Sketching as a Tool for Efficient Networked Systems

by

Zaoxing Liu

A dissertation submitted to The Johns Hopkins University

in conformity with the requirements for the degree of

Doctor of Philosophy

Baltimore, Maryland

October, 2018

Abstract

Today, computer systems need to cope with the explosive growth of data in the world. For instance, in data-center networks, monitoring systems are used to measure traffic statistics at high speed; and in financial technology companies, distributed processing systems are deployed to support graph analytics. To fulfill the requirements of handling such large datasets, we build efficient networked systems in a distributed manner most of the time. Ideally, we expect the systems to meet service-level objectives (SLOs) using the least amount of resource. However, existing systems constructed with conventional in-memory algorithms face the following challenges: (1) excessive resource requirements (e.g., CPU, ASIC, and memory) with high cost; (2) infeasibility in a larger scale; (3) processing the data too slowly to meet the objectives.

To address these challenges, we propose sketching techniques as a tool to build more efficient networked systems. Sketching algorithms aim to process the data with one or several passes in an *online*, *streaming* fashion (e.g., a stream of network packets), and compute highly accurate results. With sketching, we only maintain a compact summary of the entire data and provide theoretical guarantees on error bounds.

This dissertation argues for a sketching based design for large-scale networked systems, and demonstrates the benefits in three application contexts:

(i) *Network monitoring*: we build generic monitoring frameworks that support a range of applications on both software and hardware with universal sketches.

(ii) *Graph pattern mining*: we develop a swift, approximate graph pattern miner that scales to very large graphs by leveraging graph sketching techniques.

(iii) *Halo finding in N-body simulations*: we design scalable halo finders on CPU and GPU by leveraging sketch-based heavy hitter algorithms.

Acknowledgments

First, I would like to thank my advisor Vladimir (Vova) Braverman for his guidance, endless patience in tolerating my immature ideas and views, and continuous encouragement in pursuing exciting research problems. He has taught me how to do research, from defining research problems to designing elegant solutions. I have been amazed by Vova's insights into the core of many research questions at times. His wisdom and humbleness inspired me over the past four years and will forever influence my life. Vova is very open-minded and gives me enough freedom to seek ideas that interest me most; and he is also very supportive on what I want to pursue, from attending various conferences/workshops to connecting me to the experts in the field.

Second, I would like to express my gratitude on the support from my committee members Vyas Sekar and Xin Jin. Vyas has always offered his time on discussing ideas with me and provided insightful thoughts toward the roots of the problems. I have been very fortunate working with him on the UnivMon project and several others. His guidance on how to formulate the problems and complete the system designs benefited me a lot over the years. Xin has always been energetic and enthusiastic in sharing his thoughts on the ASAP project and some others. It is with great joy that I got a chance to work with Xin and learn from him in various ways.

During the PhD journey, I have had the opportunity to interact or collaborate with several researchers and faculty who have mentored me in many ways — Yair Amir, Randal Burns, Tamas Budavari, Michael Dinitz, Gil Einziger, Roy Friedman, Ryan Huang, Gerard Lemson, Mark Neyrinck, Shivaram Venkataraman, Vinodchandran N. Variyam, Ion Stoica, and Alex Szalay. Their advice and suggestions benefited me in accomplishing specific projects and beyond. I'm also thankful to my graduate student collaborators — Zhihao Bai, Ran Ben-Basat, Nikita Ivkin, Anand Iyer, Srinivas Suresh Kumara, Antonis Manousis, Li Song, Tejasvam Singh, Greg Vorsanger, Lin Yang, and Zhishuai Zhang.

I would like to thank my GBO committee members: Tamas Budavari, Andrei Gritsan, Xin Li, and Scott Smith, who examined my pre-proposal towards the dissertation and gave valuable suggestions.

It has been a great pleasure of studying in the department, and I want to share my thank to the wonderful faculty and staff members here. I am grateful to our awesome administrative team — Zackary Burwell, Debbie Deford, Tonette McClamy, Cathy Thornton, and others, for your responsiveness and services.

I have enjoyed my time at Hopkins. In particular, I feel incredibly fortunate to meet, interact with several talented graduate students and postdocs: Zhihao Bai, James Browne, Renyuan Cheng, Kuan Cheng, Arka Rai Choudhuri, Shuya Chu, Jinqiu Deng, Venkata Gandikota, Cong Gao, Jiaqi Gao, Yifan Ge, Aarushi Goel, Yuge Gong, Yigong Hu, Kevin Huang, Yao Huang, Nikita Ivkin, Haoyuan Ji, Zhengzhong Jin, Qian Ke, Xingguo Li, Shuo Li, Kunal Lillaney, Chang Liu, Qing Liu, Chang Lou, Hongyuan Mei, Disa Mhembere, Yasamin Nazari, Hang Ou, Fabian Prada, Jalaj Upadhyay, Enayat Ullah, Yuefan Wang, Shiwei Weng, Xiang Xiang, Yanbo Xu, Xi Yang, Lin Yang, Mo Yu, Zhuolong Yu, Andong Zhan, Shi Zhang, Zeyu Zhang, Zehua Zhao, Tuo Zhao, Chao Zheng, Yu Zheng, Da Zheng, Hang Zhu, Zhuodun Zhu, and many others. Thanks to them all for ensuring that my time at Hopkins was smooth and delightful. Special thanks to Tuo Zhao and Yanbo Xu for helping me develop and learn, and gathering us together during the time at Hopkins.

I want to thank my parents for their all-around support. Thanks to them for believing in my capabilities more than I ever will. Their love and care encourage me to chase my dream without worries behind.

Finally, I want to express my deepest gratitude to my wife, Keke, for sharing her valuable time with me in the United States. We share our joy and sorrow, happiness and pain, together. I own a great deal to her for endless love, care, and support.

This dissertation is based upon work supported in part by DARPA-111477, NSF IIS-1447639, Research Trends Det. HR0011-16-P-0014, EAGER CCF-1650041, CAREER CCF-1652257, Cisco-90073352, and ONR.

To my parents, J.H. Liu & Z.L. Zhao.

To my wife, K.K. Wen.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last decade, we experienced an exponential growth of data in the world, and this trend will continue to 2020 and beyond [1]. The challenges of handling massive-scale datasets arise in domains such as data centers, enterprises, ISPs, and scientific research. Each domain requires large networked systems to handle tasks such as cloud services, network monitoring, security, or other computation-heavy tasks. For instance, the network operators of data centers may utilize monitoring systems to (1) effectively measure the network performance, (2) efficiently detect anomalies on the network, and (3) detect and filter malicious network traffic as much as possible. While in financial technology companies, large distributed systems are deployed to (1) compute graph analytics to detect outliers, (2) mine complex graph patterns to detect fraud transactions, and (3) conduct streaming processing to drop malicious transactions in an online fashion.

However, as the amounts of data *outburst* and patterns of workloads *evolve*, the systems built with traditional in-memory algorithms are unable to handle the fast growth of the data in one or more following aspects: (i) excessive hardware resource requirements, (ii) infeasibility in a larger scale, and (iii) significant slowness on processing the data. The reason is that the improvement of hardware cannot be linearly coped with the growth of datasets. To address these challenges, we propose to build efficient systems with sketching algorithms[1] when approximated results can be allowed. While there has been a large body of work on sketching techniques regarding the streaming processing model in the theory community, there is little work done on building systems with sketches in different applications. To put the work presented in this dissertation in perspective, we discuss the sketching techniques for building efficient systems in the contexts of networking monitoring, graph pattern mining, and halo finding in astrophysical N-body simulations. In this chapter, we discuss the current practice and the background on sketching algorithms before briefing our approaches and contributions.

1.1 Current Practice

1.1.1 Network Monitoring

Network management today requires accurate estimates of metrics for many applications including traffic engineering (e.g., heavy hitters), anomaly detection (e.g., entropy of source addresses), and security (e.g., DDoS detection). Obtaining accurate estimates given router CPU and memory constraints is a challenging problem. Existing approaches fall in one of two undesirable extremes: (1) low fidelity general-purpose approaches such as packet sampling,

[1]In this thesis, we refer to "sketching algorithms" and "sketches" interchangeably.

or (2) high fidelity but complex algorithms customized to specific application-level metrics.

Packet Sampling. Existing network monitoring tools in the industry depend on sampling flow measurements from routers (e.g., NetFlow [2] or sFlow [3]). They use these tools to sample packets by either packet-based (identified by flow keys), or volume-based (counted by byte counts). The core technique here is to aggregate uniformly sampled packets into some flow reports, and compute any metrics based on the flow reports. While the sampling based approaches are useful for coarse-grained metrics (e.g., total volume or estimated flow size distribution), they cannot offer good fidelity unless running at a very high sampling rate, which is undesirable due to computation and memory overhead.

Application-specific Algorithms. To address the drawbacks of packet sampling approach, researchers have proposed a number of application-specific sketches to handle specific measurement tasks. These sketching algorithms allow for memory-efficient monitoring systems as they reduce the memory usage of measurement tasks while maintaining guaranteed fidelity. There always be a trade-off between memory and accuracy backed by rigorous theoretical proofs. Examples of monitoring tasks that are supported by sketches include:

- **Heavy Hitter Detection** to identify flows that consume more than a threshold α of the total capacity. The capacity can be packet-based (flow keys) or volume-based (byte counts). Example custom algorithm include Count-Min Sketch [4], Space-saving [5], and Count Sketch [6].

- **Cardinality Estimation** to estimate the number of distinct flows in the traffic [7].
- **Change Detection** to identify flows that contribute more than a threshold of the total capacity change over two consecutive time intervals using reversible k-ary Sketch [8, 9].
- **Entropy Estimation** to measure the entropy value of a specific head field distribution (e.g., Lall et al [10]).
- **Attack Victim Detection** to identify a destination host that receives traffic from more than a threshold number of source hosts [11].

However, although application-specific approaches have good theoretical guarantees and practical performances, this architecture still has several drawbacks: (1) when handling multiple measurement tasks, there are large memory and CPU burdens; (2) No late-binding: need to commit the resources and the set of tasks before running the measurement.

1.1.2 Graph Pattern Mining

Mining patterns in a graph represent an important class of graph processing problems. 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* [12], 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 [13, 14] 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 [13], 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.

Current graph processing systems. A large number of systems have been proposed in the literature for graph processing [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]. Of these, some [15, 17, 18] 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). However, these systems do not support general pattern mining.

Current graph mining systems. 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 [13]. While it supports efficient pattern mining, the system still requires 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.

1.2 Background on Sketching Techniques

Streaming Model. A data stream $D = D(n, m)$ is an ordered sequence of objects $a_1, a_2, \ldots, a_n$, where $a_j = 1 \ldots m$. The elements of the stream can represent any digital objects: integers, real numbers of fixed precisions, network packets, edges of a graph, messages, images, web pages, etc. Data streaming model has emerged as a natural computational model for a number of application in big data processing. In this model, algorithms are allows to access a limited amount of memory and can process the dataset in one pass (or a few passes) but are guaranteed to produce sufficiently accurate answers for some objective statistics of the dataset. In particular, the strict memory limitation in this model captures various applications in processing large-scale datasets and makes the algorithms new scalable tools for networked systems. First, let's describe some fundamental problems in this model.

L_p **Norm and Frequency Moment.** In a stream D of objects $a_1, a_2, \ldots, a_n$, we define the frequency vector $F(m)$ as a vector of dimensionality m with non-negative entries $f_i, i \in [m]$ as

$$f_i = |\{j : 1 \leq j \leq n, a_j = i\}|$$

Then, the L_p norm of the frequency vector M is defined as $(\sum_{i=1}^{m}(f_i)^p)^{\frac{1}{p}}$, and the p-frequency moment is $\sum_{i=1}^{m}(f_i)^p$.

L_p **Heavy Hitter Problem.** We say that an element is "heavy" if it appears more times than a constant fraction of some L_p norm of the stream. We consider the following heavy hitter problem.

Problem 1 . Given a stream D of n elements, the ϵ-approximate (ϕ, L_p)-heavy hitter problem is to find a set of elements T:

- $\forall i \in [m], f_i > \phi L_p \implies i \in T.$
- $\forall i \in [m], f_i < (\phi - \epsilon) L_p \implies i \notin T.$

In networking, we usually consider the L_1 heavy hitter problem.

1.3 Thesis Approach and Contributions

In this thesis, we look into the core constructions behind several networked systems and try to focus on a more resource-efficient design in the application contexts of networking, graph processing, and astrophysics. Before achieving resource-efficiency, we want to understand the bottlenecks of resources in different application settings. For instance, in network monitoring, the resource bottleneck on a hardware switch is the memory (SRAM) to store intermediate data in an online fashion while it might not be the case in a software switch. In a software switch, we focus on reducing the per-packet processing overhead of the measurement module while still achieving high accuracy. For a detailed bottleneck analysis on software switches, please refer to Section 3.2. However, we still do not want to allocate too much memory on the software switch since we may want to store the most of the data structures on CPU cache, and save cache and memory for other concurrent network services. In graph pattern mining systems, a deterministic algorithm may generate intermediate subgraph candidates in an exponential-level in terms of the graph edge size, which brings infeasibility to handle large graph due to memory overflow

Settings	Cache	RAM	CPU or ASIC
Network Monitoring (Hardware)	$\downarrow$	$\downarrow$	$\times$
Network Monitoring (Software)	$\downarrow$	$\times$	$\times$
Graph Pattern Mining	$\times$	$\downarrow$	$\downarrow$
N-body Simulation	$\times$	$\downarrow$	$\downarrow$

Table 1.1: Summary of the efficiency optimization goals in different application settings.

or slow execution due to memory bandwidth limitation. Similarly in astrophysical N-body simulations, a deterministic algorithm needs to handle huge intermediate data.

Thus, we summarize the efficiency goals in Table 1.1 based on the different resource bottlenecks in the three application contexts. $\downarrow$ means the usage should be as small as possible and $\times$ implies not important. In these cases, by trading a small loss on the accuracy of the results, we can utilize sketching algorithms as the core construction to alleviate the resource bottlenecks and achieve significant efficiency.

1.3.1 A Robust Network Monitoring Infrastructure

UnivMon: Network management requires accurate estimates of metrics for many applications. Existing monitoring approaches fall in one of two undesirable extremes: (1) low fidelity general-purpose approaches such as sampling, or (2) high fidelity but complex algorithms customized to specific application-level metrics. Ideally, a solution should be both general (i.e., supports many applications) and provide accuracy comparable to custom algorithms.

In Chapter 2, we presents *UnivMon* [26], a framework for flow monitoring which leverages recent theoretical advances and demonstrates that it is possible to achieve both generality and high accuracy. UnivMon uses an application-agnostic data plane monitoring primitive; different (and possibly unforeseen) estimation algorithms run in the control plane, and use the statistics from the data plane to compute application-level metrics. We implement UnivMon using P4 and develop coordination techniques to provide a "one-big-switch" abstraction for network-wide monitoring. We evaluate the effectiveness of UnivMon using a range of trace-driven evaluations and show that it offers comparable (and sometimes better) accuracy relative to custom sketching solutions across a range of monitoring tasks.

NitroSketch: With increasing virtualization of services and network functions, virtual switches are emerging as an important measurement vantage point. Given the tight resource requirements, sketching algorithms are a promising alternative to traditional monitoring (e.g., sampling or full packet capture). However, sketching algorithms (e.g., Count-Min Sketch and UnivMon) are typically designed with memory-oriented optimization goals in theory and incur significant computational overhead in software. Unfortunately, existing efforts that try to address this performance issue have to make compromises on the worst-case theoretical guarantees, make strong assumptions about the traffic distributions, or only work for specific sketches.

Chapter 3 presents *NitroSketch*, a general and efficient software sketching framework that enables line-rate packet processing for a broad spectrum of sketching algorithms. NitroSketch has provable worst-case guarantees, without needing any distributional assumptions about the traffic. We do this by systematically identifying the fundamental performance bottlenecks of sketches and developing rigorous solutions to tackle these. We implement a NitroSketch prototype and integrate it with two popular software switching platforms: Open vSwitch-DPDK and FD.io-VPP. We evaluate NitroSketch on commodity servers and show that accuracy is guaranteed $> 95\%$ while attaining a $27\times$ speedup in sketching and a 45% reduction in CPU usage.

1.3.2 A Fast, Approximate Graph Patterning Framework

While there has been a tremendous interest in processing data that has an underlying graph structure, existing distributed graph processing systems take several minutes or even hours to mine simple patterns on graphs.

Chapter 4 presents *ASAP* [27], a fast, approximate computation engine for graph pattern mining. ASAP[2] 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. We have implemented ASAP on a general-purpose distributed dataflow platform and evaluated it extensively on several graph patterns. Our experimental results show that ASAP outperforms existing exact pattern mining solutions by up to $77\times$. Further, ASAP can scale to graphs with billions of edges without the need for large clusters.

[2]As co-leading authors, Anand Iyer and I are both using this in our dissertations with full knowledge and support of the other.

1.3.3 A Memory-efficient Halo Finder in N-body Simulations

Astrophysical *N*-body simulations are essential for studies of the large-scale distribution of matter and galaxies in the Universe. This analysis often involves finding clusters of particles and retrieving their properties. Detecting such "halos" among a very large set of particles is a computationally intensive problem, usually executed on the same super-computers that produced the simulations, requiring huge amounts of memory.

In Chapter 5, we present a novel connection between the *N*-body simulations and the sketching algorithms [28]. In particular, we investigate a link between halo finders and the problem of finding frequent items (heavy hitters) in a data stream, that should significantly reduce the computational resource requirements, especially the memory needs. Based on this connection, we build a new halo finder by running efficient heavy hitter algorithms as a black-box. We implement two representatives of the family of heavy hitter algorithms, the Count-Sketch algorithm (CS) and the Pick-and-Drop sampling (PD), and evaluate their accuracy and memory usage. Comparison with other halo-finding algorithms from [29] shows that our halo finder can locate the largest haloes using significantly smaller memory space and with comparable running time. This streaming approach makes it possible to run and analyze extremely large data sets from *N*-body simulations on a smaller machine, rather than on supercomputers. Our findings demonstrate the connection between the halo search problem and streaming algorithms as a promising initial direction for further research.

Chapter 2

UnivMon: Universal Flow Monitoring with Sketching

Network management is multi-faceted and encompasses a range of tasks including traffic engineering [30, 31], attack and anomaly detection [32], and forensic analysis [33]. Each such management task requires accurate and timely statistics on different application-level metrics of interest; e.g., the flow size distribution [34], heavy hitters [35], entropy measures [10, 36], or detecting changes in traffic patterns [9].

At a high level, there are two classes of techniques to estimate these metrics of interest. The first class of approaches relies on *generic flow monitoring*, typically with some form of packet sampling (e.g., NetFlow [2]). While generic flow monitoring is good for coarse-grained visibility, prior work has shown that it provides low accuracy for more fine-grained metrics [37, 38, 39]. These well-known limitations of sampling motivated an alternative class of techniques based on *sketching* or *streaming* algorithms. Here, custom online algorithms and data structures are designed for specific metrics of interest that can yield provable resource-accuracy tradeoffs (e.g., [39, 10, 40, 38, 41, 42, 43]).

While the body of work in data streaming and sketching has made significant contributions, we argue that this trajectory of crafting special-purpose algorithms is untenable in the long term. As the number of monitoring tasks grows, this entails significant investment in algorithm design and hardware support for new metrics of interest. While recent tools like OpenSketch [44] and SCREAM [45] provide libraries to reduce the implementation effort and offer efficient resource allocation, they do not address the fundamental need to design and operate new custom sketches for each task. Furthermore, at any given point in time the data plane resources have to be committed (a priori) to a specific set of metrics to monitor and will have fundamental blind spots for other metrics that are not currently being tracked.

Ideally, we want a monitoring framework that offers both *generality* by delaying the binding to specific applications of interest but at the same time provides the required *fidelity* for estimating these metrics. Achieving generality and high fidelity simultaneously has been an elusive goal both in theory [46] (Question 24) as well as in practice [47].

In this chapter, we present the *UnivMon* (short for Universal Monitoring) framework that can simultaneously achieve both generality and high fidelity across a broad spectrum of monitoring tasks [10, 39, 40, 48]. UnivMon builds on and extends recent theoretical advances in *universal streaming*, where a single universal sketch is shown to be provably accurate for estimating a large class of functions [49, 50, 51, 52, 53]. In essence, this generality can enable us to delay the binding of the data plane resources to specific monitoring tasks, while still providing accuracy that is comparable (if not better) than running custom sketches using similar resources.

While our previous paper suggested the promise of universal streaming [54], it fell short of answering several practical challenges, which we address in this chapter. First, we demonstrate a concrete switch-level realization of UnivMon using P4 [55], and discuss key implementation challenges in realizing UnivMon. Second, prior work only focused on a single switch running univmon for a specific feature (e.g., source addresses) of interest, whereas in practice network operators want a panoramic view across multiple features and across traffic belonging to multiple origin-destination pairs. To this end, we develop lightweight-yet-effective coordination techniques that enable UnivMon to effectively provide a "one big switch" abstraction for network-wide monitoring [56], while carefully balancing the monitoring load across network locations.

We evaluate UnivMon using a range of traces [57, 58] and operating regimes and compare it to state-of-art custom sketching solutions based on OpenSketch [44]. We find that for a single network element, UnivMon achieves comparable accuracy, with an observed error gap $\leq$ 3.6% and average error gap $\leq$ 1%. Furthermore, UnivMon outperforms OpenSketch in the case of a growing application portfolio. In a network-wide setting, our coordination techniques can reduce the memory consumption and communication with the control plane by up to three orders of magnitude.

Contributions and roadmap: In summary, this chapter presents the following contributions:

- A practical architecture which translates recent theoretical advances to serve as the basis for a general-yet-accurate monitoring framework (§2.2, §2.3).
- An effective network-wide monitoring approach that provides a one-big switch abstraction (§2.4).
- A viable implementation using emerging programmable switch architectures (§2.5).
- A trace-driven analysis which shows that UnivMon provides comparable accuracy and space requirements compared to custom sketches (§4.5).

We begin with background and related work in the next section. We highlight outstanding issues and conclude in §6.

2.1 Background and Related Work

Many network monitoring and management applications depend on sampled flow measurements from routers (e.g., NetFlow or sFlow). While these are useful for coarse-grained metrics (e.g., total volume) they do not provide good fidelity unless these are run at very high sampling rates, which is undesirable due to compute and memory overhead.

This inadequacy of packet sampling has inspired a large body of work in data streaming or sketching. This derives from a rich literature in the theory community on streaming algorithms starting with the seminal "AMS" paper

[59] and has since been an active area of research (e.g., [49, 6, 60, 61]). At the high level, the problem they address is as follows: Given an input sequence of items, the algorithm is allowed to make a single or constant number of passes over the data stream while using sub-linear (usually poly-logarithmic) space compared to the size of the data set and the size of the dictionary. The algorithm then provides an approximate estimate of the desired statistical property of the stream (e.g., mean, median, frequency moments). Streaming is a natural fit for network monitoring and has been applied to several tasks including heavy hitter detection [39], entropy estimation [10], change detection [40], among others.

A key limitation that has stymied the practical adoption of streaming approaches is that the algorithms and data structures are tightly coupled to the intended metric of interest. This forces vendors to invest time and effort in building specialized algorithms, data structures, and corresponding hardware without knowing if these will be useful for their customers. Given the limited resources available on network routers and business concerns, it is difficult to support a wide spectrum of monitoring tasks in the long term. Moreover, at any instant the data plane resources are committed beforehand to the application-level metrics and other metrics that may be required in the future (e.g., as administrators start some diagnostic tasks and require additional statistics) will fundamentally not be available.

The efforts closest in spirit to our UnivMon vision is the minimalist monitoring work of Sekar et al. [47] and OpenSketch by Yu et al., [44]. Sekar et al.

showed empirically that flow sampling and sample-and-hold [39] can provide comparable accuracy to sketching when equipped with similar resources. However, this work offers no analytical basis for this observation and does not provide guidelines on what metrics are amenable to this approach.

OpenSketch [44] addresses an orthogonal problem of making it easier to implement sketches. Here, the router is equipped with a library of predefined functions in hardware (e.g., hash-maps or count-min sketches [61]) and the controller can reprogram these as needed for different tasks. While OpenSketch reduces the implementation burden, it still faces key shortcomings. First, because the switches are programmed to monitor a specific set of metrics, there will be a fundamental lack of visibility into other metrics for which data plane resources have not been committed, even if the library of functions supports those tasks. Second, to monitor a portfolio of tasks, the data plane will need to run many concurrent sketch instances, which increases resource requirements.

In summary, prior work presents a fundamental dichotomy: generic approaches that offer poor fidelity and are hard to reason about analytically vs. sketch-based approaches that offer good guarantees but are practically intractable given the wide range of monitoring tasks of interest.

Our previous paper makes a case for a “RISC” approach for monitoring [54], highlighting the promise of recent theoretical advances in universal streaming [49, 50]. However, this prior work fails to address several key practical challenges. First, it does not discuss how these primitives can actually be

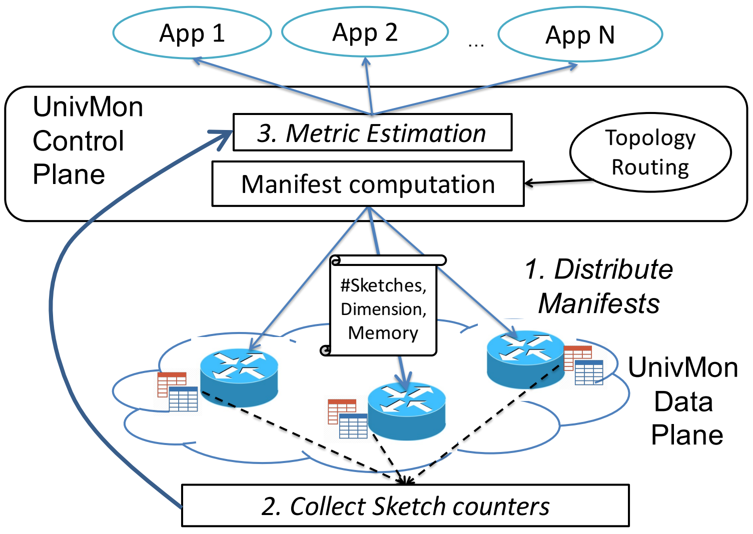


Figure 2.1: Overview of UnivMon: The data plane nodes perform the monitoring operations and report sketch summaries to the control plane which calculates application-specific metric estimates.

mapped into switch processing pipelines. In fact, we observe that the data-control plane split that they suggest is impractical to realize as they require expensive sorting/sifting primitives (see §2.5). Second, this prior work takes a narrow single-switch perspective. As we show later, naively extending this to a network-wide context can result in inefficient use of compute resources on switches and/or result in inaccurate estimates (see §2.4). This work develops network-wide coordination schemes and demonstrate an implementation in P4 [55]. Further, we show the fidelity of UnivMon on a broader set of traces and metrics.

2.2 UnivMon architecture

In this section, we provide a high-level overview of the UnivMon framework. We begin by highlighting the end-to-end workflow to show the interfaces between (a) the UnivMon control plane and the management applications and (b) between the UnivMon control and data plane components. We discuss the key technical requirements that UnivMon needs to satisfy and why these are challenging. Then, we briefly give an overview of the control and data plane design to set up the context for the detailed design in the following sections.[1]

Figure 2.1 shows an end-to-end view of the UnivMon framework. The UnivMon data plane nodes run general-purpose monitoring primitives that process the incoming stream of packets it sees and maintains a set of counter data structures associated with this stream. The UnivMon control plane assigns monitoring responsibilities across the nodes. It periodically collects statistics from the data plane, and estimates the various application-level metrics of interest.

Requirements and challenges: There are three natural requirements that UnivMon should satisfy:

- **[R1.]** *Fidelity for a broad spectrum of applications:* Ideally UnivMon should require no prior knowledge of the set of metrics to be estimated, and yet offer strong guarantees on accuracy.

- **[R2.]** *One-big-switch abstraction for monitoring:* There may be several network-wide management tasks interested in measuring different dimensions of traffic; e.g., source IPs, destination ports, IP 5-tuples. UnivMon should provide a "one big switch" abstraction for monitoring to the management applications running atop UnivMon, so that the estimates

[1]We use the terms routers, switches, and nodes interchangeably.

appear as if all the traffic entering the network was monitored at a giant switch [56].

- **[R3.]** *Feasible implementation roadmap:* While pure software solutions (e.g., Open vSwitch [62]) may be valuable in many deployments, for broader adoption and performance requirements, the UnivMon primitives used to achieve [R1] and [R2] must have a viable implementation in (emerging) switch hardware [55, 63].

Given the trajectory of prior efforts that offer high generality and low fidelity (e.g, packet sampling) vs. low generality and high fidelity (e.g., custom sketches), [R1] may appear infeasible. To achieve [R2], we observe that if each router acts on the traffic it observes independently, it can become difficult to combine the measurements and/or lead to significant imbalance in the load across routers. Finally, for [R3], we note that even emerging flexible switches [63, 64, 55] have constraints on the types of operations that they can support.

Approach Overview: Next, we briefly outline how the UnivMon control and data plane designs address these key requirements and challenges:

- *UnivMon data plane:* The UnivMon plane uses sketching primitives based on recent theoretical work on *universal streaming* [49, 50]. By design, these so-called universal sketches require no prior knowledge of the metrics to be estimated. More specifically, as long as these metrics satisfy a series of statistical properties discussed in detail in §2.3, we can prove theoretical guarantees on the memory-accuracy tradeoff for estimating these metrics in the control plane.

- *UnivMon control plane:* Given that the data plane supports universal streaming, the control plane needs no additional capabilities w.r.t. [R1] once it collects the sketch information from the router. It runs simple estimation algorithms for every management application of interest as we discuss in §2.3 and provides simple APIs and libraries for applications to run estimation queries on the collected counters. To address [R2], the UnivMon control plane generates *sketching manifests* that specify the monitoring responsibility of each router. These manifests specify the set of universal sketch instances for different dimensions of interest (e.g., for source IPs, for 5-tuples) that each router needs to maintain for different origin-destination (OD) pair paths that it lies on. This assignment takes into account the network topology and routing policies and knowledge of the hardware resource constraints of its network elements.

In the following sections, we begin by providing the background on *universal streaming* that forms the theoretical basis for UnivMon. Then, in §2.4, we describe the network-wide coordination problem that the UnivMon control plane solves. In §2.5, we show how we implement this design in P4 [55, 65].

2.3 Theoretical Foundations of UnivMon

In this section, we first describe the theoretical reasoning behind universal streaming and the class of supported functions [50, 49]. Then, we present and explain the underlying algorithms from universal streaming which serve as a basis for UnivMon. We also show how several canonical network monitoring tasks are amenable to this approach.

2.3.1 Theory of Universal Sketching

For the following discussion, we consider an abstract stream $D(m,n)$ of length m with n unique elements. Let f_i denote the frequency of the i-th unique element in the stream.

The intellectual foundations of many streaming algorithms can be traced back to the celebrated lemma by Johnson and Lindenstrauss [66]. This shows that N points in Euclidean space can be embedded into another Euclidean space with an exponentially smaller dimension while approximately preserving the pairwise distance between the points. Alon, Matias, and Szegedy used a variant of the Johnson-Lindenstrauss lemma to approximately compute the second moment of the frequency vector $= \sum_i f_i^2$ (or the L_2 norm $= \sqrt{\sum_i f_i^2}$) in the streaming model [59], using a small (polylogarithmic) amount of memory. The main question that *universal streaming* seeks to answer is whether such methods can be extended to more general statistics of the form $\sum g(f_i)$ for an arbitrary function g. We refer to this statistic as the *G-sum*.

Class of *Stream-PolyLog* Functions: Informally, streaming algorithms which have polylogarithmic space complexity, are known to exist for *G-sum* functions, where g is monotonic and upper bounded by the function $O(f_i^2)$ [49, 67].[2] Note that this only guarantees that *some* (possibly custom) sketching

[2]This is an informal explanation; the precise characterization is more technically involved and can be found in [49]. While streaming algorithms are also known for *G-sum* when its g grows monotonically faster than f_i^2 [42] they cannot be computed in polylogarithmic space

algorithm exists if *G-sum* $\in$ *Stream-PolyLog* and does not argue that a single "universal" sketch can compute all such *G-sum*s.

Intuition Behind Universality: The surprising recent theoretical result of universal sketches is that for any function $g()$ where *G-sum* belongs to the class *Stream-PolyLog* defined above can now be computed by using a *single universal sketch*.

The intuition behind universality stems from the following argument about heavy hitters in the stream. Informally, item i is a heavy hitter w.r.t. g if changing its frequency f_i significantly affects the *G-sum* value as well. For instance, consider the frequency vector $(\sqrt{n}, 1, 1, \ldots, 1)$ of size n; here the first item is a L_2 heavy hitter since its frequency is a large fraction of the L_2 norm of the frequency vector. For function g, let *G-core* be the set containing g-heavy elements. g-heavy elements can be defined as, for $0 < \gamma < 1$, any element $i \in [n]$ such that $g(f_i) > \gamma \sum_j g(f_j)$.

Now, let us consider two cases:

1. There is one sufficiently large g-heavy hitter in the stream:
   If the frequency vector has one (sufficiently) large heavy hitter, then most of mass is concentrated in this value. Now, it can be shown that a heavy hitter for the L_2 norm of the frequency vector is also a heavy hitter for computable g [49, 67]. Thus, to compute *G-core*, we can simply find L_2 heavy hitters (L2-HH) using some known techniques (e.g., [6, 59]) and use it to estimate *G-sum*.

2. There is no single g-heavy hitter in the stream and no single element

due to the lower bound $\Omega(n^{1-2/k})$ where $k > 2$ [68].

contributes significantly to the *G-sum*:

When there is no single large heavy hitter, it can be shown that *G-sum* can be approximated w.h.p. by finding heavy hitters on a series of sampled *substreams* of increasingly smaller size. The exact details are beyond the scope of this chapter [49] but the main intuition comes from tail bounds (Chernoff/Hoeffding). Each substream is defined recursively by the substream before it, and is created by sampling the previous frequency vector by replacing each coordinate of the frequency vector with a zero value with probability 0.5. Repeating this procedure k times reduces the dimensionality of the problem by a factor of 2^k. Then, summing across heavy hitters of all these recursively defined vectors, we create a single "recursive sketch" which gives a good estimate of *G-sum* [50].

2.3.2 Algorithms for Universal Sketching

Using the intuition from the two cases described above, we now have the following universal sketch construction using an online sketching stage and an offline estimation stage. The proof of the theorems governing the behavior of these algorithms is outside the scope of this chapter and we refer readers to the previous work of Braverman et al [49, 50]. In this section, we focus on providing a conceptual view of the universal sketching primitives. As we will discuss later, the actual data plane and control plane realization will be slightly different to accommodate switch hardware constraints (see §2.5).

In the online stage, as described in Algorithm 1, we maintain $\log(n)$ parallel copies of a "L_2-heavy hitter" (L2-HH) sketch (e.g., [6]), one for each

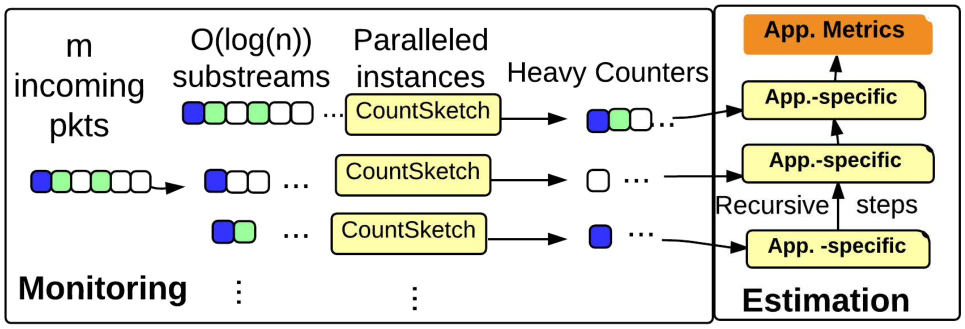


Figure 2.2: High-level view of universal sketch.

substream as described in case 2 above. For the j^{th} parallel instance, the algorithm processes each incoming packet 5-tuple and uses an array of j pairwise independent hash functions $h_i : [n] \to \{0,1\}$ to decide whether or not to sample the tuple. When 5-tuple tup arrives in the stream, if for all h_1 to h_j, $h_i(tup) = 1$, then the tuple is added to D_j, the sampled substream. Then, for substream D_j, we run an instance of L2-HH as shown in Algorithm 1, and visualized in Figure 2.2. Each L2-HH instance outputs Q_j that contains L_2 heavy hitters and their estimated counts from D_j. This creates substreams of decreasing lengths as the j-th instance is expected to have all of the hash functions agree to sample half as often as the $(j-1)$-th instance. This data structure is all that is required for the online portion of our approach.

In the offline stage, we use Algorithm 2 to combine the results of the parallel copies of Algorithm 1 to estimate different *G-sum* functions of interest. This method is based on the Recursive Sum Algorithm from [50]. The input to this algorithm is the output of Algorithm 1; i.e., a set of $\{Q_j\}$ buckets maintained by the L2-HH sketch from parallel instance j. Let $w_j(i)$ be the

[3]In this way, we obtain $\log(n)$ streams $D_1, D_2 \dots D_{\log(n)}$; i.e., for $j = 1 \dots \log n$, the number of unique items n in D_{j+1}, is expected to be half of D_j.

Algorithm 1 UnivMon Online Sketching Step

```
Input: Packet stream $D(m,n) = \{a_1, a_2, \dots, a_m\}$
  • Generate $\log(n)$ pairwise independent hash functions $h_1 \dots h_{\log(n)} : [n] \to \{0,1\}$.
  • Run L2-HH sketch on D and maintain HH set $Q_0$.
  • For $j = 1$ to $\log(n)$, in parallel:
    1. when a packet $a_i$ in $D$ arrives, if all $h_1(a_i) \times h_2(a_i) \cdots \times h_j(a_i) = 1$,
       sample and add $a_i$ to sampled substream $D_j$.[3]
    2. Run L2-HH sketch on $D_j$ and maintain heavy hitters $Q_j$.
Output: $Q = \{Q_0, \dots, Q_{\log(n)}\}$
```

Algorithm 2 UnivMon Offline Estimation Algorithm

```
Input: Set of heavy hitters $Q = \{Q_0, \dots, Q_{\log(n)}\}$
  • For $j = 0 \dots \log(n)$, call g() on all counters $w_j(i)$ in $Q_j$. After $g()$, the $i$-th
    entry in $Q_j$ is $g(w_j(i))$.
  • Compute $Y_{\log(n)} = \sum_i g(w_{\log(n)}(i))$.
  • For each $j$ from $\log(n) - 1$ to 0, compute:
        $Y_j$=2$Y_{j+1}$+$\sum_{i \in Q_j}$(1-2$h_{j+1}(i)$) $g(w_j(i))$
Output: $Y_0$
```

counter of the i-th bucket (heavy hitter) in Q_j. $h_j(i)$ is the hash of the value of the i-th bucket in Q_j where h_j is the hash function described in Algorithm 1 Step 1. It can be shown that the output of Algorithm 2 is an unbiased estimator of *G-sum* [49, 50]. In this algorithm, each Y is recursively defined, where Y_j is function g applied to each bucket of Q_j, the L2-HH sketch for substream D_j, and the sum taken on the value of those buckets and all $Y_{j'}, j' > j$. Note that $Q_{\log(n)}$ is the set of heavy hitters from the sparsest substream $D_{\log(n)}$ in Algorithm 1, and we begin by computing $Y_{\log(n)}$. Thus, Y_0 can be viewed as computing *G-sum* in parts using these sampled streams.

The key observation here is that the online component, Algorithm 1, which will run in the UnivMon data plane is *agnostic* to the specific choice of g in the offline stage. This is in stark contrast to custom sketches where the online and offline stages are both tightly coupled to the specific statistic we want to compute.

2.3.3 Application to Network Monitoring

As discussed earlier, if a function *G-sum* $\in$ *Stream-PolyLog*, then it is amenable to estimation via the universal sketch. Next, we show that a range of network measurement tasks can be formulated via a suitable *G-sum* $\in$ *Stream-PolyLog*. For the following discussion, we consider network traffic as a stream $D(n, m)$ with m packets and at most n unique flows. When referring to the definitions of Heavy Hitters, note that L_2 heavy hitters are a stronger notion that subsumes L_1 heavy hitters.

Heavy Hitters: To detect heavy hitters in the network traffic, our goal is to identify the flows that consume more than a fraction γ of the total capacity [39]. Consider a function $g(x) = x$ such that the corresponding *G-core* outputs a list of heavy hitters with$(1 \pm \epsilon)$-approximation of their frequencies. For this case, these heavy hitters are L_1-heavy hitters and $g(x)$ is upperbounded by x^2. Thus we have an algorithm that provides *G-core*. This is technically a special case of the universal sketch; we are not ever computing a *G-sum* function and using *G-core* directly in all cases.

DDoS Victim Detection: Suppose we want to identify if a host X is experiencing a Distributed Denial of Service (DDoS) attack. We can do so using sketching by checking if more than k unique flows from different sources are communication with X [44]. To show that the simple DDoS victim detection problem is solvable by the universal sketch, consider a function g that $g(x) = x^0$ and $g(0) = 0$. Here g is upper bounded by $f(x) = x^2$ and sketches already exist to solve this exact problem. Thus, we know *G-sum* is in *Stream-PolyLog* and we approximate *G-sum* in polylogarithmic space using the universal sketch. In terms of interpreting the results of this measurement, if *G-sum* is estimated to be larger than k, a specific host is a potential DDoS victim.

Change Detection: Change detection is the process of identifying flows that contribute the most to traffic change over two consecutive time intervals. As this computation takes place in the control plane, we can store the output of the universal sketches from multiple intervals without impacting online performance. Consider two adjacent time intervals t_A and t_B. If the volume for a flow x in interval t_A is $S_A[x]$ and $S_B[x]$ over interval t_B. The difference signal for x is defined as $D[x] = |S_A[x] - S_B[x]|$. A flow is a *heavy* change flow if the difference in its signal exceeds ϕ percentage of the total change over all flows. The total difference is $D = \sum_{x \in [n]} D[x]$. A flow x is defined to be a heavy change iff $D[x] \geq \phi \cdot D$. The task is to identify these heavy change flows. We assume the size of heavy change flows is above some threshold T over the total capacity c. We can show that the heavy change flows are L_1 heavy hitters on interval t_A $(a_1 \cdots a_{n/2})$ and interval t_B $(b_1 \cdots b_{n/2})$, where $L_1(t_A, t_B) = \sum |a_i - b_i|$. *G-sum* here is L_1 norm, which belongs to *Stream-PolyLog*, and *G-core* can be solved by universal sketch. The *G-sum* outputs the estimated size of the total change D and *G-core* outputs the possible heavy change flows. By comparing the outputs from *G-sum* and *G-core*, we can detect and determine the heavy change flows that are above some threshold of all flows.

Entropy Estimation: We define entropy with the expression $H \equiv -\sum_{i=1}^{n} \frac{f_i}{m}\log(\frac{f_i}{m})$ [10] and we define $0 \log 0 = 0$ here. The entropy estimation task is to estimate H for source IP addresses (but could be performed for ports or other features). To compute the entropy, $H = -\sum_{i=1}^{n} \frac{f_i}{m}\log(\frac{f_i}{m}) = \log(m)$ $-\frac{1}{m}\sum_i f_i \log(f_i)$. As m can be easily obtained,[4] the difficulty lies in calculating $\sum_i f_i \log(f_i)$. Here the function $g(x) = x\log(x)$ is bounded by $g(x) = x^2$ and thus its *G-sum* is in *Stream-PolyLog* and H can be estimated by universal sketch.

Global Iceberg Detection: Consider a system or network that consists of N distributed nodes (e.g., switches). The data set S_j at node j contains a stream of tuples $<$ $item_{id}$, c$>$ where $item_{id}$ is an item identity from a set $U = \{\mu_1 \dots \mu_n\}$ and c is an incremental count. For example, an item can be a packet or an origin-destination (OD) flow. We define $fr_i = \sum_j \sum_{<\mu_i,c>\in S_j} c$, the frequency of the item μ_i when aggregated across all the nodes. We want to detect the presence of items whose total frequency across all the nodes adds up to exceed a given threshold T. In other words, we would like to find out if there exists an element $\mu_i \in U$ such that $fr_i \geq T$. (In §2.4, we will explain a solution to gain a network-wide universal sketch. Here, we assume here that we maintain an abstract universal sketch across all nodes by correctly combining all distributed sketches.) Consider a function $g(x) =$

[4]e.g., a single counter or estimated as a *G-sum*.

x such that the corresponding *G-core* outputs a list of global heavy hitters with$(1 \pm \epsilon)-$approximation of their frequencies. For this case, since g-heavy hitters are L_1 heavy hitters, we have an algorithm that provides *G-core*.

2.4 Network-wide UnivMon

In a network-wide context, we have flows from several origin-destination (OD) pairs, and applications may want network-wide estimates over multiple packet header combinations of interest. For instance, some applications may want per-source IP estimates, while others may want characteristics in terms of the entire IP-5-tuple. We refer to these different packet header features and feature-combinations as *dimensions*.

In this section, we focus on this network-wide monitoring problem of measuring multiple dimensions of interest traversing multiple OD-pairs. Our goal is to provide equivalent coverage and fidelity to a "one big switch abstraction", providing the same level of monitoring precision at the network level as at the switch level. We focus mostly for the case where each OD-pair has a single network route and describe possible extensions to handle multi-pathing.

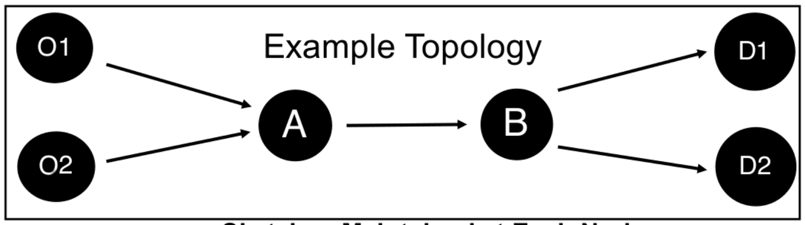
2.4.1 Problem Scope

We begin by scoping the types of network-wide estimation tasks we can support and formalize the one-big-switch abstraction that we want to provide in UnivMon. To illustrate this, we use the example in Figure 2.3 where we want to measure statistics over two dimensions of interest: 5-tuple and source-IP.

OD Pair	Label
O1,D1	P11
O1,D2	P12
O2,D1	P21
O2,D2	P21
All	P*

Sketch	Label
5-tuple	5t
Source IP	src
All	S*

Key
N – Number of Nodes
D – Number of Dimensions
Load – Sketches at a Node



Sketches Maintained at Each Node

Algorithms	O1	O2	A	B	D1	D2	Max Load	Total Sketches
RM	P*/S*	P*/S*	P*/S*	P*/S*	P*/S*	P*/S*	D	N*D
IM	P*/S*	P*/S*	0	0	0	0	D	~N*D
GDC	P11/src P12/src	P21/src P22/src	P*/5t	0	0	0	1	~D
Q&S	0	0	P*/src	P*/5t	0	0	1	D

Figure 2.3: Example topology to explain the one-big-switch notion and to compare candidate network-wide solutions.

In this example, we have four OD-pairs; suppose the set of traffic flows on each of these is denoted by P_{11}, P_{12}, P_{21}, and P_{22}. We can divide the traffic in the network into four partitions, one per OD-pair. Now, imagine we abstract away the topology and consider the union of the traffic across these partitions flowing through one logical node representing the entire network; i.e., computing some estimation function $F(P_{11} \uplus P_{12} \uplus P_{21} \uplus P_{22})$, where $\uplus$ denotes the disjoint set union operation.

For this work, we restrict our discussion to network-wide functions where we can independently compute the F estimates on each OD-pair substream and add them up. In other words, we restrict our problem scope to estimation functions Fs such that:

$$F(P_{11} \uplus P_{12} \uplus P_{21} \uplus P_{22}) = F(P_{11}) + F(P_{12}) + F(P_{21}) + F(P_{22})$$

Note that this still captures a broad class of network-wide tasks such as those mentioned in section 2.3.3. One such example measurement is finding heavy hitters for destination IP addresses. An important characteristic of the UnivMon approach is that in the network-wide setting the output of sketches in the data plane can then be added together in the control plane to give the same results as if all of the packets passed through one switch. The combination of the separate sketches is a property of the universal sketch primitive used in the data plane and is independent of the final statistic monitored in the control plane, allowing the combination to work for all measurements supported by UnivMon. We do however acknowledge that some tasks fall outside the scope of this partition model; an example statistic that is out of scope would be measuring the statistical independence of source and destination IP address pairs (i.e. if a source IP is likely to appear with a given destination IP, or not), as this introduces cross-OD-pair dependencies. We leave extensions to support more general network-wide functions for future work (see §6).

The challenge here is to achieve *correctness* and *efficiency* (e.g., switch memory, controller overhead) while also *balancing the load* across the data plane elements. Informally, we seek to minimize the total number of sketches instantiated in the network and the maximum number of sketches that any single node needs to maintain.

2.4.2 Strawman Solutions and Limitations

Next, we discuss strawman strategies and argue why these fail to meet one or more of our goals w.r.t. correctness, efficiency, and load balancing. We observe that we can combine the underlying sketch primitives at different switches as long as we use the same random seeds for our sketches, as the counters are additive at each level of the UnivMon sketch. With this, we only need to add the guarantee that we count each packet once to assure correctness. In terms of resource usage, our goal is to minimize the number of sketches used.

Redundant Monitoring (RM): Suppose for each of k dimensions of interest, we maintain a sketch on every node, with each node independently processing traffic for the OD-pairs whose paths it lies on. Now, we have the issue of combining sketches to get an accurate network-wide estimate. In particular, adding all of the counters from the sketches would be incorrect, as packets would be counted multiple times. In the example topology, to correctly count packets we would need to either only use the sketches at A or B, or, conversely, combine the sketches for source IP at $O1$ and $O2$ or $D1$ and $D2$. In terms of efficiency, this RM strategy maintains a sketch for all k dimensions at each node and thus we maintain a total of kN sketches across N nodes. Our goal, is to maintain s total sketches, where $s << kN$.

Ingress Monitoring (IM): An improvement over the RM method is to have only *ingress nodes* maintaining every sketch. Thus, for each OD pair, all sketch information is maintained in a single node. By not having duplicate sketches per OD pair, we will not double count and therefore can combine sketches together. This gives us the correctness guarantee missing in RM. In Figure 2.3, IM would maintain sketches at $O1$ and $O2$. However, for N_i ingress nodes, we would run kN_i sketches, and if $N_i \approx N$ we spend a similar amount of resources to RM, which is still high. Additionally, these sketches woul be would all be present on a small number of nodes, where other nodes with available compute resources would not run any sketches.

Greedy Divide and Conquer (GDC): To overcome the concentration of sketches in IM above, one potential solution is to evenly divide sketch processing duties across the path. Specifically, each node has a priority list of sketches, and tags packets with the current sketches that are already maintained for this flow so that downstream nodes know which remaining sketches to run. For instance, in Figure 2.3, GDC would maintain the source IP sketch at $O1$ and $O2$, and the 5-tuple sketch at A. This method is correct, as each sketch for each OD pair is maintained once. However, it is difficult to properly balance resources as nodes at the intersection of multiple paths could be burdened with higher load.

Reactive Query and Sketch (QS): An alternative approach is to use the controller to ensure better sketch assignment. For instance, whenever a new flow is detected at a node, we query the controller to optimally assign sketches. In Figure 2.3, the controller would optimally put the source IP sketch at A and the 5-tuple sketch at B (or vice versa). With this method, we can be assured of correctness. However, the reactive nature means that QS generates many requests to the controller.

2.4.3 Our Approach

Next, we present our solution, which uses the UnivMon controller to coordinate switches to guarantee correctness and efficiency but without incurring the reactive query load of the QS strategy described above.

Periodically, the UnivMon controller gives each switch a *sketching manifest*. For each switch A and for each OD-pair's route that A lies on, the manifest specifies the dimensions for which A needs to maintain a sketch. When a packet arrives at a node, the node uses the manifest to determine the set of sketching actions to apply. When the controller needs to compute a network-wide estimate, we pull sketches from all nodes and for each dimension, combine the sketches across the network for that dimension. This method minimizes communication to the control plane while still making use of the controller's ability to optimize resource use.

The controller solves a simple constrained optimization problem that we discuss below. Note that maintaining two sketches uses much more memory than adding twice as many elements to one sketch. Thus, a key part of this optimization is to ensure that we try to reuse the same sketch for a given dimension across multiple OD pairs. In Figure 2.3, we would first assign A the source IP sketch, then B the 5-tuple sketch for the OD pair $(O1, D1)$. When choosing where to place the sketches for the OD pair $(O2, D2)$, the algorithm matches the manifests such that the manifest for $(O2, D2)$ uses the source IP sketch already at A and the 5-tuple sketch already at B.

We formulate the controller's decision to place sketches as an integer linear program (ILP) shown in Figure 2.4. Let s_{jk} be a binary decision variable

Minimize: $N \times Maxload + Sumload$, subject to

$$\forall i,k : \sum_{j \in p_i} s_{jk} \geq 1 \tag{2.4.1}$$

$$\forall j : Load_j = \sum_k r_k \times s_{jk} \tag{2.4.2}$$

$$\forall j : \sum s_{jk} r_k \leq R \tag{2.4.3}$$

$$\forall j : Maxload \geq Load_j \tag{2.4.4}$$

$$\forall j : Sumload = \sum_j Load_j \tag{2.4.5}$$

Figure 2.4: ILP to compute sketching manifests.

denoting if the switch j is maintaining a sketch for dimension j. The goal of the optimization is to ensure that every OD-pair is suitably "covered" and that the load across the switches is balanced. Let r_k be the amount of memory for a sketch for dimension k and let R denote maximum amount of memory available on a single node. Note that the amount of memory for a sketch can be chosen in advance based on the accuracy required. As a simple starting point, we focus primarily on the memory resource consumption assuming that all UnivMon operations can be done at line-rate; we can extend this formulation to incorporate processing load as well.

Eq (2.4.1) captures our coverage constraint that we maintain each sketch once for each OD-pair.[5] We model the per-node load in Eq (2.4.2) and ensure that it respects the router capacity in Eq (2.4.3). Our objective function balances

[5]Our coverage constraint allows multiple sketches of the same kind to be placed in the same path. This is because in some topologies, it may not be feasible to have an equality constraint. In this case, the controller post-processes the solution and removes duplicates before assigning sketches for a given OD pair.

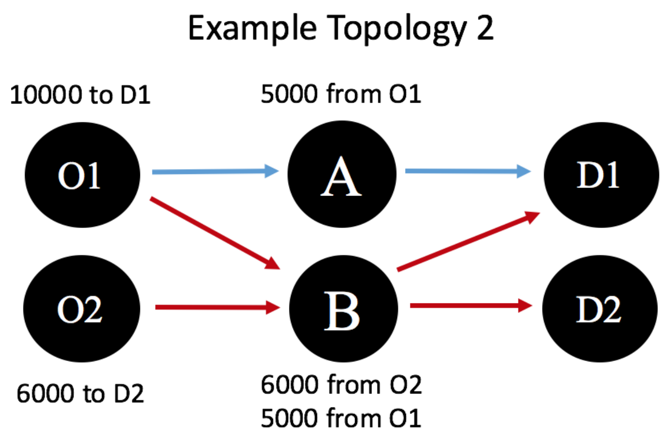


Figure 2.5: Example topology to showcase difficulty of multi-path.

two components: the maximum load that any one node faces and the total number of sketches maintained.[6]

2.4.4 Extension to Multi-path

Adapting the above technique to multi-path requires some modification, but is feasible. For simplicity, we still assume that packets are routed deterministically (e.g., by prefix rules), but may have multiple routes. We defer settings that depend on randomized or non-deterministic routing for future work.

Even in this deterministic setting, there are two potential problems. First, ensuring packets are only counted once is important to avoid false positives, as in the single path case. Second, if the packets with a heavy feature (e.g., the destination address is heavy) are divided over many routes, it can increase the difficulty of accurately finding heavy hitters, removing false positives and preventing false negatives.

[6]The N term for *MaxLoad* helps to normalize the values.

The first issue, guaranteeing packets are counted only once, is solvable by the ILP presented above. For each path used by an OD pair, we create a unique sub-pair which we treat as an independent OD pair. This is shown in Figure 2.5 by the red O1-D1 path and the blue O1-D1 path. By computing the ILP with multiple paths per OD pair as needed, sketches are distributed across nodes, and single counting is guaranteed. This method works best when the total number of paths per OD pair is constant relative to the total number of nodes, and larger numbers of paths will cause the sketches to concentrate on the source or destination nodes, possibly requiring additional solutions.

The second issue occurs when multi-path routing causes the frequency of an item to be split between too many sketches. In the single-path setting, if an OD pair has a globally heavy feature, then it will be equally heavy or heavier in the sketch where it is processed. However in the multi-path case, it is possible for some OD pairs to have more paths than others, and thus it becomes possible for items that are less frequent but have fewer routes to be incorrectly reported heavy, and in turn fail to report true heavy elements in the control plane. This problem is shown in Figure 2.5. In this case, we have 10,000 packets from node O1 to D1 split across two paths, and 6,000 packets from O2 to D2. For simplicity, assume we are only looking for the "heaviest" source IP, the source IP with the highest frequency, and that the nodes have a single IP address, (i.e. Packets go from IP_{O_1} to IP_{D_1} and IP_{O_1} IP_{D_2}). For this metric, the sketch at A will report IP_{O_1} as a heavy source address with count 5,000, and B will report IP_{O_2} as a heavy source address with count 6,000. At the data plane these values are compared again, and the algorithm would return IP_{O_2}, a false positive, and miss IP_{O_1}, a false negative. To solve this issue, instead of sending the heavy hitter report from individual sketches as described in Algorithm 1, the counters from each sketch must be sent directly to the control plane to be added, reconstructing the entire sketch and allowing the correct heavy hitters to be identified. In our example, the counters for the O1 at A and B would be added, and IP_{O_1}would be correctly identified as the heavy hitter. This approach is already used in the P4 implementation discussed below, but is not a requirement of UnivMon in general. We note that when using the method described below in Section 2.5.2, identifying the true IP address of the heavy item is harder in the multi-path setting, but is solved by increasing γ relative to the maximum number of sketches per true OD pair, which is naturally limited by the ILP. With these modifications, the heavy hitters are correctly found from the combined sketch, and the one big switch abstraction are maintained in a multi-path setting.

2.5 UnivMon Implementation

In this section, we discuss our data plane implementation in P4 [55, 65]. We begin by giving an overview of key design tradeoffs we considered. Then, we describe how we map UnivMon into corresponding P4 constructs.

2.5.1 Implementation overview

At a high level, realizing the UnivMon design described in the previous sections entails four key stages:

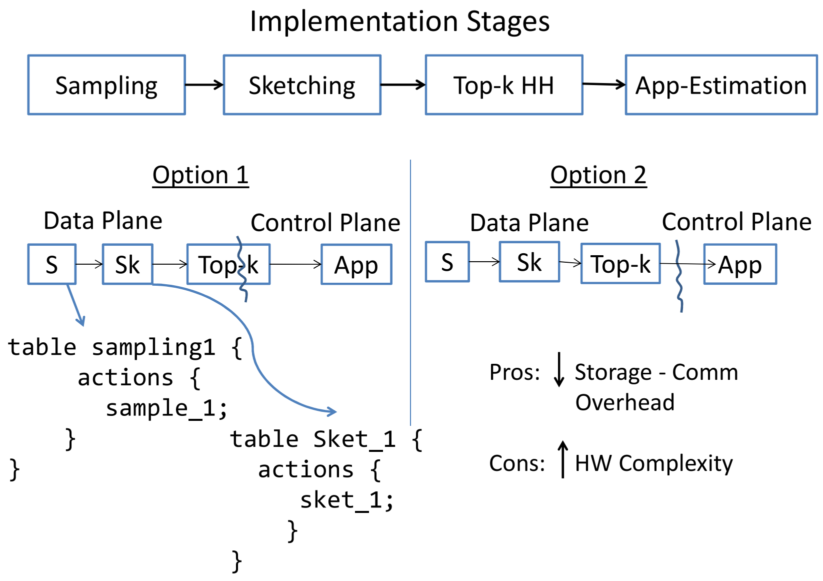


Figure 2.6: An illustration of UnivMon's stages along with the two main implementation options.

1. A **sample** stage which decides whether an incoming packet will be added to a specific substream.

2. A **sketching** stage which calculates sketch counters from input substreams and populates the respective sketch counter arrays.

3. A **top-k** computation stage which identifies (approximately) the k heaviest elements of the input stream.

4. An **estimation** stage which collects the heavy element frequencies and calculates the desired metrics.

Let us now map these stages to our data and control plane modules from Figure 2.1. Our delayed binding principle implies that the estimation stage maps to the UnivMon control plane. Since the sample and sketching are processing packets, they naturally belong in the data plane to avoid control plane overhead.

One remaining question is whether the top-k computation stage is in the data or control plane (Figure 2.6). Placing the top-k stage in the data plane has two advantages. First, the communication cost between the data and control plane will be low, as only the top-k rather than raw counters need to be transferred. Second, the data plane only needs to keep track of the flowkeys (e.g., source IP) of the k heaviest elements at any given point in time, and thus not incur high memory costs. However, one stumbling block is that realizing this stage requires (i) sorting counter values and (ii) storing information about the heavy elements in some form of a priority queue. Unfortunately, these primitives may be hard to implement in hardware and are not supported in P4 yet. Thus, we make a pragmatic choice to split the top-k stage between the control and the data planes. We identify the top-k heavy flowkeys in the dataplane and then we use the raw data counters to calculate their frequencies in the control plane. The consequence is that we incur higher communication overhead to report the raw counter data structure, but the number of flowkeys stored in the data plane remains low.

UnivMon's raw counters and flowkeys are stored on the target's on-chip memory (TCAM and SRAM). We argue that in practice the storage overhead of UnivMon is manageable even for hardware targets with limited SRAM [69, 70, 44]. We show that for the largest traces that we evaluate and without losing accuracy, the total size of the raw counters can be less than 600 KB whereas the cost of storing flowkeys (assuming k is ≤ 20) is only a few KBs per measurement epoch. Thus, this decision to split the top-k between the two planes computation is practical and simplifies the data plane requirements.

2.5.2 Mapping UnivMon data plane to P4

Based on the above discussion, the UnivMon data plane implements sample, sketching, and "heavy" flowkey storage in P4. In a P4 program, packet processing is implemented through Match+Action tables, and the control flow of the program dictates the order in which these tables are applied to incoming packets. Given the sketching manifests from the control plane, we generate a control program that defines the different pipelines that a packet needs to be assigned to. These pipelines are specific to the dimension(s) (i.e., source IP, 5-tuple) for which the switch needs to maintain a universal sketch. We begin by explaining how we implemented these functions and then describe a sample control flow.

sample: P4 enables programmable calculations on specific header fields using user-defined functions. We use this to sample incoming packets, with a configurable flowkey that can be any subset of the 5-tuple (srcIP, dstIP, srcPort, dstPort, protocol). We define l pairwise-independent hash functions, where l is the number of levels from §2.3. These functions take as input the flowkey and output a binary value. We store this output bit as packet metadata. A packet is sampled at level i if the outputs of the hash functions of all levels $\leq i$ is equal to 1. We implement sampling for each level as a table that matches all packets and whose action is to apply the sampling hash function of that level. The hash metadata in the packets are used in conditional statements in the control flow to append the packet to the first *i* substreams. Packets that are not sampled are not subject to further UnivMon processing.[7]

Sketching: The sketching stage is responsible for maintaining counters for each one of the *l* substreams. From these sketch counters, we can estimate the L2-HH for each stage and then the overall top-*k* heavy hitters and their counts. While UnivMon does not impose any constraints on the L2-HH algorithm to be used, in our P4 implementation we use Count Sketch [6]. The sketching computation for each level is implemented as a table that matches every packet belonging to that level's substream and its actions update the counters, stored in the sketch counter arrays. Similar to the sample stage, we leverage user-defined hash functions that take as input the same flowkey as in the sample stage. We use their output to retrieve the indexes of the sketch register arrays cells that correspond to a particular packet and update their value as dictated by the Count Sketch algorithm.

P4 provides a *register* abstraction which offers a form of stateful memory that can store user-defined data and that can be arranged into one dimensional arrays of user-defined length. Register cells can be read or written by P4 action statements and are also accessible through the control plane API. Given that our goal is to store sketch counter values which do not represent byte or packet counts, we use register arrays to store and update sketch counters. The size of the array and the bitlength of each array cell are user-defined and can be varied based on the required memory-accuracy tradeoff as well as on the

[7]There may be other routing/ACL actions to be applied to the packet but this is outside our scope.

available on-chip memory of the hardware target. Each sketch is an array of t rows and w columns. We instantiate register arrays of length $t * w$, and the bitlength of each cell is based on the maximum expected value of a counter.

The one remaining issue is storing flowkeys corresponding to the "heavy" elements since these will be needed by the estimation stage running in the control plane. One option is to use a priority queue to maintain the top k heavy hitters online, as it is probably the most efficient and accurate choice to maintain heavy flowkeys. However, this can incur more than constant update time for each element, which makes it difficult to implement on hardware switches. To address the issue, we use an alternative approach which is to maintain a fixed sized table of heavy keys and use constant time updates for each operation. It is practical and acceptable when the size of the table is small (e.g., 10-50) and the actual number of heavy flows doesn't greatly exceed this size. The lookup/update operations could be very fast (in a single clock cycle) when leveraging some special types of memory (e.g., TCAM) on hardware switches.

Another scheme we use is as follows, and we leave improved sketches for finding heavy flowkeys as future work. For γ-threshold heavy hitters, there are at most $1/\gamma$ of them. While packets are being processed, we maintain an up-to-date L_2 value (of the frequency vector), specifically $L_2 = (L_2^2 + (c_i + 1)^2 - (c_i)^2)^{1/2}$, where c_i is each flow's current count and we create $\log(1/\gamma)$ buckets of size k. In the online stage, when updating the counters in L2-HH, c_i is obtained by reading current sketch counters.

We then maintain buckets marked with $L_2/2, L_2/4, \ldots, \gamma L_2$. For each element that arrives, if its counter is greater than $L_2/2$, insert it into the $L_2/2$ bucket using a simple hash; otherwise, if its counter is greater than $L_2/4$, insert it into the $L_2/4$ bucket, and so forth. When the value of L_2 doubles itself, we delete the last γL_2 bucket and we add a new $L_2/2$ bucket. This scheme ensures that $O(k \log(1/\gamma))$ flowkeys are stored, and at the end of the stream we can return most top k items heavier than γL_2.

P4 Control Flow: As a simple starting point, we use a sequential control flow to avoid cloning every incoming packet l (i.e., number of levels) times. This means that every packet is processed by a sketching, a storage and a sampling table sequentially until the first level where it doesn't get sampled. More specifically, after a packet passes the parsing stage during which P4 extracts its header fields, it is first processed by the sketching table of level_0. The "heavy" keys for that stage are updated and then it is processed by the sampling table of level_1. If the packet gets sampled at level_1, it is sketched at this level, the "heavy" keys are updated and the procedure continues until the packet reaches the last level or until it is not sampled.

2.5.3 Control plane

We implement the UnivMon control plane as a set of custom C++ modules and libraries. We implement modules for (1) Assigning sketching responsibilities to the network elements, and (2) implementing the top-k and estimation stages. The P4 framework allows us to define the API for control-data plane communication. We currently use a simple RPC protocol that allows us to import sketching manifests and to query the contents of data plane register arrays

After the heavy flowkeys and their respective counters have been collected, the frequencies of the k-most frequent elements in the stream are extracted. The heavy elements along with the statistical function of the metric to be estimated are then fed to the recursive algorithm of UnivMon's estimation stage.

2.6 Evaluation

We divide our evaluation into two parts. First, we focus on a single router setup and compare UnivMon vs. custom sketches via OpenSketch [44]. Second, we demonstrate the benefits of our network-wide coordination mechanisms.

2.6.1 Methodology

We begin by describing our trace-driven evaluation setup.

Applications and error metrics: We have currently implemented translation libraries for five monitoring tasks: Heavy Hitter detection (HH), DDoS detection (DDoS), Change Detection (Change), Entropy Estimation (Entropy), and Global Iceberg Detection (Iceberg). For brevity, we only show results for metrics computed over one feature, namely the source IP address; our results are qualitatively similar for other dimensions too.

For Heavy Hitters and Global Iceberg detection, we set a threshold $T =$ 0.05% of the link capacity and identify all large flows that consume more

Trace	Loc	Date and Time
1. CAIDA'15	Equinix-Chicago	2015/02/19
2. CAIDA'15	Equinix-Chicago	2015/05/21
3. CAIDA'15	Equinix-Chicago	2015/09/17
4. CAIDA'15	Equinix-Chicago	2015/12/17
5. CAIDA'14	Equinix-Sanjose	2014/06/19

Table 2.1: CAIDA traces in the evaluation.

traffic than that threshold. We obtain the average *relative error* on the counts of each identified large flow; i.e., $\frac{|True-Estimate|}{True}$. For Change Detection, whose frequency has changed more than a threshold ϕ of the total change over all flows across two monitoring windows. We chose this threshold to be 0.05% and calculate the average relative error similar to HH. For Entropy Estimation and DDoS, we evaluate the relative error on estimated entropy value and the number of distinct source IPs.

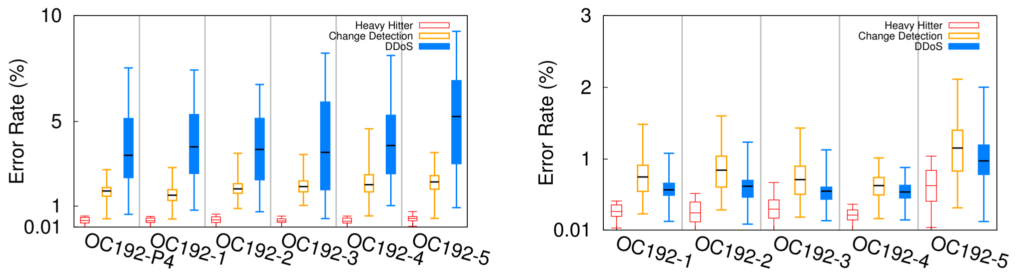
Configuration: We normalize UnivMon's memory usage with the custom sketches by varying three key parameters: number of rows t and number of columns w in Count-Sketch tables, and the number of levels l in the universal sketch. In total UnivMon uses $t \times w \times l$ counters. In OpenSketch, we configure the memory usage in a similar way by varying number of rows t and counters per row w in all the sketches they use. When comparing the memory usage with OpenSketch, we calculate the total number of sketch counters assuming that each integer counter occupies 4 bytes. Both UnivMon and OpenSketch use randomized algorithms; we run the experiment 10 times with random hash seeds and report the *median* cross these runs.

Traces: For this evaluation, we use five different one-hour backbone traces (Table 2.1) collected at backbone links of a Tier1 ISP between (i) Chicago, IL and Seattle, WA in 2015 and (ii) between San Jose and Los Angeles in 2014 [57, 58]. We split the traces into different representative time intervals (5s, 30s, 1min, 5min). For example, each one hour trace contains 720 5s-epoch data points and we report *min*, *25%*, *median*, *75%*, and *max* on whisker bars. By default, we report results for a 5-second trace. Each 5s packet-trace contains 155k to 286k packets with ∼55k distinct source IP addresses and ∼40k distinct destination IP addresses. The link speed of these traces is 10 Gbps.

Experiment Setup: For our P4 implementation prototype, we used the P4 behavioral simulator, which is essentially a P4-enabled software switch [71]. To validate the correctness of our P4 implementation, we compare it against a software implementation of the data plane and control plane algorithms, written in C++. We evaluate P4 prototype on Trace 1 and run software implementation in parallel on Trace 1- 5. The results between the two implementations are consistent as the relative error between the results of the two implementations does not exceed 0.3%. To evaluate OpenSketch, we use its simulator written in C++ [72].

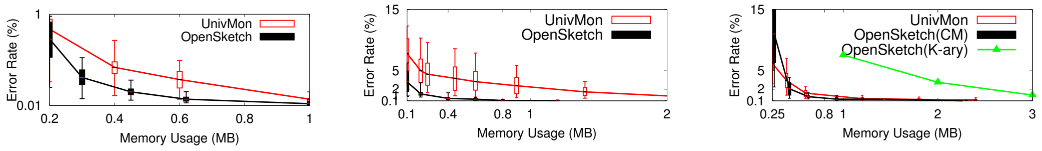
2.6.2 Single Router Evaluation

Comparison under fixed memory setting: First, we compare UnivMon and OpenSketch on the applications that OpenSketch supports: HH, Change, and DDoS. In Figures 2.7(a) and 2.7(b), we assign 600KB memory and use all traces in order to estimate the error when running UnivMon and OpenSketch. We find that the absolute error is very small for both approaches. We observe that OpenSketch provides slightly better results for all three metrics. However



(a) UnivMon (b) OpenSketch

Figure 2.7: Error rates of HH, Change and DDoS for UnivMon and OpenSketch.



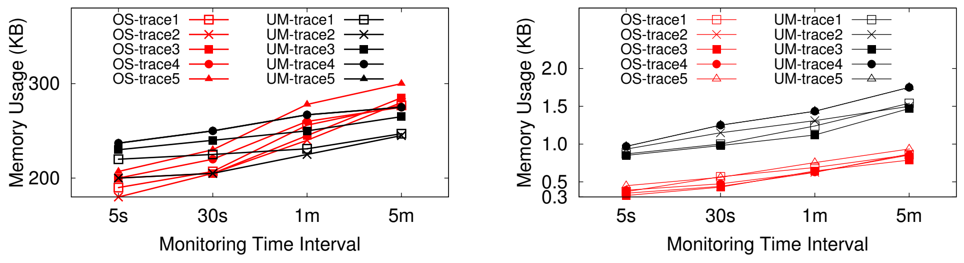
(a) HH (b) DDoS (c) Change

Figure 2.8: Error vs. Memory for HH, DDoS, Change.

we note that UnivMon uses 600KB memory to run three tasks concurrently while OpenSketch is given 600KB to run each task. Figure 2.7(a) and 2.7(b) confirm that this observation holds on multiple traces; the error gap between UnivMon and OpenSketch is $\leq$3.6%.

Accuracy vs. Memory: The previous result considered a fixed memory value. Next, we study the sensitivity of the error to the memory available. Figure 2.8(a) and 2.8(b) shows that the error is already quite small for all the HH and DDoS applications and that the gap is almost negligible with slightly increased memory $\geq$ 1MB.

Figure 2.8(c) shows the results for the Change Detection task. For this task,



(a) HH (b) Change

Figure 2.9: Average memory usage to achieve a 1% error rate for different time intervals

the original OpenSketch paper uses a streaming algorithm based on reversible k-ary sketches [9]. We implement an extension to OpenSketch using a similar idea as UnivMon.[8] Our evaluation results show that our extension offers better accuracy vs. memory tradeoff than OpenSketch's original method [9]. For completeness, we also report the memory usage of OpenSketch's original design (using the k-ary sketch). From Figure 2.8(c), we see UnivMon provides comparable accuracy even though UnivMon has a much smaller sketch table on each level of its hierarchical structure. This is because the "diff" across sketches are well preserved in UnivMon's structure.

Fixed Target Errors: Next, we evaluate the memory needed to achieve the same error rates ($\leq$1%). In Figures 2.9(a) and 2.9(b) as we vary the monitoring window, we can see that only small amount of memory increase is required for both UnivMon and OpenSketch to achieve 1% error rates. In fact, we find that UnivMon does not require more memory to maintain a stable error rate for

[8]We maintain two recent Count-Min sketches using the same hash functions; combine two sketches by one sketch "subtracts" the other; and use reversible sketch to trace back the keys.

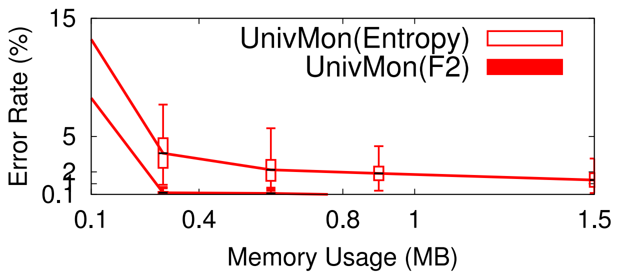


Figure 2.10: Error rates of Entropy and F2 estimation

increased number of flows in the traffic. This is largely because sketch-based approaches usually just take logarithmic memory increase in terms of input size to maintain similar error guarantees. Furthermore, the nature of traffic distribution also helps as there are only a few very heavy flows and the entire distribution is quite "flat".

Other metrics: We also considered metrics not in the OpenSketch library in Figure 2.10 to confirm that UnivMon is able to calculate a low-error estimate. Specifically, we consider the entropy of the distribution and the *second frequency moment* $F_2 = f_1^2 + f_2^2 \cdots + f_m^2$ for m distinct elements.[9] Again, we find that with reasonable amounts of memory ($\geq$ 500KB) the error of UnivMon is very low.

Impact of Application Portfolio: Next, we explore how UnivMon and OpenSketch handle a growing portfolio of monitoring tasks with a fixed memory. We set the switch memory to 600KB for both UnivMon and OpenSketch and run three different application sets: AppSet1={HH}, AppSet2

[9]This is a measure of the "skewness" and is useful to calculate repeated rate or Gini index of homogeneity.

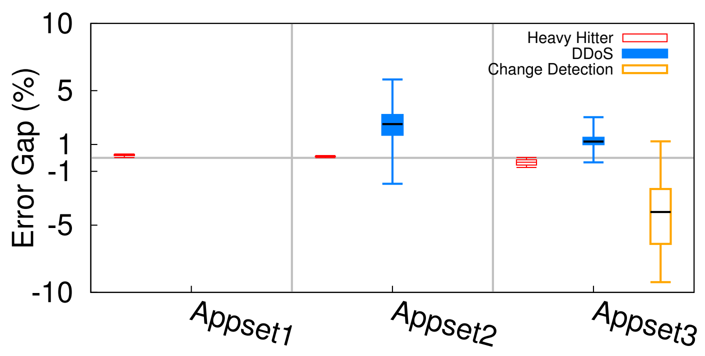


Figure 2.11: The impact of a growing portfolio of monitoring applications on the relative performance

={HH,DDoS}, and AppSet3={HH,DDoS,Change}. We assume that OpenSketch divides the memory uniformly across the constituent applications; i.e., in AppSet1 600KB is devoted to HH, but in AppSet2 and Appset3, HH only gets 300KB and 200KB respectively. Figure 2.11 shows the "error gap" between UnivMon and OpenSketch (UnivMon − OpenSketch); i.e., positive values imply UnivMon is worse and vice versa. As expected, we find that when running concurrent tasks, the error gap decreases as each task gets less memory in OpenSketch. That is, with more concurrent and supported tasks, UnivMon can still provide guaranteed results on each of the applications.

Choice of Data Structures: UnivMon uses a a sketching algorithm that identifies L_2 heavy hitters as a building block. Two natural questions arise: (1) How do different heavy hitter algorithms compare and (2) Can we use other popular heavy hitter identifiers, such as Count-Min sketch? We implemented and tested the Pick-and-Drop algorithm [41] and Count-Min sketch [61] as building blocks for UnivMon. Figure 2.12 shows that Pick-and-Drop and CM

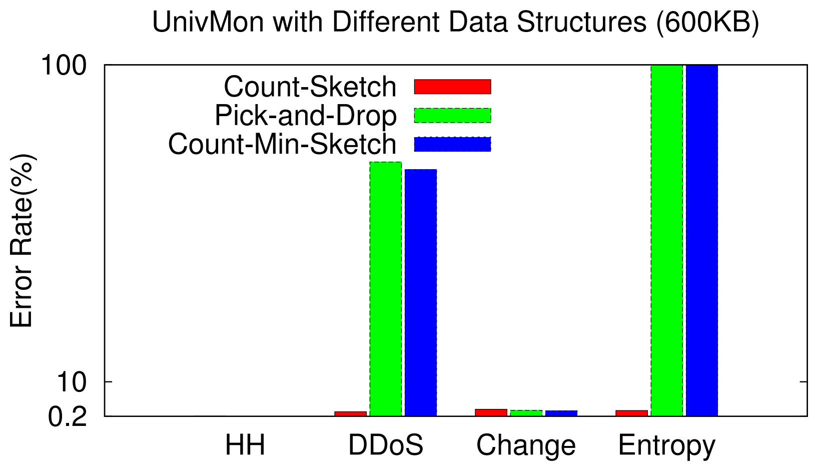
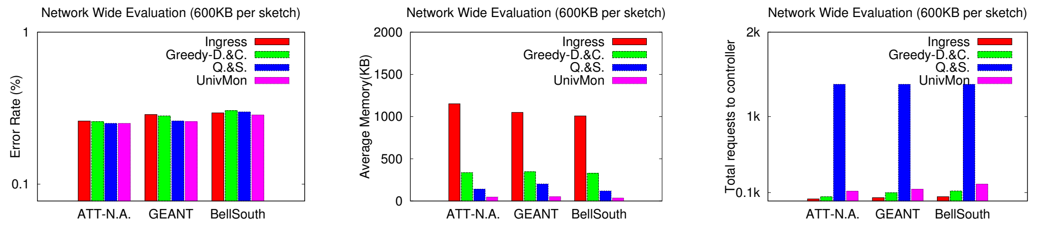


Figure 2.12: Analyzing different HH data structures

sketch lose the generality of UnivMon as they can provide accurate results only for HH and Change tasks. This is because, intuitively, only $L_p(p = 1 \text{ or } p \geq 3)$ heavy hitters are identified. The technical analysis of universal sketch shows that only L_2 heavy hitters contribute significantly to the *G-Sum* when *G-Sum* is upper bounded by some L_2 norm. As discussed in Section 2.3.3, the *G-Sum* functions corresponding to HH and Change are actually L_1 norms. Therefore, the estimated L_1 heavy hitters output by Count-Min or Pick-and-Drop work well for HH and Change tasks, but not Entropy or DDoS. When combining heavy hitter counters in the recursive step of calculation, we will simply miss too many significant heavy elements for all tasks.

Processing Overhead: One concern might be the computational cost of the UnivMon vs. custom sketch primitives. We used the Intel Performance Counter Monitor [73] to evaluate compute overhead (e.g., Total cycles on CPU) on UnivMon and OpenSketch's software simulation libraries. For any



(a) Error rates of global iceberg detection (b) Average memory consumption (c) Total number of requests to controller

Figure 2.13: Network-wide evaluation on major ISP backbone topologies

given task, our software implementation was only 15% more expensive than OpenSketch. When we look at all three applications together, however, the UnivMon takes only half the compute cycles as used by OpenSketch in total. While we acknowledge that we cannot directly translate into actual hardware processing overheads, this suggests that UnivMon's compute footprint will be comparable and possibly better.

2.6.3 Network-wide Evaluation

For the network-wide evaluation, we consider different topologies from the Topology Zoo dataset [74]. As a specific network-wide task, we consider the problem of estimating source IP and destination IP "icebergs". We report the average relative errors across these two tasks.

Benefits of Coordination: Figure 2.13(a), Figure 2.13(b), and Figure 2.13(c) present the error, average memory consumption, and total controller requests of four solutions: Ingress Monitoring(IM), Greedy Divide and Conquer(GDC), Query and Sketch(QS), and our approach(UnivMon). We pick three representative topologies: AT&T North America, Geant, and Bell South. We see

Topology	OD Pairs	Dim.	Time (s)	Total Sketches
Geant2012	1560	4	0.09	68
Bellsouth	2550	4	0.10	60
Dial Telecom	18906	4	2.8	252
Geant2012	1560	8	0.22	136
Bellsouth	2550	8	0.28	120
Dial Telecom	18906	8	12.6	504

Table 2.2: Time to compute sketching manifests using ILP.

that UnivMon provides an even distribution of resources on each node while providing results with high accuracy. Furthermore, the control overhead is several orders of magnitude smaller than purely reactive approaches.

ILP solving time: One potential concern is the time to solve the ILP. Table 2.2 shows the time to compute the ILP solution on a Macbook Pro with a 2.5 GHz Intel Core i7 processor using `glpsol` allowing at most k sketches per switch, where k is the number of dimensions maintained. We see that the ILP computation takes at most a few seconds which suggest that updates can be pushed to switches with reasonable responsiveness as the topology or routing policy changes.

2.6.4 Summary of Main Findings

Our analysis of UnivMon's performance shows that:

1. For a single router with 600KB of memory, we observe comparable median error rate values between UnivMon and OpenSketch, with a relative error gap $\leq 3.6\%$. The relative error decreases significantly with a growing application portfolio.

2. When comparing sensitivity to error and available memory, we observe

that UnivMon provides comparable accuracy with OpenSketch with similar, or smaller memory requirements.

3. The network-wide evaluation shows that UnivMon provides an even distribution of resources on each node while providing results with high accuracy.

2.7 Chapter Summary

In contrast to the status quo in flow monitoring that can offer generality or fidelity but not both simultaneously, UnivMon offers a dramatically different design point by leveraging recent theoretical advances in universal streaming. By delaying the binding of data plane primitives to specific (and unforeseen) monitoring UnivMon provides a truly software-defined monitoring approach that can fundamentally change network monitoring. We believe that this "minimality" of the UnivMon design will naturally motivate hardware vendors to invest time and resources to develop optimized hardware implementations, in the same way that a minimal data plane was key to get vendor buy-in for SDN [75].

Our work in this chapter takes UnivMon beyond just a theoretical curiosity and demonstrates a viable path toward a switch implementation and a network-wide monitoring abstraction. We also demonstrate that UnivMon is already very competitive w.r.t. custom solutions and that the trajectory (i.e., as the number of measurement tasks grows) is clearly biased in favor of UnivMon vs. custom solutions.

UnivMon already represents a substantial improvement over the status quo, That said, we identify several avenues for future work to further push the envelope. First, in terms of the data plane, while the feasibility of mapping UnivMon to P4 is promising and suggests a natural hardware mapping, we would like to further demonstrate an actual hardware implementation on both P4-like and other flow processing platforms. Second, in terms of the one-big-switch abstraction, we need to extend our coordination and sketching primitives to capture other classes of network-wide tasks that entail cross-OD-pair dependencies. Third, while the ILP is quite scalable for many reasonable sized topologies, we may need other approximation algorithms (e.g., via randomized rounding) to handle even larger topologies. Fourth, in terms of the various dimensions of interest to track, we currently maintain independent sketches; a natural question if we can avoid explicitly creating a sketch per dimension. Finally, while being application agnostic gives tremendous power, it might be useful to consider additional tailoring where operators may want the ability to adjust the granularity of the measurement to dynamically focus on sub-regions of interest [76].

Chapter 3

NitroSketch: Robust Sketch-based Monitoring in Software Switches

Traffic measurements are at the core of advanced network algorithms such as traffic engineering, fairness, load balancing, quality of service and intrusion detection [77, 78, 79, 80, 81, 82, 83]. While monitoring on dedicated switching hardware continues to be important, measurement capabilities are increasingly deployed inside *software switches* with the transition to virtualized deployments and "white-box" capabilities (e.g., Open vSwitch [84], Microsoft Hyper-V [85], Cisco Nexus 1000V [86], and FD.io VPP [87]).

Naturally, we want these measurement capabilities to run at high line rates and yet have a small resource footprint to avoid constraining the main switching functions and services that run atop the switch. In this respect, *sketching algorithms* are a promising approach for various metrics of interest such as per-flow frequency estimation [6, 4], Heavy Hitters [88, 9, 89], Hierarchical Heavy Hitters [90], Distinct flows [26], Frequency moments [91] and Change detection [26].

However, the packet processing performance of sketching algorithms on software switches is far from ideal [92, 93, 94]. This is not surprising as sketches are often optimized for (asymptotic) low memory requirement which is not the key bottleneck in software implementations.

Motivated by this, recent efforts seek to tackle of the performance issues of sketching algorithms in software, including SketchVisor [94], Hashtable-based monitoring [92, 93], and Randomized-HHH [90], but these efforts have to sacrifice the *generality* or *rigor* on one or more dimensions; i.e., either make strong assumptions about the traffic patterns at high load (e.g., SketchVisor relies on the skew of workload to achieve high accuracy and speedup, and cannot meet 10G line-rate under min-sized packets), or lose the theoretical guarantees (e.g., the Hashtable-based approach uses extensive memory to achieve high accuracy, and fails to achieve 10G line-rate when the workload is not skewed), or only apply to a specific measurement task (e.g., R-HHH achieves speedup only on measuring hierarchical heavy hitters).

This chapter is motivated by a simple question: can we *rigorously* improve the performance of sketches in software switches in *general* settings; i.e., (a) without compromising the worst-case theoretical guarantees; (b) without making assumptions about the traffic distributions; and (c) in a way that benefits a large number of sketching use cases and software platforms? To this end, we revisit the problem from first principles and systematically profile sketch implementations to identify the key performance bottlenecks. Basically, the sketch data structure is a (2D) array of counters. We find that existing sketches compute multiple hash functions while processing each packet (computational intensive), and exhibit random memory access pattern (make inefficient use of the cache if it is smaller than the sketch).

Our insight here is to reformulate the sketching problem to be optimized for software implementations. Specifically, we design sketching algorithms that require slightly more space and convergence time than the theoretical lower bounds, but run significantly faster in software. Intuitively, we allow sketches to process enough packets before fetching statistics from them, and we call this period “convergence”.

Based on this reformulation, we present **NitroSketch**, a framework to optimize the packet processing speed of sketches. The key idea is that we want the sketching algorithms to conduct fewer hash computations and counter updates while maintaining the same accuracy. We achieve this by combining sketch implementation with a sampling front-end. The front-end reduces the number of packets processed by a sampling probability of p. However, the memory increase to get comparable accuracy can be high if we naively use uniform sampling (section 3.4.2). Instead, we draw geometric samples[1] to decide what the next index of a counter update to the sketch would be (and thus determine how many packets to skip until the next update). We rigorously prove (section 3.4) that NitroSketch is accurate for a broad family of sketches that share a common structure as Count-Min Sketch [4] and Count Sketch [6].

We acknowledge that the performance speedups of NitroSketch come with two caveats. First, it consumes more memory than the original sketches for

[1]Geometric sampling also reduces the CPU requirement as we do not need to run a probabilistic computation every time — once a packet is sampled we compute the “index” of the next sampled packet, and we can avoid processing all intermediary packets.

the same error guarantee. Our theoretical analysis shows that for a given ϵL_1 guarantee[2] NitroSketch requires double space and for an ϵL_2 guarantee, it increases space by a factor of $O(p^{-1})$, where p is the sampling rate in the geometric pre-processing (section 3.4). In practice, this increase is acceptable in software switches (e.g., <6MB). Second, due to its sampling techniques, NitroSketch is only accurate after some convergence time (<10s).

We implement a NitroSketch prototype and integrate it with Open vSwitch-DPDK [84] and FD.io-VPP [87]. We port several canonical sketches—a recently proposed universal sketching framework (UnivMon [26]) and three application-specific sketches (Count-Min [4], Count Sketch [6], and K-ary Sketch [8]). We evaluate NitroSketch on OVS-DPDK and VPP using a range of traces [95, 96, 97] on commodity servers with 40GbE NICs. We show that sketches based on NitroSketch match the throughput of 40GbE OVS-DPDK, and have reduced CPU utilization and competitive accuracy after convergence. Compared to NetFlow/sFlow [2, 3], NitroSketch achieves better accuracy and uses significantly less memory when evaluated with the same sampling rate. When compared with SketchVisor [94], NitroSketch runs dramatically faster (>52Mpps vs. <7Mpps), or uses significantly less CPU to achieve the same throughput and yields more accurate results after convergence. Our in-memory benchmark also suggests that NitroSketch can keep up with future (faster than 40G) virtual switches.

[2]Here, $L_1 \triangleq \sum f_x$ and $L_2 \triangleq \sqrt{\sum f_x^2}$ refer to the first and second norms of the flow frequency vector of the packet trace, and $\epsilon > 0$.

3.1 Related Work and Motivation

In this section, we briefly survey *sketching algorithms* and explain why they are a promising alternative to traditional measurement approaches such as sampling and accurate measurements. We also show that the performance of existing sketching algorithms on software switch platforms is far from ideal and discuss recent efforts to alleviate these bottlenecks and their limitations.

Sketches as a measurement tool. Traditional network measurement tasks depend on uniformly sampled flows with statistics, e.g., NetFlow [2] and sFlow [3]. It is expensive to sample the traffic at a high sampling rate due to memory and computational limitations. However, low sampling rates cause failures in fine-grained measurement tasks. Sketches allow for memory-efficient network measurement systems as they reduce the memory usage of measurement tasks while maintaining guaranteed fidelity. Sketches are traditionally designed for hardware as high-speed memory on hardware comes at a premium [98, 99, 100]. Sketching algorithms are backed by rigorous theoretical proofs on error vs. memory trade-offs, and they make no assumptions on the workload.

Examples of measurement tasks that are supported by sketching algorithms include: (1) Heavy Hitter Detection to identify flows that consume more than a threshold α of the total capacity. Here the capacity can be packet-based (identified by flow keys) or volume-based (counted by byte counts). Concrete sketches include Count-Min Sketch [4], Space-saving [5], Count Sketch [6], and UnivMon [26]; (2) Change Detection to identify flows that

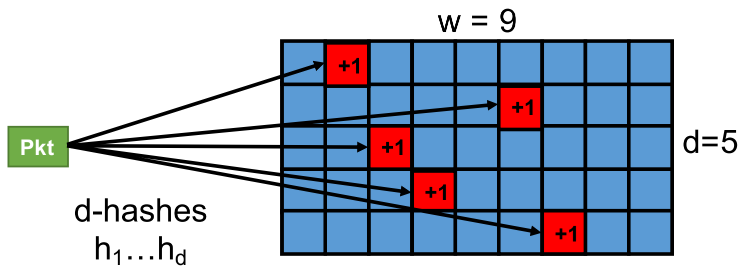


Figure 3.1: Count-Min Sketch Example.

contribute more than a threshold of the total capacity change over two consecutive time intervals using reversible k-ary Sketch [8, 9] and UnivMon [26]; (3) Cardinality Estimation to estimate the number of distinct flows in the traffic [7, 26]; (4) Entropy Estimation to estimate the entropy of different header distributions (e.g., Lall et al [10]); and (5) Attack Detection to identify a destination host that receives traffic from more than a threshold number of source hosts [11]. Instead of using a different sketch for each task, universal sketching [26] supports a broad spectrum of these tasks, while the measurement task is given only at query time. Indeed, NitroSketch supports the above sketches, including UnivMon [26], Count-Min [4], Count Sketch [6], and K-ary Sketch [8].

To illustrate the main idea of sketches, we can use Count-Min Sketch [4], where we maintain a $d \times w$ matrix of counters, as shown in Figure 3.1. On its *Count* procedure, the flow identifier (e.g., 5-tuple information) of each packet is hashed d times independently into one of the w counters of a row by $\{h_i : [n] \to [w]\}_{i \in [d]}$ hash functions, and then the corresponding counters are updated by its packet size. To obtain the size estimate of a given flow, we return *Min*, which is the minimum among the d hashed counters of the flow. Count-Min Sketch guarantees ϵL_1-error bound estimates when $d = \log_2 \delta^{-1}$

and $w = 2\epsilon^{-1}$ with $1 - \delta$ probability.

Sketch performance in software switches. The goal of conducting measurement tasks on software switches is to support line rate operations with high fidelity and low resource footprint. The low footprint is critical to ensure that other concurrent services can make maximal use of available resources, e.g., virtual machine instances.

To this end, we evaluate the I/O performance of various software sketches implemented atop OVS-DPDK (Table 3.1). We configure the memory allocation of the sketches based on the desired error targets. For Count-Min Sketch, we set 5 rows of 1000 counters; for UnivMon, its Count Sketch component has 5 rows of 10000 counters. We observe that sketches bring significant computation overhead to a single thread vanilla OVS-DPDK. Even the purportedly light-weight Count-Min Sketch [4] is far away from line rate processing. We see that existing sketches can neither meet 10G line-rate with a single CPU core under minimal-sized packets nor meet 40G line-rate under common case workloads (e.g., data center).

Therefore, while sketches optimize towards space efficiency, they incur significant per-packet computations. This is in conflict on the goal of software measurement as computation time becomes the bottleneck.

Proposed optimizations. Indeed, parallel efforts were made to address such bottlenecks. Alipourfard et al., [92, 93] suggest that simple hash tables will suffice on software switches since normal workloads are quite skewed and the management of larger L2/L3 cache in modern CPUs keeps improving. However, as we will see later, a hash table based approach is not robust for

Setting	Core/Thread	Pkt Sizes (Bytes)	Thru.(Mpps)
DPDK	1C/1T	64/714	22.93/6.95
OVS-DPDK	1C/1T	64/714	14.76/6.81
with Count-Min	1C/1T	64/714	7.29/6.29
with UnivMon	1C/1T	64/714	0.81/0.79

Table 3.1: I/O comparison on a single core.

two key reasons. First, hash tables consume excessive memory to maintain 100% or high accuracy when the number of flows in the traffic is large. The access pattern of a hash table highly depends on the skewness of the workload. In Section 3.2, we show that less-skewed traffic may cause L2/L3 cache misses and thus prevent line-rate processing. Further, even if we maintain the hash table entirely in the cache, we can still hardly achieve 10G line-rate using a single thread due to its per-packet access pattern.

SketchVisor [94] proposes a hybrid solution via a hybrid *normal path* and a *fast path* algorithm. The fast path algorithm is designed to take over the packets when the sketches in the normal path cannot handle packets at high speed. To track top k heavy hitters, the fast path, in essence, maintains a hash table of k entries and each entry has three different counters used for deciding an update or kick-out operation from the table. Even though this fast path algorithm runs faster than existing sketches, it still runs significantly slower than hash tables due to its more expensive per packet operations. The correctness of this hybrid approach crucially depends on the skewness of traffic, which may not hold under attacks or anomalous conditions. In terms of accuracy, the fast path algorithm implies worse error bounds than sketches, and there might be an issue when it handles the majority of traffic. Furthermore, their evaluation only reports numbers without DPDK and caps out at 1.7Mpps.[3]

3.2 Bottleneck Analysis

Before we propose any optimization, it is critical to identify what/where the performance bottlenecks are. Unfortunately, while the efforts above [92, 93, 94] measured the throughput limitations of existing sketches, they do not provide a systematic understanding of what the fundamental bottlenecks are. To this end, we tackle this problem from first principles and profile sketch implementations to identify the bottlenecks that ultimately inform our design innovations in the next sections. To stress-test the throughput of software sketch implementations, we use min-sized packets as a "worst case" scenario since all other real-world situations are less stressful. We have also tried on CAIDA traces whose average packet size is 717 bytes, and the performance bottlenecks are structurally similar.

Bottleneck analysis testbed. We set up a testbed with three commodity servers, each of which has an Intel Xeon E5-2620 v4 processor with 16 cores (256KB L2 cache per core, and 20MB L3 cache) and 128GB DDR4 memory. Among the servers, one acts as a switch and the other two as hosts. To alleviate the bottleneck of packet I/O, each server enables single-thread Data Plane Development Kit Polling Mode Driver (DPDK-PMD) [101] on each 40G port.

For vanilla DPDK, it does not saturate a single CPU core to transmit packets, but the bottleneck in the media access control layer of an XL710 network

[3]This is far from 10G line-rate for 64B packets, and we are unsure of how SketchVisor will perform when DPDK is enabled.

Func/Call Stack	Description	CPU Time
`miniflow_extract`	retrieve miniflow info	7.808s
`i40e_recv_pkts_vec`	dpdk packet recv	6.188s
`__memcmp_sse_4_1`	flow entry memory cmp	5.93s
`xmit_fixed_burst_vec`	dpdk packet send	5.836s
`emc_loopup`	Lookup emc table	2.449s
All others	other function calls	21.866s
Total	OVS-DPDK	50.076s

Table 3.2: CPU hotspots in a OVS-DPDK vswitchd thread.

interface card (NIC) limits the performance (see Intel XL710 Datasheet [102]). We instrument the system using Intel VTune Amplifier [103] to analyze the vswitchd process shown in Table 3.2.

Observation 1: ***In a single thread OVS-DPDK, there is small CPU headroom for sketching algorithms.***

In a one-minute profiling with a 1ms sampling interval, the total CPU time is 50.076 sec (83.86%). We can claim from the profiling results that miniflow extraction bottlenecks OVS-DPDK in the exact match cache (EMC) processing and the packet transmission path in the DPDK PMD thread. For a single OVS-DPDK vswitchd process, only a small limited portion of CPU can be used for running sketching algorithms.

Observation 2: ***Cache residency is crucial for any existing sketches to achieve high-speed or line-rate due to sketches' per-packet access pattern.***

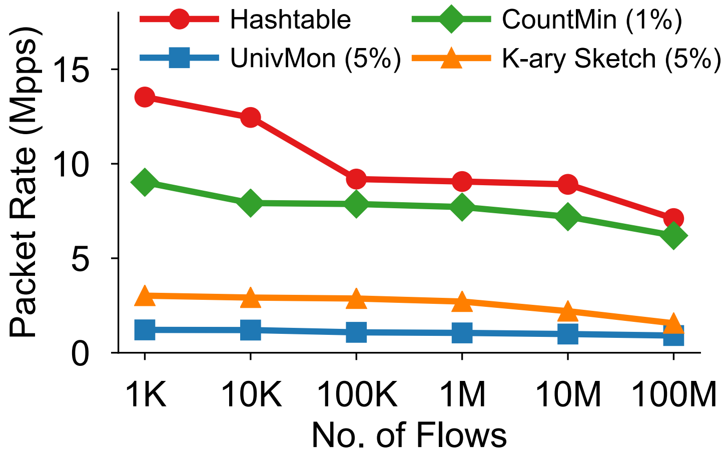


Figure 3.2: Packet rate of different data structures using random 64B packets. (Setting: single thread OVS-DPDK.)

Hash tables can be considered as the simplest "sketch" structure we can use to preserve the entire traffic's flow size distribution. Despite their huge memory usage, they incur fewer computations than sketches, i.e., per packet access pattern is light-weight with three operations: a hash computation, a counter update, and a flow key copy. Prior work [92, 93] evaluated simple hash tables (on Click Router and DPDK) against more complex sketches and observed superior throughput performance. We evaluate simple hash tables as the only measurement component integrated with OVS-DPDK.

In Figure 3.2, we evaluate hash tables and sketching algorithms with a varying number of flows in the network traffic. We implement a packet generator using a Zipf distribution generator and use the MoonGen [104] API to emulate the different number of flows in the traffic. For the hash table, the number of flows equals the number of entries in the table. For other sketches, we use calculated theoretical amount of memory to meet 1% or 5% error targets. For instance, UnivMon uses 410KB on 1M flows while Count-Min

Memory Usage	IPC	L2 Hit	L3 Hit	DRAM (per sec)
0.08 MB	2.62	0.45	1.00	<200
0.8 MB	2.52	0.42	1.00	<100
8 MB	1.32	0.03	1.00	<100
80 MB	0.71	0.00	0.25	8383K

Table 3.3: Cache and DRAM access for simple hash table.

uses 20KB.

When the number of entries that a hash table holds is small, it can nearly meet the 10G line rate with minimal sized packets (13.1Mpps out of 14.88Mpps). If the number of flows is small, a tiny hash table can beat other sketches in terms of processing speed since sketches are far from line rates. However, with the increasing number of flows, its throughput gradually moves away from the line speed. We analyzed hash table's cache residency and report in Table 3.3. Together with Figure 3.2, we confirm that cache residency is crucial for existing sketches to achieve line rate.

Observation 3: ***Even when a sketch fits into the cache, it cannot achieve line-rate due to its per-packet hash operations, data structure maintenance, and counter updates.***

Taking the recently proposed UnivMon [26] as an example, we profiled their per packet function-call hotspots. With UnivMon [26], a single sketch simultaneously collects different types of traffic statistics. At a high level, UnivMon needs to maintain a set of heavy hitter monitoring modules, which comprises $\Omega(\log \delta^{-1})$ (for δ failure probability) hash operations and priority

Func/Call Stack	Description	CPU Time
`xxhash32`	hash computations	36.79%
`heap_find`	heap operation	11.31%
`__memcpy`	memcpy and counter update	10.91%
`univmon_proc`	pkt copy and cache	9.13%
`heapify`	heap maintenance	5.12%
`miniflow_extract`	retrieve miniflow info	2.91%
`recv_pkts_vecs`	dpdk packet recv	2.73%

Table 3.4: CPU hotspots on UnivMon with OVS-DPDK.

queue updates for each packet. Thus it runs slow in software switches. We instrument a single thread OVS-DPDK with UnivMon integrated to pinpoint the significant performance bottlenecks.

From the profiling results in Table 3.4, we can pinpoint a couple of major performance bottlenecks: multiple calls to the hash function; excessive calls to locate and update the item in the heap structure of UnivMon. Besides, the large number of memory copies implies significant overheads. Similarly, for less complex sketches (e.g., Count Sketch), the performance bottlenecks are structurally the same.

Summary and key implications. Based on the above analysis, we argue that several fundamental bottlenecks prevent sketching algorithms from achieving line rate processing in software. The data structures in the measurement algorithms with per-packet random access pattern cannot be too large; when they are sized larger than the L3 cache performance drops. Per-flow data structures, such as hash tables, fall short of line rate when they are not cache-resident. Sketches as a memory efficient alternative can be sized to fit the L3 cache, but they are unable to meet line rate, due to the expensive hash computations and memory accesses per packet.

Based on these observations, we revisit prior work that tried to address the performance of sketches and understand why they do not adequately tackle this problem. We see that they are stuck in a fundamental dilemma: (1) per-flow based structures need to fit into cache to achieve 10G line rate but they are in practice too large to retain cache residency; (2) sketches can fit into cache easily but have too complicated access patterns to attain line-rate speeds. SketchVisor [94] partially addressed the bottlenecks by proposing an improved Misra-Gries algorithm [88] to simplify the per-packet access pattern, but this comes at the cost of accuracy, generality, and robustness. Moreover, even the fast path still requires per-packet hashes and updates and can suffer under worst-case workloads. As we will see, our work addresses these fundamental bottlenecks by reformulating the problem to tolerate a small increase in memory footprint and latency.

3.3 NitroSketch Framework

This section describes the design of our framework. We start by outlining the key ideas, and then define the NitroSketch algorithm and its variant. We conclude by explaining how our general approach applies to other sketch algorithms.

3.3.1 Key Idea

At a high level, answering the following is the driving intuition behind our design: *can we trade-off a small increase in the memory footprint and measurement*

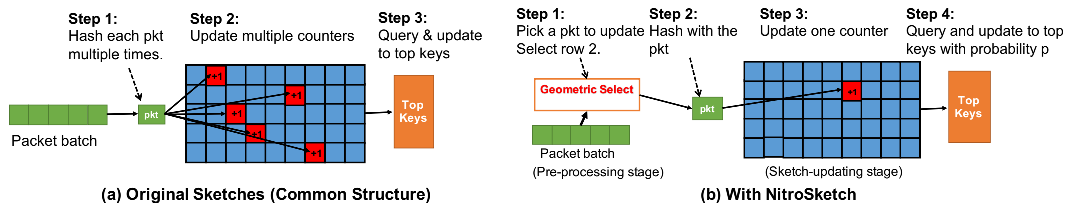


Figure 3.3: (a) Before using NitroSketch, each packet goes through multiple hash computations (e.g., $O(\log \delta^{-1})$), update multiple counters, and query and update to a top-k HH storage (e.g., heap). (b) After applying NitroSketch, only a small portion of packets (say 5% by geometric sampling) need to go through $O(1)$ hash computations, update to one row of counters (instead of all rows) and occasionally to a top-k structure. Therefore, the CPU cost is significantly reduced.

latency for a significant reduction in the CPU footprint, without compromising the error guarantees (i.e., for arbitrary traffic)?

As we show, this problem reformulation produces significant speedups even for complicated sketches such as recently proposed UnivMon [26], or the hardest case of min-sized packets. We observe that many sketches share a common structure but with different operations on that structure. This allows us to design a sampling front-end that applies to these sketches, as depicted in Figure 3.3.

Common structure of sketches. Many sketches, including popular Count Sketch [6] and Count-Min Sketch [4], are derived from the seminal AMS sketch [91]. At a high level, they project high dimensional data into a lower dimensional counter matrix while preserving some of the properties of the original data with high probability.

These sketches can be conceptually viewed as a counter matrix where for each stream input we independently update a subset of the counters in multiple rows, as shown in Figure 3.3(a). The variations between algorithms manifest in different choices of hash operations and counter configurations. Indeed, even more, general and complex sketches like UnivMon [26] are composed of multiple instances of such basic sketches. For example, Count Sketch requires $O(\log \frac{1}{\delta})$ rows of counters and two hash functions per row, where δ is the failure probability. It updates a single counter in each row, requiring $\Omega(\log \frac{1}{\delta})$ independent hash computations per packet. Further, in some cases, we also require an additional data structure to list the heavy hitters which further increases per-packet processing. Count-Min Sketch follows a very similar structure and differs in the number of counters per row and the "action" performed on each counter. That is, Count-Sketch increments and decrements the counters and Count-Min Sketch only increments them.

NitroSketch workflow. NitroSketch uses a two-stage procedure to process packets: a *pre-processing* stage and a *sketch-updating* stage, as shown in Figure 3.3(b).

- **Pre-processing stage:** In this stage, packets arrive as batches (e.g., from DPDK Polling Mode Driver). NitroSketch selects a sample of the packets and sends them to the sketch-updating stage, while ignoring the remaining packets. We consider packets in this stage as a geometric distribution, and only the packets who "succeed" in the geometric trials proceed to update in the sketch. To this extent, the majority of packets (say 95%) are "skipped" from sketching operations, and we discuss this stage in Section 3.3.2.

- **Sketch-updating stage:** Selected packets in the pre-processing stage are sent to the Sketch-updating stage to update a single counter on a specified row. This minimizes the processing overheads of selected packets.

Convergence time: Intuitively, NitroSketch skips most of the packets and performs a single counter update to the rest. This approach only works once the number of processed packets is large enough. The term “convergence time” refers to the number of packets needed to converge. We provide an upper bound in Section 3.4 and evaluate it with real workloads in Section 4.5.

3.3.2 NitroSketch Algorithms

We now introduce NitroSketch and explain how it addresses the performance bottlenecks of existing sketches. We use NitroSketch with Count Sketch (L_2 guarantee) as the illustration example. We start by showing that NitroSketch is better than a Count-Sketch with a single row.

Naively, we could simply use a single row Count-Sketch structure. We call this suggestion One-Row-Count-Sketch. This reduces the per-packet hash operations to $O(1)$ but comes with two drawbacks. First, it requires $O(\epsilon^{-2}\delta^{-1})$ space [6], which is exponentially more than the $O(\epsilon^{-2}\log\delta^{-1})$ requirement of Count-Sketch. Second, we are still required to perform a single per-packet hash computation followed by a random memory access which is not always feasible. Instead, NitroSketch amortizes the per-packet hash computations to $o(1)$. Further, it also utilize the cache in an efficient manner as the majority of operations are counting the number of packets to be ignored until the next sample.

The NitroSketch Algorithm. We maintain a similar data structure to sketches (e.g., Count-Min Sketch or Count Sketch), but update and query counters differently. Rather than updating counters in all rows for every packet, we

Algorithm 3 NitroSketch Data-plane

```
 Input: Packet stream $D(m,n) = a_0, \ldots, a_{m-1} \in [n]^{[m]}$
 1: Generate pairwise independent hash functions:
 2: $\{h_i{:}[n] \to [w]; \quad g_i{:}[n] \to \{-1,+1\}$ or $\{+1\}\}_{i \in [d]}$
 3: $r \leftarrow (-1), q \leftarrow 0$
 4: $j \leftarrow 0$                                  ▷ The next packet to select
 5: while $j \leq m$ do                              ▷ Continue to process packets
 6:     FastUpdate()
 7: procedure FASTUPDATE
 8:     $r$ += $Geo(p)$                               ▷ Geometric variable
 9:     $j$ += $\lfloor r/d \rfloor$                  ▷ Skip packets if needed
10:     $r \leftarrow r \mod d$                       ▷ The row to update
11:     $C_{r,h_r(a_j)}$ += $p^{-1} \cdot g_r(a_j)$
12: procedure QUERY($x$)
13:     return $\text{median}_{i \in [d]}\{C_{i,h_i(x)} \cdot g_i(x)\}$
```

update each row *independently* with probability p. To compensate for the sampling, each counter update changes its value by p^{-1}. When querying a flow, we calculate the *median* of its estimations in all rows, same as in the original Count Sketch. This poses a trade-off — the sketch becomes faster but requires more space as the probability p decreases.

A straightforward realization of the above algorithm needs to make an independent coin flip for every row when processing each packet, which incurs one additional operation per row in the sketch. Instead, we use a geometric sampling technique to directly calculate how many packets to ignore before the next sampled packet, and which row to select next. This is logically equivalent to per-row coin flip and allows us to "skip" multiple packets by a single uniform variate once a packet is selected.

Specifically, in Algorithm 3, we present this procedure as FASTUPDATE

(Lines 7-11). The procedure draws a geometric variable $X \sim Geo(p)$ to determine how many rows to skip until the next sampled packet. This implementation requires $o(1)$ hash computations per-packet, a significant speedup when comparing to hash table and One-Row-Count-Sketch. The evaluation in section 4.5 shows that NitroSketch reaches the limit of 40G OVS-DPDK when setting a small sampling probability $p \ll d^{-1}$ where d is the number of rows in the sketch (e.g., $d = 5$ and $p = 0.01$).

Maintaining Top-*k* Heavy-Hitters. Similarly to Count Sketch, top-k heavy hitters are maintained in a heap of size k. In Count Sketch, when a packet arrives, we need to query its flow counter to check if the minimal item should be replaced.

Performing this update for each incoming packet requires $\Omega(d)$ hash computations. Therefore, we only update the heap with probability p for each packet. As $p = o(d^{-1})$, this only adds $o(1)$ hash computations per packet. In practice, we use a geometric variable and generate one uniform variate for every *query* instead of for each packet.

NitroSketch with Delayed Sampling (DS-NitroSketch). NitroSketch only provides accuracy guarantees after a certain number of packets were processed, which poses a challenge to some applications and sketch based measurement methods. Some applications require measurement results to be available at all time and may not want to wait.

We design a DS-NitroSketch that is always accurate, but only provides an acceleration once enough packets were processed (converged). Intuitively, DS-NitroSketch starts as a Count Sketch (in which all rows are updated per packet).

Algorithm 4 NitroSketch with Delayed Sampling

```
Input: Packet stream $D(m,n) = a_0,\dots,a_{m-1} \in [n]^{[m]}$
1: Generate pairwise independent hash functions:
2: $\{h_i:[n] \to [w]; \quad g_i:[n] \to \{-1,+1\}\}_{i\in[d]}$
3: $converged \leftarrow 0, r \leftarrow (-1), q \leftarrow 0, j \leftarrow 0$
4: $T \leftarrow 121(1+\epsilon\sqrt{p})\epsilon^{-4}p^{-2}$                ▷ Threshold for sampling
5: while $j \le m$ do                                  ▷ Continue to process packets
6:     if $converged$ then FASTUPDATE()
7:     else NORMALUPDATE()
8: procedure NORMALUPDATE
9:     $j \leftarrow j+1$
10:    for $r = 0,\dots,d-1$ do                          ▷ A Count Sketch update
11:        $C_{r,h_r(a_j)} += g_r(a_j)$
12:    if $(j \mod Q) = 0 \wedge (\text{median}_{i\in[d]} \sum_{y=1}^{w} C_{i,y}^2) > T$ then
13:        $converged \leftarrow 1$
```

It periodically estimates the L_2 norm to determine when it can justify switching to NitroSketch. That way, accuracy guarantee is preserved throughput the measurement but speedup is only achieved once converged. The pseudocode of DS-NitroSketch is given in Algorithm 4. Observe that we now update rows with varying probabilities (initially as 1 and then p), we update the counters with the inverse sampling probability (initially 1 and then p^{-1}).

Formally, the sum of squared counters in each row i, serves as a $(1+\epsilon\sqrt{p})$-multiplicative estimator for the stream's L_2^2 with probability 0.5, and the rows' median with a probability of $1-\delta$. We perform this computation once per Q (e.g., $Q = 1000$) packets which reduces the overheads and ensures that sampling starts at most Q packets late. We use the Union Bound to get an overall error probability of 2δ – with probability $\le \delta$ we start sampling too early and with probability $\le \delta$ the sketch's error exceeds ϵL_2.

3.3.3 Interface to Other Sketching Algorithms

While NitroSketch and DS-NitroSketch are introduced with Count Sketch, a variety of other sketching algorithms can leverage the same key ideas. This requires setting an error budget and some minor modifications as explained below:

Error budget ϵ. For every applicable sketch to NitroSketch, the pre-processing stage is the same and provided by the system. A user can specify an error target ϵ for some application-specific sketch or a general-purpose sketch, and NitroSketch will configure the sampling rate in the pre-processing stage and the number of counters in its sketch structure in the second stage, based on the analysis in Section 3.4.

Minor modifications to existing sketches. To benefit from NitroSketch's acceleration, we make some slight adjustments of the sketches. To be specific, the pre-processing state is the same for all sketches, but users may need to modify the sketch-updating stage. For instance, original Count Sketch needs two sets of $\log \delta^{-1}$ (where δ is the failure probability) hash functions and the logic requires to update counters in all $\log \delta^{-1}$ rows of the sketch, as illustrated in Figure 3.3(a). To adopt NitroSketch (Figure 3.3(b)), we need to change this logic to update one counter based on the "index" (next row to update) from the pre-processing stage.

3.4 Analysis of NitroSketch

We now show the theoretical guarantees of NitroSketch. We consider two variants; first, combining the Count-Min Sketch with geometric sampling for achieving an ϵL_1 guarantee, and second using NitroSketch for an ϵL_2 approximation. Here, $L_k \triangleq \sqrt[k]{F_k} = \sqrt[k]{\sum_{x \in \mathcal{U}} f_x^k}$ is the k-th norm of the frequency vector and $\mathcal{U}$ is the set of all possible flows (e.g., all 2^{32} possible source IPs). Specifically, L_1 is simply the number of packets in the measurement. We note that computing an L_k approximation for $k > 2$ cannot be done using polylogarithmic space (in $|\mathcal{U}|$) and is considered infeasible [91].

The following theorem follows from the analysis in [90].

Theorem 3.4.1. Let $d \triangleq \log_2 \delta^{-1}$ and $w \triangleq 4\epsilon^{-1}$. For streams in which $L_1 \geq c \cdot \left(\epsilon^{-2} p^{-1} \sqrt{\log \delta^{-1}}\right)$ for a sufficiently large constant c, Count-Min Sketch in which every packet increases the counter of each row i independently with probability p satisfies: $\Pr\left[|\widehat{f_x} - f_x| \geq \epsilon L_1\right] \leq \delta$ where f_x is the actual frequency of flow x, and $\widehat{f_x}$ is the value returned by calling $Query(x)$ in Algorithm 3.

Next, we state Theorem 3.4.2 and Theorem 3.4.3 that establish the correctness of NitroSketch and DS-NitroSketch.

Theorem 3.4.2. Let $w = 8\epsilon^{-2}p^{-1}, d = O(\log \delta^{-1})$. NitroSketch requires $O(\epsilon^{-2}p^{-1}\log \delta^{-1})$ space, operates in amortized $O(1 + dp)$ time (constant for $p = O(1/d)$), and provides the following guarantee: $\Pr\left[\left|f_x - \widehat{f_x}\right| > \epsilon L_2\right] \leq \delta$ for streams in which $L_2 \geq 8\epsilon^{-2}p^{-1}$.

Theorem 3.4.3. Let $w = 11\epsilon^{-2}p^{-1}$ and $d = O(\log \delta^{-1})$; DS-NitroSketch provides an estimator $\widehat{f_x}$ that satisfies: $\Pr\left[|\widehat{f_x} - f_x| > \epsilon L_2\right] < 2\delta$.

3.4.1 Interpretation of Main Theorems

Intuitively, NitroSketch trades space for throughput. For example, it requires an $O(p^{-1})$ factor more space for the same accuracy when compared to Count Sketch.

Compared to One-Row-Count-Sketch, our solution provides faster updates and has lower space requirements. The improvement in update time comes from reducing the number of hash computations. While One-Row-Count-Sketch computes two hashes per packet, NitroSketch only does so for each *sampled* row. As the expected number of sampled rows per packet is $dp = o(1)$, NitroSketch significantly reduces the processing overheads. Space-wise, NitroSketch only requires $O(\epsilon^{-2}\log \delta^{-1}p^{-1})$ space compared to the $O(\epsilon^{-2}\delta^{-1})$ memory of One-Row-Count-Sketch and is therefore more cache resident. For example, we can set $p = d^{-2} = O(\log \delta^{-2})$ to get a space of $O(\epsilon^{-2}\log \delta^{-3})$. Further, One-Row-Count-Sketch makes two hash computations per packet while NitroSketch just $o(1)$.

Convergence time: For example, the first 10M source IPs of the CAIDA 2016 [95] trace has a second norm of $L_2 \approx 1.28 \cdot 10^6$ while 100M packets gives $L_2 \approx 1.03 \cdot 10^7$. This means that if $p = 1\%$, we get guaranteed convergence for $\epsilon \geq 2.5\%$ after 10M packets and $\epsilon \geq 0.88\%$ after 100M. In practice, we observe that the error is lower which suggests that our analysis is just an upper bound and one can use smaller ϵ values as well. Finally, we note that extending the row sizes further allow faster convergence of the algorithm.

Alternatively, DS-NitroSketch has no convergence time, but does not provide an acceleration over Count-Sketch before the L_2 norm is large enough. Therefore, DS-NitroSketch can accelerate UnivMon while providing provable accuracy guarantees throughout the measurement. As UnivMon is comprised of multiple L_2 sketches, those whose L_2 is large enough would converge and act like NitroSketch; the remaining sketches would not converge and would act like Count-Sketches.

3.4.2 Comparison to Uniform Sampling

A natural question is whether we gain the same performance boost by feeding a sub-sampled stream into a sketch. Indeed, one can expect to achieve speedup by considering fewer items (and thus, reducing the number of hash computations and memory accesses). The question is how to combine the two to get the same error guarantee, and how much to increase the space to make up for the sampling error.

In Section 3.4.5, we analyze the option of sampling each packet independently with probability p and feeding it into Count Sketch. We stress that the expected number of hash computations, memory accesses and uniform variable generations is similar to that of NitroSketch with the same parameter p (when using geometric sampling). We show that uniform samples are inferior to NitroSketch. To provide the same L_2 guarantee with probability $1-\delta$, the Count Sketch used for processing the uniform sample needs:

$\Omega\left(\epsilon^{-2}p^{-1}\log\delta^{-1}+\epsilon^{-2}p^{-1.5}m^{-0.5}\log^{1.5}\delta^{-1}\right)$ counters. In contrast, NitroSketch requires only $O\left(\epsilon^{-2}p^{-1}\log\delta^{-1}\right)$ counters. Here, m is the number of packets. This also gives an alternative perspective on the result - given the same space, the convergence time for uniform sampling is higher and depends on the error probability δ. We also empirically compare our solution against uniform sampling methods such as sFlow and Netflow; as depicted in Figure 3.13, our approach outperforms these solutions even if the sample is thoroughly analyzed and not sketched.

3.4.3 Proof of Theorem 3.4.2

We consider the sequence of packets that was sampled for each of the rows. That is, let $S_i \subseteq S$ be the subset of packets that updated row i (for $i \in \{1, \dots d\}$). Further, we denote by $f_{i,x} \triangleq |\{j \mid (x_j \in S_i) \wedge (x_j = x)\}|$ the frequency of x within S_i. That is, $f_{i,x}$ the number of times a packet arrived from flow x and we updated row i. Let $L_2 \triangleq \sqrt{\sum_{x\in\mathcal{U}} f_x^2}$ denote the second norm of the frequency vector of S and similarly $L_{2,i} \triangleq \sqrt{\sum_{x\in\mathcal{U}} f_{i,x}^2}$ denote that of S_i. Clearly, we have $L_{2,i} \le L_2$ for any row $i \in \{1, \dots, d\}$.

We proceed with a simple lemma that bounds $\mathbb{E}\left[L_{2,i}^2\right]$ as a function of L_2^2. Observe that $f_{i,x} \sim \text{Bin}(f_x, p)$ and thus $\text{Var}[f_{i,x}] = f_x p(1-p)$ and $\mathbb{E}[f_{i,x}] = f_x p$.

Lemma 3.4.4. $\mathbb{E}\left[L_{2,i}^2\right] \le 2pL_2^2$.

Proof.

$$\begin{aligned}\mathbb{E}\left[L_{2,i}^2\right] = \sum_{x\in\mathcal{U}} \mathbb{E}\left[f_{i,x}^2\right] &= \sum_{x\in\mathcal{U}} \mathrm{Var}[f_{i,x}] + (\mathbb{E}[f_{i,x}])^2 \\ &= \sum_{x\in\mathcal{U}} f_x p(1-p) + (f_x p)^2 \le \sum_{x\in\mathcal{U}} 2pf_x^2 = 2pL_2^2. \quad \square\end{aligned}$$

Next, we bound the variance of $\left(C_{i,h_i(x)}g_i(x) - p^{-1}f_{i,x}\right)$ – the noise that flows other than x add to its counter on the i'th row.

Lemma 3.4.5. $\mathrm{Var}\left[C_{i,h_i(x)}g_i(x) - p^{-1}f_{i,x}\right] \le 2p^{-1}L_2^2/w.$

Proof. First, observe that

$$C_{i,h_i(x)} = p^{-1} \sum_{x'\in\mathcal{U}|h_i(x)=h_i(x')} f_{i,x'}g_i(x').$$

That is, the value of x's counter, $C_{i,h_i(x)}$, is affected by all $x' \in \mathcal{U}$ such that $h_i(x) = h_i(x')$, and the contribution of each such x' is $p^{-1}f_{i,x'}g_i(x')$.

Next, notice that since $\mathbb{E}\left[g_i(x')\right] = 0$ and as g_i is two-way independent, we have that

$$\begin{aligned}\mathbb{E}\left[C_{i,h_i(x)} \cdot g_i(x)\right] &= p^{-1} \sum_{x'\in\mathcal{U}:h_i(x')=h_i(x)} \mathbb{E}\left[f_{i,x'} \cdot g_i(x') \cdot g_i(x)\right] \\ &= p^{-1}\mathbb{E}[f_{i,x}] = f_x.\end{aligned}$$

Now, as h_i is pairwise independent, we have that for any $x' \in \mathcal{U} \setminus \{x\}$:

$\Pr[h_i(x) = h_i(x')] = 1/w$. We are now ready to prove the lemma:

$$\begin{aligned}
\operatorname{Var}\left[C_{i,h_i(x)}g_i(x) - p^{-1}f_{i,x}\right] &= \mathbb{E}\left[(C_{i,h_i(x)}g_i(x) - p^{-1}f_{i,x})^2\right] \\
&= \mathbb{E}\left[(C_{i,h_i(x)}g_i(x))^2 - 2p^{-1}C_{i,h_i(x)}g_i(x)f_{i,x} + p^{-2}f_{i,x}^2\right] \\
&= \mathbb{E}\Bigg[\left(p^{-1}\sum_{x'\in\mathcal{U}|h_i(x)=h_i(x')} f_{i,x'}g_i(x')g_i(x)\right)^2 \\
&\quad - 2p^{-1}\left(p^{-1}\sum_{x'\in\mathcal{U}|h_i(x)=h_i(x')} f_{i,x'}g_i(x')g_i(x)f_{i,x}\right) + p^{-2}f_{i,x}^2\Bigg] \\
&= p^{-2}\mathbb{E}\left[\left(\sum_{x'\in\mathcal{U}|h_i(x)=h_i(x')} f_{i,x'}^2\right) - f_{i,x}^2\right] \\
&= p^{-2}\mathbb{E}\left[\sum_{x'\in\mathcal{U}\setminus\{x\}|h_i(x)=h_i(x')} f_{i,x'}^2\right] \le p^{-2}\mathbb{E}[L_{2,i}^2]/w \le 2p^{-1}L_2^2/w,
\end{aligned}$$

where the last inequality is correct per Lemma 3.4.4. □

We denote $A \equiv C_{i,h_i(x)}g_i(x)$ and $B \equiv p^{-1}f_{i,x}$ (since $\operatorname{Var}[f_{i,x}] = f_x p(1-p)$, we have $\operatorname{Var}[B] = f_x p^{-1}(1-p)$). Our goal is to bound the variance of A and use Chebyshev's inequality.

Observe that

$$A - B = \left(p^{-1}\sum_{x'\in\mathcal{U}|h_i(x)=h_i(x')} f_{i,x'}g_i(x')g_i(x)\right) - p^{-1}f_{i,x}$$

$$= p^{-1}\sum_{x'\in\mathcal{U}\setminus\{x\}|h_i(x)=h_i(x')} f_{i,x'}g_i(x')g_i(x)$$

is independent from B, and thus

$$\text{VAR}\,[A] = \text{VAR}\,[(A-B)+B] = \text{VAR}\,[A-B] + \text{VAR}\,[B]$$

$$\leq p^{-2}E[L_{2,i}^2]/w + f_x p^{-1}(1-p) \leq 2p^{-1}L_2^2/w + f_x p^{-1}.$$

We denote by $\widehat{f_x(i)} \triangleq A = C_{i,h_i(x)}g_i(x)$ the estimation for x's frequency provided by the i'th row. Then according to Chebyshev's inequality:

$$\Pr\left[|\widehat{f_x(i)} - f_x| \geq \epsilon L_2\right]$$

$$= \Pr\left[|C_{i,h_i(x)}g_i(x) - \mathbb{E}\left[C_{i,h_i(x)}g_i(x)\right]| \geq \epsilon L_2\right]$$

$$= \Pr\left[|A - \mathbb{E}\,[A]\,| \geq \epsilon L_2\right]$$

$$\leq \Pr\left[|A - \mathbb{E}\,[A]\,| \geq \frac{\sigma(A)\cdot \epsilon L_2}{\sqrt{2p^{-1}L_2^2/w + f_x p^{-1}}}\right]$$

$$\leq \frac{2p^{-1}L_2^2/w + f_x p^{-1}}{(\epsilon L_2)^2} = \frac{2L_2^2/w + f_x}{p(\epsilon L_2)^2}$$

$$= \frac{2/w}{p\epsilon^2} + \frac{f_x}{p(\epsilon L_2)^2} \leq \frac{2/w}{p\epsilon^2} + \frac{1}{p\epsilon^2 L_2}.$$

We want a constant probability of the error exceeding ϵL_2 in each row, so that the median of the rows will be correct with probability $1-\delta$. Therefore, by demanding $L_2 \geq 8p^{-1}\epsilon^{-2}$ and $w \geq 8p^{-1}\epsilon^{-2}$ we get that the error probability is

$$\Pr\left[|\widehat{f_{x,i}} - f_x| \geq \epsilon L_2\right] \leq \frac{2/w}{p\epsilon^2} + \frac{1}{p\epsilon^2 L_2} \leq 3/8.$$

As the $d = O(\log \delta^{-1})$ rows are independent, the algorithm's estimate,

$\widehat{f_x} = \text{median}_{i\in\{1,\dots d\}} \widehat{f_x(i)}$, will be correct with a probability of at least $1 - \delta$ using a standard Chernoff bound. Specifically, we have established the correctness of Theorem 3.4.2.

3.4.4 Analysis of DS-NitroSketch

We now formally analyze the accuracy guarantees of DS-NitroSketch (Algorithm 4). We start with Lemma 3.4.6 that shows that once Algorithm 4 converges (see Line 12), the L_2 is large enough to justify sampling with probability p.

Lemma 3.4.6. If *converged* $= 1$ then

$$\Pr\left[L_2 \geq 11\epsilon^{-2}p^{-1}\right] \geq 1 - \delta.$$

Proof. Since L_2 grows monotonically with the number of packets, it is enough to show that the condition of Line 12 implies the lower bound on the L_2 value. Namely, we assume that

$$\text{median}_{i\in[d]} \sum_{y=1}^{w} C_{i,y}^2 > 121(1+\epsilon\sqrt{p})\epsilon^{-4}p^{-2}. \tag{3.4.1}$$

It is known that given a Count Sketch that is configured for a (ϵ', δ)-guarantee, it is possible to compute a $(1+\epsilon')$-approximation of the L_2 with probability $1-\delta$ [91]. Specifically, as throughout the processing of $\overline{S}$ our sketch is identical to a Count Sketch (for $\epsilon' = \epsilon\sqrt{p}$), we have that:

$$\Pr\left[\left|\left(\text{median}_{i\in[d]} \sum_{y=1}^{w} C_{i,y}^2\right) - L_2^2\right| > \epsilon' L_2^2\right] \leq \delta.$$

Combining this with (3.4.1), the lemma follows. □

In DS-NitroSketch, there are $d = O(\log \delta^{-1})$ rows, each having $w = 11\epsilon^{-2}p^{-1}$ counters. As long as the sketch has not 'converged' (see Line 12), it is indistinguishable to a Count Sketch [6] with a guarantee of $\epsilon' \triangleq \epsilon\sqrt{p}$. Thus, given a flow x, if $\mathit{converged} = 0$ then Algorithm 4 guarantees:

$$\Pr\left[|\widehat{f_x} - f_x| \le \epsilon L_2'\right] \le \delta.$$

As $\epsilon' = \epsilon\sqrt{p} \le \epsilon$ the algorithm provides the desired accuracy guarantee prior to convergence. Henceforth, we assume that $\mathit{converged} = 1$ and show that the error is still at most ϵL_2.

We denote by u the index of the packet that during its processing the condition in Line 12 was satisfied and the sketch converged. That is, packets $a_i, \ldots, a_u$ were processed using a NORMALUPDATE, while $a_{u+1}, \ldots, a_m$ followed a FASTUPDATE. Further, we denote by $\overline{S} \triangleq a_1, \ldots, a_u$ the substream of the first u packets, by $\ddot{S} \triangleq a_{u+1}, \ldots, a_m$ the remaining substream, and for a flow x we use $\overline{f_x}$ and $\ddot{f_x}$ to denote its frequency in $\overline{S}$ and $\ddot{S}$. Note that the overall frequent of x is $f_x = \overline{f_x} + \ddot{f_x}$. Additionally, we denote the number of times a packet that belongs to a flow x in $\ddot{S}$ was sampled by the i'th row as $\ddot{f}_{x,i}$. Similarly to the analysis of NitroSketch, we first analyze the guarantee provided by a single row. Namely, fix some flow $x \in \mathcal{U}$ and a row $i \in \{1, \ldots, d\}$; the counter associated with x on this row is $C_{i,h_i(x)}$. Observe that we can express the value of the i'th estimator as:

$$C_{i,h_i(x)}g_i(x) \quad = \quad \sum_{y:h_i(y)=h_i(x)} \overline{f_y}g_i(x)g_i(y) + p^{-1} \cdot \sum_{y:h_i(y)=h_i(x)} \ddot{f}_{y,i}g_i(x)g_i(y). \tag{3.4.2}$$

That is, every flow y that is mapped to the same counter as x (i.e., $h_i(y) = h_i(x)$) changes the estimation by $\overline{f_y}g_i(x)g_i(y) + p^{-1}\ddot{f}_{y,i}g_i(x)g_i(y)$ – every packet of y in $\overline{S}$ surely adds $g_i(y)$ to the counter (Algorithm 3, Line 11), while every *sampled* packet in $\ddot{S}$ modifies the counter by $p^{-1}g_i(y)$ (Algorithm 4, Line 11).

Next, we denote $A \triangleq \sum_{y:h_i(y)=h_i(x)} \overline{f_y}g_i(x)g_i(y)$ and $B \triangleq p^{-1} \cdot \sum_{y:h_i(y)=h_i(x)} \ddot{f}_{y,i}g_i(x)g_i(y)$ (i.e., $C_{i,h_i(x)}g_i(x) = A + B$). We note that A and B are independent and that $\mathbb{E}\left[C_{i,h_i(x)}g_i(x)\right] = \mathbb{E}[A] + \mathbb{E}[B] = \overline{f_x} + \ddot{f}_x = f_x$. That is, the resulting estimator for row i is unbiased.

We now turn to bound the variance of the estimator by bounding $\text{Var}[A - \overline{f_x}]$ and $\text{Var}[B - p^{-1}\ddot{f}_{x,i}]$. First, since $\Pr[h_i(x) = h_i(y)] = 1/w$ for $x \neq y$, observe that:

$$\begin{aligned}\text{Var}[A - \overline{f_x}] &= \text{Var}\left[\sum_{y:h_i(y)=h_i(x)} \overline{f_y}g_i(x)g_i(y) - \overline{f_x}\right] \\ &= \text{Var}\left[\sum_{y\neq x:h_i(y)=h_i(x)} \overline{f_y}g_i(x)g_i(y)\right] \\ &= \mathbb{E}\left[\sum_{y\neq x:h_i(y)=h_i(x)} \overline{f_y}^2\right] \leq 1/w \sum_{y\in\mathcal{U}} \overline{f_y}^2. \end{aligned} \tag{3.4.3}$$

Next, let us analyze the variance of $B - p^{-1}\ddot{f}_{x,i}$:

$$\begin{aligned}
&\mathrm{Var}[B - p^{-1}\ddot{f}_{x,i}] \\
&\qquad = \mathrm{Var}\left[p^{-1} \cdot \sum_{y:h_i(y)=h_i(x)} \ddot{f}_{y,i} g_i(x) g_i(y) - p^{-1}\ddot{f}_{x,i}\right] \\
&\qquad = \mathrm{Var}\left[p^{-1} \cdot \sum_{y \neq x:h_i(y)=h_i(x)} \ddot{f}_{y,i} g_i(x) g_i(y)\right] \\
&\qquad = \mathbb{E}\left[p^{-2} \cdot \sum_{y \neq x:h_i(y)=h_i(x)} \ddot{f}_{y,i}^{\,2}\right] \leq p^{-2}/w \cdot \mathbb{E}\left[\sum_{y \in \mathcal{U}} \ddot{f}_{y,i}^{\,2}\right]. \qquad (3.4.4)
\end{aligned}$$

Similarly to Lemma 3.4.4, we have that $\mathbb{E}\left[\sum_{y \in \mathcal{U}} \ddot{f}_{y,i}^{\,2}\right] \leq 2p \sum_{y \in \mathcal{U}} \ddot{f}_y^{\,2}$, which allows reduce (3.4.4) to

$$\mathrm{Var}[B - p^{-1}\ddot{f}_{x,i}] \leq 2p^{-1}/w \cdot \sum_{y \in \mathcal{U}} \ddot{f}_y^{\,2}. \qquad (3.4.5)$$

Recall that during the processing of $\ddot{S}$, every packet is sampled with probability p and thus $\ddot{f}_{x,i} \sim Bin(\ddot{f}_x, p)$. Putting everything together we get:

$$\begin{aligned}
\mathrm{Var}\left[C_{i,h_i(x)}g_i(x) - f_x\right] &= \mathrm{Var}[A + B - f_x] \\
&= \mathrm{Var}[(A - \overline{f_x}) + (B - \ddot{f}_x)] \\
&= \mathrm{Var}[(A - \overline{f_x}) + (B - p^{-1}\ddot{f}_{x,i}) + (p^{-1}\ddot{f}_{x,i} - \ddot{f}_x)] \\
&= \mathrm{Var}[(A - \overline{f_x})] + \mathrm{Var}[(B - p^{-1}\ddot{f}_{x,i})] + \mathrm{Var}[(p^{-1}\ddot{f}_{x,i} - \ddot{f}_x)] \\
&\le 1/w \sum_{y\in\mathcal{U}} \overline{f_y}^2 + 2p^{-1}/w \cdot \sum_{y\in\mathcal{U}} \ddot{f}_y^{\,2} + \ddot{f}_x p^{-1} \\
&\le \frac{L_2^2 \cdot \left(1 + 2p^{-1}\right)}{w} + \ddot{f}_x p^{-1} \le \frac{L_2^2 \cdot \left(3p^{-1} + p^{-1}w/L_2\right)}{w}. \quad (3.4.6)
\end{aligned}$$

Here, the last inequality follows as $\ddot{f}_x \le f_x \le L_2$. We now use Lemma 3.4.6 to get that with a very high probability, $L_2 > w$. Intuitively, this follows from our convergence criteria (Algorithm 4, Line 12). This means that *conditioned on* $L_2 > w$ (which happens with probability $1 - \delta$), we have that

$$\begin{aligned}
\mathrm{Var}\left[C_{i,h_i(x)}g_i(x) - f_x\right] &\le \frac{L_2^2 \cdot \left(3p^{-1} + p^{-1}w/L_2\right)}{w} \\
&\le \frac{L_2^2 \cdot 4p^{-1}}{w} \le 3L_2^2\epsilon^2/8. \quad (3.4.7)
\end{aligned}$$

We now use Chebyshev's inequality to conclude that the estimator of the

i'th row, $\widehat{f_x}(i)$, satisfies

$$\Pr\left[|\widehat{f_x}(i) - f_x| \geq \epsilon L_2\right] = \Pr\left[|C_{i,h_i(x)} g_i(x) - f_x| \geq \epsilon L_2\right]$$
$$\leq \frac{\operatorname{Var}\left[C_{i,h_i(x)} g_i(x) - f_x\right]}{(\epsilon L_2)^2} \leq 3/8. \quad (3.4.8)$$

That is, the probability that each row estimates the frequency of x with an error no larger than $L_2\epsilon$ is at least $5/8$. Finally, standard use of Chernoff's inequality shows that $d = O(\log \delta^{-1})$ (independent) rows are required for their median to amplify the probability to $1 - \delta$. Taking the union bound over the events of sampling too early and having an error in the row's median, we have an error probability no larger than 2δ. This concludes the proof of Theorem 3.4.3.

3.4.5 Analysis of the Comparison with Uniform Sampling

Our sketch updates each row, for every packet, with probability p. An alternative approach, *uniform sampling*, would be updating *all rows* with probability $1/p$. We note that the two approaches make the same number of hash computations in expectation. Here, we claim that our approach is superior to that of uniform sampling.

Intuitively, our sketch uses the fact that for each row i, with probability $3/4$ we have $L_{2,i} = O(\sqrt{p}L_2)$. This reduction in the second norm allows one to increase the row width by a factor of p (compared to Count Sketch) to make up for the extra error introduced by the sampling. We now show that uniform sampling requires asymptotically more space as the second norm of the sampled substream is expected to be $\Omega\left(L_2\sqrt{p\log\delta^{-1}}\right)$ with probability $\Omega(\delta^{-1})$. Since Count Sketch is known to have an error of $\Omega\left(\overline{L_2}/\sqrt{w}\right)$ for streams with a second norm of $\overline{L_2}$, we get that for an error of ϵL_2 one would need to use more counters per row, or wait longer for the algorithm to converge. That is, a uniform sampling with the same update rate would require a multiplicative $\Omega\left(\log\delta^{-1}\right)$ more space.

To begin, we first discuss a *lower bound* on the error of Count Sketch. In Count Sketch, one uses a matrix of $\overline{w}$ columns and $\overline{d} = O(\log\delta^{-1})$ to get $\Pr\left[|\widehat{f_x} - f_x| \geq L_2/\sqrt{\overline{w}}\right] \leq \delta$. For an ϵL_2 guarantee, one then sets $\overline{w} = O(\epsilon^{-2})$. We now show that this is asymptotically tight. Namely, we show that there exists a distribution for which $\Pr\left[|\widehat{f_x} - f_x| \geq \epsilon L_2\right] = \Omega(\delta)$.

To prove our result, we use the following theorem.

Theorem 3.4.7. ([105, 106]) Let X be a binomial variable such that $\mathrm{Var}[X] \geq 40000$. Then for all $t \in [0, \mathrm{Var}[X]/100]$, we have

$$\Pr[X \geq E[X] + t] = \Omega\left(e^{-t^2/3\,\mathrm{Var}[X]}\right)$$

We are now ready to show a lower bound on the error of Count Sketch.

Lemma 3.4.8. Let $n \geq m+1$. Consider Count Sketch allocated with $\overline{d} = O(\log\delta_1^{-1})$ rows and $\overline{w} \leq m/c'$ columns, for a sufficiently large constant[4] c'. There exists $c = \Theta(1)$, a stream $S \in [n]^{[m]}$, and an element $x \in [n]$ such that $\Pr\left[|\widehat{f_x} - f_x| \geq c \cdot L_2/\sqrt{\overline{w}}\right] \geq \delta_1$.

[4]In practice, $w \ll m$, as otherwise we have enough memory for exact counting and would not need sketches, and this trivially holds.

Proof. We denote by c'' the constant in the $\Omega(\cdot)$ of Theorem 3.4.7, and by $z = O(1)$ the constant such that $\overline{d} = z \ln \delta_1^{-1}$. Let $c' = \max\{320000, -8 \ln \left(1 - e^{-1/2z}/c''\right)\}$ and $c = \sqrt{\frac{3}{4z}}$ be two constants. We will show that with probability of at least e^{-z}, each row has an error of at least $c \cdot L_2/\sqrt{\overline{w}}$. This would later allow us to conclude that the estimation, which is the median row, has an error of $c \cdot L_2/\sqrt{\overline{w}}$ with probability of at least δ.

Consider the stream in which all elements of $[m]$ arrive once each (and thus, $L_2 = \sqrt{m}$), and consider a query for $x \triangleq m+1$ (i.e., $f_x = 0$). Fix a row i, and let $Q \triangleq \{j \in [m] \mid h_i(j) = h_i(x)\}$ be the elements that affect x's counter on the i'th row. Intuitively, we show that the number of items that change x's counter ($C_{i,h_(x)}$) is $|Q| = \Omega\left(L_2/w\right)$ and then give a lower bound on the resulting value of the counter (given that some of the flows in Q increase it while others decrease). Observe that $|Q| \sim Bin(m, 1/\overline{w})$. According to Chernoff's bound:

$$\Pr\left[|Q| \leq m/2\overline{w}\right] \leq e^{-m/8\overline{w}} \leq e^{-c'/8} \leq 1 - e^{-1/2z}/c''. \tag{3.4.9}$$

Next, we denote by $X \triangleq \{j \in Q \mid g_i(j) = +1\}$ the number of elements from Q that increased the value of x's counter. Observe that $X \sim Bin(|Q|, 1/2)$ is binomially distributed and that x's counter satisfies $c_{i,h_i(x)} = 2X - |Q|$. Conditioned on the event $|Q| > m/2\overline{w}$ (which happens with constant probability as (3.4.9) shows), we have that $\mathrm{Var}[X] = |Q|/4 \geq m/8\overline{w} = c'/8 \geq 40000$. According to Theorem 3.4.7, we now have that

$$\Pr\left[X \geq \mathbb{E}[X] + t \;\mid\; |Q| > m/2\overline{w}\right] \geq c'' e^{-t^2/3\,\mathrm{Var}[X]} \tag{3.4.10}$$

some $c'' > 0$ and any $t \in [0, \mathrm{Var}[X]/100]$. We will now show that in each row i, $\Pr\left[c_{i,h_i(x)} \geq c \cdot L_2/\sqrt{w}\right] \geq e^{-1/z}$, for $c = \sqrt{\frac{3}{4z}}$.

$$\Pr\left[c_{i,h_i(x)} \geq c \cdot L_2/\sqrt{\overline{w}}\right] = \Pr\left[2X - |Q| \geq c \cdot \sqrt{m/\overline{w}}\right]$$

$$= \Pr\left[X \geq (|Q| + c \cdot \sqrt{m/\overline{w}})/2\right] = \Pr\left[X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right]$$

$$\geq \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \wedge \left(|Q| > m/2\overline{w}\right)\right]$$

$$= \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] \Pr\left[|Q| > m/2\overline{w}\right]$$

$$\geq \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] \cdot e^{-1/2z}/c''. \tag{3.4.11}$$

Setting $t \triangleq c \cdot \sqrt{m/4\overline{w}} = O(\sqrt{\mathrm{Var}[X]})$ and using (3.4.10), we get that

$$\Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right]$$

$$\geq c''e^{-(c\cdot\sqrt{m/4\overline{w}})^2/3\,\mathrm{Var}[X]} \geq c''e^{-(c\cdot\sqrt{m/4\overline{w}})^2/3(m/8\overline{w})}$$

$$= c''e^{-2c^2/3} = c''e^{-1/2z},$$

where the last inequality follows from $\mathrm{Var}[X] = |Q|/4 \geq m/8\overline{w}$. Plugging this back into (3.4.11) we get

$$\Pr\left[c_{i,h_i(x)} \geq c \cdot L_2/\sqrt{\overline{w}}\right]$$

$$\geq \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] e^{-1/2z}/c''$$

$$\geq c''e^{-1/2z}e^{-1/2z}/c'' = e^{-1/z}.$$

Thus, we established that in each row i with a probability of at least e^{-z}, x's counter (and thus, the error) is larger than $c \cdot L_2/\sqrt{\overline{w}}$. Finally, since the rows are independent, we get that the probability of Count Sketch returning a wrong estimate is at least

$$\Pr\left[\widehat{f_x} - f_x \geq c \cdot L_2/\sqrt{\overline{w}}\right] \geq \left(e^{-1/z}\right)^{\overline{d}} = \left(e^{-1/z}\right)^{z \ln \delta_1^{-1}} = \delta_1.$$

$\square$

To proceed, we need some inequalities that allow us to provide a lower bound on the reduction in L_2 of the sub-sampled stream. To that end, we use the following results:

Theorem 3.4.9. ([107]) Let $X \sim Bin(n,p)$; for all k such that $np \leq k \leq n(1-p)$:

$$\Pr\left[X \geq k\right] \geq 1 - \Phi\left(\frac{k-np}{\sqrt{np(1-p)}}\right),$$

where $\Phi(z) \triangleq \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$ is the cumulative distribution function of the normal distribution.

Theorem 3.4.10. ([108]) For any $z > 0$:

$$1 - \Phi(z) > \frac{z}{1+z^2}\phi(z),$$

where $\phi(z) \triangleq \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$ is the density function of the normal distribution.

For convenience, we also use the following fact:

Fact 3.4.11. For any $z \geq 2$:

$$\frac{z}{1+z^2}\phi(z) = \frac{z}{1+z^2}\frac{1}{\sqrt{2\pi}} e^{-z^2/2} \geq e^{-z^2}$$

Next, we will provide a lower bound on the reduction in L_2 when sub-sampling a stream with probability p. Once again, we consider the stream S in which m distinct elements arrived once each (and thus its L_2 is $\sqrt{m}$).

Lemma 3.4.12. Let $\overline{S}$ be a substream of S such that each packet in S appears in $\overline{S}$ independently with probability $p \leq 1/2$. Denote by L_2^S the L_2 of S and by $L_2^{\overline{S}}$ the L_2 of $\overline{S}$. Then for $\delta_2 \leq 1/4$:

$$\Pr\left[L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right] \geq \delta_2.$$

Proof. Denote by J the set of sampled elements; observe that $|J| \sim Bin(m,p)$ and that $L_2^{\overline{S}} = \sqrt{|J|}$. According to Theorem 3.4.9, Theorem 3.4.10, and Fact 3.4.11, we have that:

$$\Pr\left[|J| \geq mp + \sqrt{mp(1-p)\log\delta_2^{-1}}\right] \geq \delta_2.$$

Thus, we have that:

$$\begin{aligned}
&\Pr\left[L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right] \\
&= \Pr\left[\sqrt{|J|} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right] \\
&= \Pr\left[|J| \geq mp + \sqrt{mp(1-p)\log\delta_2^{-1}}\right] \geq \delta_2. \qquad \square
\end{aligned}$$

The above lemma shows that the L_2 of the uniformly sub-sampled stream is larger than $\sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}$ with probability $\geq \delta_2$. In contrast, in our sketch every row processes a sub-stream with an L_2 of $O(F_2\sqrt{p})$ (i.e., $O(\sqrt{mp})$ for this stream) with a constant probability, *independently from the other rows*. We now show that in some cases (when the desired error probability is small), the dependence between the rows in the case of uniform samples requires asymptotically more space than our sketch, for the same error guarantee. Therefore, we claim that our sketch has clear advantages over uniform sampling.

Theorem 3.4.13. Let $\overline{S}$ be a substream of S such that each packet in S appears in $\overline{S}$ independently with probability p. There exists a stream S such that Count Sketch with $d = \Theta(\log \delta^{-1})$ rows applied on $\overline{S}$ requires $w = \Omega\left(\epsilon^{-2}p^{-1} + \epsilon^{-2}p^{-1.5}m^{-0.5}\sqrt{\log \delta^{-1}}\right)$ counters per row to provide (with probability $1-\delta$) an ϵL_2 error for S.

Proof. We set $\delta_1 = \delta_2 = \sqrt{\delta}$ (and thus $\log \delta_1^{-1}, \log \delta_2^{-1} = \Theta\left(\log \delta^{-1}\right)$). According to Lemma 3.4.8, we have that there exists $c = \Theta(1)$ such that:

$$\Pr\left[|\widehat{f_x} - f_x| \geq c \cdot L_2^{\overline{S}}/\sqrt{\overline{w}}\right] \geq \delta_1$$

Next, we use Lemma 3.4.12 to obtain

$$\Pr\left[L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log \delta_2^{-1}}}\right] \geq \delta_2.$$

Since the Count Sketch uses randomization that is independent from the stream sampling, we have that

$$\Pr\left[\left(L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right) \wedge \left(|\widehat{f_x} - f_x| \geq c \cdot L_2^{\overline{S}}/\sqrt{w}\right)\right] \geq \delta_1\delta_2 = \delta. \quad (3.4.12)$$

Thus, with probability of at least δ, the error of the Count sketch is

$$\Omega\left(\sqrt{\frac{mp + \sqrt{mp(1-p)\log\delta^{-1}}}{w}}\right).$$

Next, recall that to estimate the frequencies in the original stream S, one need to divide the Count Sketch estimate by p. Thus, if we denote the resulting estimation error by $\mathfrak{E}_x$ we have that

$$\Pr\left[\mathfrak{E}_x = \Omega\left(p^{-1}\sqrt{\frac{mp + \sqrt{mp(1-p)\log\delta^{-1}}}{w}}\right)\right] \geq \delta.$$

To provide an $\epsilon L_2 = \epsilon\sqrt{m}$ guarantee, uniform sampling Count Sketch needs to set w such that $\Pr\left[\mathfrak{E}_x \geq \epsilon\sqrt{m}\right] \leq \delta$. Demanding

$$p^{-1}\sqrt{\frac{mp + \sqrt{mp(1-p)\log\delta^{-1}}}{w}} = \epsilon\sqrt{m}$$

the bound follows. □

We therefore conclude that while our sketch requires $O(\epsilon^{-2}p^{-1}\log\delta^{-1})$ counters overall, inserting a uniform sample into Count Sketch, for the same sampling probability p and error guarantee, requires at least

$$\Omega\left(\epsilon^{-2}p^{-1}\log\delta^{-1}+\epsilon^{-2}p^{-1.5}m^{-0.5}\log^{1.5}\delta^{-1}\right).$$

3.5 Implementation

We have built a prototype of NitroSketch in C and integrated it with Open vSwitch (OVS-DPDK) and FD.io/Vector Packet Processing (VPP). We implement four sketches with NitroSketch: UnivMon [26], Count-Min Sketch [4], Count Sketch [6], and K-ary Sketch [8]. In our implementation, we focus on a single-thread measurement daemon for both OVS and VPP. We use the xHash library's [109] hash function. When hashing the same flow-key with different hash seeds, we utilize Intel AVX2 instruction set [110] to parallelize the hash operations.

At a high level, NitroSketch includes two modules: a data-plane **Sketching** module, and a control-plane **Estimation** module. The Sketching module maintains the sketch data structure and the Estimation module fetches the data from the Sketching module. In the next section, we describe the two different versions of the Sketching module.

3.5.1 Data Plane Module

OVS-DPDK Integration. Since OVS-DPDK enables the packet processing entirely in user space, the user space `vswitchd` thread has a three-tier look-up cache hierarchy. The first-level table works as an *Exact Match Cache (EMC)* with fastest look-up speed. If a packet misses in EMC, it goes through the second-level classifier as a Tuple Space Search, and finally it might trigger the third-level table managed by an OpenFlow-compliant controller. For more details about the architecture of OVS vswitchd, please refer to the paper [84]. For efficiency, we integrate the sketching module with the OVS-DPDK's EMC module in dpif-netdev. We provide implementations for varying performance requirements:

- **All-in-one version.** In this version, the Sketching module works as a sub-module of the EMC module inside an OVS vswitchd/PMD thread. That is, for each packet batch received from DPDK PMD, NitroSketch decides which packet is replicated and measured without affecting the packet batch, as described in Section 3.3.2. This extension incurs small processing overhead to the EMC module, but there is a dedicated CPU core to all tasks, such as DPDK, table look-up, and measurement.

- **Separate-thread version.** In this version, the Sketching module works as a separate thread besides the OVS vswitchd thread. When a packet batch arrives, the extended EMC module (pre-processing stage) in vswitchd decides which packets' headers to add into a fast lock-free concurrent FIFO queue (modified from [111]). A separate NitroSketch thread (sketch-updating stage) fetches the packets' header fields concurrently and handles the sketching updates. This implementation has minimal overheads for the vanilla OVS-DPDK but requires an additional core for the measurement tasks.

VPP Integration. VPP is a modular, flexible, and extensible platform that runs entirely on the user-space. VPP is based on a "packet processing graph", where each node is a module and packets are processed node by node. For instance, in a simple VPP based L3 vSwitch, VPP first fetches packets from the network I/O as a batch. VPP then sends the packet batch to the Ethernet-input module (L2), and then through IP4-input and IP4-lookup modules (L3). We implemented a measurement module in VPP 18.02 and added it to the packet processing graph after the VPP IP stack. This module runs both stages of NitroSketch in a dedicated thread, minimizing the impact on other VPP plugins.

3.5.2 Control Plane Module

The control plane module (i) periodically (at the end of each epoch) receives sketching data from the data plane module by a simple RPC protocol through a 1GbE link connected to the virtual switch; (ii) assigns the sketching data to the corresponding measurement tasks based on user definitions; (iii) calculates the estimated results. For instance, it processes the UnivMon [26] sketching data and calculates HH, Change detection, or traffic Entropy.

3.6 Evaluation

Our evaluation demonstrates that NitroSketch: (a) can meet 10GbE line-rate with min-sized packets, and match 40GbE on real workloads with a single core; (b) runs on software switches with small CPU overheads; (c) provides accurate results once converged; (d) has significantly higher throughput ($> 7.6\times$ faster) and better accuracy once converged when compared to SketchVisor [94], and (e) is more accurate and requires less memory than NetFlow [2] and sFlow [3].

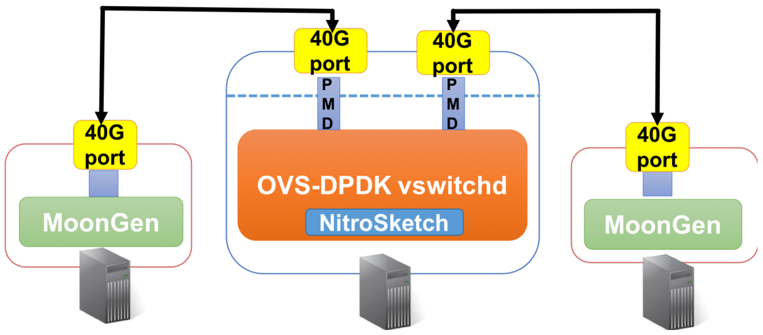


Figure 3.4: Overview of the evaluation testbed.

3.6.1 Methodology

Testbed. We evaluate NitroSketch on a set of 4 commodity servers running Ubuntu 16.04.03, each of which has an Intel Xeon E5-2620 v4 CPU@3.0Ghz, 128GB DDR4 2400Mhz memory, two Broadcom BCM5720 1GbE NICs, and an Intel XL710 Ethernet NIC with two 40-Gigabit ports. Our testbed has three hosts as the data plane (Figure 3.4), which send traffic directly through 10/40G links. The control is connected through a 1GbE link. Each virtual switch is configured with two forwarding rules for bidirectional packet forwarding.

Workloads. We use four types of workloads: (a) *CAIDA*: 10 one-hour public CAIDA traces from 2015 [58] and 2016 [95] each containing 1 to 1.9 billion packets; (b) *Min-sized*: simulated traffic with minimal sized packets for stress testing; (c) *Data center*: data center traces UNI1 and UNI2 from [97]; (d) *Cyber attack*: DDoS attack traces from [96]. The average packet sizes in the CAIDA, DDoS attack, data center traces are 714, 272, and 747 bytes respectively.

For optimal switching performance, we modify the MAC addresses of packets to avoid cache misses on the Exact-Match Cache of OVS-DPDK. We use MoonGen [104] packet sender/generator to replay traces, and to generate valid random 64B packets.

Sketches and metrics. We evaluate NitroSketch with four existing sketches: Count-Min Sketch [4], Count-Sketch [6], UnivMon [26], and K-ary Sketch [8]. We use source IP as the flow key.

We consider the following performance metrics:

- **Throughput:** the traffic volume processed per second as Gigabits per second (Gbps).

- **Packet Rate:** the number of packets transmitted per second as Million packets per second (Mpps). For 64B packets, 10Gbps throughput is equivalent to 14.88Mpps, and 40Gbps equals to 59.53Mpps.

- **CPU Utilization:** the percentage of the total CPU time spent on each module/function, measured by Intel VTune Amplifier 2018 [103].

- **Accuracy:** the accuracy of three measurement tasks: Heavy Hitter (HH), Change Detection (Change), and Entropy Estimation (Entropy). For HH and Change, we set a threshold 0.01% and estimate the relative errors on the detected flows. We report *relative error*= $\frac{|t-t_{real}|}{t_{real}}$, where t_{real} is the ground truth of a task and t is the measured value. For each data point, we run 10 times independently and report the *median* and the *standard deviation*. Also, the *recall rate* is defined as the ratio of true instances found.

Parameters. By default, we set a 95% precision guarantee. Note that this is a theoretical guarantee and NitroSketch achieves higher fidelity in practice.

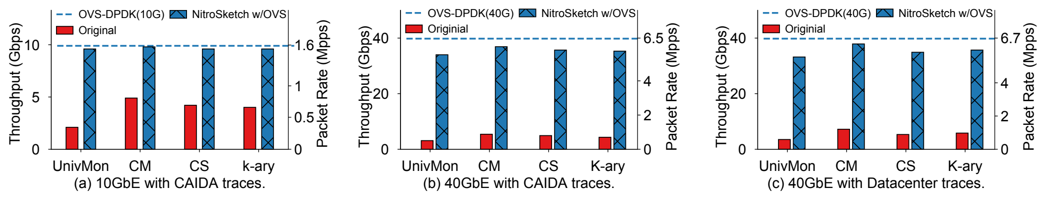


Figure 3.5: Throughput/Packet rate on OVS-DPDK with the all-in-one version using CAIDA and data center traces.

For throughput evaluation, we set a $p = 0.01$ geometric sampling rate for NitroSketch and allocate the memory based on the precision guarantee. We evaluate four sketches in NitroSketch. (a) UnivMon: we allocate 2MB, 1MB, 500KB, 250KB for the first HH sketches, and 125KB for the rest of sketches. (b) Count-Min: we use 20KB memory for 5 rows of 1000 counters. (c) Count Sketch: we allocate 2MB for 5 rows of 102400 counters. (d) K-ary Sketch: we utilize 10 rows of 51200 counters.

3.6.2 Throughput

Throughput with all-in-one. We evaluate the throughput of the all-in-one version in Figure 3.5 with 1h CAIDA traces and 1h datacenter traces. All original sketches implemented with OVS-DPDK suffer from significant throughput degradation. Among the four sketches, UnivMon only achieves 2.9Gbps and the faster Count-Min only reaches 5.5Gbps. After plugging in NitroSketch, all sketches achieve 10G and 40G line rates under CAIDA and datacenter traces, without adding an extra thread. We observe that inside this vswitchd thread, DPDK, OVS, and NitroSketch modules "squeeze" all the potential of a single core.

Throughput with separate-thread. Figure 3.6 shows the throughput of the

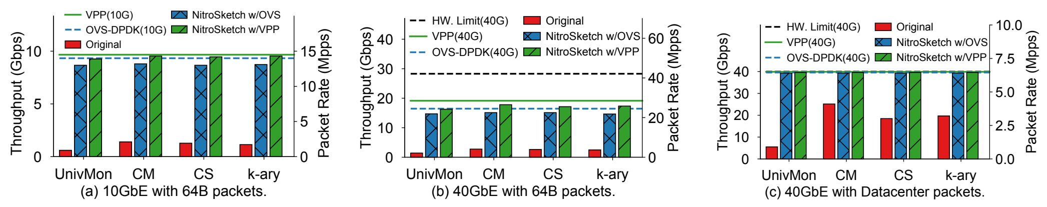


Figure 3.6: Throughput/Packet rate on OVS-DPDK and VPP with the separate-thread version using 64B packets and data center traces. In (a), virtual switches use one CPU core to switch packets while in (b) and (c) there are two cores.

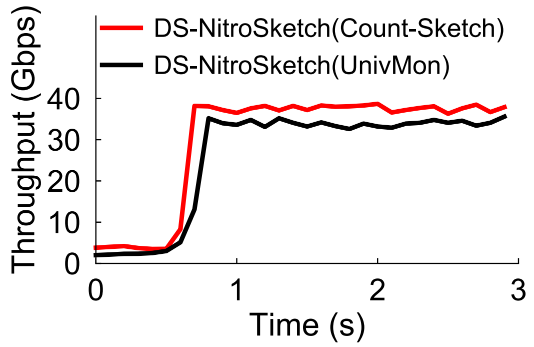


Figure 3.7: Throughput over time for the delayed sampling approach (DS-NitroSketch) with two different sketches (Setting: 40GbE with CAIDA traces).

separate-thread version. It is already difficult for virtual switches to achieve 10G line-rate on a single core with 64B packets. For 40G, even vanilla DPDK does not reach the line rate with 64B packets due to the hardware limitation in Intel XL710 NIC [102]. This means that OVS-DPDK and VPP cannot reach this line rate under 64B packet traces.

In Figure 3.6(a), we can see that NitroSketch has a negligible throughput impact on the performance of virtual switches. That is, it achieves 10G line rate under any packet workload. As is evident from Figure 3.6(b) and (c), NitroSketch is not the bottleneck in achieving 40G line rates for 64B packets and for data center workloads.

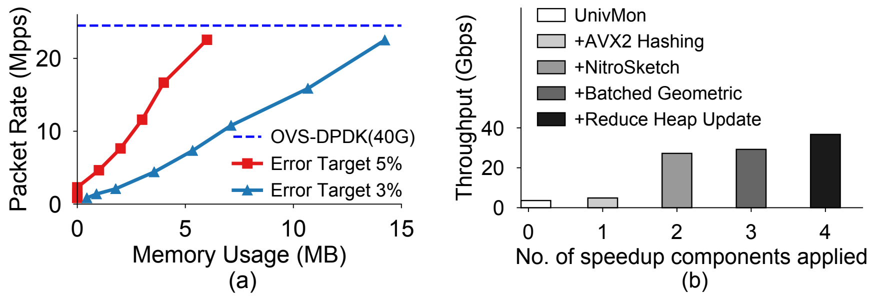


Figure 3.8: (a) Throughput vs. memory for varying error targets. (b) Throughput with different NitroSketch components applied: 0: Vanilla UnivMon; 1: add AVX2 paralleled hash; 2: apply also NitroSketch; 3: add pre-batched geometric samples; 4: apply also sampling for heap update. (Setting: one vswitchd thread with 40GbE NIC.)

Throughput with DS-NitroSketch. To evaluate the convergence time, we implement DS-NitroSketch with Count-Sketch and UnivMon in OVS-DPDK with the all-in-one version. In Figure 3.7, we report the measured throughput every 0.1sec (extra measurement overhead added) under 40GbE. We see that it needs about 0.6s for Count-Sketch and 0.8s for UnivMon to reach full throughput.

Throughput vs. Memory. To guarantee an error budget ϵ (for any distribution), the sampling probability p in the pre-processing stage depends on the amount of allocated memory. To illustrate this trade-off, we set error guarantees 3% and 5% for UnivMon with NitroSketch. Figure 3.8(a) shows that NitroSketch copes with 40G OVS-DPDK with an acceptable increase in memory.

Improvement breakdown. While implementing NitroSketch, we used multiple optimization techniques. Therefore, we evaluate the gains of each optimization separately for UnivMon with NitroSketch. Figure 3.8(b) confirms

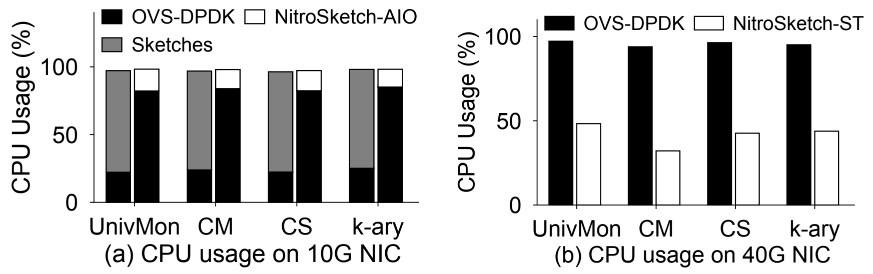


Figure 3.9: CPU usage of the all-in-one version (NitroSketch-AIO) and the separate-thread version (NitroSketch-ST)

that the NitroSketch technique offers the most significant speedup.

3.6.3 CPU Utilization

A single DPDK PMD thread is continuously polling packets from NIC. It "saturates" a core and utilizes 100% CPU reported from a universal process viewer (e.g., htop). Therefore, we profile the CPU time of each module.

CPU Time in all-in-one. We measure the CPU time in the same setting as in Figure 3.5. As shown in Figure 3.9(a), when vanilla sketches are running, the majority of the CPU time is spent on sketching, and the overall switching performance drops. After applying NitroSketch-AIO, the switch achieves line-rate while keeping the CPU time of NitroSketch-AIO to $< 20\%$.

CPU Time in separate-thread. Figure 3.9(b) compares the CPU time between OVS-DPDK and NitroSketch-Separate Thread, in a setting as in Figure 3.6(b). When the switch is saturated with min-sized packets ($\sim$22Mpps), the cores for packet switching are running at nearly 100% while NitroSketch is not running at full-speed and would handle higher packet rates, if the virtual switch could support it.

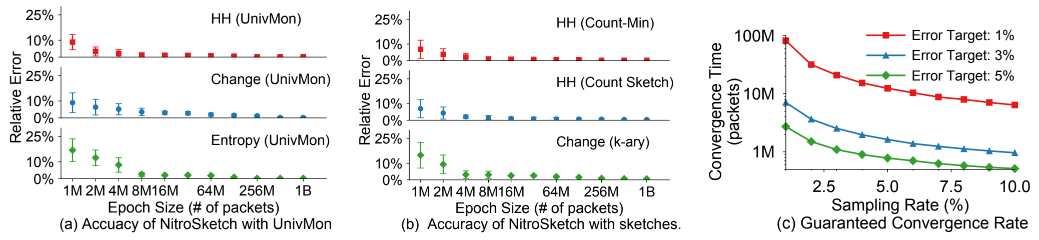


Figure 3.10: (a),(b) Error rates of NitroSketch. (c) Convergence time on CAIDA traces.

3.6.4 Accuracy and Convergence Time

We evaluate the accuracy of HH, Change, and Entropy in NitroSketch with different sized epochs and report in Figure 3.10(a) and (b). NitroSketch achieves better-than-guaranteed results ($< 5\%$ error) after seeing 2-3M packets.

Since NitroSketch uses geometric sampling to select packets, it requires a convergence time to produce a guaranteed accurate result (analyzed in section 3.4). For different error targets on CAIDA traces, we study the trade-off between geo-sampling rate p and the convergence time (in terms of the number of packets) and report in Figure 3.10(c). Further, NitroSketch is expected to converge faster on data center traces due to their expected larger L_2 value establishment.

3.6.5 Comparison with Other Solutions

Comparison with SketchVisor. Since the source code of SketchVisor [94] on Open vSwitch is not publicly available, we implement its fast-path algorithm in C and carefully integrate it with UnivMon on OVS-DPDK using the same FIFO buffer as NitroSketch [111]. The performance of Sketchvisor depends on how traffic distributes between the fast and the normal paths which is

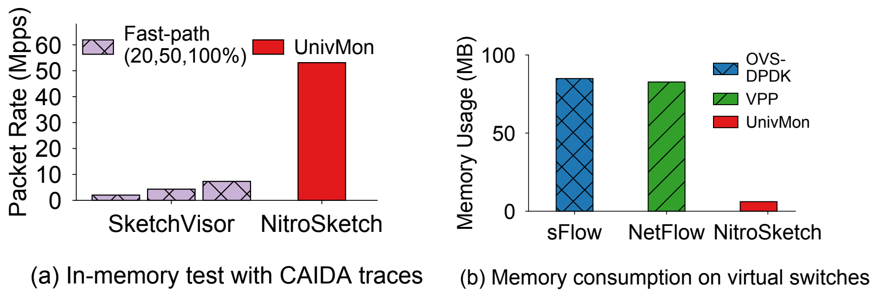


Figure 3.11: (a) In-memory packet rates: SketchVisor vs. NitroSketch. (b) Memory usage: NetFlow vs. NitroSketch.

unknown. Thus we evaluate the throughput based on in-memory testing with manually injecting 20%, 50%, 100% of traffic into the fast path. The CAIDA traces are entirely loaded into DRAM using libpcap [112] to eliminate the packet I/O between NIC and software switches. We allocate memory for SketchVisor and NitroSketch to detect top 100 HHs, we use 900 counters for the fast-path and set 5% error guarantee on UnivMon.

As reported in Figure 3.11(a), the throughput of SketchVisor improves when the percentage of traffic handled by the fast-path increases. When the fast-path processes 20% of the traffic, it achieves 2.12Mpps. Sketchvisor achieves its maximum packet rate of 6.11Mpps when 100% traffic goes into the fast-path. Meanwhile, NitroSketch runs at a dramatically faster speed of 53Mpps. Unsurprisingly, this explains the situation that SketchVisor uses 100% CPU (not shown in the figure) while NitroSketch requires less than 50% (shown in Figure 3.9(b)) when running in a separate thread on OVS-DPDK.

We observe that to cope with the full 10G speed and avoid packet drops, the fast-path has to handle 100% of the packets. For a fair comparison on OVS, we prevent drop packet by using a very large buffer. We manually redirect 20%, 50%, and 100% of the packets to the fast-path. Figure 3.12(a), (b) and (c) report

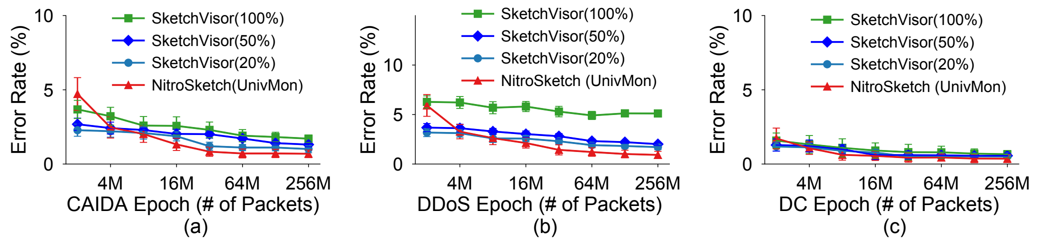


Figure 3.12: HH errors on SketchVisor and NitroSketch, in CAIDA, DDoS, and data center traces.

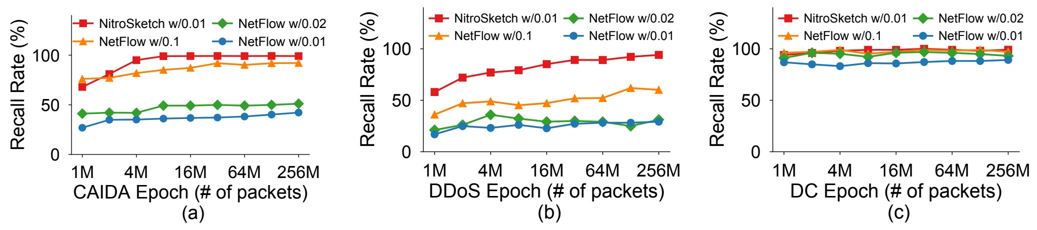


Figure 3.13: HH recall rates on NetFlow/sFlow with different sampling rates and NitroSketch with 0.01, using CAIDA, DDoS, and data center traces.

relative errors on HH in the three traces. We can see that NitroSketch has larger errors before convergence ($<$ 3.61M packets) but is more accurate than SketchVisor after convergence. In a 10G OVS-DPDK switch, this stabilization time can be as little as 0.24 seconds. Here, SketchVisor has degraded accuracy in the CAIDA and DDoS trace in Figure 3.12(a) and (b) and relatively good accuracy in the data center trace [97]. In contrast, NitroSketch achieves good accuracy on all traces.

Comparison with NetFlow/sFlow. On OVS-DPDK and VPP, NetFlow/sFlow are default monitoring tools. We configure OVS-DPDK to enable sFlow and VPP to enable NetFlow. We set a polling interval of 10 seconds with sampling rates of 0.01, 0.02, and 0.1 for NetFlow. For fairness, we configured NitroSketch with a sampling probability of 0.01. On the controller, we collect the sampled packets/reports with Wireshark [113] directly from the port. Figure 3.11(b) indicates that NetFlow consumes much more memory even with 0.01 sampling rate. In NetFlow (as in Figure 3.13), we observe that the recall rates of 100 HHs are low in the CAIDA and DDoS traces and are relatively good in the UNI2 datacenter trace [97]. This is because UNI2 is quite skewed while CAIDA and DDoS are heavy tailed. In contrast, NitroSketch achieves high recall rates in all cases.

3.7 Chapter Summary

Sketching is an attractive theoretical construction for monitoring in software switches. However, its current performance on software switches is far from ideal to serve as a viable line-rate and low CPU consumption option. By identifying the key bottlenecks and reformulating the requirements for software sketch implementation, we develop NitroSketch based on two popular software switches. Our results show that NitroSketch can serve as the basis for fast and efficient realizations of many popular sketches on software switches.

Chapter 4

ASAP: Fast, Approximate Graph Pattern Mining at Scale

The recent past has seen a resurgence in storing and processing massive amounts of graph-structured data [114, 115]. 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 [116], community detection [117] and label propagation [118]. The second, *graph pattern mining* algorithms, discover structural patterns in a graph. Examples of graph pattern mining algorithms include motif finding [119], frequent sub-graph mining (FSM) [120] and clique mining [121]. Graph mining algorithms are used in applications like detecting similarity between graphlets [122] 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 [15, 16, 17, 18, 19, 20, 21, 22, 23, 123, 124, 125, 126, 127, 25].

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 [128] 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 [13, 129].

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 [13] 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.

In this chapter, we present ASAP[1], a system that enables both *fast* and *scalable* pattern mining. ASAP is motivated by one key observation: *in many*

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

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 revealed that their application for social graph similarity uses a count of similar graphlets [122]. Another company's 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 [130, 131, 132], 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 (section 4.1.1).

Instead, in ASAP, we leverage graph approximation theory, which has a

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

rich history of proposing approximation algorithms for mining specific patterns such as triangles. 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 sampling instance 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 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.

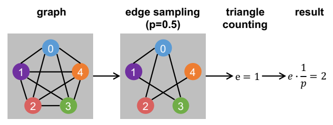
These techniques allow ASAP to outperform Arabesque [13], a state-of-the-art exact pattern mining solution by up to 77$\times$ on the LiveJournal graph while incurring less than 5% error. In addition, ASAP can scale to graphs with billions of edges—for instance, ASAP can count all the 6 patterns in 4-motifs 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 chapter:

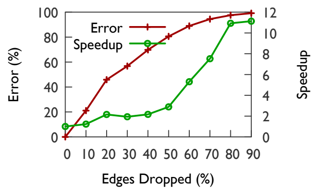
- We present ASAP, the first system to our knowledge, that does fast, scalable approximate graph pattern mining on large graphs. (section 4.2)
- We develop a general API that allows users to mine any graph pattern and present techniques to automatically distribute executions on a cluster. (section 4.3)
- We propose techniques that quickly infer the relationship between approximation error and latency, and show that it is accurate across many real-world graphs. (section 4.4)
- We show that ASAP handles graphs with billions of edges, a scale that existing systems failed to reach. (section 4.5)

4.1 Background & Motivation

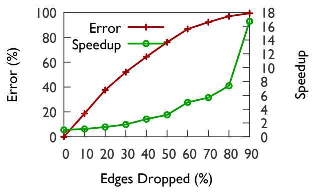
In this section, we describe recent advancements in graph pattern mining theory that we leverage.



(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.

4.1.1 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 [130, 131] and databases [133, 134, 135], 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 [130] 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 4.1(a) 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 [136] for finding 3-chains and the UK webgraph [137] 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. 4.1(b) and 4.1(c) 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, in this chapter, we propose a new approach.

4.1.2 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 [138, 139, 140, 141, 142, 143, 144].

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 [142] 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 [142]. 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

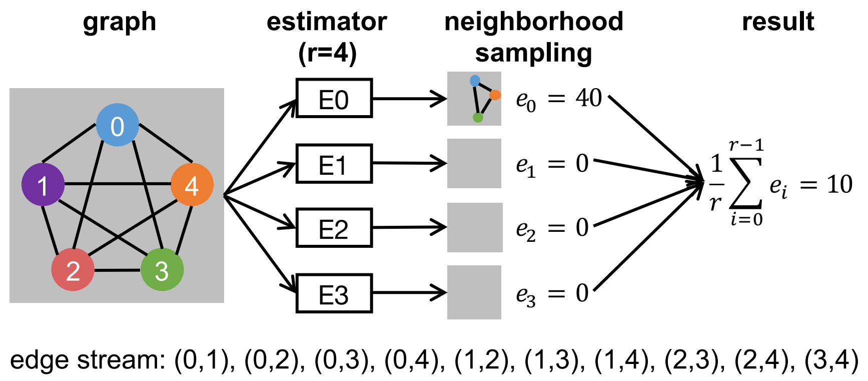

graph
estimator (r=4)
neighborhood sampling
result
0
1
4
2
3
E0
E1
E2
E3
$e_0 = 40$
$e_1 = 0$
$e_2 = 0$
$e_3 = 0$
$\frac{1}{r}\sum_{i=0}^{r-1} e_i = 10$
edge stream: (0,1), (0,2), (0,3), (0,4), (1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

Figure 4.2: Triangle count by neighborhood sampling

trial would find an instance of the given pattern, and thus requires fewer estimations to achieve the same accuracy.

4.1.2.1 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.

- **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.

4.1.3 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 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.

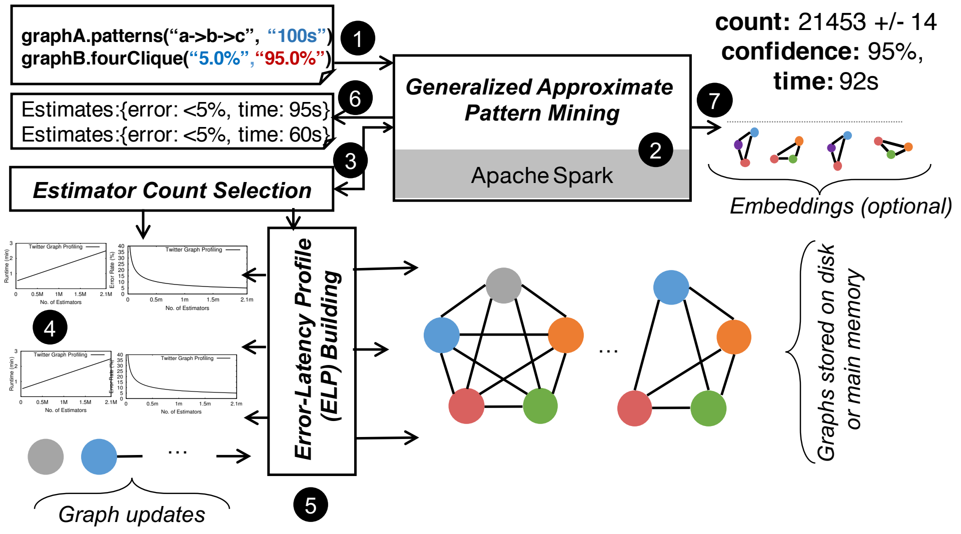

graphA.patterns("a->b->c", "100s")
graphB.fourClique("5.0%","95.0%")
1
Estimates:{error: <5%, time: 95s}
Estimates:{error: <5%, time: 60s}
6
3
Estimator Count Selection
Generalized Approximate Pattern Mining
2
Apache Spark
7
count: 21453 +/- 14
confidence: 95%,
time: 92s
Embeddings (optional)
4
Error-Latency Profile (ELP) Building
5
Graph updates
Graphs stored on disk or main memory
Twitter Graph Profiling
No. of Estimators

Figure 4.3: ASAP architecture

4.2 ASAP Overview

In this work, 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 section 4.3. 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 (section 4.3.1.2). 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 the rest of this chapter. 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 section 4.4. 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 [145, 146].

4.3 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(section 4.3.1) that generalizes neighborhood sampling. Then, we describe how ASAP handles errors that arise in distributed processing(section 4.3.2). Finally, we show how ASAP can handle queries with predicates on edges or vertices(section 4.3.3).

4.3.1 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 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

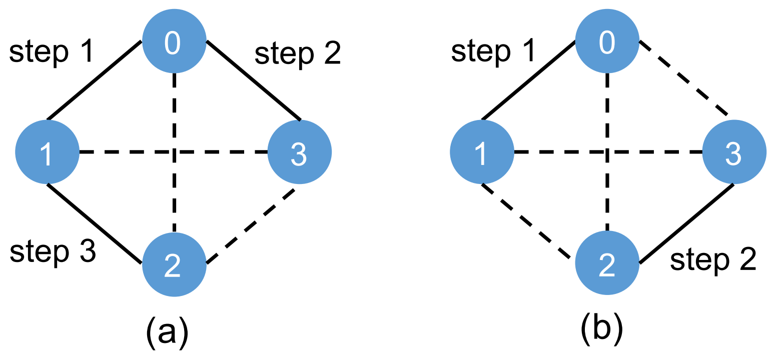


Figure 4.4: Two ways to sample four cliques. (a) Sample two adjacent edges $(0,1)$ and $(0,3)$, sample another adjacent edge $(1,2)$, and wait for the other three edges. (b) Sample two disjoint edges $(0,1)$ and $(2,3)$, and wait for the other four edges.

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 $\frac{1}{m^2}$.

4.3.1.1 Analysis of General Patterns

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

Definition 4.3.1 (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 $r \leq 1$ 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 $i \geq 1$. 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 4.3.2. 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

$$Pr[p = p^*, k = 5] = \frac{1}{m^2 \cdot c_1} \text{ (Type-I)}$$

$$\text{or } = \frac{1}{m^2 \cdot c_2} \text{ (Type-II.a)}$$

$$\text{or } = \frac{1}{m \cdot c_1 \cdot c_2 \cdot c_3} \text{ (Type-II.b)}$$

Proof. Since a pattern is connected, the sampling phase is 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 [142], 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,

$$Pr[p = p^*, k = 5] = Pr[p = p^*, k = 3] * Pr[\mathcal{E}_3^*|\mathcal{E}_1, \mathcal{E}_2]$$

$$= \frac{1}{m^2 \cdot c(e_1)} \text{ (Type-I)}$$

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,

$$Pr[p = p^*, k = 5] = Pr[p = p^*, k = 4] * Pr[\mathcal{E}_4|\mathcal{E}_1, \mathcal{E}_2^*]$$

$$= \frac{1}{m^2 \cdot c(e_1, e_2)} \text{ (Type-II.a)}$$

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

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

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

Proof. By Lemma 4.3.2, 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,

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

$\square$

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 4.3.4. Let $r \geq 1$, $0 < \epsilon \leq 1$, and $0 < \delta \leq 1$. 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 $r \geq \frac{C_1 m^2}{f(G)}$ Type-I estimators, or $r \geq \frac{C_2 m \Delta^2}{f(G)}$ Type-II estimators for some constants C_1 and C_2, to achieve ϵ-approximation in the worst case; When $k = 5$, we need $r \geq \frac{C_3 m^2 \Delta}{f(G)}$ Type-I estimators, or $r \geq \frac{C_4 m^2 \Delta}{f(G)}$ Type-II.a estimators, or $r \geq \frac{C_5 m \Delta^3}{f(G)}$ 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 4.3.3, we know that $E[X_i] = f(G)$ and $E[\bar{X}] = f(G)$. From the properties of graph G, we have $c(e) \leq \Delta$ 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, $X_i \leq m^2$ 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

API	Description
`sampleVertex`: () $\rightarrow (v, p)$	Uniformly sample one vertex from the graph.
`SampleEdge`: () $\rightarrow (e, p)$	Uniformly sample one edge from the graph.
`ConditionalSampleVertex`: (`subgraph`) $\rightarrow (v, p)$	Uniformly sample a vertex that appears after a sampled subgraph.
`ConditionalSampleEdge`: (`subgraph`) $\rightarrow (e, p)$	Uniformly sample an edge that is adjacent to the given subgraph and comes after the subgraph in the order.
`ConditionalClose`: (`subgraph`, `subgraph`) $\rightarrow boolean$	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.

expectation $f(G)$ is $Pr[\bar{X} > (1+\epsilon)f(G)] \leq \frac{\delta}{2}$, which is at most

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

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

In Type-II, $X_i \leq 6m\Delta^2$. 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, $r \geq \frac{18m\Delta^2}{\epsilon^2 f(G)} \cdot \ln(\frac{2}{\delta})$. 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 $O(r)$. $\square$

4.3.1.2 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.

- `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

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.

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 $1/(p_1 \cdot p_2)$.

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

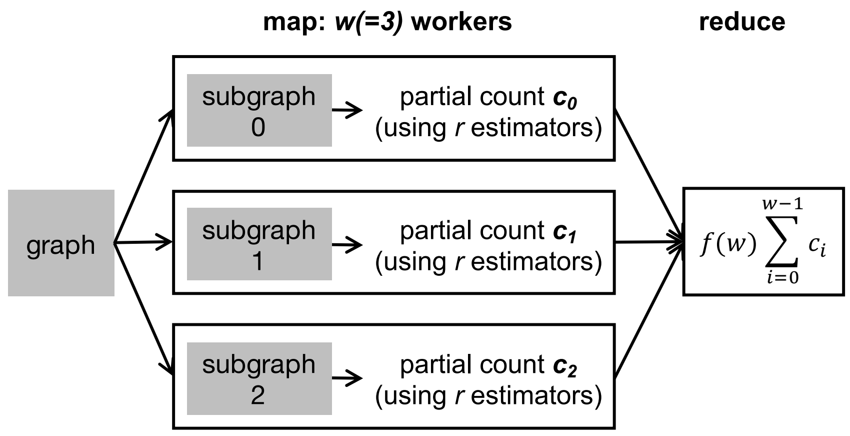

map: *w(=3)* workers
reduce
graph
subgraph 0
partial count c_0 (using *r* estimators)
subgraph 1
partial count c_1 (using *r* estimators)
subgraph 2
partial count c_2 (using *r* estimators)
$f(w)\sum_{i=0}^{w-1} c_i$

Figure 4.6: Runtime with graph partition.

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).

4.3.2 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]. ASAP first partitions the vertices in the graph across machines and executes many copies of the

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

estimator using standard dataflow operations: *map* and *reduce*. In the map phase, ASAP schedules several copies of the estimator on each of the machines. Each estimator operates on the local subgraph in each machine and produces an output, which is a partial count. ASAP's 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., 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 [16] 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 chapter. 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 [138]. Similarly, each monochromatic 4-clique is scaled by $f(w) = w^3$ and $f(w)$ can be computed for any given pattern.

4.3.3 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 [147]. 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 4.5.2.

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.

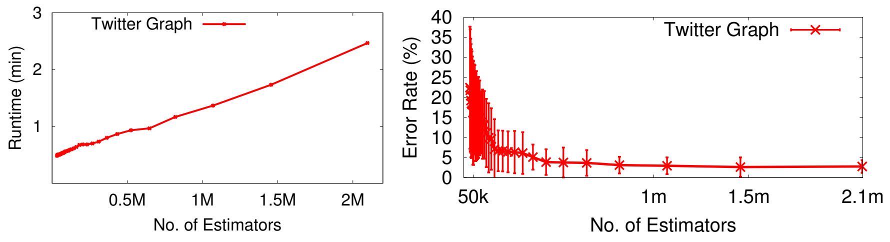


Figure 4.7: The actual relations between number of estimators and run-time or error rate.

4.4 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 [136]. To enable the user to traverse this trade-off, ASAP needs to determine the correct number of estimators given an error or time budget.

Algorithm 5 BuildTimeProfile(T^*)

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

4.4.1 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 5 shows 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.

4.4.2 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 [148] or Bayesian optimization [149] 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(section 4.1.2.1). This probability bound can be translated to an estimator of form $N_e > \frac{K*m*\Delta}{\epsilon^2 P}$ (Theorem 3.3 [142]) 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 [142]. 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 $\widehat{P_s}$, a value very close to the ground truth for the sampled graph.

(c) Using $\widehat{P_s}$ 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 $\widehat{P_s}$ to the larger graph to compute N_e. We note that the scaled $\widehat{P}$ might not be close to P for the larger graph. But as we use the worst case bound to compute $\widehat{P_s}$, the computed value of N_e offers a good bound in practice for the larger graph.

4.4.3 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 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 [150] 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 section 4.4.2.

4.5 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.
- 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.

Graph	Nodes	Edges	Degrees
CiteSeer [129]	3,312	4732	2.8
MiCo [129]	100,000	1,080,298	22
Youtube [151]	1,134,890	2,987,624	8
LiveJournal [151]	3,997,962	34,681,189	17
Twitter [136]	41.7 million	1.47 billion	36
Friendster [152]	65.5 million	1.80 billion	28
UK [137, 153]	105.9 million	3.73 billion	35

Table 4.2: Graph datasets used in evaluating ASAP.

Implementation. We built ASAP on Apache Spark [146], a general purpose dataflow engine. The implementation uses GraphX [19], 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 [13] (using its open-source release [154] built on Apache Giraph [155]) on four smaller graphs: CiteSeer [129], Mico [129], Youtube [151], and LiveJournal [151]. 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 (section 4.1) 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.

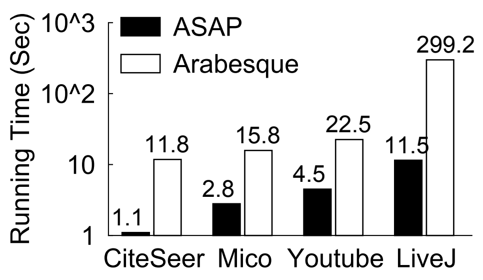
Patterns and Metrics. For evaluating ASAP, we use two types of patterns, *motif*s and *clique*s. For motifs, we consider 3-motifs (consisting of 2 individual patterns), and 4-motifs (consisting of 6 individual patterns) and for cliques, we consider 4-cliques. For our experiments, we run 10 trials for 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 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.

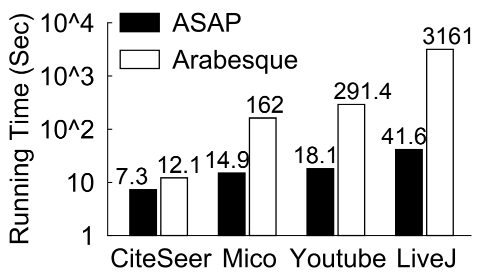
4.5.1 Overall Performance

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 section 4.5.3.

Comparison with Arabesque. In this experiment, we compare Arabesque



(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.

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. 4.8(a) and 4.8(b).

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.

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

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	20x32	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
	20x32	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.

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 [13] 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$\times$ 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.

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.

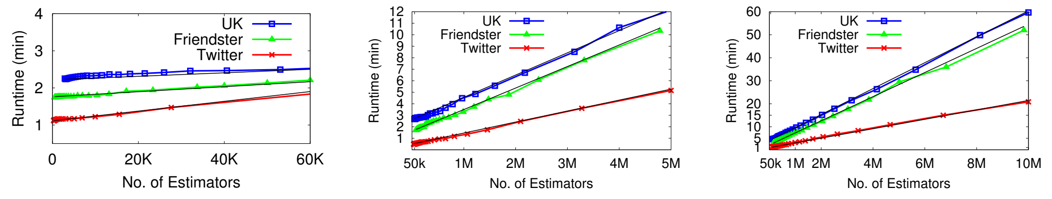
4.5.2 Advanced Pattern Mining

We next evaluate the advanced pattern mining capabilities in ASAP described in section 4.3.3.

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.



(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.

4.5.3 Effectiveness of ELP Techniques

Here, we evaluate the effectiveness of the ELP building techniques in ASAP, described in section 4.4.

Time Profile. To evaluate how well our time profiling technique (Section 4.4.1) 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 Figure 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 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

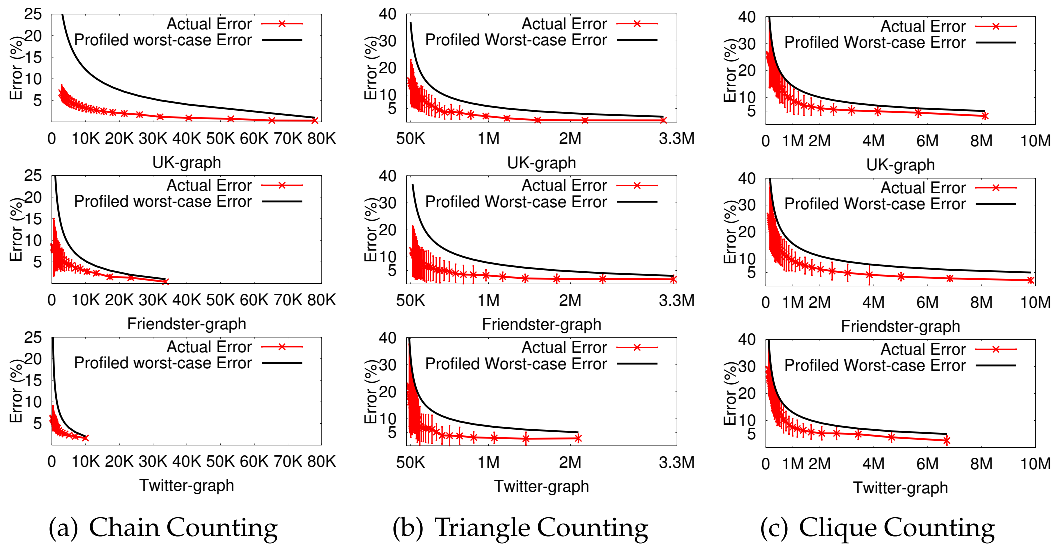


Figure 4.10: Error vs. number of estimators for Twitter, Friendster, and UK graphs.

ASAP's technique of using a small portion of the graph to build the profile. We show both in Figure 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 Section 4.5.1, 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 (Section 4.4.2) is a viable approach.

Error rate Confidence. In Figure 4.11, we evaluate the cumulative distribution

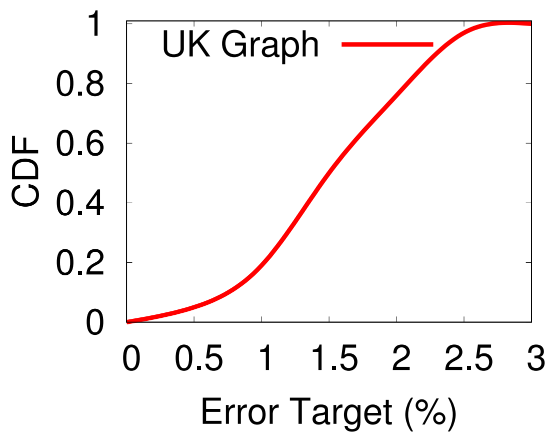


Figure 4.11: CDF of 100 runs with 3% error target.

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

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.

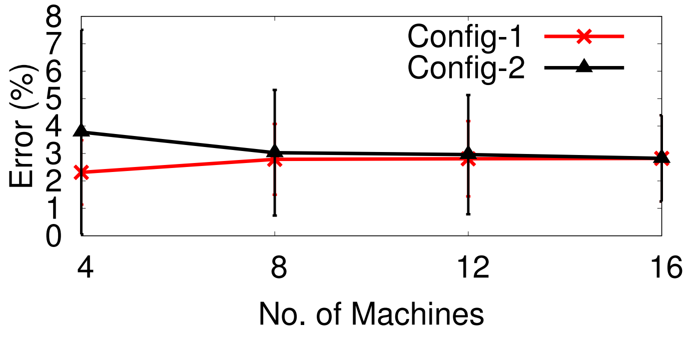


Figure 4.12: The errors from two cluster scenarios with different number of nodes. Config-1:*strong-scaling* to fix the total number of estimators as 2M $\times$ 128; Config-2: *weak-scaling* to fix the number of estimators per executor as 2M.

4.5.4 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 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.

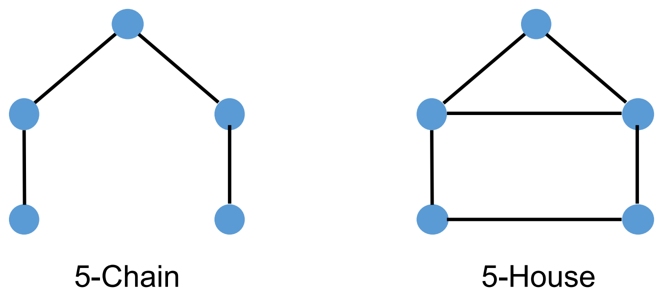


Figure 4.13: Two representative (from 21) patterns in 5-Motif.

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.

4.5.5 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 (Figure 4.13) in Table 4.6. As we see, ASAP is able to handle complex patterns on large graphs easily.

4.6 Related Work

A large number of systems have been proposed in the literature for **graph processing** [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]. Of these, some [15, 17, 18] 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 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 [13]. 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** [130, 131, 132] 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 [156, 157, 158, 159, 160, 161]. 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.

4.7 Chapter 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

Streaming Algorithms for Halo Finders

The goal of astrophysics is to explain the observed properties of the universe we live in. In cosmology in particular, one tries to understand how matter is distributed on the largest scales we can observe. In this effort, advanced computer simulations play an ever more important role. Simulations are currently the only way to accurately understand the nonlinear processes that produce cosmic structures such as galaxies and patterns of galaxies. Hence a large amount of effort is spent on running simulations modelling representative parts of the universe in ever greater detail. A necessary step in the analysis of such simulations involves locating mass concentrations, called "haloes", where galaxies would be expected to form. This step is crucial to connect theory to observations – galaxies are the most observable objects that trace the large-scale structure, but their precise spatial distribution is only established through these simulations.

Many algorithms have been developed to find these haloes in simulations.

The algorithms vary widely, even conceptually. There is no absolutely agreed-upon physical definition of a halo, although all algorithms give density peaks, i.e. clusters of particles. Galaxies are thought to form at these concentrations of matter. Some codes find regions inside an effective high-density contour, such as Friends-of-Friends (FoF) [162]. In FoF, particles closer to each other than a specified linking length are gathered together into haloes. Other algorithms directly incorporate velocity information as well. Another approach finds particles that have crossed each other as compared to the initial conditions, which also ends up giving density peaks [163]. FoF is often considered to be a standard approach, if only because it was among the first used, and is simple conceptually. The drawbacks of FoF include that the simple density estimate can artificially link physically separate haloes together, and the arbitrariness of the linking length. A halo-finding comparison project [164] evaluated the results of 17 different halo-finding algorithms; further analysis appeared in [29]. We take the FoF algorithm as a fiducial result for comparison, but compare to results from some other finders, as well.

Halo-finding algorithms are generally computationally intensive, often requiring all particle positions and velocities to be loaded in memory simultaneously. In fact most are executed during the execution of the simulation itself, requiring comparable computational resources. However, in order to understand the systematic errors in such algorithms, it is often necessary to run multiple halo-finders, often well after the original simulation has been run. Also, many of the newest simulations have several hundred billion to a trillion particles, with a very large memory footprint, making such posterior computations quite difficult. Here, we investigate a way to apply streaming algorithms as halo finders, and compare the results to those of other algorithms participating in the Halo-Finding Comparison Project.

Recently, streaming algorithms [59] have become a popular way to process massive data sets. In the streaming model, the input is given as a sequence of items and the algorithm is allowed to make a single or constant number of passes over the input data while using sub-linear, usually poly-logarithmic space compared to the storage of the data. Streaming algorithms have found many applications in networking ([165, 10, 36]), machine learning ([166, 167]), financial analytics ([168, 169, 170]) and databases ([171, 172]).

In this chapter, we apply streaming algorithms to the area of cosmological simulations and provide space and time efficient solutions to the halo finding problem. In particular, we show a relation between the problem of finding haloes in the simulation data and the well-known problem of finding "heavy hitters" in the streaming data. This connection allows us to employ efficient heavy hitter algorithms, such as Count-Sketch [6] and Pick-and-Drop Sampling [41]. By equating heavy hitters to haloes, we are implicitly defining haloes as positions exceeding some high density threshold. In our case, these usually turn out to be density peaks, but only because of the very spiky nature of the particle distributions in cosmology. Conceptually, FoF haloes are also regions enclosed by high density contours, but in practice, the FoF implementation is very different from ours.

5.1 Streaming Algorithm

In this section, we investigate the application of streaming algorithms to find haloes using a strong relation between the halo-finding problem and the heavy hitter problem, which we discuss in section 5.1.1.4. Heavy hitter algorithms find the k densest regions, that may physically correspond to haloes. In our implementation, we carefully choose k to get the desired outcome. This parameter k is as also discussed in section 5.1.1.4. We first present in the next sub-section the formal definition of streaming algorithms and the connection between heavy hitter problem and halo-finding problem. After that, we presents the basic procedures of the two heavy hitter algorithms: Count-Sketch and Pick-and-drop Sampling.

5.1.1 Streaming Data Model

5.1.1.1 Definitions

A data stream $D = D(n, m)$ is an ordered sequence of objects $p_1, p_2, \ldots, p_n$, where $p_j = 1 \ldots m$. The elements of the stream can represent any digital object: integers, real numbers of fixed precisions, edges of a large graphs, messages, images, web pages, etc. In the practical applications both n and m may be very large, and we are interested in the algorithms with $o(n + m)$ space. A streaming algorithm is an algorithm that can make a single pass over the input stream. The above constraints imply that a streaming algorithm is often a randomized algorithm that provides approximate answers with high probability. In practice, these approximate answers often suffice.

We investigate the results of cosmological simulations where the number of particles will soon reach 10^{12}. Compared to offline algorithms that require the input to be entirely in memory, streaming algorithms provide a way to process the data using only megabytes memory instead of gigabytes or terabytes in practice.

5.1.1.2 Heavy Hitter

For each element i, its frequency f_i is the number of its occurrences in D. The k^{th} frequency moment of a data stream D is defined as $F_k(D) = \sum_{i=1}^{m} f_i^k$. We say that an element is "heavy" if it appears more times than a constant fraction of some L_p norm of the stream, where $L_p = (\sum_i f_i^p)^{1/p}$ for $p > 1$. In this chapter, we consider the following heavy hitter problem.

Problem 2 . Given a stream D of n elements, the ϵ-approximate (ϕ, L_p)-heavy hitter problem is to find a set of elements T:

- $\forall i \in [m], f_i > \phi L_p \implies i \in T$.
- $\forall i \in [m], f_i < (\phi - \epsilon) L_p \implies i \notin T$.

We allow the heavy hitter algorithms to use randomness; the requirement is that the correct answer should be returned with high probability. The heavy hitter problem is equivalent to the problem of approximately finding the k most frequent elements. Indeed, the top k most frequent elements are in the set of (ϕ, L_1)-heavy hitters in the stream, where $\phi = \Theta(1/k)$. There is a $\Omega(1/\epsilon^2)$ trade-off between the approximation error ϵ and the memory usage.

Heavy hitter algorithms are building blocks of many data stream algorithms ([42, 173]).

We treat the cosmological simulation data from [164] as a data stream. To do so, we apply an online transformation that we describe in the next section.

5.1.1.3 Data Transformation

In a cosmological simulation, dark matter particles form structures through gravitational clustering in a large box with periodic boundary conditions representing a patch of the simulated universe. The box we use [164] is of size 500 Mpc/h, or about 2 billion light-years. The simulation data consists of positions and velocities of 256^3, 512^3 or 1024^3 particles, each representing a huge number of physical dark-matter particles. They are distributed rather uniformly on large scales ($\gtrsim$ 50 Mpc/h) in the simulation box, clumping together on smaller scales. A halo is a clump of particles that are gravitationally bound.

To apply the streaming algorithms, we transform the data. We discretize the spatial coordinates so that we will have a finite number of types in our transformed data stream. We partition the simulation box into a grid of cubic cells, and bin the particles into them. The cell size is chosen to be 1 Mpc/h as to match a typical size of a large halo; there are thus 500^3 cells. This parameter can be modified in practical applications, but it relates to the space and time efficiency of the algorithm. We summarize the data transformation steps as follows.

- Partition the simulation box into grids of cubic cells. Assign each cell a unique integer ID.
- After reading a particle, determine its cell. Insert that cell ID into the data stream.

Using the above transformation, streaming algorithms can process the particles in the same way as an integer data stream.

5.1.1.4 Heavy Hitter and Dense Cells

For a heavy-hitter algorithm to save memory and time, the distribution of cell counts must be very non-uniform. The simulations begin with an almost uniform lattice of particles, but after gravity clusters them together, the density distribution in cells can be modeled by a lognormal PDF ([174], [175]):

$$P_{LN}^{(1)}(\delta) = \frac{1}{(2\pi\sigma_1^2)^{1/2}} \exp\left\{ -\frac{[\ln(1+\delta) + \sigma_1^2/2]^2}{2\sigma_1^2} \right\} \frac{1}{1+\delta}, \tag{5.1.1}$$

where $\delta = \rho/\bar{\rho} - 1$ is the overdensity, $\sigma_1^2(R) = \ln[1 + \sigma_{\rm nl}^2(R)]$, and $\sigma_{\rm nl}^2(R)$ is the variance of the nonlinear density field in spheres of radius R. Our cells are cubic, not spherical; for theoretical estimates, we use a spherical top-hat of the same volume as a cell.

Let N be the number of cells, and P_c be the distribution of the number of particles per cell. The L_p heaviness ϕ_p can be estimated as

$$\phi_p \approx \frac{P_{200}}{(N\langle {P_c}^p \rangle)^{1/p}}, \tag{5.1.2}$$

where P_{200} is the number of particles in a cell with density exactly $200\bar{\rho}$. This density threshold is a typical minimum density of a halo, coming from the spherical-collapse model. We theoretically estimated σ_{nl} for the cells in our density field by integrating the nonlinear power spectrum (using the fit of [176], and the cosmological parameters of the simulation) with a spherical tophat window. The grid size in our algorithm is roughly 1.0 Mpc (500^3 cells in total), giving $\sigma_{\mathrm{nl}}(\mathrm{Cell}) \approx 10.75$. We estimated $\phi_1 \approx 10^{-6}$ and $\phi_2 \approx 10^{-3}$, matching order-of-magnitude with the measurement of the actual density variance from the simulation cells. These heaviness values are low enough to presume that a heavy-hitter algorithm will efficiently find cells corresponding to haloes.

5.1.2 Streaming Algorithms for Heavy Hitter Problem

The above relation between the halo-finding problem and the heavy hitter problem encourages us to apply efficient streaming algorithms to build a new halo finder. Our halo finder takes a stream of particles, performs the data transformation described in section 5.1.1.3 and then applies a heavy hitter algorithm to output the approximate top k heavy hitters in the transformed stream. These heavy hitters correspond to the densest cells in the simulation data as described in section 5.1.1.4. In our first version of the halo finder, we use Count-Sketch algorithm [6] and Pick-and-Drop Sampling [41].

5.1.2.1 The Count-Sketch Algorithm

For a more generalized description of the algorithm, please refer to [6]. For completeness, we summarize the algorithm as follows. The Count-Sketch algorithm uses a compact data structure to maintain the approximate counts of the top k most frequent elements in a stream. This data structure is an $r \times t$ matrix M representing estimated counts for all elements. These counts are calculated by two sets of hash functions: let $h_1, h_2, \ldots, h_r$ be r hash functions, mapping the input items to $\{1, \ldots, t\}$, where each h_i is sampled uniformly from the hash function set H. Let $s_1, s_2, \ldots, s_r$ be hash functions, mapping the input items to $\{+1, -1\}$, uniformly sampled from another hash function set S. We can interpret this matrix as an array of r hash tables, each containing t buckets.

There are two operations on the Count-Sketch data structure. Denote $M_{i,j}$ as the j^{th} bucket in the i^{th} hash table:

- $Add(M, p)$: For $i \in [1, r]$, $M_{i,h_i[p]} += s_i[p]$.
- $Estimate(M, p)$, return $median_i\{h_i[p] \cdot s_i[p]\}$

The *Add* operation updates the approximate frequency for each incoming element and the *Estimate* operation outputs the current approximate frequency. To maintain and store the estimated k most frequent elements, CountSketch also needs a priority queue data structure. The pseudocode of

```
procedure CountSketch($r, t, k, D$)                    ▷ $D$ is a stream
    Initialize an empty $r \times t$ matrix $M$.
    Initialize an min-priority queue $Q$ of size $k$
    (particle with smallest count is on the top).
    for $i = 1, \dots, n$ and $p_i \in D$ do
        $Add(M, p_i)$;
        if $p_i \in Q$ then
            $P_i.count{+}{+}$;
        else if $Estimate(M, p_i) > Q.top().count$ then
            $Q.pop()$;
            $Q.push(p_i)$;
    return $Q$
```

Figure 5.1: Count-Sketch Algorithm

Count-Sketch algorithm is presented in Figure 5.1. More details and theoretical guarantees are presented in [6].

5.1.2.2 The Pick-and-Drop Sampling Algorithm

Pick-and-Drop Sampling is a sampling-based streaming algorithm to approximate the heavy hitters. To describe the idea of Pick-and-Drop sampling, we view the data stream as a sequence of r blocks of size t. Define $d_{i,j}$ as the j^{th} element in the i^{th} block and $d_{i,j} = p_{k(i-1)+j}$ in stream D. In each block of the stream, Pick-and-Drop sampling will pick one random sample and record its remaining frequency in the block. The algorithm maintains a sample with the largest current counter and drops previous samples. The pseudocode of Pick-and-Drop sampling [41] is given in Figure 5.2 and we need the following definitions in Figure 5.2. For $i \in [r]$, $j, s \in [t]$, $q \in [m]$ define:

$$f_{i,q} = |\{j \in [t] : d_{i,j} = q\}|, \tag{5.1.3}$$

$$a_{i,s} = |\{j^* : s \le j^* \le t, d_{i,j^*} = d_{i,s}\}|. \tag{5.1.4}$$

```
1: procedure PickDrop($r, t, \lambda, D$)
2:     Sample $S_1$ uniformly at random on $[t]$.
3:     $L_1 \leftarrow d_{1,S_1}$,
4:     $C_1 \leftarrow a_{1,S_1}$,
5:     $u_1 \leftarrow 1$.
6:     for $i = 2, \ldots, r$ do
7:         Sample $S_i$ uniformly at random on $[t]$.
8:         $l_i \leftarrow d_{i,S_i}$, $c_i \leftarrow a_{i,S_i}$
9:         if $C_{i-1} < \max(c_i, \lambda u_{i-1})$ then
10:            $L_i \leftarrow l_i$,
11:            $C_i \leftarrow c_i$,
12:            $u_i \leftarrow 1$
13:        else
14:            $L_i \leftarrow L_{i-1}$,
15:            $C_i \leftarrow C_{i-1} + f_{i,L_{i-1}}$,
16:            $u_i \leftarrow q_{i-1} + 1$
17:    return $\{L_r, C_r\}$
```

Figure 5.2: Pick-and-Drop Algorithm

The detail implementation is in Section 5.2.2.

5.2 Implementation

5.2.1 Simulation Data

The N-body simulation data we use as the input to our halo finder was used in the halo-finding comparison project [164] and consists of various resolutions (numbers of particles) of the MareNostrum Universe cosmological simulation [177]. These simulations ran in a 500 h^{-1}Mpc box, assuming a standard ΛCDM (cold dark matter and cosmological constant) cosmological model.

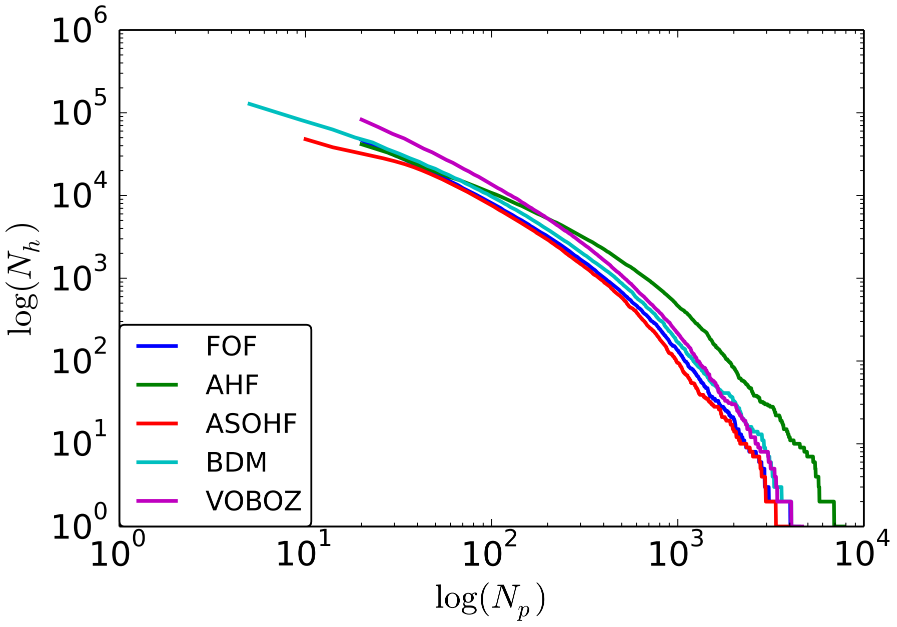


Figure 5.3: Halo mass distribution of various halo finders.

In the first implementation of our halo finder, we consider two halo properties: center position and mass (the number of particles in it). We compare to the the fiducial offline algorithm FoF. The distributions of halo sizes from different halo finders are presented in Fig. 5.3.

Since our halo finder builds on the streaming algorithms of finding frequent items, the algorithms need to transform the data as described in section 5.1.1.3 — dividing all the particles into different small cells and label each particle with its associated cell ID. For example, if an input dataset contains three particles p_1, p_2, p_3 and they are all included in a cell of ID $= 1$, then the transformed data stream becomes $1, 1, 1$. The most frequent element in the stream is obviously 1 and thus the cell 1 is the heaviest cell overall.

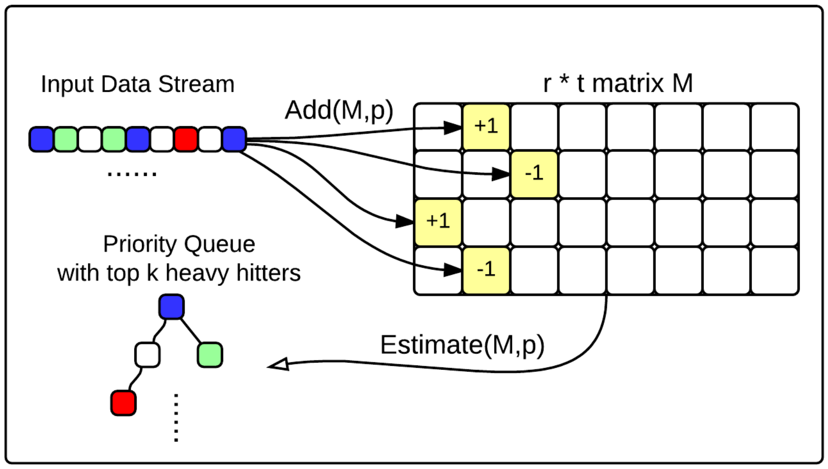


Figure 5.4: Count-Sketch Algorithm

5.2.2 Implementation Details

Our halo finder implementation is written using C++ with GNU GCC compiler 4.9.2. We implemented Count-Sketch and Pick-and-Drop sampling as two algorithms to find heavy hitters.

5.2.2.1 Count-Sketch-based Halo Finder

There are three basic steps in the Count-Sketch algorithm, which returns the heavy cells and the number of particles associated with them. (1) Allocate memory for the CountSketch data structure to hold current estimates of cell frequencies; (2) use a priority queue to record the k most frequent elements; (3) return the positions of the top k heavy cells. Figure 5.4 presents the process of the Count-Sketch.

The Count-Sketch data structure is an $r \times t$ matrix. Following [6], we set

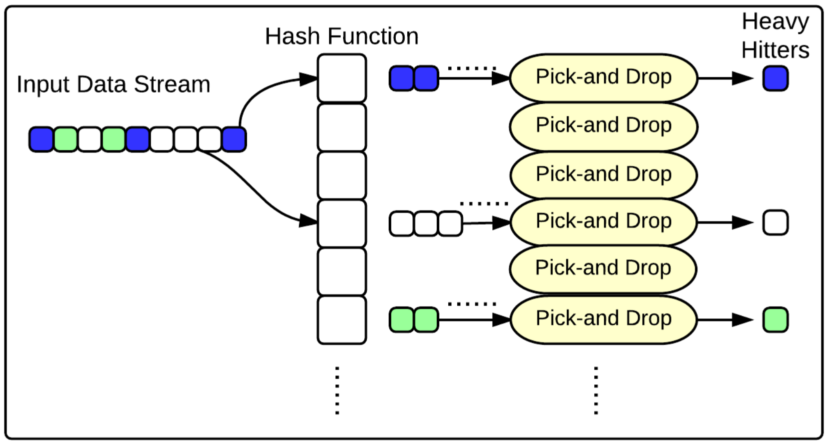


Figure 5.5: Pick-and-Drop Sampling

$r = \log(\frac{n}{\epsilon})$ and t to be sufficiently large (>1 million) to achieve an expected approximation error $\epsilon = 0.05$. We build the matrix as a 2D array with $r \times t$ 0's. For each incoming element in the stream, an *Add* operation has to be executed and an *estimate* operation needs to be executed only when this element is not in the queue.

5.2.2.2 Pick-and-Drop-based Halo Finder

In the Pick-and-Drop sampling based halo finder, we implement a general hash function H: $\mathbb{N}^+ \rightarrow \{1, 2, \ldots, ck\}$, where $c \geq 1$, to gain the probability of success to approximate the k heaviest cells. We apply the hash function H on every incoming element and put the elements with the same hash value together such that the original stream is divided into ck smaller sub-streams. Meanwhile, we initialize ck instances of Pick-and-Drop sampling so that each PD instance will process one sub-stream. The whole process of approximating

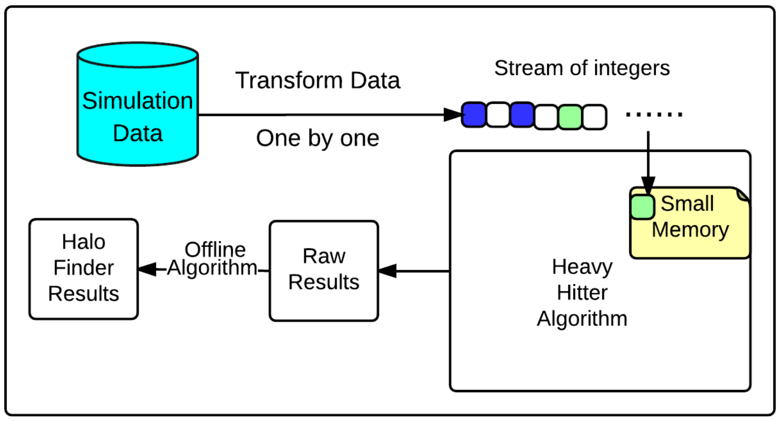


Figure 5.6: Halo Finder Procedure

the heavy hitters is presented in Figure 5.5. In this way, the repeated items in the whole stream will be distributed into the same sub-stream and they are much heavier in this sub-stream. With high probability, each instance of Pick-and-Drop sampling will output the heaviest one in each of the sub-streams, and in total we will have ck output items. Because of the randomness in the sampling method, we will expect some of inaccurate heavy hitters among the total ck outputs. By setting a large c, most of the actual top k most frequent elements should be inside the ck outputs (raw result).

To get precise properties of haloes, such as the center, and mass, an offline algorithm such as FoF [162] can be applied to the particles inside the returned heavy cells and their neighbor cells. This needs an additional pass over the data but we only need to store a small amount of particles to run those offline in-memory algorithms. The whole process of the halo finder is represented in Figure 5.6, where heavy hitter algorithms can be regarded as a black box. That is, any theoretically efficient heavy hitter algorithms could be applied to further improve the memory usage and practical performance.

5.2.3 Shifting Method

In the first pass of our halo finder, we only use the position of a heavy cell as the position of a halo. However, each heavy cell may contain several haloes and some of the haloes located on the edges between two cells cannot be recognized because the cell size in the data transformation step is fixed. To recover those missing haloes, we utilize a simple shifting method:

- Initialize 2^d instances of Count-Sketch or Pick-and-Drop in parallel, where d is the dimension. Our simulation data reside in three dimensions, so $d = 3$.
- Move all the particles to one of the 2^d directions with a distance of 0.5 Mpc/h (half of the cell size). In each of the 2^d shifting processes, assign a Count-Sketch/Pick-and-Drop instance to run. By combining the results from 2^d shifting processes, we expect that the majority of k largest haloes are discovered. All the parallel instances of the CountSketch/Pick-and-Drop are enabled by OpenMP 4.0 in C++.

5.3 Evaluation

To evaluate how well streaming based halo finders work, we mainly focus on testing it in the following four aspects:

- **Correctness:** Evaluate how close are the positions of k largest haloes found by the streaming-based algorithms to the top k large haloes returned by some widely used in-memory algorithms. Evaluate the trade-off between the selection k and the quality of result.

- **Stability:** Since streaming algorithms always require some randomness and may produce some incorrect results, we want to see how stable are streaming based heavy hitter algorithms are.
- **Memory Usage:** Linear memory space requirement is a "bottle neck" for all offline algorithms, and it is the central problem that we are trying to overcome by applying streaming approach. Thus it is significantly important to theoretically or experimentally estimate the memory usage of Pick-and-Drop and Cound-sketch algorithms.

In the evaluation, all the in-memory algorithms we choose to compare were proposed in the Halo-Finding Comparison Project [164]. We test against the fiducial FOF method, as well as four others that find density peak:

1. **FOF** by Davis et al.[162]
   "Plain-vanilla" Friends-of-Friends.
2. **AHF** by Knollmann & Knebe [178]
   Density peaks search with recursively refined grid
3. **ASOHF** by Planelles & Quilis. [179]
   Finds spherical-overdensity peaks using adaptive density refinement.
4. **BDM** [180], run by Klypin & Ceverino "Bound Density Maxima" – finds gravitationally-bound spherical-overdensity peaks.
5. **VOBOZ** by Neyrinck et al [181]
   "Voronoi BOund Zones" – finds gravitationally bound peaks using a

Voronoi tessellation.

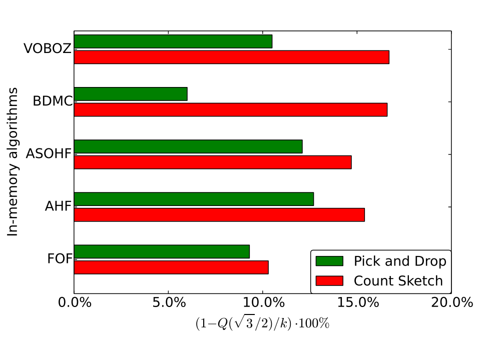
5.3.1 Correctness

As there is no agreed upon rule how to define the center and the boundary of a halo, it is impossible to theoretically define and deterministically verify the correctness of any halo finder. Therefore a comparison to the results of previous widely accepted halo finders seems to be the best practical verification of a new halo finder. To compare the outputs of two different halo finders we need to introduce some formal measure of similarity. The most straight forward way to compare them is to consider one of them H as a ground truth, and another one E as an estimator. Among this the FOF algorithm is considered to be the oldest and the most widely used, thus in our initial evaluation we decided to concentrate on the comparison with FOF. Then the most natural measure of similarity is number of elements in H that match to elements in output of E. More formally we will define "matches" as: for a given θ we will say that center $e_i \in E$ matches the element $h_i \in E$ if $dist(e_i, h_i) \leq \theta$, where $dist(\cdot, \cdot)$ is Euclidean distance. Then our measure of similarity is:

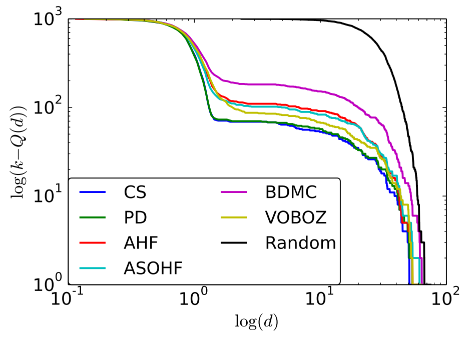
$$Q(\theta) = Q(E_k, H_k, \theta) = |\{h_i \in H_k : \min_{e_j \in E_k} dist(h_i, e_j) < \theta\}|,$$

where k represents k heaviest halos.

We compare the output of both streaming-based halo finders to the output of in-memory halo finders. We made comparisons for the $256^3, 512^3$ and 1024^3-particle simulations, finding the top 1000 and top 10000 heavy hitters. Since the comparison results in all cases were similar, the figures presented below



(a)



(b)

Figure 5.7: (a) Measure of the disagreement between PD and CS, and various in-memory algorithms. The percentage shown is the fraction of haloes farther than a half-cell diagonal ($0.5\sqrt{3}$ Mpc/h) from PD or CS halo positions. (b) The number of top-1000 FoF haloes farther than a distance d away from any top-1000 halo from the algorithm of each curve.

are for the 256^3 dataset, and $k = 1000$.

On the Figure 5.7(a) we show for each in-memory algorithm the percentage of centers that were not found by streaming-based halo finder. We can see that both the Count-Sketch and Pick-and-Drop algorithms missed not much more than 10 percent of the haloes in any of the results from the in-memory algorithms.

To understand whether the 10 percent means two halo catalogs are close to each other or not, we will choose one of the in-memory algorithms as a ground truth and compare how close the other in-memory algorithms are. Again, we choose FOF algorithm as a ground truth. The comparison is depicted in Fig. 5.7(b). From this graph you can see that the outputs of Count-Sketch and Pick-and-Drop based halo finders are closer to the FOF haloes, than other in-memory algorithms. It can be easily explained, as after finding heavy cells we apply the same FOF to these heavy cells and their neighborhoods, the

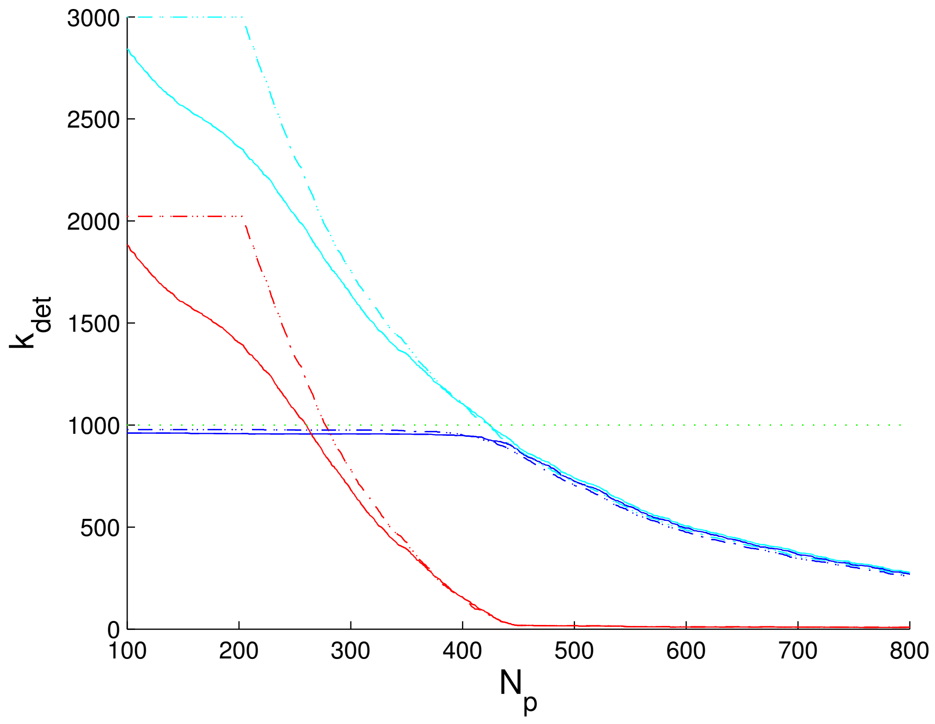


Figure 5.8: Number of detected halos by our two algorithms. The solid lines correspond to (CS) and the dashed lines to (PD). The dotted line at $k = 1000$ shows our selection criteria. The x axis is the threshold in the number of particles allocated to the heavy hitter. The cyan color denotes the total number of detections, the blue curves are the true positives (TP), and the red curves are ethe false positives (FP).

output should always have similar structure to the output of in-memory FOF on the full dataset. Also from this graph you will see that each line can be represented as a mix of two components, one of which is the component of random distribution. It means that after a distance of $\sqrt{3}/2$ all matches are the same if we just put bunch of points at random.

The classifier is using a top-k to select the halo candidates. Figure 5.8 shows how sensitive the results are to the selection threshold of $k = 1000$. It shows several curves, including the total number of heavy hitters, the ones close to

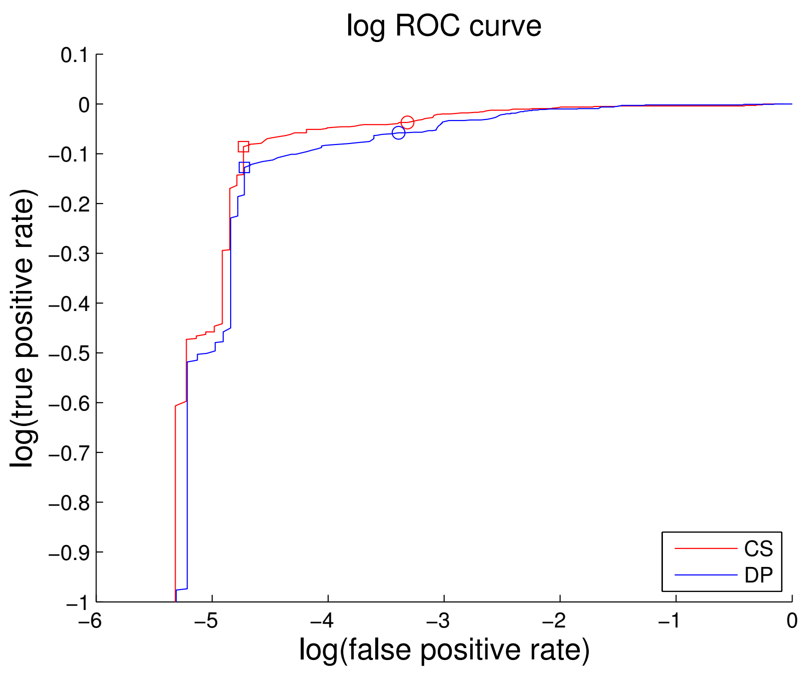


Figure 5.9: This ROC curve shows the tradeoff between true and false detections as a function of threshold. The figure plots TPR vs FPR on a log-log scale. The two thresholds are shown with symbols, the circle denotes 1000, and the square is 900.

an FoF group – we can call these true positive (TP) – and the ones detected, but not near an FoF object (false positives FP). From the figure, it is clear that the threshold of 1000 is close to the optimal detection threshold, preserving TP and minimizing FP. This corresponds to a true positive detection rate (TPR) of 96% and a false positive detection rate of 3.6%. If we lowered our threshold to $k = 900$, our TPR drops to 91% but the FPR becomes even lower, 0.88%.

These tradeoffs can be shown in Fig. 5.9 by a so-called ROC-curve (receiver operating characteristic), where the TPR is plotted against the FPR. This shows how lowering the detection threshold increases the true detections, but the false detection rate increases much faster. Using the ROC curve, shown below we can see the position of the $k = 1000$ threshold as a circle and the $k = 900$ as

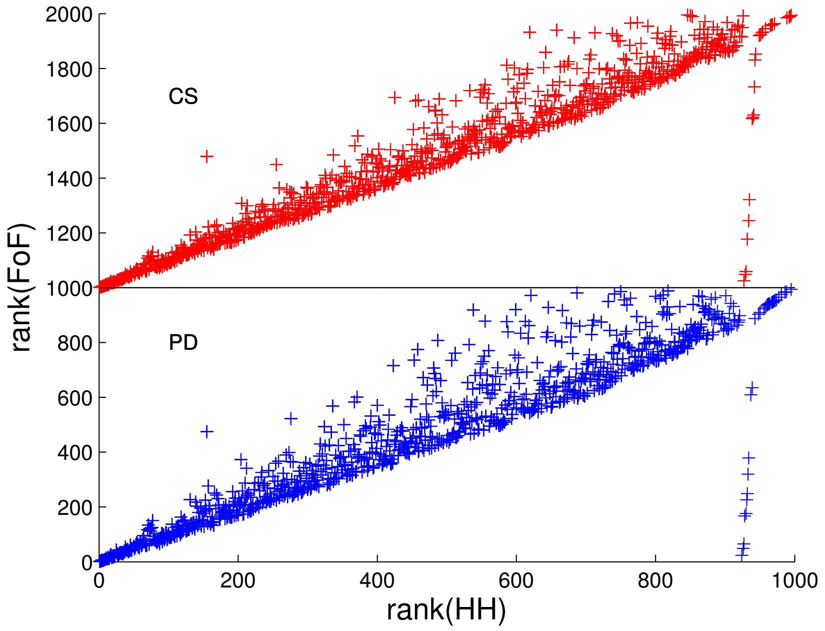


Figure 5.10: The top 1000 heavy hitters are rank-ordered by the number of their particles. We also computed a rank of the corresponding FoF halo. The linked pairs of ranks are plotted. One can see that if we adopted a cut at $k = 900$, it would eliminate a lot of the false positives.

a square.

Finally, we should also ask, besides the set comparison, how do the individual particle cardinalities counted around the heavy hitters correspond to the FoF ones. Our particle counting is restricted to neighboring cells, while the FoF is not, so we will always be undercounting. To be less sensitive to such biases, we compare the rank ordering of the two particle counts in the two samples in Fig. 5.10. The rank 1 is assigned to the most massive objects in each set.

5.3.2 Stability

As most of the streaming algorithms utilize randomness, we estimate how stable our results are compared to the results from a deterministic search. In the deterministic search algorithm, we find the actual heavy cells by counting the number of particles inside them; we perform the comparison for the dataset containing 256^3 particles. To perform this evaluation we run 50 instances of each algorithm (denoting the outputs as $\{C^i_{cs}\}_{i=1}^{50}$ and $\{C^i_{pd}\}_{i=1}^{50}$). We also count the number of cells of each result that match the densest cells returned by the deterministic search algorithm C_{ds}. The normalized number of matches will be $\rho^i_{pd} = \frac{|C^i_{pd} \cap C_{ds}|}{|C_{ds}|}$ and $\rho^i_{cs} = \frac{|C^i_{cs} \cap C_{ds}|}{|C_{ds}|}$ correspondingly. Our experiment showed:

$$\mu(\rho^i_{cs}) = 0.946,\ \sigma(\rho^i_{cs}) = 2.7 \cdot 10^{-7}$$

$$\mu(\rho^i_{pd}) = 0.995,\ \sigma(\rho^i_{pd}) = 6 \cdot 10^{-7}$$

This means that the approximation error caused by randomness is very small compared with the error caused by transition from overdense cells to halo centers. This fact can also be caught from the Fig. 5.11. On that figure you can see that shaded area below and above the red line and green line, which represents the range of outputs among 50 instances, is very thin. Thus the output is very stable.

5.3.3 Memory Usage

Comparing with current halo finding solutions, streaming approachs' low memory usage is one of the most significant advantages. To the best of our

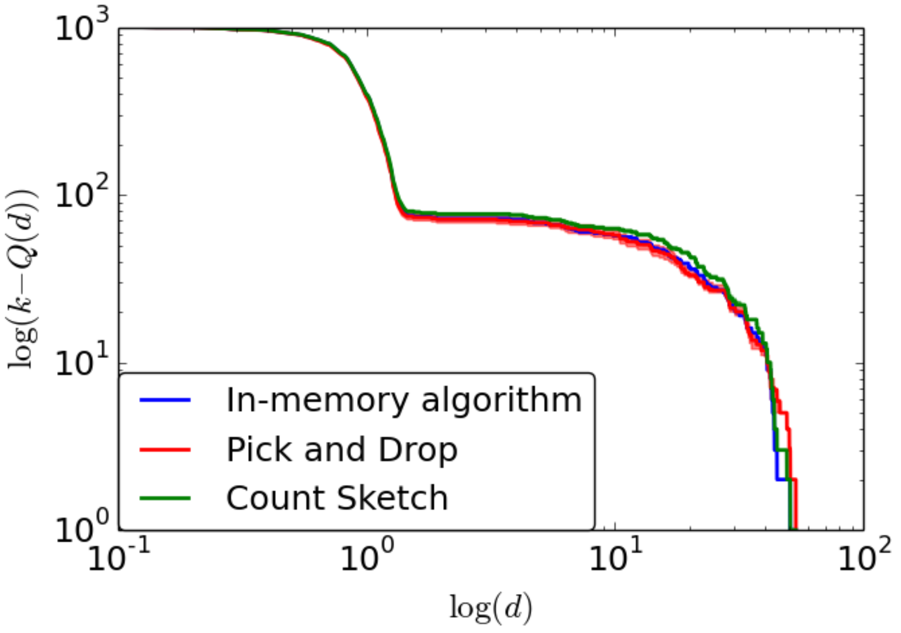


Figure 5.11: Each line on the graph represents the top 1000 halo centers found with Pick-and-Drop sampling, Count-Sketch, and in-memory algorithms. The shaded area (too small to be visible) shows the variation due to randomness.

knowledge even for the problem of locating 1000 largest haloes in the simulation data with 1024^3 particles, there is no way to run other halo finding algorithms on a regular PC since 1024^3 particles already need $\approx$ 12GB memory to only store all the particle coordinates; a computing cluster or even supercomputer is necessary. Therefore, the application of streaming techniques introduces a new direction on the development of halo-finding algorithms.

To find top k heavy cells, Count-Sketch theoretically requires following amount of space:

$$O(k \log \frac{n}{\delta} + \frac{\sum_{q'=k+1}^{m} f_{q'}^2}{(\epsilon f_k)^2} \log \frac{n}{\delta}),$$

where $1-\delta$ is probability of success, ϵ is an Q_k estimation error, and Q_k is the frequency of k-th heaviest cell. It is worth mentioning that in application to the heavy cell searching problem the second term is the dominating one. The first factor in the second term represents the linear dependency of memory usage on the heaviness of top k cells. Thus we can expect linear memory usage for small dataset. But as dataset grows the dependency becomes logarithmic if we assume the same level of heaviness. Experiments verify this observation, as for small dataset with 256^3 particles Count-Sketch algorithm used around 900 megabytes memory, while for the large 1024^3-dataset, the memory usage was increased to nearly 1000 megabytes. Thus the memory grows logarithmically with the size of dataset if we assume almost constant heaviness of the top k cells; that is why such approach is scalable for even larger datasets.

In the experiments using this particular simulation data, Pick-and-Drop sampling shows much better performance in terms of memory usage than Count-Sketch. The actual usage of memory was around 20 megabytes for the dataset with 256^3 particles and around 30 megabytes for the dataset with 1024^3 particles.

5.4 Chapter Summary

In this chapter we find a novel connection between the problem of finding the most massive halos in cosmological N-Body simulations and the problem of finding heavy hitters in data streams. According to this link, we have built a halo finder based on the implementation of Count-Sketch algorithm and Pick-and-Drop sampling. The halo finder successfully locates most ($> 90\%$) of the k largest haloes using sub-linear memory. Most halo-finders require the entire simulation to be loaded into memory. But our halo finder does not and could be run on the massive N-body simulations that are anticipated to arrive in the near future with relatively modest computing resources. We will continue to improve the performance of our halo finder, something we have as yet not paid much attention to. In the very first implementation we evaluated here, we mainly focus on the verification of precision instead of performance. But both Count-Sketch and Pick-and-Drop sampling can be easily parallelized further to achieve significantly better performance. The majority of the computation on Count-Sketch is spent on the calculations of $r \times t$ hash functions. A straight forward way to improve the performance is taking advantage of the highly parallel GPU streaming processors to improve the performance of calculating a large number of hash functions. Similarly, Pick-and-Drop sampling is also a good candidate for more parallelism since the Pick-and-Drop instances are running independently.

We also note that this halo finder finds only the k most massive haloes. These are features of interest in the simulation, but some further work is required for our methods to return a complete set of haloes as an in-memory algorithm.

Future work:

1. Optimize the current methods using Count-Sketch and Pick-and-Drop sampling. Our goal is to provide a halo finder tool that can be running on personal PCs or even laptops, and provide comparatively accurate results in a reasonable running time.

2. An application of interest to cosmologists would be to run a streaming algorithm similar to this that includes velocity information; this is

important in distinguishing small “subhaloes” from FoF-type haloes. Including additional attributes/dimensions in our algorithms clustering is quite easy, and will be investigated in the near future.

Chapter 6

Conclusions and Future Work

Building a resource-efficient networked system is challenging as you need to optimize the usage of various kinds of resources, including memory, CPU, cache, and external storage, with diverse hardware. This dissertation work is motivated by the needs of fast and memory-efficient systems for computation-heavy tasks in the contexts of network monitoring, graph analytics, and astrophysics. One key observation is that 100% accuracy may not be necessary for many computation-or-memory-heavy tasks, and thus we can summarize the data using sketches with some accuracy loss. The sketches lead to a significant gain on the efficiency of memory and processing. In conclusion, we briefly summarize the main contributions of the work presented in this dissertation before highlighting some potential avenues for future work.

6.1 Summary of Contributions

We demonstrate the benefits of sketching based design in multiple networked systems. In UnivMon and NitroSketch, we make a step forward to build a robust monitoring system on both hardware and software platforms with guaranteed accuracy for any workloads. In ASAP, we show that a graph pattern mining system that works on large graphs with any graph patterns might be well within our reach. In streaming halo finder, we make it possible to handle large-scale halo finding from N-body simulation data on your own laptop. Specifically, we make the following contributions.

Bottleneck Analysis: Before proposing any algorithmic optimizations to existing solutions, we conduct bottleneck analysis to find out what the real bottlenecks are. In NitroSketch, we instrument a number of sketching based measurement algorithms on two popular software switches (Open vSwitch and VPP), and carefully model the performance bottlenecks in these algorithms.

Algorithmic Design: We design efficient sketching techniques that optimize the network monitoring modules on both hardware and software platforms. In hardware, we build a universal monitoring framework named UnivMon that supports a range of measurement tasks simultaneously with bounded error and memory. Formally, UnivMon works for any functions defined over the frequency vector of the data if the functions computed are not beyond 2nd Frequency Moment. UnivMon is also late-binding, where you do not have to specify the metrics you need to measure beforehand. The evaluation shows UnivMon achieve good memory efficiency while maintaining comparable accuracies with state-of-the-art custom algorithms.

In software, we propose NitroSketch to address the performance bottlenecks identified by our profiling, and minimize the per-packet processing overhead. We construct a geometric sampling based frontend into existing sketching algorithms and further reduce the per-packet hash computation to $o(1)$. We formally prove that by trading a small increase in memory usage, NitroSketch maintains the same error guarantees as existing sketching algorithms under arbitrary workloads. Our evaluation shows that we push the processing performance to the limit of 40Gbps virtual switches, by using only single CPU core.

System Design and Implementation: In UnivMon, we map our data-plane algorithm to P4 and have addressed several practical issues due to hardware limitation, including how to remove the requirement of storing top K flow keys in the data-plane, and how to efficiently compute parallel hash functions with distinct seeds. In NitroSketch, we demonstrate our complete measurement framework by implementing on Open vSwitch and FD.io-VPP. We optimize the software implementations by deploying cached pseudo-random number generator, exploiting AVX2 for parallel hash computations, and probabilistic priority queue updates.

In ASAP, we design a distributed graph pattern miner based on an efficient sketching technique of *Neighborhood Sampling*. We extend the technique to support general patterns in a distributed setting and prove the error bounds. We build ASAP atop Apache Spark and optimize runtime performance by exploiting efficient hash table constructions, estimator caching, and accurate ELP.

In streaming halo finder, we build a single machine system with two heavy hitter algorithms — Count Sketch and Pick-and-Drop. We optimize the memory efficiency and time performance by reducing priority queue operations and parallel hash computations.

6.2 Potential Limitations

Multidimensionality:

The sketching algorithms used in Chapters 2 and 3 are focused on handling one-dimensional data (i.e., any one of the 5-tuple or any one combination of the 5-tuple). If we want to measure the network in a multidimensional fashion, e.g., measure the top K superspreaders (top K source IPs that send traffic to the most number of distinct destination IPs), we need to a version of multidimensional UnivMon. It is unclear how to build a memory-efficient UnivMon construction that handles multidimensional data.

Scalability:

Efficient network measurement on a larger-scale remains an issue — can our monitoring systems scale to a larger network topology with hundreds of nodes or even more? In UnivMon, we propose a simple network-wide solution by solving an ILP formula, but it might be hard to achieve satisfiable accuracy on a larger scale. We need to handle critical issues such as errors caused by multi-path, multi-counting, and rerouting. It is also complicated to give good accuracy guarantees on heterogeneous topologies. One possible direction to address this issue is to disseminate the sketches into critical nodes and improve the mergeability of UnivMon by the idea of mergeable data sketches [182].

Handling large general patterns (e.g., patterns with 6 nodes or more) on a large graph may still be hard. Due to the complexity of large patterns, the number of estimators that needed to achieve a good accuracy (say 5%) could be too large to fit into a small cluster. To improve the scalability in this case, we will be required to explore better sampling techniques.

Flow identities:

In programmable hardware switches, there is lacking an efficient way to store most frequent flow identities. This is because sketch data structures only preserve the counts of network flows not flow identities. In theory, we can utilize a priority queue alongside the sketch to store the identities of the most frequent flows, which gives the best memory efficiency. Unfortunately, it is infeasible for existing programmable ASICs to support accurate priority queue operations, and there is also nontrivial efforts needed to enable estimated operations on a priority queue with even a small number of entries. To avoid using priority queue, one potential technique we can use is putting additional reversible sketches that approximately reverse the hash functions [9].

Sensitivity to parameters:

The actual accuracy of sketching algorithms highly depends on the input parameters. For different workload patterns, the parameters in sketching algorithms define the number of independent hash trials and the probability of collisions. There (very likely) isn't an ideal parameter setting working for any workloads and it is also not straightforward for network administrators or users to pick appropriate parameters to obtain the best possible accuracy on the results.

6.3 Future Work

Can we further optimize the performance in hardware:

Network monitoring is just one of the modules running on a switch. Our work UnivMon is memory-efficient, but UnivMon's per-packet operation will occupy many processing stages on a programmable switch. The number of processing stages is limited and we want to leave enough room for other concurrent network functions. Thus, we will further explore the ways to improve the efficiency of UnivMon's stateful operations.

Sketching primitives are useful but can we make them more expressive:

We obtain traffic metrics from UnivMon, NitroSketch, or other sketching-based primitives. These metrics are important but may not be what users want to understand since they are not very expressive. The question here is that can we make the sketching-based measurement systems more expressive such that users can write SQL alike queries to the system. The users will not be required to understand the underlying metrics they need to obtain in order to understand the workload, but just an expressive query instead.

Measuring traffic metrics is nice but can we make a step further:

In network monitoring, traffic metrics such as heavy hitters, entropy, traffic change, etc., are important information collected from the workload. A natural question is that what will be the next step to utilize these metrics? We need more concrete use cases to demonstrate why these metrics are important. In this case, we will explore the usage of these statistics in the detection and diagnosis of DDoS attacks.

ASAP handles static graphs; Can we support large evolving graphs:

Evolving graphs pose significant challenges to the existing system design of ASAP. Since the core algorithm behind ASAP is an advanced sampling technique, a set of edge additions or deletions of will break the randomness of each estimator in ASAP, which breaks the proofs. Naively, to fix the theory, for each edge, we need to maintain the uniform randomness and update the state for each estimator. This simple fix will increase the per-edge computations to $O(r)$ for r estimators, which is infeasible for large graphs. We will work on designing a better update mechanism that improves the per-edge computations.

Bibliography

[1] E. Team, “insidebigdata guide to use of big data on an industrial scale,” *insideBIGDATA*, 2017.

[2] B. Claise, “Cisco systems netflow services export version 9,” vol. RFC 3954.

[3] M. Wang, B. Li, and Z. Li, in *Proc. of ICDCS*, 2004.

[4] G. Cormode and S. Muthukrishnan, “An Improved Data Stream Summary: The Count-min Sketch and Its Applications,” *J. Algorithms*, 2005.

[5] A. Metwally, D. Agrawal, and A. E. Abbadi, “Efficient computation of frequent and top-k elements in data streams,” in *Proc. of ICDT*, 2005.

[6] M. Charikar, K. Chen, and M. Farach-Colton, “Finding frequent items in data streams,” 2002.

[7] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, “Counting distinct elements in a data stream,” in *Proc. of RANDOM*, 2002.

[8] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, “Sketch-based change detection: Methods, evaluation, and applications,” in *Proc. of IMC*, 2003.

[9] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proc. of IMC*, 2004.

[10] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proc. of SIGMETRICS/Performance*, 2006.

[11] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. of NSDI*, 2013.

[12] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.

[13] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proc. of SOSP*, 2015.

[14] C. C. Aggarwal and H. Wang, Eds., *Managing and Mining Graph Data*, ser. Advances in Database Systems. Springer, 2010.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning." in *UAI*, P. Grünwald and P. Spirtes, Eds. AUAI Press, 2010, pp. 340–349. [Online]. Available: http://dblp.uni-trier.de/db/conf/uai/uai2010.html#LowGKBGH10

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, in *Proc. of OSDI*, Berkeley, CA, USA, 2012.

[17] A. Kyrola, G. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a pc,” in *Proc. of OSDI*, Hollywood, CA, 2012.

[18] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proc. of SOSP*, 2013.

[19] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, and I. Franklin, Stoica, “Graphx: Graph processing in a distributed dataflow framework,” in *Proc. of OSDI*, 2014.

[20] A. Quamar, A. Deshpande, and J. Lin, “Nscale: Neighborhood-centric large-scale graph analytics in the cloud,” 2016.

[21] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, “Chaos: Scale-out graph processing from secondary storage,” in *Proc. of SOSP*, 2015.

[22] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, “Asynchronous large-scale graph processing made easy.” in *Proc. of CIDR*, 2013.

[23] A. Buluç and J. R. Gilbert, “The combinatorial BLAS: design, implementation, and applications,” *IJHPCA*, 2011.

[24] R. Chen, J. Shi, Y. Chen, and H. Chen, “Powerlyra: Differentiated graph computation and partitioning on skewed graphs,” in *Proc. of EuroSys*, 2015.

[25] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, “Wonderland: A novel abstraction-based out-of-core graph processing system,” in *Proc. of ASPLOS*, 2018.

[26] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proc. of SIGCOMM*, 2016.

[27] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "ASAP: Fast, approximate graph pattern mining at scale," in *Proc. of OSDI*, 2018.

[28] Z. Liu, N. Ivkin, L. Yang, M. Neyrinck, G. Lemson, A. Szalay, V. Braverman, T. Budavari, R. Burns, and X. Wang, "Streaming algorithms for halo finders," in *Proc. of e-Science*, 2015.

[29] A. Knebe, F. R. Pearce, H. Lux, Y. Ascasibar, P. Behroozi, J. Casado, C. C. Moran, J. Diemand, and K. Dolag, "Structure finding in cosmological simulations: the state of affairs," *MNRAS*, vol. 435, pp. 1618–1658, Oct. 2013.

[30] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True, "Deriving traffic demands for operational ip networks: Methodology and experience," *IEEE/ACM TON*, 2001.

[31] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proc. of ACM CoNEXT*, 2011.

[32] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *Proc. of ACM CoNEXT*, 2013.

[33] Y. Xie, V. Sekar, D. A. Maltz, M. K. Reiter, and H. Zhang, "Worm origin identification using random moonwalks," in *Proc. of S&P*, 2005.

[34] A. Kumar, M. Sung, J. J. Xu, and J. Wang, “Data streaming algorithms for efficient and accurate estimation of flow size distribution,” in *Proc. of SIGMETRICS*, 2004.

[35] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi, “Fast data stream algorithms using associative memories,” in *Proc. of SIGMOD*, 2007.

[36] H. C. Zhao, A. Lall, M. Ogihara, O. Spatscheck, J. Wang, and J. Xu, “A data streaming algorithm for estimating entropies of od flows,” in *Proc. of IMC*, 2007.

[37] N. Duffield, C. Lund, and M. Thorup, “Estimating flow distributions from sampled flow statistics,” in *Proc. of SIGCOMM*, 2003.

[38] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, “Fast monitoring of traffic subpopulations,” in *Proc. of IMC*, 2008.

[39] C. Estan and G. Varghese, “New directions in traffic measurement and accounting,” in *Proc. of SIGCOMM*, 2002.

[40] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, “Sketch-based change detection: methods, evaluation, and applications,” 2003.

[41] V. Braverman and R. Ostrovsky, “Approximating large frequency moments with pick-and-drop sampling,” in *Proc. of APPROX/ROMDOM*, 2013.

[42] V. Braverman, J. Katzman, C. Seidell, and G. Vorsanger, “An optimal algorithm for large frequency moments using o(nˆ(1-2/k)) bits,” in *Proc. of APPROX/RANDOM*, 2014.

[43] V. Braverman, Z. Liu, T. Singh, N. V. Vinodchandran, and L. F. Yang, "New bounds for the CLIQUE-GAP problem using graph decomposition theory," in *Proc. of MFCS*, 2015.

[44] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc.*, ser. NSDI, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482631

[45] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: Sketch Resource Allocation for Software-defined Measurement," in *Proc., CoNEXT*, 2015.

[46] P. Indyk, A. McGregor, I. Newman, and K. Onak, "Open problems in data streams, property testing, and related topics," 2011.

[47] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the case for a minimalist approach for network flow monitoring," in *Proc. of IMC*, 2010.

[48] H. C. Zhao, A. Lall, M. Ogihara, and J. J. Xu, "Global iceberg detection over distributed data streams," in *Proc. of ICDE*, 2010.

[49] V. Braverman and R. Ostrovsky, "Zero-one frequency laws," in *Proc. of STOC*, 2010.

[50] ——, "Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams," in *Proc. of APPROX/RAMDOM*, 2013.

[51] V. Braverman, S. R. Chestnut, D. P. Woodruff, and L. F. Yang, “Streaming space complexity of nearly all functions of one variable on frequency vectors,” in *Proc. of PODS*, 2016.

[52] V. Braverman, R. Ostrovsky, and A. Roytman, “Zero-one laws for sliding windows and universal sketches,” in *Proc. of APPROX/RANDOM*, 2015.

[53] V. Braverman, S. R. Chestnut, R. Krauthgamer, and L. F. Yang, “Streaming symmetric norms via measure concentration,” *CoRR*, 2015.

[54] Z. Liu, G. Vorsanger, V. Braverman, and V. Sekar, “Enabling a "risc" approach for software-defined monitoring using universal streaming,” in *Proc. of HotNets*, 2015.

[55] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, 2014.

[56] N. Kang, Z. Liu, J. Rexford, and D. Walker, “Optimizing the "one big switch" abstraction in software-defined networks,” in *Proc. of CoNEXT*, 2013.

[57] “Caida internet traces 2014 sanjose,” http://goo.gl/uP5aqG.

[58] “The caida ucsd anonymized internet traces 2015,” http://www.caida.org/data/passive/passive_2015_dataset.xml.

[59] N. Alon, Y. Matias, and M. Szegedy, “The space complexity of approximating the frequency moments,” in *Proc.*, ser. STOC, 1996. [Online]. Available: http://doi.acm.org/10.1145/237814.237823

[60] M. Datar, A. Gionis, P. Indyk, and R. Motwani, “Maintaining stream statistics over sliding windows,” *SIAM J. Comput.*, 2002.

[61] G. Cormode and S. Muthukrishnan, “An improved data stream summary: The count-min sketch and its applications,” *J. Algorithms*, 2005.

[62] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, “Extending networking into the virtualization layer,” in *Proc. of HotNets*, 2009.

[63] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn,” in *Proc. of ACM SIGCOMM*, 2013.

[64] “Intel flexpipe,” http://goo.gl/H5qPP2.

[65] “P4 specification,” http://goo.gl/5ttjpA.

[66] S. Dasgupta and A. Gupta, “An elementary proof of a theorem of johnson and lindenstrauss,” *Random Struct. Algorithms*, 2003.

[67] V. Braverman and S. R. Chestnut, “Universal Sketches for the Frequency Negative Moments and Other Decreasing Streaming Sums,” in *Proc. of APPROX/RANDOM*, 2015.

[68] A. Chakrabarti, S. Khot, and X. Sun, "Near-optimal lower bounds on the multi-party communication complexity of set disjointness." in *Proc. of IEEE CCC*, 2003.

[69] "Why big data needs big buffer switches," https://goo.gl/ejWUIq.

[70] "Netfpga technical specifications," http://netfpga.org/1G_specs.html.

[71] "P4 behavioral simulator," https://github.com/p4lang/p4factory.

[72] "Opensketch simulation library," https://goo.gl/kyQ80q.

[73] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, P. Konsor, A. Semin, M. Kanaly, R. Brazones, and R. Shah, "Intel performance counter monitor - a better way to measure cpu utilization," http://goo.gl/tQ5gxa.

[74] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *Selected Areas in Communications, IEEE Journal on*, 2011.

[75] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, 2008.

[76] L. Yuan, C.-N. Chuah, and P. Mohapatra, "Progme: towards programmable network measurement," *IEEE/ACM TON*, 2011.

[77] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese,

"CONGA: Distributed Congestion-aware Load Balancing for Datacenters," in *Proc. of SIGCOMM*, 2014.

[78] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-optimal Datacenter Transport," in *Proc. of SIGCOMM*, 2013.

[79] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine Grained Traffic Engineering for Data Centers," in *Proc. of ACM CoNEXT*, 2011.

[80] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling Flow Management for High-performance Networks," in *Proc. of SIGCOMM*, 2011.

[81] P. Garcia-Teodoro, J. E. Diaz-Verdejo, G. Macia-Fernandez, and E. Vazquez, "Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges," *Computers and Security*, 2009.

[82] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar, "AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi-tenanted Data Centers," in *Proc. of IEEE HOTI*, 2010.

[83] L. Ying, R. Srikant, and X. Kang, "The Power of Slightly More than One Sample in Randomized Load Balancing," in *Proc. of IEEE INFOCOM*, 2015.

[84] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," in *Proc. of NSDI*, 2015.

[85] "Microsoft hyper-v virtual switch," https://technet.microsoft.com/en-us/library/hh831823.aspx.

[86] "Microsoft hyper-v virtual switch," https://www.cisco.com/c/en/us/products/switches/nexus-1000v-switch-vmware-vsphere/index.html.

[87] "Fd.io vector packet processing," https://fd.io/technology/.

[88] J. Misra and D. Gries, "Finding repeated elements," Tech. Rep., 1982.

[89] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. of ACM SOSR*, 2017.

[90] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," *ACM SIGCOMM and CoRR/1707.06778*, 2017.

[91] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proc. of STOC*, 1996.

[92] O. Alipourfard, M. Moshref, and M. Yu, "Re-evaluating measurement algorithms in software," in *Proc. of HotNets*, 2015.

[93] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu, "A comparison of performance and accuracy of measurement algorithms in software," in *Proc. of SOSR*, 2018.

[94] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proc. of SIGCOMM*, 2017.

[95] "The caida ucsd anonymized internet traces 2016 - equinix-chicago," http://www.caida.org/data/passive/passive_2016_dataset.xml.

[96] "Capture traces from mid-atlantic ccdc 2012," http://www.netresec.com/?page=MACCDC.

[97] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. of IMC*, 2010.

[98] N. Hua, B. Lin, J. J. Xu, and H. C. Zhao, "Brick: A novel exact active statistics counter architecture," in *Proc. of ACM/IEEE ANCS*, 2008.

[99] L. Yang, W. Hao, P. Tian, D. Huichen, L. Jianyuan, and L. Bin, "Case: Cache-assisted stretchable estimator for high speed per-flow measurement," in *Proc. of IEEE INFOCOM*, 2016.

[100] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," in *ACM SIGMETRICS Performance Evaluation Review*, 2008.

[101] "Data plane developer kit (dpdk)," https://software.intel.com/en-us/networking/dpdk.

[102] "Intel ethernet controller 710 series datasheet," https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf.

[103] "Intel vtune amplifier," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[104] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proc. of IMC*, 2015.

[105] J. Matouek and J. Vondrk, "The Probabilistic Method - Lecture Notess," https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f09/www/handouts/matousek-vondrak-prob-ln.pdf.

[106] W. Feller, "Generalization of a probability limit theorem of cramér," *Transactions of the American Mathematical Society*, 1943.

[107] E. V. Slud, "Distribution inequalities for the binomial law," *The Annals of Probability*, 1977.

[108] R. D. Gordon, "Values of mills' ratio of area to bounding ordinate and of the normal probability integral for large values of the argument," *The Annals of Mathematical Statistics*, 1941.

[109] "xxhash library," http://www.xxhash.com/.

[110] "Intel advanced vector extensions," https://software.intel.com/en-us/isa-extensions/intel-avx.

[111] "Fast concurrent queue," https://github.com/cameron314/readerwriterqueue.

[112] "Tcpdump and libpcap," http://www.tcpdump.org.

[113] “Wireshark,” https://www.wireshark.org.

[114] “Graph DBMS increased their popularity by 500% within the last 2 years,” http://db-engines.com/en/blog_post//43.

[115] “Enterprise DBMS, Q1 2014,” https://www.forrester.com/report/TechRadar+Enterprise+DBMS+Q1+2014/-/E-RES106801.

[116] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Technical Report, 1999.

[117] S. Fortunato, “Community detection in graphs,” *Physics reports*, 2010.

[118] X. Zhu and Z. Ghahramani, “Learning from labeled and unlabeled data with label propagation,” *Technical Report CMU-CALD-02-107*, 2002.

[119] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, “Network motifs: simple building blocks of complex networks,” *Science*, 2002.

[120] X. Yan and J. Han, “gspan: Graph-based substructure pattern mining,” in *Proc. of ICDM*, 2002.

[121] C. Bron and J. Kerbosch, “Algorithm 457: finding all cliques of an undirected graph,” *Communications of the ACM*, 1973.

[122] N. Pržulj, D. G. Corneil, and I. Jurisica, “Modeling interactome: Scale-free or geometric?” *Bioinformatics*, 2004.

[123] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, “Kineograph: Taking the pulse of a fast-changing and connected world,” in *Proc. of EuroSys*, 2012.

[124] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, “Chronos: A graph engine for temporal graph analysis,” in *Proc. of EuroSys*, 2014.

[125] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” in *Proc. of SOSP*, 2013.

[126] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, “Llama: Efficient graph analytics using large multiversioned arrays,” in *Proc. of ICDE*, 2015.

[127] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” 2015.

[128] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, “Gram: Scaling graph computation to the trillions,” in *Proc. of SoCC*, 2015.

[129] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis, “Grami: Frequent subgraph and pattern mining in a single large graph,” *Proc. of VLDB Endow.*, 2014.

[130] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, “Blinkdb: Queries with bounded errors and bounded response times on very large data,” in *Proc. of EuroSys*, 2013.

[131] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *Proc. of ASPLOS*, 2015.

[132] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics." in *Proc. of NSDI*, 2014.

[133] S. Chaudhuri, G. Das, and V. Narasayya, "Optimized stratified sampling for approximate query processing," *ACM Trans. Database Syst.*, 2007.

[134] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online aggregation and continuous query support in mapreduce," in *Proc. of SIGMOD*, 2010.

[135] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, 2012.

[136] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. of WWW*, 2010.

[137] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of WWW*, Manhattan, USA, 2004.

[138] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *CoRR*, 2011.

[139] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: Counting triangles in massive graphs with a coin," in *Proc. of KDD*, 2009.

[140] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *Proc. of PODS*, 2006.

[141] M. Al Hasan and M. J. Zaki, "Output space sampling for graph patterns," *Proc. VLDB Endow.*, 2009.

[142] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proc. VLDB Endow.*, 2013.

[143] N. K. Ahmed, N. Duffield, J. Neville, and R. Kompella, "Graph sample and hold: A framework for big-graph analytics," in *Proc. of KDD*, 2014.

[144] M. Jha, C. Seshadhri, and A. Pinar, "A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox," *ACM Trans. Knowl. Discov. Data*, 2015.

[145] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proc. of NSDI*, 2012.

[146] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. of HotCloud*, 2010.

[147] P. Ribeiro and F. Silva, "G-tries: A data structure for storing and finding subgraphs," *Data Min. Knowl. Discov.*, 2014.

[148] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. of NSDI*, 2016.

[149] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. of NSDI*, 2017.

[150] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: Building fast and reliable approximate query processing systems," in *Proc. of SIGMOD*, 2014.

[151] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[152] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, 2012.

[153] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. of WWW*, 2011.

[154] "Graph data mining with arabesque," http://arabesque.io.

[155] Apache Giraph, "http://giraph.apache.org."

[156] A. Das Sarma, S. Gollapudi, and R. Panigrahy, "Estimating pagerank on graph streams," in *Proc. of PODS*, 2008.

[157] N. Z. Gong, W. Xu, L. Huang, P. Mittal, E. Stefanov, V. Sekar, and D. Song, "Evolution of social-attribute networks: Measurements, modeling, and implications using google+," in *Proc. of IMC*, 2012.

[158] K. J. Ahn, S. Guha, and A. McGregor, "Analyzing graph structure via linear measurements," in *Proc. of SODA*, 2012.

[159] ——, "Graph sketches: Sparsification, spanners, and subgraphs," in *Proc. of PODS*, 2012.

[160] V. Braverman, R. Ostrovsky, and D. Vilenchik, "How hard is counting triangles in the streaming model?" in *Proc. of ICALP*, 2013.

[161] S. Assadi, S. Khanna, and Y. Li, "On estimating maximum matching size in graph streams," in *Proc. of SODA*, 2017.

[162] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White, "The evolution of large-scale structure in a universe dominated by cold dark matter," *ApJ*, 1985.

[163] B. L. Falck, M. C. Neyrinck, and A. S. Szalay, "ORIGAMI: Delineating Halos Using Phase-space Folds," *ApJ*, 2012.

[164] A. Knebe, S. R. Knollmann, S. I. Muldrew, F. R. Pearce, M. A. Aragon-Calvo, Y. Ascasibar, P. S. Behroozi, D. Ceverino, S. Colombi, J. Diemand, and K. Dolag, "Haloes gone MAD: The Halo-Finder Comparison Project," *MNRAS*, 2011.

[165] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proc. of IMC*, 2004.

[166] J. Beringer and E. Hüllermeier, "Efficient instance-based learning on data streams," *Intell. Data Anal.*, 2007.

[167] E. Liberty, "Simple and deterministic matrix sketching," in *Proc. of SIGKDD*, 2013.

[168] M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos, "Continuous trend-based clustering in data streams," in *Proc. of DaWaK*, 2008.

[169] L. Serir, E. Ramasso, and N. Zerhouni, "Evidential evolving gustafson-kessel algorithm for online data streams partitioning using belief function theory." *Int. J. Approx. Reasoning*, 2012.

[170] B. Ball, M. Flood, H. Jagadish, J. Langsam, L. Raschid, and P. Wiriyathammabhum, "A flexible and extensible contract aggregation framework (caf) for financial data stream analytics," in *Proc. of DSMM*, 2014.

[171] F. Rusu and A. Dobra, "Statistical analysis of sketch estimators," in *Proc. of SIGMOD*, 2007.

[172] J. Spiegel and N. Polyzotis, "Graph-based synopses for relational selectivity estimation," in *Proc. of SIGMOD*, 2006.

[173] P. Indyk and D. Woodruff, "Optimal approximations of the frequency moments of data streams," in *Proc. of STOC*, 2005.

[174] P. Coles and B. Jones, "A lognormal model for the cosmological mass distribution," *MNRAS*, 1991.

[175] I. Kayo, A. Taruya, and Y. Suto, "Probability distribution function of cosmological density fluctuations from a gaussian initial condition: Comparison of one-point and two-point lognormal model predictions with n-body simulations," *The Astrophysical Journal*, 2001.

[176] R. E. Smith, J. A. Peacock, A. Jenkins, S. D. M. White, C. S. Frenk, F. R. Pearce, P. A. Thomas, G. Efstathiou, and H. M. P. Couchman, “Stable clustering, the halo model and non-linear cosmological power spectra,” *MNRAS*, 2003.

[177] S. Gottlöber and G. Yepes, “Shape, Spin, and Baryon Fraction of Clusters in the MareNostrum Universe,” *ApJ*, 2007.

[178] insideBIGDATA, “Ahf: Amiga’s halo finder,” *The Astrophysical Journal Supplement Series*, 2009.

[179] S. Planelles and V. Quilis, “Asohf: a new adaptive spherical overdensity halo finder,” *Astronomy & Astrophysics*, 2010.

[180] A. Klypin and J. Holtzman, “Particle-Mesh code for cosmological simulations,” *ArXiv Astrophysics e-prints*, 1997.

[181] M. C. Neyrinck, N. Y. Gnedin, and A. J. S. Hamilton, “VOBOZ: an almost-parameter-free halo-finding algorithm,” *MNRAS*, 2005.

[182] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi, “Mergeable summaries,” in *Proc. of PODS*, 2012.

Vita

Zaoxing (Alan) Liu was born in 1989 in China. Zaoxing did his undergraduate work at Southeast University, Nanjing, China, where he majored in Computer Science and Physics (2007-2011). In 2012, Zaoxing enrolled in the Master's program in Computer Science at the Johns Hopkins University and earned a M.S.E degree in 2013. In 2014, Zaoxing began his Computer Science PhD program under the advice of Prof. Vladimir Braverman, where his work focused on Networked Systems and Algorithmic Design for Systems.