Scalable Systems for Large Scale Dynamic Connected Data Processing

Anand Padmanabha Iyer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-178
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-178.html

December 20, 2019

Scalable Systems for Large Scale Dynamic Connected Data Processing

by

Anand Padmanabha Iyer

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Scott Shenker
Professor Michael J. Franklin
Professor Joshua Bloom

Fall 2019

Scalable Systems for Large Scale Dynamic Connected Data Processing

Abstract

Scalable Systems for Large Scale Dynamic Connected Data Processing

by

Anand Padmanabha Iyer

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

As the proliferation of sensors rapidly make the Internet-of-Things (IoT) a reality, the devices and sensors in this ecosystem—such as smartphones, video cameras, home automation systems, and autonomous vehicles—constantly map out the real-world producing unprecedented amounts of *dynamic, connected data* that captures complex and diverse relations. Unfortunately, existing big data processing and machine learning frameworks are ill-suited for analyzing such dynamic connected data and face several challenges when employed for this purpose.

This dissertation focuses on the design and implementation of scalable systems for dynamic connected data processing. We discuss simple abstractions that make it easy to operate on such data, efficient data structures for state management, and computation models that reduce redundant work. We also describe how bridging theory and practice with algorithms and techniques that leverage approximation and streaming theory can significantly speed up connected data computations. Leveraging these, the systems described in this dissertation achieve more than an order of magnitude improvement over the state-of-the-art.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to my advisor, Ion Stoica, for guiding me throughout my graduate career at Berkeley. Ion taught me several valuable things; some of them include the value of simplicity in all aspects of research, going deep into a problem to find the *nugget* and the importance of focusing on one thing at a time. He was available whenever I wanted to meet to discuss research or get advice in general, no matter how short of a notice I gave; I'm not sure how he managed to do that with his extremely packed daily schedule.

This dissertation is the result of many successful collaborations. Chapter 3 was joint work with Qifan Pu, Kishan Patel, Joey Gonzalez and Ion Stoica [131]. Chapter 4 builds on work done with Alan Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman and Ion Stoica [88, 89]. Chapter 5 contains material from the collaboration with Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker and Ion Stoica [130]. Chapter 6 was joint work with Aurojit Panda, Mosharaf Chowdhury, Aditya Akella, Scott Shenker and Ion Stoica [90]. Finally, chapter 7 was the result of collaboration with Li Erran Li, Mosharaf Chowdhury and Ion Stoica [85].

I'm thankful to my dissertation committee members, Josh Bloom, Mike Franklin and Scott Shenker. I've always felt that Scott was my unofficial co-advisor. I've enjoyed every single conversation with him; no matter the topic—research, teaching or whether the Indian version of Dairy Milk is really better than its British counterpart. Scott's passion for teaching and elegance in research has had deep influence in shaping my views. Mike and Josh provided excellent feedback that helped tremendously in the positioning of the research presented in this dissertation.

My colleagues at the AMPLab and later at the RISELab—in no particular order David Zats, Aurojit Panda, Shivaram Venkataraman, Sameer Agarwal, Mosharaf Chowdhury, Antonio Blanca, Anurag Khandelwal, Neeraja Yadwadkar, Qifan Pu, Colin Scott and several others—made Berkeley really fun and an unforgettable experience. None of the work in AMP or RISE would be possible without the amazing support staff: Kattt, Boban, Jon, Shane to name a few.

I'm incredibly fortunate to have really loving and understanding family and in-laws. My mother, father and brother shielded me from many storms and allowed me to pursue my dreams. My in-laws never hesitated in providing a helping hand, and always encouraged and supported me throughout.

Lastly, but most importantly, I'm really indebted to my wife Sri, who has been my rock-star supporter. She made numerous sacrifices and stood by me throughout this journey; without her support I would have never reached this far. She really deserves this degree more than me.

Chapter 1

Introduction

The availability of cheap data and cheap compute has made big data analytics mainstream. However, recently there has been a paradigm shift in how we produce and process data. As the proliferation of sensors rapidly make the Internet-of-Things (IoT) a reality, the devices and sensors in this ecosystem—such as smartphones, video cameras, home automation systems and autonomous vehicles—constantly map out the real-world producing unprecedented amounts of *connected data that captures complex and diverse relations*. When coupled with the significant leap Artificial Intelligence (AI) has made in key domains where these sensors are the major source of data, there is an increasing demand in systems that can ingest such *live, dynamic, connected* data, analyze them and produce *low-latency* decisions. These systems have the potential to shape the next generation computation stack and further research in the fields of networked systems, AI & machine learning (ML) and mobile computing.

This dissertation focuses on the core problems towards realizing such infrastructure by designing scalable systems. These systems propose: (1) simple abstractions that make it easy to operate on dynamic connected data, efficient datastructures to ingest and compactly store the data and computation state, and computational models that utilize incremental approaches to reduce redundant work (e.g., Tegra in chapter 3) (2) bridging theory and practice with algorithms and techniques that leverage approximation, streaming theory and machine learning to significantly speed up computations by trading off accuracy (e.g., ASAP in chapter 4), and (3) methods for more accurate application of ML tasks on live dynamic data (e.g., CellScope in chapter 7).

We begin with an introduction to the concept of connected data, explain the necessity for processing them in an efficient manner and the describe the difficulties in doing so using real-world examples.

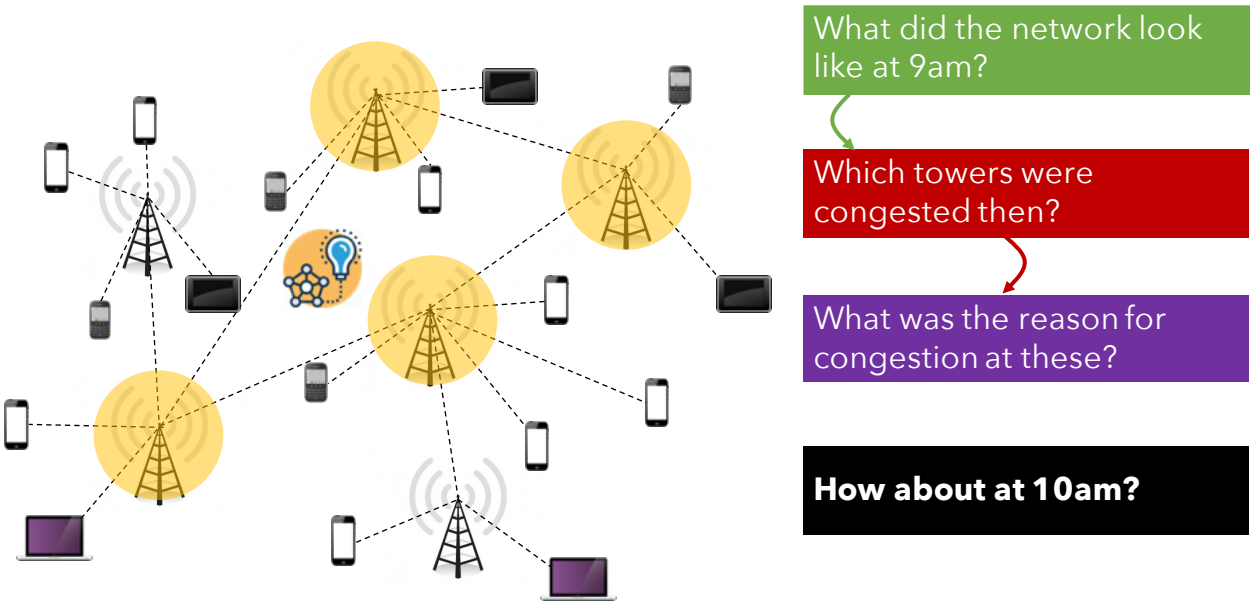


Figure 1.1: Alex, a network administrator, diagnoses issues using data collected in the network.

1.1 Motivating Examples

In this section we describe two scenarios, which we will use as running examples throughout this dissertation, to motivate the need for dynamic connected data processing. While the names of the persons used in these examples are fictional, the scenarios depict real world problems faced by enterprise organizations today.

1.1.1 Alex Diagnoses Network Issues

Alex works as a network administrator at a large cellular network operator in the United States. Alex's job is to manage several thousands of wireless base stations, deployed across a large geographic region, that serves millions of users connect to the Internet every day. Whenever problems occur, and they do occur often in today's cellular networks, Alex is tasked with finding the reason for the issue and fix them. Monitoring and managing network infrastructure is highly impactful problem for network operators today, with up to USD 22 billion spent by a single top-tier operator every year in network management and operation costs. For instance, let's assume that Alex is trying to find the answer to the question, *"What is the reason for poor download throughput for (several) users at 9:00am?"*.

Much like any data driven company today, Alex's company collects extensive data from their network. The workflow is depicted in fig. 1.1. Alex might start with asking

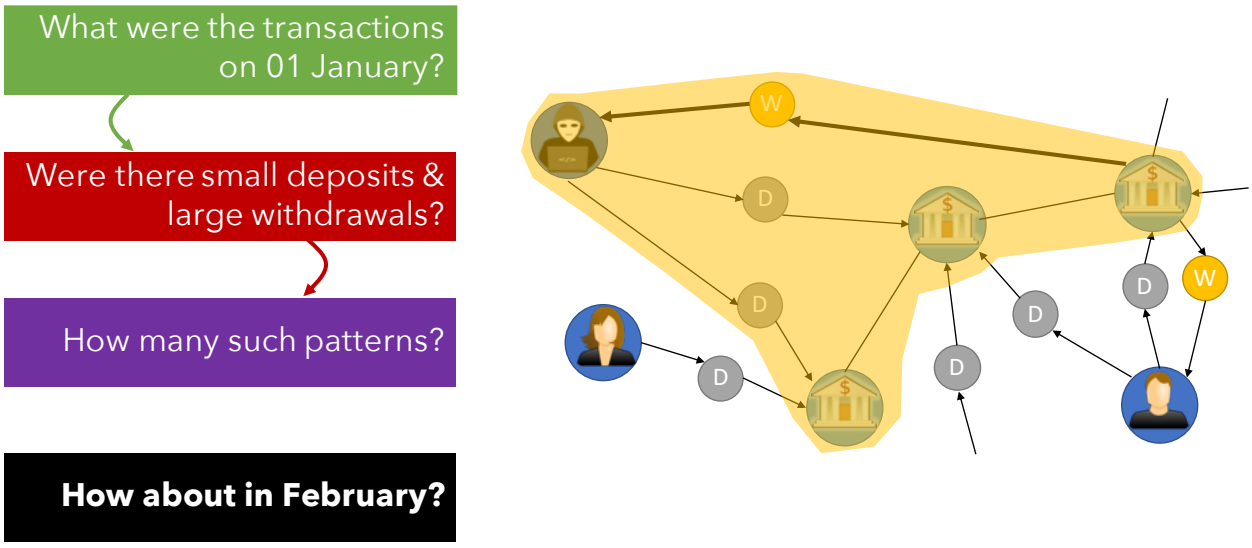


Figure 1.2: Taylor, a financial analyst, uses transaction data to train a model to detect money laundering.

"How did the network look like at 9am?", when the problem actually happened. The query returns something similar to fig. 1.1 where there are several base stations serving many users. Alex doubts congestion as the cuase for low throughput, so the next query is *"Which towers were congested at this point?"* that returns a few towers from the original answer. Then to understand the reason, Alex might run a few machine learning algorithms on the data. Now Alex has to confirm that the findings are indeed correct. To do so, Alex asks *"How about at 10am?"* meaning to repeat the entire analysis again, but now on a different subset of the data.

In order to execute these queries and machine learning algorithms on data, Alex uses handwritten scripts to parse and represent the data in a format that is amenable to the analysis, and open source tools such as Scikit Learn [155] or Tensorflow [3] to do the learning. Unfortunately, Alex faces two issues today. First, the tools available are unable to handle the *dynamicity* of the data. The network which Alex manages generates several terabytes of data every day, and wading through this massive amount of *real-time* data is difficult. Second, the tools which Alex uses for analysis do not scale to large datasets, majorly due to the nature of the queries. Thus, Alex spends a significant amount of effort today in doing such analysis.

1.1.2 Taylor Finds Financial Frauds

Taylor works as a financial analyst at one of the largest banks in the world. A major analysis involved in Taylor's daily job is in discovering financial frauds. A particular problem of interest for Taylor is money laundering, which accounts for over 200 billion USD every year, and thus an important charter for the bank.

Banks collect extensive information about financial exchanges, and a common approach to finding frauds is to train a machine learning algorithm on these financial transactions to learn a model for detecting fraudulent transaction, and then use this model in real-time transactions to mark suspicious ones. Figure 1.2 shows how Taylor does this today, and it starts by retrieving the transactions that took place around the time of known frauds. Once the data is retrieved, Taylor searches for transactions or batches of transactions where small amounts of deposits were made followed by large withdrawals, which is the distinguishing characteristic of classic money laundering. If there are such patterns in the data (there is one such occurrence in fig. 1.2), Taylor gets an estimate of how many such frauds occurred and retrieves a few of them. These retrieved "laundering patterns" are fed to a learning algorithm which learns a model to detect laundering. Taylor does this process in an ongoing fashion; as new laundering patterns pop-up the model needs to be trained to retain its performance.

Like Alex, Taylor uses a combination of tools to do this analysis; for instance, R [144] for quickly prototyping and testing some statistical methods, PyTorch [141], TensorFlow [3] or Python scripts to train the model, and NetworkX [127] or custom programs to discover the laundering patterns in the data. Similar to Alex's, Taylor faces the problems of *dynamicity* and *scalability*, albeit to a much higher extent. Due to the complexity of discovering patterns in the data, the analysis cannot even handle moderate sized data. Thus, it takes days, or even weeks for Taylor to run the analysis on the data collected by the bank.

1.1.3 Alex & Taylor Aren't Alone!

Alex and Taylor work with what we refer to in this dissertation as *connected data*. In simple terms, *connected data* is data comprising of entities and their relations. The relations can be explicitly specified (e.g., real-world entities with spatial relations [153] such as base stations and users in Alex's case, graphical models [101]) or learned (e.g., raw sensor data in deep learning tasks [27]). Such data has the power to capture diverse and complex relations, which could be immensely valuable in many areas including the potential to be the building block of future Artificial Intelligence systems [27].

Indeed, the lives of Alex and Taylor can become much more simpler by the availability of techniques and systems that can operate on large scale dynamic, connected data.

However, a key question that arises is if the problem of dynamic connected data processing is limited to Alex and Taylor's use-cases. The answer is a resounding no. Several emerging applications could benefit from scalable and efficient connected data processing systems. Connected vehicles [35] is an increasingly popular area of interest, both in the industry and academia. In connected vehicles, connected data processing systems can enable safety systems, such as using the sensor inputs from vehicles and users, combine them with the real-time path taken by vehicles to model impending accidents and provide warnings. Autonomous vehicles are undoubtedly the future of transportation, and we are making huge strides towards achieving this goal. However, managing fleets of autonomous vehicles in the real-world requires extremely robust and scalable traffic flow management systems that can avoid congestion and choke points. Connected data processing systems are likely the foundations in such systems. Finally, with sensors making their ways into our everyday equipment and home automation systems becoming more popular, we are no doubt going to see intelligent *smart cities* that would enhance our quality of life and the environment. These smart cities would require immense planning, design and ongoing optimizations for efficient operation, and connected data systems would be crucial in understanding how the different entities that constitute the smart city interact with each other and how to optimally actuate the different pieces for achieving the end goal. In short, as the era of Internet-of-Things become reality, the world is moving towards connected data, and systems that can efficient process large scale, dynamic connected data can shape the next generation computation stack.

1.2 Problems with Existing Systems

Over the past decade, growing data volumes have pushed the frontiers in large scale data processing and cloud computing. As a result, several cluster processing frameworks exist today that can scale out to a large number of machines and process tremendous amounts of data. The natural question is if these systems can help with dynamic connected data processing. Unfortunately, there are three main challenges that stand in the way of leveraging existing data processing systems for connected data processing.

1.2.1 Programmability

The first challenge is that of programmability. How do we allow end users to query dynamic connected data in a natural and intuitive way? Users like Alex and Taylor are familiar with the simple interface provided by existing Python based tools, and can benefit from similar (simple) interfaces for representing, accessing and manipulating

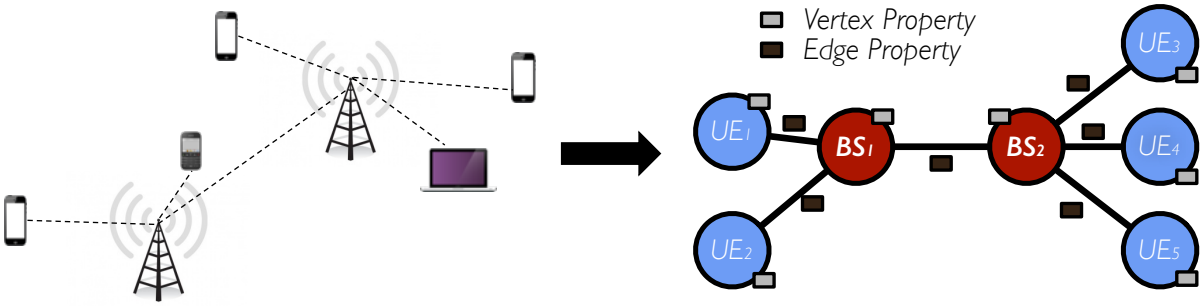


Figure 1.3: Representing connected data as property graphs.

or operating on the data they are interested in. However, no such facilities exist in today's big data frameworks. While these frameworks provide elegant primitives for unstructured data manipulation, the relation between the entities and the dynamic nature of these relations pose problems. The representation problem is fairly straightforward to address. At a given point in time, connected data depicts the state of entities and their relations. Such relations are naturally represented as *graphs*. For instance, Alex's network can be represented by a *property graph* (§2.2.1) where the entities in the network, such as users and base stations, are graph *nodes* as shown in fig. 1.3. We follow this natural representation format in this dissertation and thus *dynamic* connected data is represented as *dynamic* property graphs (§3.3.1). Unfortunately, efficient storage and operations on dynamic graphs are still open questions.

1.2.2 Storage

Connected data analysis is *data intensive*, as we saw with the examples of Alex and Taylor. In both these scenarios, data is generated in the order of several terabytes every day. The volume of data is only poised to increase with emerging Internet-of-Things applications, for instance, autonomous/connected vehicles are expected to generate over 4 terabytes of data per day by 2020 [83]. Thus, efficiently storing and retrieving large quantities of dynamic connected data is a huge challenge. This is because commonly used techniques in databases such as building indices for improving the access efficiency is not possible in connected data processing.

1.2.3 Performance

Extracting performance in the face of dynamicity forms the biggest challenge in connected data analysis. Most of connected data queries are both *interactive* and *exploratory*. They

are *interactive* because users like Alex and Taylor are firing these queries at a terminal and *waiting* for the results. The *exploratory* nature is because the queries are executed based on what the user sees; the answer to a query determines the next query. Thus, queries are executed in an *ad-hoc* fashion, and are not predetermined. In such scenarios, performance optimization techniques such as pre-processing and query specific caching do not help.

1.3 Solution Overview

This dissertation focuses on the core problems in realizing efficient dynamic connected data processing systems. Towards this goal, we design and implement the following systems:

Tegra focuses on the problem of *ad-hoc analytics* on dynamic connected data. It aims to solve the problems faced by Alex and Taylor and forms the foundation for the other systems. To do so, it represents dynamic connected data as dynamic/evolving property graphs, and proposes techniques for efficient storage and ad-hoc window operations on evolving graphs. It enables efficient access to the state of the graph at arbitrary windows, and significantly accelerates ad-hoc window queries by using a compact in-memory representation for both graph and intermediate computation state. For this, it leverages persistent datastructures to build a versioned, distributed graph state store, and couples it with an incremental computation model which can leverage these compact states. For users, it exposes these compact states using Timelapse, a natural abstraction. TEGRA significantly outperforms other systems (by up to 30×) for ad-hoc window operations.

ASAP specifically focuses on usecases similar to Taylor's roadblock, the intractability of scaling pattern discovery queries to larger datasets, and proposes a fast, approximate computation engine for graph pattern mining. It leverages state-of-the-art results in graph approximation theory, and extends it to general graph patterns in distributed settings. To enable the users to navigate the tradeoff between the result accuracy and latency, we propose a novel approach to build the Error-Latency Profile (ELP) for a given computation. ASAP outperforms existing exact pattern mining solutions by orders of magnitude. Further, it can scale to graphs with billions of edges without the need for large clusters.

GAP explores the question of applying approximation to the problem of iterative analytics on connected data and takes a first attempt at realizing approximate graph analytics engine. Here, we discuss how traditional approximate analytics techniques do not carry over to the graph usecase. Leveraging the characteristics of graph properties and algorithms, GAP proposes a graph sparsification technique, and a machine learning

based approach to choose the apt amount of sparsification required to meet a given budget.

Monarch attempts to enhance the connected data processing systems by looking at making them function when deployed in a geo-distributed fashion and thus focuses on the problem of efficient geo-distributed graph analytics. We find that optimizing the iterative processing style of graph-parallel systems is the key to achieving this goal rather than extending existing geo-distributed techniques to graph processing.

CellScope looks at the fundamental difficulties in making real-time decisions on real-time connected data. Such real-time decision tasks include simple reporting on data streams to sophisticated model building. We observe that the practicality of these analyses are impeded in several domains because they are faced with a fundamental trade-off between data collection latency and analysis accuracy. To solve this issue, we look at one particular domain, Alex's network performance diagnosis, to study the trade-off in detail and find that the trade-off can be resolved using two broad, general techniques: intelligent data grouping and task formulations that leverage domain characteristics. Based on this, CellScope applies a domain specific formulation and application of Multi-task Learning(MTL) to network performance analysis. It uses three techniques: feature engineering to transform raw data into effective features, a PCA inspired similarity metric to group data from geographically nearby base stations sharing performance commonalities, and a hybrid online-offline model for efficient model updates. We then generalize the techniques and show their efficacy in other domains.

1.4 Dissertation Plan

This dissertation is organized as follows. Chapter 2 provides background on large scale data processing. Chapter 3 describes the work on supporting ad-hoc analytics on dynamic connected data. We discuss Tegra here, which proposes efficient ways to store and perform ad-hoc window operations. In chapter 4, we focus on pattern mining on connected data, looking at problems such as Taylor's. We describe ASAP in this chapter, which is a fast, approximate pattern mining system that achieves several orders of magnitude improvement over the state of the art techniques. Chapter 5 attempts to extend the learning from ASAP to iterative analytics, and describes GAP, an approximate analytics engine. We discuss how connected data is naturally generated in a geo-distributed fashion and attempt to extend connected data processing systems to geo-distributed settings by describing Monarch in chapter 6. Chapter 7 looks at the fundamental trade-offs in making real-time decisions on dynamic connected data and describes CellScope, our solution to this problem. Finally, we conclude and discuss future work in chapter 8.

Chapter 2

Background

In this dissertation, we follow the natural representation format of viewing (dynamic) connected data as (dynamic) property graphs (§1.2.1). Thus, the systems we discuss in the following chapters build on the rich distributed graph processing literature. To aid the reader in following the rest of the dissertation, we begin with a brief background on large scale data processing systems, focusing on data-parallel and graph parallel systems.

2.1 Data Parallel Processing

Over the last decade, data parallel systems have gained popularity for processing large amounts of data. These systems were aimed at simplifying parallel computation over large amounts of *distributed* data. Typically, these systems expose simple abstractions and operators that developers use to achieve the desired processing goals, and internally manage the intricacies of efficiently running these operations on the data in *parallel*.

The most popular early proposal in this space is Google's MapReduce [52], which exposed just two operators—`map` and `reduce`—that could capture many *embarrassingly parallel* use-cases such as rebuilding Google's search indexes, the use-case it was originally intended for, and provided efficient execution in a fault-tolerant fashion. The tremendous popularity of MapReduce resulted in the emergence of a large number of systems [84, 190, 125, 193, 192, 21] and sparked research in many core areas such as scheduling, query optimization and programming models. The new generation engines, such as Naiad [125], Spark [192] and Flink [21], improved upon the original MapReduce proposal by incorporating new operators, declarative interfaces, better task execution plans and in-memory caching to reduce inefficiencies with intermediate data management.

Traditionally, most of the dataflow engines were targeted at processing batches of unstructured data. However, as new application scenarios emerged, they have increasingly

incorporated new functionalities such as the ability to do graph processing [115, 22, 71], stream processing [21, 13, 194] and machine learning [102, 162].

2.2 Graph Parallel Processing

We discuss graph parallel processing in this section, touching upon representation, abstractions and optimizations.

2.2.1 Property Graph Model

Graph processing systems typically represent graph structured data as a property graph [148], which associates user-defined properties with each vertex and edge. The properties can include meta-data (e.g., user profiles and time stamps) and program state (e.g., the number of neighbors, or rank of vertices). For example, fig. 1.3 shows Alex's network at a particular instance in time (say 9:00am) as a property graph. Here, the vertices (users and base stations) can hold properties such as the total data transferred and number of users active while the edges can be made to store properties specific to the node pairs such as signal quality, geographical distance and so on. This enables Alex to query and filter the network based on properties and thus create analysis specific instances of graphs as input to the next stage of processing.

Property graphs are logically represented in a distributed dataflow framework such as MapReduce [52] as a pair of vertex and edge property collections, where the collections contain the mapping between the vertex or edge and their properties. This enables the composition of graphs with other collections in the dataflow frameworks [71]. An alternative to property graphs for associating data with graph entities is the Resource Description Framework (RDF) format [148] which stores triplets of subject-predicate-object. However, compared to RDF, property graphs are considered to be a better format for storage and querying, especially for graph analytics.

2.2.2 Graph Parallel Abstractions

Most existing general purpose graph processing systems allow end-users to perform graph computations by exposing a *graph-parallel* abstraction. A graph-parallel abstraction consists of a *graph* and a *vertex-program*, provided by the user, which is executed in parallel on each vertex. The program running on individual vertex can interact with the vertex's *neighborhood*. This interaction between vertices is implemented using either shared state (e.g., GraphLab [111]) or message passing (e.g., Pregel [115]). Each vertex

program can read and modify its vertex property, and when all vertex programs vote to halt the program terminates.

Pregel

Pregel [115] proposes a *Bulk Synchronous Parallel (BSP)* message passing abstraction where all the vertex programs run simultaneously in a series of *super-steps*. In every super-step, each vertex program receives all messages from the previous super-step and sends messages to its neighbors in the next super-step. At the end of each super-step, a barrier is imposed in order to ensure that all the vertex programs finish processing messages before proceeding to the next. This ensures program correctness (at the cost of efficiency). A Pregel program terminates when there are no messages remaining to be sent and every vertex program has voted to halt. Pregel exposes several optimizations to improve the efficiency of the message passing, for instance, messages destined to the same vertex are combined using a commutative associative user-defined function.

GraphLab

GraphLab [111] proposes an *asynchronous distributed shared-memory* abstraction where the vertex programs have shared access to the distributed graph. Each vertex program access information about the current and adjacent vertices and edges directly and no message passing is involved. In this model, correctness of the program is ensured by the GraphLab system by serializing the program execution at neighbors. GraphLab eliminates messages and also synchronization and can thus achieve higher efficiency by isolating and handling data movements at the system level. However, this comes at the cost of an increased complexity, and the gains due to an asynchronous programming model are often offset by this additional complexity. Thus, a majority of distributed graph processing systems today adopt the bulk synchronous model.

GAS Decomposition

PowerGraph [70] observes that while Pregel and GraphLab differ in how they collect and distribute information, they share a common structure. Thus, it proposes the Gather-Apply-Scatter (GAS) decomposition to characterize this common structure and differentiate between vertex and edge specific computations. The GAS model captures the conceptual phases of the vertex program as shown in listing 2.1. In the GAS decomposition, a vertex program consists of three data parallel phases: a `gather` phase that collects information about adjacent vertices and edges and applies a function on them, the `apply` phase that uses the function's output to update the vertex, and the

```
def Gather(u, v) = Accum
def Apply(v, Accum) = v_new
def Scatter(v, j) = j_new, Accum
```

Listing 2.1: The Gather-Apply-Scatter (GAS) decomposition introduced in PowerGraph [70].

`scatter` phase that uses the new vertex value to update adjacent edges. The system executes these phases sequentially. Since the GAS decomposition leads to a pull based model of message computation, it enables vertex-cut partitioning, and improved work balance which are essential in achieving performance in real-world graphs which follow power-law distribution [70]. As a result, many popular open-source graph-processing frameworks (e.g., [71]) have adopted the GAS model.

Chapter 3

Ad-Hoc Analytics on Dynamic Connected Data

3.1 Introduction

In this chapter, we focus on the problem of *efficient ad-hoc window operations* on dynamic connected data represented as *evolving graphs*—the ability to perform ad-hoc queries on arbitrary time windows (i.e., segments in time) either in the past or in real-time. The motivating examples described in the previous chapter (§1.1) are indeed instances of such *exploratory* analysis.

Alex's transient failure diagnosis started by retrieving a *series of snapshots*[1] *of the network represented as an evolving graph* before and after the failure. The process then involved running a handful of queries on the retrieved window, and then iteratively refining the queries until a hypothesis could be formed. The process culminating with the repetition of the *same* queries on a different window. Similarly, Taylor's quest towards discovering money laundering involved improving the fraud detection algorithm by retrieving the *complete states of the transaction graph at different segments in time* to train and test variants of the algorithm. In such scenarios, neither the queries nor the windows on which the queries would be run are predetermined.

To efficiently perform ad-hoc window operations, a graph processing system should provide two key capabilities. First, it must be able to quickly retrieve arbitrary size windows starting at arbitrary points in time. There are two approaches to provide this functionality. The first is to store a *snapshot* every time the graph is updated, i.e., a vertex or edge is added or deleted. While this allows one to efficiently retrieve the state of

[1] A snapshot is a full copy of the graph, and can be viewed as a window of size zero. Non-zero windows have several snapshots.

the graph at *any* point in the past, it can result in prohibitive overhead. An alternative is to store only the changes to the graph and reconstruct a snapshot on demand. This approach is space efficient, but can incur high latency, as it needs to re-apply all updates to reconstruct the requested snapshot(s). Thus, there is a fundamental trade-off between in-memory storage and retrieval time.

Second, we must be able to efficiently execute queries (e.g., connected components) not only on a single window, but also across multiple related windows of the graph. Existing systems, such as Chronos [76] allows executing queries on a single window, while Differential Dataflow [125] supports continuously updating queries over sliding windows. However, none of the systems support efficient execution of queries across multiple windows, as they do not have the ability to share the computation state across windows and computations. This fundamental limitation of existing systems arises from their inability to efficiently store intermediate state from within a query for later reuse.

We present TEGRA[2], a system that enables efficient ad-hoc window operations on time-evolving graphs. TEGRA is based on two key insights about such real-world evolving graph workloads: *(1) during ad-hoc analysis graphs change slowly over time relative to their size , and (2) queries are frequently applied to multiple windows relatively close by in time.*

Leveraging these insights TEGRA is able to significantly accelerate window queries by reusing both storage and computation across queries on related windows. TEGRA solves the storage problem through a highly efficient, distributed, versioned *graph state store* which compactly represents graph snapshots in-memory as logically separate versions that are efficient for arbitrary retrieval. We design this store using persistent data-structures that lets us heavily share common parts of the graph thereby reducing the storage requirements by several orders of magnitude (§3.5). Second, to improve the performance of ad-hoc queries, we introduce an efficient in-memory representation of intermediate state that can be stored in our graph state store and enables non-monotonic[3] incremental computations. This technique leverages the computation pattern of the familiar graph-parallel models to create compact intermediate state that can be used to eliminate redundant computations across queries. (§3.4).

TEGRA exposes these compact persistent snapshots of the graph and computation state using a logical abstraction named *Timelapse*, which hides the intricacies of state management and sharing from the developer. At a high level, a timelapse is formed by a sequence of graph snapshots, starting from the original graph. Viewing the time-evolving graph as consisting of a sequence of independent static *snapshots* of the entire graph makes it easy for the developer to express a variety of computation patterns naturally,

[2]for **T**ime **E**volving **GR**aph **A**nalytics.

[3]Allows vertex/edge deletions, additions and modifications on any graph algorithm implemented in a graph-parallel fashion.

while letting the system optimize computations on those snapshots with much more efficient incremental computations (§3.3.1). Finally, since Timelapse is backed by our persistent graph store, users and computations always work on independent *versions* of the graph, without having to worry about consistency issues. TEGRA outperforms existing systems by up to 30× on ad-hoc window operation workloads (§3.7).

In summary, we make the following contributions:

- We present TEGRA, a time-evolving graph processing system that enables efficient ad-hoc window operations on both historic and live data. To achieve this, TEGRA shares storage, computation and communication across queries by compactly representing the evolving graph and intermediate computation state in-memory.
- We propose *Timelapse*, a new abstraction for time-evolving graph processing. TEGRA exposes timelapse to the developer using a simple API that can encompass many time-evolving graph operations. (§3.3.1)
- We design (DGSI), an efficient distributed, versioned property graph store that enables timelapse APIs to perform efficient operations. (§3.5)
- Leveraging timelapse and DGSI, we present an iterative, incremental graph computation model which supports non-monotonic computations. (§3.4)

3.2 Background & Challenges

3.2.1 Time-evolving Graph Workloads

Time-evolving graph workloads, an important graph workload [153], can be of three categories:

Temporal Queries: Here, an analyst is querying the graph at different points in the past and evaluates how the result changes over time. Examples are *"How many friends did Alice have in 2017?"* or *"How did Alice's friend circle change in the last three years?"*. Such queries may have time windows of the form $[T - \delta, T]$ and are performed on offline data, and are executed in batch.

Streaming/Online Queries: These workloads are aimed at keeping the result of a graph computation up-to-date as new data arrives (i.e., $[Now - \delta, Now]$). For example, the analyst may ask *"What are the trending topics now?"*, or use a moving window (e.g., *"What are the trending topics in the last 10 minutes?"*). These queries focus on the most recent data, thus streaming systems operate on live graph.

Ad-hoc Queries: In these workloads, an analyst is likely to explore the graph by performing ad-hoc queries on arbitrary windows. For example, consider the exact queries used by Alex (§1.1), a network administrator troubleshooting a transient failure that occurred at 09:00AM. To do so, Alex may ask *"What were the hotspots at 08:00AM?"* which runs a connected component algorithm on the snapshot. Based on what is seen, the next query may ask *"What were the hotspots at 10:00AM?"* followed by *"At 10:00AM, what is the shortest path of hotspot X to the controller?"*. Alex iteratively refines this query by taking many snapshots and running the query. In another example, Taylor, a financial expert is interested in improving the fraud-detection algorithm[4]. Taylor queries *"Who were the top influencers 1 month around April 1?"* which retrieves the window of 1 month around a known fraud and runs an algorithm (e.g., personalized page rank) on the window, and then launches follow-up queries based on what the output is. Taylor repeats this on different windows to learn new rules that would detect the fraud. Taylor then tests her changes on a different set of windows, possibly also with injecting artificial data.

In ad-hoc workloads, not only does the analyst need to access arbitrary windows, but also the queries and the windows on which they are executed are determined just-in-time (i.e., not predetermined). Further, the analyst applies the same query to multiple (close-by, discontinuous) windows.

3.2.2 Limitations of Existing Solutions

Recent work in graph systems has made considerable progress in the area of evolving graph processing. (§3.8)

Temporal analysis engines (e.g., Chronos [76], ImmortalGraph [120]) operate on *offline data* and focus on executing queries on one or a sequence of snapshots in the graph's history. Upon execution of a query, these systems load the relevant history of the graph and utilize a pre-processing step to create an in-memory layout that is efficient for analysis. Such preprocessing can often dominate the algorithm execution time [116]. As a result, these systems are tuned for operating on a large number of snapshots in each query (e.g., temporal changes over months or year), and are efficient in such cases. Fundamentally, the in-memory representation in these systems cannot support updates. Additionally, these systems do not allow updating the results of a query.

Streaming systems (e.g., Kineograph [45], DifferentialDataflow [119], Kickstarter [175], GraphBolt [117]) operate on *live data* and allow query results to be updated *incrementally* (rather than doing a full computation) when *new* data arrives. These systems only allow queries on the live graph, and do not support ad-hoc retrieval of previous state.

[4]Typically a combination of machine learning and expert rules.

Additionally, the incremental computation is tied to the live state of the graph, and cannot be utilized over multiple windows. Further, most systems (with the exception of Differential Dataflow, to the best of our knowledge) do not support *non-monotonic* computations in their incremental model, and either assume some properties of the algorithm, or leave it up to the developer to ensure correctness.

Differential Dataflow (DD) allows general, non-monotonic incremental computations using special versions of operators. Each operator stores "differences" to its input and produces the corresponding differences in output (hence full output is not materialized), automatically incrementalizing algorithms written using them. While this technique is very efficient for real-time streaming queries, incorporating ad-hoc window operations in it is fundamentally hard. Since the computation model is based on the operators maintaining state (*differences* to their input and output) indexed by data (rather than time), accessing a particular snapshot can require a full scan of the maintained state. Further, since every operator needs to maintain state, the system accumulates large state over time which must be compacted (at the expense of forgoing the ability to retrieve the past). Finally, intermediate state of a query is cleared once it completes and storing these efficiently for reuse is an open question[5].

3.2.3 Challenges

Meeting the requirements necessary to support efficient ad-hoc window operations in practice is hard. There are three main challenges that stand in the way of building such a system. First is that of *programmability*. The system must be able to provide the end user a natural and intuitive way to operate on time-evolving graphs. Second, the system must support efficient *storage* of evolving graphs. While it is ideal to store the history of the graph as individual snapshots for zero overhead ad-hoc retrieval, but the system needs to consider the storage overhead due to duplication with every snapshot stored. Finally, ad-hoc analytics is both interactive (the user is waiting for answers) and exploratory (new queries are based on the answers to previous ones) and thus, extracting *performance* under these constraints is difficult. To accelerate queries across windows, the system must not only be able to store intermediate states efficiently, but also be able to leverage them in its computation model. In essence, it must be able to compactly represent and share data and state between queries across multiple windows and users.

[5]Our conversations with the author of DD revealed that incorporating the state management techniques we propose in this work in DD is fundamentally hard and requires modification of its execution engine.

3.3 Tegra Design

Our solution, TEGRA, consists of three components:

Timelapse Abstraction (§3.3.1): In TEGRA, users interact with time-evolving graphs using the *timelapse* abstraction, which logically represents the evolving graph as a sequence of *static*, *immutable* graph snapshots. TEGRA exposes this abstraction via a simple API that allows users to save/retrieve/query the materialized *state* of the graph at any point.

Computational Model (§3.4): TEGRA proposes a computation model that allows *ad-hoc queries across windows to share computation and communication*. The model stores compact intermediate state as a timelapse, and uses it to perform general, *non-monotonic* incremental computations.

Distributed Graph Snapshot Index (§3.5): TEGRA stores evolving graphs, intermediate computation state and results in DGSI, an efficient indexed, distributed, versioned property graph store which *shares storage* between versions of the graph. In fact, Timelapse can be seen as "views" on the data stored in DGSI. Such decoupling of state from queries and operators allow TEGRA to share it across queries and users.

3.3.1 Timelapse Abstraction & API

TEGRA introduces *Timelapse* as a new abstraction for time-evolving graph processing that enables efficient ad-hoc analytics. The goal of timelapse is to provide the end-user with a simple, natural interface to run queries on time-evolving graphs, while giving the system opportunities for efficiently executing those queries. In timelapse, TEGRA *logically* represents a time-evolving graph as a sequence of immutable, static graphs, each of which we refer to as *snapshot* in the rest of this chapter. A snapshot depicts a consistent state of the graph at a particular instance in time. TEGRA uses the popular property graph model [71], where vertices and edges in the graph are associated with arbitrary properties, to represent each snapshot in the timelapse. For the end-user, timelapse provides the abstraction of having access to a *materialized* snapshot at any point in the history of the graph. This enables the usage of the familiar static graph processing model in evolving graphs (e.g., queries on arbitrary snapshot).

Timelapses are created in TEGRA in two ways—by the system and by the users. When a new graph is introduced to the system, a timelapse is created for it that contains a single snapshot of the graph. Then, as the graph evolves, more snapshots are added to the timelapse. Similarly, users may create timelapses while performing analytics. Because

`save(id): id`	Save the state of the graph as a snapshot in its timelapse. ID can be autogenerated. Returns the id of the saved snapshot.
`retrieve(id): snapshot`	Return one or more snapshots from the timelapse. Allows simple matching on the id.
`diff(snapshot, snapshot): delta`	Difference between two snapshots in the timelapse. (§3.4)
`expand(candidates): subgraph`	Given a list of candidate vertices, expand the computation scope by marking their 1-hop neighbors. Used for implementing incremental computations (§3.4)
`merge(snapshot,snapshot,func): snapshot`	Create a new snapshot using the union of vertices and edges of two snapshots. For common vertices, run func to compute their value. Used for implementing incremental computations (§3.4)

Table 3.1: TEGRA exposes Timelapse via simple APIs.

snapshots in a timelapse are immutable, any operation on them creates new snapshots as a result (e.g., a query on a snapshot results in another snapshot as a result). Such newly created snapshots during an analytics session may be added to an existing timelapse, or create a new one depending on the kind of operations performed. For instance, for an analyst performing what-if analysis by introducing artificial changes to the graph, it is logical to create a new timelapse. Meanwhile, snapshots created as a result of updating a query result should ideally be added to the same timelapse. The system does not impose restrictions on how users want to book-keep timelapses. Instead, it simply tracks their lineage and allows users to efficiently operate on the timelapses. (§3.5)

Since timelapse logically represents a sequence of related graph snapshots, it is intuitive to expose the abstraction using the same semantics as that of static graph. In TEGRA, users interact with timelapses using a language integrated API. The API extends the familiar `Graph` interface, common in static graph processing systems, with a simple set of additional operations, listed in table 3.1. This enables users to continue using existing static graph operations on any snapshot in the timelapse obtained using the `retrieve()` API.

3.3.2 Evolving Graph Analytics Using Timelapse

The natural way to do graph computations over the time dimension is to iterate over a sequence of snapshots. For instance, an analyst interested in executing the `degrees` query on three snapshots, G_1, G_2 and G_3 depicted in fig. 3.1 can do:

```
for(id <- Array(G1,G2,G3))
  result = G.retrieve(id).degrees
```

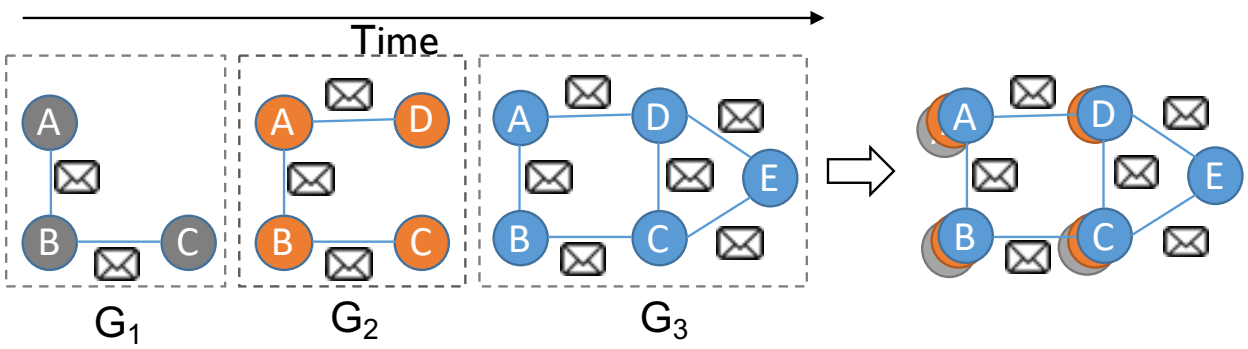


Figure 3.1: A timelapse of graph G consisting of three snapshots. For temporal analytics, instead of applying graph-parallel operations independently on each snapshot (left), timelapse enables them to be applied to all snapshots in parallel (right).

However, applying the same operation on multiple snapshots of a time-evolving graph independently is inefficient. In graph-parallel systems (§3.2), `degrees()` computation is typically implemented using a user-defined program where every vertex sends a message with value 1 to their neighbors, and all vertices adding up their incoming message values. Such message exchange accounts for a non-trivial portion of the analysis time [154]. In the earlier example, sequentially applying the query to each snapshot results in 11 messages of which 5 are duplicates (fig. 3.1).

To avoid such inefficiencies, timelapse allows access to the *lineage* of graph entities. That is, it provides efficient retrieval of the state of graph entities in any snapshot. Using this, graph-parallel phases can operate on the evolution of an entity (vertex or edge) as opposed to a single (at a given snapshot) value. In simple terms, each processing phase is able to see the history of the node's property changes. This allows *temporal* queries (§3.2.1) involving multiple snapshots, such as the degree computation, to be efficiently expressed as:

```
results = G.degrees(Array(G1,G2,G3))
```

where `degrees` implementation takes advantage of timelapse by combining the phases in graph-parallel computation for these snapshots. That is, the user-defined vertex program is provided with state in all the snapshots. Thus, we are able to eliminate redundant messages and computation.

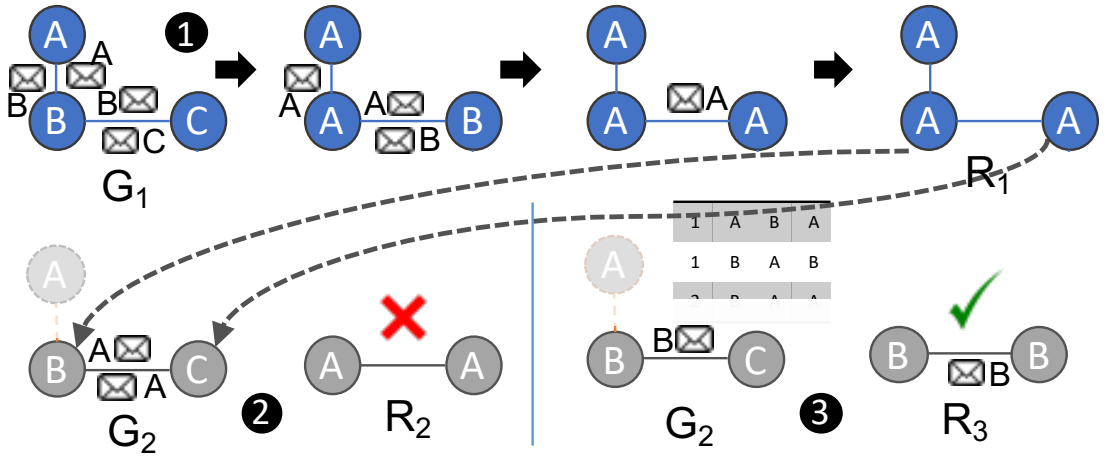


Figure 3.2: ❶ Connected components by label propagation on snapshot G_1 produces R_1. ❷ Vertex A and edge $A - B$ is deleted in G_2. Using the last result to bootstrap computation results in incorrect answer R_2. ❸ A strawman approach of storing all messages during the initial execution and replaying it produces correct results, but needs to store large amounts of state.

3.4 Computation Model

To improve interactivity, TEGRA must be able to efficiently execute queries by effectively reusing previous query results to reduce or eliminate redundant computations, commonly referred to as performing *incremental computation*. Here, we describe TEGRA's incremental computation model.

3.4.1 Incremental Graph Computations

Supporting incremental computation requires the system to manage *state*. The simplest form of state is the previous computation result. However, many graph algorithms are iterative in nature, where the graph-parallel stages are repeatedly applied in sequence until a fixed point. Here, simply restarting the computations from previous results do not lead to correct answers. To illustrate this, consider a connected components algorithm using label propagation on a graph snapshot, G_1 as shown in ❶ in fig. 3.2 which entails result R_1 after three iterations. When the query is to be repeated on G_2, restarting the computation from R_1 as shown in ❷ computes incorrect result. In general, correctness in such techniques depend on the properties of the algorithm (e.g., abelian group) and the monotonicity of updates (e.g., the graph only grows).

Supporting general non-monotonic iterative computations require maintaining *intermediate* state. In the previous example, one solution is to store every message exchanged between graph entities during the initial execution of the algorithm. When the query is executed on the updated graph, the system can selectively replay these stored messages to ensure correctness of the results as depicted in ❸. However, this approach requires storing and effectively using large amounts of state (proportional to the number of edges for every iteration), which may pose prohibitive overheads when applied to real-world graphs where the number of edges are significantly more compared to vertices [70]. Further, the state is tied to the computation performed and thus doesn't provide opportunities to share it across queries.

TEGRA proposes a general, incremental iterative graph-parallel computation model that significantly reduces the state requirements. It leverages the fact that graph-parallel computations proceed by making iterative changes to the original graph. Thus, *iterations of a graph-parallel computation can be seen as a time-evolving graph*, where the snapshots are the materialized state of the graph at the end of each iteration. Since timelapse can efficiently store and retrieve these snapshots, we can perform incremental computations without the need to store the message exchanges. We call this model *Incremental Computation by entity Expansion* (ICE).

3.4.2 ICE Computation Model

ICE executes computations only on the subgraph that would be affected by the updates *at each iteration*. To do so, it needs to find the relevant entities that should participate in computation at any given iteration. For this, it uses the state stored as timelapse, and the computation proceeds in four phases:

Initial execution: When an algorithm is executed for the first time, ICE stores the state of the vertices (and edges if the algorithm demands it) as properties in the graph. At the end of every iteration, a snapshot of the graph is added to the timelapse. The ID is generated using a combination of the graph's unique ID, an algorithm identifier and the iteration number. As depicted in ❶ in the examples in fig. 3.3, the timelapse contains three and four snapshots, respectively.

Bootstrap: When the computation is to be executed on a new snapshot, ICE needs to bootstrap the incremental computation. Intuitively, the subgraph that must participate in the computation at bootstrap consists of the updates to the graph, and the *entities affected by the updates*. For instance, any newly added or changed vertices should be included. Similarly, edge modifications would result in the source and/or destination vertices to be included in the computation. However, the changes alone are not sufficient to ensure correctness of the results. This is because in graph-parallel execution, the state of a

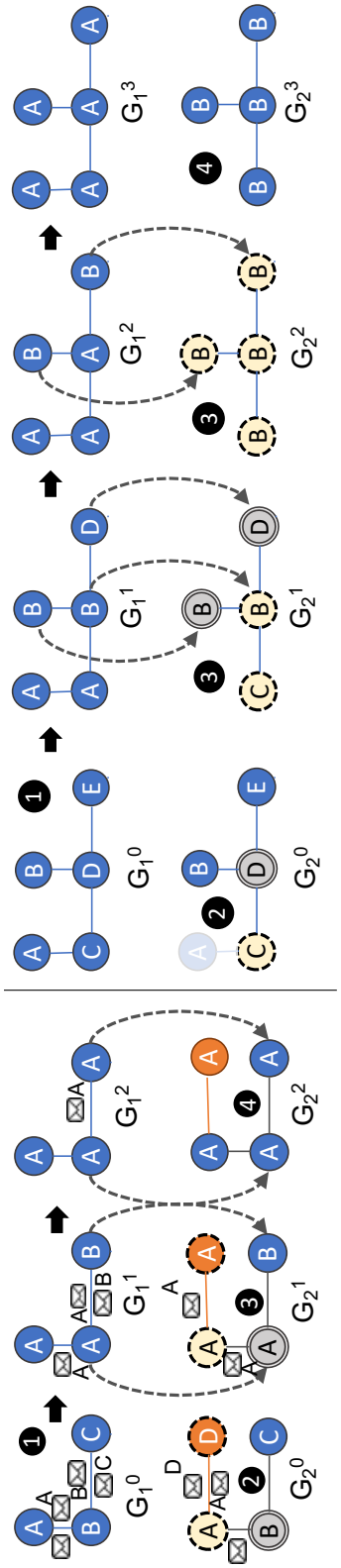


Figure 3.3: Two examples that depict how ICE works. Dotted circles indicate vertices that recompute, and double circles indicate vertices that need to be present in the subgraph to compute the correct answer, but do not recompute state themselves. ❶ Iterations of initial execution is stored in the timelapse. ❷ ICE bootstraps computation on a new snapshot, by finding the subgraph consisting of affected vertices and their dependencies (neighbors). In the second example, C is affected by the deletion of $A - C$. To recompute state it needs D (yields subgraph $C - D$). ❸ At every iteration, after execution of the computation on the subgraph, ICE copies state for entities that did not recompute. Then finds the new subgraph to compute by comparing the previous subgraph to the timelapse snapshot. In the second example, though C recomputes the same value as in previous iteration, its state is different from the snapshot in timelapse and hence needs to be propagated. ❹ ICE terminates when the subgraph converges and no entity in the graph needs the state copied from stored snapshots in the timelapse.

graph entity is dependent on the collective input from its neighbors. Thus, ICE must also include the one-hop neighbors of *affected entities*, and so the bootstrapped subgraph consists of the affected entities and their one-hop neighbors. ICE uses the `expand()` API for this purpose. The graph computation is run on this subgraph. The first example's ❷ in fig. 3.3 shows how ICE bootstraps when a new vertex D and a new edge between A and D is added. D and A should recompute state, but for A to compute the correct state, it must involve its one-hop neighbor B, yielding subgraph $D - A - B$.

Iterations: At each iteration, ICE needs to find the right subgraph to perform computations. ICE exploits the fact that the nature of the graph-parallel abstraction restricts the propagation distance of updates in an iteration. Intuitively, the graph entities that might possibly have a different state at any iteration will be contained in the subgraph that ICE has already executed computation on from the last iteration. Thus, after the initial bootstrap, ICE can find the new subgraph at a given iteration by examining the changes to the subgraph from the previous iteration and expanding to the one-hop neighborhood of affected entities. For the vertices/edges that did not recompute the state, ICE simply copies the state from the timelapse. In ❸ in fig. 3.3 (first example), though A and D recomputed, only D changed state and needs to be propagated to its neighbor A which needs B.

Termination: It is possible that modifications to the graph may result in more (or less) number of iterations compared to the initial execution. Unlike normal graph-parallel computations, ICE does not necessarily stop when the subgraph converges. If there are more iterations stored in the timelapse for the initial execution, ICE needs to check if the unchanged parts of the graph must be copied over. Conversely, if the subgraph has not converged and there are no more corresponding iterations, ICE needs to continue. To do so, it simply switches to normal (non-incremental) computation from that point. Thus, ICE converges only when the subgraph converges and no entity needs their state to be copied from the stored snapshot in the timelapse. (❹ in fig. 3.3)

3.4.3 ICE vs Streaming Systems

ICE provides several desirable properties for ad-hoc exploratory analysis on evolving graphs, and differs from the computation models in existing streaming graph processing systems (e.g., Differential Dataflow, Kickstarter, GraphBolt) in two major ways. First, the state in ICE model is decoupled from computation, and ICE generates the exact same intermediate states as a system that executes the algorithm from scratch (i.e., non-incremental computation) at every iteration. In addition to guaranteeing the correctness even for non-monotonic computations, this allows ICE to leverage *any* user specified previous state for incremental computations compared to streaming systems which can

only utilize the last state (i.e., there are no ordering constraints e.g., an analyst can run a query on a graph snapshot at 9:00am and use the state to run the query on 8:00am snapshot). Second, streaming systems typically cannot utilize newly available data when a computation is in progress—for example, a query might be in progress when a new snapshot of the graph is available. TEGRA's choice of immutable snapshots allows it to separate computation and data ingestion. Thus, ICE can switch to a new snapshot when available (and leverage the computation until the switch due to state decoupling).

3.4.4 Improving ICE Model

Sharing State Across Different Queries Many graph algorithms consist of several stages of computations, some of which are common across different algorithms. For example, variants of connected components and pagerank algorithms both require the computation of vertex degree as one of the steps. Since ICE decouples state, such common computations can be stored as separate state that is shared across different queries. Thus, ICE enables developers to generate and *compose* modular states. This reduces the need to duplicate common state across queries which results in reduced memory consumption and better performance.

Incremental Computations Can Be Inefficient Incremental computation is not useful in all cases. For instance, in graphs with high degree vertices, a small change may result in a domino effect in terms of computation—that is, during later iterations, a large number of graph entities might need to participate in computation (e.g., Example 2 in fig. 3.3). To perform incremental computation, ICE needs to spend computations cycles to identify the set of vertices that should recompute (using `diff`) and copy the state of vertices that did not do computations (using `merge`). Due to this, the total work done by the system may exceed that of completely executing the computation from scratch [61, 175]. Since ICE generates the same intermediate states at every iteration as full re-execution, it can switch to full re-execution.

We propose a simple learning based technique to determine when to do this switch. We use a decision tree classifier to predict if the current iteration would be faster using incremental or non-incremental execution. To train the classifier, we use offline data that consists of several runs of queries both in incremental and non-incremental fashion. At each iteration of every run, we note the number of vertices that participated in computation in the last iteration, the time taken for the last iteration, and whether full execution or incremental execution was faster in that iteration. The label is defined as whether incremental execution was faster than full execution at this iteration. At run time, we use the classifier to predict whether to switch to full execution.

3.5 Distributed Graph Snapshot Index (DGSI)

To make timelapse abstraction and ICE computation model practical, TEGRA needs to back them with a storage that satisfies the following three requirements: (1) enable ingestion of updates in real-time, and make it available for analysis in the minimum time possible, (2) support space-efficient storage of snapshots and intermediate computation state in a timelapse, and (3) enable fast retrieval and efficient operations on stored timelapses. These requirements, crucial for efficiently supporting ad-hoc analytics on time-evolving graphs, pose several challenges. For instance, they prohibit the use of pre-processing, typically employed by many graph processing systems, to compactly represent graphs and to make computations efficient. In this section, we describe how TEGRA achieves this by building DGSI. It addresses requirements (1) and (2) by leveraging persistent datastructures to build a graph store (§3.5.1, §3.5.2) that enables efficient operations (§3.5.3) while managing memory over time (§3.5.4).

3.5.1 Leveraging Persistent Datastructures

In TEGRA, we leverage persistent datastructures [55] to build a distributed, versioned graph state store. The key idea in persistent datastructures is to maintain the previous versions of data when modified, thus allowing access to earlier versions. DGSI uses a persistent version of the Adaptive Radix Tree [108] as its datastructure. ART provides several properties useful for graph storage such as efficient updates and range scans. Persistent Adaptive Radix Tree (PART) [137] adds persistence to ART by simple path-copying. For the purpose of building DGSI, we reimplemented PART (hereafter pART) in Scala and made several modifications to optimize it for graph state storage. We also heavily engineered our implementation to avoid performance issues, such as providing fast iterators, avoiding unnecessary small object creation and optimizing path copying under heavy writes.

3.5.2 Graph Storage & Partitioning

TEGRA stores graphs using two pART datastructures: a *vertex* tree and an *edge* tree. The vertices are identified by a 64-bit integer key. For edges, we allow arbitrary keys stored as byte arrays. By default, the edge keys are generated from their source and destination vertices and an additional short field for supporting multiple edges between vertex pairs. pART supports prefix matching, so using matching on this key enables retreiving all the destination edges of a given vertex. The leaves in the tree store pointers

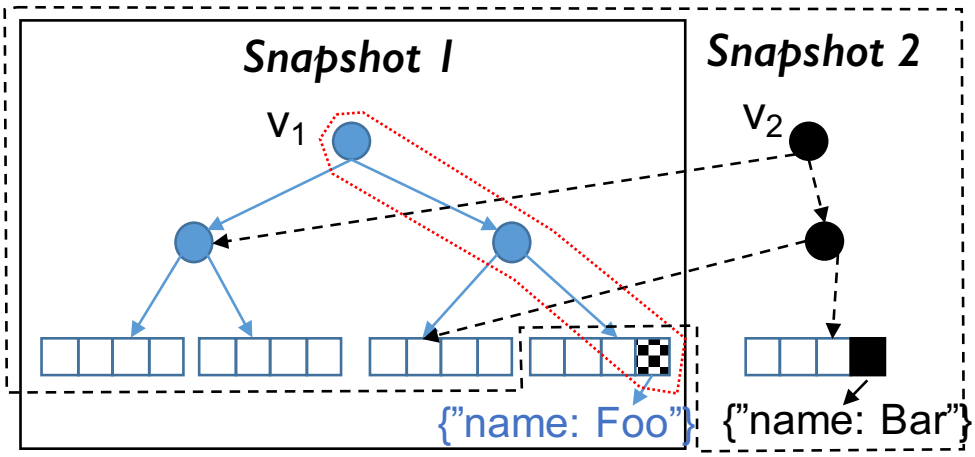


Figure 3.4: TEGRA's DGSI consists of one pART datastructure for vertices and one for edges on each of the partitions. Here, a vertex pART stores properties in its leaves. Vertex id traverses the tree to the leaf storing its property. Changes generate new versions.

to arbitrary properties.We create specialized versions of pART to avoid (un)boxing costs when properties are primitve types.

TEGRA supports several graph partitioning schemes, similar to GraphX [71], to balance load and reduce communication. To distribute the graph across machines in the cluster, vertices are hash partitioned and edges are partitioned using one of many schemes (e.g., 2D partitioning). We do not partition the pART structures, instead TEGRA partitions the graph and creates *separate* pART structures locally in each partition. Hence logically, in each partition, the vertex and edge trees store a subgraph (fig. 3.4). By using local trees, we further amortize the (already low) cost[6] associated with modifying the tree upon graph updates. To consume graph updates, TEGRA needs to send the updates to the right partition. For this, we impose the *same* partitioning as the original graph on the vertices and edges in the update.

3.5.3 Version Management

DGSI is a versioned graph state store. Every "version" corresponds to a root in the vertex and edge tree in the partitions—traversing the trees from the root pair materializes the graph snapshot. For version management, DGSI stores a mapping between a root and the corresponding "version id" in every partition. The version id is simply a byte array.

For operating on versions, DGSI exposes two low level primitives inspired by existing version management systems: `branch` and `commit`. A `branch` operation creates a new

[6]Modifications to nodes in ART trees only affect the $O(\log_{256} n)$ ancestors

working version of the graph by creating a new (transient) root that points to the original root's children. Users operate on this newly created graph without worrying about conflicts because the root is exclusive to them and not visible in the system. Upon completing operations, a `commit` finalizes the version by adding the new root to version management and makes the new version available for other users in the system. Once a `commit` is done on a version, modifications to it can only be done by "branching" that version. Any timelapse based modifications cause `branch` to be called, and the timelapse `save()` API invokes `commit`.

TEGRA can interface with external graph stores, such as Neo4J [126], Titan [170] or Weaver [180] for importing and exporting graphs. While importing new graphs, DGSI automatically assigns an integer id (if not provided) and commits the version when the loading is complete. We create a version by batching updates. The batch size is user-defined. In order to be able to retrieve the state of the graph in between snapshots, TEGRA stores the updates between snapshots in a simple log file, and adds a pointer to this file to the root.

The simplest retrieval is by using its id. In every partition, DGSI then gets a handle to the root element mapped to this id, thus enabling operations on the version (e.g., branching, materialization). By design, versions in DGSI have no global ordering because branches can be created from any version at any time. However, in some operations, it may be desirable to have ordered access to versions, such as in incremental computations where the system needs access to the consecutive iterations. For this purpose, we enable suffix, prefix and simple ranges matching primitives on the version id.

3.5.4 Memory Management

Over time, DGSI stores several versions of a graph, and hence TEGRA needs to manage these versions efficiently. We employ several ways to do this. Between `branch` and `commit` operations, it is likely that many transient child nodes are formed. We aggressively remove them during the `commit` operation. In addition, we enable in-place updates when the operations are local, such as after a branch and before a commit. Further, during ad-hoc analysis, analysts are likely to create versions that are never committed. We periodically mark such orphans and adjust the reference counting in our trees to make sure that they are garbage collected.

For managing stored versions, we leverage a simple Least Recently Used (LRU) eviction policy. Each time a version is accessed, we annotate the version and all its children with a timestamp. The system then employs a thread for periodically removing versions that were not accessed in a long time. The eviction is done by saving the version to local disk (or distributed file system). We do this in the following way. Since every

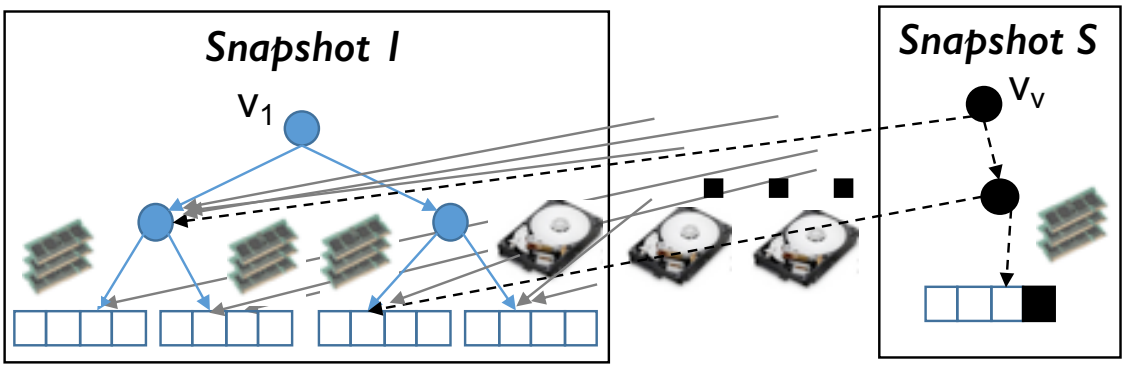


Figure 3.5: DGSI has fine-grained control over leaves (where data is stored). Here DGSI has 1000s of snapshots. All snapshots except S is on disk, their parents just hold pointers to the file. Parents are also dynamically written to disk if all of their children are on disk. Datastructure uses adaptive leaf sizes for efficiency.

version in DGSI is a branch, we write each subtree in that branch to a separate file and then point its root to the file identifier (e.g., in fig. 3.4, we can store v_2's leaf that is different from v_1 in disk as a file and point the parent node to this file). By writing subtrees to separate files, we ensure that different versions sharing tree nodes in memory can share tree nodes written to files. Due to this technique, we can ensure that leaf nodes (which are most memory consuming) that are specific to a version (not shared with any other version) are always written to disk if the version is evicted. As depicted in fig. 3.5, a large number of versions can be flushed to disk over time while still being retrievable when necessary. Thus, only active snapshots are fully materialized in memory, thereby allowing Tegra to store several snapshots.

3.6 Implementation

We have implemented Tegra on Apache Spark [192] as a drop-in replacement for GraphX [71][7]. We utilize the newly available barrier execution mode to implement direct communication between tasks to avoid most Spark overheads.

3.6.1 ICE on GAS Model

As described in §3.4.2, the `diff()` API marks the candidates that must perform graph-parallel computation in a given iteration. In GAS decomposition, the `scatter()` function,

[7]Tegra is a new implementation and does not extend GraphX's codebase.

invoked on `scatter_nbrs`, determines the set of active vertices which must perform computation. Starting with an initial candidate set (e.g., at bootstrap the changes to the graph, and at any iteration the candidates from the previous iteration) the **`diff()`** API uses `scatter_nbrs` (`EdgeDirection` in GraphX) in the user-defined vertex program to mark all necessary vertices for computation. We mark all `scatter_nbrs` of a vertex if its state differs from the previous iteration, or from the previous execution stored in the timelapse. For instance, a vertex addition must inspect all its neighbors (as defined by `scatter_nbrs`) and include them for computation.

The vertices in GAS parallel model perform computation using the user-defined `gather()`, `sum()` and `apply()` functions, where `gather_nbrs` determine the set of neighbors to gather state from. The **`expand()`** API enables correct `gather()` operations on the candidates marked for recomputation by also marking the `gather_nbrs` of the candidates. After the **`diff()`** and **`expand()`**, TEGRA has the complete subgraph on which the graph-parallel computation can be performed.

3.6.2 Using Tegra as a Developer

TEGRA provides feature compatibility with GraphX, and expands the existing APIs in GraphX to provide ad-hoc analysis support on evolving graphs. It extends all the operators to operate on user-specified snapshot(s) (e.g., `Graph.vertices(id)` retrieves vertices at a given snapshot id, and `Graph.mapV([ids])` can apply a map function on vertices of the graph on a set of snapshots). Graph-parallel computation is enabled in GraphX using the `Graph.aggregateMessages()` (previously `mrTriplets()`) API. Because this API works on the `Graph` interface, developers can directly apply it to the subgraph found by using TEGRA's **`diff()`** and **`expand()`** calls and then use **`merge()`** to materialize the complete result.

GraphX further offers iterative graph-parallel computation support through a Pregel API which captures the GAS decomposition using repeated invocation of the `aggregateMessages` and `joinVertices` until a fixed point. TEGRA provides an incremental version of this Pregel API that can perform ICE from a previously saved computation state provided as an argument as shown in listing 3.1. In general, a developer can write incremental versions of any iterative graph parallel algorithm by using the TEGRA APIs along with `aggregateMessages`. TEGRA provides a library of incremental versions of commonly used graph algorithms.

3.7 Evaluation

We have evaluated TEGRA through a series of experiments.

```
def IncPregel(g: Graph[V, E],
     prevResult: Graph[V, E],
     vprog: (Id, V, M) => V,
     sendMsg: (Triplet) => M,
     gather: (M, M) => M): Graph[V, E] = {
 iter = 0
 // Loop until no active vertices and nothing to copy
 // from previous results in timelapse.
 while (!converged) {
   // Restrict to vertices that should recompute
   val msgs: Collection[(Id, M)] =
     g.expand(g.diff(prevResult.retrieve(iter))).
      .aggregateMessages(sendMsg, gather)
   iter += 1
   // Receive messages and copy previous results
   g = g.leftJoinV(msgs).mapV(vprog)
        .merge(prevResult.retrieve(iter)).save(iter) }
 return g }
```

Listing 3.1: Implementation of incremental Pregel using TEGRA API.

Dataset	Vertices / Edges
twitter [31]	41.6 M / 1.47 B
uk-2007 [33]	105.9 M / 3.74 B
Facebook Synthetic Data [2]	Varies / 5, 10, 50 B

Table 3.2: Datasets in our evaluation. M = Millions, B = Billions.

Comparisons. We compare TEGRA against a streaming engine and a temporal engine. For streaming system, we use the Rust implementation of Differential Dataflow (DD) [54]. Since we were unable to obtain an open source implementation of a temporal engine, we developed a simplified version of Chronos [76] in GraphX [71], which we call Chlonos (Clone of Chronos) in this section. This implementation emulates an array based in-memory layout of snapshots and the incremental computation model in Chronos. For graph updates, we use a mixture of additions and deletions.

Evaluation Setup. All of our experiments were conducted on 16 commodity machines available as Amazon EC2 instances, each with 8 virtual CPU cores, 61GB memory, and 160GB SSDs. The cluster runs a recent 64-bit version of Linux. We use Differential Dataflow v0.8.0 and Apache Spark v2.4.0. We warm up the JVM before measurements.

Caveats. While perusing the evaluation results, we wish to remind the reader a few caveats. Though many of the graphs we use fit in the memory of a modern single machine, TEGRA is focused on ad-hoc analytics which requires storage of multiple snapshots of the graph. Further, ad-hoc analytics requires the use of property graphs, and thus TEGRA supports edge and vertex properties (and creates a default value) which blows up the graph size several magnitudes (and also affects performance). DD does not support properties which makes it highly memory efficient. Further, DD only outputs differences (not fully materialized results) which TEGRA materializes the entire result on every computation. Finally, DD's connected component implementation uses union-find (hard to fit in a vertex centric model) which is superior to TEGRA's label propagation based implementation.

3.7.1 Microbenchmarks

Graph Update Throughput: To evaluate the ability of TEGRA to sustain high update volumes, we load the Twitter graph. We randomly add and remove 1 million edges (no computations are performed) and note the time to store the updates and compute the throughput. We repeat the experiment 10 times each with varying number of machines and average the results. We observe that TEGRA is able to average about 1 million updates per machine and this number scales linearly with more machines, which is expected as there is no coordination required for updating the graph. DD is able to achieve 20 million updates per machine, but it is simply inserting the updates in native arrays while TEGRA is applying the updates to the graph stored in DGSI.

Snapshot Retrieval Latency: Next, we repeat the experiment to evaluate the snapshot retrieval latency. After 1000 updates, we retrieve random snapshots from graph. For each retrieval, we note the time for the system to materialize the output. We note the average of 10 retrievals each on different number of machines in fig. 3.6. DD does periodic state compaction to reduce state overheads, but this results in the inability to retrieve the past. One potential solution is to create a snapshot at the point of compaction and store it separately along with a redo log. However, this requires a solution similar to DGSI (§3.2) to store these snapshots efficiently without duplication. Since this is beyond the scope of this work, we disable compaction in DD and modify it to retrieve the snapshot at a given time.

We see that TEGRA is able to return the queried snapshot with no computation at all, since it materializes the snapshot at ingestion time. In contrast, DD needs to reconstruct the graph when it is queried. Reconstructing the graph takes about 20 seconds on a single machine, and reduces linearly with number of machines. However, as more updates are

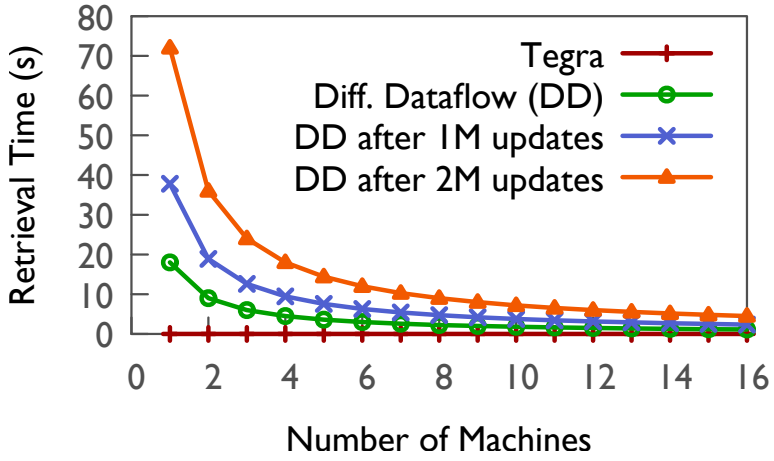


Figure 3.6: Snapshot retrieval latency in DD incurs cost and degrades with time.

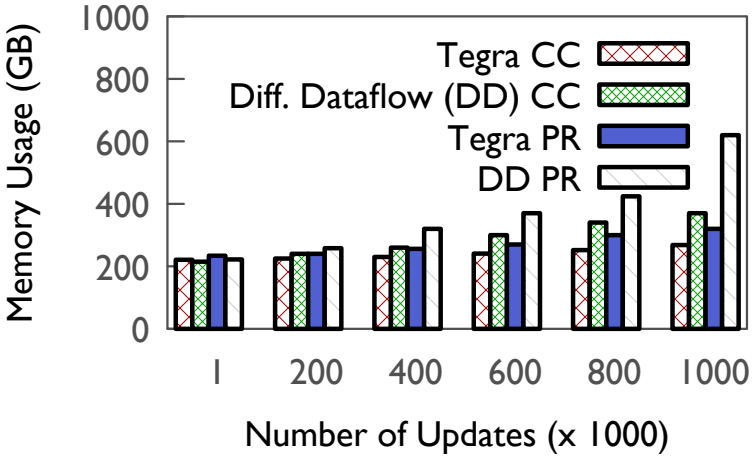


Figure 3.7: Differential dataflow generates state at every operator, while TEGRA's state is proportional to the number of vertices.

added, the retrieval time degrades. DD also exhibits high variance in retrieval time based on the snapshot retrieved.

Computation State Storage Overhead: Finally, we measure the memory overhead due to computation state. We perform page rank (PR) and connected components (CC) computation on the Twitter graph in an incremental fashion, where we add and delete 1000 edges to create a snapshot. We note the memory usage by each system after every 200 such computations until 1000 computations (for a total of 1 million edge updates). Figure 3.7 shows this experiment's results. When the number of updates are small, both TEGRA and DD use comparable amount of memory to store the state, even with DD's

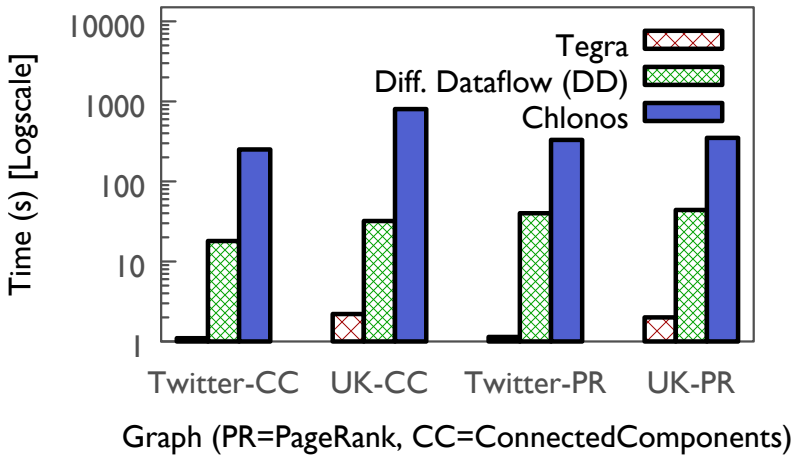


Figure 3.8: On ad-hoc queries on snapshots, TEGRA is able to significantly outperform due to state reuse.

highly compact layout (native arrays compared to TEGRA's property graph), but DD's state size increases rapidly as it does more incremental computation and takes up to 2× that of TEGRA. TEGRA's memory requirement also increases over time, but much more gracefully. The reason is that TEGRA's state requirement is proportional to the number of vertices, while DD needs to keep state at every operator. The amount of increase also depends on the algorithm. For instance, page rank generates the same amount of state in every iteration while connected component's state requirement reduces over iterations. Note that DD uses compaction in this experiment which is automatically done by the system.

3.7.2 Ad-hoc Window Operations

Here, we present evaluations that focus on TEGRA's main goal. In these experiments, we emulate an analyst. We load the graph, and apply a large number of sequential updates to the graph, where each update modifies 0.1% of the edges. We then retrieve 100 random windows of the graph that are close-by, and apply queries in each. We use page rank and connected components as the algorithms. Page rank is either run until a specific tolerance, or 20 iterations, whichever is smaller. We assume that earlier results are available so that the system could do incremental computations. We do not consider the window retrieval time in this experiment for DD and Chlonos. We present the average time taken to compute the query result once the window is retrieved.

Single Snapshot Operations: In the first experiment, we set the window size to zero so every window retrieval returns a single snapshot. The results are depicted in fig. 3.8.

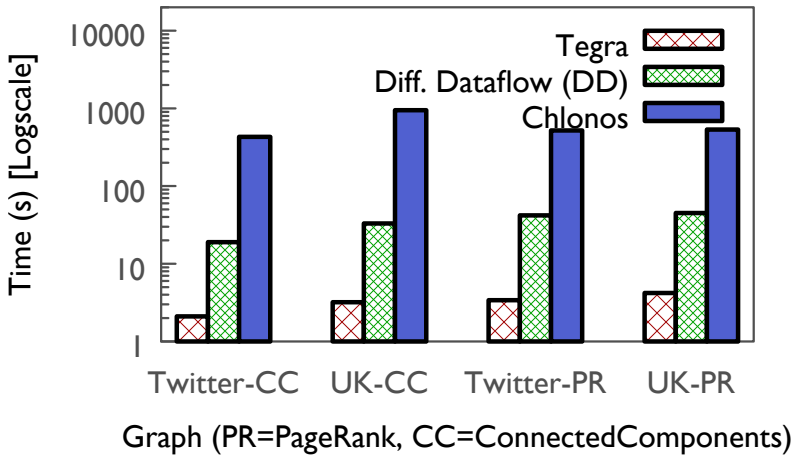


Figure 3.9: TEGRA's performance is superior on ad-hoc window operations even with materialization of results.

DD and Chlonos do not allow reusing computation across queries, so they compute from scratch for every retrieval. In contrast, TEGRA is able to leverage the compact computation state stored in its DGSI from earlier queries for much faster computation. In this case, most of the snapshots incurs no computation overhead because of the extremely small amount of changes between them, and TEGRA is able to produce an answer within a few seconds. DD takes a few 10s of seconds, while Chlonos requires 100s of seconds. TEGRA's benefits range from 18-30× compared to DD.

Window Operations: Here we set the window size to be 10 snapshots. Chlonos and DD are able to apply incremental computations once the query has been computed on the first snapshot. Figure 3.9 shows the results. We see that DD is very fast once the first result has been computed, and incurs minimal overheads after that. In contrast, Chlonos incurs a penalty because it uses the first result to bootstrap the rest. Because TEGRA needs to materializes the result and also store it separately for each snapshot, and due to scheduling overheads in Spark it incurs a slight penalty compared to a single snapshot. TEGRA is still 9-17× faster compared to DD.

Large Graphs: Finally, we evaluate the ability of TEGRA to support ad-hoc window analysis on very large graphs. For this, we use synthetic graphs provided by Facebook [2] modeled using the social network's properties. There are 3 graphs with 5B, 10B and 50B edges respectively.

Here, we load the graph and execute the queries once. Then we modify the graph by a tiny percentage, 0.01% randomly 1000 times to create 1000 snapshots. We then randomly pick a snapshot, and run the queries on it. We provide the average time over 100 such

Graph	5B		10B		50B	
	PR	CC	PR	CC	PR	CC
DD	1m	8s	2m	34s	-	-
Giraph	2m	1m	4m	1.5m	22.5m	4.5m
GraphX	6m	3.5m	18m	12m	-	-
TEGRA	**10s**	**5s**	**19s**	**7s**	**1.5m**	**18s**

Table 3.3: Ad-hoc analytics on big graphs. A '-' indicates the system failed to run the workload.

runs. The results are shown in table 3.3. DD works reasonably well when both the graph and the updates are small (hence generates less state). However, as the graph becomes larger, DD needs to push a large number of updates through the computation, and hence the state it generates becomes a huge bottleneck in its performance (on the largest graph, we were unable to get DD to work as it failed due to excessive memory usage during initial execution). In contrast, TEGRA is not only able to keep the state compact and scale to large graphs, but also provide significant benefits by using previous computation state. TEGRA carries the performance to larger graphs, the runtime increases linearly, which is intuitive for PageRank. Note that Giraph and GraphX recompute the entire results as they do not support incremental computations.

3.7.3 Timelapse & ICE

We evaluate the ability of Timelapse abstraction to provide efficient ways for graph-parallel phases to perform operations (§3.3.2) and the ICE computation model (§3.4.4).

Parallel computations. We develop a simple parallel computation model (§3.3.2) in which a query applied to a sequence of snapshots can be run in parallel on all the snapshots. That is, instead of running the query snapshot-by-snapshot, we use timelapse to compute across snapshots.

We create 20 snapshots of the Twitter graph by starting with 80% of the edges and adding 1% to it repeatedly. We then apply the connected components algorithm on these snapshots where we vary the number of snapshots on which the algorithm runs. Thus, the value in the X-axis of this plot indicates the number of snapshots included in the computation. In each run, we measure the time take to obtain the results on all the snapshots considered. For comparison, we use GraphX and apply the algorithm to each snapshot in a serial fashion. The results are depicted in fig. 3.10. We see that TEGRA significantly outperforms GraphX for a single snapshot due to its use of barrier execution

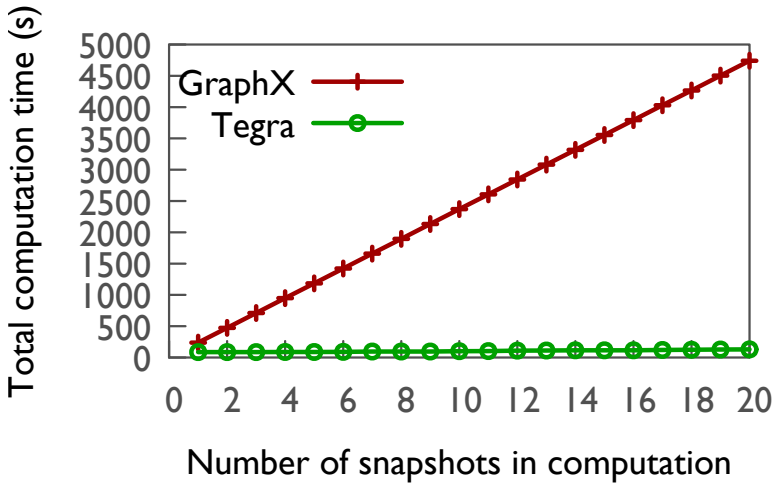


Figure 3.10: Timelapse can be used to optimize graph-parallel stage.

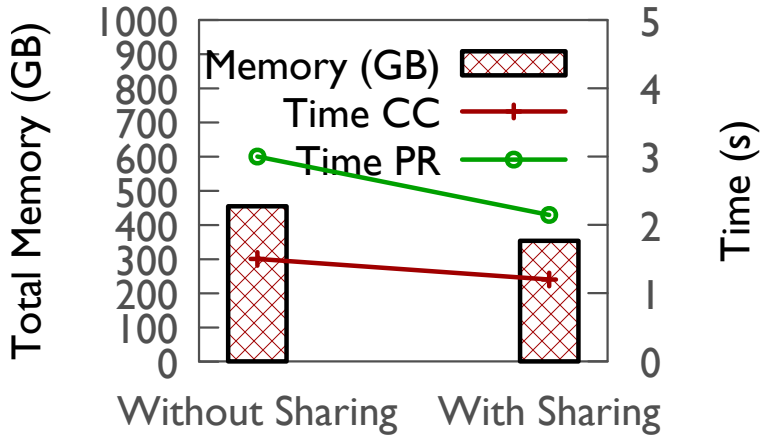


Figure 3.11: Sharing state across queries results in reduction in memory usage and also improvement in performance.

mode in Spark. Further, we see a linear trend with increasing number of snapshots. By sharing computation and communication, TEGRA is able to achieve up to 36× speedup.

Sharing state across queries: To evaluate how much benefits sharing state between different queries provides, we run an experiment with connected components and page rank. For these queries, the degree computation can be shared. We evaluate with and without this sharing. We use the Twitter graph and average the result of 10 runs of incremental computations on random snapshots. The results in fig. 3.11 show 20% and 30% reduction in memory usage and runtime.

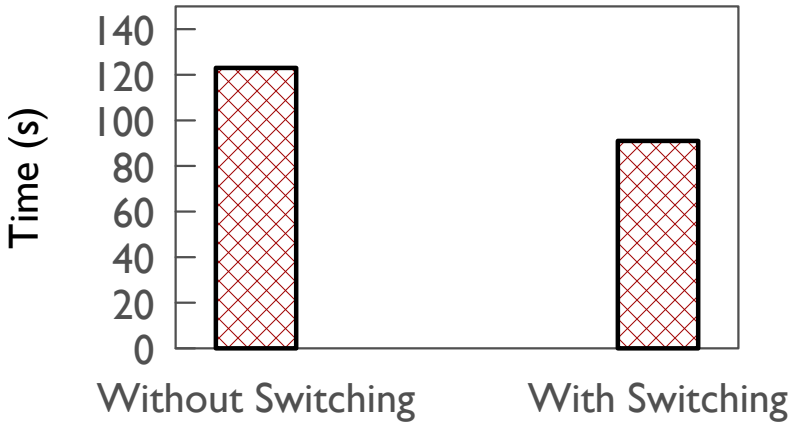


Figure 3.12: Incremental computations are not always useful. TEGRA can switch to full re-execution when this is the case.

ICE's switching capability: To test ICE's switching capability to full re-execution when incremental computations are not useful (§3.4.4), we run the connected component algorithm on Twitter graph. Then we introduce a batch of deletions in the largest components so that incremental computation executes on a large portion of the graph. We then make TEGRA recompute with and without the switching enabled and average the results over 10 such runs. The results are shown in fig. 3.12. We see that without the switching, TEGRA incurs a penalty—the incremental execution takes more time than fully re-executing the algorithm (which takes on average 90 seconds). With the switching enabled, TEGRA is easily able to identify that it needs to switch.

ICE's versatility: Since ICE differs from streaming engines (§3.4.3), it can also provide flexibility in how it uses state. For instance, if updates are monotonic (only additions), then ICE can simply restart from the last answer rather than using fully incremental computations. Figure 3.13 shows this on two algorithms on the UK graph. Both pagerank and connected components can benefit from monotonicity. Pagerank is faster since it only needs to converge within tolerance.

3.7.4 Tegra Shortcomings

Finally, we ask the question "What does TEGRA **not** do well?".

Purely Streaming Analysis: For this experiment, we consider an online query (§3.2) of connected components. To emulate a streaming graph, we first load the graph and continuously change 0.01% of the graph by adding and deleting edges.We assume that earlier results are available so that the system could perform incremental computation.

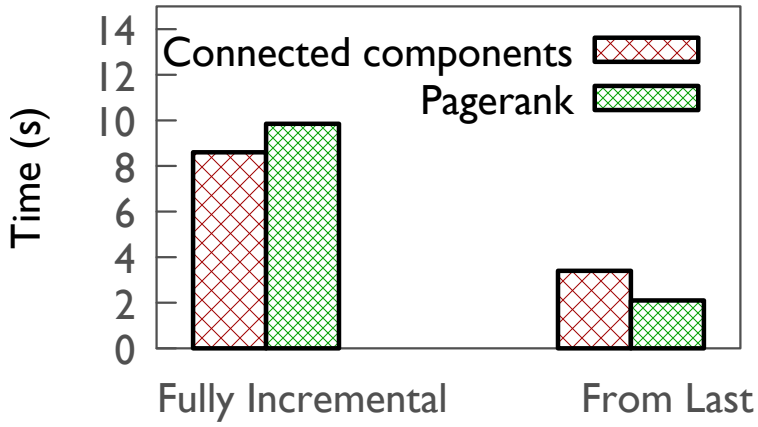


Figure 3.13: Monotonicity of updates (additions only) can be leveraged to speed up computations by starting from the last answer.

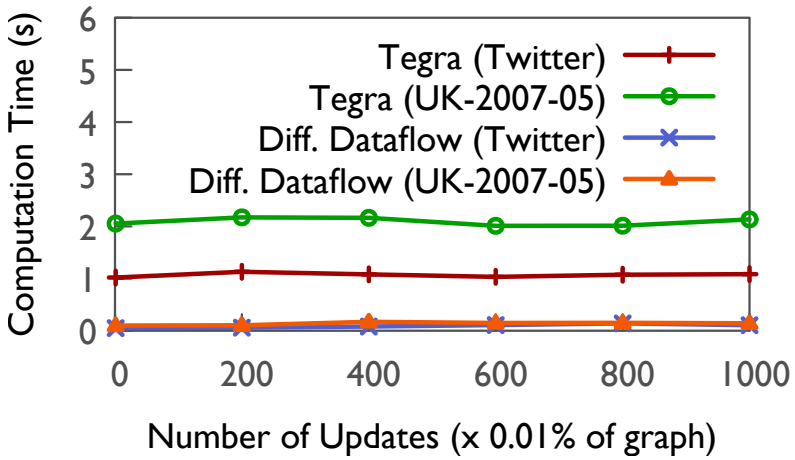


Figure 3.14: DD significantly outperforms TEGRA for purely streaming analysis.

We do this as follows: after a fixed number of updates, we stop and do a complete computation to create previous state. Then we do incremental computation from the next update. The average runtime of 10 runs is shown in fig. 3.14. We see that DD is significantly better than TEGRA for such workloads, providing 20-30X improvements. This is due to a combination of DD optimized for online queries (pushing each updates really fast through computation) and its Rust implementation. In contrast, TEGRA materializes the result after each computation, and also is tuned for updates in batches.

Purely Temporal Analysis: We also consider a purely temporal query. Here, we assume that the system knows the queries and the window before the start, and thus it has

optimized the data layout for the query. We run a simple query on a window size of 10 and compare TEGRA and Chlonos. Excluding processing time, we noticed that TEGRA takes around a 15% performance hit due to its use of tree structure.
COST Analysis: The COST metric [118] is not designed for incremental systems, but we note that TEGRA is able to match the performance of an optimized single threaded implementation using 4 machines, each with 8 cores and has a COST of 32 cores. However, TEGRA uses property graphs while the optimized implementation does not.

3.8 Related Work

(Transactional) Graph Stores: The problem of managing time-evovling graph has been studied in the context of graph stores [38, 123, 124, 180, 138]. These focus on optimizing point queries which retrieves graph entities and do not support storing multiple snapshots. This yields a different set of challenges compared to iterative graph analytics.
Managing Graph Snapshots: A lot of systems took the idea to manage snapshots for evolving graphs, so the problem is converted to analytics on a series of static graphs. DeltaGraph [98] proposes a hierarchical index that can manage multiple snapshots of a graph using deltas and event lists for efficient retrievals, but lacks the ability to do windowed iterative analytics. TAF [99] fixes this, but it is a specialized framework that does not provide a generalized incremental model or ad-hoc operations. LLAMA [114] uses a multiversion array to support incremental ingestion. It is a single machine system, and it is unclear how the multiversion array can be extended to support data parallel operations required for iterative analytics. Version Traveler [95] achieves swift switching between snapshots of a graph by loading the common subgraph in the compressed-sparse-row format and extending it with deltas. However, it does not support incremental computation. Chronos [76] and ImmortalGraph [120] optimizes for efficient computation across a series of snapshots. They propose an efficient model for processing temporal queries, and support snapshot storage of the graph on-disk using a hybrid model. While their technique reduces redundant computations in a given query, they cannot store and reuse intermediate computation results. Their in-memory layout of snapshots requires preprocessing and cannot support updates. Further, their incremental computation model does not support non-monotonic computations. None of these allow compactly representing computation state.
Incremental Maintenence on Evolving Graphs: Another important body of work are the streaming systems. CellIQ [91] is a specialized system for cellular network analytics, but it does not support ad-hoc analysis or compactly storing graph and state. Kineograph [45] supports constructing consistent snapshots of an evolving graph for streaming computa-

tions but does not allow ad-hoc analysis. WSP [191] focuses on streaming RDF queries. GraphInc [41] supports incremental graph processing using memoization of the messages in graph parallel computation, but does not support snapshot generation or maintenance. Kickstarter [175] and GraphBolt [117] supports edge deletions, but does not support ad-hoc analysis. Differential Dataflow [125, 121, 119] leverages indexed differences of data in its computation model to do non-monotonic incremental computations. However, it is challenging to do ad-hoc window operations using indexed differences (§3.2.2). As we demonstrate in our evaluation, compactly representing graph and computation state is the key to efficient ad-hoc window operations on evolving graphs.

Incremental View Maintenance (IVM): In databases, IVM algorithms [75, 30] maintain a consistent view of the database by reuse of computed results. However, they are tuned for different kinds of queries and not iterative graph computations. Further, they generate large intermediate state and hence require significant storage and computation cost [119].

Versioned File Systems (e.g., [161]) allow several versions of a file to exist at a time. However, they are focused on disk based files in contrast to in-memory efficiency.

3.9 Summary

In this chapter, we presented Tegra, a system that enables efficient ad-hoc window operations on evolving graphs. The key to Tegra's superior performance in such workloads is a compact, in-memory representation of both graph and intermediate computation state, and a computation model that can utilize it efficiently. For this, Tegra leverages persistent datastructures and builds DGSI, a versioned, distributed graph state store. It further proposes ICE, a general, non-monotonic iterative incremental computation model for graph algorithms. Finally, it enables users to access these states via a natural abstraction called Timelapse. Our evaluation shows that Tegra is able to outperform existing temporal and streaming graph systems significantly on ad-hoc window operations.

The storage solution presented in this chapter, DGSI, can be used as a standalone state store for dynamic connected data represented as an evolving property graph, and forms the storage layer for other systems presented in the rest of this dissertation. While this chapter focused on simple scenarios such as exact analysis and single datacenter processing, we look at more complex scenarios in later chapters.

Chapter 4

Pattern Mining in Dynamic Connected Data

4.1 Introduction

In the last chapter, we looked at how to enable *ad-hoc window* operations that is required to support *analysis queries* in dynamic connected data systems. This chapter looks at a different, equally important class of queries, *pattern mining queries*, similar to Taylor's effort to discover money laundering patterns in the transaction graph. Such queries are intractable even in static data for medium to large sized data sets. We describe the reason for this and propose a novel solution in the rest of this chapter.

Algorithms for graph processing can broadly be classified into two categories. The first, *graph analysis* algorithms, compute properties of a graph typically using neighborhood information. Examples of such algorithms include PageRank [132], community detection [63] and label propagation [203]. The second, *graph pattern mining* algorithms, discover structural patterns in a graph. Examples of graph pattern mining algorithms include motif finding [122], frequent sub-graph mining (FSM) [187] and clique mining [37]. Graph mining algorithms are used in applications like detecting similarity between graphlets [139] in social networking and for counting pattern frequencies to do credit card fraud detection.

Today, a deluge of graph processing frameworks exist, both in academia and open-source [111, 70, 105, 151, 71, 143, 150, 178, 39, 45, 76, 125, 114, 46, 199]. These frameworks typically provide high-level abstractions that make it easy for developers to implement many graph algorithms. A vast majority of the existing graph processing frameworks however have focused on graph analysis algorithms. These frameworks are fast and can scale out to handle very large graph analysis settings: for instance, GraM [182] can run

one iteration of page rank on a trillion-edge graph in 140 seconds in a cluster. In contrast, systems that support graph pattern mining fail to scale to even moderately sized graphs, and are slow, taking several hours to mine simple patterns [166, 57].

The main reason for the lack of the scalability in pattern mining is the underlying complexity of these algorithms—mining patterns requires complex computations and storing exponentially large intermediate candidate sets. For example, a graph with a million vertices may possibly contain 10^{17} triangles. While distributed graph-processing solutions are good candidates for processing such massive intermediate data, the need to do expensive joins to create candidates severely degrades performance. To overcome this, Arabesque [166] proposes new abstractions for graph mining in distributed settings that can significantly optimize how intermediate candidates are stored. However, even with these methods, Arabesque takes over 10 hours to *count* motifs in a graph with less than 1 billion edges.

We propose ASAP[1], a system that enables both *fast* and *scalable* pattern mining. ASAP is motivated by one key observation: *in many pattern mining tasks, it is often not necessary to output the exact answer*. For instance, in FSM the task is to find the *frequency* of subgraphs with an end-goal of ordering them by occurrences. Similarly, motif counting determines the number of occurrences of a given motif. In these scenarios, it is sufficient to provide an *almost* correct answer. Indeed, our conversations with a social network firm [147] revealed that their application for social graph similarity uses a count of similar graphlets [139]. Another company's [147] fraud detection system similarly counts the frequency of pattern occurrences. In both cases, an approximate count is good enough. Furthermore, it is not necessary to materialize *all* occurrences of a pattern[2]. Based on these use cases, we build a system for *approximate* graph pattern mining.

Approximate analytics is an area that has gathered attention in big data analytics [5, 67, 20], where the goal is to let the user trade-off accuracy for much faster results. The basic idea in approximation systems is to execute the *exact* algorithm on a small portion of the data, referred to as *samples*, and then rely on the statistical properties of these samples to compose partial results and/or error characteristics. The fundamental assumption underlying these systems is that there exists a relationship between the input size and the accuracy of the results which can be inferred. However, this assumption falls apart when applied to graph pattern mining. In particular, running the exact algorithm on a sampled graph may not result in a reduction of runtime or good estimation of error (§4.2.2).

Instead, in ASAP, we leverage graph approximation theory, which has a rich history of proposing approximation algorithms for mining specific patterns such as triangles.

[1] for *A S*wift *A*pproximate *P*attern-miner

[2] In fact, it may even be infeasible to output all embeddings of a pattern in a large graph.

ASAP exploits a key idea that approximate pattern mining can be viewed as equivalent to probabilistically sampling random instances of the pattern. Using this as a foundation, ASAP extends the state-of-the-art probabilistic approximation techniques to *general patterns* in a *distributed* setting. This lets ASAP massively parallelize instance sampling and provide a drastic reduction in run-times while sacrificing a small amount of accuracy. ASAP captures this technique in a simple API that allows users to plugin code to detect a single instance of the pattern and then automatically orchestrates computation while adjusting the error bounds based on the parallelism.

Further, ASAP makes pattern mining practical by supporting predicate matching and introducing caching techniques. In particular, ASAP allows mining for patterns where edges in the pattern satisfy a user-specified property. To further reduce the computation time, ASAP leverages the fact that in several mining tasks, such as motif finding, it is possible to cache partial patterns that are building blocks for many other patterns. Finally, an important problem in any approximation system is in allowing users to navigate the tradeoff between the result accuracy and latency. For this, ASAP presents a novel approach to build the Error-Latency Profile (ELP) for graph mining: it uses a small sample of the graph to obtain necessary information and applies Chernoff bound analysis to estimate the worst-case error profile for the original graph.

The combination of these techniques allows ASAP to outperform Arabesque [166], a state-of-the-art exact pattern mining solution by up to 77× on the LiveJournal graph while incurring less than 5% error. In addition, ASAP can scale to graphs with billions of edges—for instance, ASAP can count all the 6 patterns in 4-motifs on the Twitter (1.5B edges) and UK graph (3.7B edges) in 22 and 47 minutes, respectively, in a 16 machine cluster.

We make the following contributions in this work:

- We present ASAP, the first system to our knowledge, that does fast, scalable approximate graph pattern mining on large graphs. (§4.3)
- We develop a general API that allows users to mine any graph pattern and present techniques to automatically distribute executions on a cluster. (§4.4)
- We propose techniques that quickly infer the relationship between approximation error and latency, and show that it is accurate across many real-world graphs. (§4.5)
- We show that ASAP handles graphs with billions of edges, a scale that existing systems failed to reach. (§4.6)

4.2 Background & Motivation

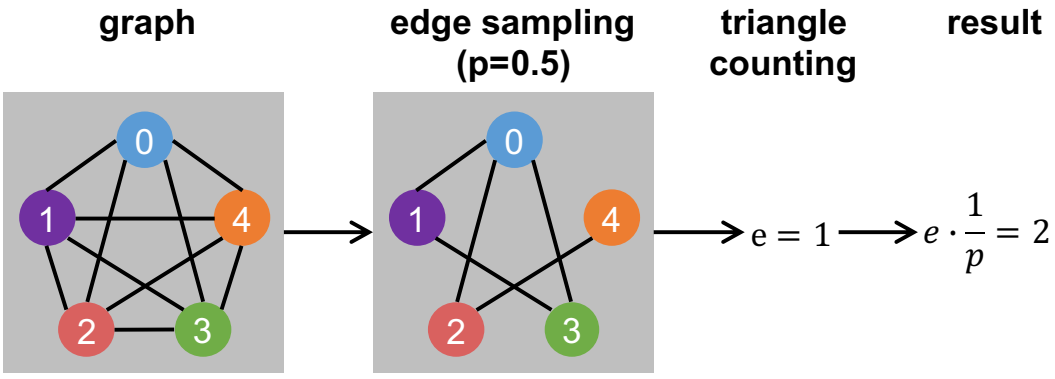
We begin by discussing graph pattern mining algorithms and then motivate the need for a new approach to approximate pattern mining. We then describe recent advancements in graph pattern mining theory that we leverage.

4.2.1 Graph Pattern Mining

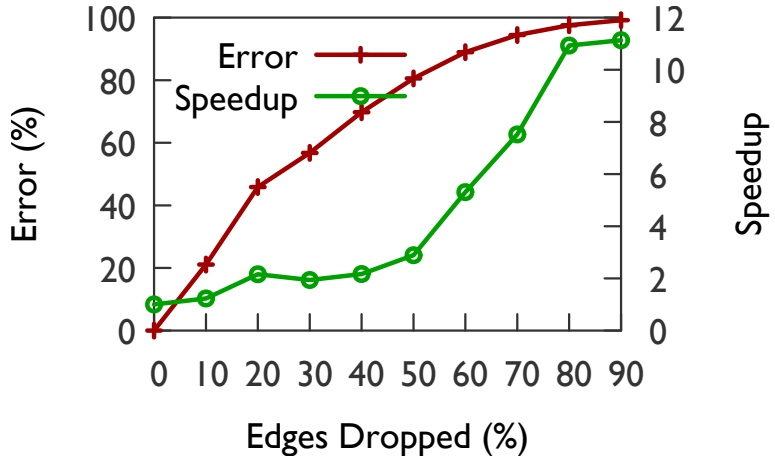
Mining patterns in a graph represent an important class of graph processing problems. Here, the objective is to find instances of a given pattern in a graph or graphs. The common way of representing graph data is in the form of a *property graph* [148], where user-defined properties are attached to the vertices and edges of the graph. A *pattern* is an arbitrary subgraph, and pattern mining algorithms aim to output all subgraphs, commonly referred to as *embeddings*, that match the input pattern. Matching is done via sub-graph isomorphism, which is known to be NP-complete. Several varieties of graph pattern mining problems exist, ranging from finding cliques to mining frequent subgraphs. We refer the reader to [166, 7] for an excellent, in-depth overview of graph mining algorithms.

A common approach to implement pattern mining algorithms is to iterate over all possible embeddings in the graph starting with the simplest pattern (e.g., a vertex or an edge). We can then check all *candidate* embeddings, and prune those that cannot be a part of the final answer. The resulting candidates are then expanded by adding one more vertex/edge, and the process is repeated until it is not possible to explore further. The obvious challenge in graph pattern mining, as opposed to graph analysis, is the exponentially large candidate set that needs to be checked.

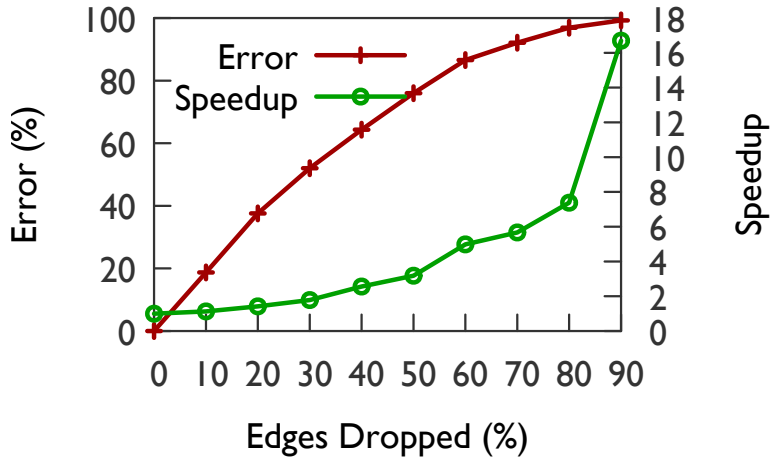
Distributed graph processing frameworks are built to process large graphs, and thus seem like an ideal candidate for this problem. Unfortunately when applied to graph mining problems, they face several challenges in managing the candidate set. Arabesque [166], a recently proposed distributed graph mining system, discusses these challenges in detail, and proposes solutions to tackle several of them. However, even Arabesque is unable to scale to large graphs due to the need to materialize candidates and exchange them between machines. As an example, Arabesque takes over 10 hours to count motifs of size 3 in a graph with less than a billion edges on a cluster of 20 machines, each having 256GB of memory.



(a) Uniform edge sampling.



(b) 3-chains in Twitter graph



(c) Triangles in UK graph

Figure 4.1: Simply extending approximate processing techniques to graph pattern mining does not work.

4.2.2 Approximate Pattern Mining

Approximate processing is an approach that has been used with tremendous success in solving similar problems in both the big data analytics [5, 67] and databases [43, 48, 49], and thus it is natural to explore similar techniques for graph pattern mining. However, simply extending existing approaches to graphs is insufficient.

The common underlying idea in approximate processing systems is to *sample the input* that a query or an algorithm works on. Several techniques for sampling the input exists, for instance, BlinkDB [5] leverages stratified sampling. To estimate the error, approximation systems rely on the assumption that the sample size relates to the error in the output (e.g., if we sample K items from the original input, then the error in aggregate queries, such as SUM, is inversely proportional to $\sqrt{K}$). It is straightforward to envision extending this approach to graph pattern mining—given a graph and a pattern to mine in the graph, we first sample the graph, and run the pattern mining algorithm on the sampled graph.

Figure 4.1a depicts the idea as applied to triangle counting. In this example, the input graph consists of 10 triangles. Using uniform sampling on the edges we obtain a graph with 50% of the edges. We can then apply triangle counting on this sample to get an answer 1. To scale this number to the actual graph, we can use several ways. One naive way is to double it, since we reduced the input by half. To verify the validity of the approach, we evaluated it on the Twitter graph [104] for finding 3-chains and the UK webgraph [34] graph for triangle counting. The relation between the sample size, error and the speedup compared to running on the original graph ($\frac{T_{orig}}{T_{sample}}$) is shown in figs. 4.1b and 4.1c respectively.

These results show the fundamental limitations of the approach. We see that there is no relation between the size of the graph (sample) and the error or the speedup. Even very small samples do not provide noticeable speedups, and conversely, even very large samples end up with significant errors. We conclude that the existing approximation approach of *running the exact algorithm on one or more samples of the input is incompatible with graph pattern mining*. Thus, we propose a new approach.

4.2.3 Graph Pattern Mining Theory

Graph theory community has spent significant efforts in studying various approximation techniques for *specific patterns*. The key idea in these approaches is to model the edges in the graph as a *stream* and *sample instances of a pattern* from the edge stream. Then the *probability of sampling* is used to bound the number of occurrences of the pattern.

There has been a large body of theoretical work on various algorithms to sample specific patterns and analysis to prove their bounds [133, 171, 40, 14, 135, 8, 93].

While the intuition of using such sampling to approximate pattern counts is straightforward, the technical details and the analysis are quite subtle. Since sampling once results in a large variance in the estimate, multiple rounds are required to bound the variance. Consider triangle counting as an example. Naively, one would design an technique that uniformly samples three edges from the graph without replacement. Since the probability of sampling one edge is $1/m$ in a graph of m edges, the probability of sampling three edges is $1/m^3$. If the sampled three edges form a triangle, we estimate the number of triangles to be m^3 (the expectation); otherwise, the estimation is 0. While such a sampling technique is unbiased, since m is large in practice, the probability that the sampling would find a triangle is very low and the variance of the result is very large. Obtaining an approximated count with high accuracy, would require a large number of trials, which not only consumes time but also memory.

Neighborhood sampling [135] is a recently proposed approach that provides a solution to this problem in the context of a specific graph pattern, triangle counting. The basic idea is to sample one edge and then gradually add more edges until the edges form a triangle or it becomes impossible to form the pattern. This can be analyzed by Bayesian probability [135]. Let's denote E as the event that a pattern is formed, $E_1, E_2, \ldots, E_k$ are the events that edges $e_1, e_2, \ldots, e_k$ are sampled and stored. Thus the probability of a pattern is actually sampled can be calculated as $Pr(E) = Pr(E_1 \cap E_2 \cdots \cap E_k) = Pr(E_1) \times Pr(E_2|E_1) \cdots \times Pr(E_k|E_1, \ldots, E_{k-1})$. Intuitively, compared to the naive sampling, neighborhood sampling increases the probability that each trial would find an instance of the given pattern, and thus requires fewer estimations to achieve the same accuracy.

Example: Triangle Counting

To illustrate neighborhood sampling, we will revisit the triangle counting example discussed earlier. To sample a triangle from a graph with m edges, we need three edges:

- **First edge l_0.** Uniformly sample one edge from the graph as l_0. The sampling probability $Pr(l_0) = 1/m$.
- **Second edge l_1.** Given that l_0 is already sampled, we uniformly sample one of l_0's adjacent edges (neighbors) from the graph, which we call l_1. Note that neighborhood sampling depends on the ordering of edges in the stream and l_1 appears after l_0 here. The sampling probability $Pr(l_1|l_0) = 1/c$, where c is the number l_0's neighbors appearing after l_0.

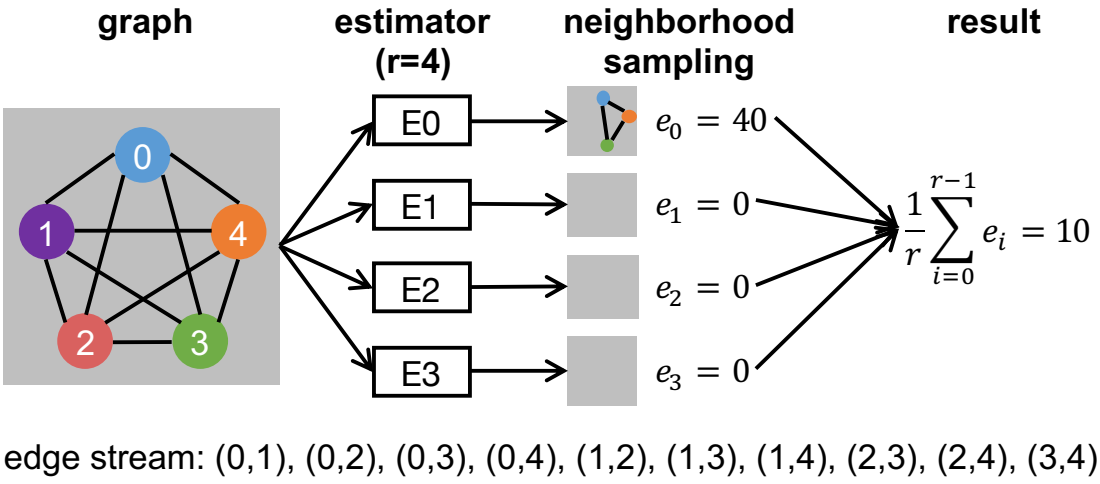


Figure 4.2: Triangle count by neighborhood sampling

- **Third edge l_2.** Find l_2 to finish if edges l_2, l_1, l_0 form a triangle and l_2 appears after l_1 in the stream. If such a triangle is sampled, the sampling probability is $\Pr(l_0 \cap l_1 \cap l_2) = \Pr(l_0) \times \Pr(l_1|l_0) \times \Pr(l_2|l_0, l_1) = 1/mc$.

The above technique describes the behaviors of one sampling trial. For each trial, if it successfully samples a triangle, converting probabilities to expectation, $e_i = mc$ will be the estimate of the triangles in the graph. For a total of r trials, $\frac{1}{r}\sum_r e_i$ is output as the approximate result. Figure 4.2 presents an example of a graph with five nodes.

4.2.4 Challenges

While the neighborhood sampling algorithm described above has good theoretical properties, there are a number of challenges in building a general system for large-scale approximate graph mining. First, neighborhood sampling was proposed in the context of a specific graph pattern (triangle counting). Therefore, to be of practical use, ASAP needs to generalize neighborhood sampling to other patterns. Second, neighborhood sampling and its analysis assume that the graph is stored in a single machine. ASAP focuses on large-scale, distributed graph processing, and for this it needs to extend neighborhood sampling to computer clusters. Third, neighborhood sampling assumes homogeneous vertices and edges. Real-world graphs are *property graphs*, and in practice pattern mining queries require *predicate matching* which needs the technique to be aware of vertex and edge types and properties. Finally, as in any approximate processing system, ASAP needs

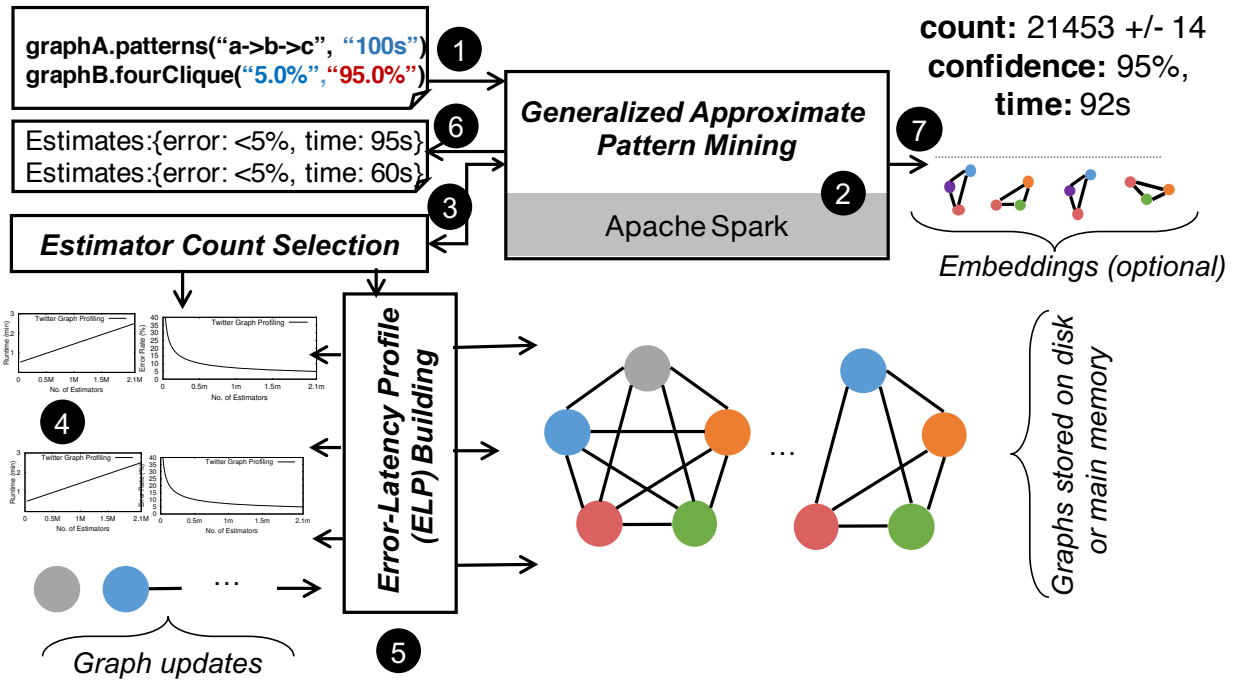


Figure 4.3: ASAP architecture

to allow the end user to trade-off accuracy for latency and hence needs to understand the relation between run-time and error in a distributed setting.

4.3 ASAP Overview

We design ASAP, a system that facilitates fast and scalable approximate pattern mining. Figure 4.3 shows the overall architecture of ASAP. We provide a brief overview of the different components, and how users leverage ASAP to do approximate pattern mining in this section to aid the reader in following the rest of this chapter.

User interface. ASAP allows the users to tradeoff accuracy for result latency. Specifically, a user can perform pattern mining tasks using the following two modes ❶:

- **Time budget T.** The user specifies a time budget T, and ASAP returns the most accurate answer within T with a error rate guarantee e and a configurable confidence level (default of 95%).
- **Error budget ϵ.** The user gives an error budget ϵ and confidence level, and ASAP returns an answer within ϵ in the shortest time possible.

Before running the algorithm, ASAP first returns to the user its estimates on the time or error bounds it can achieve ❻. After user approves the estimates, the algorithm is run and the result presented to the user consists of the count, confidence level and the actual run time ❼. Users can also optionally ask to output actual (potentially large number of) embeddings of the pattern found.

Development framework. All pattern mining programs in ASAP are versions of generalized approximate pattern mining ❷ we describe in detail in §4.4. ASAP provides a standard library of implementations for several common patterns such as triangles, cliques and chains. To allow developers to write program to mine *any* pattern, ASAP further provides a simple API that lets them utilize our approximate mining technique (§4.4.1). Using the API, developers simply need to write a program that finds a *single instance* of the pattern they are interested in, which we refer to as *estimator*. In a nutshell, our approximate mining approach depends on running multiple such estimators in parallel.

Error-Latency Profile (ELP). In order to run a user program, ASAP first must find out how many estimators it needs to run for the given bounds ❸. To do this, ASAP builds an ELP. If the ELP is available for a graph, it simply queries the ELP to find the number of estimators ❹. Otherwise, the system builds a new ELP ❺ using a novel technique that is extremely fast and can be done online. We detail our ELP building technique in §4.5. Since this phase is fast, ASAP can also accommodate graph updates; on large changes, we simply rebuild the ELP.

System runtime. Once ASAP determines the number of estimators necessary to achieve the required error or time bounds, it executes the approximate mining program using a distributed runtime built on Apache Spark [192, 193].

4.4 Approximate Pattern Mining in ASAP

We now present how ASAP enables large-scale graph pattern mining using neighborhood sampling as a foundation. We first describe our programming abstraction(§4.4.1) that generalizes neighborhood sampling. Then, we describe how ASAP handles errors that arise in distributed processing(§4.4.2). Finally, we show how ASAP can handle queries with predicates on edges or vertices(§4.4.3).

4.4.1 Extending to General Patterns

To extend the neighborhood sampling technique to general patterns, we leverage one simple observation: at a high level, neighborhood sampling can be viewed as consisting

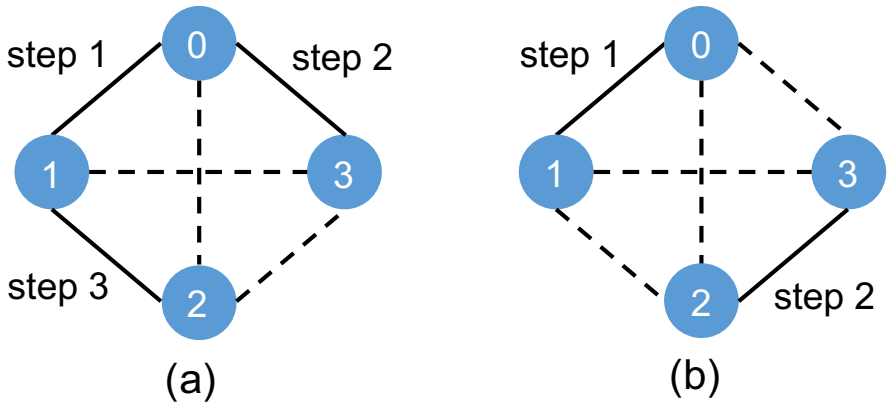


Figure 4.4: Two ways to sample four cliques. (a) Sample two adjacent edges $(0,1)$ and $(0,3)$, sample another adjacent edge $(1,2)$, and wait for the other three edges. (b) Sample two disjoint edges $(0,1)$ and $(2,3)$, and wait for the other four edges.

of two phases, *sampling* phase and *closing* phase. In the *sampling* phase, we select an edge in one of two ways by treating the graph as an ordered stream of edges: (a) sample an edge randomly; (b) sample an edge that is adjacent to any previously sampled edges, from the remainder of the stream. In the *closing* phase, we wait for one or more specific edges to complete the pattern.

The probability of sampling a pattern can be computed from these two phases. The closing phase always has a probability of 1 or 0, depending on whether it finds the edges it is waiting for. The probability of the sampling phase depends on how the initial pattern is formed and is a choice made by the developer. For a general graph pattern with multiple nodes, there can be multiple ways to form the pattern. For example, there are two ways to sample a four-clique with different probabilities, as shown in Figure 4.4. (i) In the first case, the sampling phase finds three adjacent edges, and the closing phase waits for rest three edges to come, in order to form the pattern. The sampling probability is $\frac{1}{mc_1c_2}$, where c_1 is the number of the first edge's neighbors and c_2 represents the neighbor count of the first and the second edges. (ii) In the second case, the sampling phase finds two disjoint edges, and the closing phase waits for other four edges to form the pattern. The sampling probability in this case is $\frac{1}{m^2}$.

Analysis of General Patterns

We now show how neighborhood sampling, when captured using the two phases, can extend to general patterns.

Definition 4.4.1 (General Pattern). *We define a "general pattern" as a set of* k *connected vertices that form a subgraph in a given graph.*

First, let's consider how an estimator can (possibly) find any general patterns. We show how to sample one general pattern from the graph uniformly with a certain success probability, taking 2 to 5-node patterns as examples. Then, we turn to the problem of maintaining $r \leqslant 1$ pattern(s) sampled with replacement from the graph. We sample r patterns and a reasonably large r will yield a count estimate with good accuracy. For the convenience of the analysis, we define the following notations: input graph $G = (V, E)$ has m edges and n vertices, and we denote the occurrence of a given pattern in G as $f(G)$. A pattern $p = \{e_i, e_j, \dots\}$ contains a set of ordered edges, i.e., e_i arrives before e_j when $i < j$. When describing the operation of an estimator, $c(e)$ denotes the number of edges adjacent to e and appearing after e, and c_i is $c(e_1, \dots, e_i)$ for any $i \geqslant 1$. For a given a pattern p^* with k^* vertices, the technique of neighborhood sampling produces p^* with probability $\Pr[p = p^*, k = k^*]$. The goal of one estimator is to fix all the vertices that form the pattern, and complete the pattern if possible.

Lemma 4.4.2. *Let* p^* *be a k-node pattern in the graph. The probability of detecting the pattern* $p = p^*$ *depends on* k *and the different ways to sample using neighborhood sampling technique.*
(1) When $k = 2$*, the probability that* $p = p^*$ *after processing all edges in the graph by all possible neighborhood sampling ways is*

$$\Pr[p = p^*, k = 2] = \frac{1}{m}$$

(2) When $k = 3$*, the probability that* $p = p^*$ *is*

$$\Pr[p = p^*, k = 3] = \frac{1}{m \cdot c_1}$$

(3) When $k = 4$*, the probability that* $p = p^*$ *is*

$$\Pr[p = p^*, k = 4] = \frac{1}{m^2} \text{ (Type-I) or } \frac{1}{m \cdot c_1 \cdot c_2} \text{ (Type-II)}$$

(4) When $k = 5$*, the probability that* $p = p^*$ *is*

$$\begin{aligned} \Pr[p = p^*, k = 5] &= \frac{1}{m^2 \cdot c_1} \text{ (Type-I)} \\ \text{or } &= \frac{1}{m^2 \cdot c_2} \text{ (Type-II.a)} \\ \text{or } &= \frac{1}{m \cdot c_1 \cdot c_2 \cdot c_3} \text{ (Type-II.b)} \end{aligned}$$

Proof. Since a pattern is connected, the operations in the sampling phase are able to reach all nodes in a sampled pattern. To fix such a pattern, the neighborhood sampling needs to confirm all the vertices that form the pattern. Once the vertices are found, the probability of completing such a pattern is fixed.

When $k = 2$, let $p^* = \{e_1\}$ be an edge in the graph. Let $\mathcal{E}_1$ be the event that e_1 is found by neighborhood sampling. There is only one way to fix two vertices of the pattern—uniformly sampling an edge from the graph. By reservoir sampling, we claim that

$$\Pr[p = p^*, k = 2] = \Pr[\mathcal{E}_1] = \frac{1}{m}$$

When $k = 3$, we need to fix one more vertex beyond the case of $k = 2$. As shown in [135], we need to sample an edge e_2 from e_1's neighbors that occur in the stream after e_1. Let $\mathcal{E}_2$ be the event that e_2 is found. Since $\Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{1}{c(e_1)}$,

$$\Pr[p = p^*, k = 3] = \Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2|\mathcal{E}_1] = \frac{1}{m \cdot c(e_1)}$$

When $k = 4$, we require one more step from the case of $k = 2$ or the case of $k = 3$, from extending neighborhood sampling. By extending from the case of $k = 2$ (denoted as Type-I), two more vertices are needed to fix a 4-node pattern. In Type-I, we independently find another edge e_2^* that is not adjacent to the sampled edge e_1. Let $\mathcal{E}_2^*$ be the event that e_2^* is found. Since $\Pr[\mathcal{E}_2^*|\mathcal{E}_1] = \frac{1}{m}$,

$$\begin{aligned}\Pr[p = p^*, k = 4] &= \Pr[p = p^*, k = 2] * \Pr[\mathcal{E}_2^*|\mathcal{E}_1] \\ &= \frac{1}{m^2} \text{ (Type-I)}\end{aligned}$$

When extending from the case $k = 2$ (denoted as Type-II), one more vertex is needed to fix a 4-node pattern. In Type-II, we sample a "neighbor" e_3 that comes after e_1 and e_2. Let $\mathcal{E}_3$ be the event that e_3 is found. Since e_3 is sampled uniformly from the neighbors of e_1 and e_2 and is appearing after e_1, e_2, $\Pr[\mathcal{E}_3|\mathcal{E}_1, \mathcal{E}_2] = \frac{1}{c(e_1,e_2)}$. Thus,

$$\begin{aligned}\Pr[p = p^*, k = 4] &= \Pr[p = p^*, k = 3] \cdot \Pr[\mathcal{E}_3|\mathcal{E}_1, \mathcal{E}_2] \\ &= \frac{1}{m \cdot c(e_1) \cdot c(e_1, e_2)} \text{ (Type-II)}\end{aligned}$$

When $k = 5$, we again need one more step from the case $k = 3$ or the case $k = 4$. By extending from $k = 3$ (denoted as Type-I), we require two separate vertices to fix a 5-node pattern. In Type-I, we independently sample another edge e_3^* that is not adjacent to e_1, e_2.

Let $\mathcal{E}_3^*$ be the event that e_3^* is found. $\Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2] = \frac{1}{m}$. Therefore,

$$\begin{aligned}\Pr[p = p^*, k = 5] &= \Pr[p = p^*, k = 3] * \Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2] \\ &= \frac{1}{m^2 \cdot c(e_1)} \text{ (Type-I)}\end{aligned}$$

When extending from the case $k = 4$, we need to consider the two types separately. By extending Type-I of case $k = 4$ (denoted as Type-II.a), we need one more vertex to construct a 5-node pattern and thus we sample a neighboring edge e_4. Let $\mathcal{E}_4$ be the event that e_4 is found. Since e_4 is sampled from the neighbors of e_1, e_2,

$$\begin{aligned}\Pr[p = p^*, k = 5] &= \Pr[p = p^*, k = 4] * \Pr[\mathcal{E}_4|\mathcal{E}_1, \mathcal{E}_2^*] \\ &= \frac{1}{m^2 \cdot c(e_1, e_2)} \text{ (Type-II.a)}\end{aligned}$$

Similarly, by extending Type-II of case $k = 4$ (denoted as Type-II.b),

$$\Pr[p = p^*, k = 5] = \frac{1}{m \cdot c(e_1) \cdot c(e_1, e_2) \cdot c(e_1, e_2, e_3)}$$

□

Lemma 4.4.3. *For pattern p^* with k^* nodes, let's define*

$$\tilde{t} = \begin{cases} \frac{1}{\Pr[p=p^*, k=k^*]} & \text{if } p \neq \emptyset \\ 0 & \text{if } p = \emptyset \end{cases}$$

Thus, $E[\tilde{t}] = f(G)$.

Proof. By Lemma 4.4.2, we know that one estimator samples a particular pattern p^* with probability $\Pr[p = p^*, k = k^*]$. Let $p(G)$ be the set of a given pattern in the graph,

$$E[\tilde{t}] = \sum_{p^* \in p(G)} \tilde{t}(p \neq \emptyset) \cdot \Pr[p = p^*, k = k^*] = |p(G)| = f(G)$$

□

The estimated count is the average of the input of all estimators. Now, we consider how many estimators are needed to maintain an ϵ error guarantee.

Theorem 4.4.4. *Let $r \geqslant 1$, $0 < \epsilon \leqslant 1$, and $0 < \delta \leqslant 1$. There is an $O(r)$-space bounded algorithm that return an ϵ-approximation to the count of a k-node pattern, with probability at least $1 - \delta$. For a certain ϵ, when $k = 4$, we need $r \geqslant \frac{C_1 m^2}{f(G)}$ Type-I estimators, or $r \geqslant \frac{C_2 m \Delta^2}{f(G)}$ Type-II estimators for some constants C_1 and C_2, to achieve ϵ-approximation in the worst case; When $k = 5$, we need $r \geqslant \frac{C_3 m^2 \Delta}{f(G)}$ Type-I estimators, or $r \geqslant \frac{C_4 m^2 \Delta}{f(G)}$ Type-II.a estimators, or $r \geqslant \frac{C_5 m \Delta^3}{f(G)}$ Type-II.b estimators, for some constants C_3, C_4, C_5 in the worst case.*

Proof. Let's first consider the case $k = 4$. Let X_i for $i = 1, \ldots, r$ be the output value of i-th estimator. Let $\bar{X} = \frac{1}{r}\sum_{i=1}^{r} X_i$ be the average of r estimators. By Lemma 4.4.3, we know that $E[X_i] = f(G)$ and $E[\bar{X}] = f(G)$. From the properties of graph G, we have $c(e) \leqslant \Delta$ for $\forall e \in E$, where Δ is the maximum degree (note that in practice Δ isn't a tight bound for the edge neighbor information). In Type-I, $X_i \leqslant m^2$ and we construct random variables $Y_i = \frac{X_i}{m^2}$ such that $Y_i = [0, 1]$. Let $Y = \sum_{i=1}^{r} Y_i$ and $E[Y] = \frac{f(G)r}{m^2}$. Thus the probability that the estimated number of patterns has a more than ϵ relative error off its expectation $f(G)$ is $Pr[\bar{X} > (1+\epsilon)f(G)] \leqslant \frac{\delta}{2}$, which is at most

$$Pr[\sum_{i=1}^{r} Y_i > (1+\epsilon)E[Y]] \leqslant e^{-\frac{\epsilon^2}{2+\epsilon}E[Y]} \leqslant e^{-\frac{\epsilon^2}{3}E[Y]} \leqslant \frac{\delta}{2}$$

by Chernoff bound. Thus $r \geqslant \frac{3m^2}{\epsilon^2 f(G)} \cdot \ln\frac{2}{\delta}$. Similarly, this lower bound of r holds for $Pr[\bar{X} < (1-\epsilon)f(G)]$.

In Type-II, $X_i \leqslant 6m\Delta^2$. Let $Y_i = \frac{X_i}{6m\Delta^2}$ such that $Y_i = [0, 1]$. Let $Y = \sum_{i=1}^{r} Y_i$ and $E[Y] = \frac{f(G)r}{6m\Delta^2}$. By Chernoff bound, $r \geqslant \frac{18m\Delta^2}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$. Similarly, when $k = 5$, we (theoretically) need $\frac{6m^2\Delta}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-I estimators, $\frac{12m^2\Delta}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-II.a estimators, and $\frac{24m\Delta^3}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$ Type-II.b estimators. Since each estimator stores $O(1)$ edges, the total memory is $O(r)$. □

Programming API

ASAP automates the process of computing the probability of finding a pattern, and derives an expectation from it by providing a simple API that captures two phases. The API, shown in Table 4.1, consists of the following five functions:

- `SampleVertex` uniformly samples one vertex from the graph. It takes no input, and outputs v and p, where v is the sampled vertex, and p is the probability that sampled v, which is the inverse of the number of vertices.
- `SampleEdge` uniformly samples one edge from the graph. It also takes no input, and outputs e and p, where e is the sampled edge, and p is the sampling probability, which is the inverse of the number of edges of the graph.
- `ConditionalSampleVertex` conditionally samples one vertex from the graph, given $subgraph$ as input. It outputs v and p, where v is the sampled vertex and p is the probability to sample v given that $subgraph$ is already sampled.

API	Description
`sampleVertex`: ()→(v, p)	Uniformly sample one vertex from the graph.
`SampleEdge`: ()→(e, p)	Uniformly sample one edge from the graph.
`ConditionalSampleVertex`: (`subgraph`)→(v, p)	Uniformly sample a vertex that appears after a sampled subgraph.
`ConditionalSampleEdge`: (`subgraph`)→(e, p)	Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.
`ConditionalClose`: (`subgraph`, `subgraph`) →$boolean$	Given a sampled subgraph, check if another subgraph that appears later in the order can be formed.

Table 4.1: ASAP's Approximate Pattern Mining API.

- `ConditionalSampleEdge`(`subgraph`) conditionally samples one edge adjacent to $subgraph$ from the graph, given that $subgraph$ is already sampled. It outputs e and p, where e is the sampled edge and p is the probability to sample e given $subgraph$.
- `ConditionalClose`(`subgraph`, `subgraph`) waits for edges that appear after the first $subgraph$ to form the second $subgraph$. It takes the two subgraphs as input and outputs yes/no, which is a boolean value indicating whether the second $subgraph$ can be formed. This function is usually used as the final step to sample a pattern where all nodes of a possible instance have been fixed (thereby fixing the edges needed to complete that instance of the pattern) and the sampling process only awaits the additional edges to form the pattern.

These five APIs capture the two phases in neighborhood sampling and can be used to develop pattern mining algorithms. To illustrate the use of these APIs, we describe how they can be used to write two representative graph patterns, shown in Figure 4.5.

Chain. Using our API to write a sampling function for counting three-node chains is straightforward. It only includes two steps. In the first step, we use `SampleEdge`() to uniformly sample one edge from the graph (line 1). In the second step, given the first sampled edge, we use `ConditionalSampleEdge` (`subgraph`) to find the second edge of the three-node chain, where `subgraph` is set to be the first sampled edge (line 2). Finally, if the algorithm cannot find e_2 to form a chain with e_1 (line 3), it estimates the number of three-node chains to be 0; otherwise, since the probability to get e_1 and e_2 is $p_1 \cdot p_2$, it estimates the number of chains to be $1/(p_1 \cdot p_2)$.

SampleThreeNodeChain

```
(e1, p1) = SampleEdge()
(e2, p2) = ConditionalSampleEdge(Subgraph(e1))
    ↪
if (!e2)
  return 0
else
  return 1/(p1.p2)
```

SampleFourCliqueType1

```
(e1, p1) = SampleEdge()
(e2, p2) = ConditionalSampleEdge(Subgraph(e1))
    ↪
if (!e2) return 0
(e3, p3) = ConditionalSampleEdge(Subgraph(e1,
    ↪  e2))
if (!e3) return 0
subgraph1 = Subgraph(e1,e2,e3)
subgraph2 = FourClique(e1,e2,e3)-subgraph1
if (ConditionalClose(subgraph1,subgraph2))
  return 1/(p1.p2.p3)
else return 0
```

Figure 4.5: Example approximate pattern mining programs written using ASAP API

Four clique. Similarly, we can extend the algorithm of sampling three node chains to sample four cliques. We first sample a three-node chain (line 1-2). Then we sample an adjacent edge of this chain to find the fourth node (line 4). Again, during the three steps, if any edges were not sampled, the function would return 0 as no cliques would be found (line 3 and 5). Given e_1, e_2 and e_3, all the four nodes are fixed. Therefore, the function only needs to wait for all edges to form a clique (line 8-9). If the clique is formed, it estimates the number of cliques to be $1/(p_1 \cdot p_2 \cdot p_3)$; otherwise, it returns 0 (line 10). Figure 4.4(a) illustrates this sampling procedure (CliqueType1).

4.4.2 Applying to Distributed Settings

Capturing general graph pattern mining using the simple two phase API allows ASAP to extend pattern mining to distributed settings in a seamless fashion. Intuitively, each execution of the user program can be viewed as an instance of the sampling process. To scale this up, ASAP needs to do two things. First, it needs to parallelize the sampling processes, and second, it needs to combine the outputs in a meaningful fashion that preserves the approximation theory.

For parallelizing the pattern mining tasks, ASAP's runtime takes the pattern mining program and wraps it into an *estimator*[3] task. ASAP first partitions the vertices in the graph across machines and executes many copies of the estimator task using standard dataflow operations: *map* and *reduce*. In the map phase, ASAP schedules several copies of the estimator task on each of the machines. Each estimator task operates on the local subgraph in each machine and produces an output, which is a partial count. ASAP's

[3]Since each program is providing an estimate of the final answer.

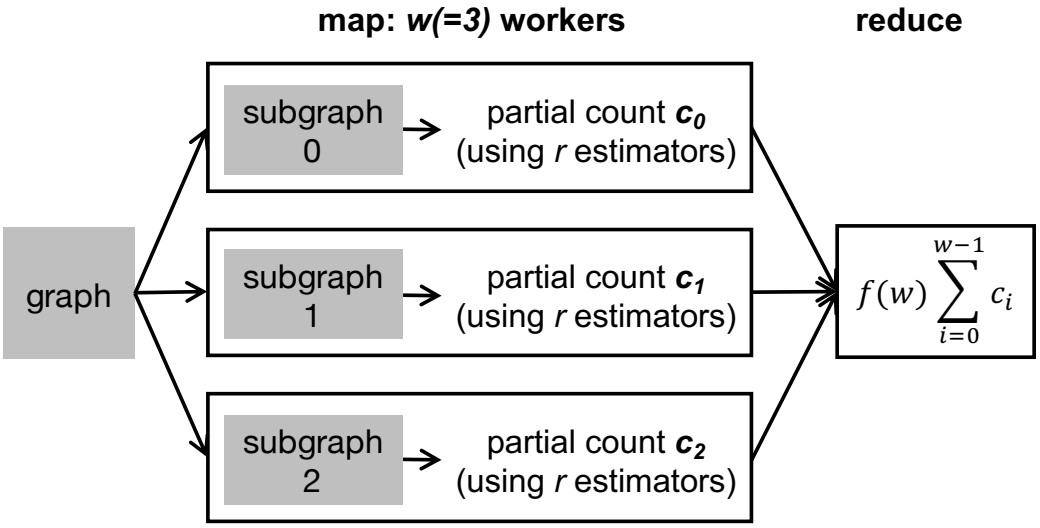


Figure 4.6: Runtime with graph partition.

runtime ensures that each estimator in a machine sees the graph's edges and vertices in the *same order*, which is important for the sampling process to produce correct results. Note that although every estimator in each partition sees the graph in the same order, there is *no restriction* on what the order might be (e.g., there is no sorting requirement), thus ASAP uses a random ordering which is fast and requires no pre-processing of the graph. Once this is completed, ASAP runs a reduce task to combine the partial counts and obtain the final answer. This is depicted in fig. 4.6. This massively parallel execution is one of the reasons for huge latency reduction in ASAP. Since the input to the reduce phase is simply an array of numbers, ASAP's shuffle is extremely light-weight, compared to a system that produces exact answers (and needs to exchange intermediate patterns).
Handling Underestimation. Only summing up the partial counts in the reduce phase underestimates the total number of instances, because when vertices are partitioned to the workers, the instances that span across the partitions are not counted. This results in our technique underestimating the results, and makes the theoretical bounds in neighborhood sampling invalid. Thus, ASAP needs to estimate the error incurred due to distributed execution and incorporate that in the total error analysis.

We use probability theory to do this estimation. We enforce that the vertices in the graph are uniformly randomly distributed across the machines. ASAP is not affected by the normal shortcomings of random vertex partitioning [70] as the amount of data communication is independent of partitioning scheme used. In this case random vertex partitioning is in fact simple to implement, and allows us to theoretically analyze the underestimation.

The theoretical proof for handling the underestimation is outside the scope of this work. Intuitively, we can think of the random vertex partitioning into w workers as uniform vertex coloring from w available colors. Vertices with the same color are at the same worker and each worker estimates patterns locally on its monochromatic vertices. By doing this coloring, the occurrence of a pattern has been reduced by a factor of $1/f(w)$, where f is a function of the number of workers and the pattern. For instance, a locally sampled triangle has three monochromatic vertices and the probability that this happens among all triangles is $1/w^2$. Thus by the linearity of expectation, each such triangle is scaled by $f(w) = w^2$. A rigorous proof on the maximum possible w with small errors (in practice w can be $>> 100$), can be shown using concentration bounds and Hajnal-Szemerédi Theorem [133]. Similarly, each monochromatic 4-clique is scaled by $f(w) = w^3$ and $f(w)$ can be computed for any given pattern.

4.4.3 Advanced Mining Patterns

Predicate Matching. In property graphs, the edges and vertices contain properties; and thus many real-world mining queries require that matching patterns satisfy some predicates. For example, a *predicate query* might ask for the count of all four cliques on the graph where every vertex in the clique is of a certain type. ASAP supports two types of predicates on the pattern's vertices and edges **all** and **atleast-one**.

For "all" predicate, queries specify a predicate that is applied to *every vertex or edge*. For example, such query may ask for "four cliques where all vertices have a weight of atleast 10". To execute such queries, ASAP introduces a *filtering* phase where the predicate condition is applied before the execution of the pattern mining task. This results in a new graph which consists only of vertices and edges that satisfy the predicate. On this new graph, ASAP runs the pattern mining algorithm. Thus, the "all" predicate query does not require any changes to ASAP's pattern mining algorithm.

The "atleast-one" predicate allows specifying a condition that *atleast* one of the vertices or edges in the pattern satisfies. An example of such a query is "four cliques where atleast one edge has a weight of 10". To execute such predicate queries, we modify the execution to take two passes on the edge list. In the first pass, edges that match the predicate are copied from the `original` edge list to a `matched` edge list. Every entry in the `matched` list is a tuple, `(edge, pos)`, where `pos` is the position in the original list where the matched edge appears. In the second pass, every estimator picks the first edge randomly from the `matched` list. This ensures that the pattern found by the estimator (if it finds one) satisfies the predicate. For the second edge onwards, the estimator uses the `original` list but starts the search from the position at which the first matched edge was found. This ensures that ASAP's probability analysis to estimate the error holds.

Motif mining. Another query used in many real-world workloads is to find all patterns with a certain number of vertices. We define these as *motif queries*; for example a 3-motif query will look for two patterns, triangles and 3-chains. Similarly a 4-motif query looks for six patterns [146]. For motif mining we notice that several patterns have the same underlying *building block*. For example, in 4-motifs, 3-chains are used in many of the constituent patterns. To improve performance, ASAP saves the *sampling* phase's state for the building block pattern. This state includes (i) the currently sampled edges, (ii) the probability of sampling at that point, and (iii) the position in the edge list up to which the estimator has traversed. All the patterns that use this building block are then executed starting from the saved state. This technique can significantly speedup the execution of motif mining queries and we evaluate this in Section 4.6.2.

Refining accuracy. In many mining tasks, it is common for the user to first ask for a low accuracy answer, followed by a higher accuracy. For example, users performing exploratory analysis on graph data often would like to iteratively refine the queries. In such settings, ASAP caches the state of the estimator from previous runs. For instance, if a query with an error bound of 10% was executed using 1 million estimators, ASAP saves the output from these estimators. Later, when the same pattern is being queried, but with an error bound of 5% that requires 3 million estimators, ASAP only needs to launch 2 million, and can reuse the first 1 million.

4.5 Building the Error-Latency Profile (ELP)

A key feature in any approximate processing system is allowing users to trade-off accuracy for result latency. To do this for graph mining, we need to understand the relation between running time and error.

In ASAP's general, distributed graph pattern mining technique described earlier, the only configurable parameter is the number of *estimator* processes used for a mining task. By using r estimators and making r sufficient large, ASAP is able to get results with bounded errors. Since an estimator takes computation and memory resource to sample a pattern, picking the number of estimators r provides a trade-off between result accuracy and resource consumption. In other words, setting a specific number of estimators, N_e, results in a fixed runtime and an error within a certain bound. As an example, fig. 4.7 depicts the relation between the number of estimators, runtime and error for triangle counting run on the Twitter graph [104]. To enable the user to traverse this trade-off, ASAP needs to determine the correct number of estimators given an error or time budget.

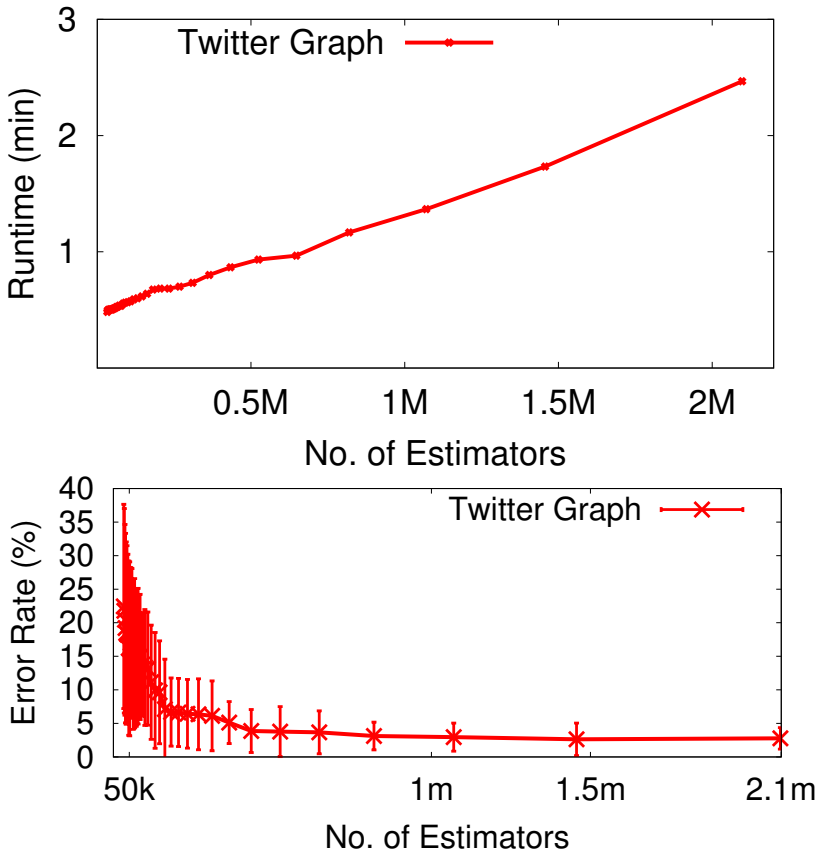


Figure 4.7: The actual relations between number of estimators and run-time or error rate.

4.5.1 Building Estimator vs. Time Profile

The time complexity of our approximation algorithm is linearly related to the number of edges in the graph and the number of estimators. Given a graph and a particular pattern, we find the computation time is dominated by the number of estimators when the number of estimators is large enough. From fig. 4.7, we see that the estimator-time curve is close to linear when the number of estimators is greater than 0.5M. Thus we propose using a linear model to relate the running time to the number of estimators.

When the number of estimators is small, the computation time is also affected by other factors and thus the curve is not strictly linear. However, for these regions, it is not computationally expensive to profile more exhaustively. Therefore, to build the time profile, we exponentially space our data collection, gathering more points when the number of estimators is small and fewer points as the number of estimators grows. We use a profiling budget T^* to bound the total time spent on profiling. Algorithm 1 shows

Algorithm 1 BuildTimeProfile(T^*)

```
1: P ← ∅ // store points for the profile
2: T ← 0, t ← 0, α ← α* // α* can be a reasonable random start
3: while T + t <= T* do
4:     t ← run approximation algorithm with α estimators
5:     P.add((α, t))
6:     α ← 2α
7:     T ← T + t
```

the pseudo code. ASAP starts from using a small number of estimators ($\alpha \leftarrow \alpha^*$), and doubles α each time until the total profiling time exceeds the profiling cost T^*. In practice, we have found that setting T^* in the minute granularity gives us good results.

4.5.2 Building Estimator vs. Error Profile

Since error profile is non-linear (fig. 4.7), techniques like extrapolating from a few data points is not directly applicable. Some recent work has leveraged sophisticated techniques, such as experiment design [173] or Bayesian optimization [19] for the purpose of building non-linear models in the context of instance selection in the cloud. However, these techniques also require the system to compute the error for a given setting for which we need to know the ground-truth, say, by running the exact algorithm on the graph. Not only is this infeasible in many cases, it also undermines the usefulness of an approximation system.

In ASAP, we design a new approach to determine the relationship between the number of estimators N_e and error ϵ. Our approach is based on two main insights: first, we observe that for every pattern based on the probability of sampling, a loose upper bound for the number of estimators required can be computed using Chernoff bounds. For instance for triangle counting, the sampling probability is $1/mc$ where m is the number of edges and c is the degree of first chosen edge(§4.2.3). This probability bound can be translated to an estimator of form $N_e > \frac{K*m*\Delta}{\epsilon^2 P}$ (Theorem 3.3 [135]) where K is a constant, m is the number of edges, Δ is the maximum degree and P is the ground truth or the exact number of triangles. At a high level, the bound is based on the fact that the maximum degree vertex leads to the worst case scenario where we have the minimum probability of sampling. Similar bounds exist for 4-cliques and other patterns [135]. These theoretical bounds provide a relation between the number of estimators (N_e), error bound (ϵ) and ground truth (P) in terms of the graph properties such as m and Δ.

The second insight we use is that for smaller graphs we can get a very close approximation to the ground truth by using a very large number of estimators. This is useful

in practice as this avoids having to run the exact algorithm to get a good estimate of the ground truth. Based on these two insights, the steps we follow are:

(a) We first uniformly sample the graph by edges to reduce it to a size where we can obtain a nearly 100% accurate result. In our experiments, we find that $5-10\%$ of the graph is appropriate according to the size of the graph.

(b) On the sampled graph, we run our algorithm with a large number of estimators (N_{gt}) to find $\hat{P_s}$, a value very close to the ground truth for the sampled graph.

(c) Using $\hat{P_s}$ as the ground truth value and the theoretical relationship described above, we compute the value of other variables on the sampled graph. For example, in the sampled graph, it is easy to compute m_s and Δ_s, and then infer K by running varying number of estimators.

(d) Finally we scale the values m_s, Δ_s and $\hat{P_s}$ to the larger graph to compute N_e. We note that the scaled $\hat{P}$ might not be close to P for the larger graph. But as we use the worst case bound to compute $\hat{P_s}$, the computed value of N_e offers a good bound in practice for the larger graph.

4.5.3 Handling Evolving Graphs

The ELP building process in ASAP is designed to be fast and scalable. Hence, it is possible to extend our pattern mining technique to evolving graphs [86] by simply rebuilding the ELP every time the graph is updated. However, in practice, we don't need to rebuild the ELP for every update. and that it is possible to reuse an ELP for a limited number of graph changes. Thus we use a simple heuristic where are a fixed number of changes, say 10% of edges, we rebuild the ELP. The general problem of accurately estimating when a profile is incorrect for approximate processing systems is hard [4] and in the future we plan to study if we can automatically determine when to rebuild the ELP by studying changes to the smaller sample graph we use in §4.5.2.

4.6 Evaluation

We evaluate ASAP using a number of real-world graphs and compare it to Arabesque, a state-of-the-art distributed graph mining system. Overall, our evaluations show that:

- Compared to Arabesque, we find ASAP can improve performance by up to $77\times$ with just 5% loss of accuracy for counting 3-motifs and 4-motifs.

Graph	Nodes	Edges	Degrees
CiteSeer [56]	3,312	4732	2.8
MiCo [56]	100,000	1,080,298	22
Youtube [109]	1,134,890	2,987,624	8
LiveJournal [109]	3,997,962	34,681,189	17
Twitter [104]	41.7 million	1.47 billion	36
Friendster [188]	65.5 million	1.80 billion	28
UK [34, 32]	105.9 million	3.73 billion	35

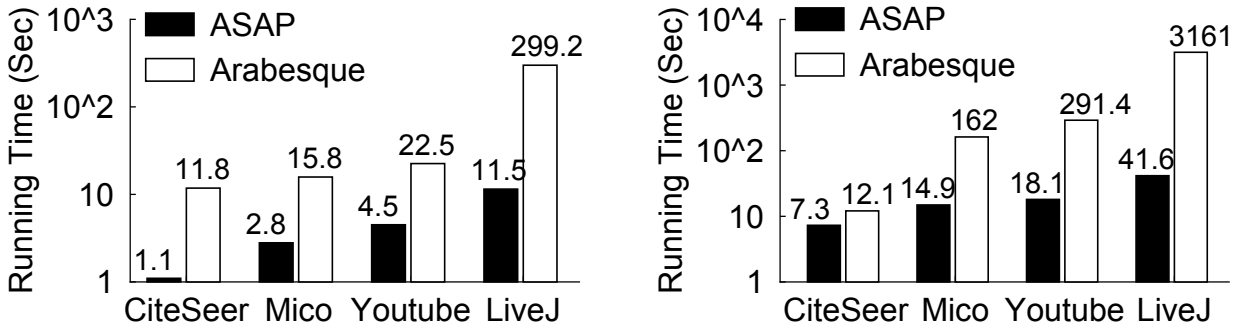
Table 4.2: Graph datasets used in evaluating ASAP.

- We find that ASAP can also scale to much larger graphs (up to 3.7B edges) whereas existing systems fail to complete execution.
- Our techniques to build error profile and time profile (ELP) are highly accurate across all the graphs while finishing within a few minutes.

Implementation. We built ASAP on Apache Spark [193], a general purpose dataflow engine. The implementation uses GraphX [71], the graph processing library of Spark to load and partition the graph. We do not use any other functionality from GraphX, and our techniques only use simple dataflow operators like `map` and `reduce`. As such, ASAP can be implemented on any dataflow engine.

Datasets and Comparisons. Table 4.2 lists the graphs we use in our experiments. We use 4 small and 3 large graphs and compare ASAP against Arabesque [166] (using its open-source release [72] built on Apache Giraph [22]) on four smaller graphs: CiteSeer [56], Mico [56], Youtube [109], and LiveJournal [109]. For all other evaluations, we use the large graphs. Our experiments were done on a cluster of 16 Amazon EC2 `r4.2xlarge` instances, each with 8 virtual CPUs and 61GiB of memory. While all of these graphs fit in the main memory of a single server, the intermediate state generated (§4.2) during pattern mining makes it challenging to execute them. Arabesque, despite being a highly optimized distributed solution, fails to scale to the larger graphs in our cluster. We note that Arabesque (or any exact mining system) needs to enumerate the edges significantly more number of times compared to ASAP which only needs to do it once or twice, depending on the query.

Patterns and Metrics. For evaluating ASAP, we use two types of patterns, *motif*s and *clique*s. For motifs, we consider 3-motifs (consisting of 2 individual patterns), and 4-motifs (consisting of 6 individual patterns) and for cliques, we consider 4-cliques. For our experiments, we run 10 trials for each point and report the median, and error bar in the ELP evaluation. We do not include the time to load the graph for any of the experiments



(a) 3-Motif Counting **(b)** 4-Motif Counting

Figure 4.8: ASAP is able to gain up to 77× improvement in performance against Arabesque. The gains increase with larger graphs and more complex patterns. Y-axis is in log-scale.

for ASAP and Arabesque. We use total runtime as the metric when raw performance is evaluated. When evaluating ASAP on its ability to provide errors within the requested bound, we need to know the *actual* error so that it can be compared with ASAP's output. We compute actual error as $\frac{|t-t_{real}|}{t_{real}}$, where t_{real} is the ground truth number of a specific pattern in a given graph. Since this requires us to know the ground-truth, we use simpler, known patterns, such as triangles and chains, where the ground-truth can be obtained from verified sources for such experiments. Note that the *actual error* is only used for evaluation purposes. Unless otherwise stated, the ASAP evaluations were done with an error target of 5% at 95% confidence.

4.6.1 Overall Performance

We first present the overall performance numbers. To do so, we perform comparisons with Arabesque and evaluate ASAP's scalability on larger graphs. We do not include ELP building time in these numbers since it is a one-time effort for each graph/task and we measure this in §4.6.3.

Comparison with Arabesque. In this experiment, we compare Arabesque and ASAP on the 4 smaller graphs (Table 4.2). In each of these systems, we load the graph first, and then warm up the JVM by running a few test patterns. Then we use each system to perform 3-motif and 4-motif mining, and measure the time taken to complete the task. In Arabesque, we do not consider the time to write the output. Similarly, for ASAP we do not output the patterns embeddings. The results are depicted in figs. 4.8a and 4.8b.

3-Motif	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 8	Twitter	42M	1.5B	2.5m
	16 x 8	Friendster	66M	1.8B	5.0m
	16 x 8	UK	106M	3.7B	5.9m
Arabesque	20 x 32	Inst[4]	180M	0.9B	10h45m
4-Motif	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 8	Twitter	42M	1.5B	22m
	16 x 8	UK	106M	3.7B	47m
	16 x 8	LiveJ	**4M**	**34M**	0.7m
Arabesque	16 x 8	LiveJ	**4M**	**34M**	53m
	20 x 32	SN[4]	**5M**	**199M**	6h18m

Table 4.3: Comparing the performance of ASAP and Arabesque on large graphs. The System column indicates the number of machines used and the number of cores per machine.

We see that ASAP significantly outperforms Arabesque on all the graphs on both the patterns, with performance improvements up to 77× with under 5% loss of accuracy. The performance improvements will increase if the user is able to afford a larger error (e.g., 10%). We also noticed that the performance gap between Arabesque and ASAP increases with larger graph and/or more complex patterns. In this experiment, mining the more complex pattern (4-motif) on the largest graph (LiveJournal) provides the highest gains for ASAP. This validates our choice of using approximation for large-scale pattern mining.

Scalability on Larger Graphs. We repeat the above experiment on the larger graphs. Since Arabesque fails to execute on these graphs on our cluster, we also provide performance numbers that were reported by its authors [166] as a rough comparison. The results are shown in Table 4.3.

When mining for 3-motif, ASAP performs vastly superior on the Twitter, the Friendster, and the UK graphs. Arabesque's authors report a run time of approximately 11 hours on a graph with a similar number of edges. This translates to a 258× improvement for ASAP. In the case of 4-motifs, ASAP is easily able to scale to the more complex pattern on larger graphs. In comparison, Arabesque is only able to handle a much smaller graph with less than 200 million edges. Even then, it takes over 6 hours to mine all the 4-motif patterns. These results indicate that ASAP is able to not only outperform state-of-the-art solutions significantly, but do so in a much smaller cluster. ASAP is able to effortlessly scale to large graphs.

[4]These graph datasets in Arabesque are not publicly available.

Pattern	Baseline	ASAP	Improv.
Motif Mining	32.2min	22min	32%
Predicate Matching	2.5min	27s	82%
Accuracy Refinement	2.5min	1.5min	40%

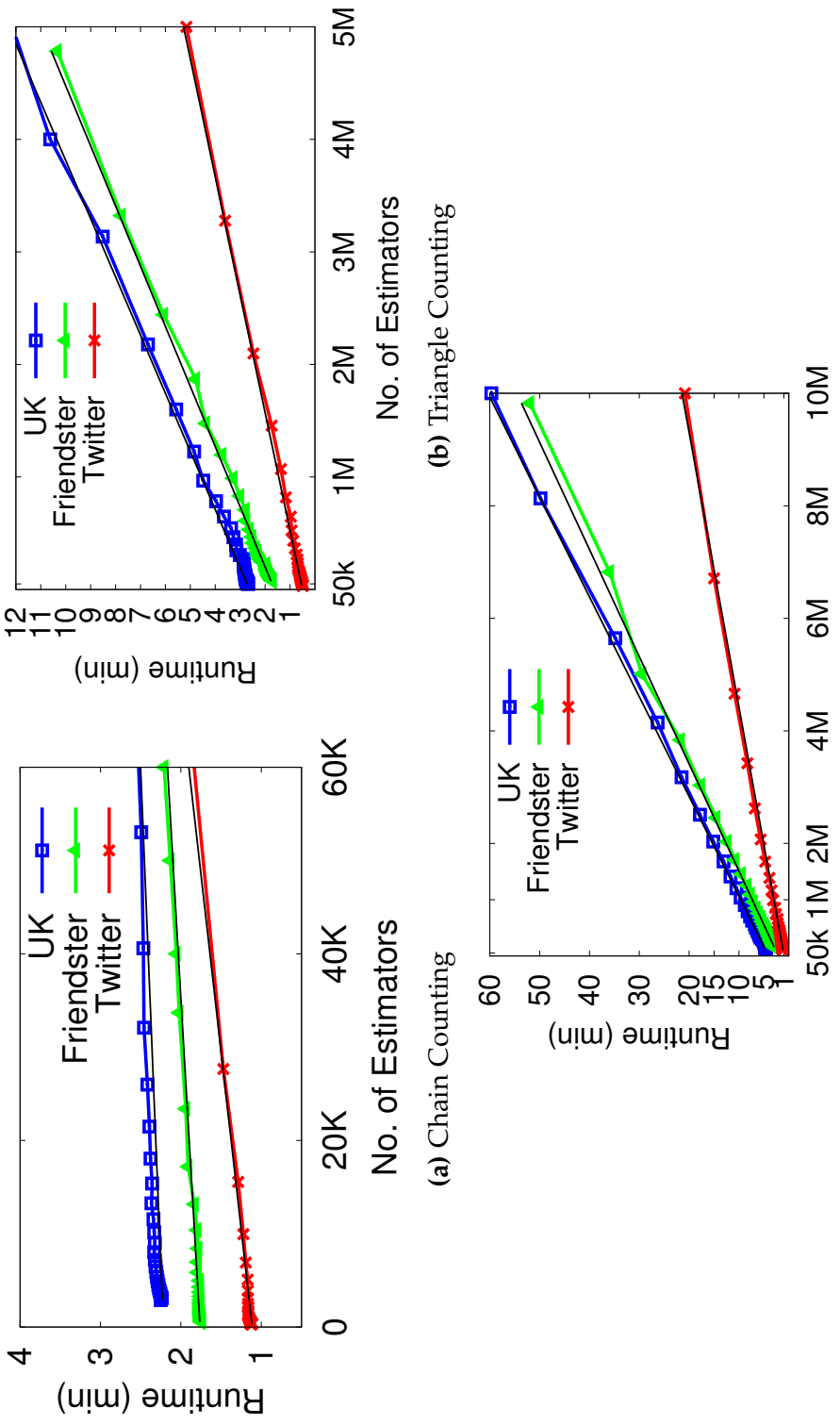
Table 4.4: Improvements from techniques in ASAP that handle advanced pattern mining queries.

4.6.2 Advanced Pattern Mining

We next evaluate the advanced pattern mining capabilities in ASAP described in §4.4.3.
Motif mining. We first evaluate the impact of ASAP's optimization when handling motif queries for multiple patterns. We use the Twitter graph and study a 4-motif query that looks for 6 different patterns. In this case ASAP caches the 3-node chain that is shared by multiple patterns. As shown in Table 4.4, we see a 32% performance improvement from this.
Predicate Matching. To study how well predicate matching queries work, we annotate every edge in the Twitter graph with a randomly chosen property. We then consider a 3-motif query which matches 10% of the edges. With ASAP's filtering based technique, the "all" query completes in 27 seconds, compared to 2.5 minutes when running without pre-filtering.
Accuracy Refinement. We study a scenario where the user first launches a 3-motif query on the Twitter graph with 10% error guarantee and then refines the results with another query that has a 5% error bound. We find that the running time goes from 2.5min to 1.5min (40% improvement) when our caching technique is enabled.

4.6.3 Effectiveness of ELP Techniques

Here, we evaluate the effectiveness of the ELP building techniques in ASAP, described in §4.5.
Time Profile. To evaluate how well our time profiling technique (§4.5.1) works, we run three patterns—3-chains, triangles, and 4-cliques—on the three large graphs. In each graph, we obtain the time vs. estimator curve by exhaustively running the mining task with varying number of estimators and noting the time taken to complete the task. We then use our time profiling technique which uses a small number of points instead of exhaustive profiling to obtain ASAP's estimate. We plot both the curves in fig. 4.9 for each of the three graphs. In these figures, the colored lines represent the actual (exhaustively profiled) curve, and the black line shows ASAP's estimate. From the figure we can see



(a) Chain Counting **(b)** Triangle Counting **(c)** Clique Counting

Figure 4.9: Runtime vs. number of estimators for Twitter, Friendster, and UK graphs. The black solid lines are ASAP's fitted lines.

Graph	Task	Time Profile	Error Profile
UK-2007-05	3-Chain	5.2m	2.1m
	3-Motif	6.1m	2.7m
	4-Clique	9.5m	4.8m
	4-Motif	11.2m	5.9m

Table 4.5: ELP building time for different tasks on UK graph

that the time profile estimated by ASAP very closely tracks the actual time taken, thereby showing the effectiveness of our technique.

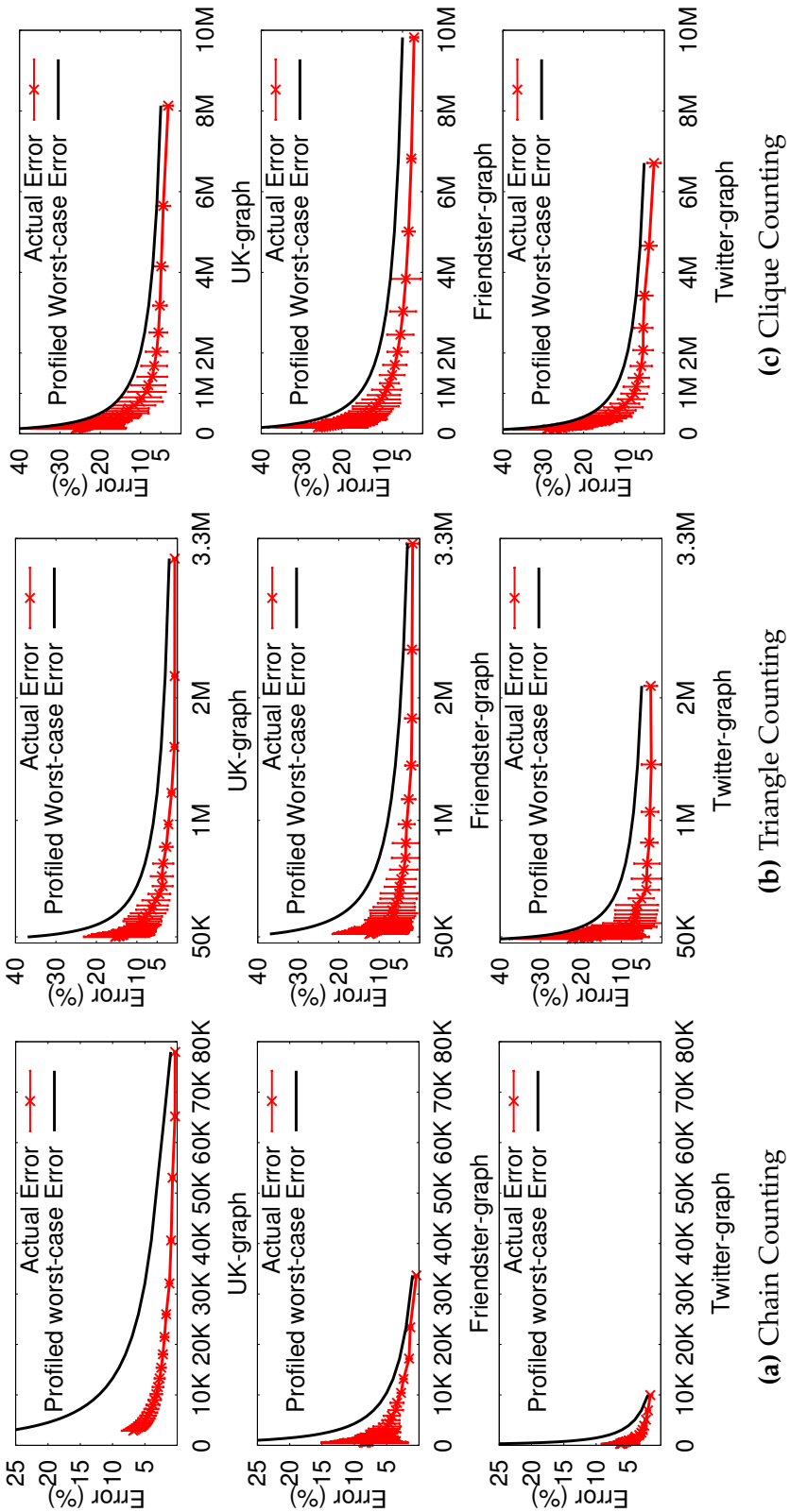
Error Profile. We repeat the experiment for evaluating ASAP's error profile building technique. Here, we exhaustively build the error profile by running a different number of estimators on each graph, and note the error. Then we use ASAP's technique of using a small portion of the graph to build the profile. We show both in fig. 4.10. We see that the actual errors are always within the estimated profile. This means that ASAP is able to guarantee that the answer it returns is within the requested error bound. We also note that in real-world graphs, the worst-case bounds are never really reached. In edge cases, where the number of patterns in the graphs are high like the chains in UK graph, the overestimation may be large, and one concern might be that we run more estimators than required. We are working on techniques that can help us determine a tighter bound for the number of estimators in the future but as discussed in §4.6.1, even with this overestimation we get significant speedups in practice. This experiment confirms that ASAP's heuristic of using a very small portion of the graph and leveraging the Chernoff bound analysis (§4.5.2) is a viable approach.

Error rate Confidence. In Figure 4.11, we evaluate the cumulative distribution function (CDF) of 100 independent runs on the UK graph with 3% error target and 99% confidence. We can see that 100/100 actual results are not worse than 3% error and 74/100 results are within 2% error. Thus the actual results are even better than the theoretical analysis for 99% confidence.

ELP Building Time. Finally, we evaluate the time taken for building the profiling curves. For this, we use the UK graph and configure ASAP to use 1% of the graph to build the error profile. The results are shown in table 4.5 for different patterns, which shows that the time to build the profiles is relatively small, even for the largest graph.

4.6.4 Scaling ASAP on a Cluster

ASAP partitions the graph into different subgraphs based on random vertex partition, and aggregates scaled results in the final reduce phase. In this section we evaluate how



(a) Chain Counting **(b)** Triangle Counting **(c)** Clique Counting

Figure 4.10: Error vs. number of estimators for Twitter, Friendster, and UK graphs.

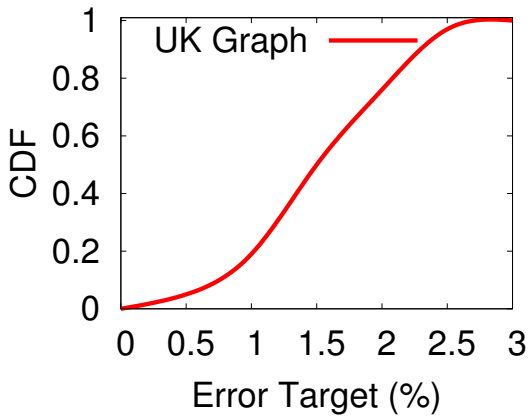


Figure 4.11: CDF of 100 runs with 3% error target.

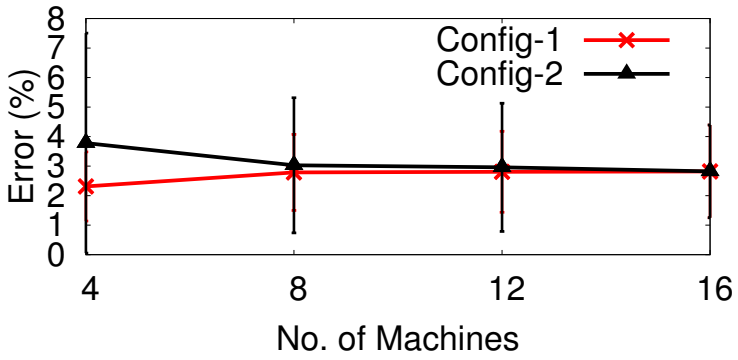


Figure 4.12: The errors from two cluster scenarios with different number of nodes. Config-1:*strong-scaling* to fix the total number of estimators as 2M × 128; Config-2: *weak-scaling* to fix the number of estimators per executor as 2M.

configurations with different numbers of machines impact the accuracy. In Fig. 4.12, we consider two scenarios: *strong-scaling*, where we fix the total number of estimators used for the entire graph, and increase the number of machines used; and *weak-scaling* where we fix the number of estimators used per-machine and thus correspondingly scale the number of estimators as we add more machines. We run the triangle counting task with the Twitter graph on different cluster sizes of 4, 8, 12, and 16 machines. From the figure we see that in the strong-scaling regime, adding more machines has no impact on the accuracy of ASAP and that we are able to correctly adjust the accuracy as more graph partitions are created. In the weak-scaling case we see that the accuracy improves as we increase more machines, which is the expected behavior when we have more estimators.

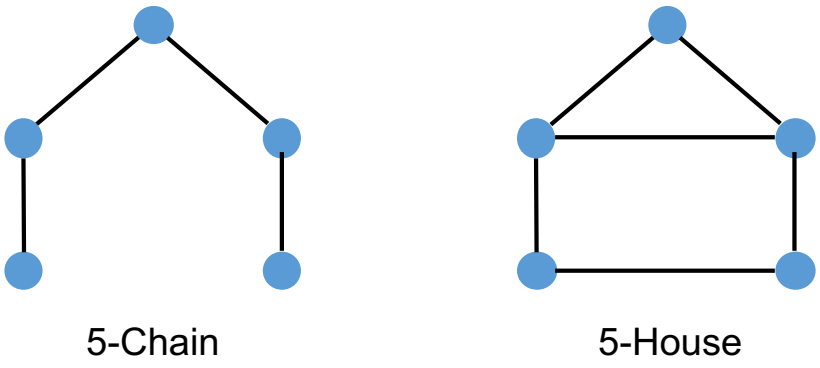


Figure 4.13: Two representative (from 21) patterns in 5-Motif.

5-Chain	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 16	Twitter	42M	1.5B	9.2m
	16 x 16	UK	106M	3.7B	17.3m
ASAP (10%)	16 x 16	Twitter	42M	1.5B	3.2m
	16 x 16	UK	106M	3.7B	6.5m
5-House	System	Graph	|V|	|E|	Runtime
ASAP (5%)	16 x 16	Twitter	42M	1.5B	12.3m
	16 x 16	UK	106M	3.7B	22.1m
ASAP (10%)	16 x 16	Twitter	42M	1.5B	5.6m
	16 x 16	UK	106M	3.7B	14.2m

Table 4.6: Approximating 5-Motif patterns in ASAP.

4.6.5 More Complex Patterns

Finally, we evaluate the generality of ASAP's techniques by applying to mine 5-motifs, consisting of 21 individual patterns. This choice was influenced by our conversations with industry partners, who use similar patterns in their production systems. Due to the complexity of the patterns, we used a larger cluster for this experiment, consisting of 16 machines, each with 16 cores and 128GB memory. Due to space constraints, and also because of the absence of a comparison, we only provide ASAP's performance on two representative patterns in table 4.6. As we see, ASAP is able to handle complex patterns on large graphs easily.

4.7 Related Work

A large number of systems have been proposed in the literature for **graph processing** [111, 70, 105, 151, 71, 143, 150, 178, 39, 44, 199]. Of these, some [111, 105, 151] are single machine systems, while the rest supports distributed processing. By using careful and optimized operations, these systems can process huge graphs, in the order of a trillion edges. However, these systems have focused their attention mainly on *graph analysis*, and do not support efficient graph pattern mining. Some systems implement very specific versions of simple pattern mining (e.g., triangle count). They do not support general pattern mining.

Similar to graph processing systems, a number of **graph mining** systems have also been proposed. Here too, the proposals contain a mix of centralized systems and distributed systems. These proposals can be classified into two categories. The first category focuses on mining patterns in an input consisting of multiple small graphs. This problem is significantly easier, since the system only finds one instance of the pattern in the graph, and is trivially incorporated in ASAP. Since this approach can be massively parallelized, several distributed systems exist that focus specifically on this problem. The state-of-the-art in distributed, general purpose pattern mining systems is Arabesque [166]. While it supports efficient pattern mining, the system still requires a significant amount of time to process even moderately sized graphs. A few distributed systems have focused on providing approximate pattern mining. However, these systems focus on a specific algorithm, and hence are not general-purpose.

In distributed data processing, **approximate analysis systems** [5, 67, 20] have recently gained popularity due to the time requirements in processing large datasets. Following the approximate query processing theory in the database community, these systems focus on reducing the amount of data used in the analysis process in the hope that the analysis time is also reduced. However, as we show in this work, applying the exact algorithm on a sampled graph does not yield desired results. In addition, doing so complicates, or even makes it infeasible to provide good time or error guarantees.

Theory community has invested a significant amount of time in analyzing and proposing **approximate graph algorithms** for several graph analysis tasks [51, 68, 9, 10, 36, 23]. None of these are aimed at distributed processing, nor do they propose ways to understand the performance profile of the algorithms when deployed in the real world. We leverage this rich theoretical foundation in our work by extending these algorithms to mine general patterns in a distributed setting. We further devise a strategy to build accurate profiles to make the approach practical.

4.8 Summary

We present ASAP, a distributed, sampling-based approximate computation engine for graph pattern mining. ASAP leverages graph approximation theory and extends it to general patterns in a distributed setting. It further employs a novel ELP building technique to allow users to trade-off accuracy for result latency. Our evaluation shows that not only does ASAP outperform state-of-the-art exact solutions by more than a magnitude, but it also scales to large graphs while being low on resource demands.

Chapter 5

Approximate Analytics on Dynamic Connected Data

5.1 Introduction

The last chapter showed how embracing approximation can significantly speed up pattern queries for use cases such as Taylor's and make pattern discovery feasible in medium to large datasets, which are much more realistic in enterprises [153]. Now we look at the feasibility of extending approximation for *analytic* queries.

Approximate analytics is an area that has garnered attention recently in big data analytics [5, 67, 20], where the goal is to let the end-user trade-off accuracy for much faster results. Several proposals for approximate analytics exist, but the underlying key idea is to use a small portion of the dataset to compute the results. Some approximation systems leverage the scheduler, and kill tasks selectively to achieve the desired accuracy or latency budget. However, all of the approximation systems focus on simple aggregate queries or analytics and thus do not consider complex, iterative workloads such as distributed graph processing.

Extending approximate analytics systems to support graph analytics is a challenging task because of the differences in the underlying assumptions. The fundamental assumption of a linear relationship between the sample size and execution time falls apart in graph processing. Further, approximation systems rely on statistical properties of the samples to compose partial results and/or error characteristics. Finally, these systems store multiple samples and cherry pick the right amount based on the linearity assumption. These techniques are difficult to incorporate in distributed graph processing due to the iterative nature of the algorithms.

System	*PageRank Runtime (s)*
PowerGraph [70]	300
GraphX [71]	419
Giraph [22]	596
GraphLab [111]	442

Table 5.1: Runtimes for pagerank algorithm as reported by popular graph processing systems used in production.

We explore the feasibility of bringing approximate analytics to distributed graph processing. Achieving efficient *approximate graph processing* faces a number of challenges, including the question of how to sample graphs and how to pick the right sampling parameter given a budget (§5.2). To solve these challenges, we leverage the recent advancements in spectral sparsification theory [163] literature. Specifically, we propose a spectral graph sparsification strategy that reduces the graph size significantly. We then devise a machine learning based approach to modeling performance and picking the right sparsification parameter (§5.3). We implement our proposed techniques in a system called GAP (for **G**raph **A**nalytics by **P**roximation). Our evaluation of GAP has shown encouraging results (§5.4).

5.2 Background & Challenges

We begin with a brief overview of graph-parallel systems, approximate analytics and then list the challenges in building a system for approximate graph analytics.

5.2.1 Graph Processing Systems

The bottleneck in distributed graph-parallel processing arises mainly from the message passing between vertices. In a big data system, these are implemented as shuffles which are quite expensive. As a result, executing graph algorithms take a non-negligible amount of time. Table 5.1 reproduces the reported results from recent graph processing literature for running 20 iterations of page rank algorithm on a moderately sized graph of 1B edges using 16 machines. We see that the execution time is in the order of several minutes. The performance numbers worsen significantly as the input graph becomes larger.

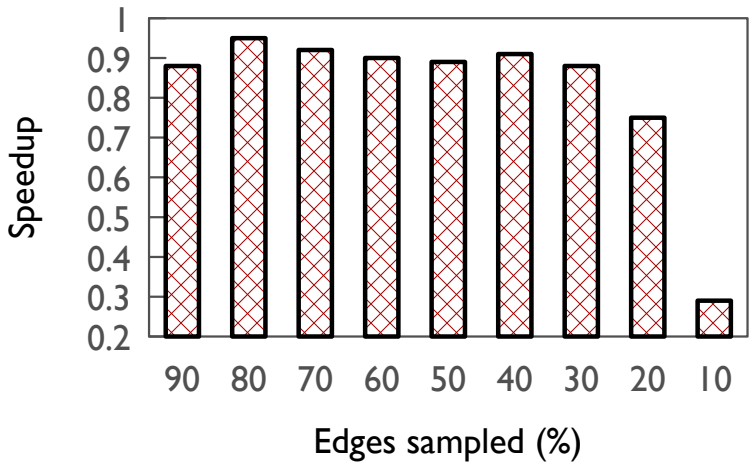


Figure 5.1: Sampling randomly leads to undesirable effects. Here, execution time (speedup) increases (reduces) with smaller samples.

5.2.2 Approximate Analytics

Approximate analytics is based on the premise that results from partial execution is often good enough. Systems supporting approximate analytics usually provide bounds on two dimensions—latency and accuracy—and lets users trade-off one for the other. These systems have been used successfully for query processing [5], dataflow jobs and straggler mitigation [20]. To provide this trade-off, they leverage sampling strategies. The basic observation is that the more data the system works on, the more accurate the results and vice-versa. Thus, given a corpus of data, approximation systems save samples of it using various criteria. Given a latency or accuracy budget, the job of the system is then to pick the right amount of samples to process and/or drop tasks when desired result is achieved.

5.2.3 Challenges

While it may seem straightforward to marry approximate analytics with graph-processing systems, making approximate graph analytics a reality is far from trivial. A system for approximate graph analytics faces a number of challenges. First, approximation systems rely on the fact that there exists a linear relationship between the amount of data in the sample and the execution time. However, such linear relationship does not exist in graph processing. While this could be helpful (i.e., a small reduction in input could lead to a large reduction in execution time), it also means that sampling could lead to undesirable outputs. To illustrate this, consider fig. 5.1, which shows the result of running connected

components on different random samples of a graph. Surprisingly, the execution time does not improve at all, rather it even becomes worse when the sample is small. This is because blind sampling destroys the structure of the graph leading to much longer paths. Thus, simple sampling strategies employed by existing approximate analytics systems are not applicable in our setting.

Second, due to this non-linearity, picking the right amount of samples is difficult. Traditional approximation systems create, store and precompute query results on samples based on the assumption that partial results and errors could be composed. However, this may not hold true in graphs. Thus, precomputing aggregates by creating and storing samples is not a feasible approach.

Finally, existing approximation systems support only simple queries, such as aggregates, where computing the error on the result is intuitive. However, graph algorithms are executed in an iterative manner and thus estimating error on the output of a graph algorithm operating on a sampled graph is hard. Theoretical bounds exist for a few specific algorithms, but to the best of our knowledge, there are no general guarantees.

5.3 Our Approach

We now describe our vision and approach for an approximate graph analytics system. In addition to solving the challenges listed earlier, we wish to achieve the following goals in our quest towards an efficient approximate graph analytics solution:

- A large body of graph theoretical work exists in the area of approximation algorithms. These works propose efficient approximate versions of various graph processing algorithms. We do not want to depend on such approximate version of any graph algorithms. In other words, we would like to be *approximation algorithm agnostic*.
- Similarly, several flavors of distributed graph-processing engines exist. Some of them offer asynchronous processing mode [70], while some of them offer the favorable properties of dataflow [71]. We would like to propose techniques that are generic and not specific to one graph-parallel model.
- Finally, existing graph processing systems support varied workloads. In this respect, we would like our solution to have low overhead when it needs to accommodate new workloads.

The overall architecture of our solution GAP is depicted in fig. 5.2. It consists of two main components. Leveraging the work in spectral graph theory, a graph sparsifier is used to reduce the input graph's size. Based on the observation that a graph workload's

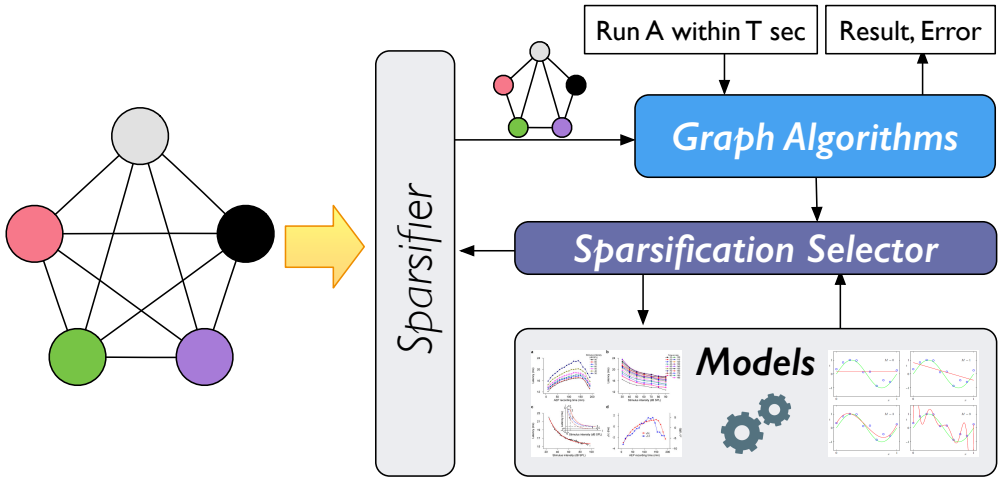


Figure 5.2: GAP System Architecture.

performance characteristics is majorly dependent on the input graph [29], a machine learning (ML) based model is used to learn and predict the amount of sparsification required for a given budget. When an input graph is provided, we map it to one of the benchmark models by a simple mapping technique. Our intuition is that since the number of graph algorithms are limited and graph characteristics are described by a few variables, ML models are apt at this job. We discuss these components in detail in the rest of this section.

5.3.1 Graph Sparsification

The fundamental building block of any approximation system is sampling. Carrying this over to graphs, a straightforward approach is to sample edges and vertexes using some criteria. This approach, commonly referred to as *graph sparsification*[1] has been studied extensively in the literature on graph theory. The main idea in this body of work is to compute a (much) smaller graph that preserves crucial properties of the input graph.

While several proposals on the type of sparsifier exists, many of them are either computationally intensive, or are not amenable to a distributed implementation (which is the focus of our work). We developed a simple sparsifier adapted from the work of Spielman and Teng [163] that is based on vertex degrees. The sparsifier uses the following probability to decide to keep an edge between vertex a and b:

$$\frac{d_{AVG} \times s}{\min(d_a^o, d_b^i)} \tag{5.1}$$

[1] Also referred to as graph sketching.

where d_{AVG} is the average degree of the graph, d^o_a is the out-degree of vertex a and d^i_b is the in-degree of vertex b and s is a tunable parameter that controls the level of sparsification.

Intuitively, we would like to drop one of *many* edges from a vertex with large degree as opposed to dropping the *only* edge from a vertex with low degree. The sparsifier in eq. (5.1) does exactly this. The cost of running the sparsifier is negligible. We further reduce this cost by computing vertex degrees when the graph is first loaded into the system. One potential problem with the sparsifier is that it takes decision solely on local information. To reduce the ill effects of this, we leverage how the algorithm operates. For instance, we can avoid removing an edge it is in the spanning tree and so on.

Estimating Error due to Sparsification

An important task when using sampling strategies is to estimate the error in the output. In a non-graph setting, error estimation is straightforward. However, it is unclear how to estimate the error due to sparsification on the output of graph algorithms. We take a simple approach to this problem: we define a few error metrics, and leave the flexibility of defining additional error metrics to the user. In our system, one default error metric is the *degree of reordering*. This metric is applicable to algorithms that output a ranking for the vertexes, for example page rank or triangle count. In these algorithms, we can define the degree of reordering as the amount of reordering of the ranking compared to the ground truth. This flexibility exists because we learn the relation between error and sparsification.

5.3.2 Picking Sparsification Parameter

Once the sparsification strategy is in place, the next question is how to pick the right sparsification parameter s for a given accuracy requirement. To the best of our knowledge, theoretical bounds on error for the graph sparsification in a general setting is an open problem, hence we develop heuristics to solve this problem. Specifically, we use simple machine learning techniques to learn a model for the relation between s and performance (latency/error).

Building a Model for s

At the simplest level, one can build a model for s by running every possible algorithm on a given graph at varying values of s and then feeding the observed results to a learning algorithm. However, this requires too much time and effort. Thus, an approximation system needs a smarter solution.

In GAP, we take a simple approach. We consider a set of standard graph algorithms. These algorithms are then run on a set of representative graphs at varying values of s. The objective of this task is to learn a function H that maps s and the characteristics of the graph and algorithm to the performance profile. That is, we would like to learn:

$$H : (s, a, g) \implies e/p$$

where a is the algorithm specific features (if any), g is the graph specific features and e/p is the error / performance. Since there is no standard benchmark for distributed graph processing, we choose the representative set of algorithms and workloads from the Graph500 benchmark [73]. Our observation (§5.2.3) indicates that performance profiles are non-linear, hence we pick learning techniques that can accommodate discontinuity (e.g., random forests).

Accommodating New Workloads

Once models are built, the final step is to use the model to pick the sparsification parameter s when the system needs to run a graph algorithm on an unknown/new graph workload. We do not want to build a model per workload online (the model building phase is intensive and hence is typically done offline). Thus, we need to find an existing model that can operate on the new workload.

For this, we propose a light-weight mechanism[2]. We randomly pick a few values of sparsification parameters and run the algorithm on the new workload in an online fashion. Simultaneously, we use the models to predict the output. We then pick the model(s) with the least error. The random values of s could be chosen to complete the tests within a given time budget. For every new workload, we also use the results of running analytics as a feedback to our learning component. This lets us refine and improve our models over time.

5.4 Evaluation

We picked five openly available graph datasets [164, 33, 31] (with number of edges up to 3.7 billion for the largest graph) based on the characteristics of the underlying graph, such as the diameter and clustering coefficient.

We evaluate our hypothesis of building a model for sampling based on the characteristics of the graph in the following way. We ran two algorithms, page rank and triangle count, on the datasets with varying values of the sparsification parameter s. We then

[2]We are pursuing better techniques here at the time of writing.

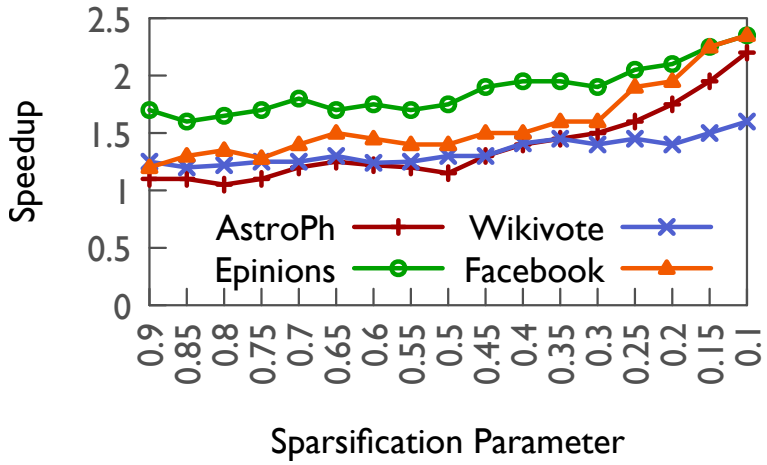


Figure 5.3: In triangle counting, we see similar trends in performance in graphs with similar characteristics (e.g., AstroPh & Facebook).

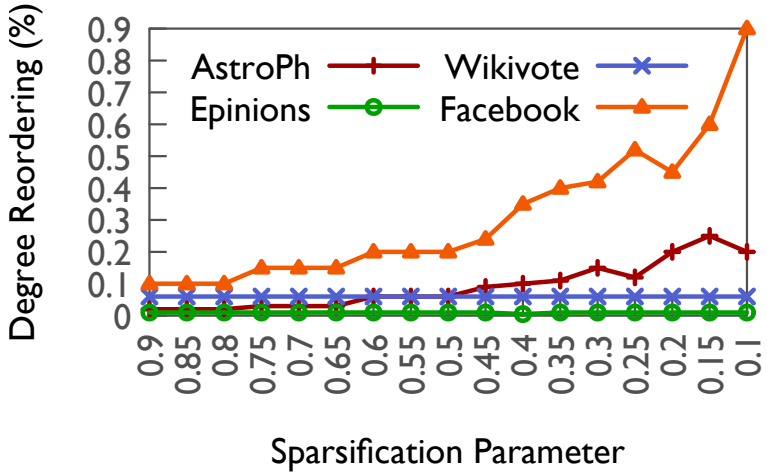


Figure 5.4: Error due to sparsification. Like the speedup, we see similarity in the error profile of graphs with similar characteristics.

recorded the speedup of the algorithm compared to the execution on the complete graph. We also note the error by evaluating the degree of reordering (§5.3) for each value of s.

Figure 5.3 shows the speedup obtained on triangle count algorithm, while fig. 5.4 depicts the error in terms of reordering. We see that even with a small reduction in input, the system is able to speedup the execution. As seen in the error characteristics, this speedup does not come at the expense of large errors. The error remains small for a wide range of the sparsification parameter. It may be troubling to see the diminishing returns with increase in sparsification, but this is due to the use of small datasets and also due to

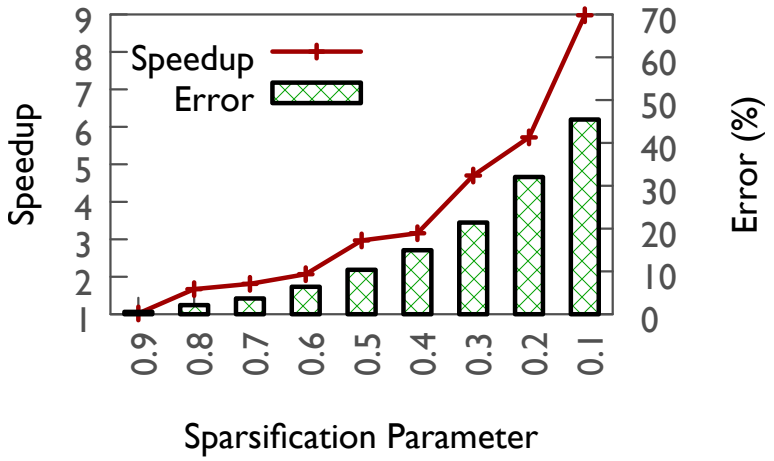


Figure 5.5: Larger graph (uk-2007-05 [33, 31] with 3.7B edges) sees better speedup due to the distributed nature of the execution.

the fact that the experiment was done on a single machine (the system executes the same way as in a distributed setting but does not incur network penalties). As the graph grows larger and the computation is spread across many machines, sparsification reduces the shuffled data, and we see much larger gains as shown in fig. 5.5.

A more intriguing question is if our proposed approach is feasible. That is, is it possible to learn a model at all for approximate analytics? In our dataset, the Facebook and AstroPh datasets share similarity in the diameter and clustering coefficients. Similarly, Wikivote and Epinions share similar graph characteristics. Moreover, Facebook and AstroPh are social relationships while Wikivote and Epinions represent voting/rating relationships. In our results, we see that the performance and error trends follow the same observation—-the performance and error curves of the Facebook and AstroPh datasets exhibit similar trends, the same is true for the Wikivote and Epinions datasets. Results from our experiments using page rank algorithm also show similar trends. Thus, we believe that our approach is feasible.

5.5 Related Work

A large number of graph processing systems exist in the literature. [111, 70, 105, 151, 71, 143, 150, 178, 39] focus on iterative analytics on static graphs. [114, 129, 45, 41, 76, 125, 121, 91] focus on analytics on evolving graphs. None of these systems support approximate analytics. GraphTau [129], which focuses on evolving graph processing, supports approximate page rank computations. However, it does not allow user to specify

a budget. Our techniques can be used to bring approximation to several of these graph processing systems.

Approximate analytics systems have gained much popularity in the big data analytics community recently, and thus several proposals exist. BlinkDB [5] uses stratified sampling to generate samples and then chooses samples to satisfy the query budget. [20] uses approximation techniques to mitigate stragglers. ApproxHadoop [67] enables approximation enabled map-reduce jobs. These systems do not support graph processing.

5.6 Summary

For many graph-processing application scenarios, computing an approximate answer is good enough. Yet, existing graph processing frameworks, in an effort to compute the exact answer, take several minutes or even hours to execute popular graph algorithms. In this chapter, we looked at the problem of approximate graph analytics. We presented our proposal, which uses a spectral sparsifier to reduce the size of the graph, and a machine learning model to pick the right amount of sparsification given a budget.

Chapter 6

Geo-Distributed Analytics on Connected Data

6.1 Introduction

In this chapter, we look at a possible enhancement to connected data processing, *geo-distributed processing*.

All the systems we have described till now assume that the data to be processed is available at a single data center which processes this data. However, today, many applications that could benefit from graph analysis are deployed on data centers *across the globe* and generate data in a geo-distributed fashion. For example, users of social networks are located around the globe. Similarly, cellular networks collect data at base stations that are geo-distributed across various locations [91]. This is also true for emerging applications. Internet-of-Things (IoT) applications, such as the much anticipated driverless vehicles, may generate data across multiple aggregation points. In analyzing such datasets, it may not always be feasible to aggregate the data to a central location due to many reasons. First, Wide Area Network (WAN) bandwidth is expensive and transferring large amounts of data may incur high costs. Second, more importantly, many of these scenarios could benefit from timely, low-latency analytics. Finally, political reasons may prevent data from moving to a different location.

The problem of analyzing datasets spanning geographical boundaries is not new; the field of Geo-distributed Analytics (GDA), that has gained much attention recently, focuses on precisely the same problem [176, 174, 140]. GDA brings WAN awareness to big data analytics, thus eliminating the need to move all the data to one location. Existing GDA proposals look at different aspects of this problem, ranging from low-level task placement [140] to higher-level query optimization [174]. However, current works

on GDA have focused on simple queries and aggregates, and largely ignored iterative workloads such as machine learning and distributed graph processing, two important and emerging workloads in many applications.

Performing graph analytics in a geo-distributed fashion differs from traditional GDA in many ways. Due to the iterative nature of graph algorithms and the complex dependency in tasks that perform these iterations, simple task placement techniques do not work well as there is a need to deal with task affinity. Further, in traditional GDA, many datasets are amenable to clean sharding. This is not the case in graph processing where locality plays an important role in the performance of graph algorithms. Because of the expensive joins that must be performed at every iteration in a graph-parallel setting, simple join optimizations in GDA may not be effective. Finally, many graph algorithms generate large amounts of intermediate data. Thus, geo-distributed graph analytics solutions need to account for the iterative nature of graph processing.

We focus on the problem of *geo-distributed graph analytics*. While combining traditional GDA with graph analytics may seem straightforward, our experience indicates that it is far from trivial. It is tempting to see this as a graph partitioning problem, since the goal of graph partitioning is to improve locality and thus reduce communication. However, due to the nature of data creation, repartitioning may not be feasible. Other similar challenges exist which should be addressed (§6.2). We observe that the key in geo-distributed graph analytics is to optimize the iterative processing style of graph-parallel systems. Based on this, we propose three techniques. First, we reduce the data to be processed using sampling strategies that leverages graph algorithms. Second, we remove the inefficiencies in current graph processing models by proposing a modification that reduces data exchange using a simple incremental computation strategy. Finally, we discuss how to bring WAN awareness into the picture (§6.3). To evaluate our proposal, we built MONARCH, a system that incorporates our proposed techniques. To the best of our knowledge, MONARCH is the first system to focus on geo-distributed graph analytics.

6.2 Background & Challenges

We begin with a brief overview of geo-distributed analytics and then list the challenges in geo-distributed graph analytics.

6.2.1 Geo-Distributed Analytics

A number of recent works have made the case for Geo-Distributed Analytics (GDA) [176, 140, 174]. While traditional data analytics assumes that data resides in a single, centralized

datacenter, GDA forgoes that assumption. In GDA, data is collected and stored at geographically distributed datacenters. Analytic tasks are run across these datacenters without aggregating data to a central location. The key challenge in GDA systems is to ensure low response times for the analytic tasks being performed.

GDA systems solve this challenge by being Wide Area Network (WAN) aware. Specifically, these systems consider intermediate data movement to be the bottleneck and thus optimize the placement of such data and tasks that operate on them based on the bandwidth available between datacenters. Some systems [82] go further by switching between different join strategies and task coordination.

6.2.2 Challenges

There are several challenges in building a geo-distributed graph processing system.

First, GDA systems assume simple jobs and queries. In contrast, graph processing systems execute graph algorithms in an iterative manner, with multiple message exchanges in every iteration. Extending this to a geo-distributed setting means that every iteration would generate data exchange across WAN. While traditional GDA's task placement and scheduling can optimize where the tasks are placed, they do not alleviate the problem with the iterative model of graph processing.

Second, in GDA systems, data is susceptible to sharding. Hence, there is fine grained control over data movement that could be beneficial—for instance, a small amount of data could be moved to a different data center for a significant improvement in task placement flexibility. While graph partitioning has similar goals of improving locality and reducing communication between partitions, cleanly partitioning graphs is a hard problem [70]. Additionally, as graph algorithms progress the partitioning may need to be changed for the best performance. On top of this, since partitioning graphs cannot be done at fine granularity, a complete repartitioning may need to be done due to the nature of data generation. Thus, a one-time partitioning or data placement strategy is unlikely to be of help.

Finally, graph algorithms are complex, and their distributed implementations are demanding since they involve expensive operations [71]. The immutability assumption made by many graph-processing frameworks make things worse in terms of bandwidth usage. For task scheduling purposes, some GDA systems assume that intermediate data could be estimated and placed efficiently. This assumption breaks down in graph processing, where the intermediate data size could be large. As an example, running connected components on the openly available Twitter data [164] results in shuffling more than 50GB of data during the initial iterations in GraphX [71], a popular graph processing framework.

6.3 Our Proposal

We now describe our proposal for geo-distributed graph analytics after discussing our assumptions.

6.3.1 Assumptions

Geo-Distributed Graph: We assume that the graph is *generated* in a geo-distributed fashion. That is, a graph $G(V, E)$ exists across P partitions, distributed across D data centers ($P >= D$). Each partition $p \in P$ consists of v vertices and e edges. In this setting, it is obvious that aggregating the graph to one data center is expensive.

PowerGraph [70] argued that many naturally occurring graphs follow power-law distribution and hence make a case for vertex-cuts rather than edge-cuts for entities spanning partitions. Following this, we choose vertex-cuts, and mirror vertices which have edges spanning partitions. We note that this is not fundamental to our approach, as our approach could use edge-cuts also.

As discussed earlier, we do not assume that a complete repartitioning (e.g., using a communication efficient partitiong scheme such as 2D partitioning [70]) could be done. Thus, we restrict ourselves to the partitioning provided by the data naturally. Leveraging partitioning flexibility is something that we wish to pursue in the future.

Algorithms: While a large body of algorithms exist for the analysis of graphs, we restrict our scope in this work to algorithms that are implementable in a GAS decomposition model. Most of the commonly used graph algorithms can be expressed in GAS decomposition format; for instance, GraphX [71] provides implementations for six such graph algorithms (connected components, label propagation, page rank, SVD, shortest path, and triangle count).

WAN vs LAN Bandwidth: We assume that the LAN bandwidth is significantly higher than the WAN bandwidth. We further assume that the WAN bandwidth between pairs of DCs can differ significantly. This is true in most cloud provider settings. For instance, [174] notes that inter-DC bandwidth in major cloud providers is 1-2 orders of magnitude less than intra-DC bandwidth, and that the pair-wise WAN bandwidth can vary by over $20\times$.

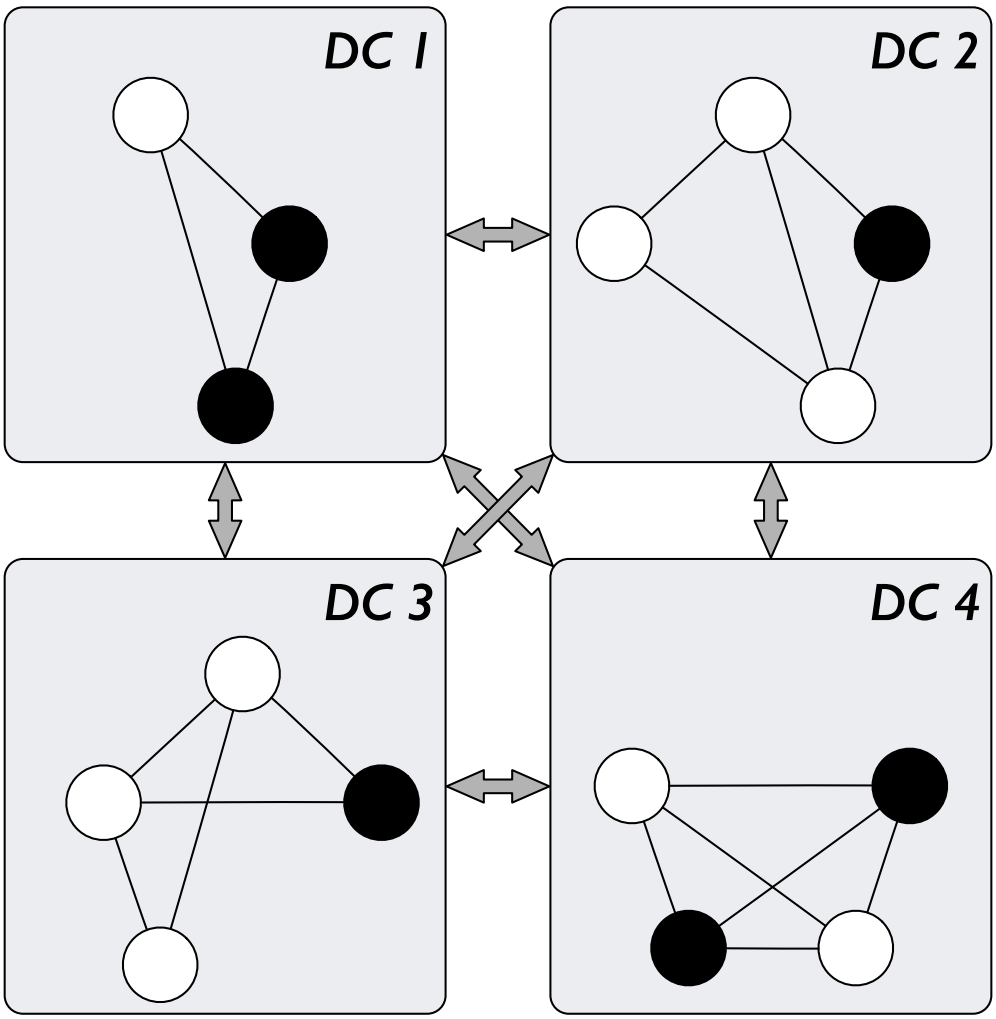


Figure 6.1: MONARCH system architecture. Each data center (DC) contains part of the graph. Communication between DCs happen through border vertices (shaded vertex in the picture), who exchange and synchronize state as described in §6.3.

6.3.2 Approach

The overall architecture of the system we are currently building, which we call MONARCH, is depicted in Figure 6.1. Based on our observation that optimizing iterative processing style of graph-parallel algorithm is the key to efficient geo-distributed graph analytics, the main idea in our approach is to *leverage the characteristics of graph-parallel computation model and the algorithms they support* to reduce WAN usage. To achieve this, we propose three simple, but powerful techniques: first, we reduce the data itself using an accuracy preserving sampling process. This results in less data to exchange. Second, we propose a

modification to the GAS model that removes the inefficiencies with iterative processing. Finally, we bring WAN awareness to this modified model. We explain them in the rest of this section.

Sparsification without Accuracy Loss

In geo-distributed graph analytics, inter-DC data exchange happens only if there are entities spanning multiple data centers. Hence, our first goal is to reduce these spanning entities and/or reduce the data flowing through them.

In MONARCH, we call vertices that interface with other datacenters *border* vertices. Since we use vertex-cut by default, border vertices are mirrors. In the GAS model, vertices gather (scatter) messages from (to) their neighborhood (§6.2). Thus, by reducing the size of the graph, we can reduce the amount of data transferred across data centers. Unfortunately, randomly eliminating graph entities to reduce the graph size leads to incorrect results.

To solve this, we plan on using a simple technique. We observe that in iterative graph-parallel model, many graph algorithms generate and exchange redundant intermediate data. This is because the GAS model only considers the immediate neighborhood in each iteration. Leveraging this, we can design a sparsification strategy that eliminates graph entities that will not contribute to the final solution. To illustrate a simple case, consider the connected components algorithm. By examining the connectivity information of border vertices, we can eliminate the need to mirror all but one vertex across DC pairs if the vertices are connected. This reduces the amount of data transferred across DCs. Further improvements can be obtained if only partial results are required (e.g., only required to compute components that contain particular vertices, or only find top components), as such cases can discard large parts of the graph and even eliminate the need for border vertices.

While the sparsification technique is beneficial in our setting, we note two shortcomings with it. First, not all algorithms can leverage such sparsification strtegies. Second, our sparsification strategy may result in a slightly longer convergence time. This is because by dropping entities, we may eliminate a shorter path. However, we assume that the cost of WAN transfer is much higher than this small sacrifice in convergence time.

Geo-distributed Graph Computation Model

Once the graph is sparsified, the next step is to run graph algorithms on it in a geo-distributed manner. As discussed in §6.2, GAS computations result in two data exchanges (`gather` and `scatter`) in every iteration. In our setting, this means that the border vertices potentially need to exchange data twice per iteration. When the data to be exchanged

is large and/or when multiple iterations are involved, these transfers can become the bottleneck.

To solve this problem, we propose a simple modification to the GAS model. In this model, we restrict the execution of the GAS model to each datacenter. We then use a merging strategy to combine the results from each datacenter. Our enhanced GAS model consists of the following stages:

Bootstrap: When a graph algorithm is to be executed, MONARCH first invokes the bootstrap stage. In the bootstrap phase, MONARCH runs vanilla GAS on every datacenter independently. We consider the subgraph in each data center to be a graph of its own, and compute the algorithm result on this subgraph. At the end of this stage, we end up with local solutions in each datacenter.

Global Sync: After the bootstrap GAS execution has converged, we invoke a global synchronization stage. In this stage, only border vertices participate. A `gather`-like operation is invoked on them which enables the vertices to collect the partial state from other mirrors. Then, an algorithm specific function f_a is used to combine these partial states to generate an updated state for the border vertices. After this stage, all the mirrors of each border vertices have the same state. However, the global graph is in an inconsistent state. This is because the partial results in each DC may no longer be valid because of the updates to the border vertices.

iGAS: A strawman approach to recompute the correct partial results is to reset the local graph's vertices (except the border vertices) and restart the local GAS computation. However, this is wasteful. Hence we propose a different approach. We observe that after the synchronization, each subgraph is equivalent to an updated graph (with just the border vertices updated). Thus, we can leverage an incremental computation model to update the results on the local graph. In MONARCH, we design an incremental version of the GAS model, which we call iGAS.

The iGAS computation leverages both the GAS computation model and algorithm properties. Specifically, we exploit the fact that GAS computations consider only the immediate neighborhood. In each iteration of the iGAS, we mark the immediate neighborhood of vertices that changed their state, and force computations on them. Obviously, in the first iteration, we mark the neighbors of the border vertices. We then repeat this marking and re-computation step until the change in the border vertex is propagated across the entire local graph. One problem with this approach is that potentially all the vertices may recompute if the graph is fully connected. To avoid this, we leverage

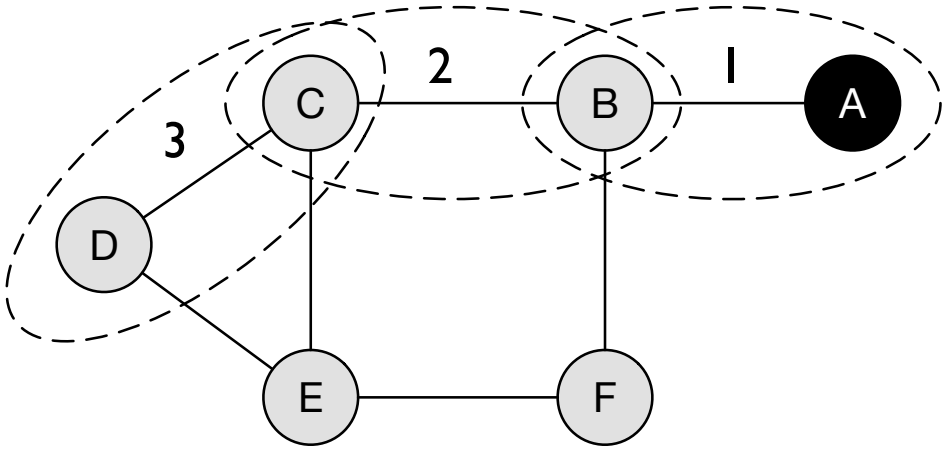


Figure 6.2: In incremental GAS model, each iteration marks and activates the neighborhood it influences. In this example, border vertex A is updated. It marks and activates B in the first iteration, B marks and activates C in the second iteration and C marks and activates D in the third. By leveraging the characteristics of the algorithm being executed, we avoid marking E and F although they are in the immediate neighborhood of C and B.

algorithm properties. We mark only neighbors which might use the updated value. Figure 6.2 shows our iGAS approach. At a high level, iGAS can be seen as a backtracking and rectification process.

To summarize, our enhanced GAS model starts with a normal execution of the GAS model, then switches to iterations of global sync followed by iGAS until convergence. Essentially, we are amortizing the cost of synchronization after every iteration of the GAS step by batching multiple GAS iterations. We note that there is one caveat to our approach. The global synchronization step assumes that the algorithm specific function, f_a, is able to correctly combine the partial results for each border vertices. While we have derived f_a for many common graph algorithms, generalizing our technique to any graph algorithm is part of our future work.

Bringing WAN Awareness

While MONARCH specifically optimizes for WAN bandwidth, our techniques consider WAN bandwidths to be equal. However, this is not true in practice. To tackle this problem, we envision two approaches.

First, we plan to generalize our computation model to support arbitrary interleaving of the global sync and iGAS phases per datacenter. Thus, depending on the current WAN bandwidth, a datacenter may decide whether to participate in the global sync or not. In

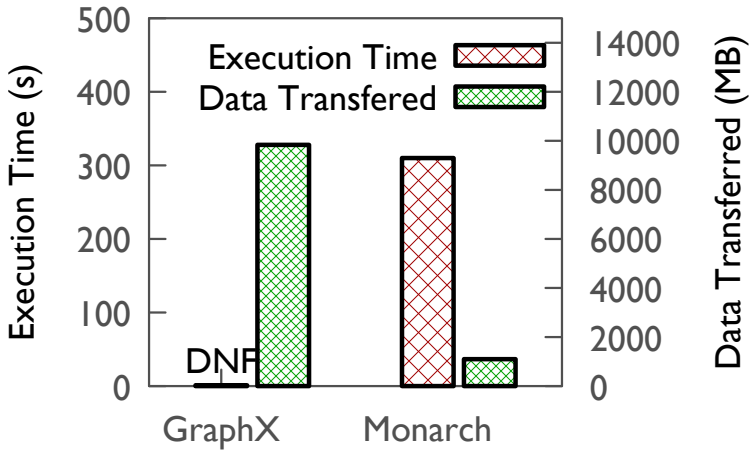


Figure 6.3: Our proposal is able to complete the execution of connected components when GraphX is unable to complete. This is because it tries to transfer too much data across WAN.

the second approach, if the system is given some flexibility in terms of data movement, then we plan to explore moving some border vertices based on the amount of bandwidth they might consume. Both are hard problems, and we are actively exploring different ways to solve them.

6.4 Evaluation

We evaluate the usefulness of the enhanced GAS model that includes the local incremental computations and global syncs. We chose the open source Twitter dataset [104], and use 16 machines on Amazon EC2 that simulates the same setting as in [71]. We picked 4 machines in each region, and then ran the connected components algorithm. The results are depicted in fig. 6.3. We see that GraphX is unable to complete the computation since it fails during the shuffle stage, while the enhanced model we propose is able to complete the computation in around 310 seconds. While this is longer than the reported numbers in GraphX (251s), we believe a lot of optimization opportunities are left for us to explore. We also see that while GraphX tries to transfer almost 10GB of data across WAN, MONARCH only transfers around 1 GB. In this experiment, we did not use the sparsification process.

6.5 Related Work

A large number of graph processing systems exist in the literature, of which [111, 70, 44, 177, 185, 105, 151, 71, 143, 150, 178, 39, 166, 197, 204, 112, 11, 201, 181, 138, 183, 184, 74, 113, 198, 12, 66, 196, 200] focus on iterative analytics on static graphs, while [98, 99, 120, 114, 45, 41, 76, 125, 121, 129, 91] focus on analytics on evolving graphs. However, none of them support geo-distributed processing and thus focus on a single datacenter where the graph is aggregated. While our work focuses on the GAS decomposition model, these techniques can be incorporated into other models. [62] parallelizes sequential graph algorithms using partial evaluations and *algorithm specific* incremental computations, but does not consider geo-distributed settings.

On the other hand, many recent works have proposed techniques for geo-distributed data analytics. Iridium [140] uses WAN aware task placement and scheduling. [176] looks at join algorithm selection strategies for WAN optimization. SWAG [82] coordinates tasks across DCs. Finally, Clarinet [174] argues for WAN aware query optimization. [80] looks at the problem of geo-distributed machine learning using approximation techniques that are specific to ML algorithms. None of these systems consider iterative graph processing.

6.6 Summary

Graph processing and geo-distributed analytics are two areas that have seen increasing interest in the recent past. Yet, neither of them support the other. Geo-distributed graph analytics could be beneficial for many application scenarios that generate graph-structured data. In this chapter, we took the first step towards marrying geo-distributed analytics with graph-parallel processing. We listed the challenges in doing so, and proposed a solution to address these challenges.

Chapter 7

Real-time Decisions on Dynamic Connected Data

7.1 Introduction

Increasingly, the trend in connected data analyses is moving towards tasks that operate on data that is procured in a real-time fashion to produce low-latency decisions. Unlike traditional tasks such as aggregates or datacubes, these real-time analytic tasks often involve model building and refinement for the purpose of manual or automatic decision making. For instance, Alex's network can benefit from real-time diagnosis of problems use the results to self-heal and reduce impact to users in the network. Similarly, Taylor's tasks can leverage real-time updates to fraud detection models. However, such analyses are faced with a fundamental trade-off between *having not enough data to build accurate-enough models in short timespans* and *waiting to collect enough data that entails stale results* in several domains. In this chapter, we seek to answer the question of whether it is possible to mitigate this trade-off. Towards this goal, we take the first step and investigate this trade-off in detail, expose the effects of it, and build techniques to mitigate it in Alex's domain-specific problem: *performance diagnostics in cellular Radio Access Networks (RAN)s*.

While RAN technologies have seen tremendous improvements over the past decade [160, 152, 156], performance problems are still prevalent [158]. Factors impacting RAN performance include user mobility, skewed traffic pattern, interference, lack of coverage, unoptimized configuration parameters, inefficient algorithms, equipment failures, software bugs and protocol errors [157]. Though some of these factors are present in traditional networks and troubleshooting these networks has received considerable attention in the literature [24, 202, 47, 6, 179], RAN performance diagnosis brings out a

unique challenge: the performance of multiple base stations exhibit complex temporal and spatial interdependencies due to the shared radio access media and user mobility.

Existing systems [58, 16] for detecting performance problems rely on monitoring aggregate metrics, such as connection drop rate and throughput per cell, over minutes-long time windows. Degradation of these metrics trigger mostly manual—hence, time-consuming and error-prone—root cause analysis. Furthermore, due to their dependence on aggregate information, these tools either overlook many performance problems such as temporal spikes leading to cascading failures or are unable to isolate root causes. The challenges associated with leveraging just aggregate metrics has led operators to collect detailed traces from their network [59] to aid domain experts in diagnosing problems.

However, the sheer volume of the data and its high dimensionality make the troubleshooting using human experts and traditional rule-based systems very hard, if not infeasible [97]. Machine learning (ML) is one natural alternative to these approaches that has been used recently to troubleshoot other complex systems with considerable success. However, simply applying ML to RAN diagnosis is not enough. The desire to troubleshoot RANs as fast as possible exposes the inherent tradeoff between *latency* and *accuracy* that is shared by many ML algorithms.

To illustrate this tradeoff, consider the natural solution of building a model on a per-base station basis. On one hand, if we want to troubleshoot quickly, the amount of data collected for a given base station may not be enough to learn an accurate model. On the other hand, if we wait long enough to learn a more accurate model, this will come at the cost of delaying troubleshooting and the learned model may not be valid any longer. Another alternative would be to learn one model over the entire data set. Unfortunately, since base stations can have very different characteristics using a single model for all of them can also result in low accuracy (§7.2).

We present CellScope, a system that enables fast and accurate RAN performance diagnosis by resolving the *latency* and *accuracy* trade-off using two broad techniques: intelligent data grouping and task formulations that leverage domain characteristics. More specifically, CellScope applies Multi-task Learning (MTL) [168, 42], a state-of-the-art machine learning approach, to RAN troubleshooting. In a nutshell, MTL learns multiple related models in parallel by leveraging the commonality between those models. To enable the application of MTL, CellScope uses two techniques. First, it uses *feature engineering* to identify the relevant features to use for learning. Second, it uses a PCA based similarity metric to group base stations that share common features, such as interference and load. This is necessary since MTL assumes that the models have some commonality which is not necessarily the case in our setting, e.g., different base stations might exhibit different features. Note that while PCA has been traditionally used to find network anomalies, CellScope uses it for finding the common features instead.

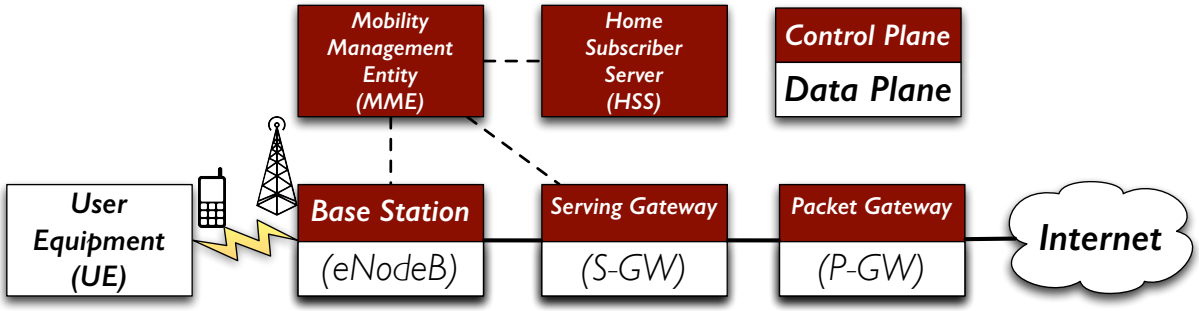


Figure 7.1: LTE network architecture

To this end, CellScope uses MTL to create a hybrid model: an offline base model that captures common features, and an online per-base station model that captures the individual features of the base stations. This hybrid approach allows us to incrementally update the online model based on the base model. The resulting models are both accurate and fast to update. Finally, in this approach, finding anomalies is equivalent to detecting concept drift [65]. To demonstrate the effectiveness of our proposal, we have built CellScope on Spark [192, 162, 102]. Our evaluation shows that CellScope is able to achieve accuracy improvements up to 4.4× without incurring the latency overhead associated with normal approaches (§7.6). We have also used CellScope to analyze a live LTE network consisting of over 2 million subscribers, where we show that it could save the operator several orders of magnitude savings in troubleshooting efforts (§7.7).

We then investigate if the techniques we present in this chapter can be general. To do so, we take a new domain-specific problem, *energy bug diagnosis in mobile phones*, and illustrate that the trade-off exists in this domain too. Using a dataset from 800,000+ users, we how the proposed techniques in CellScope can easily be adapted and demonstrate their effectiveness in mitigating the trade-off (§7.8).

7.2 Background and Motivation

We begin with a brief primer on LTE networks and the current state of RAN troubleshooting. Then, we illustrate the difficulties in applying ML for RAN performance diagnosis.

7.2.1 LTE Network Primer

LTE networks provide User Equipments (UEs) such as smartphones with Internet connectivity. When a UE has data to send to or receive from the Internet, it sets up a

communication channel between itself and the Packet Data Network Gateway (P-GW). This involves message exchanges between the UE and the Mobility Management Entity (MME). In coordination with the base station (eNodeB), the Serving Gateway (S-GW), and P-GW, data plane (GTP) tunnels are established between the base station and the S-GW, and between the S-GW and the P-GW. Together with the connection between the UE and the base station, the network establishes a communication channel called EPS bearer (short for bearer). The LTE network architecture is shown in fig. 7.1.

For network access and service, LTE network entities exchange control plane messages. A specific sequence of such control plane message exchange is called a *network procedure*. For example, when a UE powers up, it initiates an *attach procedure* with the MME which consists of establishing a radio connection, authentication and resource allocation. Each network procedure involves the exchange of several messages between two or more entities. Their specifications are defined by 3GPP Technical Specification Groups (TSG) [165].

Network performance degrades and end-user experience is affected when procedure failures happen. The complex nature of these procedures (due to the multiple underlying message and entity interactions) make diagnosing problems challenging. Thus, to aid RAN troubleshooting, operators collect extensive measurements from their network. These measurements typically consist of per-procedure information (e.g., attach). To analyze a procedure failure, it is often useful to look at the associated variables. For instance, a failed attachment procedure may be diagnosed if the underlying signal strength information was captured. Hence, relevant metadata is also captured with procedure information. Since there are hundreds of procedures in the network and each procedure can have many possible metadata fields, the collected data contains several hundreds of fields.

7.2.2 RAN Troubleshooting Today

Current RAN network monitoring depends on cell-level aggregate Key Performance Indicators (KPI). Existing practice is to use performance counters to derive these KPIs. The derived KPIs are then monitored by domain experts, aggregated over certain pre-defined time window. Based on domain knowledge and operational experience, these KPIs are used to determine if service level agreements (SLA) are met. For instance, an operator may have designed the network to have no more than 0.5% call drops in a 10 minute window. When a KPI that is being monitored crosses the threshold, an alarm is raised and a ticket created. This ticket is then handled by experts who investigate the cause of the problem, often manually. Several commercial solutions exists [15, 16, 17, 58] that aid in this troubleshooting procedure by enabling efficient slicing and dicing on data.

However, we have learned from domain experts that often it is desirable to apply different models or algorithms on the data for detailed diagnosis. Thus, many of the RAN trouble tickets end up with experts who work directly on the raw measurement data.

7.2.3 Machine Learning for RAN Diagnostics

The large volume of data collected in the RAN makes it an ideal candidate for the application of machine learning. We now discuss the difficulties in using ML for the purpose of RAN performance diagnostics.

Data

We obtained measurement data from the live network of a top tier operator in the United States. The data consists of four types of records:

Bearer Records: These log bearer level information. In our logs, such information includes extensive information, such as frame loss rate, physical radio resources allocated, radio channel quality, physical layer modulation and coding rate, bearer start and end time, bearer setup delay, failure reason code (if any), associated base station, MME, S-GW and P-GW.

Signaling Records: These are logs of network procedures, such as paging, attach/detach, and handoff information. Every procedure in the network creates a new record along with metadata information such as the time of the event.

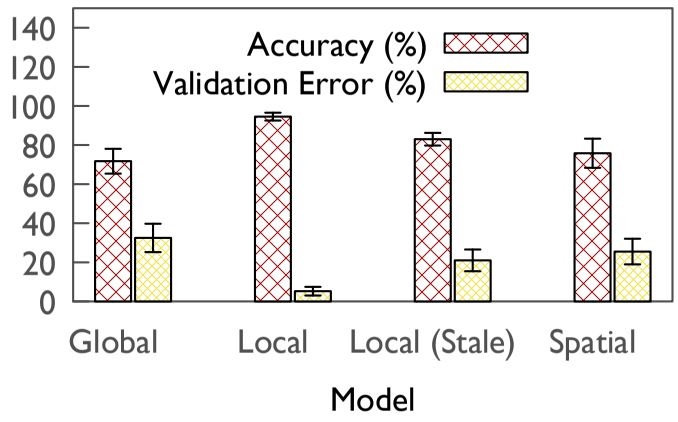
TCP Flow Records: These logs are from strategically placed probes in the network, and consists of TCP flow level information. They are associated with the bearer records to get more insights on application level information.

Network Element Records: These are aggregate information at network elements such as eNodeB or MME. Some fields in this record include total failures and downlink/uplink frames.

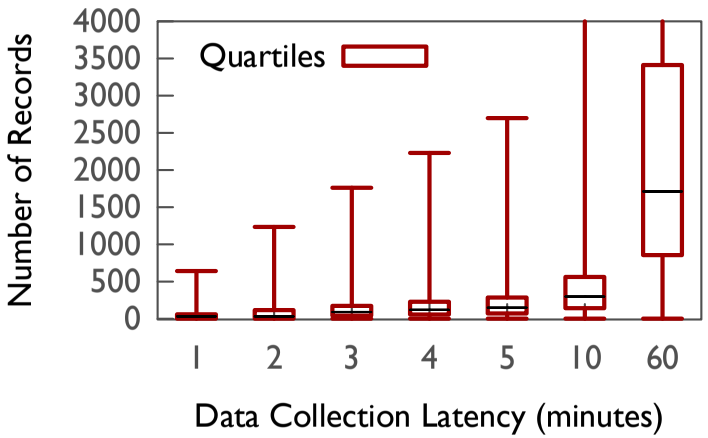
Collectively, the dataset contains over four hundred fields which could potentially be leveraged as individual features by a machine learning algorithm.

Ineffectiveness of Global Model

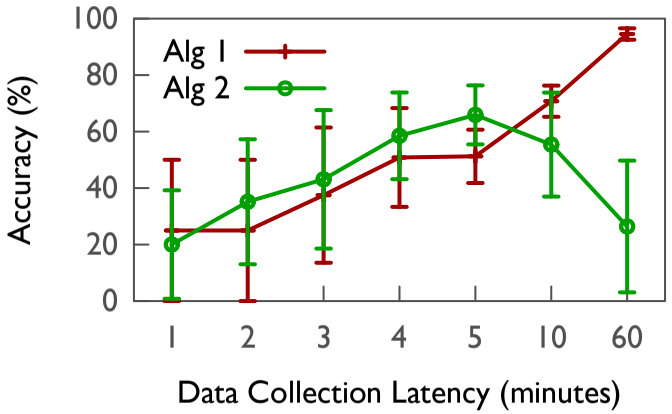
A common approach in applying ML on a dataset is to consider the dataset as a single entity and build one model over the entire data. However, base stations in a cellular



(a) A single global model incurs poor accuracy. Local model is the best, but requires enough data to build and frequent updates (otherwise staleness affects its performance). Combining data from geographically nearby base stations results in a reduction in accuracy.



(b) Distribution of data collected by base stations under various data collection latencies. It takes approximately an hour for a vast majority of the base stations to collect data to build statistically significant models.



(c) For building local models, it may not be possible to simply wait for enough data. A random forest model (Alg 1) gains from more data, but a lasso regression model (Alg2) degrades with more data due to temporal effects.

Figure 7.2: Simply applying ML for RAN performance diagnosis results in a fundamental trade-off between latency and accuracy.

network exhibit different characteristics. This renders the use of a global model ineffective. To illustrate this problem, we conducted an experiment where the goal is to build a model for call drops in the network (similar to [87]) using information in our traces. Specifically, we build a decision tree model using an hour worth of data to ensure sufficient data for the algorithm to produce statistically significant results. Figure 7.2a shows the result of this experiment, where we see that the global model achieves poor accuracy and high variance. This underlines the heterogeneity in the characteristics of base stations and hence the ineffectiveness of global models.

Latency/Accuracy Issues with Local Models

The alternative to a single global model is to build a model for every base station. We evaluate this approach by repeating the last experiment, but this time segregating the data for every base station and building an independent model for each. The results of this experiment is shown in fig. 7.2a, which indicates that local models are far superior, with up to 20% more accuracy while showing much lower variance.

It is natural to think of a per base station model as the final solution to this problem. However, this approach has issues too. Due to the difference in characteristics of the base stations, the amount of data they collect in a given time interval varies vastly. Thus, in small intervals, they may not generate enough data to produce statistically significant results. Figure 7.2b shows the distribution of the amount of data generated by these base stations under different data collection latencies. It shows that at small intervals (e.g., under 10 minutes), most base stations do not generate enough data, and that it takes about an hour for all quartiles of base stations to log reasonable number of records.

To illustrate the effect of this discrepancy, we conduct another experiment. We use two machine learning algorithms—a random forest model to predict connection drops (Alg 1), and a lasso regression model using stochastic gradient descent to predict the throughput (Alg 2)—at various data collection latencies. These two algorithms represent some of the commonly used models from the broad categories of classification and regression. The result of this experiment is shown in fig. 7.2c. The behavior of Alg 1 is obvious; as it gets more data its accuracy improves due to the slow varying nature of the underlying causes of failures. After an hour latency[1], it is able to reach a respectable accuracy. However, the second algorithm's accuracy initially seems to improve with more data, but falls quickly. This is counterintuitive in normal settings, but the explanation lies in the spatio-temporal characteristics of cellular networks. Many of the performance metrics exhibit high temporal variability, and thus need to be analyzed in smaller intervals. Thus,

[1]Such high latencies may not be acceptable in many scenarios.

in models like that in Alg 2, it is not enough to just "wait" for enough data to be collected, and hence local modeling is ineffective.

Need for Model Updates

An obvious, but flawed conclusion from our previous experiment is that models similar to that built by Alg 1 would work after the data collection latency (of an hour) has been incurred once. Put differently, can we just use historic data? In any application of ML, models need to be updated to retain their performance. This is true in cellular networks too, where temporal variations affect the performance of the model. To depict this, we repeated the experiment where we built per base station decision tree model for call drops. However, instead of training and testing on parts of the same dataset, we train on an hours worth of data, and test it on the next hour. Figure 7.2a shows that the accuracy drops by 12% with a stale model (because the model built using historic data is no longer valid). Thus, it is important to keep the model fresh by incorporating incoming data and removing old data. Such sliding updates to ML models in a general setting is difficult due to the overheads in retraining them from scratch. To add to this, cellular networks consist of several thousands of base stations. Thus, a per base station approach requires creating and updating a huge amount of models (e.g., our network consisted of over 13000 base stations). This makes scaling hard.

Why not Spatial/Temporal Partitioning?

Our experiments point towards the need for obtaining enough data for ML algorithms to produce statistically significant results with low latency. The obvious solution to combating this trade-off is to *intelligently* combine data from multiple base stations. It is intuitive to think of this as a spatial partitioning problem, since base stations in the real world are geographically separated. Thus, it is reasonable to assume that a spatial partitioner which combines data from base stations within a geographical region must be able to give good results. Unfortunately, this isn't the case which we motivate using a simple example. Consider two base stations, one situated at the center of times square in New York and the other a mile away at a residential area. Using a spatial partitioning scheme that divides the space into equal sized planes would likely result in combining data from these base stations. However, this is not desirable because of the difference in characteristics of these base stations[2]. We illustrate this using the drop modeling experiment as before. Figure 7.2a shows the performance where we combine data from

[2]In our measurements, a base station in a highly popular spot serves more than 300 UEs and carries multiple times uplink / downlink traffic compared to another base station situated just a mile from it that serves only 50 UEs.

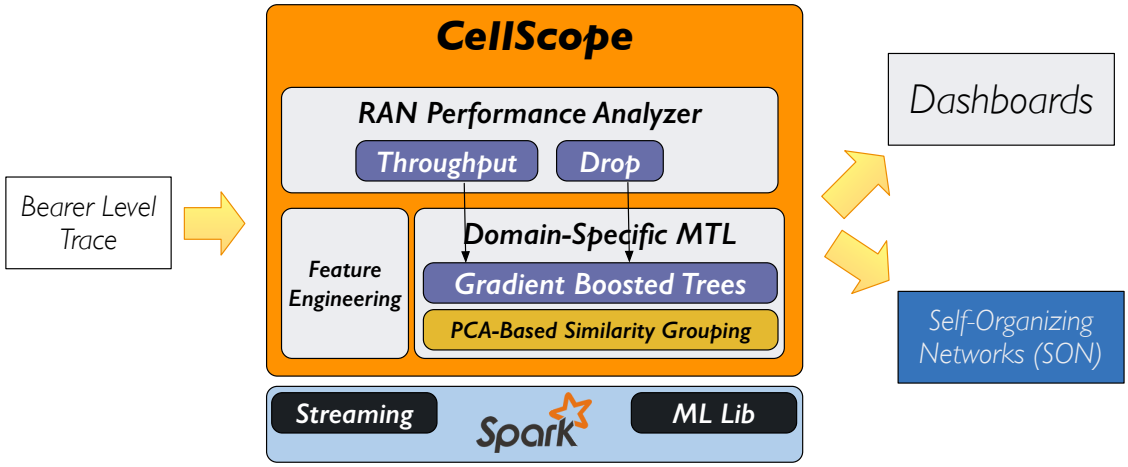


Figure 7.3: CellScope System Architecture.

nearby base stations using a simple grid partitioner, and then build a model in each of the partitions. The result shows that this technique is only slightly better compared to a single global model. We evaluate other spatial partitioning schemes in §7.6.

7.3 CellScope Overview

We now present our solution, CellScope, which mitigates the latency-accuracy trade-off using a domain-specific formulation and application of Multi-Task Learning (MTL).

7.3.1 Problem Statement

CellScope's ultimate goal is to enable *fast and accurate RAN performance diagnosis by resolving the trade-off between data collection latency and the achieved accuracy*. The key difficulty arises from the fundamental trade-off between *having not enough data to build accurate-enough models in short timespans* and *waiting to collect enough data that entails stale results*. Additionally, CellScope must also support efficient modifications to the learned models to account for the temporal nature of our setting to avoid data and model staleness.

7.3.2 Architectural Overview

Figure 7.3 shows the high-level architecture of CellScope, which consists of the following key components:

Input data: CellScope uses measurement traces that are readily available in modern cellular networks (§7.2.1). Base stations collect traces independently and send them to the associated MME, which merges records if required and uploads them to a data center.[3]
Feature engineering: Next, CellScope uses domain knowledge to transform the raw data and construct a set of features amenable to learning (e.g., computing interference ratios)(§7.4.1). We also leverage protocol details and algorithms (e.g., link adaptation in the physical layer).
Domain-specific MTL: CellScope uses a domain specific formulation and application of MTL that allows it to perform accurate diagnosis while updating models efficiently (§7.4.2).
Data partitioner: To enable correct application of MTL, CellScope implements a partitioner based on a similarity score derived from Principal Component Analysis (PCA) and geographical distance (§7.4.3). The partitioner segregates data to be analyzed into independent sets and produces a smaller co-located set relevant to the underlying analysis. This also minimizes the need to shuffle data during training.
RAN performance analyzer: This component binds everything together to build diagnosis modules. It leverages the MTL component and uses appropriate techniques to build call drop and throughput models. We discuss our experience of applying these techniques to a live LTE network in §7.7. This component can be easily replaced to extend CellScope to a new domain, as we show in §7.8.
Output: Finally, CellScope can output analytics results to external modules such as RAN performance dashboards. It can also provide inputs to Self-Organizing Networks (SON).

7.4 Mitigating Latency Accuracy Trade-off

In this section, we present how CellScope mitigates the trade-off between latency and accuracy. We first discuss a high-level overview of RAN specific feature engineering that prepares the data for learning (§7.4.1). Next, we describe CellScope's MTL formulation (§7.4.2), and how it lets us build fast, accurate and incremental models. Then, we explain how CellScope achieves grouping that captures commonalities among base stations using a novel PCA based partitioner that makes application of MTL possible (§7.4.3).

[3]The transfer of traces to a data center is not fundamental. Extending CellScope to do geo-distributed learning in a future work.

7.4.1 Feature Engineering

Feature engineering, the process of transforming the raw input data to a set of features that can be effectively utilized by machine learning algorithms, is a fundamental part of ML applications [195]. Generally carried out by domain experts, it is often the first step in ML.

In CellScope, the network measurement data contains several hundreds of fields (§7.2). These fields range from simple bearer identification information to fields associated with LTE network procedures. Unfortunately, many of these fields are not suitable for model building as it is. Additionally, several fields are collected in a format that utilizes a compact representation. Finally, these records are not self-contained, and multiple records may need to be "joined" to create a feature for a certain procedure. We use simple feature engineering to obtain fields that can be used in ML algorithms. As an example, for modeling connection drop rates, we use block error rate (BLER) as a feature. However, the records do not directly provide this value, thus it is computed using the block transfer information. Similarly, for throughput modeling, the downlink and uplink throughput values are computed using the amount of physical resource blocks allocated and the transfer time. While we depend on manual feature engineering in this work (automating this is part of our future work), not all fields need to be feature engineered. Further, we found that the engineered fields can be used across several ML algorithms.

7.4.2 Multi-Task Learning

The latency-accuracy trade-off makes it hard to achieve both low latency and high accuracy in ML tasks (§7.2). The ideal-case scenario in CellScope is if infinite amount of data is available per base station with zero latency. In this scenario, we would have a learning task for each base station that produce a model as an output with the best achievable accuracy. In reality, our setting has several tasks, each with its own data. However, each task does not have enough data to produce models with acceptable accuracy in a given latency budget. This makes our setting an ideal candidate for multi-task learning (MTL), a research area in machine learning that has been successful in many ML applications. The key idea behind MTL is to *learn from other tasks by weakly coupling their parameters so that the statistical efficiency of many tasks can be boosted* [42, 168, 60, 28]. Specifically, if we are interested in building a model of the form

$$h(x) = m(f_1(x), f_2(x), ..., f_k(x)) \tag{7.1}$$

where m is a model (e.g., to predict connection drop) composed of feature functions f_1 through f_k, then the traditional MTL formulation, given dataset $\mathcal{D} = \{(x_i, y_i, bs_i) : i =$

$1, ..., n\}$, where $x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$ and bs_i denotes the i^{th} base station, is to learn

$$h(x) = m_{bs}(f_1(x), f_2(x), ..., f_k(x)) \tag{7.2}$$

where m_{bs} is a per base station model.

In this MTL formulation, the core assumption is a shared structure or dependency across each of the learning problems. Unfortunately, in our setting, the base stations do not share a structure at a global level (§7.2). Due to their geographic separation and the complexities of wireless signal propagation, the base stations share a spatio-temporal structure instead. Thus, we proposes a new domain-specific MTL formulation.

CellScope's MTL Formulation

In order to address the difficulty in applying MTL due to the violation of task dependency assumption in RANs, we can leverage domain-specific characteristics. Although independent learning tasks (learning per base station) are not correlated with each other, they exhibit specific non-random structure. For example, the performance characteristics of base stations nearby are influenced by similar underlying features. Thus, we propose exploiting this knowledge to segregate learning tasks into groups of dependent tasks on which MTL can be applied. MTL in the face of dependency violation has been studied in the machine learning literature in the recent past [100, 69]. However, they assume that each group has its own set of features. This is not entirely true in our setting, where multiple groups may share most or all features but still need to be treated as separate groups. Furthermore, some of the techniques used for automatic grouping without a priori knowledge are computationally intensive.

Assuming we can club learning tasks into groups, we can rewrite the MTL equation in eq. (7.2) as:

$$h(x) = m_{g(bs)}(f_1(x), f_2(x), ..., f_k(x)) \tag{7.3}$$

where $m_{g(bs)}$ is the per-base station model in group g. We describe a simple technique to achieve this grouping based on domain knowledge in §7.4.3 and experimentally show that just grouping can achieve significant gains in §7.6.

In theory, the MTL formulation in eq. (7.3) should suffice for our purposes as it would perform much better by capturing the inter-task dependencies using grouping. However, this formulation still builds an independent model for each base station. Building and managing a large amount of models leads to significant performance overhead and would impede our goal of scalability. Scalable application of MTL in a general setting is an active area of research in machine learning [134], so we turn to problem-specific optimizations to address this challenge.

The model $m_{g(bs)}$ could be built using any class of learning functions. In this work, we restrict ourselves to functions of the form $F(x) = w \cdot x$ where w is the weight vector associated with a set of features x. This simple class of function gives us tremendous leverage in using standard algorithms that can easily be applied in a distributed setting, thus addressing the scalability issue. In addition to scalable model building, we must also be able to update the built models fast. However, machine learning models are typically hard to update in real time. To address this challenge, we discuss a hybrid approach to building the models in our MTL setting next.

Hybrid Modeling for Fast Model Updates

Estimation of the model in eq. (7.3) could be posed as an ℓ_1 regularized loss minimization problem [169]:

$$min \sum L(h(x : f_{bs}), y) + \lambda ||R(x : f_{bs})|| \tag{7.4}$$

where $L(h(x : f_{bs}), y)$ is a non-negative loss function composed of parameters for a particular base station, hence capturing the error in the prediction for it in the group, and $\lambda > 0$ is a regularization parameter scaling the penalty $R(x : f_{bs})$ for the base station. However, the temporal and streaming nature of the data means that the model must be refined frequently for minimizing staleness.

Fortunately, grouping provides us an opportunity to solve this. Since the base stations are grouped into correlated task clusters, we can decompose the features used for each base station into a *shared common set* f_c and a *base station specific* set f_s. Thus, we can modify eq. (7.4) as minimizing

$$\sum \left(\sum L(h(x : f_s, f_c), y) + \lambda ||R(x : f_s)|| \right) + \lambda ||R(x : f_c)|| \tag{7.5}$$

where the inner summation is over dataset specific to each base station. This separation gives us a powerful advantage. Since we grouped base stations, the feature set f_s is minimal, and in most cases just a weight vector on the common feature set. Because the core common features do not change often, we need to update only the base station-specific parts in the model frequently, while the common set can be reused. Thus, we end up with a hybrid offline-online model. Furthermore, the choice of our learning functions lets us apply stochastic methods [159] which can be efficiently parallelized.

Anomaly Detection Using Concept Drift

A common use case of learning tasks for RAN performance analysis is in detecting anomalies. For instance, an operator may be interested in learning if there is a sudden

increase in call drops. At the simplest level, it is easy to answer this question by monitoring the number of call drops at each base station. However, just a yes or no answer to such questions are seldom useful. If there is a sudden increase in drops, then it is useful to understand if the issue affects a complete region and its root cause.

Our MTL approach and the ability to do fast incremental learning enables a better solution for anomaly detection and diagnosis. Concept drift is a term used to refer the phenomenon where the underlying distribution of the training data for a machine learning model changes [65]. CellScope leverages this to detect anomalies as concept drifts and proposes a simple technique for it. Since we process incoming data in mini-batches (§7.5), each batch can be tested quickly on the existing model for significant accuracy drops. An anomaly occurring just at a single base station would be detected by one model, while one affecting a larger area would be detected by many. Once anomaly is detected, finding cause is as easy as updating the model and comparing it with the old.

7.4.3 Data Grouping for MTL

Having discussed CellScope's MTL formulation, we now turn our focus towards how CellScope achieves efficient grouping of cellular datasets that enables accurate learning. Our data partitioning is based on Principal Component Analysis (PCA), a widely used technique in multivariate analysis [136]. PCA uses an orthogonal coordinate transformation to map a given set of points into a new coordinate space. Each of the new subspaces are commonly referred to as a principal component. Since the coordinate space is smaller than the original , PCA is used for dimensionality reduction.

In their pioneering work, Lakhina et.al. [106] showed the usefulness of PCA for network anomaly detection. They observed that it is possible to segregate normal behavior and abnormal (anomalous) behavior using PCA—the principal components explain most of the normal behavior while the anomalies are captured by the remaining subspaces. Thus, by filtering normal behavior, it is possible to find anomalies that may be otherwise undetected.

While the most common usecase for PCA has been dimensionality reduction (in machine learning domains) or anomaly detection (in networking domain), we use it in a novel way, to enable grouping of datasets for multi-task learning. Due to the lack of the ability to collect sufficient amount of data from individual base stations, detecting anomalies in them will not yield results. However, the data would still yield an explanation of normal behavior. We use this observation to partition the dataset.

Notation

As bearer level traces are collected continuously, we consider a buffer of bearers as a *measurement matrix* A. Thus, A consists of m bearer records, each having n observed parameters making it an $m \times n$ time-series matrix. It is to be noted that n is in the order of a few 100 fields, while m can be much higher depending on how long the buffering interval is. We enforce n to be fixed in our setting—every measurement matrix must contain n columns. To make this matrix amenable to PCA analysis, we adjust the columns to have zero mean. By applying PCA to any measurement matrix A, we can obtain a set of k principal components ordered by amount of data variance they capture.

PCA Similarity

It is intuitive to see that many measurement matrices may be formed based on different criteria. Suppose we are interested in finding if two measurement matrices are *similar*. One way to achieve this is to compare the principal components of the two matrices. Krzanowski [103] describes such a *Similarity Factor* (SF). Consider two matrices A and B having the same number of columns, but not rows. The similarity factor between A and B is:

$$SF = trace(LM'ML') = \sum_{i=1}^{k} \sum_{j=1}^{k} \cos^2 \theta_{ij}$$

where L, M are the first k principal components of A and B respectively, and θ_{ij} is the angle between the i^{th} component of A and the j^{th} component of B. The similarity factor considers all combinations of k components from both matrices.

CellScope's Similarity Metric

Similarity in our setting bears a slightly different meaning: we do not want strict similarity between measurement matrices, but only similarity between corresponding principal components. This ensures that algorithms will still capture the underlying major influences and trends in observation sets that are not exactly similar. So we propose a simpler metric.

Consider two measurement matrices A and B as before, where A is of size $m_A \times n$ and B is of size $m_B \times n$. By applying PCA on the matrices, we can obtain k principal components using a heuristic. We obtain the first k components which capture 95% of the variance. From the PCA, we obtain the resulting weight vector, or *loading*, which is a $n \times k$ matrix: for each principal component in k, the loading describes the weight on the original n features. Intuitively, this can be seen as a rough measure of the influence of

each of the n features on the principal components. The Euclidean distance between the corresponding loading matrices gives

$$SF_{\text{CellScope}} = \sum_{i=1}^{k} d(a_i, b_i) = \sum_{i=1}^{k} \sum_{j=1}^{n} |a_{ij} - b_{ij}|$$

where a and b are the column vectors representing the loadings for the corresponding principal components from A and B. Thus, $SF_{\text{CellScope}}$ captures how closely the underlying features explain the variation in the data.

Due to the complex interactions between network components and the wireless medium, many of the performance issues in RANs are geographically tied (e.g., congestion might happen in nearby areas, and drops might be concentrated)[4]. However, $SF_{\text{CellScope}}$ doesn't capture this phenomenon because it only considers similarity in normal behavior. Consequently, it is possible for anomaly detection algorithms to miss geographically-relevant anomalies. To account for this domain-specific characteristic, we augment our similarity metric to also capture the geographical closeness by weighing the metric by geographical distance between the two measurement matrices. Our final similarity metric is[5]

$$SF_{\text{CellScope}} = w_{\text{distance}_{(A,B)}} \times \sum_{i=1}^{k} \sum_{j=1}^{n} |a_{ij} - b_{ij}|$$

Using Similarity Metric for Partitioning

With similarity metric, CellScope can now partition bearer records. We first group the bearers into measurement matrices by segregating them based on the cell on which the bearer originated. The grouping is based on our observation that the cell is the lowest level at which an anomaly would manifest. We then create a graph $G(V, E)$ where the vertices are the individual cell measurement matrices. An edge is drawn between two matrices if the $SF_{\text{CellScope}}$ between them is below a threshold. To compute $SF_{\text{CellScope}}$, we simply use the geographical distance between the cells as the weight. Once the graph has been created, we run connected components on this graph to obtain the partitions. The use of connected component algorithm is not fundamental, it is also possible to use a clustering algorithm instead. For instance, a k-means clustering algorithm that could leverage $SF_{\text{CellScope}}$ to merge clusters would yield similar results.

[4]Proposals for conducting geographically weighted PCA (GW-PCA) exist [78], but they are not applicable since they assume a smooth decaying user provided bandwidth function.

[5]A similarity measure for multivariate time series is proposed in [189], but it is not applicable due to its stricter form and dependence on finding the right eigenvector matrices to extend the Frobenius norm.

Managing Partitions Over Time

One important consideration is managing group changes over time. To detect group changes, it is necessary to establish correspondence between groups across time intervals. Once this correspondence is established, CellScope's hybrid modeling makes it easy to accommodate changes. Due to the segregation of our model into common and base station specific components, small changes to the group do not affect the common model. In these cases, we can simply bootstrap the new base station using the common model, and then start learning specific features. On the other hand, if there are significant changes to a group, then the common model may no longer be valid, which is easy to detect using concept drift. In such cases, the offline model could be rebuilt.

7.4.4 Summary

We now summarize how CellScope resolves the fundamental trade-off between latency and accuracy. To cope with the fact that individual base stations cannot produce enough data for learning in a given time budget, CellScope uses MTL. However, our datasets violate the assumption of learning task dependencies. As a solution, we proposed a novel way of using PCA to group data into sets with the same underlying performance characteristics. Directly applying MTL on these groups would still be problematic in our setting due to the inefficiencies with model updates. To solve this, we proposed a new formulation for MTL which divides the model into an offline and online hybrid. On this formulation, we proposed using simple learning functions are amenable to incremental and distributed execution. Finally, CellScope uses a simple concept drift detection to find and diagnose anomalies.

7.5 Implementation

We have implemented CellScope on Spark [192]. We describe its API that exposes our commonality based grouping based on PCA (§7.5.1), and implementation details on the hybrid offline-online MTL models (§7.5.2).

7.5.1 Data Grouping API

CellScope's grouping API is built on Spark Streaming [194], since the data arrives continuously, and we need to operate on this data in a streaming fashion. Spark Streaming already provides support for windowing functions on streams of data, thus we extended it with the three APIs in listing 7.1.

```
grouped = DStream.groupBySimilarityAndWindow(
                        windowDuration, slideDuration)

reduced = DStream.reduceBySimilarityAndWindow(
                        func, windowDuration, slideDuration)

 joined = DStream.joinBySimilarityAndWindow(
                        windowDuration, slideDuration)
```

Listing 7.1: Grouping API

The APIs leverage the `DStream` abstraction provided by Spark Streaming. `groupBySimilarityAndWindow` takes the buffered data from the last window duration, applies the similarity metric to produce outputs of grouped datasets every slide duration. `reduceBySimilarityAndWindow` allows an additional user defined associative reduction operation on the grouped datasets. Finally, `joinBySimilarityAndWindow` joins multiple streams using similarity.

7.5.2 Hybrid MTL Modeling

We use Spark's machine learning library, MLlib [162] for implementing our hybrid MTL model. MLlib contains implementation for many distributed learning algorithms. The MTL formulation we presented in §7.4.2 allows us to utilize these existing models in our framework.

In MTL, the tasks learn from each other. These tasks in our setting consist of building a model, $m_{g(bs)}$ for each base station in every group created by the PCA based grouping. In eq. (7.3), we presented our MTL formulation, and described a simplified loss minimization method to estimate this model. Further, in eq. (7.5), we decomposed this into shared and base station specific set, so the model $m_{g(bs)}$ is of the general form $h(x : f_s, f_c), y)$. Since we restrict ourselves to learning functions of the form $w \cdot x$ for this model, our model per base station is simply a weight vector on the shared group model. This allows the usage of existing ensemble methods [53]. Ensemble methods use multiple learning algorithms to obtain better performance. In our case, we use the ensemble method to learn the shared group model. This can be done in many ways: we can directly employ existing ensemble methods, or we can leverage multiple algorithms to be components of the ensemble. However, unlike normal ensemble methods where the output is aggregated, we use the MTL approach of a task per base station to learn the per-base station model. This is equivalent to a linear model on the individual ensemble components, which gives us the weight vector.

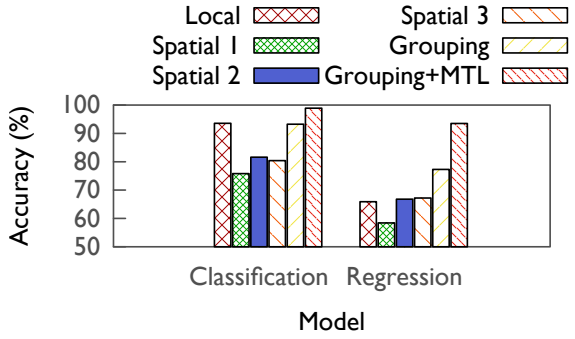
We modified the MLLib implementation of Gradient Boosted Tree (GBT) [64]. This implementation supports both classification and regression, and internally uses stochastic methods. We implement the group's shared feature model using either the GBT's ensemble, or individual algorithms. As an example, for connection drop prediction, the shared model can be obtained using the standard ensembles such as the GBT itself, or random forests. Then, we use individual base station data to fit a linear model on the individual ensemble components. Note that it is not necessary to build the base model this way—we could also use multiple learning methods as ensemble components. In the same example, our ensemble could consist of a combination of SVM and decision trees. Similarly, for throughput prediction, the shared model is built as an ensemble of regression models—for instance, we may use one model for low throughput and another for high throughput, and each of these tasks could use a different standard learning method. In this method, we can update the base station specific weight vector in real time as data is streamed in, as we simply need to update the linear model. Further, the group specific model can be periodically retrained. One way to do so is to simply add more models to the ensemble when new data comes in. Our implementation allows weighing the outcome to give more weights to the latest models.

7.6 Evaluation

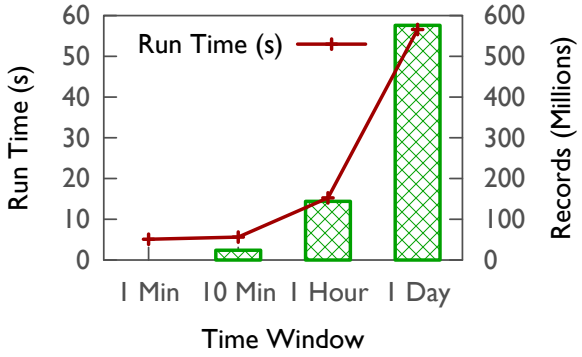
We have evaluated CellScope through a series of experiments on real-world cellular traces from a live LTE network in a large geographical area. Our results show that:

- CellScope's similarity based grouping provides up to 10% improvement in accuracy on its own compared to the best space partitioning scheme.
- With MTL, CellScope's accuracy improvements range from 2.5× to 4.4× over different collection latencies.
- Our hybrid online-offline model is able to reduce model update times upto 4.8× and is able to learn changes in an online fashion with no loss in accuracy.

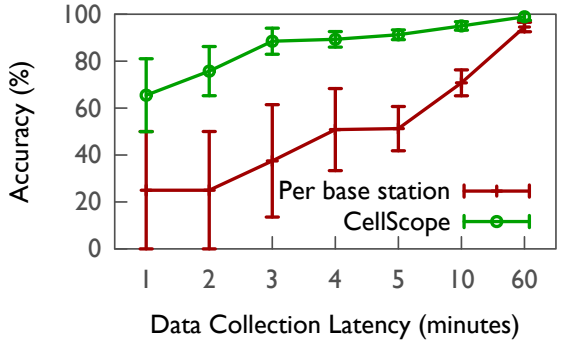
Evaluation Setup: We use a private cluster of 20 machines, each consisting of 4 CPUs, 32GB RAM and 200GB hard disk.
Dataset: We collected data from a major metro-area LTE network for a time period of over 10 months. It serves over 2 million active users and carries over 6TB traffic per hour.



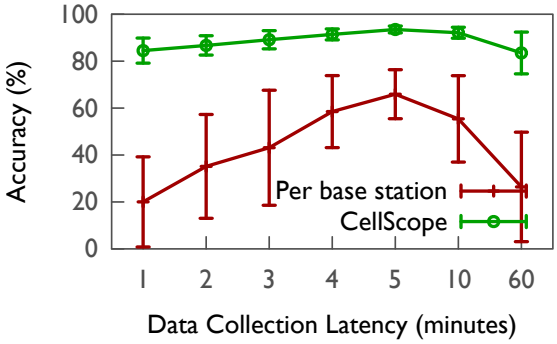
(a) Grouping by itself is able to provide significant gains. MTL provides further gains.



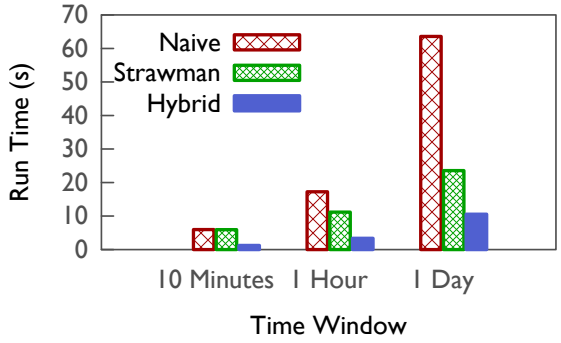
(b) Grouping is not computationally intensive, even a days worth of data (with >500M records) can be grouped in under a minute.



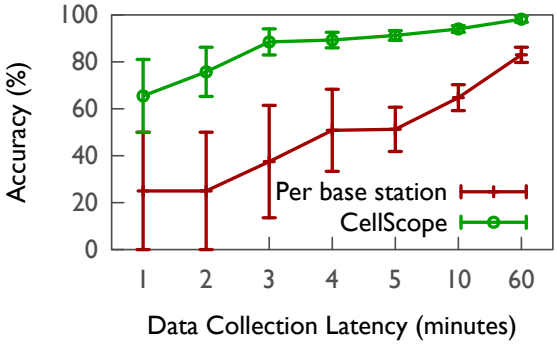
(c) CellScope achieves up to 2.5× accuracy improvements in drop rate classification.



(d) Improvements in throughput regression go up to 4.4×



(e) CellScope's hyrid model allows efficient updates, and reduces update time by up to 4.8×.



(f) Online training due to the hybrid model helps avoid the loss in accuracy due to staleness of the model.

Figure 7.4: CellScope achieves high accuracy while reducing the data collection latency.

7.6.1 Benefits of Similarity Based Grouping

We first attempt to answer the question *"How much benefits do the similarity based grouping provide?"*. For this, we conducted two experiments, each with a different learning algorithm. The first experiment, detection of call drops, uses a classification algorithm while the second, throughput prediction, uses a regression algorithm. We chose these to evaluate the benefits in two different classes of algorithms. In both these cases, we pick the data collection latency where the per base station model gives the best accuracy, which was 1 hour for classification and 5 minutes for regression. In order to compare the benefits of our grouping scheme alone, we build a single model per group instead of applying MTL. We compare the accuracy obtained with three different space partitioning schemes. The first scheme (Spatial 1) just partitions space into grids of equal size. The second (Spatial 2) uses a sophisticated space-filling curve based approach [92] that could create dynamically sized partitions. Finally, the third (Spatial 3) creates partitions using base stations that are under the same region. Figure 7.4a shows the results.

CellScope's similarity grouping performs as good as the per base station model which gives the highest accuracy. It is interesting to note the performance of spatial partitioning schemes which ranges from 75% to 80%. None of the spatial schemes come close to the similarity grouping results. This is because the drops are few, and concentrated. Spatial schemes club base stations not based on underlying drop characteristics, but only based on spatial proximity. This causes the algorithms to underfit or overfit. Since our similarity based partitioner groups base stations using the drop characteristics, it is able to do as much as 17% better.

The benefits are even higher in the regression case. Here, the per base station model is unable to get enough data to build an accurate model and hence is only able to achieve around 66% accuracy. Spatial schemes are able to do slightly better than that. Our similarity based grouping emerges as a clear winner in this case with 77.3% accuracy. This result depicts the highly variable performance characteristics of the base stations, and the need to capture them for accuracy. These benefits do not come at the cost of computational overhead due to grouping. Figure 7.4b shows the overhead of similarity based grouping on various dataset sizes.

7.6.2 Benefits of MTL

Next, we characterize the benefits of CellScope's use of MTL. We repeated the experiment before, and apply MTL to the grouped data to see if the accuracy improves compared to the earlier approach of a single model per group. The results are presented in figure 7.4a. The ability of MTL to learn and improve models from other similar base

stations' data results in an increase in the accuracy. Over the benefits of grouping, we see an improvement of 6% in the connection drop diagnosis experiment, and 16.2% in the case of throughput prediction experiment. The higher benefits in the latter comes from CellScope's ability to capture individual characteristics of the base station. This ability is not so crucial in the former because of the limited variation in individual characteristics.

7.6.3 Combined Benefits

Here, we are interested in evaluating how CellScope handles the latency accuracy trade-off. We do the same classification and regression experiments, but on different data collection latencies. We show the results from the classification and regression experiment in fig. 7.4c and fig. 7.4d, which compares CellScope's accuracy against a per base station model's.

When the opportunity to collect data at individual base stations is limited, CellScope is able to leverage our MTL formulation to combine data from multiple base stations, and build customized models to improve the accuracy. The benefits of CellScope ranges up to 2.5× in the classification experiment, to 4.4× in the regression experiment. Lower latencies are problematic in the classification experiment due to the extremely low probability of drops, while higher latencies are a problem in the regression experiment due to the temporal changes in performance.

7.6.4 Hybrid model benefits

Finally, we are interested in learning how much overhead it reduces during model updates, and if it do online learning.

To answer the first question, we conducted the following experiment: we considered three different data collection latencies: 10 minute, 1 hour and 1 day. We then learn a decision tree model on this data in a tumbling window fashion. So for the 10 minute latency, we collect data for 10 minutes, then build a model, wait another 10 minutes to refine the model and so on. We compare our hybrid model strategy to two different strategies: a naive approach which rebuilds the model from scratch every time, and a better, strawman approach which reuses the last model, and makes changes to it. Both builds a single model while CellScope uses our hybrid MTL model and only updates the online part of the model. The results of this experiment is shown in figure 7.4e.

The naive approach incurs the highest overhead, which is obvious due to the need to rebuild the entire model from scratch. The overhead increases with the increase in input data. The strawman approach, on the other hand, is able to avoid this heavy overhead. However, it still incurs overheads with larger input because of its use of a single model

which requires changes to many parts of the tree. CellScope incurs the least overhead, due to its use of multiple models. When data accumulates, it only needs to update a part of an existing tree, or build a new tree. This strategy results in a reduction of up to 2.2× to 4.8× in model building time for CellScope.

To wrap up, we evaluated the performance of the hybrid strategy on different data collection intervals. Here we are interested in seeing if the hybrid model is able to adapt to data changes and provide reasonable accuracies. We use the connection drop experiment again, but do it in a different way. At different collection latencies, we build the model at the beginning of the collection and use the model for the next interval. Hence, for the 1 minute latency, we build a model using the first minute data, and use the model for the second minute (until the whole second minute has arrived). The results are shown in figure 7.4f. We see here that the per base station model suffers an accuracy loss at higher latencies due to staleness, while CellScope incurs almost zero loss in accuracy. This is because it doesn't wait until the end of the interval, and is able to incorporate data in real time.

7.7 Real World RAN Analysis

We now turn to the question of how could operators benefit from a system such as CellScope? We try to answer this question in two ways: first, we try to evaluate what are the benefits of automatic root-causing and how much effort is reduced for the operator because of this feature. Second, we evaluate CellScope's ability to analyze in the wild.

7.7.1 Time Savings to the Operator

Operators spend several billions of dollars in diagnosing network problems. Often, finding the cause of a network problem takes hours, or even days of effort. To evaluate how CellScope could cut down this effort, we collected network trouble tickets from the operator. The operator logs tickets at different levels, so we look at trouble tickets that were investigated by domain experts using state-of-the-art tools such as datacubes. For each ticket where the operator has network data available, we used CellScope to diagnose the problem. This way, we can evaluate the potential time savings CellScope provides. We discuss four real trouble tickets, the time taken by CellScope is depicted in table 7.1.

Ticket	*Resolution Time*	CellScope
§7.7.1	3 days	10 minutes
§7.7.1	1 day	2 minutes
§7.7.1	7 days	15 minutes
§7.7.1	1 hour	1 minute

Table 7.1: CellScope is able to reduce operator effort by several orders of magnitude. Resolution time includes field trials & expert analysis using datacubes / state-of-the-art tools [15].

Throughput Degradation After Upgrade

This ticket reported that a number of users experienced degraded network throughput after a network upgrade. In many cases, throughput decrease of up to 30% was observed. Since not all of the users saw this problem, the operator had to conduct field trials to find the root cause of the problem. We used CellScope to model the throughput **before** and **after** the upgrade. Comparing the models, we noticed that a cluster of base stations had one feature influencing the model heavily. This matched the operator's ticket resolution—the field trials in the ticket indicated that the problem was cluster-wise and that it was because the feature CellScope was erroneously turned on after the upgrade. The base stations CellScope identified matched those reported in the resolution. In this case, the ticket was resolved in three days including the field trials, while our modeling on CellScope took less than 10 minutes. Note that manually applying learning techniques would not have found the problem without grouping.

Specific Patterns of Call Drops

Here, the operator reported consistent call drops (specifically, VoLTE call drops) in certain areas of the network. Manually analyzing this would have required a domain expert to slice and dice several TB of data to find a pattern and then dig deep into the pattern. To reduce this effort, the operator conducted field trials in parts affected to obtain test data that is manageable for the expert, who was able to identify the problem: a missing neighbor configuration in a group of base stations.

We used CellScope to model the call drop in an expanded portion of the network. After the grouping process, one particular group's model indicated that drops happened when a handoff procedure was triggered and the procedure failed due to a specific error code at the base station, `missingneighbor`. Here, the field trial, and domain expert's analysis was completed in one business day, while CellScope did the grouping and modeling on one day's data in 2 minutes.

Periodic Throughput KPI Degradation

The operator noticed a degradation of KPI in the network. The degradation happened in some serving cells. However, this was not consistently noticed, and occurred irregularly. To add, the problem was transient. Thus, the ticket required a week worth of effort to diagnose since field trials did not prove to be of help. We used CellScope in the following fashion: we replayed the data for days when KPI degradation was reported. We then built incremental models for drop rate and throughput. We then look at the intervals when CellScope refines the model due to accuracy loss using the *concept drift* and look at the model changes. We noticed that during some specific intervals, call drops spiked in some cells while the throughput of the entire cell dropped. The difference in the models built by CellScope indicated that device specific features influenced the drops. The reason was that a particular model and software version of a device creating a deluge of control messages that affected the entire cell when it was near capacity. The ticket closure confirmed this.

Periodic Call Drops

Here, the operator noticed periodic increase in call drops. The domain expert was able to identify the problem in an hour as PCI collision due to her vast expertise in the domain by looking through the logs from affected period. We used the same logs in CellScope, and were able to generate a call drop model that explained the drops using inter-cell interference. When expertise is not available, the ticket would have been time-consuming.

7.7.2 Analysis in the Wild: Findings

To validate our system in the real world, we used CellScope for RAN performance analysis on the live LTE network. Based on our experience with trouble tickets, we considered two metrics that are of significant importance for end-user experience: *throughput* and *connection drops*. In this section, we present some of our main findings (which were previously unknown to the operator) and the role played by CellScope.

SINR Anomaly

In a particular week, we noticed that an implementation of a learning task for connection drop predicted unusually high numbers of drops. These high drops happened at some base stations, all of which were assigned to the same group in CellScope's grouping. Upon further investigation with help from network experts, it was revealed that these base stations had been experiencing unusually higher levels of interference.

Incorrect Parameters

Similarly, we implemented a throughput prediction model. During a month long observation, we noticed that the predicted throughput for a set of base stations had fallen below its normal average after a certain date. It was found that the base stations were connected to the same MME and that a software upgrade had set some parameters affecting the throughput incorrectly. This was one of several misconfigurations we found in the network that caused performance degradation. Others included incorrect neighbor assignments and hand-off problems.

Real-time Monitoring

We simulated real-time monitoring of the network and CellScope's ability to detect performance problems. The current approach taken by the operator is to define SLAs for KPIs and then monitor them for SLA violations. However, such aggregate metrics are likely to miss many events. We used CellScope to monitor the network over a month, and verify if the events predicted by CellScope matches ground truth. Not only did CellScope detect 100% of the KPI SLA violations, it also found a few issues that were missed by the KPI based monitoring system, and later logged as trouble tickets.

Measurement Error

We also found problems in network measurements. Specifically, during initial deployment trials of CellScope, we noticed that using the feature engineered field of block error rate resulted in poor accuracy. The reason for this was an uninitialized field in the measurement record logger, which resulted in random values.

7.8 Extending CellScope to a New Domain

To show the generality of the techniques presented in this chapter, we now apply these techniques to a new domain: *energy anomaly detection in mobile phones* [128]. We obtained a dataset of measurements from approximately 800,000 users obtained using the Carat app. The goal here is to suggest actions to users that help improve their battery life. This can be done by building a battery usage model for each user.

Data: The Carat app periodically collects a variety of data from the mobile phone it is running on, including the phone model, version of the operating system, the state of the battery, the CPU and memory utilization and the applications that are running. We use these fields to build a ML model that predicts the battery drain rate for a user. Using

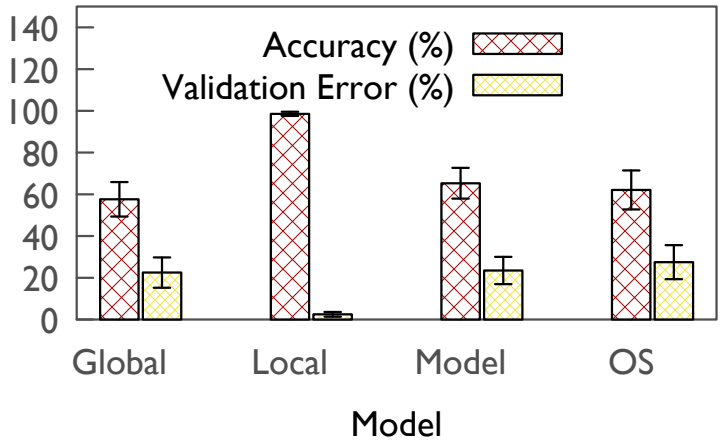


Figure 7.5: Other domains suffer from latency-accuracy trade-off. Here, we see the problem in the domain of energy debugging for mobile devices. Grouping by phone model or phone operating system does not give benefits.

this model, it is possible to point out potential application that are responsible for an increased battery drain.

Latency-Accuracy Trade-off: For users signing up for the Carat app, it is desirable to provide suggestions as soon as possible. However, currently, it takes several weeks for the app to collect enough data for a new user. Figure 7.5 shows the results of building a model for suggesting apps that are bugs for a particular user once enough data has been collected. It can be seen that a per-user model (denoted *Local*) works the best, but at the cost of latency. The local model performs poorly until enough data has been collected as depicted in fig. 7.6. A global model can be built immediately, but has poor accuracy. It is intuitive to think of grouping users who have the same model device together, or same operating system together. However, these grouping (denoted *Model* and OS) does not yield significant benefits. Further, as people install/uninstall apps, the models need to be updated. This make the domain ideal for testing CellScope's techniques.

Extending Similarity Metric and MTL: To extend our techniques to a new domain, we need to (i) customize the similarity metric (used for grouping) to the domain under consideration, and (ii) modify the MTL formulation in eq. (7.5) for this domain. In the cellular networks domain, our similarity metric was weighted by geographic distance between base stations. However, geographic distance does not have an effect here. From fig. 7.5, we notice that device model and operating system also do not make much

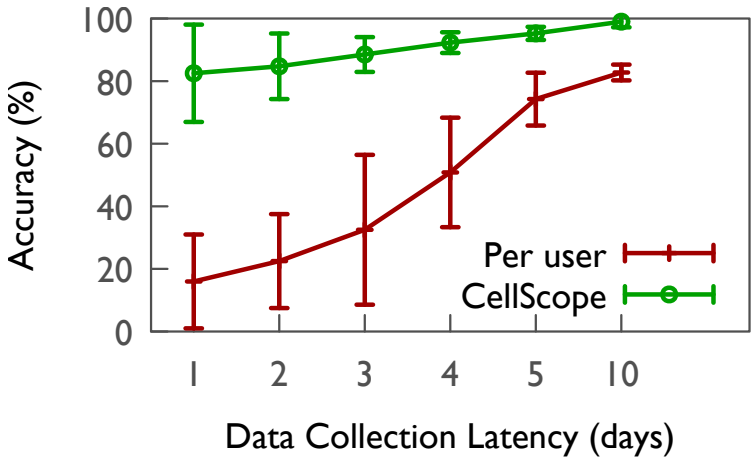


Figure 7.6: CellScope's techniques can easily be extended to new domains, and can benefit them. Here, using our techniques, models built are usable immediately while without CellScope, Carat [128] takes more than a week to build a model that is usable.

difference either. Intuitively, the subset of apps common between the users should provide better results. However, just that alone is not enough as usage patterns vary across users with similar apps. The Carat dataset provides enough information to determine the number of times each app is active, which is roughly an indicator of the usage pattern for the user. We use that to derive usage similarity between users, $u_{usage_{(A,B)}}$, and utilize that to form the similarity metric:

$$SF_{CellScope} = u_{usage_{(A,B)}} \times \sum_{i=1}^{k} \sum_{j=1}^{n} |a_{ij} - b_{ij}|$$

The MTL formulation remains the same as in eq. (7.5), we simply replace f_s with per-user features f_u.

We implemented a *Mobile Energy Diagnosis* module in CellScope at the same level as the RAN Performance Analyzer in fig. 7.3 that uses our modified similarity metric and MTL formulation. We then applied the grouping and learning to the measurement data we obtained to build a model for suggesting bugs to a new user. The results are shown in fig. 7.6 which shows the accuracy of models built with (denoted CellScope) and without CellScope (denoted *per-user*) starting from the day a user installs Carat. We see that on the day of signing up, the accuracy of the model built without using CellScope is unusable. This is intuitive, since only a few samples have been sent by the new user's device. Over time, the user sends enough data and the accuracy improves. However, it takes over a week for Carat to offer usable suggestions to a new user. In

contrast, with CellScope, we are able to build models that are immediately usable, and Carat can begin offering suggestions on day 1.

7.9 Related Work

Monitoring and Troubleshooting Network monitoring and troubleshooting has been an active area of research in both wired networks [77, 96, 186] and wireless networks [50, 15, 58, 16]. These techniques do not employ machine learning for troubleshooting. Systems targeting RAN [58, 16] typically monitor aggregate KPIs and per-bearer records separately. Their root cause analysis of KPI problems correlates with aggregation air interface metrics such as SINR histograms and configuration data. Because these systems rely on traditional database technologies, it is hard for them to provide fine-grained prediction based on bearer models. Recent research [92] and commercial offerings [18] have looked at the problem of scalable cellular network analytics by leveraging big data frameworks. However, they do not support learning tasks. In contrast, CellScope focuses on scalable and accurate application of machine learning in such domains.

Self-Organizing Networks (SON) The goal of SON [1] is to make the network capable of self-configuration (e.g. automatic neighbor list configuration) and self-optimization. CellScope's techniques can provide the necessary diagnostics capabilities for assisting SON.

Modeling and Diagnosis Techniques Problem diagnosis in cellular networks has been explored extensively in the literature in various forms [26, 79, 110, 145, 167, 87]. The focus of these has either been detecting faults or finding the root cause of failures. A vast majority of such techniques depend on aggregate information and correlation based fault detection. [87] discusses the shortcomings of using aggregate KPIs, and propose the use of fine-grained information. Some studies have focused on understanding the interaction of applications and cellular networks [142, 149, 172, 81, 94]. These are largely orthogonal to our work.

Finally, some recent proposals leverage the use of ML for specific tasks. In [167], the authors discuss the use of ML tools in predicting impending call drops and its duration. A probabilistic system for auto-diagnosing faults in RAN is presented in [26]. It uses KPIs as inputs to the model. [25] shows that improving signal-to-noise ratio, decreasing load and reducing handovers in cellular networks can improve web quality of experience by using ML to model the influence of radio network characteristics on user experience metrics. Our previous work [87] proposed the use of simple, explainable ML models

towards the quest of automating RAN problem detection and diagnosis, and discussed several challenges in leveraging ML. In this work, we present techniques that can solve the challenges in leveraging ML in many domains.

Multi-Task Learning MTL builds on the idea that related tasks can learn from each other to achieve better statistical efficiency [42, 168, 60, 28]. Since the assumption of task relatedness do not hold in many scenarios, techniques to automatically cluster tasks have been explored in the past [100, 69]. However, these techniques consider tasks as black boxes and hence cannot leverage domain specific structure. CellScope proposes a hybrid offline-online MTL formulation on domain-specific grouping of tasks based on the underlying performance characteristics.

7.10 Summary

The practicality of real-time decision making in many domains generating connected data is impeded by a fundamental trade-off between data collection latency and analysis accuracy. In this chapter, we first exposed this trade-off using the domain of cellular networks RAN. We presented CellScope to resolve this trade-off by applying a domain specific formulation of MTL. To apply MTL effectively, CellScope proposed a novel PCA inspired similarity metric that groups data from geographically nearby base stations sharing performance commonalities. Finally, it also incorporates a hybrid online-offline model for efficient model updates. Our evaluations show significant benefits. We have also used CellScope to analyze a live LTE network, where it could offer significant reduction in troubleshooting efforts. We then explored the generality of our techniques by applying them to a new domain, energy anomaly diagnosis in smartphones. We show that extending our grouping and learning techniques to a new domain is easy and effective. Thus we believe our proposals form a solid framework for mitigating the effects of latency-accuracy trade-off in real-time dynamic connected data analytics systems.

Chapter 8

Conclusions & Future Work

Over the last several chapters, we have discussed the design and implementation of systems for dynamic connected data analysis, focusing on different aspects of the problem such as compact storage, efficient execution and low latency decisions. However, several pieces remain in achieving the vision of building the next generation computing infrastructure for *real-time* decisions on large-scale *dynamic, connected* data. In this chapter, we touch upon a few directions for future work in this space.

8.1 Future Work

Learning Based Connected Data Processing

Here, we look at leveraging learning to improve connected data processing. Possible future work in this direction include:

Pluggable Learned Sparsifiers: In chapter 5, we presented our work on GAP, an effort to bring approximation to graph processing. The effectiveness of GAP depends on the sparsifier used, and we discussed simple techniques to create sparsifiers. An interesting question worth exploring is if the sparsifier could be learned, perhaps using techniques such as deep learning and/or by leveraging the rich graph theory literature, such as [107], which presents a theoretical analysis of input reduction to some popular graph algorithms. The sparsifier can then be made *pluggable*; that is we could learn not one, but many sparsifiers. Based on the characteristics of the choices, it may be possible to cherry pick one or an ensemble of them to meet the latency/error bound.

Program Synthesis for Approximation: An alternative approach to applying sparsification to the graph is to use an approximate version of the algorithm. In graph theory

literature, there are numerous proposals on approximate versions of algorithms for a specific analysis problem. However, using them directly is problematic since it introduces dependency on the algorithm and hence impedes generality. Instead, a radical question is if it is possible to automatically *synthesize* an approximate version of an analysis given the exact version of it.

Incremental Approximate Analysis: In both the approximate techniques presented in this dissertation, the system re-executes the queries when the underlying dataset changes. This is acceptable in a majority of cases because the approximation is fast enough to support interactivity. However, when real-time analysis is required on complex analysis, rerunning the analysis from scratch may not be feasible. Here, it may be worth looking at incremental approximate analysis. The interesting question here is to understand how the approximation error accumulates over time, and how to impose bounds on it.

Learning On Connected Data

This direction looks at how to improve learning techniques on connected data, and the possible future work may consist of:

Scalable Graph Convolutions for Deep Learning: Graph convolutions have emerged as a popular technique in combining deep learning and structured approaches, and *graph networks* has recently been touted as the fundamental building blocks for next generation AI [27]. In these approaches, the convolution of the graph is often the bottleneck for real-time applications. The main reason for this bottleneck is the use of a message passing based approach for processing. Based on the usefulness of approximation, it may be possible to take a non-traditional approach of eschewing message passing and favoring approximate graph isomorphism based techniques for faster and better convolutions.

Approximation as a First Class Citizen: Many AI techniques, such as reinforcement learning (RL), depend on complex distributed algorithms and parallel processing to learn policies by performing huge number of compute intensive simulations. A unique characteristic of learning techniques is that they can accommodate approximation to some extent without affecting the performance of the model. This provides us with an opportunity to leverage approximation as a first class citizen in building systems for learning on connected data in an effort to enable real-time AI, for instance, it may make real-time policy updates possible.

New Scenarios

Now we discuss work which could potentially improve connected data analysis systems by leveraging new settings/scenarios.

Error Code Based State Store: As data volumes keep increasing, future connected data analysis systems would need to handle magnitudes more data in the future compared to what we presented in this dissertation. A major bottleneck in connected data analysis is the way data is partitioned across multiple machines. Several factors affect how data could be partitioned, and in many cases, the data layout and query distribution results in heavy skews and some machines becoming hotspots. The use of error correcting codes to reduce this performance degradation is an idea worth considering.

Leveraging Other/New Storage Hardware: The systems in this dissertation assumes that the entire data under analysis is stored in main memory. This is a reasonable assumption today, but with the rising data volume, and new analyses (e.g., ones that generate significantly more intermediate data than input, such as pattern mining) it is worthwhile pursuing alternative storage layers. There are two possible paths that could be explored here: (i) looking at new ways to represent data in traditional disk based storage (spinning disks or SSDs) and possibly leveraging the (little) computation capability in them, or (ii) leverage new hardware, such as 3D XPoint to aid connected data analysis.

Simultaneous Queries and Analysis: This dissertation looked at *analysis* on connected data, but an equally important area is to support *point queries* where every query looks only at a very small amount of data but the number of queries are large. However, supporting queries is fundamentally at odds with analysis because they require completely different data layouts. The systems we presented are tuned for working on large sets of data (e.g., the computation model executes the same program on every node), and thus are not suited for point querying. Work in this direction would need to look at new data layouts and computation engines that can support both large scale and fine-grained tasks. Can we develop data structures that can be transformed into each other easily?

Universal Sketching for Connected Data Queries: Building on the previous context, a possible approach to support faster querying might be to use sketches, or compact digests of dataset which can be used to answer the queries. However, traditional sketching techniques are specific to the query; that is a given sketch can only answer a particular query. There is an opportunity to generate universal sketching mechanisms targeted at connected data which can answer several kinds of queries.

Incorporating the *Edge*: The type of connected data analysis we presented in this dissertation resorted to aggregating data from the sources to one (or more) data centers in the cloud and processing them. However, as we move towards real-time applications, such aggregations may not be possible. Thus, we believe that the next generation real-time connected data processing systems will span *edges* and the cloud. Including the edge in the next generation system architecture can provide vast benefits. For instance, delegating model learning and inference between the cloud and the edge can result in significant latency and cost reduction. However, several open questions need to be answered before this becomes a reality: how should the work be partitioned? How much data should each edge send? How can it do so in the face of constrained bandwidth? Can we leverage sampling to reduce bandwidth requirements? How would accuracy be affected?

Serverless Connected Data Processing: The systems presented in this dissertation, similar to a vast majority of data processing systems today, assume that they are deployed on a dedicated cluster. Recently, serverless computing architectures have gained attention for cloud workloads. The significant ease of deployment, management and the highly elastic nature of serverless architecture makes it a favorable candidate for real-time connected data processing systems as it can easily handle the dynamic nature of both the data volume and computation requirement over time. However existing serverless solutions are not well suited for iterative and communication heavy workloads, both of which are common traits in real-time learning tasks in connected data. It might be possible to leverage approximation and intelligent data structures to achieve highly scalable and elastic connected data analytics and machine learning engines that can be deployed on serverless offerings.

8.2 Closing Thoughts

In this dissertation, we designed and developed scalable systems for connected data processing by proposing new abstractions for operating on such data, compact datastructures to store data and state, efficient computation models that reduce redundant work using incremental techniques, techniques that significantly improve performance by embracing approximation and methods for accurate applications of machine learning that mitigating fundamental trade-offs. With the IoT fast becoming a reality, and edges incorporating advanced actuation capabilities, we are entering an exciting era in *real-time* connected data processing. We believe that there are tremendous opportunities for cross-disciplinary research in the areas of distributed systems, databases, machine learning and mobile computing. We hope that the systems and techniques we presented here can potentially influence the next generation of data intensive systems.

Bibliography

[1] 3gpp. *Self-Organizing Networks SON Policy Network Resource Model (NRM) Integration Reference Point (IRP)*. http://www.3gpp.org/ftp/Specs/archive/32_series/32.521/.

[2] *A comparison of state-of-the-art graph processing systems*. https://code.facebook.com/posts/319004238457019/a-comparison-of-state-of-the-art-graph-processing-systems/. 2016 (accessed January 2019).

[3] Martin Abadi et al. "TensorFlow: A System for Large-scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283.

[4] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. "Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems". In: *Proceedings of SIGMOD '14*. Snowbird, Utah, USA: ACM, pp. 481–492.

[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 29–42.

[6] Bhavish Aggarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. "NetPrints: diagnosing home network misconfigurations using shared knowledge". In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 349–364.

[7] Charu C. Aggarwal and Haixun Wang, eds. *Managing and Mining Graph Data*. Vol. 40. Advances in Database Systems. Springer, 2010.

[8] Nesreen K. Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. "Graph Sample and Hold: A Framework for Big-graph Analytics". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. New York, New York, USA: ACM, 2014, pp. 1446–1455.

[9] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. "Analyzing Graph Structure via Linear Measurements". In: *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '12. Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, pp. 459–467.

[10] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. "Graph Sketches: Sparsification, Spanners, and Subgraphs". In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 5–14.

[11] 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: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 125–137.

[12] 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: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 125–137.

[13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1033–1044.

[14] Mohammad Al Hasan and Mohammed J. Zaki. "Output Space Sampling for Graph Patterns". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 730–741.

[15] Alcatel Lucent. *9900 Wireless Network Guardian*. `http://www.alcatel-lucent.com/products/9900-wireless-network-guardian`. 2013.

[16] Alcatel Lucent. *9959 Network Performance Optimizer*. `http://www.alcatel-lucent.com/products/9959-network-performance-optimizer`. 2014.

[17] Alcatel Lucent. *Alcatel-Lucent Motive Big Network Analytics for service creation*. `http://resources.alcatel-lucent.com/?cid=170795`. 2014.

[18] Alcatel Lucent. *Motive Big Network Analytics*. `http://www.alcatel-lucent.com/solutions/motive-big-network-analytics`. 2014.

[19] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482.

[20] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. "GRASS: Trimming Stragglers in Approximation Analytics." In: *NSDI*. 2014, pp. 289–302.

[21] Apache Flink. `https://flink.apache.org`.

[22] Apache Giraph. `http://giraph.apache.org`.

[23] Sepehr Assadi, Sanjeev Khanna, and Yang Li. "On Estimating Maximum Matching Size in Graph Streams". In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '17. Barcelona, Spain: Society for Industrial and Applied Mathematics, 2017, pp. 1723–1742.

[24] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. "Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: ACM, 2007, pp. 13–24.

[25] Athula Balachandran, Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Srinivasan Seshan, Shobha Venkataraman, and He Yan. "Modeling Web Quality-of-experience on Cellular Networks". In: *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*. MobiCom '14. Maui, Hawaii, USA: ACM, 2014, pp. 213–224.

[26] Raquel Barco, Volker Wille, Luis Díez, and Matías Toril. "Learning of Model Parameters for Fault Diagnosis in Wireless Networks". In: *Wirel. Netw.* 16.1 (Jan. 2010), pp. 255–271.

[27] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. "Relational inductive biases, deep learning, and graph networks". In: *arXiv preprint arXiv:1806.01261* (2018).

[28] Jonathan Baxter. "A Model of Inductive Bias Learning". In: *J. Artif. Int. Res.* 12.1 (Mar. 2000), pp. 149–198.

[29] Scott Beamer, Krste Asanovic, and David Patterson. "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server". In: *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*. IISWC '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 56–65.

[30] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. "Efficiently Updating Materialized Views". In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. SIGMOD '86. Washington, D.C., USA: ACM, 1986, pp. 61–71.

[31] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[32] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th international conference on World Wide Web*. Ed. by Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar. ACM Press, 2011, pp. 587–596.

[33] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[34] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[35] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: ACM, 2012, pp. 13–16.

[36] Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. "How Hard Is Counting Triangles in the Streaming Model?" In: *Automata, Languages, and Programming: 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*. Ed. by Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 244–254.

[37] Coen Bron and Joep Kerbosch. "Algorithm 457: finding all cliques of an undirected graph". In: *Communications of the ACM* 16.9 (1973), pp. 575–577.

[38] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 49–60.

[39] Aydin Buluç and John R. Gilbert. "The Combinatorial BLAS: design, implementation, and applications". In: *IJHPCA* 25.4 (2011), pp. 496–509.

[40] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. "Counting Triangles in Data Streams". In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '06. Chicago, IL, USA: ACM, 2006, pp. 253–262.

[41] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. "Facilitating Real-time Graph Mining". In: *Proceedings of the Fourth International Workshop on Cloud Data Management*. CloudDB '12. Maui, Hawaii, USA: ACM, 2012, pp. 1–8.

[42] Richard Caruana. "Multitask Learning: A Knowledge-Based Source of Inductive Bias". In: *Proceedings of the Tenth International Conference on Machine Learning*. Morgan Kaufmann, 1993, pp. 41–48.

[43] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. "Optimized Stratified Sampling for Approximate Query Processing". In: *ACM Trans. Database Syst.* 32.2 (June 2007).

[44] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. "PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 1:1–1:15.

[45] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. "Kineograph: Taking the Pulse of a Fast-changing and Connected World". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 85–98.

[46] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. "One Trillion Edges: Graph Processing at Facebook-scale". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1804–1815.

[47] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 16–16.

[48] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. "Online Aggregation and Continuous Query Support in MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 1115–1118.

[49] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". In: *Foundations and Trends in Databases* 4.1-3 (2012), pp. 1–294.

[50] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. "Gigascope: a stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 647–651.

[51] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. "Estimating PageRank on Graph Streams". In: *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '08. Vancouver, Canada: ACM, 2008, pp. 69–78.

[52] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10.

[53] Thomas G Dietterich. "Ensemble methods in machine learning". In: *Multiple classifier systems*. Springer, 2000, pp. 1–15.

[54] *Differential Dataflow Rust Implementation*. `https://github.com/frankmcsherry/differential-dataflow`.

[55] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. "Making data structures persistent". In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. ACM. 1986, pp. 109–121.

[56] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. "GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph". In: *Proc. VLDB Endow.* (2014).

[57] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. "GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph". In: *Proc. VLDB Endow.* 7.7 (Mar. 2014), pp. 517–528.

[58] Ericsson. *Ericsson RAN Analyzer*. `http://www.ericsson.com/ourportfolio/products/ran-analyzer`. 2014.

[59] Ericsson. *Ericsson RAN Analyzer Overview*. `http://www.optxview.com/Optimi_Ericsson/RANAnalyser.pdf`. 2012.

[60] Theodoros Evgeniou and Massimiliano Pontil. "Regularized Multi–task Learning". In: *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '04. Seattle, WA, USA: ACM, 2004, pp. 109–117.

[61] Wenfei Fan, Chunming Hu, and Chao Tian. "Incremental Graph Computations: Doable and Undoable". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 155–169.

[62] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. "Parallelizing Sequential Graph Computations". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 495–510.

[63] Santo Fortunato. "Community detection in graphs". In: *Physics reports* 486.3 (2010), pp. 75–174.

[64] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[65] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. "A Survey on Concept Drift Adaptation". In: *ACM Comput. Surv.* 46.4 (Mar. 2014), 44:1–44:37.

[66] P. Gao, M. Zhang, K. Chen, Y. Wu, and W. Zheng. "High Performance Graph Processing with Locality Oriented Design". In: *IEEE Transactions on Computers* 66.7 (July 2017), pp. 1261–1267.

[67] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. "ApproxHadoop: Bringing Approximations to MapReduce Frameworks". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015, pp. 383–397.

[68] Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. "Evolution of Social-attribute Networks: Measurements, Modeling, and Implications Using Google+". In: *Proceedings of the 2012 Internet Measurement Conference*. IMC '12. Boston, Massachusetts, USA: ACM, 2012, pp. 131–144.

[69] Pinghua Gong, Jieping Ye, and Changshui Zhang. "Robust Multi-task Feature Learning". In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: ACM, 2012, pp. 895–903.

[70] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 17–30.

[71] Joseph Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014.

[72] *Graph Data Mining with Arabesque*. `http://arabesque.io`.

[73] *Graph500 Benchmarks*. `http://www.graph500.org`.

[74] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. "Making Pull-based Graph Processing Performant". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. Vienna, Austria: ACM, 2018, pp. 246–260.

[75] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. "Maintaining Views Incrementally". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93. Washington, D.C., USA: ACM, 1993, pp. 157–166.

[76] 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: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 1:1–1:14.

[77] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 71–85.

[78] Paul Harris, Chris Brunsdon, and Martin Charlton. "Geographically weighted principal components analysis". In: *International Journal of Geographical Information Science* 25.10 (2011), pp. 1717–1736.

[79] Chi-Yao Hong, Matthew Caesar, Nick Duffield, and Jia Wang. "Tiresias: Online Anomaly Detection for Hierarchical Operational Network Data". In: *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*. ICDCS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 173–182.

[80] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 629–647.

[81] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. "An In-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 363–374.

[82] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. "Scheduling Jobs Across Geo-distributed Datacenters". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 111–124.

[83] *Intel CEO on the Future of Automated Driving*. `https://newsroom.intel.com/editorials/krzanich-the-future-of-automated-driving/`.

[84] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72.

[85] Anand Padmanabha Iyer, Li Erran Li, Mosharaf Chowdhury, and Ion Stoica. "Mitigating the Latency-Accuracy Trade-off in Mobile Data Analytics Systems". In: *ACM MobiCom*. 2018.

[86] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. "Time-evolving Graph Processing at Scale". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. GRADES '16. Redwood Shores, California: ACM, 2016, 5:1–5:6.

[87] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. "Automating Diagnosis of Cellular Radio Access Network Problems". In: *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. MobiCom '17. Snowbird, Utah, USA: ACM, 2017, pp. 79–87.

[88] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. "ASAP: Fast, Approximate Graph Pattern Mining at Scale". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 745–761.

[89] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. "Towards Fast and Scalable Graph Pattern Mining". In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, July 2018.

[90] Anand Padmanabha Iyer, Aurojit Panda, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. "Monarch: Gaining Command on Geo-Distributed Graph Analytics". In: *USENIX HotCloud*. 2018.

[91] Anand Iyer, Li Erran Li, and Ion Stoica. "CellIQ : Real-Time Cellular Network Analytics at Scale". In: *Proceedings of the 12th USENIX conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015.

[92] Anand Iyer, Li Erran Li, and Ion Stoica. "CellIQ : Real-Time Cellular Network Analytics at Scale". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 309–322.

[93] Madhav Jha, C. Seshadhri, and Ali Pinar. "A Space-Efficient Streaming Algorithm for Estimating Transitivity and Triangle Counts Using the Birthday Paradox". In: *ACM Trans. Knowl. Discov. Data* 9.3 (Feb. 2015), 15:1–15:21.

[94] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. "Tackling Bufferbloat in 3G/4G Networks". In: *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*. IMC '12. Boston, Massachusetts, USA: ACM, 2012, pp. 329–342.

[95] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. "Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 523–536.

[96] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. "Detailed diagnosis in enterprise networks". In: *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. SIGCOMM '09. Barcelona, Spain: ACM, 2009, pp. 243–254.

[97] Gunjan Khanna, Mike Yu Cheng, Padma Varadharajan, Saurabh Bagchi, Miguel P. Correia, and Paulo J. Veríssimo. "Automated Rule-Based Diagnosis Through a Distributed Monitor System". In: *IEEE Trans. Dependable Secur. Comput.* 4.4 (Oct. 2007), pp. 266–279.

[98] U. Khurana and A. Deshpande. "Efficient snapshot retrieval over historical graph data". In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. Apr. 2013, pp. 997–1008.

[99] Udayan Khurana and Amol Deshpande. "Storing and Analyzing Historical Graph Data at Scale". In: *CoRR* abs/1509.08960 (2015).

[100] Seyoung Kim and Eric P. Xing. "Tree-Guided Group Lasso for Multi-Task Regression with Structured Sparsity". In: *Intenational Conference on Machine Learning (ICML)* (2010).

[101] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[102] Tim Kraska, Ameet Talwalkar, John C. Duchi, Rean Griffith, Michael J. Franklin, and Michael I. Jordan. "MLbase: A Distributed Machine-learning System". In: *CIDR*. 2013.

[103] WJ Krzanowski. "Between-groups comparison of principal components". In: *Journal of the American Statistical Association* 74.367 (1979), pp. 703–707.

[104] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a social network or a news media?" In: *WWW '10: Proceedings of the 19th international conference on World wide web*. Raleigh, North Carolina, USA: ACM, 2010, pp. 591–600.

[105] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-Scale Graph Computation on Just a PC". In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 31–46.

[106] Anukool Lakhina, Mark Crovella, and Christophe Diot. "Diagnosing Network-wide Traffic Anomalies". In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '04. Portland, Oregon, USA: ACM, 2004, pp. 219–230.

[107] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. "Filtering: A Method for Solving Graph Problems in MapReduce". In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 85–94.

[108] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases". In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 38–49.

[109] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[110] Yan Liu, Jing Zhang, M. Jiang, D. Raymer, and J. Strassner. "A model-based approach to adding autonomic capabilities to network fault management system". In: *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*. Apr. 2008, pp. 859–862.

[111] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. "GraphLab: A New Framework For Parallel Machine Learning." In: *UAI*. Ed. by Peter Grünwald and Peter Spirtes. AUAI Press, 2010, pp. 340–349.

[112] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. "Mosaic: Processing a Trillion-Edge Graph on a Single Machine". In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 2017, pp. 527–543.

[113] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. "Mosaic: Processing a Trillion-Edge Graph on a Single Machine". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: ACM, 2017, pp. 527–543.

[114] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. "LLAMA: Efficient graph analytics using Large Multiversioned Arrays". In: *2015 IEEE 31st International Conference on Data Engineering*. Apr. 2015, pp. 363–374.

[115] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146.

[116] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. "Everything you always wanted to know about multicore graph processing but were afraid to ask". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 631–643.

[117] Mugilan Mariappan and Keval Vora. "GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: ACM, 2019, 25:1–25:16.

[118] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! But at what COST?" In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[119] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. "Differential Dataflow". In: *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. 2013.

[120] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. "ImmortalGraph: A System for Storage and Analysis of Temporal Graphs". In: *Trans. Storage* 11.3 (July 2015), 14:1–14:34.

[121] Microsoft Naiad Team. *GraphLINQ: A graph library for Naiad*. `http://bigdataatsvc.wordpress.com/2014/05/08/graphlinq-a-graph-library-for-naiad/`. 2014.

[122] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. "Network motifs: simple building blocks of complex networks". In: *Science* 298.5594 (2002), pp. 824–827.

[123] Jayanta Mondal and Amol Deshpande. "EAGr: Supporting Continuous Ego-centric Aggregate Queries over Large Dynamic Graphs". In: *CoRR* abs/1404.6570 (2014).

[124] Jayanta Mondal and Amol Deshpande. "Stream Querying and Reasoning on Social Data". In: *Encyclopedia of Social Network Analysis and Mining*. 2014, pp. 2063–2075.

[125] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455.

[126] *Neo4J*. http://www.neo4j.com. (accessed March 2019).

[127] *NetworkX*. http://networkx.github.io/.

[128] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. "Carat: Collaborative Energy Diagnosis for Mobile Devices". In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. SenSys '13. Roma, Italy: ACM, 2013, 10:1–10:14.

[129] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. "Time-evolving graph processing at scale". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. ACM. 2016, p. 5.

[130] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. "Bridging the GAP: Towards Approximate Graph Analytics". In: *GRADES-NDA*. Houston, Texas, 2018.

[131] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. "Tegra: Efficient Ad-Hoc Analytics on Time-evolving Graphs (Under Submission)". In: 2018.

[132] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999.

[133] Rasmus Pagh and Charalampos E. Tsourakakis. "Colorful Triangle Counting and a MapReduce Implementation". In: *CoRR* abs/1103.6073 (2011).

[134] Yan Pan, Rongkai Xia, Jian Yin, and Ning Liu. "A Divide-and-Conquer Method for Scalable Robust Multitask Learning". In: *Neural Networks and Learning Systems, IEEE Transactions on* 26.12 (Dec. 2015), pp. 3163–3175.

[135] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. "Counting and Sampling Triangles from a Graph Stream". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1870–1881.

[136] K. Pearson. "On lines and planes of closest fit to systems of points in space". In: *Philosophical Magazine* 2.6 (1901), pp. 559–572.

[137] *Persistent Adaptive Radix Tree*. https://github.com/ankurdave/part.

[138] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. "Managing Large Graphs on Multi-Cores with Graph Awareness". In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 41–52.

[139] N. Pržulj, D. G. Corneil, and I. Jurisica. "Modeling Interactome: Scale-free or Geometric?" In: *Bioinformatics* 20.18 (Dec. 2004), pp. 3508–3515.

[140] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. "Low Latency Geo-distributed Data Analytics". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 421–434.

[141] *PyTorch*. `https://pytorch.org/`.

[142] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. "Profiling Resource Usage for Mobile Applications: A Cross-layer Approach". In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys '11. Bethesda, Maryland, USA: ACM, 2011, pp. 321–334.

[143] Abdul Quamar, Amol Deshpande, and Jimmy Lin. "NScale: Neighborhood-centric Large-scale Graph Analytics in the Cloud". In: *The VLDB Journal* 25.2 (Apr. 2016), pp. 125–150.

[144] *R*. `https://www.r-projec.org/`.

[145] Sudarshan Rao. "Operational Fault Detection in Cellular Wireless Base-stations". In: *IEEE Trans. on Netw. and Serv. Manag.* 3.2 (Apr. 2006), pp. 1–11.

[146] Pedro Ribeiro and Fernando Silva. "G-Tries: A Data Structure for Storing and Finding Subgraphs". In: *Data Min. Knowl. Discov.* 28.2 (Mar. 2014), pp. 337–377.

[147] *RISELab Sponsors*. `https://rise.cs.berkeley.edu/sponsors/`. 2018.

[148] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

[149] Sanae Rosen, Haokun Luo, Qi Alfred Chen, Z. Morley Mao, Jie Hui, Aaron Drake, and Kevin Lau. "Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks". In: *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*. MobiCom '14. Maui, Hawaii, USA: ACM, 2014, pp. 177–188.

[150] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. "Chaos: Scale-out Graph Processing from Secondary Storage". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 410–424.

[151] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. "X-Stream: Edge-centric Graph Processing Using Streaming Partitions". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 472–488.

[152] Ahmed M Safwat and Hussein Mouftah. "4G network technologies for mobile telecommunications". In: *Network, IEEE* 19.5 (2005), pp. 3–4.

[153] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing". In: *Proc. VLDB Endow.* 11.4 (Dec. 2017), pp. 420–431.

[154] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. "Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 979–990.

[155] *Scikit-Learn*. `https://scikit-learn.org/`.

[156] Stefania Sesia, Issam Toufik, and Matthew Baker. *LTE: the UMTS long term evolution*. Wiley Online Library, 2009.

[157] Muhammad Zubair Shafiq, Jeffrey Erman, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. "Understanding the Impact of Network Dynamics on Mobile Video User Engagement". In: *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '14. Austin, Texas, USA: ACM, 2014, pp. 367–379.

[158] Muhammad Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, Shobha Venkataraman, and Jia Wang. "A First Look at Cellular Network Performance During Crowded Events". In: *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '13. Pittsburgh, PA, USA: ACM, 2013, pp. 17–28.

[159] Shai Shalev-Shwartz and Ambuj Tewari. "Stochastic Methods for L1-regularized Loss Minimization". In: *J. Mach. Learn. Res.* 12 (July 2011), pp. 1865–1892.

[160] Clint Smith. *3G wireless networks*. McGraw-Hill, Inc., 2006.

[161] Craig AN Soules, Garth R Goodson, John D Strunk, and Gregory R Ganger. "Metadata Efficiency in a Comprehensive Versioning File System". In: (2002).

[162] Evan R. Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph E. Gonzalez, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. "MLI: An API for Distributed Machine Learning". In: *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*. Ed. by Hui Xiong, George Karypis, Bhavani M. Thuraisingham, Diane J. Cook, and Xindong Wu. IEEE Computer Society, 2013, pp. 1187–1192.

[163] Daniel A. Spielman and Shang-Hua Teng. "Spectral Sparsification of Graphs". In: *CoRR* abs/0808.4134 (2008).

[164] *Stanford Large Network Dataset Collection*. `https://snap.stanford.edu/`.

[165] Technical Specification Group. *3GPP Specifications*. `http://www.3gpp.org/specifications`.

[166] 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: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 425–440.

[167] Nawanol Theera-Ampornpunt, Saurabh Bagchi, Kaustubh R. Joshi, and Rajesh K. Panta. "Using Big Data for More Dependability: A Cellular Network Tale". In: *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*. HotDep '13. Farmington, Pennsylvania: ACM, 2013, 2:1–2:5.

[168] Sebastian Thrun. "Is Learning The n-th Thing Any Easier Than Learning The First?" In: *Advances in Neural Information Processing Systems*. The MIT Press, 1996, pp. 640–646.

[169] Robert Tibshirani. "Regression Shrinkage and Selection Via the Lasso". In: *Journal of the Royal Statistical Society, Series B* 58 (1994), pp. 267–288.

[170] *Titan Distributed Graph Database*. `http://thinkaurelius.github.io/titan/`.

[171] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. "DOULION: Counting Triangles in Massive Graphs with a Coin". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '09. Paris, France: ACM, 2009, pp. 837–846.

[172] Guan-Hua Tu, Yuanjie Li, Chunyi Peng, Chi-Yu Li, Hongyi Wang, and Songwu Lu. "Control-plane Protocol Interactions in Cellular Networks". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 223–234.

[173] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 363–378.

[174] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. "CLARINET: WAN-Aware Optimization for Analytics Queries". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 435–450.

[175] Keval Vora, Rajiv Gupta, and Guoqing Xu. "KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 237–251.

[176] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. "WANalytics: Geo-Distributed Analytics for a Data Intensive World". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1087–1092.

[177] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. "Asynchronous Large-Scale Graph Processing Made Easy." In: *CIDR*. www.cidrdb.org, 2013.

[178] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. "Asynchronous Large-Scale Graph Processing Made Easy." In: *CIDR*. 2013.

[179] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. "Automatic Misconfiguration Troubleshooting with Peerpressure". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 17–17.

[180] *Weaver: A scalable, fast, consistent graph store*. `http://weaver.systems`.

[181] Ming Wu and Rong Jin. "A Graph-based Framework for Relation Propagation and Its Application to Multi-label Learning". In: *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '06. Seattle, Washington, USA: ACM, 2006, pp. 717–718.

[182] 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: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 408–421.

[183] 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: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 408–421.

[184] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. "Tux²: Distributed Graph Computation for Machine Learning". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 669–682.

[185] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. "Fast Iterative Graph Computation with Block Updates". In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 2014–2025.

[186] He Yan, A. Flavel, Zihui Ge, A. Gerber, D. Massey, C. Papadopoulos, H. Shah, and J. Yates. "Argus: End-to-end service anomaly detection and localization from an ISP's point of view". In: *INFOCOM, 2012 Proceedings IEEE*. 2012, pp. 2756–2760.

[187] Xifeng Yan and Jiawei Han. "gspan: Graph-based substructure pattern mining". In: *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*. IEEE. 2002, pp. 721–724.

[188] Jaewon Yang and Jure Leskovec. "Defining and Evaluating Network Communities based on Ground-truth". In: *CoRR* abs/1205.6233 (2012).

[189] Kiyoung Yang and Cyrus Shahabi. "A PCA-based Similarity Measure for Multivariate Time Series". In: *Proceedings of the 2nd ACM International Workshop on Multimedia Databases*. MMDB '04. Washington, DC, USA: ACM, 2004, pp. 65–74.

[190] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 1–14.

[191] Haibo Chen Yunhao Zhang Rong Chen. "Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017.

[192] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 2–2.

[193] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[194] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438.

[195] Ce Zhang, Arun Kumar, and Christopher Ré. "Materialization Optimizations for Feature Selection Workloads". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 265–276.

[196] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 544–557.

[197] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. "Exploring the Hidden Dimension in Graph Processing". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 285–300.

[198] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. "Exploring the Hidden Dimension in Graph Processing". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 285–300.

[199] 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: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: ACM, 2018, pp. 608–621.

[200] 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: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: ACM, 2018, pp. 608–621.

[201] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. "Making caches work for graph analytics". In: *2017 IEEE International Conference on Big Data (Big Data)*. Dec. 2017, pp. 293–302.

[202] Alice X. Zheng, Jim Lloyd, and Eric Brewer. "Failure Diagnosis Using Decision Trees". In: *Proceedings of the First International Conference on Autonomic Computing*. ICAC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 36–43.

[203] Xiaojin Zhu and Zoubin Ghahramani. "Learning from labeled and unlabeled data with label propagation". In: (2002).

[204] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. "Gemini: A Computation-Centric Distributed Graph Processing System". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 301–316.