Wijk. A User Study on Visualizing Directed Edges in Graphs. In *Human Factors in Computing Systems*, 2009. https://doi.org/10.1145/1518701.1519054.

[102] F. Holzschuher and R. Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j. In *Joint International Conference on Extending Database Technology/International Conference on Database Theory Workshops*, 2013. https://doi.org/10.1145/2457317.2457351.

[103] W. H. Ip and D. Wang. Resilience Evaluation Approach of Transportation Networks. In *International Joint Conference on Computational Sciences and Optimization*, 2009. https://doi.org/10.1109/CSO.2009.294.

[104] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *USENIX Symposium on Networked Systems Design and Implementation*, 2021. https://www.usenix.org/conference/nsdi21/presentation/iyer.

[105] H. V. Jagadish and A. Zhou, editors. *Proceedings of the VLDB Endowment, Volume 7*. 2013/2014. http://www.vldb.org/pvldb/volumes/7.

[106] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. Querying knowledge graphs by example entity tuples. In *IEEE International Conference on Data Engineering*, 2016. https://doi.org/10.1109/ICDE.2016.7498391.

[107] John West and Cherri M. Pancake, editors. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016. https://ieeexplore.ieee.org/xpl/conhome/7875333/proceeding.

[108] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An Active Graph Database. In *ACM International Conference on Management of Data*, 2017. https://doi.org/10.1145/3035918.3056445.

[109] K. Karanasos, A. Katsifodimos, and I. Manolescu. Delta: Scalable Data Dissemination Under Capacity Constraints. *Proceedings of the VLDB Endowment*, 7(4), 2013, https://doi.org/10.14778/2732240.2732241.

[110] A. Katifori, C. Halatsis, G. Lepouras, C. Vassilakis, and E. Giannopoulou. Ontology Visualization Methods: A Survey. *ACM Computing Surveys*, 39(4), 2007, https://doi.org/10.1145/1287620.1287621.

[111] *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. 2017. https://doi.org/10.1145/3097983.

[112] *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. 2018. https://doi.org/10.1145/3219819.

[113] K. Keeton and T. Roscoe, editors. *Proceedings of the Symposium on Operating Systems Design and Implementation*. 2016. https://www.usenix.org/conference/osdi16.

[114] A. Kemper. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *The VLDB Journal*, 4(3), 1995, https://doi.org/10.1007/BF01231646.

[115] L. T. Kou. Polynomial Complete Consecutive Information Retrieval Problems. *SIAM Journal on Computing*, 6(1), 1977, https://doi.org/10.1137/0206004.

[116] H. Kwak, C. Lee, H. Park, and S. B. Moon. What Is Twitter, a Social Network or a News Media? In *The Web Conference*, 2010. https://doi.org/10.1145/1772690.1772751.

[117] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. LeanStore: In-Memory Data Management beyond Main Memory. In *IEEE International Conference on Data Engineering*, 2018. https://doi.org/10.1109/ICDE.2018.00026.

[118] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), 2007, https://doi.org/10.1145/1217299.1217301.

[119] Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams, editors. *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. 2015. http://dl.acm.org/citation.cfm?id=2783258.

[120] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 8(3), 2014, https://doi.org/10.14778/2735508.2735517.

[121] M. Mariappan, J. Che, and K. Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *European Conference on Computer Systems*, 2021. https://doi.org/10.1145/3447786.3456230.

[122] M. Mariappan and K. Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *European Conference on Computer Systems*, 2019. https://doi.org/10.1145/3302424.3303974.

[123] F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe. Shared Arrangements: Practical Inter-Query Sharing for Streaming Dataflows. *Proceedings of the VLDB Endowment*, 13(10), 2020, https://doi.org/10.14778/3401960.3401974.

[124] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *Conference on Innovative Data Systems Research*, 2013. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[125] J. Mondal and A. Deshpande. CASQD: Continuous detection of activity-based subgraph pattern queries on dynamic graphs. In *Distributed and Event-based Systems*, 2016. https://doi.org/10.1145/2933267.2933316.

[126] B. Motik, Y. Nenov, R. Piro, and I. Horrocks. Maintenance of Datalog Materialisations Revisited. *Artificial Intelligence*, 269, 2019, https://doi.org/10.1016/j.artint.2018.12.004.

[127] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles*, 2013. https://doi.org/10.1145/2517349.2522738.

[128] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *USENIX Symposium on Networked Systems Design and Implementation*, 2011. https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing.

[129] T. Neumann and M. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*, 2020. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[130] M. Nikolic and D. Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *ACM International Conference on Management of Data*, 2018. https://doi.org/10.1145/3183713.3183758.

[131] D. Olteanu and J. Závodný. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems*, 40(1), 2015, https://doi.org/10.1145/2656335.

[132] S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, VU Amsterdam, 2004. https://www.cs.vu.nl/en/Images/SM_Orzan_5-11-2004_tcm210-258582.pdf.

[133] A. Pacaci, A. Bonifati, and M. T. Özsu. Regular Path Query Evaluation on Streaming Graphs. In *ACM International Conference on Management of Data*, 2020. https://doi.org/10.1145/3318464.3389733.

[134] A. Pacaci, A. Bonifati, and M. T. Özsu. Evaluating Complex Queries on Streaming Graphs. In *IEEE International Conference on Data Engineering*, 2022. https://doi.org/10.1109/ICDE53745.2022.00025.

[135] R. Pienta, A. Tamersoy, A. Endert, S. Navathe, H. Tong, and D. H. Chau. VISAGE: Interactive Visual Graph Querying. In *Advanced Visual Interfaces*, 2016. https://doi.org/10.1145/2909132.2909246.

[136] D. Precup and Y. W. Teh, editors. *Proceedings of the International Conference on Machine Learning*. 2017. http://jmlr.org/proceedings/papers/v70.

[137] M. Rath, D. Akehurst, C. Borowski, and P. Mäder. Are Graph Query Languages Applicable for Requirements Traceability Analysis? In *Requirements Engineering - Foundation for Software Quality*, 2017. https://ceur-ws.org/Vol-1796/poster-paper-2.pdf.

[138] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *ACM International Conference on Management of Data*, 2000. https://doi.org/10.1145/342009.335419.

[139] L. Ryzhyk and M. Budiu. Differential Datalog. In *Datalog 2.0 - CEUR Workshop*, volume 2368, 2019. http://ceur-ws.org/Vol-2368/paper6.pdf.

[140] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proceedings of the VLDB Endowment*, 11(4), 2017, https://doi.org/10.1145/3186728.3164139.

[141] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *The VLDB Journal*, 29(2-3), 2020, https://doi.org/10.1007/s00778-019-00548-x.

[142] S. Sahu and S. Salihoglu. Graphsurge: Graph Analytics on View Collections Using Differential Computation. In *ACM International Conference on Management of Data*, 2021. https://doi.org/10.1145/3448016.3452837.

[143] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to Roll a Join: Asynchronous Incremental View Maintenance. In *ACM International Conference on Management of Data*, 2000. https://doi.org/10.1145/342009.335393.

[144] S. Salihoglu and M. T. Özsu. Response to "Scale up or scale out for graph processing". *IEEE Internet Computing*, 22(5), 2018, https://doi.org/10.1109/MIC.2018.053681359.

[145] *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018. http://dl.acm.org/citation.cfm?id=3291656.

[146] S. Schelter. "Amnesia" - A Selection of Machine Learning Models That Can Forget User Data Very Fast. In *Conference on Innovative Data Systems Research*, 2020. http://cidrdb.org/cidr2020/papers/p32-schelter-cidr20.pdf.

[147] T. K. Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems*, 13(1), 1988, https://doi.org/10.1145/42201.42203.

[148] Shahram Ghandeharizadeh, Sumita Barahmand, Magdalena Balazinska, and Michael J. Freedman, editors. *Proceedings of the Symposium on Cloud Computing*. 2015. http://dl.acm.org/citation.cfm?id=2806777.

[149] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin. CrocodileDB: Efficient Database Execution through Intelligent Deferment. In *Conference on Innovative Data Systems Research*, 2020. http://cidrdb.org/cidr2020/papers/p14-shang-cidr20.pdf.

[150] R. Shirkova and J. Yang. Materialized Views. *Foundations and Trends in Databases*, 4(4), 2011, https://doi.org/10.1561/1900000020.

[151] O. Shmueli. Equivalence of Datalog Queries is Undecidable. *Journal of Logic Programming*, 15(3), 1993, https://doi.org/10.1016/0743-1066(93)90040-N.

[152] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 48(8), 2013, https://doi.org/10.1145/2517327.2442530.

[153] *Proceedings of the Symposium on Cloud Computing*. 2017. http://dl.acm.org/citation.cfm?id=3127479.

[154] *Proceedings of the Symposium on Cloud Computing*. 2018. http://dl.acm.org/citation.cfm?id=3267809.

[155] *Proceedings of the Symposium on Operating Systems Principles*. 2017. http://dl.acm.org/citation.cfm?id=3132747.

[156] J. P. G. Sterbenz, E. K. Çetinkaya, M. A. Hameed, A. Jabbar, and J. P. Rohrer. Modelling and Analysis of Network Resilience. In *International Conference on Communication Systems and Networks*, 2011. https://doi.org/10.1109/COMSNETS.2011.5716502.

[157] C. Stuecklberger. Expressing the Routing Logic of a SDN Controller as a Differential Dataflow. Master's thesis, ETH Zürich, 2016. http://hdl.handle.net/20.500.11850/155817.

[158] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent Query Processing. *Proceedings of the VLDB Endowment*, 12(11), 2019, https://doi.org/10.14778/3342263.3342278.

[159] Q.-C. To, J. Soto, and V. Markl. A Survey of State Management in Big Data Processing Systems. *The VLDB Journal*, 27(6), 2018, https://doi.org/10.1007/S00778-018-0514-9.

[160] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: A property graph query language. In *Graph Data Management Experiences and Systems*, 2016. https://doi.org/10.1145/2960414.2960421.

[161] C. Vehlow, F. Beck, and D. Weiskopf. Visualizing Group Structures in Graphs: A Survey. *Computer Graphics Forum*, 36(6), 2017, https://doi.org/10.1111/cgf.12872.

[162] K. Vora, R. Gupta, and G. Xu. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization*, 13(4), 2016, https://doi.org/10.1145/2992784.

[163] K. Vora, R. Gupta, and G. H. Xu. Kickstarter: Fast and Accurate Computations on Streaming Graphs Via Trimmed Approximations. In *Architectural Support for Programming Languages and Operating Systems*, 2017. https://doi.org/10.1145/3037697.3037748.

[164] C. Wang and J. Tao. Graphs in Scientific Visualization: A Survey. *Computer Graphics Forum*, 36(1), 2017, https://doi.org/10.1111/cgf.12800.

[165] G. Yadav and S. Babu. NEXCADE: Perturbation Analysis for Complex Networks. *PLOS ONE*, 7(8), 2012, https://doi.org/10.1371/journal.pone.0041827.

[166] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[167] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2010. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[168] P. Zhang, Y. Huang, A. Gember-Jacobson, W. Shi, X. Liu, H. Yang, and Z. Zuo. Incremental Network Configuration Verification. In *ACM Workshop on Hot Topics in Networks*, 2020. https://doi.org/10.1145/3422604.3425936.

[169] Q. Zhang, D. Balasubramanian, T. Kecskes, and J. Sztipanovits. Differential-FORMULA: Towards a Semantic Backplane for Incremental Modeling. In *ACM SIGPLAN Workshop on Domain-Specific Modeling*, 2021. https://doi.org/10.1145/3486603.3486779.

[170] Y. Zhao, C. Yuan, G. Liu, and I. Grinberg. Graph-based Preconditioning Conjugate Gradient Algorithm for "N-1" Contingency Analysis. In *IEEE Power & Energy Society General Meeting*, 2018. https://doi.org/10.1109/PESGM.2018.8586214.