UCLA
UCLA Electronic Theses and Dissertations

Title
Big Graph Analytics on Just A Single PC

Permalink
https://escholarship.org/uc/item/35m1r3rk

Author
Wang, Kai

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Big Graph Analytics on Just A Single PC

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Kai Wang

2019

© Copyright by
Kai Wang
2019

ABSTRACT OF THE DISSERTATION

Big Graph Analytics on Just A Single PC

by

Kai Wang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Harry Guoqing Xu, Chair

As graph data becomes ubiquitous in modern computing, developing systems to efficiently process large graphs has gained increasing popularity. There are two major types of analytical problems over large graphs: graph computation and graph mining. Graph computation includes a set of problems that can be represented through liner algebra over an adjacency matrix based representation of the graph. Graph mining aims to discover complex structural patterns of a graph, for example, finding relationship patterns in social media network, detecting link spam in web data.

Due to their importance in machine learning, web application and social media, graph analytical problems have been extensively studied in the past decade. Practical solutions have been implemented in a wide variety of graph analytical systems. However, most of the existing systems for graph analytics are distributed frameworks, which suffer from one or more of the following drawbacks: (1) many of the (current and future) users performing graph analytics will be domain experts with limited computer science background. They are faced with the challenge of managing a cluster, which involves tasks such as data partitioning and fault tolerance they are not familiar with; (2) not all users have access to enterprise cluster in their daily development tasks; (3) distributed graph systems commonly suffer from large startup and communication overhead; and (4) load balancing in a distributed system is another major challenge. Some graph algorithms have dynamic working sets and and it is

thus hard to distribute the workload appropriately before the execution.

In this dissertation, we identify three categories of graph workloads for which single-machine systems are more suitable than distributed systems: (1) analytical queries that do not need exact answers; (2) program analysis tasks that are widely used to find bugs in real-world software; and (3) graph mining algorithms that are important for many information-retrieval tasks.

Based on these observations, we have developed a set of single-machine graph systems to deliver efficiency and scalability specifically for these workloads. In particular, this dissertation makes the following contributions. The *first* contribution is the design and implementation of a single-machine graph query system named GraphQ, which divides a large graph into partitions and merges them with the guidance from an abstraction graph. By using multiple levels of abstraction, it can quickly rule out infeasible solutions and identify mergeable partitions. GraphQ uses the memory capacity as a budget and tries its best to find solutions before exhausting the memory, making it possible to answer analytical queries over very large graphs with resources affordable to a single PC. The *second* contribution is the design and implementation of Graspan, a single-machine, disk-based graph processing system tailored for interprocedural static analyses. Given a program graph and a grammar specification of an analysis, Graspan uses an edge-pair centric computation model to compute dynamic transitive closures on very large program graphs. With the help of novel graph processing techniques, we turn sophisticated code analyses into scalable Big Graph analytics. The *third* contribution of this dissertation is a single-machine, out-of-core graph mining system, called RStream, which leverages disk support to support efficient edge streaming for mining very large graphs. RStream employs a rich programming model that exposes relational algebra for developers to express a wide variety of mining tasks and implements a runtime engine that delivers efficiency with tuple streaming.

In conclusion, this dissertation attempts to explore the opportunities of building single-machine graph systems for scenarios where distributed systems do not work well. Our experimental results demonstrate that the techniques proposed in this dissertation can ef-

ficiently solve big graph analytical problems on a single consumer PC. We hope that these promising results will encourage future work to continue building affordable single-machine systems for a rich set of datasets and analytical tasks.

The dissertation of Kai Wang is approved.

Jens Palsberg

Miryung Kim

Todd D. Millstein

Harry Guoqing Xu, Committee Chair

University of California, Los Angeles

2019

To my family.

TABLE OF CONTENTS

List of Figures . . . . **x**

List of Tables . . . . **xii**

Acknowledgments . . . . **xv**

Vita . . . . **xvi**

1 Introduction **1**

2 GraphQ: Graph Query Processing with Abstraction Refinement **7**

2.1 Overview and Programming Model 9

2.2 Abstraction-Guided Query Answering 18

2.3 Design and Implementation 22

2.4 Queries and Methodology 24

2.5 Evaluation 27

2.5.1 Query Efficiency 27

2.5.2 Comparison to GraphChi-ET 32

2.5.3 Impact of Abstraction Refinement 33

2.6 Summary and Interpretation 34

3 Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code **37**

3.1 Background 42

3.1.1 Graph Reachability 42

3.1.2 Pointer Analysis 43

3.2 Graspan's Programming Model 48
3.3 Graspan Design and Implementation 51
3.3.1 Preprocessing 52
3.3.2 Edge-Pair Centric Computation 53
3.3.3 Postprocessing 57
3.4 Evaluation 58
3.4.1 Effectiveness of Interprocedural Analyses 60
3.4.2 Graspan Performance 64
3.4.3 Comparisons with Other Analysis Implementations 66
3.4.4 Comparisons with Other Backend Engines 67
3.5 Summary and Interpretation 68

4 RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine 70
4.1 Background and Overview 75
4.1.1 Background 75
4.1.2 RStream Overview 76
4.2 Programming Model 81
4.3 RStream Implementation 88
4.3.1 Preprocessing 89
4.3.2 Join Implementation 89
4.3.3 Redundancy Removal via Automorphism Checks 92
4.3.4 Pattern Aggregation via Isomorphism Checks 93
4.4 Evaluation 95
4.4.1 Comparisons with Mining Systems 97

4.4.2 Comparisons with Datalog Engines 102
4.4.3 RStream Performance Breakdown 103
4.5 Summary and Interpretation 106

5 Related Work 107
5.1 Single-Machine Graph Computation Systems 107
5.2 Distributed Graph Computation Systems 108
5.3 Approximate Queries 109
5.4 Static Bug Finding 109
5.5 Grammar-guided Reachability 110
5.6 Distributed Mining Systems 111
5.7 Specialized Graph Mining Algorithms 111
5.8 Datalog Engines 112
5.9 Dataflow Systems 112

6 Conclusions and Future Work 114
6.1 Conclusions 114
6.2 Future Work 115

References 117

LIST OF FIGURES

2.1 An example graph, its abstraction graph, and the computation steps for finding a clique whose size is $\geq$5. The answer of the query is highlighted. 10

2.2 Programming for answering clique Queries. 11

2.3 Ratios between the running times of GraphChi-ET and GraphQ over twitter-2010: (a) PageRank: Max = 3.0, Min = 0.5, GeoMean = 1.6; (b) Clique: Max = 48.3, Min = 4.1, GeoMean = 13.4; and (c) Community: Max = 7.5, Min = 1.4, GeoMean = 4.2. 32

2.4 GraphQ's running time (in seconds) for answering PageRank queries over twitter-2010 using different abstraction graphs: *Random* means no refinement is used and partitions are merged randomly; $a = i$ means a partition is represented by i abstract vertices. 34

3.1 A program and its expression graph: solid, horizontal edges represent assignments (A- and M- edges); dashed, vertical edges represent dereferences (D-edge); dotted, horizontal edges represent transitive edges labeled non-terminals. *A4* indicates the allocation site at Line 4. 45

3.2 (a) An example graph, (b) its partitions, and (c) the in-memory representation of edge lists. 51

3.3 Two representative bugs in the Linux kernel 4.4.0-rc5 that were missed by the baseline checkers. 62

3.4 Percentages of added edges across supersteps. 66

4.1 A Triangle Counting example in RStream; highlighted in each table is its key column. For each table, only a small number of relevant tuples are shown. . . . 77

4.2 Triangle counting in RStream. 78

4.3 Major data structures. 82

4.4 API functions. 83

4.5 A graphical illustration of `join_on_all_columns`; the streaming partitions #1 and #2 contain vertices [0, 10] and [11, 25], respectively; suppose `new_key` returns 2 (which is column C_3). Structural info is not shown. 86

4.6 An FSM program; structural info is needed. 87

4.7 A graphical illustration of multiple producers, multiple consumers and reshuffling buffers. 91

4.8 A graph and its canonical tuples of size 3. 93

4.9 Aggregation example of three isomorphic tuples. 94

4.10 FSM performance comparisons with different pattern sizes and supports over the Patents graph. Tall red bars on the right of each group represent Arabesque failures. 100

4.11 (a) Comparisons between RStream (RS), BigDatalog (BD-n), and SociaLite (SL) on TC and CC; (b) Closure comparison over CiteSeer. 102

4.12 RStream's scalability (a), I/O throughput when running CC over UK (b), and I/O throughput when running TC over UK (c). I/O was measured with `iostat`. 105

LIST OF TABLES

2.1 A summary of queries performed in the evaluation: reported are the names and forms of the queries, initial partition selection, priority of partition merging, whole-graph computation times in GraphChi for the uk-2005 and the twitter-2010 graphs, and the time for pre-processing them; $\uparrow$ ($\downarrow$) means the higher (lower) the better; each pre-processing time has two components $a+b$, where a represents the time for partitioning and AG construction, and b represents the time for initial (local) computation; "?" means the whole-graph computation cannot finish in 48 hours. 24

2.2 Our graph inputs: reported in each section are their names, types, numbers of vertices, numbers of edges, numbers of initial partitions (IP), numbers of maximum partitions allowed to be merged before out of budget (MP), and numbers of partitions increased at each step (δ, *cf.* line 13 in Figure 2.2). 25

2.3 GraphQ performance for answering PageRank queries over uk-2005; each section shows the performance of answering queries on pagerank values that belong to an interval in the top 100 vertex list; reported in each section are the number of entities requested to find (Δ), the average query answering time in seconds ($Time$), and the number of partitions merged when a query is answered (Par). 28

2.4 GraphQ's performance for answering Clique queries over twitter-2010; a "-" sign means some queries in the group could not be answered. 29

2.5 GraphQ's performance for answering Community queries over uk-2005; each section reports the average time for finding communities whose sizes belong to different intervals in the top 100 community list. 30

2.6 GraphQ's performance for answering Path queries over twitter-2010. 30

2.7 GraphQ's performance for answering Triangle queries over uk-2005. 31

2.8 A breakdown of time on computation and I/O for GraphQ and GraphChi-ET for PageRank, Clique, and Comm; measurements were obtained by running the most difficult queries from Figure 2.3. 33

3.1 A subset of checkers used by [44] and [113] to find bugs in the Linux kernel, their target problems, their limitations, the potential ways to improve them using a sophisticated interprocedural analysis; the first six have been used by Chou et al. [53] and Palix et al. [113] to study Linux bugs; the last one was described in a recent paper by Brown et al. [44] to find potential NULL pointer dereferences; positive/negative indicates whether the limitation can result in false positives/negatives. 38

3.2 Programs analyzed, their versions, numbers of lines of code, and numbers of function inlines. 59

3.3 Checkers implemented, their numbers of bugs reported by the baseline checkers (BL), and *new bugs* reported by our Graspan analyses (GR) on top of the BL checkers on the Linux kernel 4.4.0-r5; **RE** shows total numbers of bugs reported while **FP** shows numbers of false positives determined manually; to provide a reference of how bugs evolve over the last decade, we include an additional section **BL(2.6.1)** with numbers of *true* bugs reported by the same checkers in 2011 on the kernel version 2.6.1 from [113]. UNTest is a new *interprocedural* checker we implemented to identify unnecessary NULL tests; '+' means new problems found. 60

3.4 A breakdown of the new Linux bugs found by our analyses; in parentheses are numbers of false positives. 63

3.5 Graspan performance: reported are the numbers of vertices and edges before (**IS**) and after (**PS**) being processed by Graspan, Graspan's pre-processing time (**PT**), numbers of supersteps taken (**#SS**), and total running time (**T**). 64

3.6 A comparison on the performance of Graspan, on-demand pointer analysis (ODA) [174] implemented in standard ways, as well as SociaLite [90] processing our program graphs in Datalog. The Graspan section shows a breakdown of the running times into computation time (**CT**), I/O time (**I/O**), and garbage collection time (**GC**); P and D represent pointer/alias analysis and dataflow analysis. OOM means out of memory. 65

4.1 Real world graphs. 96

4.2 Algorithms experimented. 96

4.3 Comparisons between RStream (RS), Arabesque (AR-n), ScaleMine (SM-n), and DistGraph(DG-n) on four mining algorithms — triangle counting (TC), Clique (k-C), Motif Counting (k-M), and FSM (k-F) — over three graphs CiteSeer (CS), MiCo (MC), and Patents (PA); n represents the number of nodes the distributed systems use; k is the size of the structure to be mined; '-' indicates execution failures. For FSM, four different support parameters (300, 500, 1K, and 5K) are used and explicitly shown in each 3-F row. Highlighted rows are the shortest times (in seconds). 98

4.4 FSM performance comparisons between RStream and GraMi over Patents and Mico; time is measured in seconds. 101

4.5 The number of tuples (**Tuples**) generated for each phase execution, the size of each tuple (**TS**), and the number of bytes (**#MB**) shuffled for 4-Motif over the Patents graph and 4-FSM, S=10K over the Mico graph. 104

4.6 Ratios between the final disk usage and original graph size (in the binary format). 105

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Professor Harry Guoqing Xu, who has spent a tremendous amount of effort providing continuous support and guidance through my Ph.D. studies. He is thoughtful and passionate. During the past six years, I have learned how to pursue my research interests, collaborate with others, overcome challenges, and build practical systems. My graduate career would have not been possible without his dedicated mentorship. I am so proud to join his research group and be one of his students.

I would like to thank Professor Jens Palsberg, Professor Miryung Kim and Professor Todd Millstein, for serving on my final defense committee. Their valuable comments have always strengthened my research.

I would like to thank my colleagues, Zhiqiang Zuo, Khanh Nguyen, Lu Fang, Yingyi Bu, Aftab Hussain, Cheng Cai, Bojun Wang, John Thorpe, Tim Nguyen, Christian Navasca for their great support.

This work would not have been possible without the amazing encouragement and support from my wife and my parents. I thank my wife Yamin for her love and support, for every day we have spent together, for her taking care of most of the family duties during my six years' Ph.D. studies. I thank my daughter Annie for filling my life with love and happiness. I thank my parents Dequn and Gaixiang for their continuous support, for believing in me.

The material presented in this dissertation is based upon work supported by the National Science Foundation under the grants CCF-1054515, CCF-1117603, CNS-1321179, CNS-1319187, CCF-1349528, and CCF-1409829, and by the Office of Naval Research under grant N00014-14- 1-0549 and N00014-16-1-2913.

VITA

2018-2019	Graduate Research Assistant, Computer Science Department, UCLA
2013-2018	Graduate Research Assistant, Computer Science Department, UC Irvine
Jun 2009	Master of Science in Computer Science, Chinese Academy of Sciences, Beijing, China
Jun 2006	Bachelor of Science in Computer Science, Huazhong University of Science and Technology, Wuhan, China

PUBLICATIONS

Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale System Code. In *European Conference on Computer Systems (EuroSys'19)*, Article No. 38, 2019.

Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 763-782, 2018.

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, pages 389–404,

2017.

Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. Understanding and Combating Memory Bloat in Managed Data-Intensive Systems. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26, 4, Article 12 (January 2018).

Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *2015 USENIX Annual Technical Conference (USENIX ATC'15)*, pages 387–401, 2015.

Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, pages 675-690, 2015.

CHAPTER 1

Introduction

As graph data becomes ubiquitous in modern computing, developing systems to efficiently process large graphs has gained increasing popularity. Due to their importance in machine learning, web application and social media, graph analytical problems have been extensively studied in the past decade. Practical solutions have been implemented in a wide variety of graph systems [68, 52, 67, 72, 98, 88, 135, 105, 177, 126, 153, 173, 73, 125, 147, 176].

There are two major types of analytical problems over large graphs: graph computation and graph mining. Graph computation includes a set of problems that can be represented through liner algebra over an adjacency matrix based representation of the graph. As a typical example of graph computation, PageRank [112] can be modeled as iterative sparse matrix and vector multiplications. Graph mining aims to discover complex structural patterns of a graph, for example, finding relationship patterns in social media network, detecting link spam in web data. As a typical example of graph mining, frequent sub-graph mining finds all sub-graphs with frequency above a user-defined threshold in a labeled input graph.

However, most of the existing systems for graph analytics are distributed frameworks, which suffer from one or more of the following drawbacks:

- In order to scale to large graphs, graph systems often need enterprise clusters with hundreds or even thousands of computation nodes. Many of the (current and future) users performing graph analytics will be domain experts with limited computer science background. It is much easier for them to host the system on their own machines rather than relying on a cluster, which involves tasks such as fault tolerance and cluster management they are not familiar with.

- Not all users have access to enterprise cluster in their daily development tasks. Even if they do, running a simple graph analytics on a relatively small graph does not seem to justify very well the cost of blocking hundreds or even thousands of machines for several hours.

- Load balancing in a distributed system is another major challenge. Algorithms such as frequent sub-graph mining have dynamic working sets. Their search space is often unknown in advance and it is thus hard to partition the graph and distribute the workload appropriately before the execution.

- Distributed graph systems commonly suffer from large startup and communication overhead. For small graphs, it is difficult for the startup/communication overhead to get amortized over the processing.

In this dissertation, we identify three categories of graph workloads for which single-machine systems can outperform distributed systems.

- **Category 1: Analytical queries that do not need exact answers.** For example, queries such as "find one path between LA and NYC whose length is less than 3,000 miles" have many usage scenarios *e.g.*, any path whose length is smaller than a threshold between two cities is acceptable for a navigation system. It appears that many of these analytical queries can be effectively computed by exploring only a small fraction of the graph, and traversing the complete graph is an overkill. If partial graphs are sufficient, we can answer analytical queries on one single PC so that the client can be satisfied without resorting to clusters.

- **Category 2: Graph-based program analysis tasks that are widely used to find bugs in real-world software.** Our key observation is that many interprocedural analyses can be formulated as a *graph reachability* problem [119, 139, 117, 129, 165]. Since program analysis is intended to assist developers to find bugs in their daily development tasks, their machines are the environments in which we would like our

system to run, so that developers can check their code on a regular basis without needing to access a cluster. Hence, disk-based graph system naturally becomes our choice.

- **Category 3: Graph mining algorithms that are important for many information retrieval tasks.** Mining workloads are memory-intensive because the amount of intermediate data for a typical mining algorithm grows exponentially with the size of the graph. By utilizing disk space available in modern machines, a disk-based system can satisfy the large storage requirement of mining algorithms.

The overarching goal of this dissertation is to build a set of efficient and scalable single-machine systems for important graph-analytical tasks. Our key insight is consistent with the recent trend on building single-machine graph computation systems [88, 126, 153, 148, 101, 173, 17, 177] — given the increasing accessibility of high-volume SSDs, a disk-based system can satisfy the large storage requirement of graph algorithms by utilizing disk space available in modern machines; yet it does not suffer from any startup and communication inefficiencies that are inherent in distributed computing. We make the following contributions:

- **Contribution 1: A single-machine graph querying framework.** We build GraphQ, a novel graph processing framework for analytical queries. The centerpiece of GraphQ is the novel idea of *abstraction refinement*, where the very large graph is represented as multiple levels of abstractions, and a query is processed through iterative refinement across graph abstraction levels. As a result, GraphQ enjoys several distinctive traits unseen in existing graph processing systems: query processing is naturally *budget-aware*, friendly for *out-of-core processing* when "Big Graphs" cannot entirely fit into memory, and endowed with strong correctness properties on query answers. With GraphQ, a wide range of complex *analytical queries* over very large graphs can be answered with resources affordable to *a single PC*, which complies with the recent trend advocating single-machine-based Big Data processing.

  Experiments show GraphQ can answer queries in graphs 4-6 times bigger than the

memory capacity, only in several seconds to minutes. In contrast, GraphChi, a state-of-the-art graph processing system, takes hours to days to compute a whole-graph solution. An additional comparison with a modified version of GraphChi that terminates immediately when a query is answered shows that GraphQ is on average 1.6–13.4× faster due to its ability to process partial graphs.

- **Contribution 2: A single-machine graph system for interprocedural static analyses.** We build Graspan, a *single machine, disk-based* parallel graph processing system tailored for interprocedural static analyses. Given a program graph and a grammar specification of an analysis, Graspan offers two major performance and scalability benefits: (1) the core computation of the analysis is automatically parallelized and (2) out-of-core support is exploited if the graph is too big to fit in memory. At the heart of Graspan is a parallel *edge-pair* (EP) centric computation model that, in each iteration, loads two partitions of edges into memory and "joins" their edge lists to produce a new edge list. Whenever the size of a partition exceeds a threshold value, its edges are repartitioned. Graspan supports both in-memory (for small programs) and out-of-core (for large programs) computation. Joining of two edge lists is fully parallelized, allowing multiple transitive edges to be simultaneously added.

  We have implemented Graspan in both Java and C++. Graspan can be readily used as a "backend" analysis engine to enhance the existing static checkers such as BugFinder, PMD, or Coverity. We have performed a thorough evaluation of Graspan on three systems programs including the Linux kernel, the PostgreSQL database, and the Apache httpd server. Our experiments show very promising results: (1) the two Graspan-based analyses scale easily to these systems, which have many millions of function inlines, with several hours processing time, while their traditional implementations crashed in the early stage; (2) in terms of LoC, the Graspan-based implementations of these analyses are an order-of-magnitude simpler than their traditional implementations; (3) using the results of these interprocedural analyses, the static checkers in [113] have uncovered a total of 85 potential bugs.

- **Contribution 3: A single-machine graph mining framework.** We build RStream, a *single-machine, out-of-core* mining system that leverages disk support to store intermediate data. At its core are two innovations: (1) To enable easy programming of mining algorithms with and without statically-known structural patterns, we propose a novel programming model, referred to as *GRAS*, which adds relational algebra into gather-apply-scatter (GAS) model. Many mining algorithms, including FSM, Triangle and Motif Counting, or Clique, can all be easily developed with less than 80 lines of code under GRAS; and (2) We build a runtime engine that implements relational algebra efficiently with *tuple streaming*. Since the number of edges/updates is much larger than the number of vertices for a graph, edge streaming provides efficiency by sequentially accessing edge data from disk (as edges are sequentially read but not stored in memory) and randomly accessing vertex data held in memory. Streaming essentially provides an *efficient, locality-aware join implementation*. RStream leverages this insight to implement relational operations.

  A comparison between RStream and four state-of-the-art distributed mining/Datalog systems — Arabesque, ScaleMine, DistGraph, and BigDatalog — demonstrates that RStream outperforms all of them, running on a 10-node cluster, *e.g.* by at least a factor of 1.7×, and can process large graphs on an inexpensive machine.

Impact GraphQ proposed in this dissertation is the first graph processing system that can answer analytical queries over partial graphs. GraphQ is built on a key insight that many interesting graph properties — such as finding cliques of a certain size, or finding vertices with a certain page rank — can be effectively computed by exploring only a small fraction of the graph, and traversing the complete graph is an overkill. With GraphQ, a wide range of complex analytical queries over very large graphs can be answered with resources affordable to a single PC. We hope that GraphQ will open up new possibilities to scale up Big Graph processing with small amounts of resources.

Graspan is the first attempt to turn sophisticated code analysis into *Big Graph analytics*

and leverage novel graph processing techniques to solve this traditional programming language problem. RStream uses an *edge-pair* centric computation model to compute *dynamic transitive closures* on very large program graphs. An evaluation of these static analyses on large codebases such as Linux shows that their Graspan implementations scale to millions of lines of code and are much simpler than their original implementations. Moreover, we show that these analyses can be used to augment the existing checkers. We hope that our work will open up a new direction for scaling various sophisticated static program analyses (*e.g.*, symbolic execution, theorem proving, *etc.*) to large systems.

RStream is the first single-machine, out-of-core graph mining system. RStream employs a new GRAS programming model that uses a combination of GAS and relational algebra to support a wide variety of mining algorithms. At the low level, RStream leverages tuple streaming to efficiently implement relational operations. Our experimental results demonstrate that RStream can be more efficient than state-of-the-art distributed mining systems. We hope that these promising results will encourage future work that builds disk-based systems to scale expensive mining algorithms.

Organization We propose GraphQ, a single-machine scalable querying framework for very large graphs in Chapter 2. Chapter 3 presents the design and implement of Graspan, a single-machine graph system tailored for interprocedural static analyses. Chapter 4 proposes RStream, a single-machine, out-of-core graph mining system that leverages disk support to store intermediate data. Related work is discussed in Chapter 5. Chapter 6 concludes the dissertation and presents future work.

CHAPTER 2

GraphQ: Graph Query Processing with Abstraction Refinement

Developing scalable systems for efficient processing of very large graphs is a key challenge faced by Big Data developers and researchers. Given a graph analytical task expressed as a set of user-defined functions (UDF), existing processing systems compute a *complete solution* over the input graph. Despite much progress, computing a complete solution is still time-consuming. For example, using a 32-node cluster, it takes Preglix [45], a state-of-the-art graph processing system, more than 2,500 seconds to compute a complete solution (*i.e.*, all communities in the input graph) over a 70GB webgraph for a simple community detection algorithm.

While necessary in many cases, the computation of complete solutions — and the overhead of maintaining them — seems an overkill for many real-world applications. For example, queries such as "find one path between LA and NYC whose length is $\leq$ 3,000 miles" or "find 10 programmer communities in Southern California whose sizes are $\geq$ 1000" have many real-world usage scenarios *e.g.*, any path whose length is smaller than a threshold between two cities is acceptable for a navigation system. Unlike database queries that can be answered by filtering records, these queries need (iterative) computations over graph vertices and edges. In this chapter, we refer to such queries as *analytical queries*. Furthermore, it appears that many of them can be answered by exploring only a *small fraction* of the input graph — if a solution can be found in a subgraph of the input graph, why do we have to exhaustively traverse the entire graph?

This chapter is a quest driven by two simple questions: given the great number of real-

world applications that need analytical queries, can we have a ground-up redesign of graph processing systems — from the programming model to the runtime engine — that can facilitate query answering over *partial graphs*, so that a client application can quickly obtain satisfactory results? If partial graphs are sufficient, can we answer analytical queries on one single PC so that the client can be satisfied without resorting to clusters?

We propose *GraphQ*, a novel graph processing framework for analytical queries. In GraphQ, an analytical query has the form *"find n entities from the graph with a given quantitative property"*, which is general enough to express a large class of queries, such as page rank, single source shortest path, community detection, connected components, *etc.* At its core, GraphQ features two interconnected innovations:

- A simple yet expressive *partition-check-refine* programming model that naturally supports programmable analytical queries processed through *incremental* accesses to graph data

- A novel *abstraction refinement* algorithm to support efficient query processing, fundamentally decoupling the resource usage for graph processing from the (potentially massive) size of the graph

From the perspective of a GraphQ user, the very large input graph can be divided into *partitions*. How partitions are defined is programmable, and each partition on the high level can be viewed as a subgraph that GraphQ queries operate on. Query answering in GraphQ follows a repeated lock-step *check-refine* procedure, until either the query is answered or the budget is exhausted.

In particular, (1) the *check* phase aims to answer the query over each individual partition without considering inter-partition edges connecting these partitions. A query is successfully answered if a *check* predicate returns true; (2) if not, a *refine* process is triggered to identify a set of inter-partition edges to add back to the graph. These recovered edges will lead to a broader scope of partitions to assist query answering, and the execution loops back to step (1). Both the *check* procedure (determining whether the query is answered) and the

refine procedure (determining what new inter-partition edges to include) are programmable, leading to a programming model suitable for defining complex analytical queries with significant in-graph computations.

Key to finding the most profitable inter-partition edges to add in each step is a novel *abstraction refinement* algorithm at the core of its query processing engine. Conceptually, the "Big Graph" under GraphQ is summarized into an *abstraction graph*, which can be intuitively viewed as a "summarization overlay" on top of the complete concrete graph (CG). The abstraction graph serves as a compact "navigation map" to guide the query processing algorithm to find profitable partitions for refinement.

Usage Scenarios We envision that GraphQ can be used in a variety of real-world data analytical applications. Example applications include:

- *Target marketing:* GraphQ can help a business quickly find a target group of customers with given properties;
- *Navigation:* GraphQ can help navigation systems quickly find paths with acceptable lengths
- *Memory-constrained data analytics:* GraphQ can provide good-enough answers for analytical applications with memory constraints

2.1 Overview and Programming Model

Background Common to graph processing systems, the graph operated by GraphQ can be mathematically viewed as a directed (sparse) graph, $G = (V, E)$. A value is associated with each vertex $v \in V$, indicating an application-specific property of the vertex. For simplicity, we assume vertex values are labeled from 1 to $|V|$. Given an edge e of the form $u \rightarrow v$ in the graph, e is referred to as vertex v's *in-edge* and as vertex u's *out-edge*. The developer specifies an `update`(v) function, which can access the values of a vertex and its neighboring

vertices. These values are fed into a function f that computes a new value for the vertex. The goal of the computation is to "iterate around" vertices to update their values until a global "fixed-point" is reached. This vertex-centric model is widely used in graph processing systems, such as Pregel [102], Pregelix [45], and GraphLab [99].

Figure 2.1 shows a simple directed graph that we will use as a running example throughout this chapter. For each GraphQ query, the user first needs to find a related *base application* that performs whole-graph vertex-centric computation. This is not difficult, since many of these algorithms are readily available. In our example, the base application is Maximal Clique, and the query aims to find a clique whose size is no less than 5 (*i.e.* goal) over the input graph.

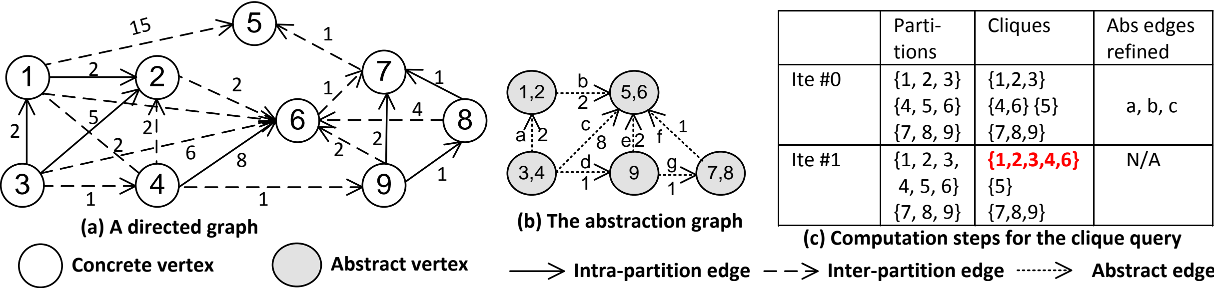


Figure 2.1: An example graph, its abstraction graph, and the computation steps for finding a clique whose size is ≥5. The answer of the query is highlighted.

GraphQ first divides the concrete graph in Figure 2.1 (a) into three *partitions* — {1, 2, 3}, {4, 5, 6}, and {7, 8, 9} — a "pre-processing" step that only needs to be performed once for each graph. When the query is submitted, the goal of GraphQ is to use an *abstraction graph* to guide the selection of partitions to be merged, hoping that the query can be answered by merging only a very small number of partitions. Initially, inter-partition edges (shown as arrows with dashed lines) are disabled; they will be gradually recovered.

Programming Model A sample program for answering the clique query can be found in Figure 2.2. Overall, GraphQ is endowed with an expressive 2-tier programming model to

```
// end-user
Graph g = new ExampleGraph(); // A partitioned graph
CliqueQuery cq = new CliqueQuery(g, 5);
List<Clique> qr = cq.submit();

// library programmer
class CliqueQuery extends Query {
    final Graph G; // graph
    final int N; // goal
    final int M; // max # of results to refine with
    final int K; // max # of partitions to merge
    final int delta; // the inc over K at each refinement

    List<Partition> initPartitions()
    { return g.partitions; }

    boolean check(Clique c) {
        if (c.size()>=N) { report(c); return true; }
     }

    List<AbstractEdge> refine(Clique c1, Clique c2) {
        List<AbstractEdge> list;
        foreach(Vertex v in c1.vertices())
        foreach(Vertex u in c2.vertices())
        AbstractEdge ae = g.abstractEdge(u, v);
        if (ae != null) { list.add(ae); }
        return list;
    }

    int resultThreshold() { return M; }
    int partitionThreshold() { return K; }

    CliqueQuery(Graph g, int n) {
        this.G = g; this.N = n;
    }
}
class Clique extends QueryResult {
    int refinePriority()  { return size(); }
    int size() {...}
}
```

Figure 2.2: Programming for answering clique Queries.

balance simplicity and programmability:

- First, GraphQ *end users* only need to write 2-3 lines of code to submit a query. For example, the end user writes lines 2-5, submitting a `CliqueQuery` to look for `Clique` instances whose size is no fewer than 5 over the `ExampleGraph`.
- Second, GraphQ *library programmers* define how a query can be answered through a flexible programming model that fully supports in-graph computation. In the example, the clique query is defined between lines 7-40, by extending the `Query` and `QueryResult` classes in our library.

We expect regular GraphQ users — those who only care about *what* to query but not *how* to query it — to program only the first tier (between lines 2-5). The appeal of the GraphQ programming model lies in its flexibility. On one hand, the simplicity of the GraphQ first-tier interface is on par with query languages for similar purposes (such as SQL). On the other hand, for programmers concerned with graph processing efficiency, GraphQ provides opportunities for full-fledged programming "under the hood" at the second tier.

Partitions Given a very large graph, one can specify how it is partitioned using GraphQ parameters. A partition is both a logical and a physical concept. Logically, a partition is a subgraph (connected component) of the concrete graph. Physically, it is often aligned with the physical storage unit of data, such as a disk file. In our formulation where the graph vertices are labeled with numbers from 1 to $|V|$, we select partitions as containing vertices with continuous label numbers, and edges connecting those vertices in the concrete graph. Beyond this mathematical formulation is an intuitive goal: if we use labels 1 to $|V|$ to mimic the physical sequence of vertex storage, the partitions should be created to be as aligned with physical storage as possible. Thanks to this design, loading a partition is very efficient due to sequential disk accesses with strong locality.

When a query is defined — such as `CliqueQuery` — the programmer first decides what partitions should be *initially* considered to compute local solutions (*e.g.* cliques). This is

supported by overriding the `initPartitions` method of the `Query` class, as in line 16. In our example, this method selects all partitions because we have no knowledge of whether and what cliques exist in each partition initially. GraphQ loads one partition into memory at a time and performs vertex-centric computation on the partition to compute local cliques independently of other partitions.

Observe that this does not contradict with our early discussion of incremental graph data processing: at the local computation phase, all partition-based computations are independent of each other. Therefore, when the data for one partition is loaded, the data for previously loaded partitions can be written back to disk, and at this phase GraphQ does not need to hold data in memory for more than one partition. Overall, this phase is very efficient because all inter-partition edges are ignored and there are only a very small number of random disk accesses.

Abstraction Graph The abstraction graph (AG) summarizes the concrete graph. Each *abstract vertex* in the AG abstracts a set of concrete vertices and each *abstract edge* connects two abstract vertices. An abstract edge can have an *abstract weight* that abstracts the weights of the actual edges it represents.

To see the motivation behind the design of AG, observe that inter-partition edges can scatter across the partitions (*i.e.*, disk files) they connect, and knowing whether a concrete edge exists between two partitions requires loading both partitions into memory and a linear scan of them, a potentially costly step with a large number of disk accesses. As a "summarization" of the concrete graph, the AG is much smaller in size and can be *always* held in memory.

GraphQ first checks the existence of an abstract edge on the AG: the non-existence of an abstract edge between two abstract vertices $\bar{u}$ and $\bar{v}$ guarantees the non-existence of a concrete edge between any pair of concrete vertices (u, v) abstracted by $\bar{u}$ and $\bar{v}$; hence, we can safely skip the check of concrete edges. On the other hand, the existence of an abstract edge does not necessarily imply the existence of a concrete edge, and hence, the abstract

edge needs to be *refined* to recover the concrete edges it represents.

The granularity of the AG is a design issue to be determined by the user. At one extreme, each partition can be an abstract vertex in the AG. This very coarse-grained abstraction may not be precise enough for GraphQ to quickly eliminate infeasible concrete edges. At the other extreme, a very fine-grained AG may take much space and the computation over the AG (such as a lookup) may take time. Since the AG is always in memory to provide quick guidance, a rule of thumb is to allow the abstraction granularity (*i.e.*, the number of concrete vertices represented by one abstract vertex) to be proportional to the memory capacity.

Using parameters, the user can specify the ratio between the size of the AG and the main memory — the more memory a system has, the larger AG will be constructed by GraphQ to provide more precise guidance. Figure 2.1 (b) shows the AG for the concrete graph in Figure 2.1 (a). The GraphQ runtime uses the simple *interval domain* [55] to abstract concrete vertices — each abstract vertex represents two concrete vertices that have consecutive labels. This simple design turns out to be friendly for performance as well: each abstract edge represents a set of concrete edges stored together in the partition file; since refining an abstract edge needs to load its corresponding concrete edges, storing/loading these edges together maximizes sequential disk accesses and data locality. A detailed explanation of the storage structure can be found in Section 2.3.

An alternative route we decide not to pursue is to provide the user full programmability to construct their own AGs. The issue at concern is *correctness*. Our design of the abstraction graph is built upon the principled idea of abstraction refinement, with correctness guarantees (Section 2.2). The correctness is hinged upon that the AG is indeed a “sound” abstraction of the concrete graph. We rely on the GraphQ runtime to maintain this notion of sound abstraction.

Abstraction Refinement At the end of each local computation (*i.e.*, over a partition), GraphQ invokes the `check` method of the `Query` object. The method returns **`true`** if the query can be answered, and the result is reported through the `report` method (see line 19).

Query processing terminates. If all local computations are complete and all `check` invocations return **false**, GraphQ tries to merge partitions to provide a larger scope for query answering. Recall that in our initial partition definition, all inter-partition edges have been ignored. The crucial challenge of partition merging thus becomes recovering the inter-partition edges, a process we call *abstraction refinement*.

In GraphQ, the refinement process is guided by the `QueryResult` — `Clique` in our example — from local computations. The key insight is that the results so far should offer clues on which partitions should be merged at a higher priority. The "priority" here can be customized by programmers through overriding the `refinePriority` method of class `QueryResult`. In the clique query example here, the programmer uses the size of the clique as the metric for priority (see line 39). Intuitively, merging partitions where larger cliques have been discovered is more likely to reach the goal of finding a clique of a certain size.

GraphQ next selects M (returned by `resultThreshold` in line 31) results with the highest priorities (*i.e.* largest cliques) for pairwise inspection. For each pair of cliques resulting from different partitions, the `refine` method (line 22) is invoked to verify if there is any potential for the two input cliques to combine into a larger clique. `refine` returns a list of abstract edges that should be refined. The implementation of `refine` is provided by programmers, typically involving the consultation of the AG. In our example, the method returns a list of candidate abstract edges whose corresponding concrete edges may potentially connect vertices from the two existing cliques (in two partitions) in order to form a larger clique.

Based on the returned abstract edges, GraphQ consults the AG to find the concrete edges these abstract edges represent. GraphQ then merges the partitions in which these concrete edges are located. To avoid a large number of partitions to be merged at a time — that would require the data associated all partitions to be loaded into memory at the same time — programmers can set a threshold specified by `partitionThresold`, in line 32. GraphQ adopts an iterative merging process: in each pass, merging only happens when the refinement leads to the merging of no more than K (returned by `partitionThresold`) partitions. If the merged partitions cannot answer the queries, GraphQ increases K by δ (line 13) at each

subsequent pass to explore more partitions. This design enables GraphQ to gradually use more memory as the query processing progresses.

GraphQ terminates query processing in one of the 3 scenarios: (1) the `check` method returns **true**, in which case the query is answered; (2) all partitions are merged in one, and the `check` method still returns **false** — a situation in which this query is impossible to answer; and (3) a (memory) budget runs out, in which case GraphQ returns the best `QueryResult`s that have been found *so far*. We will rigorously define this notion in Section 2.2.

Example Figure 2.1 (c) shows the GraphQ computational steps for answering the clique query. The three columns in the table show the partitions considered in the beginning of each iteration, the local maximal cliques identified, and the abstract edges selected by GraphQ to refine at the end of the iteration, respectively. Before iteration #0, the user selects all the three partitions via `initPartitions`. The vertex-centric computation of these partitions identifies four local cliques {1, 2, 3}, {4, 6}, {5}, and {7, 8, 9}.

Since the `check` function cannot find a clique whose size is ≥ 5, GraphQ ranks the four local cliques based on their sizes (by calling `refinePriority`) and invokes `refine` five times with the following clique pairs: ({1, 2, 3}, {7, 8, 9}), ({1, 2, 3}, {4, 6}), ({4, 6}, and {7, 8, 9}), ({5},{1, 2, 3}), ({5},{7, 8, 9}). For instance, for input ({1, 2, 3}, {7, 8, 9}), no abstract edge exists on the AG that connects any vertex in the first clique with any vertex in the second. Hence, `refine` returns an empty list.

For input ({1, 2, 3}, {4, 6}), however, GraphQ detects that there is an abstract edge between every abstract vertex that represents {1, 2, 3} and every abstract vertex that represents {4, 6}. The abstract edges connecting these two cliques (*i.e.*, a, b, and c) are then added into list `list` and returned.

After checking all pairs of cliques, GraphQ obtains 6 lists of abstract edges, among which five span two partitions and one spans three. Suppose K is 2 at this moment. The one spanning three partitions is discarded. For the remaining five lists, (a, b, c) is the first list

returned by `refine` (on input ({1, 2, 3}, {4, 6})). These three abstract edges are selected and their refinement adds the following four concrete (inter-partition) edges back to the graph: 4→2, 3→4, 1→5, and 2→6. The second iteration repeats vertex-centric computation by considering a merged partition {1, 2, 4, 5, 6}. When the partition is processed, a new clique {1, 2, 3, 4, 6} is found. Function `check` finds that the clique answers the query; so it reports the clique and terminates the process.

Programmability Discussions In addition to answering queries with user-specified goals, our programming model can also support aggregation queries (min, max, average, *etc.*). For example, to find the largest clique under a memory budget, only minor changes are needed to the `CliqueQuery` example. First, we can define a private field called `max` to the class. Second, we need to update the `check` method as follows:

```
if(c.size()>max)
  { max=c.size(); return false;}
```

The observation here is that `check` should always return **false**. GraphQ will continue the refinement until the (memory) budget runs out, and the result `c` aligns with our intuition of being "the largest `Clique` under the budget based on the user-specified refinement heuristics", a flavor of the budget-aware query processing.

GraphQ can also support *multiplicity* of results, such as the top 30 largest cliques. This is just a variation of the example above. Instead of reporting a clique `c`, the `CliqueQuery` should maintain a "top 30" list, and use it as the argument for `report`.

Trade-off Discussions It is clear that GraphQ provides several trade-offs that the user can explore to tune its performance. First, the memory size determines GraphQ's answerability. A higher budget (*i.e.* more memory) will lead to (1) finding more entities with higher goals, or (2) finding the same number of entities with the same goals more quickly. Since GraphQ can be embedded in a data analytical application running on a PC, imposing a memory budget allows the application to perform intelligent resource management between GraphQ

an other parts of the system, obtaining satisfiable query answers while preventing GraphQ from draining the memory.

Another tradeoff is defined by abstraction granularity, that is, the ratio between the size of the AG and the memory size. The larger this ratio is, the more precise guidance the AG provides. On the other hand, holding a very large AG in memory could hurt performance by eclipsing the memory that could have been allocated for data loading and processing. Hence, achieving good performance dictates finding the sweetspot.

2.2 Abstraction-Guided Query Answering

This section formally presents our core idea of applying abstracting refinement to graph processing. In particular, we rigorously define GraphQ's answerability.

Definition 2.2.1 (Graph Query). *A user query is a 5-tuple* (Δ, ϕ, π, $\diamond$, g) *that requests to find, in a directed graph* $G = (V_G, E_G)$, Δ *entities satisfying a pair of predicates* $\langle \phi, \pi \diamond g \rangle$. *Definition predicate* $\phi \in \Phi$ *is a logical formula* ($\mathbb{P}(G) \rightarrow \mathbb{B}$) *over the set of all G's subgraphs that defines an entity,* $\pi \in \Pi$ *is a quantitative function* ($\mathbb{P}(G) \rightarrow \mathbb{R}$) *over the set of subgraphs satisfying* ϕ, *measuring the entity's* size, *and* $\diamond$ *is a numerical comparison operator* (e.g., $\geq$ *or* $=$) *that compares the output of* π *with a user-specified* goal of the query $g \in \mathbb{R}$.

This definition is applicable to a wide variety of user queries. For example, for the clique query discussed in Section 2.1, ϕ is the following predicate on the vertices and edges of a subgraph $S \subseteq G$, defining a clique:

$\forall\ v_1, v_2 \in V_S\colon\ \exists e \in E_S\colon\ e = (v_1, v_2) \vee e = (v_2, v_1)$,

while π is a simple function that returns the number of vertices $|V_S|$ in the subgraph. $\diamond$ and g are $\geq$ and 5, respectively. From this point on, we will refer to ϕ and π as the *definition predicate* and the *size function*, respectively.

Definition 2.2.2 (Monotonicity of the Size Function). *A query* (Δ, ϕ, π, $\diamond$, g) *is GraphQ-answerable if* $\pi \in \Pi$ *is a* monotone *function with respect to operator* $\diamond$: $\forall S_1 \in \mathbb{P}(G), S_2 \in$

$\mathbb{P}(G) : S_2 \subseteq S_1 \wedge \phi(S_1) \wedge \phi(S_2) \implies \pi(S_1) \diamond \pi(S_2)$.

While the user can specify an arbitrary size function π or goal g, π has to be *monotone* in order for GraphQ to answer the query. More precisely, for any subgraphs S_1 and S_2 of the input graph G, if S_2 is a subgraph of S_1 and they both satisfy the definition predicate ϕ, the relationship between their sizes $\pi(S_1)$ and $\pi(S_2)$ is $\pi(S_1) \diamond \pi(S_2)$. For example, if S_2 is a clique with N vertices, and S_1 is a supergraph of S_2 and also a clique, S_1's size must be $\geq N$. Monotonicity of the size function implies that once GraphQ finds a solution that satisfies a query at a certain point, the solution will *always* satisfy the query because GraphQ will only find better solutions in the forward execution. It also matches well with the underlying vertex-centric computation model that gradually propagates the information of a vertex to distant vertices (*i.e.*, which has the same effect as considering increasingly large subgraphs).

Definition 2.2.3 (Partition). *A partition P of graph G is a subgraph (V_P, E_P) of G such that vertices in V_P have contiguous labels [i, i + $|V_P|$], where $i \in I$ is the minimum integer label a vertex in V_P has and $|V_P|$ is the number of vertices of P. A* partitioning *of G produces a set of partitions $P_1, P_2, \ldots P_k$ such that $\forall j \in [1, k-1] : max_{v \in V_{P_j}} label(v) + 1 = min_{v \in V_{P_{j+1}}} label(v)$. An edge $e = (v_1, v_2)$ is an* intra-partition *edge if v_1 and v_2 are in the same partition; otherwise, e is an* inter-partition *edge.*

Logically, each partition is defined by a label range, and physically, it is a disk file containing the edges whose targets fall into the range. The physical structure of a partition will be discussed in Section 2.3.

Definition 2.2.4 (Abstraction Graph). *An abstraction graph ($\bar{V}, \bar{E}, \alpha, \gamma$) summarizes a concrete graph (V, E) using abstraction relation α: $V \rightarrow \bar{V}$. The AG is a* sound *abstraction of the concrete graph if $\forall e = (v_1, v_2) \in E : \exists \bar{e} = (\bar{v_1}, \bar{v_2}) \in \bar{E} : \bar{v_1}, \bar{v_2} \in \bar{V} \wedge (v_1, \bar{v_1}) \in \alpha \wedge (v_2, \bar{v_2}) \in \alpha$. γ: $\bar{V} \rightarrow V$ is a* concretization relation *such that $(\bar{v}, v) \in \gamma$ iff $(v, \bar{v}) \in \alpha$.*

α and γ form a *monotone Galois connection* [55] between G and AG (which are both posets). There are multiple ways to define the abstraction function α. In GraphQ, α is defined based on an interval domain [55]. Specifically, each abstract vertex $\bar{v}$ has an associated

interval $[i,\ j]$; $(v, \bar{v}) \in \alpha$ iff *label*$(v) \in [i,\ j]$. The primary goal is to make concrete edges whose target vertices have contiguous labels stay together in a partition file. To concretize an abstract edge, GraphQ will only need *sequential accesses* to a partition file, thereby maximizing locality and refinement performance. Different abstract vertices have disjoint intervals. The length of the interval is determined by a user-specified percentage r and the maximum heap size M—the size of the AG cannot be greater than $r \times M$. The implementation details of the partitioning and the AG construction can be found in Section 2.3. Clearly, the AG constructed by the interval domain is a sound abstraction of the input graph.

Lemma 2.2.5 (Edge Feasibility). *If no abstract edge exists from $\bar{v_1}$ to $\bar{v_2}$ on the AG, there must not exist a concrete edge from v_1 to v_2 on the concrete graph such that $(v_1, \bar{v_1}) \in \alpha$ and $(v_2, \bar{v_2}) \in \alpha$.*

The lemma can be easily proved by contradiction. It enables GraphQ to inspect the AG first to quickly skip over infeasible solutions.

Definition 2.2.6 (Abstraction Refinement). *Given a subgraph $S = (V_s,\ E_s)$ of a concrete graph $G = (V,\ E)$ and its AG $= (\bar{V},\ \bar{E})$ of G, an abstraction refinement $\sqsubseteq$ on S selects a set of abstract edges $\bar{e} \in \bar{E}$ and adds into E_s all such concrete edges e that $e \in E \setminus E_s : (\bar{e}, e) \in \alpha$. An abstraction refinement of the form $S \sqsubseteq S'$ produces a new subgraph $S' = (V_s',\ E_s')$, such that $V_s = V_s'$ and $E_s \subseteq E_s'$. A refinement is an* effective *refinement if $E_s \subset E_s'$.*

The concretization function is used to obtain concrete edges for a selected abstract edge. After an effective refinement, the resulting graph S' becomes a (strict) supergraph of S, providing a larger scope for query answering.

Lemma 2.2.7 (Refinement Soundness). *An entity satisfying the predicates $(\phi,\ \pi \diamond g)$ found in a subgraph S is preserved by an abstraction refinement on S.*

The lemma shows an important property of our analysis. Since our goal is to find Δ entities, this property guarantees that the entities we find in previous iterations will stay as

we enlarge the scope. The lemma can be easily proved by considering Definition 2.2.2: since the size function π is monotone, if the predicate $\pi(S) \diamond g$ holds in subgraph S, the predicate $\pi(S') \diamond g$ must also hold in subgraph S' that is a strict supergraph of S. Because S' contains all vertices and edges of S, the fact the definition predicate ϕ holds on S implies that ϕ also holds on S' (*i.e.*, $\phi(S) \implies \phi(S')$).

Definition 2.2.8 (Essence of Query Answering). *Given an initial subgraph $S = (V, E_s)$ composed of a set $\mathbb{P}$ of disjoint partitions $((V_1, E_1), \ldots, (V_j, E_j))$ such that $V = V_1 \cup \ldots \cup V_j$ and $E_s = E_1 \cup \ldots \cup E_j$, as well as an $AG = (\bar{V}, \bar{E})$, answering a query $(\Delta, \phi, \pi, \diamond, g)$ aims to find a refinement chain $S \sqsubseteq^* S''$ such that there exist at least Δ distinct entities in S'', each of which satisfies both ϕ and $\pi \diamond g$.*

In the worst case, S'' becomes G and graph answering has (at least) the same cost as computing a whole-graph solution. Each refinement step bridges multiple partitions. Suppose we have a partition graph (PG) for G where each partition is a vertex. The refinement chain starts with a PG without edges (*i.e.*, each partition is a connected component), and gradually adds edges and reduces the number of connected components. Suppose PG_S is the PG for a subgraph S, ρ is a function that takes a PG as input and returns the maximum number of partitions in a connected component of the PG, and each initial partition has the (same) size η. We have the following definition:

Definition 2.2.9 (Budget-Aware Query Answering). *Answering a query under a memory budget M aims to find a refinement chain $S \sqsubseteq^* S''$ such that $\forall\ (S_1 \sqsubseteq S_2) \in \sqsubseteq^*$: $\eta \times \rho(PG_{S_2}) \leq M$.*

In other words, the number of (initial) partitions connected by each refinement step must not exceed a threshold t such that $t \times \eta \geq M$. Otherwise, the next iteration would not have enough memory to load and process these t partitions.

Theorem 2.2.10 (Soundness of Query Answering). *GraphQ either returns correct solutions or does not return any solution if the vertex-centric computation is correctly implemented.*

Limitations Despite its practical usefulness, GraphQ can only answer queries whose vertex update functions are monotonic, while many real-world problems may not conform to this property. For example, for machine learning algorithms that perform probability propagation on edges (*e.g.*, belief propagation and the coupled EM (CoEM)), the probability in a vertex may fluctuate during the computation, preventing the user from formulating a probability problem as GraphQ queries.

2.3 Design and Implementation

We have implemented GraphQ based on GraphChi [88], a high-performance single-machine graph processing system. GraphChi has both C++ and Java versions; GraphQ is implemented on top of its Java version to provide an easy way for the user to write UDFs. Our implementation has an approximate of 5K lines of code and is available for download on BitBucket. The pre-processing step splits the graph file into a set of small files with the same format, each representing a partition (*i.e.*, a vertex interval). We modify the *shard* construction algorithm in GraphChi to partition the graph. Similarly to a shard in [88], each partition file contains all in-edges of the vertices that logically belong to the partition; hence, edges stored in a partition file whose sources do not belong to the partition are inter-partition edges.

The AG is constructed when the graph is partitioned. To allow concrete edges (*i.e.*, lines in each text file) represented by the same abstract edge to be physically located together, we first sort all edges in a partition based on the labels of their source vertices — it moves together edges from contiguous vertices. Next, for each abstract vertex (*i.e.*, an interval), we sort edges that come from this interval based on the labels of their target vertices — now the concrete edges represented by the same abstract edge are restructured to stay in a contiguous block of the file. This is a very important handling and will allow efficient disk accesses, provided that large graph processing is often I/O dominated.

For example, for an abstract edge [40, 80] $\rightarrow$ [1024, 1268], its concrete edges are located

physically in the partition file containing the vertex range [1024, 1268]. The first sort moves all edges coming from [40, 80] together. However, among these edges, those going to [1024, 1268] and those not are mixed. The second sort moves them around based on their target vertices, and thus, edges going to contiguous vertices are stored contiguously. Although the interval length used in the abstraction is statically fixed (*i.e.*, defined as a user parameter), we do not allow an abstract vertex to represent concrete vertices spanning two partitions — we adjust the abstraction interval if the number of the last set of vertices in a partition is smaller than the fixed interval size.

Each abstract edge consists of the starting and ending positions of the concrete edges it represents (including the partition ID and the line offsets), as well as various summaries of these edges, such as the number of edges, and the minimum and maximum of their weights. The AG is saved as a disk file after the construction. It will be loaded into memory upon query answering. When an (initial or merged) partition is processed, we modify the parallel sliding window algorithm in GraphChi to load the entire partition into memory. In GraphChi, a *memory shard* is a partition being processed while *sliding shards* are partitions containing out-edges for the vertices in the memory shard. Since inter-partition edges are ignored, GraphQ eliminates sliding shards and treats each partition p as a memory shard. The number of random disk accesses at each step thus equals the number of initial partitions contained in p.

The loaded data may include both enabled and disabled edges; the disabled edges are ignored during processing. Initially, all inter-partition edges are disabled. Refining an abstract edge loads the partitions to be merged and enables the inter-partition edges it represents before starting the computation. We treat the refinement process as an *evolving graph*, and modify the incremental algorithm in GraphChi to only compute and propagate values from the newly added edges.

A user-specified ratio r is used to control the size of the AG. Ideally, we do not want the size of the AG to exceed $r\times$ the memory size. However, this makes it very difficult to select the interval size (*i.e.* abstraction granularity) before doing partitioning, because

Name	*Query GraphQ to Find*	*Init*	*RefinePriority*	*GraphChi Time*	*GraphQ Pre-proc. Time*
PageRank	Δ vertices whose pageranks are $\geq N$	none	X-percentages (↑)	1754, 2880 secs.	120+0, 200+0 secs.
Clique	Δ cliques whose sizes are $\geq N$	all	clique sizes (↑)	5.5, 50.2 hrs.	400+500, 800+1060 secs.
Community	Δ communities whose sizes are $\geq N$	all	community sizes (↑)	3.4, 6.4 hrs.	150+200, 300+400 sec.
Path	Δ paths whose lengths are $\leq N$	none	path lengths (↓)	?, ?	200+0, 400+0 secs.
Triangle	Δ vertices whose edge triangles are $\geq N$	all	triangle counts (↑)	1990, 3194 secs.	200+300, 400+600 secs.

Table 2.1: A summary of queries performed in the evaluation: reported are the names and forms of the queries, initial partition selection, priority of partition merging, whole-graph computation times in GraphChi for the uk-2005 and the twitter-2010 graphs, and the time for pre-processing them; ↑ (↓) means the higher (lower) the better; each pre-processing time has two components $a+b$, where a represents the time for partitioning and AG construction, and b represents the time for initial (local) computation; "?" means the whole-graph computation cannot finish in 48 hours.

the size of the AG is related to its number of edges and it is unclear how this number is related to the interval size before scanning the whole graph. To solve the problem, we use the following formula to calculate the interval size i: $i = \frac{size(G)}{M \times r}$, under a rough estimation that if the number of vertices is reduced by i times, the number of edges (and thus the size of the graph) is also reduced by i times. In practice, the size of the AG built using i is always close to $M \times r$, although it often exceeds the threshold.

2.4 Queries and Methodology

We have implemented UDFs for five common graph algorithms shown in Table 2.1. The pre-processing time is a one-time cost, which does not contribute to the actual query answering time. For PageRank and Path, GraphQ does not need to compute local results; what partitions to be merged can be determined simply based on the structure of each partition. We experimented GraphQ with a variety of graphs. This section reports our results with the two largest graphs, shown in Table 2.2. Since the focus of this work is *not* to improve the whole-graph computation, we have not run other distributed platforms.

Name	*Type*	$\|V\|$	$\|E\|$	*#IP*	*#MP*	δ
uk-2005 [38]	webgraph	39M	0.75B	50	30	10
twitter-2010 [87]	social network	42M	1.5B	100	50	10

Table 2.2: Our graph inputs: reported in each section are their names, types, numbers of vertices, numbers of edges, numbers of initial partitions (IP), numbers of maximum partitions allowed to be merged before out of budget (MP), and numbers of partitions increased at each step (δ, *cf.* line 13 in Figure 2.2).

PageRank Answering PageRank queries is based on the whole-graph PageRank algorithm used widely to rank pages in a webgraph. The algorithm is not strictly monotone, because vertices with few incoming edges would give out more than they gain in the beginning and thus their pageranks values would drop in the first few iterations. However, after a short "warm-up" phase, popular pages would soon get their values back and their pageranks would continue to grow until the convergence is reached. To get meaningful pagerank values to query upon, we focus on the top 100 vertices reported by GraphChi (among many million vertices in a graph). Their pageranks are very high and these vertices represent the pages that a user is interested in and wants to find from the graph.

Focusing on the most popular vertices also allows us to bypass the non-monotonic computation problem—since the goals are very high, it is only possible to answer a query during monotonic phase (after the non-monotonic warm-up finishes). The refinement logic we implemented favors the merging of partitions that can lead to a larger *X-percentage*. The X-percentage of a partition is defined as the percentage of the outgoing edges of the vertex with the largest degree that stay in the partition. It is a metric that measures the completeness of the edges for the most popular vertex in the partition. The higher the X-percentage is, the quicker it is for the pagerank computation to reach a high value and thus the easier for GraphQ to find popular vertices. PageRank does not need a local phase—from the AG, we directly identify a list of partitions whose merging may yield a large X-percentage.

Clique is based on the Maximal Clique algorithm that computes a maximal clique for each

vertex in the graph. Since the input is a directed graph, a set of vertices forms a clique if for each pair of vertices u and v, there are two edges between them going both directions. GraphChi does not support variable-size edge and vertex data, and hence, we used 10 as the upper-bound for the size of a clique we can find. In other words, we associated with each edge and vertex a 44-byte buffer (*i.e.*, 10 vertices take 40 bytes and used an additional 4-byte space in the beginning to save the actual length). Due to the large amount of data swapped between memory and disk, the whole-graph computation over twitter-2010 took more than 2 days.

Path is based on the SSSP algorithm and aims to find paths with acceptable length between a given source and destination. Similarly to Clique, we associated a (fixed-length) buffer with each edge/vertex to store the shortest path for the edge/vertex. Since none of our input graphs have edge weights, we assigned each edge a random weight between 1 and 5. However, the whole-graph computation could not finish processing these graphs in 2 days. To generate reasonable queries for GraphQ, we sampled each graph to get a smaller graph (that is 1/5 of the original graph) and ran the whole-graph SSSP algorithm to obtain the shortest paths between a specified vertex S (randomly chosen) and all other vertices in the sample graph. If there exists a path between S and another vertex v in the small graph, a path must also exist in the original graph. The SSSP computation over even the small graphs took a few hours.

Community is based on a community detection algorithm in which a vertex chooses the most frequent label of its neighbors as its own label. Triangle uses a triangle counting algorithm that counts the number of edge triangles incident to each vertex. This problem is used in social network analysis for analyzing the graph connectivity properties [155]. For both applications, we obtained their whole-graph solutions and focus on the 100 largest entities (*i.e.*, communities and vertices with most triangles). Community and Triangle favor the merging of partitions that can yield large communities and triangle counts, respectively.

2.5 Evaluation

Test Setup All experiments were performed on a normal PC with one Intel Core i5-3470 CPU (3.2GHz) and 10GB memory, running Ubuntu 12.04. The JVM used was the HotSpot Client VM (build 24.65-b04, mixed mode, sharing). Some of our results for GraphChi may look different from those reported in [88] due to different versions of GraphChi used as well as different hardware configurations. We have conducted three sets of experiments. First, we performed queries with various goals and Δ to understand the query processing capability of GraphQ. Second, we compared the query answering performance between GraphQ and *GraphChi-ET* (i.e., acronym for "GraphChi with early termination") — a modified version of GraphChi that terminates immediately when a query is answered. Third, we varied the abstraction granularity to understand the impact of abstraction refinement. The first and third sets of experiments ran GraphQ on the PC's embedded 500GB HDD to understand the query performance on a normal PC while a Samsung 850 250GB SSD was used for the second set to minimize the I/O costs, enabling a fair comparison with GraphChi-ET.

2.5.1 Query Efficiency

In this experiment, the numbers of initial partitions for the two graphs are shown in Table 2.2. The maximum heap size is 8GB, and the ratio between the AG size and the heap size is 25%. For the two graphs, the maximum number of partitions that can be merged before out of budget is 30 and 50. For each algorithm, GraphQ first performed local computation on initial partitions (as specified by the UDF `initPartitions`). Next, we generated queries whose goals were randomly chosen from different value intervals. Queries with easy goals/small Δ were asked earlier than those with more difficult goals/larger Δ, so that the computation results for earlier queries could serve a basis for later queries (*i.e.*, incremental computation). This explains why answering a difficult query is sometimes faster than answering an easy query (as shown later in this section).

PageRank To better understand the performance, we divided the top 100 vertices (with

the highest pagerank values from the whole-graph computation) into several intervals based on their pagerank values. Each interval is thus defined by a pair of lower- and upper-bound pageranks. We generated 20 queries for each interval, each requesting to find Δ vertices with the goal being a randomly generated value that falls into the interval. For each interval reported in Table 2.3, all 20 queries were successfully answered. The average running time for answering these queries over uk-2005 is shown in the *Time* sections.

Δ	(a) Top20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	56.1	20	5.6	10	3.0	10	4.3	10
2	32.2	20	5.0	10	5.1	10	6.6	10
4	120.0	20	27.0	10	19.2	10	21.6	10
8	350.1	30	182.9	30	54.3	20	41.9	20

Table 2.3: GraphQ performance for answering PageRank queries over uk-2005; each section shows the performance of answering queries on pagerank values that belong to an interval in the top 100 vertex list; reported in each section are the number of entities requested to find (Δ), the average query answering time in seconds (*Time*), and the number of partitions merged when a query is answered (*Par*).

The largest Δ we have tried is 8—GraphQ ran out of budget for most of the queries when a larger Δ was used. When $\Delta \leq 4$, GraphQ could successfully answer all queries even including those from the top 10 category. For twitter-2010, GraphQ always failed on queries whose goals were selected from the top 10 category. Otherwise, it successfully answered all queries. For example, the average time for answering 8 queries whose goals are from the top 10-20 category is 754.7 seconds.

Clique The biggest clique found in twitter-2010 (by the 52-hour whole-graph computation) has 6 vertices and there are totally 66 of them. The (relatively small) size of the maximum clique is expected because a clique in a directed graph has a stronger requirement: bi-directional edges must exist between each pair of vertices. The largest Δ we have tried is 64. Table 2.4 shows GraphQ's performance as we changed Δ; the running time reported is

Δ	(a) *Size = 6*		(b) *Size = 5*		(c) *Size = 4*		(d) *Size = 3*	
	Time	*Par*	*Time*	*Par*	*Time*	Par	*Time*	*Par*
1	98.3	10	2.0	10	2.0	10	2.0	10
2	248.1	10	2.0	10	2.3	10	2.0	10
4	489.5	20	2.1	10	2.0	10	8.3	10
8	823.9	20	51	10	2.1	10	8.2	10
16	5960.3	30	49.1	10	2.1	10	9.6	10
32	-	50	144.1	10	2.8	10	16.4	10
64	-	50	460.0	10	128.3	10	20.0	10

Table 2.4: GraphQ's performance for answering Clique queries over twitter-2010; a "-" sign means some queries in the group could not be answered.

the average time across answering 20 queries in each interval. GraphQ could easily find 8 of the 66 6-clique (in 823 seconds), but the time increased significantly when we asked for 16 of them. GraphQ could find no more than 26 6-cliques before running out of budget. If a user is willing to sacrifice her goal and look for smaller cliques (say 5-cliques), GraphQ can find 64 of them in 460 seconds (by merging only 10 partitions).

Community The whole-graph computation of community detection took 1.5 hours on uk-2005 and 6.4 hours on twitter-2010. Similarly to PageRank, we focused on the top 100 largest communities and asked GraphQ for communities of different sizes (that belong to different intervals on the top 100 list). For each interval, we picked 20 random sizes to run GraphQ and the average running time over uk-2005 is reported in Table 2.5. The whole-graph result shows that there are a few (less than 10) communities that are much larger than the other communities on the list. These communities have many millions of vertices and none of them could be found by GraphQ before the budget ran out. Hence, Table 2.5 does not include any measurement for queries with a size that belongs to the top 10 interval.

Interestingly, we found that GraphQ performed much better over twitter-2010 than uk-2005: for twitter-2010, GraphQ could easily find (in 162.1 seconds) 256 communities from the top 10-20 range by merging only 20 partitions as well as 1024 communities (in 188.2 seconds)

Δ	(a) Top10-20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	8.2	10	4.9	10	4.3	10	4.5	10
2	51.8	10	34.5	10	20.1	10	14.2	10
4	142.1	20	63.3	10	27.1	10	25.4	10
8	292.3	20	160.6	20	56.9	10	35.5	10
16	563.4	30	236.7	30	196.7	20	97.7	20
32	-	30	-	30	-	30	332.8	30

Table 2.5: GraphQ's performance for answering Community queries over uk-2005; each section reports the average time for finding communities whose sizes belong to different intervals in the top 100 community list.

from the top 20-40 range by merging 20 partitions. This is primarily because twitter-2010 is a social network graph in which communities are much "better defined" than a webgraph such as uk-2005.

Path We inspected the whole-graph solution for each sample graph (*cf.* Section 2.4) and found a set t of vertices v such that the shortest path on the small graph between S (the source) and each v is between 10 and 25 and contains at least 5 vertices. We randomly selected 20 vertices u from t and queried GraphQ for paths between S and u over the original graph. Based on the length of their shortest paths on the small graph, we used 10, 15, 20, and 25 as the goals to perform queries (recall that each edge has an artificial length between 1 and 5). The average time to answer these queries on twitter-2010 is reported in Table 2.6.

Δ	(a) 10		(b) 15		(c) 20		(d) 25	
	Time	*Par*	*Time*	*Par*	*Time*	*Par*	*Time*	*Par*
1	59.5	10	57.6	10	58.1	10	45.2	10
2	55.5	20	53.2	20	49.1	20	65.0	10
4	230.1	50	111.8	20	110.7	20	115.6	20

Table 2.6: GraphQ's performance for answering Path queries over twitter-2010.

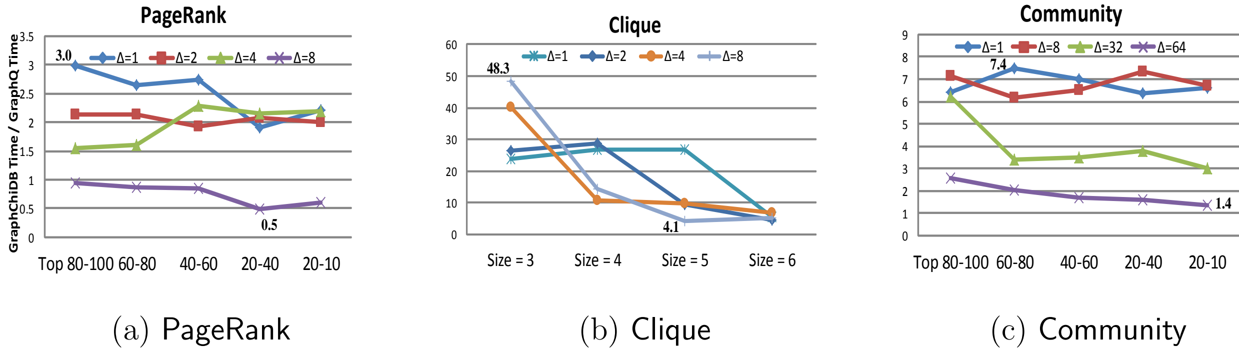
Our results for Path clearly demonstrate the benefit of GraphQ: it took the whole-graph computation 6.2 hours to process a graph only 1/5 as big as twitter-2010, while GraphQ can quickly find many paths of reasonable length in the original twitter graph.

Triangle A similar experiment was performed for Triangle (as shown in Table 2.7): we focused on the top 100 vertices with the largest numbers of edge triangles. GraphQ could find only two vertices when a value from the top 10 triangle count list was used as a query goal. However, if the goal is chosen from the top 10-20 interval, GraphQ can easily find 16 vertices (which obviously include some top 10 vertices). It is worth noting that GraphQ found these vertices by combining only 10 partitions. This is easy to understand—edge triangles are local to vertices; computing them does not need to propagate any value on the graph. Hence, vertices with large triangle counts can be easily found as long as (most of) their own edges are recovered.

Δ	(a) Top10-20		(b) 20-40		(c) 40-60		(d) 60-100	
	Time	*Par*	*Time*	*Par*	*Time*	Par	*Time*	*Par*
1	3.3	10	3.0	10	2.9	10	4.5	10
2	3.2	10	3.6	10	3.9	10	7.6	10
4	3.4	10	3.2	10	3.1	10	8.7	10
8	2.8	10	3.3	10	3.0	10	19.6	10
16	2.9	10	2.9	10	3.2	10	313.3	10

Table 2.7: GraphQ's performance for answering Triangle queries over uk-2005.

The measurements in Table 2.3–2.7 also demonstrate the impact of the budget. For twitter-2010, merging 50, 30, 20, and 10 partitions requires, roughly, 6GB, 3.6GB, 2.4GB, and 1.2GB of memory, while, for uk-2005, the amounts of memory needed to merge 30, 20, and 10 partitions are 5.5GB, 4GB, and 2GB, respectively. From these measurements, it is easy to see what queries can and cannot be answered given a memory budget.



(a) PageRank (b) Clique (c) Community

Figure 2.3: Ratios between the running times of GraphChi-ET and GraphQ over twitter-2010: (a) PageRank: Max = 3.0, Min = 0.5, GeoMean = 1.6; (b) Clique: Max = 48.3, Min = 4.1, GeoMean = 13.4; and (c) Community: Max = 7.5, Min = 1.4, GeoMean = 4.2.

2.5.2 Comparison to GraphChi-ET

GraphChi-ET is a modified version of GraphChi in which we developed a simple interface that allows the user to specify the Δ and goal for a query and then run GraphChi's whole-graph computation to answer the query – the computation is terminated immediately when the query is answered. Figures 2.3 shows performance comparisons between GraphQ and GraphChi-ET over twitter-2010 on three algorithms using SSD. A similar trend can also be observed on the other two algorithms.

Note that for PageRank, GraphQ outperforms GraphChi-ET in all cases except when $\Delta = 8$. In this case, GraphQ is about $2\times$ slower than GraphChi-ET because GraphQ needs to merge 50 partitions and is always close to running out of budget. The memory pressure is constantly high, making in-memory computation less efficient than GraphChi-ET's PSW algorithm. For all the other benchmarks, GraphQ runs much faster than GraphChi-ET. An extreme case is when $\Delta = 1$ for Clique, as shown in Figure 2.3 (b), GraphChi-ET found a 3-clique in 159.5 seconds, while GraphQ successfully answered the query only in 3.3 seconds. This improvement stems primarily from GraphQ's ability of prioritizing partitions and intelligently enlarging the processing scope.

Table 2.8 shows a detailed breakdown of running time on I/O and computation for

System	Time(s)	Comp.	Comp.Perc.	I/O	IO.Perc.
Q:PR	520.0	147.6	28.4%	372.4	71.6%
ET:PR	301.0	69.0	22.9%	232.0	77.1%
Q:Clique	637.0	548.5	86.1%	88.5	13.9%
ET:Clique	3208.0	2857.1	89.1%	351.0	10.9%
Q:Comm	81.5	25.6	31.4%	55.9	68.6%
ET:Comm	112.0	45.0	40.2%	68.0	60.7%

Table 2.8: A breakdown of time on computation and I/O for GraphQ and GraphChi-ET for PageRank, Clique, and Comm; measurements were obtained by running the most difficult queries from Figure 2.3.

answering the most difficult queries from Figure 2.3 (i.e., those represented by points at the bottom right corner of each plot). These queries have the longest running time, which enables an easier comparison. Clearly, GraphQ reduces both computation and I/O because it loads and processes fewer partitions. However, the percentages of I/O and computation in the total time of each query are roughly the same for GraphQ and GraphChi-ET.

2.5.3 Impact of Abstraction Refinement

To understand the impact of abstraction refinement, we varied the abstraction granularity by using 0.5GB, 1GB, and 2GB of the heap to store the AG. The numbers of abstract vertices for each partition corresponding to these sizes are a = 25, 50, and 100, respectively, for twitter-2010. We fixed the budget at 50 partitions (which consume 6GB memory), so that we could focus on how performance changes with the abstraction granularity.

Figure 2.4 compares performance under different abstraction granularity for Δ = 1, 4, and 8. While configuration $a = 100$ always yields the best performance, its running time is very close to that of $a = 50$. It is interesting to see that, in many cases (especially when $\Delta = 4$), $a = 25$ yields worse performance than random selection. We carefully inspected this AG and found that the abstraction level is so high that different abstract vertices have

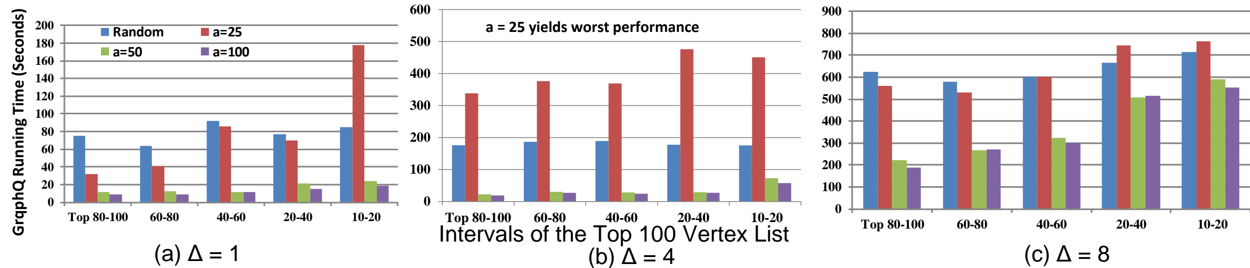


Figure 2.4: GraphQ's running time (in seconds) for answering PageRank queries over twitter-2010 using different abstraction graphs: *Random* means no refinement is used and partitions are merged randomly; $a = i$ means a partition is represented by i abstract vertices.

similar degrees. The X-percentages for different partitions computed based on the AG are also very similar, and hence, partitions are merged almost in a sequential manner (*e.g.*, partitions 1–10 are first merged, followed by 10–20, *etc.*). In this case, the random selection has a higher probability of finding the appropriate partitions to merge.

Despite its slow running time, random selection found all vertices requested by the queries. This is because, in the twitter graph, the edges of high-degree vertices are reasonably evenly distributed in different partitions of the graph. A similar observation was made for Triangle. But for the other three algorithms, their dependence on the AG is much stronger. For example, GraphQ could not answer any path query without the AG. As another example, no cliques larger than 3 could be found by using random selection.

2.6 Summary and Interpretation

To the best of our knowledge, our technique is the first to borrow the idea of abstraction refinement from program analysis and verification [54] to process graphs, resulting in a query system that can quickly find correct answers in partial graphs. While there exists a body of work on graph query systems and graph databases (such as GraphChi-DB [89], Neo4j[2], and Titan[3]), the refinement-based query answering in GraphQ provides several unique features

unseen in existing systems.

First, GraphQ reflects a ground-up redesign of graph processing systems in the era of "Big Data": unlike the predominant approach of graph querying where only simple graph analytics—those often involving SQL-like semantics where graph vertices/edges are filtered by meeting certain conditions or patterns [74, 79, 50], GraphQ has a strong and general notion of "answerability" which allows for a much wider range of analytical queries to be performed with flexible in-graph computation (*cf.* Section 2.2). Furthermore, the abstraction-guided search process makes it possible to answer a query by exploring the most relevant parts of the graph, while a graph database treats all vertices and edges uniformly and thus can be much less efficient.

Second, the idea of abstraction refinement in GraphQ provides a natural data organization and data movement strategy for designing efficient *out-of-core* Big Data systems. In particular, ignoring inter-partition edges (that are abstracted) allows GraphQ to load one partition at a time and perform vertex-centric computation on it independently of other partitions. The ability of exploring only a small fraction of the graph at a time enables GraphQ to answer queries over very large graphs *on one single PC*, thus in compliance with the recent trend that advocates single-machine-based Big Data processing [88, 126, 177, 89]. While our partitions are conceptually similar to GraphChi's shards (*cf.* Section 2.3), GraphQ does not need data from multiple partitions simultaneously, leading to significantly reduced random disk accesses compared to GraphChi's parallel sliding window (PSW) algorithm.

Third, GraphQ enjoys a strong notion of *budget awareness*: its query answering capability grows proportionally with the budget used to answer queries. As the refinement progresses, small partitions are merged into larger ones and it is getting increasingly difficult to load a partition into memory. Allowing a big partition to span between memory and disk is a natural choice (which is similar to GraphChi's PSW algorithm). However, continuing the search after the physical memory is exhausted will involve frequent disk I/O and significantly slow down query processing, rendering GraphQ's benefit less obvious compared to whole-graph computation. Hence, we treat the capacity of the main memory as a *budget* and

terminate GraphQ with an out-of-budget failure when a merged partition is too big to fit into memory. There are various trade-offs that can be explored by the user to tune GraphQ.

It is important to note that GraphQ is fundamentally different from approximate computing [32, 178, 49], which terminates the computation early to produce *approximate* answers that may contain errors. GraphQ always produces correct answers for the user-specified query goals, but improves the computation scalability and efficiency by finding a scope on the input graph that is sufficient to answer a query.

CHAPTER 3

Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code

Static analysis has been used to find bugs in systems software for more than a decade now [162, 154, 108, 66, 62, 56, 34, 35, 47, 61, 113, 44, 71, 18, 128]. Based on a set of systems rules, a static checker builds patterns and inspects code statements to perform "pattern matching". If a code region matches one of the patterns, a violation is found and reported. Static checkers have many advantages over recent, more advanced bug detectors based on SAT solvers or symbolic execution [44]: they are simple, easy to implement, and scalable. Furthermore, they produce deterministic and easy-to-understand bug reports compared to, for example, a symbolic execution technique, which often produces non-deterministic bug reports that are difficult to reason about [60].

Problems Unfortunately, the existing static checkers use many heuristics when searching for patterns, resulting in missing bugs and/or reporting false warnings. For example, Chou et al. [53] and Palix et al. [113] developed nine checkers to find bugs in the Linux kernel. Most of these checkers generate both false negatives and false positives. For instance, their **Null** checker tries to identify NULL pointer dereference bugs by inspecting only the functions that directly return NULL. However, a NULL value can be generated from the middle of a function and propagated a long way before it is dereferenced at a statement. Such NULL value propagation will be missed entirely by the **Null** checker.

As another example, the **Pnull** checker developed recently by Brown et al. [44] checks

Checker	Target Problems	Limitations	Potential Improvement with Interprocedural Analyses
Block	Deadlocks	Focus on "direct" invocations of the blocking functions (Negative)	Use a **pointer/alias analysis** to identify indirect invocations via function pointers of the blocking functions
Null	NULL pointer derefs	Inspect a closure of functions that return NULL explicitly (Negative)	Use a **dataflow analysis** to identify functions where NULL can be propagated to their return variables
Range	Use user data as array index without checks	Only check indices directly from user data (Negative)	Use a **dataflow analysis** to identify indices coming transitively from user data as well
Lock/Intr	Double acquired locks and disabled interrupts not appropriately restored	Identify lock/interrupt objects by var names (Negative)	Use a **pointer/alias analysis** to understand aliasing relationships among lock objects in different lock sites
Free	Use of a freed obj	Identify freed/used objects by var names (Negative)	Use a **pointer/alias analysis** to check if there is aliasing between objects freed and used afterwards
Size	Inconsistent sizes between an allocated obj and the type of the RHS var	Only check alloc sites (Negative)	Use a **pointer/alias analysis** to identify other vars that *point to* the same object with an inconsistent type
Pnull	NULL pointer derefs	Report all derefs post-dominated by NULL tests (Positive)	Use a **dataflow analysis** to filter out cases where the involved pointers *must not* be NULL

Table 3.1: A subset of checkers used by [44] and [113] to find bugs in the Linux kernel, their target problems, their limitations, the potential ways to improve them using a sophisticated interprocedural analysis; the first six have been used by Chou et al. [53] and Palix et al. [113] to study Linux bugs; the last one was described in a recent paper by Brown et al. [44] to find potential NULL pointer dereferences; positive/negative indicates whether the limitation can result in false positives/negatives.

whether a pointer dereference such as $a = b{-}{>}\ f$ is *post-dominated* by a NULL test on the pointer such as `if`(b). The heuristic here is that if the developer checks whether b can be NULL after dereferencing b, the dereferencing can potentially be on a NULL pointer. However, in many cases, the dereferencing occurs in one of the many control flow paths and in this particular path the pointer can never be NULL. The developer adds the NULL test simply because the NULL value may flow to the test point from a different control branch.

Our key observation in reducing the number of false positives and negatives reported by these checkers is to leverage *interprocedural analysis*. Among the aforementioned nine checkers, six that check flow properties can be easily improved (*e.g.*, producing fewer false positives and false negatives) using an interprocedural analysis, as shown in Table 3.1.

While using interprocedural analyses to improve bug detection appears to be obvious, there seems to be a large gap between the state of the art and the state of the practice. On the one hand, the past decade has seen a large number of sophisticated and powerful analyses developed by program analysis researchers. On the other hand, none of these techniques are widely used to find bugs in systems software.

We believe that the reason is two-fold. First, an interprocedural analysis is often not scalable enough to analyze large codebases such as the Linux kernel. In order for such an analysis to be useful, it often needs to be *context-sensitive*, that is, distinct solutions need to be produced and maintained for different calling contexts (*i.e.*, a chain of call sites representing a runtime call stack). However, the number of calling contexts grows *exponentially* with the size of the program and even a moderate-sized program can have as large as 10^{14} distinct contexts [156], making the analysis both compute- and memory-intensive. Furthermore, most interprocedural analyses are difficult to parallelize, because they frequently involve decision making based on information discovered *dynamically*. Thus, most of the existing implementations of such analyses are entirely sequential.

Second, the sheer implementation complexity scares practitioners away. Much of this complexity stems from optimizing the analysis rather than implementing the base algorithm. For example, in a widely-used Java pointer analysis [139], more than three quarters of the code performs approximations to make sure some results can be returned before a user-given time budget runs out. The base algorithm implementation takes a much smaller portion. This level of tuning complexity simply does not align with the "simplest-working-solution" [91] philosophy of systems builders.

Insight Our idea is inspired by the way a graph system enables scalable processing of large graphs. Graph system support pioneered by Pregel [102] provides a "one-stone-two-birds" solution, in which the optimization for scalability is mainly achieved by the (distributed or disk-based) system itself, requiring the developers to only write simple vertex programs using the interfaces provided by the system.

In this chapter, we demonstrate a similar "one-stone-two-birds" solution for interprocedural program analysis. Our key observation in this work is that many interprocedural analyses can be formulated as a *graph reachability* problem [119, 139, 117, 129, 165]. Pointer/alias analysis and dataflow analysis are two typical examples. In a pointer/alias analysis, if an object (*e.g.*, created by a malloc) can directly or transitively reach a variable on a directed graph representation of the program, the variable may point to the object. In a dataflow analysis that tracks NULL pointers, similarly, a transitive flow from a NULL value to a variable would make NULL propagate to the variable. Therefore, we turn the programs into graphs and treat the analyses as graph traversal. This approach opens up opportunities to leverage parallel graph processing systems to analyze large programs efficiently.

Existing Systems Several graph systems are available today. These systems are either distributed (*e.g.*, GraphLab [98], PowerGraph [67], or GraphX [68]) or single-machine-based (*e.g.*, GraphChi [88], XStream [126], or GridGraph [177]). Since program analysis is intended to assist developers to find bugs in their daily development tasks, their machines are the environments in which we would like our system to run, so that developers can check their code on a regular basis without needing to access a cluster. Hence, disk-based systems naturally become our choice.

We initially planned to use an existing system to analyze program graphs. We soon realized that a ground-up redesign (*i.e.*, from the programming model to the engine) is needed to build a system for analyzing large programs. The main reason is that the graph workload for interprocedural analyses is significantly different from a regular graph algorithm (such as PageRank) that iteratively performs computations on vertex values on a static graph. An interprocedural analysis, on the contrary, focuses on computing reachability by repeatedly adding *transitive edges*, rather than on updating vertex values. For instance, a pointer analysis needs to add an edge from each allocation vertex to each variable vertex that is transitively reachable from the allocation.

More specifically, many interprocedural analyses are essentially *dynamic reachability*

problems in the sense that the addition of a new edge is guided by a constraint on the labels of the existing edges. In a static analysis, the label of an edge often represents the semantics of the edge (*e.g.*, an assignment or a dereference). For two edges $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$, a transitive edge from a to c is added only if the concatenation of l_1 and l_2 forms a string of a (context-free) grammar.

This constraint-guided reachability problem, in general, requires *dynamic transitive closure* (DTC) computation [80, 164, 123], which has a wide range of applications in program analysis and other domains. The DTC computation dictates two important abilities of the graph system. First, at each vertex, all of its incoming and outgoing edges need to be visible to perform label matching and edge addition. In the above example, when b is processed, both $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$ need to be accessed to add the edge from a to c. This requirement immediately excludes edge-centric systems such as XStream [126] from our consideration, because these systems stream in edges in a random order and, thus, this pair of edges may not be simultaneously available.

Second, the system needs to support a large number of edges added dynamically. The added edges can be even more than the original edges in the graph. While vertex-centric systems such as GraphChi [88] support dynamic edge addition, this support is very limited. In the presence of a large number of added edges, it is critical that the system is able to (1) quickly check edge duplicates and (2) appropriately repartition the graph. Unfortunately, GraphChi supports neither of these features.

Our Contributions This chapter presents Graspan, the first *single machine, disk-based* parallel graph processing system tailored for interprocedural static analyses. Given a program graph and a grammar specification of an analysis, Graspan offers two major performance and scalability benefits: (1) the core computation of the analysis is automatically parallelized and (2) out-of-core support is exploited if the graph is too big to fit in memory. At the heart of Graspan is a parallel *edge-pair* (EP) centric computation model that, in each iteration, loads two partitions of edges into memory and "joins" their edge lists to produce

a new edge list. Whenever the size of a partition exceeds a threshold value, its edges are repartitioned. Graspan supports both in-memory (for small programs) and out-of-core (for large programs) computation. Joining of two edge lists is fully parallelized, allowing multiple transitive edges to be simultaneously added.

Graspan provides an intuitive programming model, in which the developer only needs to generate the graph and define the grammar that guides the edge addition, a task orders-of-magnitude easier than coming up with a well-tuned implementation of the analysis that would give trouble to skillful researchers for months.

We have implemented fully context-sensitive pointer/alias and dataflow analysis on Graspan. Context-sensitivity is achieved by making aggressive inlining [131]. That is, we clone the body of a function for every single context leading to the function. This approach is feasible only because the out-of-core support in Graspan frees us from worrying about additional memory usage incurred by inlining. We treat the functions in recursions *context insensitively* by merging the functions in each strongly connected component on the call graph into one function without cloning function bodies.

3.1 Background

While there are many types of interprocedural analyses, this chapter focuses on a pointer/alias analysis and a dataflow analysis, both of which are enablers for all other static analyses. This section discusses necessary background information on how pointer/alias analysis is formulated as graph reachability problems. Following Rep et al.'s interprocedural, finite, distributive, subset (IFDS) framework [117], we have also formulated a fully context-sensitive dataflow analysis as a grammar-guided reachability problem.

3.1.1 Graph Reachability

Pioneered by Reps et al. [117, 129], there is a large body of work on graph reachability based program analyses [84, 158, 160, 116, 36, 171, 169, 142]. The reachability computation is

often guided by a context-free grammar due to the *balanced parentheses* property in these analyses. At a high level, let us suppose each edge is labeled either an open parenthesis '(' or a close parenthesis ')'. A vertex is reachable from another vertex if and only if there exists a path between them, the string of labels on which has balanced '(' and ')'.

The parentheses '(' and ')' have different semantics for different analyses. For example, for a C pointer analysis, '(' represents an address-of operation & and ')' represents a dereference *. A pointer variable can point to an object if there is an assignment path between them that has balanced & and *. For instance, a string "&&**" has balanced parentheses while "&**&" does not. This balanced parentheses property can often be captured by a context-free grammar.

3.1.2 Pointer Analysis

A pointer analysis computes, for each pointer variable, a set of heap objects (represented by allocation sites) that can flow to the variable. This set of objects is referred to as the variable's *points-to* set. Alias information can be derived from this analysis — if the points-to sets of two variables have a non-empty intersection, they may alias.

Our graph formulation of pointer analysis is adapted from a previous formulation in [174]. This section briefly describes this formulation. The analysis we implement is *flow-insensitive* in the sense that we do not consider control flow in the program. Flow sensitivity can be easily added, but it does not contribute much to the analysis precision [76]. A program consists of a set of pointer assignments. Assignments can execute in any order, any number of times.

Pointer Analysis as Graph Reachability For simplicity of presentation, the discussion here focuses on four kinds of three-address statements (which are statements that have at most three operands):

Complicated statements are often broken down into these three-address statements in the

$a = b$	Value assignment	$a = *b$	Load
$*b = a$	Store	$a = \&b$	Address-of

compilation process by introducing temporary variables. Our analysis does not distinguish fields in a struct. That is, an expression $a\text{->} f$ is handled in the same way as $*a$, with offset f being ignored. As reported in [174], ignoring offsets only has little influence on the analysis precision, because most fields are of primitive types.

For each function, an *expression graph* – whose vertices represent C expressions and edges represent value flow between expressions — is generated; graphs for different functions are eventually connected to form a whole-program expression graph. Each vertex on the graph represents an expression, and each edge is of three kinds:

- *Dereference edge (D)*: for each dereference $*a$, there is a D-edge from a to $*a$; there is also an edge from an address-of expression $\&a$ to a because a is a dereference of $\&a$.
- *Assignment edge (A)*: for each assignment $a = b$, there is an A-edge from b to a; a and b can be arbitrary expressions.
- *Alloc edge (M)*: for each assignment $a = malloc()$, there is an M-edge from a special Alloc vertex to a.

Figure 3.1 shows a simple program and its expression graph. Each edge has a label, indicating its type. Solid and dashed edges are original edges in the graph and they are labeled M, A, or D, respectively. Dotted edges are transitive edges[1] added by Graspan into the graph, as discussed shortly.

[1] We use term "transitive edges" to refer to the edges dynamically added to represent non-terminals rather than the transitivity of a relation.

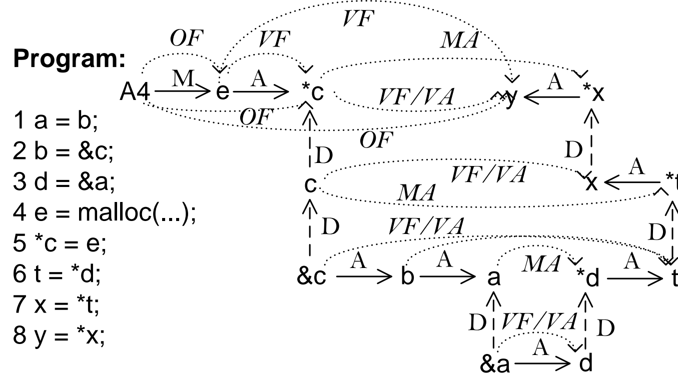


Figure 3.1: A program and its expression graph: solid, horizontal edges represent assignments (A- and M- edges); dashed, vertical edges represent dereferences (D-edge); dotted, horizontal edges represent transitive edges labeled non-terminals. *A4* indicates the allocation site at Line 4.

Context-free Grammar The pointer information computation is guided by the following grammar:

$$\begin{array}{llll} \text{Object flow:} & OF & ::= & M \quad VF \\ \text{Value flow:} & VF & ::= & (A \quad MA?)^{*} \\ \text{Memory alias:} & MA & ::= & \overline{D} \quad VA \quad D \\ \text{Value alias:} & VA & ::= & \overline{VF} \quad MA? \quad VF \end{array}$$

This grammar has four non-terminals *OF*, *VF*, *MA*, and *VA*. For a non-terminal T, a path in the graph is called a T-path if the sequence of the edge labels on the path is a string that can be reduced to T. In order for a variable v to point to an object o (*i.e.*, a malloc), there must exist an *OF* path in the expression graph from o to v. The definition of *OF* is straightforward: it must start with an alloc (M) edge, followed by a *VF* path that propagates the object address to a variable. A *VF* path is either a sequence of simple assignment (A) edges or a mix of assignments edges and *MA* (memory alias) paths.

There are two kinds of aliasing relationships in C: memory aliasing (MA) and value aliasing (VA). Two lvalue expressions are memory aliases if they may denote the same

memory location while they are value aliases if they may evaluate to the same value.

An *MA* path is represented by $\overline{D}$ *VA* D. Each edge has an inverse edge with a "bar" label. For example, for each edge $a \xrightarrow{D} b$, the edge $b \xrightarrow{\overline{D}} a$ exists automatically. $\overline{D}$ represents the inverse of a dereference and is essentially equivalent to an address-of. $\overline{D}$ *VA* D represents the fact that if (1) we take the address of a variable a and writes it into a variable b, (2) b is a *value alias of* another variable c, and (3) we perform dereferencing on c, the result is the same as the value in a.

A *VA* path is represented by $\overline{VF}$ *MA* *VF*. This has the meaning that if (1) two variables a and b are memory aliases, and (2) the values of a and b are propagated to two other variables c and d, respectively, through two *VF* paths, c and d contain the same pointer value. In other words, the path – c $\overline{VF}$ a *MA* b *VF* d – induces c *VA* d.

Note that *MA*, *VA*, and *VF* mutually refer each other. This definition captures the recursive nature of a flow or alias path. In this grammar, $\overline{D}$ and D are the open and close parentheses that need to be balanced.

Example In Figure 3.1, e points to *A4*, since the M edge between them forms an *OF* path. There is a *VF* path from $\&a$ to d, which is also a *VA* path (since *VA* includes *VF*). The *VA* path enables an *MA* path from a to $*d$ due to the balanced parentheses D and $\overline{D}$. This path then induces two additional *VF*/*VA* paths from b to t and from $\&c$ to t, which, in turn, contribute to the forming of the *VF*/*VA* path from c to x, making $*c$ and $*x$ memory aliases. Hence, there exists a *VF* path from e to y, which, together with the M edge at the beginning, forms an *OF* path from *A4* to y. This path indicates that y points to *A4*. The dotted edges in Figure 3.1 show these transitive edges.

Traditional Solution The traditional way to implement this analysis is to maintain a worklist, each element of which is a pair of a newly discovered vertex and a stack simulating a pushdown automaton. The implementation loops over the worklist, iteratively retrieving vertices and processing their edges. The traditional implementation does not add any phys-

ical edges into the graph (due to the fear of memory blowup), but instead, it tracks path information using pushdown automata. When a CFL-reachable vertex is detected, the vertex is pushed into the worklist together with the sequence of the labels on the path leading to the vertex. When the vertex is popped off of the list, the information regarding the reachability from the source to the vertex is discarded.

This traditional approach has at least two significant drawbacks. First, it does not scale well when the analysis becomes more sophisticated or the program to be analyzed becomes larger. For example, when the analysis is made *context-sensitive*, the grammar needs to be augmented with the parentheses representing method entries/exists; the checking of the balanced property for these parentheses also needs to performed. Since the number of distinct calling contexts can be very large for real-world programs, naïvely traversing all paths is guaranteed to be not scalable in practice. As a result, various abstractions and tradeoffs [139, 137, 82, 138] have been employed, attempting to improve scalability at the cost of precision as well as implementation straightforwardness.

Second, the worklist-based model is notoriously difficult to parallelize, making it hard to fully utilize modern computing resources. Even if multiple traversals can be launched simultaneously, since none of these traversals add transitive edges into the program graph as they are being detected, every traversal performs path discovery completely independently, resulting in a great deal of wasted efforts.

A "Big Data" Perspective Our key insight here is that adding *physical* transitive edges into the program graph makes it possible to devise a Big Data solution to this static analysis problem for two reasons. First, representing transitive edges *explicitly* rather than *implicitly* leads to addition of a great number of edges (*e.g.*, even larger than the number of edges in the original graph). This gives us a large (evolving) dataset to process. Second, the computation only needs to match the labels of consecutive edges with the productions in the grammar and is thus simple enough to be "systemized". Of course, dynamically adding many edges can make the computation quickly exhaust the main memory. However, this should not be

a concern, as there are already many systems [99, 88, 153, 125, 69, 148] built to process very large graphs (*e.g.*, the webgraph for the whole Internet).

3.2 Graspan's Programming Model

In this section, we describe Graspan's programming model, *i.e.*, the tasks that need to be done by the programmer to use Graspan. There are two main tasks. The first task is to modify a compiler frontend to generate the graph. The second task is to use the Graspan API to specify the grammar. Next, we will elaborate on these two tasks. We will then finish the section by discussing the applicability of Graspan's programming model to interprocedural analyses.

Generating Graph For Graspan to perform an interprocedural analysis, the user first needs to generate the *Graspan graph*, which is a specialized program graph tailored for the analysis, by modifying a compiler frontend. Note that since this task is relatively simple, the developer can generate the Graspan graph in a mechanical way without even thinking about performance and scalability. In this subsection, we briefly discuss how we generate the Graspan graph in the context of the pointer/alias analysis. We finish by generalizing graph generation for other interprocedural analyses.

For the pointer/alias analysis, we generate the Graspan graph by making two modifications to the program expression graph described in Section 3.1. These modifications include (1) inclusion of inverse edges and (2) context sensitivity achieved through inlining. For the former, we model inverse edges explicitly. That is, for each edge from a to b labeled X, we create and add to the graph an edge from b to a labeled $\overline{X}$.

For the latter, we perform a bottom-up (*i.e.*, reverse-topological) traversal of the call graph of the program to inline functions. For each function, we make a *clone* of its entire expression graph for each call site that invokes the function. Formal and actual parameters are connected explicitly with edges. The cloning of a graph not only copies the edges and

vertices in one function; it does so for *all* edges and vertices in its (direct and transitive) callees.

For recursive functions, we follow the standard treatment [156] – strongly connected components (SCC) are computed and then functions in each SCC are collapsed into one single function, and treated context insensitively. Clearly, the size of the graph grows exponentially as we make clones and the generated graph is often large. However, the out-of-core support in Graspan guarantees that Graspan can analyze even such large graphs effectively. For each copy of a vertex, we generate a unique ID in a way so that we can easily locate the variable its corresponds to and its containing function from the ID. In the Graspan graph, edges carry data (*i.e.*, their labels) but vertices do not. Finally, the graph is dumped to disk in the form of an edge list.

In general, the approach of aggressive inlining provides *complete information* that an analysis intends to uncover. Among all the existing analysis implementations, only Whaley et al. [156] could handle such aggressive inlining but they only clone variables (*not* objects) and have to use a binary decision diagram (BDD) to merge results. In addition, no evidence was shown that their analysis could process the Linux kernel. On the contrary, Graspan processes the exploded kernel graph in a few hours on a single machine.

Although this subsection focuses on the generation of pointer analysis graphs, graphs for other analyses can be generated in a similar manner. Here we briefly summarize the steps. First, vertices and edges need to be defined based on a grammar; this step is analysis-specific. Second, if inverse edges are needed in the grammar, they need to be explicitly added. Finally, context sensitivity can be generally achieved by function inlining. The developer can easily control the degree of context sensitivity by using different inlining criteria. For example, we perform *full context sensitivity* and thus our inlining goes from the bottom functions all the way up the top functions of the call graph. But if one wishes to perform only *one-level* context sensitivity, each function only needs to be inlined once.

Specifying Grammar Once the program graph is generated, the user needs to specify a grammar that guides the addition of transitive edges at run time. Unlike any traditional implementation of the analysis, Graspan adds transitive edges (*e.g.*, dotted edges in Figure 3.1) to the graph in a parallel manner. Specifically, for each production in the grammar, if Graspan finds a path whose edge labels match the RHS terms of the production, a transitive edge is added covering the path and labeled with the LHS of the production.

Since Graspan uses the edge-pair-centric model, it focuses on a pair of edges at a time, which requires each production in the grammar to have no more than two terms on its RHS. In other words, the length of a path Graspan checks at a time must be ≤ 2.

For example, the above mentioned pointer analysis grammar cannot be directly used, because the RHSes of *VF*, *MA*, and *VA* all have more than two terms. This means that to add a new *VF* edge, we may need to check more than two consecutive edges, which does not fit into Graspan's EP-centric model. Fortunately, every context free grammar can be *normalized* into an equivalent grammar with at most two terms on its RHS [117], similar to the Chomsky normal form. After normalization, our pointer analysis grammar becomes:

Object flow:	OF	::=	$M \;\; VF$
Temp:	T_1	::=	$A \mid MA$
Value flow:	VF	::=	$T_1 \mid VF \;\; T_1 \mid \epsilon$
Mem alias:	MA	::=	$T_2 \;\; D$
Temp:	T_2	::=	$\overline{D} \;\; VA$
Value alias:	VA	::=	$T_3 \;\; VF$
Temp:	T_3	::=	$\overline{VF} \;\; MA \mid \overline{VF}$

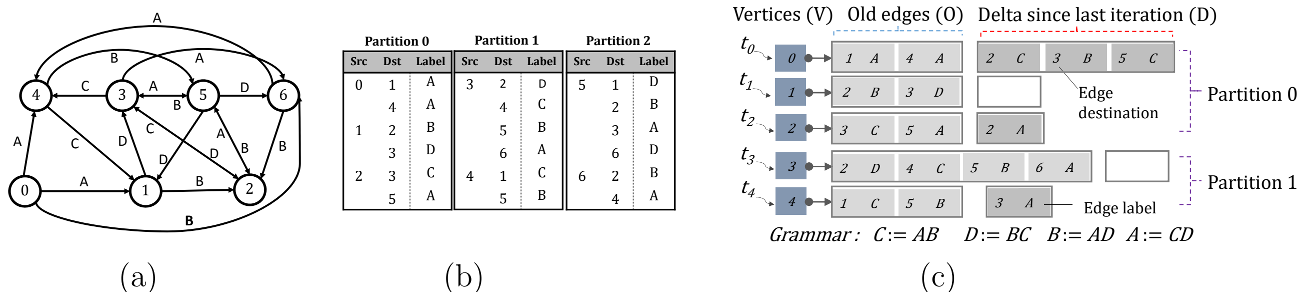
At the center of Graspan's programming model is an API `addConstraint(Label` *lhs*, `Label` *rhs1*, `Label` *rhs2*), which can be used by the developer to register each production in the grammar. *lhs* represents the LHS non-terminal while *rhs1* and *rhs2* represent the two RHS terms. If the RHS has only one term, *rhs2* should be NULL.

Graspan Applicability How many interprocedural analyses can be powered by Graspan? First, we note that pointer analysis and dataflow analysis are already representatives of a large number of analysis algorithms that can be formulated as a grammar-guided graph reachability problem. Second, work has been done to establish the convertibility from other types of analysis formulation (*e.g.*, set-constraint [84] and pushdown systems [22, 21]) to context-free language reachability. Analyses under these other formulations can all be parallelized and made scalable by Graspan.

Note that Graspan currently does not support analyses that require constraint solving, such as path-sensitive analysis and symbolic execution. In our future work, we plan to add support for constraint-based analyses by encoding constraints into edge values. Two edges match if a satisfiable solution can be found for the conjunction of the constraints they carry.

3.3 Graspan Design and Implementation

We implemented Graspan first in Java. Due to performance issues in the JVM (most of which were caused by the GC when copying arrays), we re-implemented Graspan in C++. The Java and C++ versions have an approximate 6K and 4K lines of code, respectively. Graspan can analyze programs written in any languages.



(a) (b) (c)

Figure 3.2: (a) An example graph, (b) its partitions, and (c) the in-memory representation of edge lists.

3.3.1 Preprocessing

Preprocessing partitions the Graspan graph generated for an analysis. The graph is in the edge-list format on disk. Similar to graph sharding in GraphChi [88], partitioning in Graspan is done by first dividing vertices into *logical intervals*. However, unlike GraphChi that groups edges based on their target vertices, one interval in Graspan defines a partition that contains edges whose *source vertices* fall into the interval. Edges are sorted on their source vertex IDs and those that have the same source are stored consecutively and *ordered on their target vertex IDs*. The fact that the outgoing edges for each vertex are sorted enables quick edge addition, as we will discuss shortly. Figure 3.2(a) shows a simple directed graph. Suppose Graspan splits its vertices into three intervals 0–2, 3–4, and 5–6; Figure 3.2(b) shows the partition layout.

When a new edge is found during processing, it is always added to the partition to which the source of the edge belongs. Graspan loads two partitions at a time and joins their edge-lists (Section 3.3.2), a process we refer to as a *superstep*. Given that only two partitions reside in memory at a given time, the size and hence the total number of partitions are determined automatically by the amount of memory available to Graspan.

Preprocessing also produces three pieces of meta-information: a *degree file* for each partition, which records the (incoming and outgoing) degrees of its vertices, a global *vertex-interval table* (VIT), which specifies vertex intervals, and a *destination distribution map* (DDM) for each partition p that maps, for each other partition q, the percentage of the edges in p that go into q. The DDM is essentially a matrix, each cell containing a percentage.

Graspan uses the degree file to calculate the size of the array to be created to load a partition. Without the degree information, a variable-size data structure (*e.g.*, ArrayList) has to be used, which would incur array resizing and data copying operations. The VIT records the lower and upper-bounds for each interval (*e.g.*, (0, [0, 10000]), (1, [10001, 23451]), *etc.*). Graspan maintains the table because the intervals will be redefined upon repartitioning. The DDM measures the "matching" degree between two partitions and will be used by the

Graspan scheduler to determine which two to load.

3.3.2 Edge-Pair Centric Computation

Graspan supports in-memory and out-of-core computations. For small graphs that can be held in memory, our preprocessing only generates two partitions, both of which are resident in memory. For large graphs with more than two partitions, Graspan uses a scheduling algorithm (discussed shortly) to load two partitions in each superstep, joins their edge lists, updates their edges, and performs repartitioning if necessary. Each superstep itself performs a fixed point computation — newly added edges may give rise to further edges.

The computation is finished when no new edges can be added. The updated edge lists may or may not be written back to disk depending on the next two partitions selected by the scheduler. This iterative process is repeated until a global fixed point is reached, that is, no new edges can be added for any pair of partitions. Since the VIT and the DDM are reasonably small in size, they are kept in memory throughout the processing.

In-Memory Edge Representation The edge list of a vertex v is represented as two arrays of (vertex, label) pairs, as shown in Figure 3.2(c). The first array (O_v) contains "old" edges that have been inspected before and the second (D_v) contains edges newly added in the current iteration. The goal is to avoid repeatedly matching edge pairs (discussed shortly).

Parallel Edge Addition Algorithm 1 shows a BSP-like algorithm for the parallel EP-centric computation. With two partitions p_1 and p_2 loaded, we first merge them into one single partition with combined edge lists (Line 1 – 2). Initially, for each vertex v, its two arrays O_v and D_v are set to empty list and the original edge list of v, respectively (Line 4 and Line 5). The loop between Line 6 and Line 24 creates a separate thread to process each vertex v and its edge list, computing transitive edges using a fixed-point computation with two main components.

The first component (Line 7 – 14) attempts to match each "old" edge in O_v that goes to

Algorithm 1: The parallel EP-centric computation.

Input: Partition p_1, Partition p_2

```
1  Combine the vertices of p1 and p2 into V
2  Combine the edge lists of p1 and p2 into E
3  for each edge list v : (e1, e2, ..., en) ∈ E do in parallel
4      Set Ov to ()
5      Set Dv to (e1, e2, ..., en)
6  for each vertex v : (Ov, Dv) whose Dv is NOT empty do in parallel
7      Array mergeResult ← ()
8      Let V1 be the intersection of the target vertices of Ov and V
9      /*Merge Ov with only Dv of other vertices*/
10     List listsToMerge ← {Ov}
11     foreach Vertex v' ∈ V1 do
12         Add Dv' into listsToMerge
13     /*Merge the sorted input lists into a new sorted list*/
14     mergeResult ← MatchAndMergeSortedArrays(listsToMerge)
15     /*Merge Dv with Ov ∪ Dv of other vertices*/
16     Let V2 be the intersection of the target vertices of Dv and V
17     listsToMerge ← {Dv, mergeResult}
18     foreach Vertex v' ∈ V2 do
19         Add Ov' and Dv' into listsToMerge
20     mergeResult ← MatchAndMergeSortedArrays(listsToMerge)
21     /*Update Ov and Dv*/
22     listsToMerge ← {Ov, Dv}
23     Ov ← MergeSortedArrays(listsToMerge)
24     Dv ← mergeResult − Ov
```

vertex u with each "new" edge of u in D_u. The second component (Line 15 – 20) matches each "new" edge in D_v with both "old" and "new" edges in O_u and D_u of vertex u. The idea is that we do not need to match an "old" edge of v with an "old" edge of u, because this work has been done in a previous iteration. O_v and D_v are updated at the end of each iteration.

An important question is how to perform edge matching. A straightforward approach is that, for each edge $v \xrightarrow{L_1} u$, we inspect each of u's outgoing edges $u \xrightarrow{L_2} x$, and add an edge $v \xrightarrow{K} x$ if a production $K ::= L_1\ L_2$ exists. However, this simple approach suffers from significant practical limitations. First, before the edge is added into v's list, we need to scan v's outgoing edges one more time to check if the same edge already exists. Checking and avoiding duplicates is very important – duplicates may cause the analysis either not to terminate or to suffer from significant redundancies in both time and space.

Doing a linear scan of the existing edges is expensive – it has an O($|E|^2$) complexity to add edges for each vertex, where $|E|$ is the total number of edges loaded. An alternative is to implement an "offline" checking mechanism that removes duplicates when writing updated partitions to disk. While this approach eliminates the cost of online checks, it may prevent the computation from terminating — if the same edge is repeatedly added, missing the online check would make the loop at Line 6 keep seeing new edges and run indefinitely.

Our algorithm performs *quick edge addition* and *online duplicate checks*. Our key insight is that edge addition can be done *in batch* much more efficiently than *individually*. To illustrate, consider Figure 3.2(a) where vertex 0 initially has two outgoing edges $0 \rightarrow 1$ and $0 \rightarrow 4$. Adding new edges for vertex 0 is essentially the same as *merging* the (outgoing) edges of vertex 1 and 4 into vertex 0's edge list and then filtering out those that have mismatched labels.

In Algorithm 1, to add edges for vertex v, we first compute set V_1 by intersecting the set of target vertices of the edges in O_v and the set V of all vertices in the loaded partitions (Line 8). V_1 thus contains the vertices whose edge lists need to be merged with v's edge list. If an out-neighbor of v is not in V, we skip it in the current superstep — this vertex will

be processed later in a future superstep in which its partition is loaded together with v's partition.

Next, we add O_v into a list *listsToMerge* together with D_u of each vertex u in V_1 (Line 10 – 12), and merge these lists into a new sorted list (Line 14). Since all input lists are already sorted, function MATCHANDMERGESORTEDARRAYS can be efficiently implemented by repeatedly finding the minimum (using an O($log|V|$) min-heap algorithm [27]) among the elements in a slice of the input lists and copying it into the output array. This whole algorithm has an O($|E|log|V|$) complexity, which is more efficient, both theoretically and empirically, than scanning edges individually (O($|E|^2$)) because $|V|$ is much smaller than $|E|$. Furthermore, edge duplicate checking can be automatically done during the merging — if multiple elements have the same minimum value, only one is copied into the output array. Label matching is performed before copying — an edge is not copied into the output if it has an inconsistent label.

Line 15 – 20 perform the same logic by computing a new set of vertices V_2, and merging D_v and all the edges (*i.e.*, $O_u \cup D_u$) of each vertex $u \in V_2$. At Line 20, all the new edges to be added to vertex v are in *mergeResult*. Finally, to prepare for the next iteration, O_v and D_v are merged (Line 23) to form the new O_v; D_v is then updated to contain the newly added edges (excluding those that already exist in O_v).

Example Figure 3.2(c) shows the in-memory edge lists at the end of the first iteration of the loop at Line 6 in Algorithm 1. In the next iteration, thread t_0 would merge O_0 with D_1 and D_4, and D_0 with $O_2 \cup D_2$, $O_3 \cup D_3$, and $O_5 \cup D_5$. O_0 and O_1 (and O_4) do not need to be merged again as this has been done before.

Another advantage of this algorithm is that it runs completely in parallel without needing any synchronization. While the edge list of a vertex may be *read* by different threads, edge addition can only be done by one single thread, that is, the one that processes the vertex.

3.3.3 Postprocessing

When a superstep is done, the updated edge lists need to be written back to their partition files. In addition, the degree file is updated with the new vertex degree information. The (in-memory) DDM needs to be updated with the new edge distribution information.

Repartitioning If the size of a partition exceeds a threshold (*e.g.*, a parameter), repartitioning occurs. It is easy for Graspan to repartition an oversized partition since the edge lists are sorted. Graspan breaks the original vertex interval into two small intervals, and edges are automatically restructured. The goal is to have the two small vertex intervals to have similar numbers of edges, so that the resulting partitions have similar sizes. The VIT needs to be updated with the new interval information. Repartitioning can also be triggered in the middle of a superstep if too many edges are added in the superstep and the size of the loaded partitions is close to the memory size.

Scheduling When a new superstep starts, two new partitions will be selected by the scheduler to join. Since a partition on which the computation was done in the previous superstep may be chosen again, Graspan delays the writing of a partition back to disk until the new partitions are chosen by the scheduler. If a chosen partition is already in memory, significant amounts of disk I/O can be saved.

We have developed a novel scheduling algorithm that has two objectives: (1) maximize the number of edge pairs that can potentially match and (2) favor the reuse of in-memory partitions. For (1), the scheduler consults the DDM. While the percentage information recorded in the DDM measures the matching opportunities between two partitions, it is an overall measurement that does not reflect the changes. Hence, we add another field to each cell of the DDM that records, for a pair of partitions p and q, the change in the percentage of the edges going from p into q since the last time p and q are both loaded. If p and q have never been loaded together, the change is the same as the full percentage.

Using $\delta(p, q)$ to denote this change, our scheduler selects a pair of partitions that have

the largest $\delta(p, q) + \delta(q, p)$ score. If multiple pairs of partitions have similar scores (*e.g.*, in a user-defined range), Graspan picks one that involves an in-memory partition. These delta fields in the DDM also determine the termination of the computation — for p and q whose $\delta(p, q) + \delta(q, p)$ is zero, no computation needs to be done on them. Graspan terminates when the delta field in every single cell of the DDM becomes 0.

Reporting Results Graspan provides an API for the user to iterate over edges with a certain label. For example, for the pointer analysis, edges with the *OF* label indicate a points-to solution, while edges with the *MA* and *VA* label represent aliasing variables. Graspan also provides translation APIs that can be used to map vertices and edges back to variables and statements in the program.

3.4 Evaluation

We built our frontend based on LLVM Clang. Our graph generators for the pointer/alias and dataflow analysis have 1.2K and 800 lines of C++ code, respectively. To use Graspan, the pointer/alias analysis has a grammar with 12 productions (*i.e.*, invoking the API function `addConstraint` 12 times) while the dataflow analysis has 2 productions. We first performed the pointer analysis. The dataflow analysis was designed specifically to track NULL value propagation. It was built based on the pointer analysis because it needs to query pointer analysis results when analyzing heap loads and stores.

We used the call graph generated by LLVM to perform inlining. Three large system programs were selected: the Linux kernel, the PostgreSQL database, and the httpd server. Their detailed statistics are reported in Table 3.2. Linux kernels are not directly compilable with LLVM. Thanks to the LLVMLinux project [9] that provides kernel patches for LLVM compilation, we were able to build the kernel version 4.4.0-rc5 (the latest version supported by LLVMLinux). Although we spent much effort trying to link as many modules as possible, some modules might still not be appropriately linked at the time of evaluation.

Program	Ver	#LoC	#Inlines
Linux kernel	4.4.0-rc5	16M	31.7M
PostgreSQL	8.3.9	700K	290820
Apache httpd	2.2.18	300K	58269

Table 3.2: Programs analyzed, their versions, numbers of lines of code, and numbers of function inlines.

For the other two systems, we picked their latest versions that could be successfully compiled by LLVM. **#Inlines** reports the total number of times functions are inlined – the larger this number, the more calling contexts a program has. For the Linux modules that were not linked appropriately, inlining only occurred inside.

Since our goal is to enable developers to use Graspan on development machines, we ran Graspan on a Dell desktop, with a quad-core 3.2GHZ Intel i5-4570 CPU, 8GB memory, and a 1TB SSD, running Linux 4.2.0. The size of the Java heap given to Graspan was 6GB. 8 threads were used when the EP-centric computation was performed.

Our evaluation focuses on the understanding of the following four research questions:

- Q1: Can the two analyses we implemented find new bugs in large-scale systems? (Section 3.4.1)
- Q2: How does Graspan perform in terms of time and space and how does it compare to existing graph systems? (Section 3.4.2)
- Q3: How do Graspan-based analysis implementations compare with other analysis implementations in terms of development effort and performance? (Section 3.4.3)
- Q4: How does Graspan compare with other backend systems when processing analysis workloads? (Section 3.4.4)

Since our analyses have already achieved the highest level of context sensitivity, we did not compare their precision with that of existing analyses. The main goal of this evaluation

is to (1) demonstrate the usefulness of these interprocedural analyses through the detection of new bugs, and (2) show the efficiency and scalability of Graspan when performing such expensive analyses that would be extremely difficult to make scalable otherwise.

Checker	**BL(4.4.0)**		**GR(4.4.0)**		**BL(2.6.1)**
	RE	FP	RE	FP	RE
Block	0	0	0	0	43
Null	20	20	**+108**	23	98
Free	14	14	+4	4	21
Range	1	1	0	0	11
Lock	15	15	+3	3	5
Size	25	23	+11	11	3
UNTest	N/A	N/A	**+1127**	0	N/A

Table 3.3: Checkers implemented, their numbers of bugs reported by the baseline checkers (BL), and *new bugs* reported by our Graspan analyses (GR) on top of the BL checkers on the Linux kernel 4.4.0-r5; **RE** shows total numbers of bugs reported while **FP** shows numbers of false positives determined manually; to provide a reference of how bugs evolve over the last decade, we include an additional section **BL(2.6.1)** with numbers of *true* bugs reported by the same checkers in 2011 on the kernel version 2.6.1 from [113]. UNTest is a new *interprocedural* checker we implemented to identify unnecessary NULL tests; '+' means new problems found.

3.4.1 Effectiveness of Interprocedural Analyses

To understand the effectiveness of our interprocedural analyses, we re-implemented the seven static checkers listed in Table 3.1 in Clang. We used these existing checkers as the baseline to understand whether the combination of interprocedural pointer/alias and dataflow analyses are able to improve them in finding new bugs or reducing false positives (as described in Table 3.1). Note that our interprocedural analyses are not limited to these checkers; they

can be used in a much broader context to find other types of bugs as well (*e.g.*, data races, deadlocks, *etc.*). We would also like to evaluate our analyses on commercial static checkers such as Coverity and GrammaTech. Unfortunately, we could not obtain a license that allows us to publish the comparisons, and hence, we had to develop these checkers from scratch.

We have added a new interprocedural checker UNTest that aims to find unnecessary, over-protective NULL tests – tests on pointers that must have non-NULL values – before dereferencing these pointers. Although these checks are not bugs, they create unnecessary code-level basic blocks that prevent compiler from performing many optimizations such as common sub-expression elimination or copy propagation, leading to performance degradation. Hence, these checks should be removed for compiler to fully optimize the program.

We manually checked *all bug reports* from both the baseline checkers and our analyses (except those reported by UNTest as described shortly) to determine whether a reported bug is a real bug. Since some of these checkers (such as Block, Range, and Lock) are specifically designed for Linux, Table 3.3 only reports information *w.r.t.* the Linux kernel. For checkers that check generic properties (*i.e.*, Null and UNTest), we have also run them on the two other programs; their results are described later in this section.

For the first six baseline checkers that found many real bugs in older versions of the kernel (used by [113] in 2011 to check Linux 2.6.x and by Chou et al. [53] in 2001 to check Linux 2.4.x), they could find only 2 real bugs in Linux 4.4.0-r5 (with the Size checker). This is not surprising because they were designed to target very simple bug patterns. Given that many static bug checkers have been developed in the past decade (including both commercial and open source), it is likely that most of these simple bugs have been fixed in this (relatively) new version of Linux. For example, the Null checker detected most of the bugs in [113] and [53]. In this current version, while it reported 20 potential bugs, a manual inspection confirmed that all of them were false positives.

Unnecessary NULL Tests We used our interprocedural analyses to identify NULL tests (*i.e.*, `if(p)`) in which the pointers checked *must not* be NULL. We have identified a total

```c
void*probe_kthread_data(
    task_struct *task){
 void *data = NULL;
 probe_kernel_read(&data);

 /*data will be
  dereferenced after
  return.*/
 return data;
}

long probe_kernel_read
 (void *dst){
 if(...)
   return -EFAULT;
 return
 __probe_kernel_read(dst);
}
```

(a) NULL deref in kernel/kthread.c

```c
#define page_private(page)
    ((page)->private)

bool swap_count_continued
 (...){
 head=vmalloc_to_page(...);
 if(page_private(head)
      != ...){
   ...
 }
}
page*vmalloc_to_page(...){
 page *page = NULL;
 if (!pgd_none(*pgd)){
  //...
 }
 return page;
}
```

(b) NULL deref in mm/swapfile.c

Figure 3.3: Two representative bugs in the Linux kernel 4.4.0-rc5 that were missed by the baseline checkers.

of 1127 unnecessary NULL tests in Linux, 149 in PostgreSQL, 32 in httpd. These are over-protective actions in coding, and may result in performance degradation. Because these warnings are too many to inspect manually, we took a sample of 100 warnings and found these tests are truly unnecessary. This is the *first time* that unnecessary NULL tests in the Linux kernel are identified and reported.

New Bugs Found Our analyses reported 108 new NULL pointer dereference bugs in Linux, among which 23 are false positives. All of these 85 new bugs involve complicated value propagation logic that cannot be detected by intraprocedural checkers. Figure 3.3 shows two example bugs.

In Figure 3.3 (a), function `probe_kthread_data` invokes `probe_kthread_read` to initialize pointer `data`. However, in `probe_kthread_read`, if a certain condition holds, an error code (`-EFAULT`) is returned and the pointer never gets initialized. Function `probe_kthread_data` then returns `data` directly without any check and the pointer gets dereferenced immediately

Modules	NULL pointer defs	Unnecessary NULL Tests
arch	0	75
crypto	0	15
init	0	1
kernel	4 (2)	65
mm	3 (0)	84
security	0	78
block	6 (2)	31
fs	19 (3)	84
ipc	0	17
lib	0	39
net	10 (8)	269
sound	15 (5)	83
drivers	25 (3)	286
Total	108 (23)	1127

Table 3.4: A breakdown of the new Linux bugs found by our analyses; in parentheses are numbers of false positives.

after the function returns to its caller. In Figure 3.3 (b), `page_private` may dereference a NULL pointer since function `vmalloc_to_page` may return NULL. This bug was missed by the baseline because the origin of the NULL value and the statement that dereferences it are in separate functions. These types of bugs can only be found by interprocedural analyses. In fact, we show these two bugs because they are relatively simple and easy to understand; most of our bugs involve more than 3 functions and more complicated logic.

For PostgreSQL and httpd, we detected 33 and 14 new NULL pointer bugs; our manual validation did not find any false positives among them.

Linux Bug Breakdown Table 3.4 lists the new bugs and NULL tests in Linux into modules. We make two observations on this breakdown. First, the code quality of the Linux kernel has been improved significantly over the past decade. Note that the bugs we found are all complicated bugs detected by our interprocedural analyses; the baseline checkers could not

find any (shallow) bug in this version of the kernel. Second, consistent with the observations made in both [53] and [113], drivers is still the directory that contains most (NULL Pointer) bugs. This is not surprising as drivers is still the largest module in the codebase. On the other hand, drivers is also the module of which developers are most cautious (perhaps due to the findings in [53] and [113]), demonstrated by the most unnecessary NULL tests it contains.

Prog	Pointer/Alias Analysis					Dataflow Analysis				
	IS=(E,V)	PS=(E,V)	PT	SS	T	IS=(E,V)	PS=(E,V)	PT	SS	T
Linux	(249.5M,52.9M)	(1.1B,52.9M)	91 secs	27	1.7 hrs	(69.4M, 63.0M)	(211.3M, 63.0M)	65 secs	33	11.9 hrs
PSQL	(25.0M,5.2M)	(862.2M,5.2M)	10 secs	16	6.0 hrs	(34.8M,29.0M)	(56.1M, 29.0M)	35 secs	16	2.4 hrs
httpd	(8.2M, 1.7M)	(904.3M, 1.7M)	3 secs	13	8.4 hrs	(10.0M, 5.3M)	(19.3M, 5.3M)	9 secs	16	11.4 mins

Table 3.5: Graspan performance: reported are the numbers of vertices and edges before (**IS**) and after (**PS**) being processed by Graspan, Graspan's pre-processing time (**PT**), numbers of supersteps taken (**#SS**), and total running time (**T**).

3.4.2 Graspan Performance

Table 3.5 reports various statistics of Graspan's executions (C++ version). Note that there is a large difference between the initial size and the post-processing size of each graph. For example, in Linux, the number of edges increases 3-5 times after the computation, while for httpd, the Graspan graph for pointer analysis increases more than 100 times. The computation time depends on both program characteristics and analysis type. For example, while the pointer analysis graph for `httpd` has a large number of edges added, its dataflow analysis graph does not change as much and thus Graspan finishes the computation quickly in 11.4 minutes. We found that this is because our dataflow analysis only tracks NULL values and in httpd the distances over which NULL can flow are often short.

We have also attempted to run these graphs *in memory* on the desktop we used and all of them except the dataflow analysis of httpd ran out of memory. While the initial size of each graph is relatively small, when edges are added dynamically, the graph soon becomes very big and Graspan needs to repartition it many times to prevent the computation from

running out of memory.

Analysis	Graspan		ODA [174]	SociaLite [90]
	CT	I/O		
Linux-P	99.7 mins	46.6 secs	OOM	OOM
Linux-D	713.8 mins	4.2 mins	-	OOM
PostgreSQL-P	353.1 mins	4.5 mins	> 1 day	OOM
PostgreSQL-D	143.8 mins	57.1 secs	-	OOM
httpd-P	497.9 mins	7.6 mins	> 1 day	> 1 day
httpd-D	11.3 mins	3.3 secs	-	4 hrs

Table 3.6: A comparison on the performance of Graspan, on-demand pointer analysis (ODA) [174] implemented in standard ways, as well as SociaLite [90] processing our program graphs in Datalog. The Graspan section shows a breakdown of the running times into computation time (**CT**), I/O time (**I/O**), and garbage collection time (**GC**); P and D represent pointer/alias analysis and dataflow analysis. OOM means out of memory.

The **Graspan** section of Table 3.6 reports the breakdown of Graspan's running time into computation and I/O (*i.e.*, disk writes/reads). Clearly, the EP-centric computation dominates the execution. While Graspan needs to perform many disk accesses, the I/O cost is generally low because most disk accesses are sequential accesses. Compared against the Java version of Graspan, its C++ version is 2 – 5 × faster due to (1) the elimination of garbage collection as well as (2) the increased memory packing factor and decreased I/O costs.

Figure 3.4 depicts the percentages of added edges across supersteps, measured as the number of added edges divided by the number of edges in each original graph. In general, an extremely large number of edges are added within the first 10 supersteps (*e.g.*, more than 500M for Linux), and as the computation progresses, fewer edges are added.

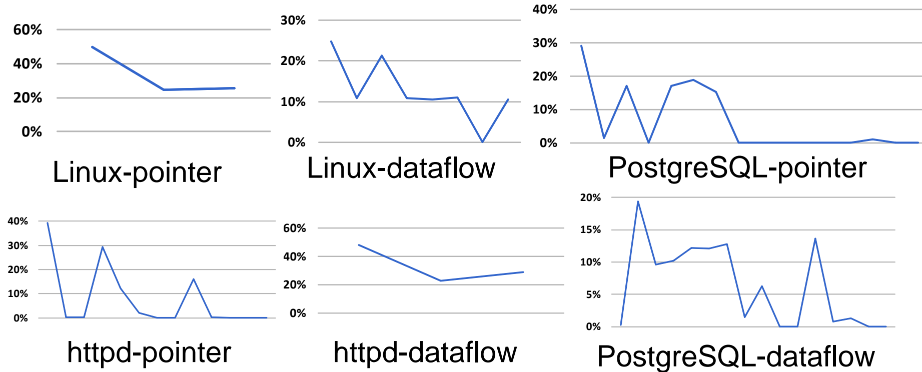


Figure 3.4: Percentages of added edges across supersteps.

3.4.3 Comparisons with Other Analysis Implementations

Data Structure Analysis [92] To understand whether Graspan-based analyses are more scalable and efficient than traditional analysis implementations, we wanted to compare our analyses with existing context-sensitive pointer/alias and dataflow analyses. While we had spent much time looking for publicly available implementations, we could not find anything available except the data-structure analysis (DSA) [92] in LLVM itself. DSA (implemented in 2007) is much more complicated than our pointer/alias analysis implementation — it has more than 10K lines of code while our pointer/alias analysis (*i.e.*, the graph generation part) only has 1.2K lines of code. According to a response from the LLVM mailing list [11], DSA was buggy and removed from LLVM since version 3.3. We tried to use LLVM 3.2 but it could not compile any version of the Linux kernel due to lack of patches.

On-demand Pointer Analysis [174] As no other implementations were available, we implemented the context-sensitive version of Zheng and Rugina's C pointer analysis [174] ourselves. We took the expression graph generated by our frontend and used a worklist-based algorithm to compute transitive closures, following closely the original algorithm described in [174]. The **ODA** section of Table 3.6 reports its performance. For all but httpd, ODA either ran out of memory or took a very long time (longer than one day) on the same desktop where we ran Graspan. For example, when processing Linux, it ran out of memory in 13

minutes. When we moved it onto a server with 32 2.60GHZ Xeon(R) processors and 32GB memory, it took this implementation 3.5 days to analyze Linux and it consumed 29GB out of the 32GB memory. On the contrary, Graspan finished processing Linux in a few hours with less than 6GB memory on the desktop with a much less powerful CPU.

3.4.4 Comparisons with Other Backend Engines

Datalog Since Datalog has been used to power static analyses, it is important to understand the pros/cons of using Graspan v.s. a Datalog engine as the analysis backend. While there are many Datalog engines available [90, 10, 151, 133], SociaLite [90] and LogicBlox [10] are designed for shared-memory machines while others [151, 133] are distributed engines running on large clusters. Since a distributed engine is not a choice for code checking in daily development tasks, we focused our comparison against shared-memory engines. LogicBlox is a commercial tool that has been previously used to power the Doop pointer analysis framework [41] for Java. However, it was the same licensing issue that prevented us from publishing comparison results with LogicBlox. Hence, this subsection only compares Graspan with SociaLite, a Datalog engine developed by Stanford that has been demonstrated to outperform other shared-memory engines.

The **SociaLite** section of Table 3.6 reports SociaLite's performance on the same desktop. SociaLite programs were easy to write — it took us less than 50 LoC to implement either analysis. However, SociaLite clearly could not scale to graphs that cannot fit into memory. For both pointer/alias and dataflow analysis, it ran out of memory for Linux and PostgreSQL. For httpd, although SociaLite processed the graphs successfully, it was much slower than Graspan.

GraphChi To understand whether other graph systems can efficiently process the same (program analysis) workload, we ran GraphChi — a disk-based graph processing system — because GraphChi is the only available system that supports both out-of-core computation and dynamic edge addition. GraphChi provides an API `add_edge` for the developer to add an

edge; it maintains a buffer for newly added edges during computation and uses a threshold to prevent the buffer from growing aggressively. When the size of the buffer exceeds the threshold, the edge adding thread goes to sleep and the function always returns false. The thread periodically wakes up and checks whether the main data processing thread comes to the commit point, at which the edges in the buffer can be flushed out to disk. GraphChi does not check edge duplicates and thus its computation would never terminate on our workloads. We added a naïve support that checks, before an edge is added, whether the same edge exists in the buffer. Note that this support does not solve the entire problem because it only checks the buffer but duplicates may have been flushed to shards. Checking duplicates in shards would require a re-design of the whole system.

We ran GraphChi on the same desktop to process the Linux dataflow graph. GraphChi ran into assertion failures in 133 seconds with around 65M edges added. This is primarily because GraphChi was not designed for the program analysis workload that needs to add an extremely large number of edges (with many duplicates) dynamically.

3.5 Summary and Interpretation

In this chapter, we revisit the scalability problem of interprocedural static analysis from a "Big Data" perspective. That is, we turn sophisticated code analysis into *Big Data analytics* and leverage novel data processing techniques to solve this traditional programming language problem. We develop *Graspan*, a disk-based parallel graph system that uses an *edge-pair* centric computation model to compute *dynamic transitive closures* on very large program graphs.

We implement *context-sensitive* pointer/alias and dataflow analyses on Graspan. An evaluation of these analyses on large codebases such as Linux shows that their Graspan implementations scale to millions of lines of code and are much simpler than their original implementations. Moreover, we show that these analyses can be used to augment the existing checkers.

Graspan is the *first attempt* to turn sophisticated code analysis into scalable Big Data analytics, opening up a new direction for scaling various sophisticated static program analyses (*e.g.*, symbolic execution, theorem proving, *etc.*) to large systems.

CHAPTER 4

RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine

There are two major types of analytical problems over large graphs: *graph computation* and *graph mining*. Graph computation includes a set of problems that can be represented through linear algebra over an adjacency matrix based representation of the graph. As a typical example of graph computation, PageRank [112] can be modeled as iterative sparse matrix and vector multiplications. Due to their importance in information retrieval and machine learning, graph computation problems have been extensively studied in the past decade; practical solutions have been implemented in a wide variety of graph systems [68, 52, 67, 72, 98, 88, 135, 105, 177, 126, 153, 173, 73, 125, 147, 176], most of which follow the "think like a vertex" programming paradigm pioneered by Pregel [102]. These systems have been highly optimized for locality, partitioning, and communication in order to deliver efficiency and scalability for processing very large graphs.

While this programming model makes it easy for developing computation algorithms, it is *not* designed for mining algorithms that aim to discover complex *structural patterns* of a graph rather than perform value computations. Fitting such algorithms into this model requires significant reformulation. For many mining tasks such as frequent subgraph mining (FSM), their patterns are not known *a priori*; hence, it is impossible to express these tasks using a vertex-centric model.

There is a body of work that uses declarative models to solve mining problems. Representative examples are Datalog [10, 90, 151, 134, 133], Arabesque [144], ScaleMine [13], or DistGraph [141]. For instance, due to its support for relational algebra, Datalog provides

simple interfaces for developing mining tasks [90, 133]. A Datalog program for Triangle Counting, for example, needs only the following two lines of code, with R representing the relation of edges and U representing a new relation of triangles:

```
U(a,b,c) <- R(a,b), R(b,c), R(a,c)
count U(a,b,c)
```

However, Datalog's support for graph mining is rather limited since the declarative nature of its programming model dictates that only mining algorithms whose patterns are known *a priori* can be expressed by Datalog. Arabesque is a Giraph-based graph mining system that presents developers a view of "embeddings". Embeddings are subgraphs that developers can easily check to find structural patterns. Using a *filter-process* programming model, Arabesque provides full support for developing a broad set of mining algorithms. For example, Arabesque enumerates all possible subgraphs and invokes the user-defined `filter` function on each subgraph. The user logic in the function determines whether the given subgraph is an instance of the specified motif (for motif counting) or turns the subgraph into a canonical form to count the number of instances of the form (for FSM).

Specialized systems have been developed for FSM due to its broad applications. Examples are ScaleMine [13] and DistGraph [141], but these systems do not work for other mining algorithms such as Triangle Counting or Cliques.

Problems with State-of-the-Art Systems Mining workloads are memory-intensive. Even simple mining algorithms can generate an enormous amount of intermediate data, which cannot fit into the main memory of any single machine. Early single-machine techniques such as gSpan [161] and GraMi [59] can analyze only small graphs as they are fundamentally limited by the size of the main memory of the machine on which they run. Recent mining tools such as Arabesque [144], ScaleMine [13], and DistGraph [141] are distributed systems — they leverage distributed memory resources to store intermediate mining data.

Distributed mining systems have several drawbacks that significantly impact their prac-

ticality. First, they commonly suffer from large startup and communication overhead. For small graphs, it is difficult for the startup/communication overhead to get amortized over the processing. For example, when FSM was executed on Arabesque to process a small graph (CiteSeer, with 4K edges) on a 10-node cluster, it took Arabesque 35 seconds to boost the system and load the graph, while executing the algorithm itself only took 3 seconds.

Second, in order to scale to large graphs, mining systems often need enterprise clusters with large amounts of memory. This is because the amount of intermediate data for a typical mining algorithm grows exponentially with the size of the graph. For example, built on top of MPI, a recent mining system DistGraph [141], using 128 IBM BlueGene/Q compute nodes, could only run 3-FSM with support = 25000[1] on a million-edge graph — even on such a small graph, the computation requires a total of $128 \times 256 =$ 32,768GB memory. Obviously, not all users have access to such enterprise clusters. Even if they do, running a simple mining algorithm on a relatively small graph does not seem to justify very well the cost of blocking hundreds or even thousands of machines for several hours.

When many compute nodes are employed primarily to offer memory, their CPU resources are often underutilized. Unlike the "think-like-a-vertex" computation algorithms that are amenable to the bulk synchronous parallel (BSP) model, mining workloads are not massively parallel by nature — a mining algorithm enumerates subgraphs of increasing sizes to find those that match a pattern; finer-grained partitioning of the input graph to exploit parallelism often does not scale well with increased CPU resources because subgraphs often cross partitions, creating great numbers of dependencies between tasks.

Load balancing in a distributed mining system is another major challenge. Algorithms such as FSM have dynamic working sets. Their search space is often unknown in advance and it is thus hard to partition the graph and distribute the workload appropriately before the execution. When we executed FSM on DistGraph, we observed that some nodes had high memory pressure and ran out of memory in several minutes while the memory usage of

[1]25000 is a very large frequency threshold for FSM — a subgraph is considered frequent only if its frequency exceeds this threshold. The smaller the support is, the more computation is needed.

some other nodes was below 10%.

The major problem of dataflow systems or Datalog engines is that they do not have a programming model flexible enough for expressing complex graph mining algorithms. For example, for mining frequent subgraphs whose structures have to be dynamically discovered, none of the Datalog systems can directly support it.

A possible way to develop a more cost-effective graph mining system is to add simple support for data spilling in an existing system (such as Arabesque or DistGraph) rather than developing a new system from scratch — if intermediate data can be swapped between memory and disk, the amount of compute resources needed may be significantly reduced. In fact, data spilling is already implemented in many existing systems: Arabesque is based on Giraph, which places on disk partitions that do not fit in memory; BigDatalog is based on Spark, which spills data throughout the execution. However, generic data spilling does not work well due to the lack of semantic information of how each data partition is used in the program.

To understand whether semantics-agnostic data spilling is effective, we ran transitive closure computation on BigDatalog over the MiCo graph [59] (with 1.1M edges) using a cluster of 10 nodes each with 32GB memory. Despite Spark's disk support, which spilled a total of 6.006GB of data to disk across all executors, BigDatalog still crashed in 1375 seconds.

Challenges and Contributions To address the shortcomings of the existing mining tools, we developed RStream, the *first disk-based, out-of-core* system that supports efficient mining of large graphs. Our key insight is consistent with the recent trend on building single-machine graph computation systems [88, 126, 153, 148, 101, 173, 17, 168] — given the increasing accessibility of high-volume SSDs, a disk-based system can satisfy the large storage requirement of mining algorithms by utilizing disk space available in modern machines; yet it does not suffer from any startup and communication inefficiencies that are inherent in distributed computing.

Building RStream has two major challenges. *The first challenge* is how to provide a programming interface rich enough to support a wide variety of mining algorithms. The design of RStream's programming model is inspired from both Datalog and the gather-apply-scatter (GAS) model used widely in the existing computation systems [67, 88, 126]. On the one hand, the relational operations in Datalog enable the composition of structures of smaller sizes into a structure of a large size, making it straightforward for the developer to program mining algorithms. On the other hand, GAS is a powerful programming model that supports iterative graph processing with a well-defined termination semantics. To enable easy programming of mining algorithms with and without statically-known structural patterns, we propose a novel programming model (Section 4.2), referred to as *GRAS*, which adds relational algebra into GAS. We show, with several examples, that under GRAS, many mining algorithms, including FSM, Triangle and Motif Counting, or Clique, can all be easily developed with less than 80 lines of code.

The second challenge is how to implement relational operators (especially join) efficiently for graphs. Since join is expensive, its efficiency is critical to the system performance. Instead of treating edges and vertices generically as relational tables as in Datalog, we take inspirations from graph computation systems to leverage the domain knowledge in graphs. In particular, we are inspired by recent systems (*e.g.* X-Stream [126] and GridGraph [177]) that use streaming to reduce I/O costs.

The scatter/gather phase in these systems loads vertices into memory and *streams in* edges/updates to generate updates/new vertex values. The insight behind streaming is that since the number of edges/updates is much larger than the number of vertices for a graph, edge streaming provides efficiency by sequentially accessing edge data from disk (as edges are sequentially read but not stored in memory) and randomly accessing vertex data held in memory. Streaming essentially provides an *efficient, locality-aware join implementation.* RStream leverages this insight (Section 4.3) to implement relational operations.

4.1 Background and Overview

Since RStream builds on streaming, we provide a brief discussion of this idea and the related systems. We then use a concrete example to overview RStream's design.

4.1.1 Background

RStream's tuple streaming idea is inspired by a number of prior works, and in particular, the X-Stream graph computation system [126] that uses edge streaming to reduce I/O. X-Stream partitions a graph into *streaming partitions* based on vertex intervals. Each streaming partition consists of (1) a vertex set, which contains vertices in a logical interval and their values, (2) an edge set, containing edges whose *source vertices* are in its vertex set, as well as (3) an update set, containing updates over the edges whose *destinations* are in its vertex set. X-Stream's design is based on the GAS model. It first conducts the scatter phase, which, for each partition, loads its vertex set into memory and streams in edges from the edge set to generate updates (*i.e.* propagate the value of the source to the destination for each edge).

The update over each edge is shuffled into the update set of the partition containing the destination of the edge. This enables an important *locality property* — for each vertex in a streaming partition, updates from all of its incoming edges are present in the update set of the same partition. The property leads to an efficient gather-apply phase, because vertex computation can be performed *locally* in each partition without accessing other partitions.

The following gather-apply phase loads the vertex set for each partition into memory, streams in updates from the update set of the partition, and invokes the user vertex function to compute a new value for each vertex. During scatter and gather-apply, edges/updates are streamed in *sequentially* from disk while in-memory vertices are randomly accessed to compute vertex values. This design leads to high performance because the number of edges is much larger than that of vertices.

4.1.2 RStream Overview

We use X-Stream's partitioning technique as the starting point to build RStream. RStream adds a number of relational (R) phases into the GAS programming/execution model, resulting in a new model referred to as *GRAS* in this chapter. To accommodate the relational semantics, RStream's programming interface treats vertex set, edge set, and update set all as relational tables. From this point on, we use *vertex table*, *edge table*, and *update table* to refer to these sets.

Since edges do not carry data, the edge table has a fixed schema of two columns (source and destination) – its numbers of rows and columns never change. Both the vertex and update table may change their schema during computation. For example, the vertex table, initially with two columns (ID and initial value), may grow to have multiple columns (due to joins) where each vertex corresponds to a row with multiple elements; an example can be found shortly in Figure 4.2. In the update table, one vertex may have multiple corresponding rows since the vertex can receive values from multiple edges. The update table can also change due to joins. Tuples in these tables remain *unsorted* throughout the execution.

RStream first conducts scatter to generate the update table. Similarly to X-Stream, the vertex table is loaded into memory in this phase; edges are streamed in and updates are shuffled. The user-defined relational phases are then performed over the update table and the edge table in each streaming partition. What and how many relational phases are needed is programmable. These relational phases produce a new set of update tables, which will be fed as input to the gather-apply phase to compute new tuples for each vertex. The new tuples are saved into the vertex table at the end of an iteration.

Example We use Triangle Counting as an example. Although Triangle Counting is also supported by many computation systems, it is a typical structure mining algorithm that has a simple logic and thus provides a good introductory example. Figure 4.1 depicts the dataflow of the computation while the RStream code is shown in Figure 4.2. The execution contains three phases: scatter and two additional relational phases. The scatter phase has the same

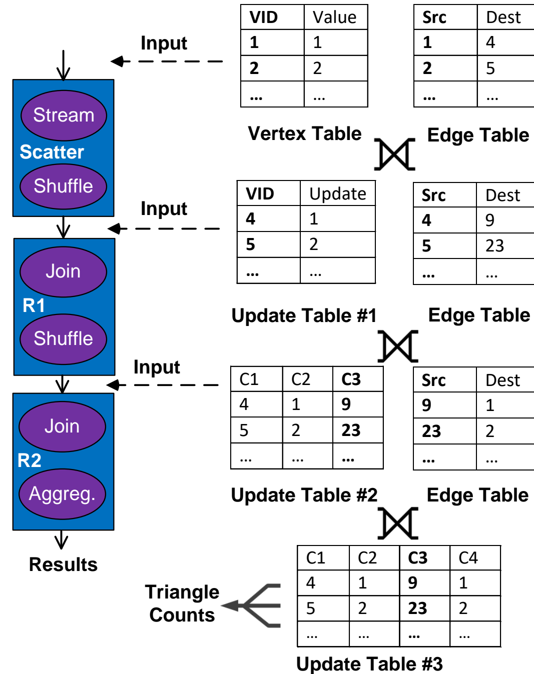


Figure 4.1: A Triangle Counting example in RStream; highlighted in each table is its key column. For each table, only a small number of relevant tuples are shown.

semantics as in X-Stream — the vertex table is loaded into memory; edges are streamed in and updates are shuffled. The relational phases are user-defined and their implementations are shown in Line 13–49. RStream lets the developer register the dataflow by connecting phases (Line 4–8). Each node on the dataflow graph is a `Phase` object. Class `TCScatter` is a scatter phase with a standard semantics. The developer adds relational phases into the dataflow.

Initially, we let the value of each vertex be its own ID (shown in the vertex table in Figure 4.1). The scatter phase streams edges in from the edge table. For each edge e, RStream retrieves the tuple from the vertex table corresponding to e's source vertex and produces an update based on it. In the beginning, since each vertex has only one value (*i.e.* its own ID), the update over each edge e is essentially e's source vertex ID. These updates are shuffled into the update tables (#1 in Figure 4.1) across the streaming partitions. Specifically, the update for e, which is e's source vertex ID, goes into the update table of the

```
1  class TriangleCounting : public Application {
2    void run(Engine e){
3      /*Create a dataflow graph*/
4      TCScatter s;
5      e.set_start(&s);
6      R1 r1; R2 r2;
7      e.insert_phase(r1, s);
8      e.insert_phase(r2, r1);
9      e.run();
10   }
11 };
12
13 class R1 : public RPhase{
14   /*Called from join: only keep such <a, b, c> that b < a < c */
15   bool filter(Tuple t1, Tuple t2){
16     if(t1.element(1) > t1.element(0))
17        return FALSE;
18     if(t2.element(0) > t2.element(2))
19        return FALSE;
20     return TRUE;
21   }
22
23   /*Called from join: new key column*/
24   int new_key(){
25     return 2; /* set 'C3' as key*/
26   }
27
28   /*The main entry point*/
29   void execute(StreamingPartition sp){
30     UpdateTable ut = sp.update_table;
31     ut.set_key(0); //set 'VID' as key
32     EdgeTable et = sp.edge_table;
33     /*Join ut with et; et's key is 'Src'; generated tuples are shuffled on new_key*/
34     super::join(sp);
35   }
36 };
37
38 class R2: public RPhase{
39   bool filter(Tuple t1, Tuple t2){
40    if(t2.element(1) != t1.element(0))
41      return FALSE;
42    return TRUE;
43  }
44
45  void execute(StreamingPartition sp){
46    super::join(sp);
47    super::aggregate(sp, COUNT, null);
48  }
49 };
```

Figure 4.2: Triangle counting in RStream.

partition that contains e's destination.

The program has two relational phases R1 and R2. R1 essentially joins all such edges (a, b) with (b, c) to produce relation (a, b, c), while R2 joins (a, b, c) with (c, a) to detect triangles. To implement R1, the developer invokes the `join` function defined in class `RPhase`. This function takes a streaming partition (sp) as input and implements a *fixed semantics* of joining sp's update table (ut) with its own edge table (et) on their key columns. The key column for the update table can be set by using `set_key`, while the edge table always uses the source vertex column as its key column.

Joining the two tables also conducts (1) filtering, (2) tuple reshuffling, and (3) updating of sp's update table. Filtering uses the user-defined `filter` function (Line 15–21). Tuples produced by this join form the new update table of each partition. The user can override the function `new_key` to specify the key column of this new table. If the new key is different than the current key of the update table, the generated tuples need to be reshuffled across partitions — each tuple is sent to the partition that contains the key element of the tuple.

For instance, the invocation of `join` in Line 34 joins the update table #1 with the edge table in Figure 4.1 using the filter defined in Line 15 of Figure 4.2. Specifically, it joins (a, b) with (b, c) and produces tuples of the form (a, b, c). The `filter` function specifies that we select only rows (a, b, c) with $b < a < c$, to filter out duplicates. Next, since function `new_key` specifies C3 as the new key column, each generated (a, b, c) will be shuffled to the streaming partition whose vertex table contains vertex ID c. This provides a benefit of locality for the next join, which will be performed on column C3 of the update table and Src of the edge table. Finally, the update table of each streaming partition sp is updated to the new table containing such (a, b, c) tuples.

The second invocation of `join` in Line 46 joins the update table resulting from R1 (*i.e.* #2 in Figure 4.1) and the same edge table with the filtering condition defined in Line 39–43. The goal of this join is to find tuples of the form (a, b, c) and (c, b) to confirm that (a, b, c) indeed forms a triangle. After R2, the new update table (#3) in each partition contains triangles that can be counted using the aggregation function `aggregate` (Line 47). Here we

do not need a cycle in the dataflow graph and the algorithm ends after the two joins.

Since the example aims to count the total number of triangles, a gather-apply phase is not needed. However, if one wants to count the number of distinct triangles for each vertex, an additional gather-apply phase would be required to stream in triangle tuples from the update table #3 and gather them based on their key element to compute per-vertex triangle counts. The gather phase essentially implements a group-by operation.

Observation on Expressiveness We make several observations with the example. The first one is the expressiveness of the GRAS model. Joins performed by the relational phases over the update table and the edge table enable us to "grow" existing subgraphs we have found (*i.e.* stored in the update table) with edges (*i.e.* stored in the edge table) to form larger subgraphs. This is the key ability enabling Datalog and Arabesque to express mining algorithms. Our GRAS model is *as expressive as Arabesque's filter-process model* – the `filter` function in a relational phase achieves the same functionality as Arabesque's filter while Arabesque's embedding enumeration and processing can be achieved with relational joins between the update and edge tables.

Clearly GRAS is *more expressive than Datalog* – the combination of dataflow cycles and relational joins allows RStream to express algorithms that aim to discover structures whose shapes cannot be described *a priori*, such as subgraph mining.

A surprising side effect of building our programming model on top of GAS is that RStream can also support graph computation algorithms and even the transitive closure computation, which none of the existing mining systems can support. Developing computation algorithms such as PageRank is easy — they need the traditional scatter, gather, and apply, rather than any relational phases.

Observation on Efficiency The locality property of X-Stream is preserved in RStream. Tuple shuffling performed at the end of each join (based on `new_key`) makes it possible for joins to occur locally within each streaming partition *sp*. This is because (1) all the update

tuples whose key column contains a vertex ID belonging to *sp* have been shuffled into the *sp*'s update table, and (2) all the edges whose source vertex (*i.e.* key column) belonging to *sp* are already in *sp*'s edge table. Random accesses may occur only during shuffling; accesses are conducted sequentially in all other phases. Our join is implemented efficiently by tuple streaming (Section 4.3) – since the update table is often orders of magnitude larger than the edge table, RStream loads the edge table in memory and streams in tuples from the update table.

Limitation A limitation of RStream is that it currently assumes a static graph and does not deal with graph updates without restarting the computation. Hence, it cannot be used for interactive mining tasks at this moment.

4.2 Programming Model

This section provides a detailed description of RStream's programming model. Figure 4.3 and Figure 4.4 show the data structures and interface functions provided by RStream. An RStream program is made up of a dataflow graph constructed by the developer. The main entry of an RStream application is a subclass of `Application`, which the developer needs to provide to implement a given algorithm.

Adding Structural Info A special function to be implemented in an application is `need_structure`, which, by default, returns FALSE. As shown in Figure 4.1, each join grows an existing group of vertices with a new edge, generating a new (larger) structure. However, since each tuple currently only contains vertex IDs, the *structural information* of these vertices (*i.e.* edges connecting them) is missing. This will not create a problem for applications such as Triangle Counting because the structure of a triangle is known *a priori*. However, for applications like FSM, the shape of a frequent subgraph needs to be discovered dynamically. Missing structural information in tuples would create two challenges for these applications. First, tuples with identical elements may represent different structures. For example, a tuple

```
/*Data structures*/
template <class T>
class Tuple {
    int num_elements() {...}
    T element(int i){...}
    virtual bool is_automorphic(Tuple t){...}
    virtual bool is_isomorphic(Tuple t){...}
};

class Edge : public Tuple {...};
class Vertex: public Tuple {...};

class Table {
    int get_key(){...}
    void set_key(int i) {...}
};

class UpdateTable : public Table {...};
class EdgeTable : public Table {...};
class VertexTable : public Table {...};

struct StreamingPartition {
    UpdateTable update_table;
    EdgeTable edge_table;
    VertexTable vertex_table;
    virtual void set_init_value(Vertex v);
};
```

Figure 4.3: Major data structures.

$\langle 1, 2, 3, 4\rangle$ may come from the joining of $\langle 1, 2, 3\rangle$ and $\langle 3, 4\rangle$ or of $\langle 1, 2, 3\rangle$ and $\langle 2, 4\rangle$; these are clearly two different subgraphs. The lack of structural information causes RStream to recognize them as the same subgraph instance, leading to incorrect aggregation.

Conversely, missing structural information makes it difficult for RStream to find and merge identical (automorphic) subgraphs that are represented by different tuples. For instance, joining $\langle 1, 2, 4\rangle$ and $\langle 2, 3\rangle$ on the two columns #1 and #0 generates the same subgraph instance as joining $\langle 1, 2, 3\rangle$ and $\langle 2, 4\rangle$ on the columns (#1, #0), although the tuples produced look different ($\langle 1, 2, 4, 3\rangle$ and $\langle 1, 2, 3, 4\rangle$). Failing to identify such duplicates would lead not only to mis-aggregation but also inefficiencies.

To develop applications requiring structural information, a RStream developer can override function `need_structure` to make it return TRUE. This informs RStream to append a

```
class Application{
    /* Dataflow graph registered here */
    virtual void run();
    /* Whether we need structural info*/
    virtual bool need_structure() {return FALSE;}
};

/*Phases*/
class Phase {
    virtual bool converged(TerminationLogic l);
};
class Scatter : public Phase {
    virtual Tuple generate_update(Edge e){...};
};
class GatherApply : public Phase {
    virtual void apply_update(Vertex v, Tuple update);
};

class RPhase : public Phase{
    /* Functions called from join or select*/
    virtual bool filter(Tuple t1, Tuple t2) {return TRUE;}
    virtual int new_key();

    /* Called from the engine*/
    virtual void execute(StreamingPartition p);

    /* == A set of relational functions ==*/
    /* Join ut and et of p and updates ut*/
    void join(StreamingPartition p){...}
    /* Join ut and et of p on all columns of ut and updates ut*/
    void join_on_all_columns(StreamingPartition p){...}
    /* Select rows from ut of p and updates ut*/
    void select(StreamingPartition p){...}
    /* Aggregate rows from ut of p*/
    void aggregate(StreamingPartition p, int type){...}
};
```

Figure 4.4: API functions.

piece of information regarding each join to each tuple produced by the join. For example, joining $\langle 1, 2\rangle$ with $\langle 2, 3\rangle$ on the columns (#1, #0) produces a tuple $\langle 1, 2, 3, (1)\rangle$, where (1) indicates that this tuple comes from expanding a previous tuple with an edge on its 2nd column.

A further join between $\langle 1, 2, 3, (1)\rangle$ and $\langle 2, 4\rangle$ on the columns (#1, #0) generates tuple $\langle 1, 2, 3, 4, (1, 1)\rangle$, which indicates that this tuple comes from first expanding the second col-

umn with an edge and then the second column with another edge. This piece of information is added (implicitly) at the end of each tuple, encoding the history of joins, which, in turn, represents the edges that connect the vertices in the tuple.

This structural information is needed in the following two scenarios. First, it is used to encode a subgraph represented by a tuple into a coordination-free *canonical form*, which can be used by the function `is_isomorphic` (defined in `Tuple`) during aggregation to find *isomorphic subgraphs*. Two subgraphs (*i.e.* tuples) are *isomorphic* iff there exists a one-to-one mapping between their vertices and between their edges, *s.t.* (1) each vertex/edge in one subgraph has one matching vertex/edge in another subgraph, and (2) each matching edge connects matching vertices. Tuples are aggregated at the end based on isomorphism-induced equivalence classes.

Second, the structural information is used to identify tuples representing the same subgraph instance (*i.e.* by `is_automorphic`). Two subgraphs are *automorphic* iff they contain the same edges and vertices. Tuples that represent the same subgraph instance need to be merged during computation for correctness and performance. The implementation of these functions is discussed in Section 4.3.

RStream tuples are essentially vertex-based representations of subgraphs. Edges are represented as structural information appended at the end of each tuple. Compared to Arabesque where each subgraph (embedding) has an edge-based representation, RStream's representation allows the application to express whether the edge information is needed, providing space efficiency for applications that aim to find statically-known patterns and thus do not need the edge information.

Relational Phases Operations that can be performed in a relational phase include `join`, `select`, `aggregate`, and `join_on_all_columns`. `join` joins the update table with the edge table of each streaming partition on their key columns; `select` selects rows from the update table based on the user-defined filter; and `aggregate` aggregates values from all rows in the update table. The "type" parameter of `aggregate` indicates the type of aggregation such as

MAX, MIN, SUM, COUNT, or STRUCTURE_SUM. A special type is STRUCTURE_SUM, which counts the number of subgraphs that belong to the same isomorphism class. If a programmer needs to aggregate over a subset of rows, she can first invoke `select` and then `aggregate`. `join` and `select` change the update table while `aggregate` does not. `join_on_all_columns` will be discussed shortly.

The two callback functions `filter` and `new_key` in class `RPhase` are invoked by `join`, `select`, and `join_on_all_columns` to determine what rows need to be considered and how results should be shuffled, respectively. For either `join` or `select`, changing the key column of the update table (*i.e.* using `new_key`) will trigger tuple shuffling across streaming partitions.

Note that `RPhase` does not provide a `group-by` function, because group-by can be essentially implemented by a gather-apply phase. During a gather-apply, the vertex table is loaded into memory and tuples from the update table (produced either by a scatter phase or by a relational phase) are streamed in. RStream gathers tuples that have the same key element (*i.e.* vertex ID) and invokes the user-defined `apply_update` function at Line 16 to compute a new tuple for the vertex. These new tuples are then saved into the vertex table, which is written back to disk at the end of each iteration. In other words, gather-apply produces a new vertex table.

`join_on_all_columns` is the same as `join` except that it joins the update table with the edge table *multiple times*, each time using a different column from the update table as key. The key of the edge table remains unchanged (*i.e.* source vertex column). The number of joins performed by this function equals the number of columns in the update table. This function is necessary to implement mining algorithms that need to grow a subgraph from all of its vertices, such as Clique or FSM.

Figure 4.5 illustrates `join_on_all_columns`. Since it changes the key of the update table for each join, RStream shuffles tuples *twice* after a join — the first one, referred to as input shuffle (I-shuffle), shuffles tuples from the update table based on the next key to be used to prepare for the next join; the second one, referred to as output shuffle (O-shuffle), shuffles

the result tuples based on the new key defined by `new_key` to prepare for the final output, which will eventually become the new update table (UT′).

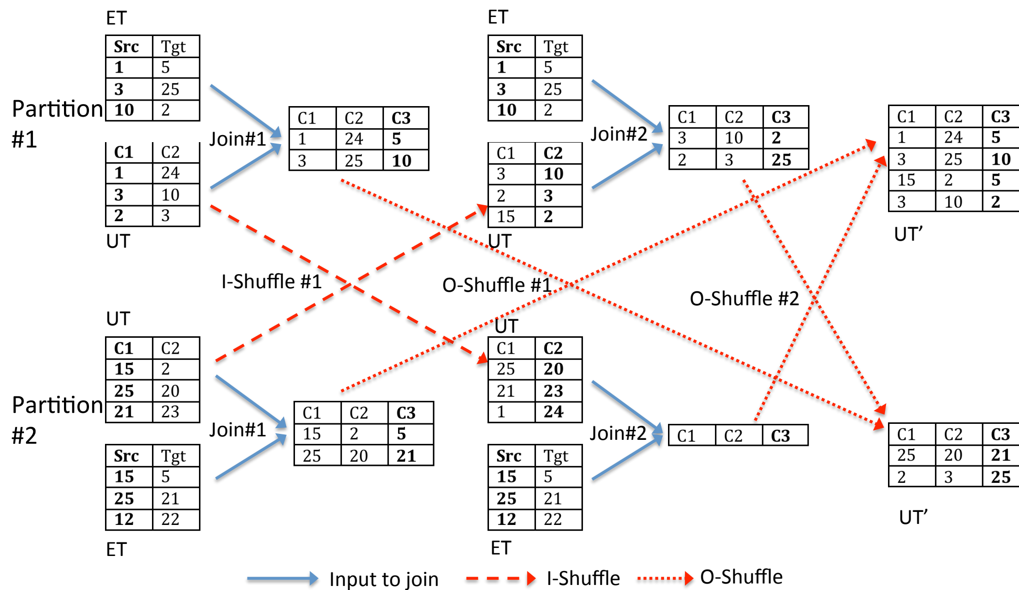


Figure 4.5: A graphical illustration of `join_on_all_columns`; the streaming partitions #1 and #2 contain vertices [0, 10] and [11, 25], respectively; suppose `new_key` returns 2 (which is column C_3). Structural info is not shown.

Termination Class `Phase` contains an abstract function `converged` that needs to be implemented in user-defined phases. This function defines termination logic for iterative computation algorithms (with back edges on the dataflow graph). Note that RStream invokes this function only for the phases that are sources of dataflow back edges to determine whether further iterations are needed.

Example: FSM on RStream We use one more example — frequent subgraph mining — to demonstrate the power of RStream's programming model, and in particular, the usage of dataflow cycles and the function `join_on_all_columns`. Figure 4.6 shows the computation logic. It consists of two phases: a (standard) scatter phase and an iterative relational phase `FSMPhase`. The basic idea is that each execution of `FSMPhase` performs `join_on_all_columns`

```
1  class FSMProgram : public Application {
2     /*FSM needs structural info*/
3     bool need_structure() { return TRUE; }
4
5     void run(Engine e){
6        Scatter cs;
7        e.set_start(cs);
8        FSMPhase fsm;
9        e.insert_phase(fsm, cs);
10       /* This forms a cycle */
11       e.insert_phase(fsm, fsm);
12       e.run();
13    }
14 };
15
16 class AggregateFilter : public RowFilter{
17    AggregationStream aggStream;
18    int threshold;
19
20    bool filter_out_row(Tuple t){
21       int support = get_support(aggStream, t);
22       if(support >= threshold) return FALSE;
23    /*It couldn't be a frequent subgraph.*/
24       return TRUE;
25   }
26 };
27
28 class FSMPhase : public RPhase{
29    static int MAX_ITE = MAX_FSM_SIZE * (MAX_FSM_SIZE - 1)/2;
30
31    bool converged(TerminationLogic l) {
32       if(l.get_ite_id() == MAX_ITE) return TRUE;
33       return FALSE;
34   }
35
36   int new_key(){ return LAST_COLUMN;}
37
38   void execute(StreamingPartition sp){
39      UpdateTable ut = sp.update_table;
40      ut.set_key(0);
41      EdgeTable et = sp.edge_table;
42      et.set_key(0);
43      super::join_on_all_columns(sp);
44      super::aggregate(sp, STRUCTURE_SUM);
45      AggregateFilter af;
46      super::select(sp, af);
47   }
48 };
```

Figure 4.6: An FSM program; structural info is needed.

between the update and edge table. Each tuple in the update table represents a new subgraph we have found. This special join attempts to "grow" each subgraph with one edge on each vertex in the subgraph. For example, for a tuple (a, b, c, d), this join will join it with the edge table *four times*, each on a different column. Each join generates five-tuples of the form (a, b, c, d, e), which is keyed at e (*i.e.* LAST_COLUMN specified in Line 36). Such tuples are shuffled into the partitions to which e belongs.

Given the max size of subgraphs to be considered (*e.g.* $MAX_FSM_SIZE = 4$), all we need is to execute `FSMPhase` for a fixed number of times; this number equals the maximum number of edges that can be involved in the largest FSM: $MAX_FSM_SIZE \times (MAX_FSM_SIZE - 1)/2$, as shown in Line 29.

At the end of each `FSMPhase`, we aggregate all tuples in the update table (Line 44) to count the number of each distinct structural pattern. After the aggregation, a `select` is performed to filter out tuples corresponding to infrequent subgraphs (Line 46). This function takes as input a variable of class `AggregateFilter`, which contains a function `filter_out_row` that will be applied to each tuple. This function eliminates tuples that represent structural patterns whose supports are not high enough (Lines 20-25). The intuition here is that if a subgraph is infrequent, then any supergraphs generated based on it must be infrequent — referred to as the Downward Closure Property [16]. These infrequent tuples can be safely ignored in the next iteration. Similarly to Arabesque [144], we use the *minimum image-based support metric* [42] as it can be efficiently computed. This metric defines the frequency of a structural pattern as the *minimum* number of distinct mappings for any vertex in the pattern over all instances of the pattern.

4.3 RStream Implementation

RStream's implementation has an approximate of 7K lines of C++ code and is available on Github.

4.3.1 Preprocessing

For graphs that cannot fit into memory, they are first partitioned by a *preprocessing* step. The graph is in the edge-list or adjacency-list format on disk. RStream divides vertices into logical intervals. One interval in RStream defines a partition that contains edges whose *source vertices* fall into the interval. Edges that belong to the same partition do not need to be further sorted. To achieve work balance, we ensure that partitions have similar sizes. Since our join implementation (discussed shortly) needs to load each edge table entirely into memory, the number of streaming partitions is determined automatically to guarantee that the edge table for each streaming partition does not exceed the memory capacity while memory can still be fully utilized.

For graphs that can be fully loaded, RStream generates one single partition and no tuple shuffling will be incurred for joins. However, unlike share-memory graph computation systems that can hold all computations in memory, mining algorithms in RStream can cause update tables to keep increasing — even for very small graphs, their update tables can grow to be several orders of magnitude larger than the size of the original graph. Hence, RStream requires disk support regardless of the initial graph size.

4.3.2 Join Implementation

As the update table grows quickly, to implement join, we load the edge table into memory and *stream in* tuples from the update table for each streaming partition. RStream performs *sequential disk accesses* to both the update table and the edge table, and *random memory accesses* to the loaded edge data.

Note that the edge table represents the original graph while the update table contains intermediate data generated during computation. Since the edge table never changes, the amount of memory required by RStream is bounded by the maximum size of a partition in the original graph, *not* the intermediate computation data, which can be much larger than the graph size.

Scatter and gather-apply are implemented in the same way as in X-Stream — for scatter, the vertex table is loaded while edges are streamed in; for gather-apply, the vertex table is loaded while updates are streamed in.

Filtering is performed by invoking the user-defined `filter` function upon the generation of a new tuple. When *join_on_all_columns* is used, different tuples generated may represent identical (automorphic) structures. Similarly to Arabesque, we define *tuple canonicality* by selecting a unique (canonical) tuple from its automorphic set as a representative and remove all other tuples. Details of this step are discussed shortly in Section 4.3.3.

Multi-threading RStream uses a producer-consumer paradigm for implementing join. The main thread pushes the IDs of the streaming partitions to be processed into a worklist as tasks, and starts multiple producer and consumer threads. Each producer thread pops a task off the list, loads its edge table, and streams in its update table into the producer's thread-local buffer. The producer thread joins each "old" update tuple with the edge table and produces a "new" update tuple.

We allocate a *reshuffling buffer*, for each streaming partition, to store new update tuples entering this partition. Producers and consumers synchronize using locks to ensure concurrent accesses to reshuffling buffers. Each producer sends each generated tuple to its corresponding reshuffling buffer when the buffer has room, while each consumer flushes a buffer into its corresponding "new" update table on disk when the buffer is full.

Figure 4.7 illustrates multiple producers and consumers. There are four producer threads and two consumer threads. Eight tasks are pushed onto the task worklist. Each producer takes one task from the list, loads its edge partition, and streams in its update partition. Each producer conducts the computation and generates output updates locally. Reshuffling is synchronized using std::mutex.

Load (Re)balancing Unlike X-Stream where the size of each streaming partition stays unchanged, in RStream, the size of each partition can grow significantly for two reasons.

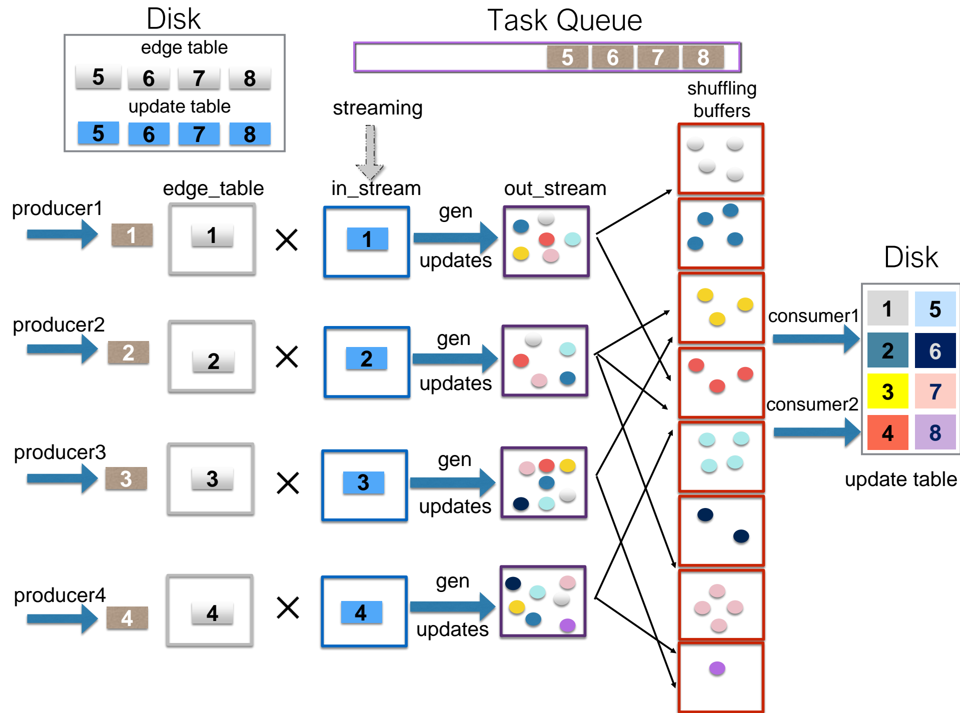


Figure 4.7: A graphical illustration of multiple producers, multiple consumers and reshuffling buffers.

First, mining algorithms keep looking for graph patterns of increasing sizes, leading to the ever-growing update table. Second, tuple reshuffling at the end of each join can result in unbalanced partitions. These unbalanced partitions, if handled inappropriately, can result in significant inefficiencies (*e.g.* underutilized CPU).

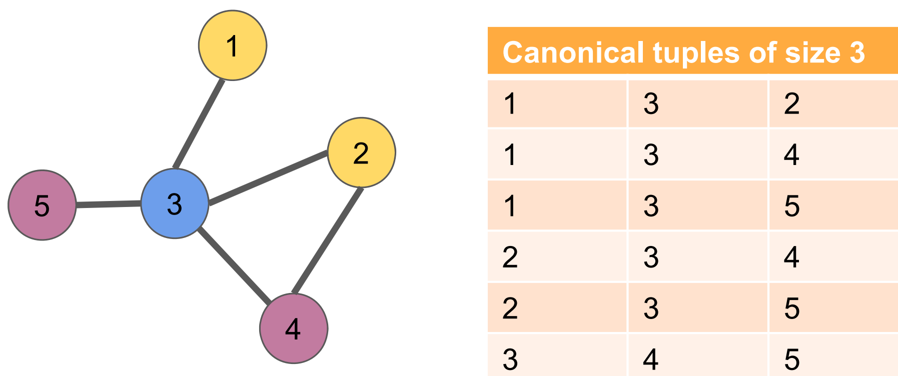
One possible solution would be to repartition the streaming partitions at the end of each relational phase for load rebalancing. However, repartitioning can incur significant disk I/O, slowing down the computation. Rather than repartition the graph, we use fine-grained tasks by dividing each update table into multiple smaller update chunks. Instead of pushing an entire update partition into the list, we push one chunk at a time. For work balancing, we also order these tasks based on their sizes so that "larger" tasks have a higher priority to be processed.

Enumeration Note that, by joining the update table with the edge table, RStream performs *breadth-first* enumeration of subgraphs. While this approach requires more storage to materialize tuples compared to a depth-first approach, it enables easier parallelization as all tuples of a given size are materialized and available for processing. Further, as a disk-based approach, RStream's breadth-first enumeration increases disk usage rather than memory usage — As shown in Figure 4.7, the enumeration delivers each newly generated tuple to a shuffling buffer and once the buffer is full, RStream flushes the buffer to disk.

4.3.3 Redundancy Removal via Automorphism Checks

Since different workers can reach identical (automorphic) tuples during processing, we need to identify and filter out such tuples. RStream adopts the idea of *embedding canonicality* used in Arabesque [144]. We select exactly one of the automorphic tuples and elect it as "canonical". RStream runs a tuple canonicality check to verify whether a tuple t can be pruned. This algorithm runs on a single tuple without coordination. It starts with an existing canonical tuple t and checks, when t is grown with a new vertex v into a new tuple t', whether t' is also canonical. The basic idea is based on a notion of *uniqueness*: given the set S_m of all tuples automorphic to a tuple m, there exists exactly one canonical tuple t_c in S_m. The goal of this algorithm is, thus, to check whether the newly generated tuple t' is this t_c.

The tuple t' is canonical if and only if its vertices are visited in an order that is consistent with their IDs: a vertex with a smaller ID is visited earlier than one with a larger ID. In other words, RStream characterizes a tuple as the list of its vertices sorted by the order in which they are visited. When we check the canonicality of tuple t' that comes from growing an existing canonical tuple t with a vertex v, we first find the first neighbor v' of v, and then verify that there is no vertex $\in t$ after v' with a larger ID than v. Figure 4.8 shows a simple graph and its canonical tuples of size 3. Because RStream only processes canonical tuples, *uniqueness* is maintained in our tuple encoding (with structural information). A more detailed description can be found in [143].



Canonical tuples of size 3		
1	3	2
1	3	4
1	3	5
2	3	4
2	3	5
3	4	5

Figure 4.8: A graph and its canonical tuples of size 3.

4.3.4 Pattern Aggregation via Isomorphism Checks

For mining algorithms, aggregation needs to be done on tuples to count the number of each distinct shape (*i.e.* structural pattern) at the end of the computation. Aggregation boils down to isomorphism checks — among all non-automorphic tuples, we count the number of those that belong to each isomorphism class. A challenge here is that isomorphism checks are expensive to compute — it is known to be isomorphism (GI)-complete and the bliss library [12] we use employs an exponential time algorithm.

RStream adopts the aggregation idea from Arabesque by turning each tuple into a *quick pattern* and then into a *canonical pattern* [30, 144]. The canonical pattern of a subgraph, which is different than the canonical tuple described earlier for automorphism checks, encodes the shape of the subgraph with all vertex information removed. Two tuples are isomorphic iff they have the same canonical patterns. The quick pattern of a subgraph is simply a total order of edges in the subgraph with vertex information removed. Two tuples may have different quick patterns even if they are isomorphic.

Given that canonical checks are expensive, we use the same two-step aggregation as in Arabesque — the first step uses quick patterns that can be efficiently computed to perform *coarse-grained* pattern aggregation, while the second step takes as input results from the first step, converts them into canonical patterns, based on which *fine-grained* aggregation is

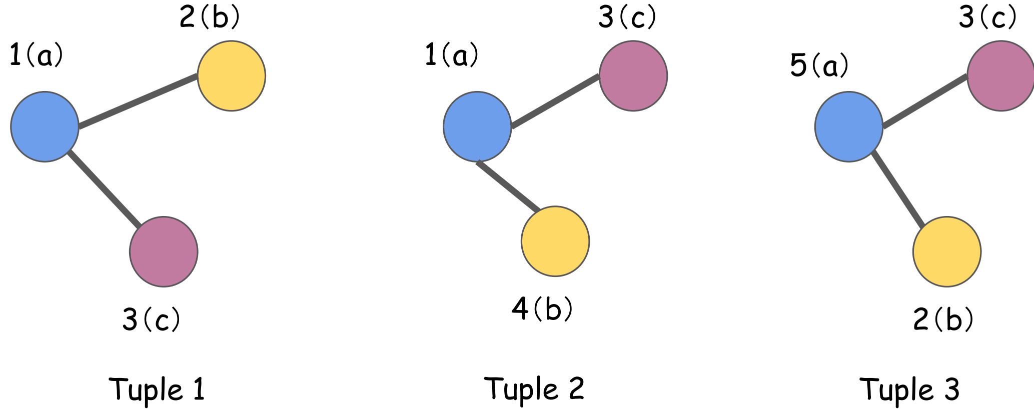


Figure 4.9: Aggregation example of three isomorphic tuples.

done. The aggregation conducts a two-stage MapReduce computation — the first on quick patterns and the second on canonical forms — across *all* streaming partitions. Although the aggregation idea originates from Arabesque [144], we provide a detailed example in the rest of this section to make this paper more self-contained.

Example The map phase takes quick patterns and canonical forms as input, performs local aggregation, and shuffles them into hash buckets defined by the hash value of these patterns. The reduce phase aggregates key/value pairs in the same bucket. Figure 4.9 depicts an example with three tuples: $tuple_1 : \langle 1(a), 2(b), 3(c), (0)\rangle$, $tuple_2 : \langle 1(a), 3(c), 4(b), (0)\rangle$, and $tuple_3 : \langle 5(a), 3(c), 2(b), (0)\rangle$. Here numbers represent vertex IDs and characters represent labels for each vertex. Note that mining algorithms often require graphs to have vertices and edges explicitly labeled. These labels represent vertex/edge properties that never change during the computation and they are needed for isomorphism checks. (0) represents the structural information obtained from the past joins.

RStream first turns each tuple into a quick pattern to reduce the number of distinct tuples. A quick pattern is obtained by simply extracting the label information and renaming vertex IDs in a given tuple, with vertex ID always starting at 1 and increasing consecutively. In the previous example, the quick patterns for the three tuples are $qp_1 : \langle 1(a), 2(b), 3(c), (0, 0)\rangle$, $qp_2 : \langle 1(a), 2(c), 3(b), (0, 0)\rangle$, $qp_3 : \langle 1(a), 2(c), 3(b), (0, 0)\rangle$, respectively. In the map phase,

RStream emits three quick pattern pairs: (qp_1, 1), (qp_2, 1), (qp_3, 1); the reduce phase further aggregates them into (qp_1, 1), (qp_2, 2) as qp_2 and qp_3 are identical.

Due to the coarse-grained modeling of quick patterns, tuples that are actually isomorphic may correspond to different quick patterns. As a next step, quick patterns are turned into canonical forms (by bliss) to perform fine-grained aggregation. A canonical form uniquely identifies a class of isomorphic subgraphs. In the example, the two quick patterns correspond to the same canonical form $cf_1 : \langle 1(a), 2(b), 3(c), (0, 0)\rangle$. RStream eventually reports (cf_1, 3) as the final result. Since the number of quick patterns is much smaller than the number of distinct tuples, the cost of isomorphic checks can be significantly reduced.

One possible optimization is to perform *eager aggregation* — tuples are aggregated as they are being streamed into their respective partitions. We have implemented this optimization, but our experimental results showed only a minor improvement (5% in the aggregation phase and less than 2% for the overall execution).

4.4 Evaluation

Our evaluation focuses on three research questions:

- Q1: How does RStream compare to state-of-the-art graph mining systems? (Section 4.4.1)
- Q2: How does RStream compare to state-of-the-art Datalog engines? (Section 4.4.2)
- Q3: What is RStream's overall and I/O throughput and how quickly does data grow for mining algorithms? (Section 4.4.3)

Experimental Setup We ran our experiments using six algorithms (Table 4.2) over six real-world graphs (Table 4.1). CiteSeer, MiCo, and Patents are the graphs that were used by Arabesque and DistGraph in their evaluations. We used them primarily for comparisons with the mining systems. Similarly, Orkut and LiveJournal were used by BigDatalog [133]

Graphs	#Edges	#Vertices	Description
CiteSeer [59]	4,732	3,312	CS pub graph
MiCo [59]	1.1M	100K	Co-authorship graph
Patents [70]	14.0M	2.7M	US Patents graph
LiveJournal [31]	69M	4.8M	Social network
Orkut [1]	117M	3M	Social network
UK-2005 [39]	936M	39.5M	Web graph

Table 4.1: Real world graphs.

Program	LoC	Description
Triangle Counting (TC)	75	Counting # triangles
Closure	68	Computing transitive closure
N-Clique	36	Identify cliques of size N
N-Motif	36	Counting motifs of size N
Frequent Subgraph Mining (FSM)	40	Identify FSM of size N
Connected Components (CC)	40	Identify connected components

Table 4.2: Algorithms experimented.

and we used them to compare RStream with BigDatalog. UK-2005 has almost a billion edges and is much larger than all the graphs used by Arabesque [144].

For mining algorithms, we developed Triangle Counting (TC), Clique, Motif Counting (MC), Transitive Closure Computation (Closure), and Frequent Subgraph Mining (FSM). Closure is a typical Datalog workload, and hence, we used it specifically to compare RStream with Datalog. Connected Components (CC) is a graph computation algorithm. Since RStream can also support computation (with just GAS and no relational phases), we added CC into our algorithm set to help us develop a deep understanding of the behavioral differences between graph computation and graph mining (Section 4.4.3).

Our experiments were conducted on a 10-node cluster, each with 2 Xeon(R) CPU E5-

2640 v3 processors, 32GB memory, and 3 SSDs with a total of 5.2TB disk space, running CentOS 6.8. Data was split evenly on the three disks. RStream ran on one single node with 32 threads to fully utilize CPU resources and disk bandwidth, while distributed systems used all the nodes.

4.4.1 Comparisons with Mining Systems

Systems and Algorithms We compared RStream with three state-of-the-art distributed mining systems: Arabesque [144], ScaleMine [13], and DistGraph [141]. We ran these three systems with 10 nodes, 5 nodes, and 1 node to have a precise understanding of where RStream stands. In this first set of experiments, all Motif executions were run with a maximum size of 4; Clique was run with a maximum size of 5; and FSM was run with size of 3.

As discussed earlier, to run FSM we used the minimum image-based support metric [42], which defines the frequency of a pattern as the minimum number of distinct mappings for any vertex in the pattern, over all instances of the pattern. We explicitly state the support, denoted S, used in each experiment since this parameter is sensitive to the input graph. Clearly, the smaller S is, the more computation is needed.

In this experiment, we used CiteSeer, MiCo, and Patent as our input graphs. These three graphs came with labels[2] and were also used to evaluate Arabesque, ScaleMine, and DistGraph. Our initial goal was to evaluate RStream with all graphs used in prior works, but other graphs were either unavailable or do not have labels. Although these are relatively small graphs from the perspective of graph computation, running mining algorithms on them can generate orders-of-magnitude more data (see Table 4.5).

Table 4.3 reports the running times of the four systems. Note that ScaleMine and DistGraph were designed specifically to mine frequent subgraphs, and hence we could obtain only FSM's performance for these two systems. It is clear that RStream **outperforms all three systems in all cases but 3-FSM with support = 5000**. Arabesque, ScaleMine,

[2]Mining algorithms require *labeled graphs* (*i.e.* vertices and edges have semantic labels).

		CS	MC	PA
TC	RS	**0.04**	**15.8**	**6.7**
	AR-10	38.1	43.1	114.9
	AR-5	39.8	44.9	116.4
	AR-1	34.2	40.7	131.5
5-C	RS	**0.01**	**115.1**	**35.3**
	AR-10	42.8	132.0	174.5
	AR-5	39.3	171.7	183.0
	AR-1	34.9	404.3	227.9
3-M	RS	**0.02**	**43.0**	**89.1**
	AR-10	40.6	51.7	116.0
	AR-5	39.7	52.8	110.5
	AR-1	32.7	47.0	132.9
4-M	RS	**1.41**	**52926**	**8849**
	AR-10	41.7	-	-
	AR-5	40.4	-	-
	AR-1	34.2	-	-
3-F 300	RS	**0.89**	**402.1**	**517.4**
	AR-10	35.9	-	-
	AR-5	39.3	-	-
	AR-1	33.7	-	-
	SM-10	2.1	69431.7	-
	SM-5	2.6	66604.3	-
	SM-1	3.5	77332.7	-
	DG-10	12.3	-	-
	DG-5	4.1	-	-
	DG-1	5.2	-	-
3-F 500	RS	**0.10**	**384.3**	**502.1**
	AR-10	35.7	-	-
	AR-5	39.3	-	-
	AR-1	34.4	-	-
	SM-10	2.0	15867.5	-
	SM-5	2.3	15209.4	-
	SM-1	3.2	21043.3	-
	DG-10	0.4	-	-
	DG-5	0.12	-	-
	DG-1	0.11	-	-
3-F 1K	RS	**0.06**	**351.7**	**383.7**
	AR-10	35.6	5790.1	-
	AR-5	39.9	5397.9	-
	AR-1	33.9	5848.2	-
	SM-10	1.2	802.6	-
	SM-5	1.1	790.8	-
	SM-1	1.1	1175.1	-
	DG-10	0.4	-	-
	DG-5	0.12	-	-
	DG-1	0.10	-	-
3-F 5K	RS	**0.02**	51.0	**376.4**
	AR-10	41.6	120.8	-
	AR-5	37.7	192.7	-
	AR-1	31.8	610.3	-
	SM-10	1.0	12.1	-
	SM-5	1.1	**11.6**	-
	SM-1	1.3	14.5	-
	DG-10	0.3	-	-
	DG-5	0.05	-	-
	DG-1	0.08	-	-

Table 4.3: Comparisons between RStream (RS), Arabesque (AR-n), ScaleMine (SM-n), and DistGraph(DG-n) on four mining algorithms — triangle counting (TC), Clique (k-C), Motif Counting (k-M), and FSM (k-F) — over three graphs CiteSeer (CS), MiCo (MC), and Patents (PA); n represents the number of nodes the distributed systems use; k is the size of the structure to be mined; '-' indicates execution failures. For FSM, four different support parameters (300, 500, 1K, and 5K) are used and explicitly shown in each 3-F row. Highlighted rows are the shortest times (in seconds).

and DistGraph failed when the size of a pattern increases. These failures were primarily due to their high memory requirement (for storing intermediate data) that could not be fulfilled by our cluster.

For FSM, on small graphs such as CiteSeer, DistGraph appears to be more efficient than the other two systems. However, DistGraph could not scale to the MiCo graph on our 10-node cluster. ScaleMine successfully processed MiCo, but took a long time, because ScaleMine trades off computation for memory; instead of caching intermediate results in memory, it always re-computes from scratch, which explains why it has better scalability but lower efficiency. None of these three systems could process FSM over the Patents graph even when support = 5000. By contrast, RStream successfully executed FSM over all the graphs under all the configurations.

RStream underperforms ScaleMine in only one case: 3-FSM (S=5000) over MiCo. RStream outperforms Arabesque (on 10 nodes) by an overall (GeoMean) of **60.9**×, ScaleMine by an overall of **12.1**×, and DistGraph by an overall of **7.2**×. As Arabesque was developed in Java, the 60.9× speedup may be partly due to RStream's use of an efficient language (C++). ScaleMine and DistGraph were both C++ applications and, hence, the wins over them provide a closer approximation of the benefit a disk-based system could offer.

UK Graph To understand RStream's performance on larger graphs, we ran 3-FSM on RStream to process the UK-2005 graph that has almost a billion edge. Note that none of the three distributed systems could process the graph when running 3-FSM with even a 5K support on our 10-node cluster. In all prior works, the only evidence of a mining system successfully processing a billion-edge graph was reported in [141] where DistGraph, using 512–2048 IBM BlueGene/Q machines each with 16 cores and 256GB memory, processed several synthetic graphs with 1B–4B edges in 2000 – 7000 seconds (with varying supports). Here we experimented RStream with four support parameters – 2K, 3K, 4K, and 5K – on one single machine with only 32GB memory. RStream successfully processed all of them, *e.g.* in 4080.9, 3016.3, 2228.9, and 2146.2 seconds, respectively.

RStream ran out of memory when a relatively small support was used (*i.e.* ≤1000) to compute frequent subgraphs over UK. After spending a great amount of time investigating the problem, we found that the large memory consumption was potentially due to memory leaks in the bliss library rather than RStream, which guarantees that the amount of data to be loaded from each streaming partition never exceeds the memory capacity.

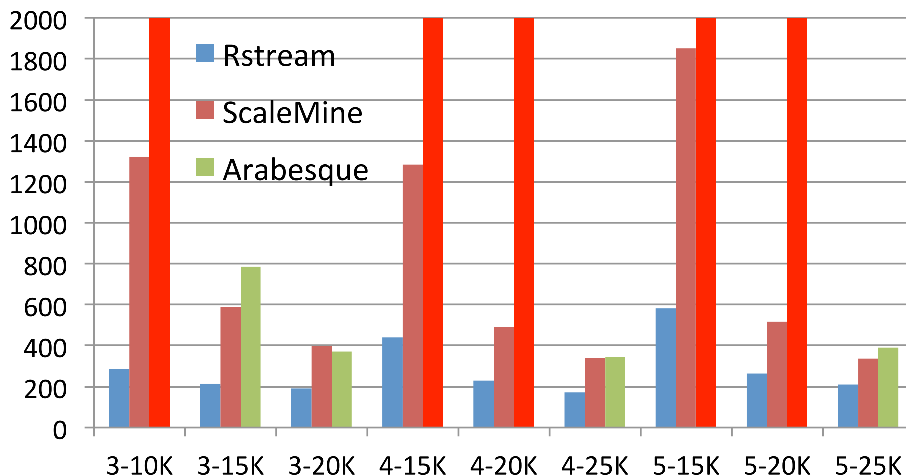


Figure 4.10: FSM performance comparisons with different pattern sizes and supports over the Patents graph. Tall red bars on the right of each group represent Arabesque failures.

Larger FSMs To evaluate how RStream performs on k-FSMs with larger k, we conducted a set of experiments over the Patents graph with various k and supports. Since DistGraph failed in most cases when we increased k, this set of experiments focused on RStream, ScaleMine, and Arabseque, and the results of the comparisons are reported in Figure 4.10. Both Arabesque and ScaleMine were executed with 10 nodes. Overall, RStream is **2.46**× and **2.28**× faster than ScaleMine and Arabesque.

We have also compared RStream with GraMi [59], which is a specialized graph mining library designed to perform single-machine shared-memory FSM computation, over the Patents and Mico graphs. Table 4.4 reports the results. Note that, for each support, GraMi reports patterns of all sizes with respect to the support. RStream was executed in a similar

Support	**Patents**		**Mico**	
	RStream	GraMi	RStream	GraMi
5K	**504.6**	-	**51.0**	-
10K	**286.7**	-	**23.2**	36.5
15K	**213.3**	-	**14.3**	18.7
20K	**190.8**	-	**8.6**	9.2

Table 4.4: FSM performance comparisons between RStream and GraMi over Patents and Mico; time is measured in seconds.

way to provide a fair comparison. GraMi ran out of memory for all cases over the Patents graph. On the Mico graph, RStream outperforms GraMi even for large (*e.g.* 20K) supports.

There are two reasons that could explain RStream's superior efficiency. First, joins performed by RStream grow subgraphs *in batch* while the other systems enumerate and grow embeddings individually. Second, the three systems RStream was compared against are all distributed systems that have a large startup and communication overhead. While the data size quickly grows to be larger than the memory capacity of a single machine, this size is often small in an early stage of the execution. Distributed systems suffer from communication overhead throughout the execution, while RStream does not have heavy I/O in this early stage.

The fact that the three distributed systems failed in many cases does not necessarily indicate that RStream can scale to larger graphs than them. We believe that these systems, if given enough memory, should have performed better than what is reported in Table 4.3. However, their exceedingly high memory requirement is very difficult to satisfy — the 10-node cluster we used is the only cluster to which we have exclusive access. According to [144], running 4-motif on a 200M-edge graph took Arabesque 6 hours consuming 20 $\times$ 110GB = 2200GB memory. As a reference point, the most memory-optimized cluster (x1.32xlarge) Amazon EC2 offers has only 1952GB memory, which is still not enough to run the algorithm.

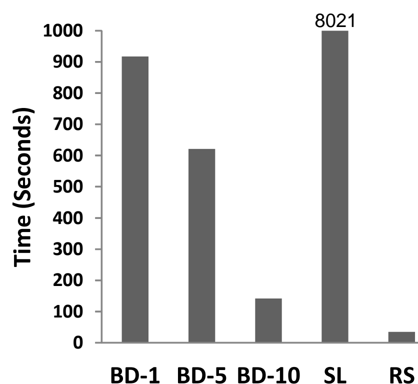
These results do suggest, though, that if a user has only a limited amount of computing resources, RStream should be a better choice than these other systems because RStream's disk requirement is much easier to fulfill and yet it can scale to large enough real-world graphs.

4.4.2 Comparisons with Datalog Engines

Since our GRAS model is inspired partly by the way Datalog enables easy programming of mining algorithms, we have also compared RStream with the state-of-the-art Datalog engines. We use BigDatalog [133] with Spark joins and SociaLite [90], a shared memory Datalog engine. We used the LiveJournal and Orkut graphs, which were initially used to evaluate BigDatalog [133] to evaluate BigDatalog. We used three algorithms: Triangle Counting (TC), Connected Components (CC), and Closure Computation (Closure). Although CC and Closure are not typical mining algorithms, they are Datalog programs regularly used to evaluate the performance of a Datalog engine. Hence, we included them in this experiment. Note that BigDatalog has been shown to outperform vanilla Spark over these workloads due to several optimizations implemented over Spark joins [133].

		LiveJournal	Orkut
TC	**RS**	**87.0**	**827.4**
	BD-10	94.8	1205.3
	BD-5	109.6	1850.3
	BD-1	567.3	-
	SL	896.1	-
CC	**RS**	**101.1**	**352.0**
	BD-10	179.6	441.2
	BD-5	251.3	420.3
	BD-1	187.8	445.2
	SL	379.8	-

(a)



(b)

Figure 4.11: (a) Comparisons between RStream (RS), BigDatalog (BD-n), and SociaLite (SL) on TC and CC; (b) Closure comparison over CiteSeer.

Figure 4.11(a) compares the performance of RStream with that of BigDatalog and So-

ciaLite. For TC and CC, RStream outperforms BigDatalog (with 10 nodes) by a GeoMean of **1.37**×, while SociaLite failed in most cases. For transitive closure, CiteSeer was the only graph that RStream, BigDatalog, and SociaLite could all successfully process. Their performance comparison is shown in Figure 4.11(b): RStream is **4**× faster than BigDatalog running on 10 nodes, while it took SociaLite a large amount of time (8021 seconds) to finish closure computation.

These results appear to be different from what was reported in the prior works [133] and [90]. We found that the difference was primarily due to the input graphs — both the works [133] and [90] used synthetic acyclic graphs for transitive closure, while real graphs have both cycles and very high density that synthetic graphs do not have. Neither BigDatalog nor SociaLite could finish closure computation for any graph other than CiteSeer, while RStream successfully computed closure for LiveJournal in 4578 seconds.

4.4.3 RStream Performance Breakdown

To fully understand RStream's performance, throughput, I/O efficiency, and disk usage, we have conducted a set of experiments using various graphs and algorithms.

Intermediate Data Generation Table 4.5 reports, for 4-Motif (over the Patents graph) and 4-FSM (over the Patents graph), the number of tuples generated at the end of each phase, the size of each tuple, as well as the storage consumption of these tuples. The amount of data generated during the execution can easily exceed the memory capacity. For 4-Motif, the total amount of intermediate data generated requires 1.21TB of disk space. This motivates our out-of-core design that leverages large SSDs to store these intermediate subgraphs.

To understand how large the total amount of data generated is, Table 4.6 further reports, for each graph, the ratio between the amount of storage needed at the end of each execution and the original size of the graph. This growth can be as large as 5 orders of magnitude (4-Motif over the MiCo graph). These ratios also reflect (1) the density of each graph (regardless of the size of the graph), which determines how difficult the graph is to process; and (2) the

	Phase	#Tuples	TS	#MB
4-Motif MiCo	0	1,080,156	16	16.5
	1	91,151,339	24	2,086.3
	2	29,044,509,725	32	886,378.1
	3	17,621,170,674	40	672,194.3
	Total	4.7×10^{10}	-	1,560,675.2 (1.49TB)
4-FSM, S=10K Patents	0	13,965,409	16	213.1
	1	625	28	0.02
	2	5,861,830	16	89.4
	3	93,313,116	24	2,135.8
	4	13,764	36	0.5
	5	29,462,761	24	674.3
	6	816,909,842	32	24,930.1
	7	101,254	44	4.2
	8	633,673,981	32	19,338.2
	9	57,361,813	40	2,188.2
	10	30,283	52	1.5
	11	509,304	40	19.4
	Total	1.65×10^{9}	-	49,594.72 (48.4GB)

Table 4.5: The number of tuples (**Tuples**) generated for each phase execution, the size of each tuple (**TS**), and the number of bytes (**#MB**) shuffled for 4-Motif over the Patents graph and 4-FSM, S=10K over the Mico graph.

computation complexity of each algorithm, which determines how difficult the algorithm is to run. The MiCo graph is the one with the highest density, although it is relatively small in size. 4-Motif is the algorithm that needs the most computations as it generates the most intermediate data compared to other algorithms.

	FSM(300)	FSM(500)	FSM(1000)	3-Motif	4-Motif	5-Clique
CiteSeer	129	110	76	83	1914	26
MiCo	2388	2366	2285	1206	12408	6968
Patents	1234	1151	936	110	2791	275
UK	1367	2379	1461	1001	8914	7231

Table 4.6: Ratios between the final disk usage and original graph size (in the binary format).

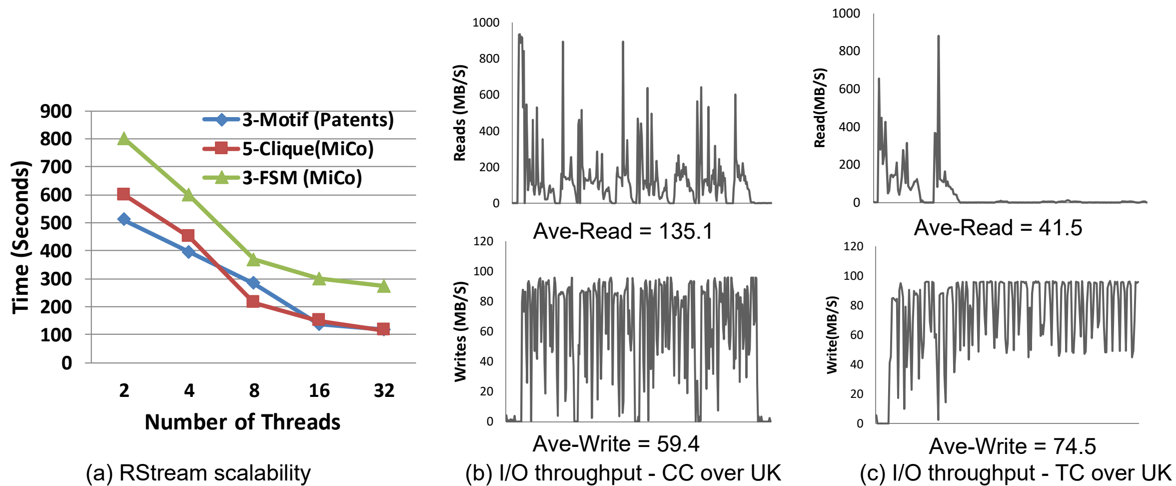


Figure 4.12: RStream's scalability (a), I/O throughput when running CC over UK (b), and I/O throughput when running TC over UK (c). I/O was measured with `iostat`.

Scalability and I/O Figure 4.12(a) shows RStream's running time for varying numbers of threads. In general, RStream scales with the number of threads. However, RStream's scalability decreases when the number of threads exceeds 8 because the disk bandwidth was almost saturated when 8 threads were used.

To understand how RStream performs for mining and computation algorithms, Figure 4.12(b) and (c) depict RStream's I/O throughput for a computation program (CC) and a mining program (TC), respectively. For CC, we monitored I/O in a full scatter-gather-apply iteration, while for TC, our measurement covered the full cycle of a join – loading the edge table, streaming in update tuples, performing joining, and writing back to the update table.

The file system cache was flushed during monitoring. Note that the high read throughput (*e.g.* 800+MB/s) achieved by RStream was primarily due to data stripped across the SSDs.

These two plots reveal the differences of these two types of algorithms: computation algorithms such as CC are dominated by I/O — *e.g.* disk reads/writes occur throughout the iteration. By contrast, relational joins in the mining algorithms such as TC are more compute-intensive, as most of the reads occur in an early stage of the join and the rest of the time is all spent on the in-memory computation (of joining and aggregation). For TC, writes still scatter all over the window due to the producer-consumer model used in RStream — the number of consumer threads is often small and hence many of the disk writes overlap with the computation.

4.5 Summary and Interpretation

This chapter presents RStream, the first single-machine, out-of-core graph mining system that leverages disk support to store intermediate data. At its core are two innovations: (1) a rich programming model that exposes relational algebra for developers to express a wide variety of mining tasks; and (2) a runtime engine that implements relational algebra efficiently with *tuple streaming*. Our experimental results demonstrate that RStream can be more efficient than state-of-the-art distributed mining systems. We hope that these promising results will encourage future work that builds disk-based systems to scale expensive mining algorithms.

CHAPTER 5

Related Work

5.1 Single-Machine Graph Computation Systems

Single-machine graph computation systems [88, 126, 177, 153, 96, 173, 152, 73, 148, 101, 17] have become popular as they do not need expensive computing resources and free developers from managing clusters and developing/maintaining distributed programs. State-of-the-art single-machine systems include Ligra [135], Galois [110], GraphChi [88], X-Stream [126], GridGraph [177], GraphQ [153], MMap [96], FlashGraph [173], TurboGraph [73], Mosaic [101], and Graspan [152].

Ligra [135] provides a shared memory abstraction for vertex algorithms. The abstraction is suitable for graph traversal. Galois [110] is a shared-memory implementation of graph DSLs on a generalized Galois system, which has been shown to outperform their original implementations. GRACE [150], a shared-memory system, processes graphs based on message passing and supports asynchronous execution by using stale messages.

Efforts have been made to improve scalability for systems over semi-external memory and SSDs. GraphChi [88] uses shards and a parallel sliding algorithm to reduce disk I/O for out-of-core graph processing. Bishard Parallel Processor [106] reduces non-sequential I/O by using separate shards to contain incoming and outgoing edges. X-Stream [126] uses an edge-centric approach in order to minimize random disk accesses. GridGraph [177] uses partitioned vertex chunks and edge blocks as well as a dual sliding window algorithm to process graphs residing on disks. Vora *et al.* from [148] reduces disk I/O using dynamic shards.

FlashGraph [173] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. It is built on the assumption that all vertices can be held in memory and a high-speed user-space file system for SSD arrays is available to merge I/O requests to page requests. TurboGraph [73] is an out-of-core engine for graph database to process graphs using SSDs. Pearce *et al.* [114] uses an asynchronous approach to execute graph traversal algorithms with semi-external memory. It utilizes in-memory prioritized visitor queues to execute algorithms like breadth-first search and shortest paths.

5.2 Distributed Graph Computation Systems

Google's Pregel [102] provides a synchronous vertex centric framework for large scale graph processing. Many other distributed systems [102, 98, 67, 51, 125, 52, 176, 167, 132, 147, 105, 157, 45, 146] have been developed on top of the same graph-parallel abstraction.

GraphLab [98] is a framework for distributed asynchronous execution of machine learning and data mining algorithms on graphs. PowerGraph [67] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. Cyclops [51] provides a distributed immutable view, granting vertices read-only accesses to their neighbors and allowing unidirectional communication from master vertices to their replicas. Chaos [125] utilizes disk space on multiple machines to scale graph processing. PowerLira [52] is a system that dynamically applies different computation and partitioning strategies for different vertices. Gemini [176] is a distributed system that adapts Ligra's hybrid push-pull computation model to a distributed form, facilitating efficient vertex-centric data update and message passing. Cube [167] uses a 3D partitioning strategy to reduce network traffic for certain machine learning and data mining problems. KickStarter [147] and Naiad [105] are systems that deal with streaming graphs.

GraphX [68] is a distributed graph system built on top of Spark, which is a general-purpose dataflow framework. GraphX provides a middle layer that offers a graph abstraction and "think like a vertex" interface for graph computation using low-level dataflow operators

such as join and group-by available in Spark. GraphX's design goal is completely opposite to that of RStream — GraphX aims to *hide* the relational representation of data and operations in the underlying dataflow system to provide a user-familiar graph computation interface while RStream aims to *expose* relational representation of data and operations over the underlying graph engine to enable the expression and processing of graph mining algorithms that focus on patterns and structures.

5.3 Approximate Queries

There is a vast body of work [74, 79, 50] on providing approximate answers to relational queries. These techniques use synopses like samples [74], histograms [79], and wavelets [50] to efficiently answer database queries. However, they have limited applicability to graph queries. Graph compression/clustering/summarization [107, 175, 63, 145, 64] has been extensively studied in the database community. These techniques focus on (lossy and lossless) algorithms to summarize the input graph so that graph queries can be answered efficiently on the summary graph. Unlike the graph compression techniques that trade off graph accuracy for efficiency, GraphQ never answers queries over a summary graph, but instead, it only uses the summary graph to rule out infeasible solutions and always resorts to the concrete graph to find a solution. In addition, the graphs used to evaluate the aforementioned systems are relatively small—they only have a few hundreds of vertices and edges, which can be easily loaded into memory. In comparison, the graphs GraphQ analyzes are at the scale of several billions of edges and cannot be entirely loaded into memory.

5.4 Static Bug Finding

Static analysis has been used extensively in the systems community to detect bugs [65, 58, 162, 154, 108, 66, 62, 56, 34, 35, 47, 61, 113, 44, 71, 18, 128, 127, 93, 94, 4] and security vulnerabilities [48, 46, 81]. Engler et al. [62] use a set of nine checkers to empirically study bugs in OS kernels. Palix et al. [113] implemented the same checkers using Coccinelle [111].

Commercial static checkers [5, 6, 8, 7] are also available for finding bugs and security problems. Most of these checkers are based on pattern matching. Despite their commendable bug finding efforts, false positive and negatives are inherent with these checkers.

Interprocedural analyses such as pointer and dataflow analysis can significantly improve the effectiveness of the checkers, but their implementations are often not scalable. There exists a body of work that makes program analysis declarative [156, 41] — analysis designers specify rules in Datalog and these rules are automatically translated into analysis implementations. However, the existing Datalog engines perform generic table joining and do not support disk-based computation on a single machine. While declarative analyses reduce the development effort, they still suffer from scalability issues. For example, although the pointer analysis from Whaley et al. [156] can scale to reasonably large Java programs (*e.g.*, using BDD), it only clones pointer variables, not objects. Furthermore, there is no evidence that they can perform fully context-sensitive analyses on codebases as large as the Linux kernel on a commodity PC.

5.5 Grammar-guided Reachability

There is a large body of work that can be formulated as a context-free language (CFL) reachability problem [163]. CFL-recognition was first studied by Yannakakis [163] for Datalog query evaluation. Work by Reps *et al.* [119, 117, 77, 118, 120] proposes to model realizable paths using a context-free language that treats method calls and returns as pairs of balanced parentheses. CFL-reachability can be used to formulate a variety of static analyses, such as polymorphic flow analysis [116], shape analysis [121], points-to and alias analysis [142, 139, 174, 140, 171, 160, 158, 169, 36], and information flow analysis [97]. The works in [84, 104, 85] study the connection between CFL-reachability and set-constraints, show the similarity between the two problems, and provide implementation strategies for problems that can be formulated in this manner. Kodumal *et al.* [85] extend set constraints to express analyses involving one context-free and any number of regular reachability properties. CFL-reachability has also been investigated in the context of recursive state machines [22],

streaming XML [21], and pushdown languages [23]. Recent work uses CFL-reachability to formulate pointer and alias analysis [142, 139, 174, 140, 171, 158, 160, 169, 36, 170]. and specification inference [36].

5.6 Distributed Mining Systems

Arabesque [144] is a distributed system designed to support mining algorithms. Arabesque presents to the developer an "embedding" view. Arabesque enumerates all possible embeddings with increasing sizes and the developer processes each embedding with a filter-process programming model. ScaleMine [13] is a parallel frequent subgraph mining system that contains two phases. The first phase computes an approximate solution by quickly identifying subgraphs that are frequent with high probability and collecting various statistics. The second phase computes the exact solution by using the results of the approximation to prune the search space and achieve load balancing. DistGraph [141] is an MPI-based distributed mining system that uses a set of optimizations and efficient operations to minimize communication costs. With these optimizations, DistGraph scales to billion-edge graphs with 2048 IBM BlueGene/Q nodes. G-thinker [159] is another distributed mining system that provides an intuitive graph-exploration API and a runtime engine. However, G-thinker does not support FSM and Motif-counting, which are arguably the most important mining algorithms. In addition, G-thinker's implementation is not publicly available.

5.7 Specialized Graph Mining Algorithms

gSpan [161] is an efficient frequent subgraph mining algorithm designed for mining multiple input graphs. Michihiro *et al.* [86] uses an anti-monotonic definition of support based on the maximal independent set to find edge-disjoint embeddings. GraMi [59] is an effective method for mining large input graph. Ribeiro *et al.* [122] proposes an approach for counting frequencies of motifs [115]. Maximal clique is a well-studied problem. There exist a lot of approaches to this problem, among which work from Bron-Kerbosch [43] has the highest

efficiency. Recently, a body of algorithms have been developed to leverage parallel [57, 25, 130, 136], distributed systems (such as Map/Reduce) [75, 37, 95, 100, 149, 15, 78, 172, 33], or GPUs [83].

5.8 Datalog Engines

There exists a great deal of work that aims to improve efficiency and scalability for Datalog engines [26, 90, 151, 124, 133, 103]. These existing graph computation and Datalog systems are orthogonal to our work because none of them could support full graph mining. LogicBlox [26] is a system designed to reduce the complexity of software development for modern applications. It provides a LogiQL language, a unified and declarative language based on Datalog, for developers to express relations and constraints. SociaLite [90] is a Datalog engine designed for social network analysis. SociaLite programs are evaluated by parallel workers that use message passing to communicate.

Myria [151] provides runtime support for Datalog evaluation using a pipelined, parallel, distributed execution engine that evaluates a graph of operators. Datasets are sharded and stored in PostgreSQL instances at worker nodes. Both SociaLite and Myria support monotonic aggregation inside recursion using aggregate semantics based on the lattice-semantics of Ross and Sagiv [124]. BigDatalog [133] is a distributed Datalog engine built on top of Spark. It bases its monotonic aggregate (operational and declarative) semantics on work [103] that uses monotonic *w.r.t.* set-containment semantics. While RStream takes inspiration from Datalog, our experimental results show that RStream is much more efficient than existing Datalog engines on graph mining workloads.

5.9 Dataflow Systems

Many dataflow systems [166, 24, 19, 40] were developed. Systems such as Spark [166] and Asterix [20] provide relational operations for dataset transformations. While RStream takes inspiration from these systems, it is built specifically for graph mining, and thus has to

overcome unique challenges that do not exist in existing systems.

At first glance, RStream's GRAS model is similar to a chain of MapReduce tasks — *e.g.* the input data first gets shuffled into streaming partitions; relational expressions are next applied and the generated data is re-shuffled before the next relational phase comes. The key difference between these two model lies in the semantics — the GRAS abstraction that we built enables developers to easily develop and reason about mining algorithms by composing structures of smaller sizes into large sizes, while generic MapReduce tasks do not have any semantics. Join is a critical relational operation that has been extensively studied in the database community [14, 109, 29, 28]. While there exist many efficient join implementations such as worst-case optimal join [109], RStream is largely orthogonal to these prior works — future work could improve RStream with a more efficient join implementation.

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

Graph analytical problems have been extensively studied in the past decade due to their importance in machine learning, web application and social media. Practical solutions have been implemented in a wide variety of graph analytical systems, most of which are distributed frameworks. However, most of the users are faced with the challenge of managing a cluster, which involves tasks such as data partitioning and fault tolerance. In addition, distributed graph systems usually suffer from large startup and communication overhead, as well as poor load balancing.

In this dissertation, we have presented a set of single-machine graph systems which can be more efficient than the state-of-art distribute graph systems. In Chapter 2, we proposed GraphQ, a graph query system based on abstraction refinement. GraphQ divides a graph into partitions and merges them with the guidance from a flexible programming model. An abstraction graph is used to quickly rule out infeasible solutions and identify mergeable partitions. GraphQ uses the memory capacity as a budget and tries its best to find solutions before exhausting the memory. Experiments show GraphQ can answer queries in graphs 4-6 times bigger than the memory capacity, only in several seconds to minutes.

In Chapter 3, we presented Graspan, a single-machine graph system which turns sophisticated code analysis into Big Graph analytics and leverages novel graph processing techniques to solve traditional programming language problem. We implement context-sensitive pointer/alias and dataflow analyses on Graspan. An evaluation of these analyses on large codebases such as Linux shows that their Graspan implementations scale to millions of lines

of code and are much simpler than their original implementations.

In Chapter 5, we discussed the development of RStream, a single-machine graph mining system which leverages disk support to store intermediate data. RStream employs a new GRAS programming model that uses a combination of GAS and relational algebra to support a wide variety of mining algorithms. At the low level, RStream leverages tuple streaming to efficiently implement relational operations. Our experimental results demonstrate that RStream can be more efficient than state-of-the-art distributed mining systems.

6.2 Future Work

Systemizing static code analysis Our work Graspan is the first attempt to turn sophisticated code analysis into scalable Big Graph analytics. Future research may consider how to develop similar Big Data systems to scale a broader set of static analyses, such as verification, model checking, or bug detection to large systems. We hope that with the help of systems support, these static analysis techniques could be widely adopted in industry to help design reliable and robust software applications.

Approximate graph mining RStream is the first single-machine, out-of-core graph mining system. In many graph mining tasks, it is not always necessary to compute the exact answer. For example, for a user who wants to find top 10 frequent subgraph patterns, it is sufficient to provide an approximate answer instead of an exact result. Since the amount of intermediate data for a typical mining algorithm grows exponentially with the size of the graph, one potential research direction is to investigate approximate graph mining, which has a potential to significantly reduce the amount of intermediate data generated, thereby offering an efficient and scalable solution for expensive mining tasks.

Streaming graph mining A limitation of RStream is that it currently assumes a static graph and does not deal with graph updates without restarting the computation. Hence, it cannot be used for interactive mining tasks at this moment. Developing systems to efficiently

process streaming graphs has gained increasing popularity, since real-world graphs are changing continuously. Future research may consider how to leverage incremental computation to support streaming graph mining.

REFERENCES

[1] Orkut social network. http://snap.stanford.edu/data/com-Orkut.html.

[2] The neo4j graph database. http://neo4j.com/, 2014.

[3] The Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/, 2014.

[4] The findbugs Java static checker. http://findbugs.sourceforge.net/, 2015.

[5] The Coverity code checker. http://www.coverity.com/, 2016.

[6] The GrammaTech CodeSonar static checker, 2016.

[7] The HP Fortify static checker, 2016.

[8] The KlocWork static checker, 2016.

[9] The LLVMLinux project. http://llvm.linuxfoundation.org/, 2016.

[10] The LogicBlox Datalog engine. http://www.logicblox.com/, 2016.

[11] Personal communication with John Criswell, 2016.

[12] Bliss: A tool for computing automorphism groups and canonical labelings of graphs. http://www.tcs.hut.fi/Software/bliss/, 2017.

[13] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. ScaleMine: Scalable parallel frequent subgraph mining in a single large graph. In *SC*, pages 61:1–61:12, 2016.

[14] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[15] Ashraf Aboulnaga, Jingen Xiang, and Cong Guo. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.

[16] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE*, pages 3–14.

[17] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *USENIX ATC*, pages 125–137, 2017.

[18] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. In *PASTE*, pages 43–48, 2007.

[19] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Nicola Onose, Pouria Pirzadeh, Rares Vernica, and Jian Wen. ASTERIX: An open source system for "big data" management and analysis (demo). *Proc. VLDB Endow.*, 5(12):1898–1901, 2012.

[20] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. Storage management in asterixdb. *Proc. VLDB Endow.*, 7(10):841–852, 2014.

[21] Rajeev Alur. Marrying words and trees. In *PODS*, pages 233–242, 2007.

[22] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[23] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

[24] Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`.

[25] D. O. Aparicio, P. M. P. Ribeiro, and F. M. A. d. Silva. Parallel subgraph counting for multicore architectures. In *IPDPS*, pages 34–41, 2014.

[26] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.

[27] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29(10):996–1000, 1986.

[28] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748, 2008.

[29] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[30] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *STOC*, pages 171–183, 1983.

[31] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD*, pages 44–54, 2006.

[32] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[33] Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest subgraph in streaming and MapReduce. *Proc. VLDB Endow.*, 5(5):454–465, 2012.

[34] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.

[35] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[36] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

[37] M. A. Bhuiyan and M. Al Hasan. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):608–620, 2015.

[38] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[39] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

[40] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[41] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

[42] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In Takashi Washio, Einoshin Suzuki, Kai Ming Ting, and Akihiro Inokuchi, editors, *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD '08)*, pages 858–863, 2008.

[43] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, 1973.

[44] Fraser Brown, Andres Notzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. In *ASPLOS*, pages 143–157, 2016.

[45] Yingyi Bu, Vinayak Borkar, Michael J. Carey, and Tyson Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proc. VLDB Endow.*, 7, 2015.

[46] Suhabe Bugrara and Alex Aiken. Verifying the safety of user pointer dereferences. In *IEEE S&P*, pages 325–338, 2008.

[47] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[48] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[49] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, pages 169–180, 2012.

[50] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3):199–223, 2001.

[51] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, pages 215–226, 2014.

[52] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[53] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[54] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[55] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[56] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

[57] Giuseppe Di Fatta and Michael R. Berthold. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Trans. Parallel Distrib. Syst.*, 17(8):773–785, 2006.

[58] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, pages 12–22, 2004.

[59] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. GraMi: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, 2014.

[60] Dawson Engler. Making finite verification of raw C code easier than writing a test case. In *RV*. Invited talk.

[61] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, pages 1–1, 2000.

[62] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.

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

[64] Wenfei Fan, Xin Wang, and Yinghui Wu. Querying big graphs within bounded resources. In *SIGMOD*, pages 301–312, 2014.

[65] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, 2006.

[66] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, 1999.

[67] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[68] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[69] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[70] Bronwyn H. Hall, Adam B. Jaffe, and Manuel Trajtenberg. The NBER patent citation data file: Lessons, insights and methodological tools. Technical Report 8498, National Bureau of Economic Research, 2001.

[71] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[72] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *EuroSys*, pages 1:1–1:14, 2014.

[73] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[74] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[75] Steven Hill, Bismita Srichandan, and Rajshekhar Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *BCB*, pages 661–666, 2012.

[76] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.

[77] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *FSE*, pages 104–115, 1995.

[78] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. Large scale real-time ridesharing with service guarantee on road networks. *Proc. VLDB Endow.*, 7(14):2017–2028, 2014.

[79] Yannis E. Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

[80] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(2-3):273–281, 1986.

[81] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security*, pages 9–9, 2004.

[82] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434, 2013.

[83] Robest Kessl, Nilothpal Talukder, Pranay Anchuri, and Mohammed J. Zaki. Parallel graph mining with gpus. In *BIGMINE*, pages 1–16, 2014.

[84] John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.

[85] John Kodumal and Alex Aiken. Regularly annotated set constraints. In *PLDI*, pages 331–341, 2007.

[86] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov.*, 11(3):243–271, November 2005.

[87] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[88] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[89] Aapo Kyrola and Carlos Guestrin. GraphChi-DB: Simple design for a scalable graph database system – on just a PC. `http://arxiv.org/pdf/1403.0701v1.pdf`.

[90] Monica S. Lam, Stephen Guo, and Jiwon Seo. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

[91] Butler W. Lampson. Hints for computer system design. In *SOSP*, pages 33–48, 1983.

[92] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

[93] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 20–20, 2004.

[94] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, pages 306–315, 2005.

[95] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in MapReduce. In *ICDE*, pages 844–855, 2014.

[96] Zhiyuan Lin, Minsuk Kahng, Kaeser Md. Sabrin, Duen Horng (Polo) Chau, Ho Lee, , and U Kang. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData*, pages 159–164, 2014.

[97] Ying Liu and Ana Milanova. Static analysis for inference of explicit information flow. In *PASTE*, pages 50–56, 2008.

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

[99] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[100] W. Lu, G. Chen, A. K. H. Tung, and F. Zhao. Efficiently extracting frequent subgraphs using MapReduce. In *Big Data*, pages 639–647, 2013.

[101] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *EuroSys*, pages 527–543, 2017.

[102] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[103] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. Extending the power of datalog recursion. *The VLDB Journal*, 22(4):471–493, August 2013.

[104] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[105] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *SOSP*, pages 439–455, 2013.

[106] Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *Journal of Multimedia & Ubiquitous Engineering*, 9(2):199–212, 2014.

[107] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *SIGMOD*, pages 419–432, 2008.

[108] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[109] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[110] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[111] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys*, pages 247–260, 2008.

[112] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[113] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *ASPLOS*, pages 305–318, 2011.

[114] Roger Pearce, Maya Gokhale, and Nancy M Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC*, pages 1–11, 2010.

[115] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[116] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[117] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[118] Thomas Reps. Solving demand versions of interprocedural analysis problems. In *CC*, pages 389–403, 1994.

[119] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[120] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *FSE*, pages 11–20, 1994.

[121] Tom Reps. Shape analysis as a generalized path problem. In *PEPM*, pages 1–11, 1995.

[122] Pedro Ribeiro and Fernando Silva. G-Tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov.*, 28(2):337–377, 2014.

[123] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, pages 184–191, 2004.

[124] Kenneth A. Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *PODS*, pages 114–126, 1992.

[125] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[126] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[127] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *PLDI*, pages 270–280, 2009.

[128] Cindy Rubio-González and Ben Liblit. Defective error/pointer interactions in the linux kernel. In *ISSTA*, pages 111–121, 2011.

[129] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[130] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD*, pages 625–636, 2014.

[131] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[132] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *USENIX ATC*, pages 317–332, 2016.

[133] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD*, pages 1135–1149, 2016.

[134] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE*, pages 867–878, 2015.

[135] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[136] George M. Slota and Kamesh Madduri. Parallel color-coding. *Parallel Comput.*, 47(C):51–69, 2015.

[137] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

[138] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *PLDI*, pages 485–495, 2014.

[139] Manu Sridharan and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[140] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

[141] Nilothpal Talukder and Mohammed J. Zaki. A distributed approach for graph mining in massive networks. *Data Mining and Knowledge Discovery: Special Issue on ECML/PKDD 2016 Journal Track Papers*, 30(5):1024–1052, 2016.

[142] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95, 2015.

[143] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining - extended version. *ArXiv e-prints*, October 2015.

[144] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A system for distributed graph mining. In *SOSP*, pages 425–440, 2015.

[145] Hannu Toivonen, Fang Zhou, Aleksi Hartikainen, and Atte Hinkka. Compression of weighted graphs. In *KDD*, pages 965–973, 2011.

[146] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic analysis of evolving graphs. *ACM Trans. Archit. Code Optim.*, 13(4):32:1–32:27, 2016.

[147] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *ASPLOS*, 2017.

[148] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC*, pages 507–522, 2016.

[149] Chao Wang and Srinivasan Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *ICS*, pages 31–40, 2004.

[150] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[151] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[152] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*, pages 389–404, 2017.

[153] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015.

[154] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, pages 260–275, 2013.

[155] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[156] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[157] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

[158] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[159] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. G-thinker: Big graph mining made easier and faster. *CoRR*, abs/1709.03110, 2017.

[160] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

[161] Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–, 2002.

[162] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *OSDI*, pages 10–10, 2006.

[163] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *PODS*, pages 230–242, 1990.

[164] Daniel M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Inf.*, 30(4):369–384, 1993.

[165] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *PLDI*, pages 91–103, 1999.

[166] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

[167] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, pages 285–300, 2016.

[168] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *ASPLOS*, pages 608–621, 2018.

[169] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, pages 435–446, 2013.

[170] Qirun Zhang and Zhendong Su. Context-sensitive data dependence analysis via linear conjunctive language reachability. In *POPL*, pages 344–358, 2017.

[171] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for C. In *OOPSLA*, pages 829–845, 2014.

[172] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. SAHAD: Subgraph analysis in massive networks using hadoop. In *IPDPS*, pages 390–401, 2012.

[173] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST*, pages 45–58, 2015.

[174] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 2008.

[175] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. Graph clustering based on structural/attribute similarities. *Proc. VLDB Endow.*, 2(1):718–729, 2009.

[176] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.

[177] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC*, 2015.

[178] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.