Scalable, Network-Wide Telemetry with Programmable Switches

Walter Robert J. Harrison

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Professor Jennifer Rexford

June 2019

Abstract

Managing modern networks requires collecting and analyzing network traffic from distributed switches in real time, i.e., performing network-wide telemetry. Telemetry systems must be flexible and fine-grained to support myriad queries about the security, performance, and reliability of networks. Yet, they must also scale as the number of queries, link speeds, and the size of the networks increase. Realizing these goals requires balancing the division of labor between high-speed, but resource constrained, network switches and general-purpose CPUs to support flexible telemetry at scale.

First, we present Sonata, a flexible and scalable network telemetry system that uses the compute resources of both stream-processing servers and a single Protocol Independent Switch Architecture (PISA) switch. PISA switches offer both high-speed processing and limited programmability. We show how to execute Sonata's high-level queries at line rate by first compiling them to PISA primitives. Next, we model the resource constraints of PISA switches to solve an optimization problem that minimizes the load on the stream processor by executing portions of queries directly in the switch. Sonata can support a wide range of monitoring queries and reduces the stream processor's workload by orders of magnitude over existing telemetry systems.

Second, we present Herd, a system for implementing a subset of Sonata queries distributed across several switches. Herd determines network-wide heavy hitters, i.e., flows that consist of many more packets than most others, by counting flows at the switches, without maintaining per-flow state, and probabilistically reporting to a central coordinator. Based on these reports, the coordinator adapts parameters at each switch based on the spatial locality of the flows. Simulations using packet traces show that our prototype can detect network-wide heavy hitters accurately with 17% savings in communication overhead and 38% savings in switch state compared to existing approaches. We then present an algorithm to tune system parameters in order to maximize detection accuracy under switch memory and bandwidth constraints.

Together, Sonata and Herd provide network operators the ability to execute a set of network-wide telemetry queries from a single interface that combines the strengths of both programmable data planes and general-purpose CPUs.

Acknowledgments

I will be forever grateful to my adviser, Jennifer Rexford, for guiding me throughout this process. Jen is the best adviser a graduate student could hope for. Her insight, mentorship, and encouragement were invaluable on this journey, and I have learned so much from her. Anyone who has ever worked with Jen knows what a joy it is, and I am forever indebted to her for affording me that privilege.

I want to thank the members of my committee – Nick Feamster, Nate Foster, David Walker, and Kyle Jamieson. Nick was a great collaborator who always pushed me to sharpen my work, consistently raising its quality. Dave and Nate have been incredible collaborators and mentors over the past nine years – first when I was a student in the Master's program and then again in the Ph.D. program. One reason that I am here today is because of the enjoyable experience I had collaborating with them and Jen on my first research project.

Research is always a group endeavor, and I have had the good fortune of working with some great people. In particular, I must thank Arpit Gupta for all of his support throughout our collaborations. Working with Arpit was a very rewarding experience for which I will be forever grateful. Mina Tahmasbi Arashloo has also been a fantastic colleague over the past several years. Mina's sharp insights have consistently improved my work and, on many occasions, helped us to get over the goal line. I also appreciate working with Shir Landau Feibish who brought a new perspective to our work. Shir was a dedicated collaborator whom I relied on and the only person in the lab who could relate to me on the adventures of parenthood.

The entire Cabernet research group, past and present, is an incredible group of people who support each other day in and day out, deadline after deadline. I cannot imagine a better group of people to work with, and I am truly grateful for all of their support. The Cabernet group supports each other in life as in work – I must thank Robert MacDavid and

Theano Stavrinos for ensuring that we all had an opportunity for respite during our regular happy-hour festivities.

I must thank the US Army and the Department of Electrical Engineering and Computer Science at the United States Military Academy for providing me the opportunity to pursue this Ph.D. Were it not for David Raymond, who encouraged me to major in Computer Science as an undergraduate, and Greg Conti, who encouraged me to return to West Point as faculty, I very likely would not be here today. Thank you for having the prescience to know what I wanted to do, even before I knew that I wanted to do it.

Most importantly, I am blessed to have an incredible family who supported me throughout this arduous process. Earning a Ph.D. is no small feat; earning one with five children aged ten and under falls just short of a miracle. I could never have completed this journey without the love and support of my wife, Kate. I am forever indebted to her for the sacrifices that she and our children made so that I could complete this work. To my sons, I hope that this inspires and encourages you to work hard and pursue your dreams, whatever they may be. Finally to my parents, thank you for instilling in me the work ethic and grit required to get here.

This dissertation was supported by U.S. Army Advanced Civil Schooling, the National Science Foundation (NSF) Grant CNS-1704077, and the Defense Advanced Research Projects Agency (DARPA) Contract HR0011-17-C-0047.

To my wife, Kate, and our five sons, RJ, Connor, Patrick, Sean, and Quinn.

Contents

List of Tables

List of Figures

Bibliographic Notes

This dissertation includes work presented in the following peer-reviewed publications and public talks with the listed co-authors.

Peer-Reviewed Publications

An early version of some material presented in Chapter 2 appeared in an ACM HotNets paper (2016) by Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, and Chris MacStoker [27]; and an Arxiv paper (2017) co-authored with Arpit Gupta, Rüdiger Birkner, Ankita Pawar, Marco Canini, Nick Feamster, Jennifer Rexford [28] and Walter Willinger. However, much of the material in Chapter 2 appears in an ACM SIGCOMM paper (2018) co-authored with Arpit Gupta, Marco Canini, Nick Feamster, Jennifer Rexford and Walter Willinger. Material presented in Chapter 3 was joint work completed with Shir Landau Feibish, Arpit Gupta, Ross Teixeira, Shan Muthukrishnan, and Jennifer Rexford.

Talks

1. *Sonata: Scalable Streaming Analytics for Network Telemetry*. 4th P4 Workshop, May 2017. North American Network Operators' Group (NANOG) 70, June 2017.
2. *Sonata: Query-Driven Streaming Network Telemetry*. ACM SIGCOMM, August 2018.

Chapter 1

Introduction

Every day, millions of devices (e.g., mobile phones and smart devices) connect to the Internet through edge networks to access myriad services that are hosted in far-away data centers. Dozens of intermediate transit networks forward terabytes of packets to connect these endpoints by providing best-effort delivery. While edge, data center, and transit networks each have slightly different goals, the operators of each network must answer a common and fundamental question in order to maintain them: *What is going on in my network?* Answering this important question is the purpose of network telemetry.

1.1 Network Telemetry

Although network operators have been trying to build network telemetry systems for several years [87], the primary standards-making body for the Internet has yet to agree on a definition for what network telemetry is [70]. At its core, a network telemetry system performs collection and analysis to report the state of the network in a timely and fine-grained fashion [82]. These systems report the state of the network in response to custom queries, which could, "require...data with arbitrary source, granularity, and precision which are beyond the capabilit[ies] of...existing techniques" [70]. To better understand the range of queries that a telemetry system must support, let us consider a few use cases that highlight

the diversity of tasks that network telemetry systems should support in edge, transit, and data center networks.

Detect Compromises. Edge networks are where users and their devices access the Internet. The user-facing nature of these networks makes them a common target for miscreants seeking to expand their botnets or spread malware. Various traffic attributes, e.g., transport layer ports, or domain names, observed in the network can help identify compromised devices for remediation. However, rapidly evolving attack vectors and tools requires continuously updating these signatures and monitoring accordingly [61].

Troubleshoot Service Failures. Data center networks connect the front ends of services that receive and balance the deluge of inbound requests to an enormous amount of compute capacity on the back end. Performing a root-cause analysis of service failures in these complex distributed systems can require fine-grained detail about a single packet's experience through every interface on each device in the network [6].

Detect Changes in Traffic Patterns. Intermediate transit networks must provision and configure their networks to handle peak data rates between source and destination network pairs. When traffic patterns shift, operators must first detect the shift, attribute the shift to a root cause (e.g., attack, misconfiguration, or failure), and then react to them. Some traffic shifts occur predictably in diurnal patterns, while others, such as microbursts, occur on very short timescales.

While each network's objective in these use cases is slightly different, we can distill from these use cases some properties that a network telemetry system should have. Network telemetry systems should be: (1) *flexible*, to support a wide range of queries based on arbitrary traffic attributes, granularities, and timescales; (2) *scalable*, to answer the queries within available resource constraints and as traffic volumes grow large; and (3) *network-wide* to support queries both from the vantage point of a single device in the network and

the combined view from multiple devices. In Sections 1.3 and 1.4, we will describe some of the challenges in realizing these latter two properties. In the next section, we first describe a language for expressing telemetry queries in a flexible and extensible way.

1.2 Expressing Telemetry Queries

Declarative query interfaces to complex systems have a long history in the database community [13]. This type of interface gives system users the ability to express, within the constraints of the language, all of the system's capabilities without worrying about how the system efficiently executes the query. Declarative query interfaces have also shielded network programmers from the low-level complexities of programming switch flow tables [25]. Similarly, recent work [27] demonstrated that a declarative query interface can express a wide-range of network telemetry queries using a familiar, dataflow-like query language where packets are represented as abstract tuples. This interface gives operators the flexibility to express a wide-range of telemetry queries as a sequence of transformations on the entire packet stream observed by the network.

Packets as Tuples. Information contained in packet headers are naturally represented as key-value pairs (e.g., source and destination IP address, etc.) where the name of the header field constitutes the key along with its associated value in the packet. This structure also naturally lends itself to representing the entire collection of packet attributes, including the payload, as a tuple consisting of each key-value pair. Telemetry queries, however, might also want details about a packet's experience in the network, such as, the queue length it experienced or on which port it arrived. Such metadata can also be represented as key-value pairs in a tuple. Extending the tuple to include new features requires no change to the abstraction or to the semantics of the queries; it merely requires extending the set of key names available in the tuple. With the packet represented as a tuple, we can also relax the traditional definition of a flow to be any group of packets that share a common key-value

Operator	Description
filter(p)	Filter packets that satisfy predicate p.
map(f)	Transform each tuple with function f.
distinct()	Emit tuples with unique combinations of fields.
reduce(k, f)	Emit result of function f applied on key k over the input stream.
join(k, q)	Join the output of query q on key field k

Table 1.1: Dataflow Operators. Stateful operators' semantics are with respect to a notion of time, or window interval.

attribute, e.g., an exact five tuple or simply a common source-destination pair. This relaxed definition better aligns the term with the diverse types of queries intended to be supported by network telemetry.

Dataflow Operators. Many network-monitoring queries require computing aggregate statistics over a subset of traffic and joining the results from other queries. These queries can be expressed as a sequential composition of dataflow operators (e.g., filter, map, reduce). Many monitoring systems [20, 10, 53] have found this model a natural and familiar way to express queries. Table 1.1 summarizes some commonly-used dataflow operators. Stateful dataflow operators are all executed with respect to a query-defined time interval, or window. For example, applying a `reduce` operator with the `sum` function will return the value at the end of each window.

Example: DNS Reflection Attacks To illustrate the expressiveness of this declarative interface, let us consider a query to detect a common network attack [61]: DNS reflection. In this attack, miscreants send a large number of requests to open DNS resolvers with a false source address, i.e., the victim's address. Fooled into thinking that the victim sent these requests, the DNS resolvers all respond and overwhelm the victim. Figure 1.1 shows how we would express a query to detect the victim of a DNS reflection attack. In line 1, we start with the keyword `pktStream` to indicate that all operators following it are applying transformations on the entire packet stream observed by the network. In line 2, we apply a `filter` operator to select only DNS traffic. In lines 3-4, we transform the stream of

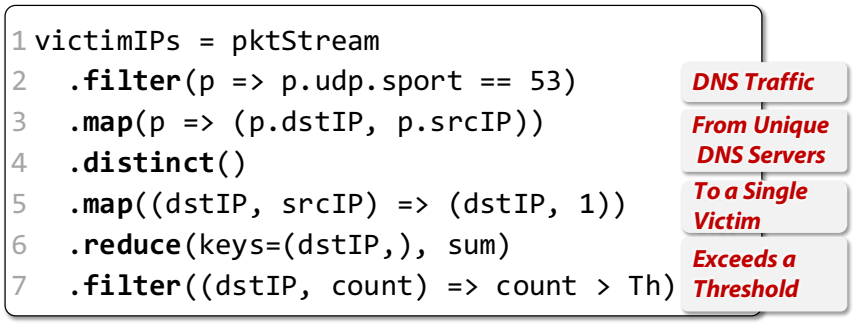


Figure 1.1: DNS Reflection Query.

packets into a stream of unique destination-source pairs. In line 5, we further transform the unique destination-source pairs as just the destination and the number 1. In lines 6-7, we count those pairs to determine which destinations have been sent traffic from a number of unique DNS servers that exceeds some threshold `Th`.

1.3 Scaling Query Execution

The declarative interface does indeed provide the flexibility desired in a network telemetry system and it frees the query author from reasoning about how to efficiently execute a given query. However, the telemetry system must still be able to scale query execution – a tall order for a system that provides the abstraction of querying over the entire packet stream. A strawman execution plan might send all packets observed in the network to a central point for collection and analysis. However, such a strategy would be both inefficient, because *all* traffic is not needed, but also ineffective, because it would be computationally infeasible to process as the volume of traffic grows large. Determining an execution plan that does not exhaust available resources and scales as traffic volumes grow large is a non-trivial task.

1.3.1 Challenges

Three current trends create a scalability challenge for network telemetry systems in edge, data center, and transit networks alike.

Increasing Number of Endpoints. At the edge, the number of Internet-connected devices had already reached 27 Billion devices by 2017 and that number continues to grow today [61]. These devices increase the volume of traffic transiting our networks without a human-in-the-loop when operating as expected; worse yet, these devices can also generate large volumes of traffic when co-opted as part of a botnet. The increasing number of endpoints does not just create additional traffic volume in edge networks. These additional endpoints connect through transit networks to remote services offered in data center networks, necessarily increasing the traffic volume in all three types of networks. When coupled with the additional addressing space available in IPv6, the increasing number of endpoints also creates more unique identifiers and active flows to monitor. Altogether, these forces make scaling a network telemetry system more challenging.

Increasing Link Speeds. In data centers and transit networks, link speeds continue to increase. Link speeds of 10 Gbps are now commodity and 100 Gbps links are becoming more common. For switches to maintain a throughput commensurate with these line rates, they will have to process packets very quickly. For example, a switch has about 12 ns to process 64 Byte packets at 40 Gbps.

Increasing Number of Queries. The current goal for network telemetry as stated by the Internet Engineering Task Force (IETF) should enable, "a smooth evolution toward intent-driven autonomous networks" [70]. In this vision, the telemetry system itself presents a view of the network for automated decision-making and control. Realizing this vision will require simultaneously executing dozens, if not hundreds, of queries to enable machines to perform the myriad management tasks that humans now do. With each additional query presented to the system, more resources will be required to support them.

1.3.2 Resources Needed for Execution

Executing arbitrary queries with fine granularity and in real time means inspecting millions of packets per second for traffic of interest. After identifying traffic of interest, the system must perform query-specific computation or aggregation. This process requires two key resources from devices in the network: computation and memory. In reality, networks consist of dozens of heterogeneous devices (e.g., routers, switches, firewalls, etc.), each with varying capabilities which we discuss in Section 1.3.3. If a query requires some computation that cannot be performed on a device where traffic of interest is observed, the telemetry system must now divert the original packet or generate a copy of it for another device with the capabilities demanded by the query [87]. Regardless of whether the packet is diverted or cloned, a third key resource is consumed to support query execution: network bandwidth. In scaling query execution, the telemetry system must ensure that it does not exhaust these available resources.

1.3.3 Diversity of Network Device Resources

As discussed, network devices have heterogeneous capabilities in terms of computation and memory. However, we can classify network devices into three general classes based on their computational capabilities and memory capacity. In this case, we will define computational capability in terms of what parts of the packet the device can inspect, or match, and what actions the devices can take on a packet. Table 1.2 summarizes the capabilities for traditional fixed-function switches, general-purpose CPUs, and emerging programmable switches [12, 11].

Fixed-Function Switches. Fixed-function switches are based on ASICs that can match on fixed, well-known packet headers. These devices can provide some coarse-grained statistics at coarse timescales, e.g., link utilization every five minutes, or sampled and aggregate statistics, e.g., IPFiX [16] or sFlow [56], but they fall short of the fine-grained,

	Fixed-function Switches	Programmable Switches	End-host CPUs
Match	Fixed headers	Arbitrary headers	All headers and payload fields
Actions	forward, drop, modify	forward, drop, add, subtract, bitwise operations	Arbitrary
Memory	MB	MB	GB
Speed	$O(ns)$	$O(ns)$	$O(\mu s)$

Table 1.2: Network Device Resource Diversity. Fixed-function switches can process network traffic the most scalably with the least computational capability, whereas CPUs can perform arbitrary computations and have abundant memory. Programmable switches can process traffic as scalably as fixed-function switches but with richer computational capacity.

real-time queries we want to support. However, these devices do process traffic at line rate, allowing them to handle large traffic volumes.

End-Host CPUs. General-purpose CPUs can perform arbitrary computation over the entire packet and payload, but they cannot process traffic at line rate. To scale query execution, a network telemetry system will need to limit the amount of traffic that must be processed by end-host CPUs. Some end hosts may have additional computational resources (e.g.,, Graphical Processing Units (GPU), or Field Programmable Gate Arrays (FPGA)), but we focus for now on CPUs because they are capable of the most general-purpose computation.

Programmable Switches. Emerging programmable switches combine the strengths of both fixed-function switches and end-host CPUs. These devices have a programmable parser that can extract arbitrary and user-defined header fields. These devices can also support custom packet processing pipelines that match on parsed fields and perform limited arithmetic, logical, and bitwise operations. They also contain registers for computing values across successive packets, such as flowlet timeouts [40]. These devices offer the same scalability properties of fixed-function devices but with additional computing capacity.

1.3.4 Partitioning to Scale Execution

In order to scale to a large number of queries and high traffic volume, a network telemetry system should *partition* the high-level queries across all of the diverse network devices. Rather than building a solution based solely on a single network device, a network telemetry system should combine the strengths of all the device types. We can use fixed-function switches for simple operations that must be applied to large volumes of traffic, e.g., `map` and `filter`; use programmable switches for more complex, but still high-volume operations, e.g., `reduce`; and use general purpose CPUs for the most complex, but lower-volume operations, e.g., `join`. By combining the strengths and resources of all available devices, we can efficiently execute the high-level queries in a scalable way.

1.4 Network-Wide, Scalable Telemetry

Partitioning high-level queries across a diverse set of network devices will help us achieve the desired scalability of a network telemetry system, but it does not address the challenges of answering queries from a network-wide perspective. A network-wide perspective could mean a single packet's experience traversing all of the links in a network [32, 30]. We will focus on network-wide measurements that can be calculated by a collection of edge switches abstracted as one big switch [41]. For example, the aggregate bytes or packet counts transmitted to a source or from a destination could be a network-wide metric of interest.

Coordinate for Network-Wide View. In order to calculate a network-wide metric of interest, the distributed collection of switches must *coordinate* with a central entity to unify the disparate perspectives of each individual device. Consider the scenario depicted in Figure 1.2. Each switch maintains a count for a flow of interest. However, no single switch has the accurate network-wide count for the flow. To unify the disparate views of the flow's

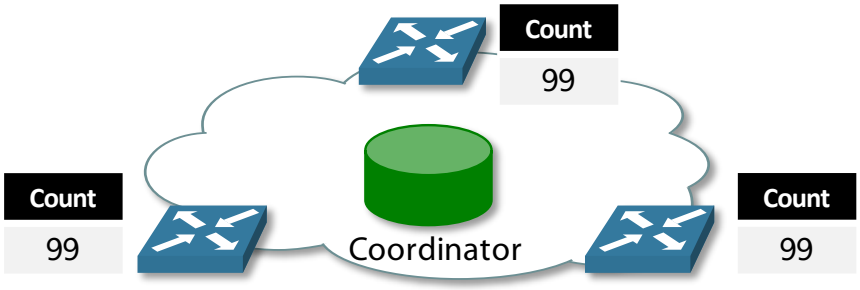


Figure 1.2: Coordinate for Network-Wide View. Here a collection of ingress switches each count a particular flow of interest. In order to get the network-wide count for this flow, the switches must unify their disparate views with a central coordinator.

count, each switch must tell a central coordinator the value of their individual counter to determine that the network-wide count is actually 297. To correctly calculate network-wide statistics of interest, computation, memory, and bandwidth will again be required.

Distributed Threshold Detection. In the previous example, only three messages need be exchanged to determine the network-wide count from the distributed counts. In practice, we are often interested in determining when a network-wide count exceeds a threshold. For example, if in Figure 1.2 we were interested in determining when the network-wide count for a flow of interest exceeds some threshold in real time, we could instead report every instance of the observed flow to the coordinator. This solution would require no memory for counters on the distributed switches and would be very accurate, but at an unscalable bandwidth cost. We could instead report each occurrence of the flows observed at the switches with some probability, but this sampling technique can yield inaccurate results on short time scales. To calculate network-wide statistics accurately and scalably, we need an efficient *coordination protocol* that consumes as little network bandwidth as possible and does not grow in proportion to the number of switches in the network.

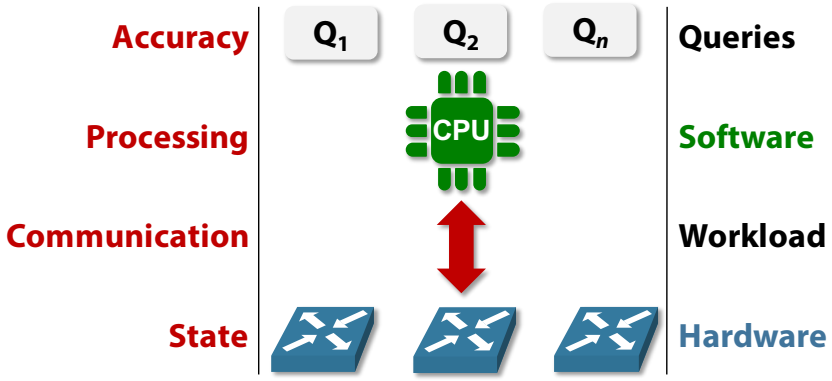


Figure 1.3: Network Telemetry Architecture. The system integrates software and hardware processing resources given a workload to answer a set of input queries. The system does this while considering computation, memory, and bandwidth constraints to provide maximum accuracy.

1.5 Contributions

This thesis presents an architecture for a network telemetry system that is flexible, scalable, and network-wide. We combine query partitioning with efficient coordination to enable scalable, network-wide telemetry that can support diverse, fine-grained, and real-time queries.

1.5.1 Architecture for Network Telemetry Systems

Figure 1.3 shows an overarching architecture for a network telemetry system. Our system takes as input a set of high-level telemetry queries and a target workload. The system then must choose which (portions of) queries to execute either in software or in hardware. The system must then choose a partitioning given the available computation, memory, and bandwidth resources provided by those devices while maximizing the accuracy of results returned to the queries.

1.5.2 Sonata: Expressive Telemetry Queries with PISA Switches

First, we present Sonata, a flexible and scalable network telemetry system that performs the collection and analysis of network traffic using the compute resources of both stream-processing servers and a Protocol Independent Switch Architecture (PISA) switch.

Compiling Sonata Queries to PISA Switches. The dataflow programming model, used by stream processors, and the PISA architecture are fundamentally similar. A dataflow program is a directed acyclic graph (DAG) of operators applied to structured data, i.e., a tuple. A PISA program is also a DAG where the operators are match-action tables and the structured data is a packet. Sonata takes advantage of this similarity to compile portions of high-level dataflow operators into PISA primitives for execution in the data plane.

Partitioning and Refining Sonata Queries with PISA Switches. Sonata then partitions the high-level queries into a portion that executes at a scalable stream processor and a portion that executes in a switch data plane. Sonata performs this partitioning by first modeling the constraints of PISA switches, such as memory available and number of processing stages, to solve an optimization problem. Sonata selects the partitioning plan that minimizes the processing load on the stream processor running on general purpose CPUs based on the set of input queries themselves and representative training data.

1.5.3 Herd: Network-Wide, Continuous Telemetry

A common idiom found in telemetry queries (See Table 2.4) counts flows to determine whether a specific threshold has been met (e.g., Figure 1.1). Herd is a system for implementing these kind of count-threshold queries distributed over a collection of switches under switch memory and bandwidth constraints. One such kind of count-threshold query determines *network-wide heavy hitters*, i.e., those flows that exceed some heavy-hitter threshold of packet counts from a network-wide perspective.

Continuous, Network-Wide Telemetry Herd builds upon existing techniques [18, 23] to practically implement a network-wide telemetry system. Herd counts flows at distributed switches without maintaining per-flow state and probabilistically reports them to a central coordinator. Based on these reports, the coordinator adapts system reporting parameters based on the distribution of flows among ingress switches. Whereas other monitoring systems have some interval of time on which they report statistics, Herd's reporting mechanism enables continuous network-wide monitoring irrespective of the specific interval chosen.

Flow Taxonomy for Modeling Resource Constraints. The traditional taxonomy of flows as either mice (i.e., insignificant) or elephants (i.e., sizeable) fails to sufficiently describe flows in a way that helps us reason about resource consumption. We extend this traditional taxonomy to reason about which flows affect switch memory usage (moles) and which flows affect bandwidth consumption (mules). We calculate the size of these sets in an offline algorithm for various parameter combinations to reason about the effect of those parameters on resource consumption.

Tunable Accuracy Under Resource Constraints. Performing network-wide telemetry on distributed switches communicating with a central coordinator involves a large number of parameters subject to memory and bandwidth constraints. We demonstrate the fundamental relationships between system parameters, such as number of nodes and reporting probability, and provide an offline algorithm to tune system parameters for highest accuracy subject to analytical bounds on those parameters.

1.6 Summary

Together, Sonata and Herd begin to realize the goal of a flexible, scalable, and network-wide telemetry system. Sonata gives network operators the ability to execute a set of telemetry queries from a single interface that combines the strengths of both programmable

data planes and general-purpose CPUs for flexibility and scalability. Herd efficiently coordinates to execute a subset of Sonata queries across a set of distributed switches continuously, scalably and with a network-wide view. We now explore each of these systems in more detail.

Chapter 2

Sonata: Expressive Telemetry Queries with PISA Switches

This chapter introduces the design, implementation, and evaluation of a flexible and scalable network telemetry system, Sonata, that partitions queries expressed in a high-level dataflow language across both a programmable switch and a stream processor. Sonata optimizes the use of switch resources to scalably execute telemetry queries as both the number of queries and overall traffic volume increase. For the purposes of exposition, we will refer to early versions of this system as the Sonata architecture and the early Sonata prototype.

2.1 Overview and Background

Existing telemetry systems can collect and analyze network data in real time, but they either support a limited set of telemetry tasks [53, 62], or they incur substantial processing and storage costs as traffic rates and the number of queries increase [20, 85, 10]. This dichotomy arises, in part, due to the choice of technology that underlies these approaches.

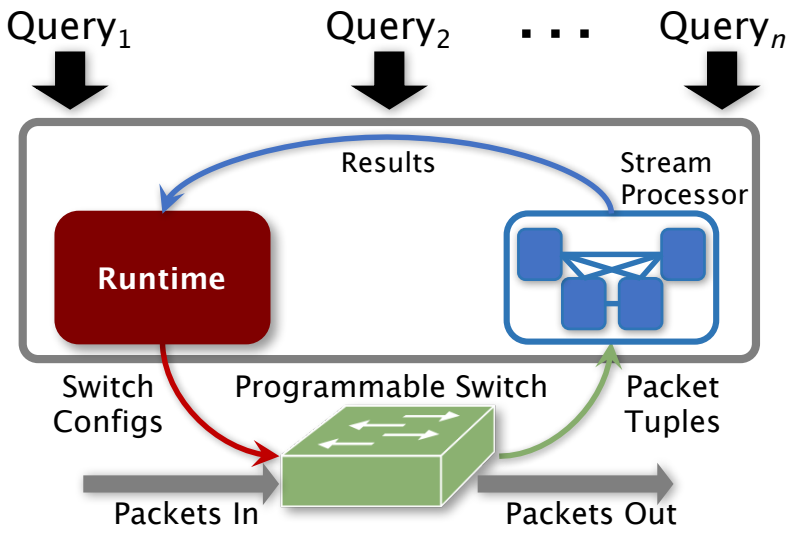


Figure 2.1: Sonata Architecture.

2.1.1 Existing Approaches: An Apparent Trade-Off

Existing telemetry systems typically trade off scalability for expressiveness, or vice versa. Telemetry systems that rely solely on stream processors are expressive but not scalable. For example, systems such as NetQRE [85] and OpenSOC [62] can support a wide range of queries using stream processors running on general-purpose CPUs, but they incur substantial bandwidth and processing costs to do so. Large networks can require performing as many as 100 million operations per second for rates of 1 Tbps and packet sizes of 1 KB. Scaling to these rates using modern stream processors is prohibitively costly due to the lower (2–3 orders of magnitude) processing capacity per core [58, 60, 86, 63]. On the other hand, telemetry systems that rely on programmable switches alone can scale to high traffic rates, but they give up expressiveness to achieve this scalability. For example, Marple [53] and OpenSketch [83], can perform telemetry tasks by executing queries solely in the data plane at line rate, but the queries that they can support are limited by the capabilities and memory in the data plane.

2.1.2 Sonata Architecture: Expressive and Scalable Telemetry

Rather than accepting this apparent tradeoff between expressiveness and scalability, prior work [27, 28] observed that an opportunity exists to combine the strengths of both technologies into a single telemetry system. Such a system could support expressive queries, while still operating at line rate for high traffic volumes, when possible. Figure 2.1 shows the high-level design of the Sonata architecture; it provides a single declarative interface that can express queries for a wide range of telemetry tasks and also frees the network operator from reasoning about where or how the query will execute. To scalably execute these expressive queries, Sonata relies on two techniques: query partitioning, and dynamic refinement [27].

Query Partitioning

While scalable stream processors can support very expressive queries, they still process traffic at speeds slower than current (and future) line rates. However, prior work [27] observed that stream processors do not have to process *all* of the network traffic flowing at line rates. Instead, we can reduce the amount of data that the stream processor must handle by partitioning queries into a portion that can be executed directly in programmable switches and a portion that can be executed at the stream processor. For example, suppose a network operator wanted to analyze DNS requests in their network. Even simple, fixed-function switches could support partitioning this query into two portions: one that runs on the switch matching packets destined for DNS servers and forwarding them to a stream processor, and another that performs the remainder of the analysis on a flexible stream processor. In this case, we avoid processing all non-DNS traffic at the stream processor.

Iterative Refinement

For certain queries and workloads, partitioning a portion of the queries to the switch does not reduce the workload on the stream processor enough. In these situations, the Sonata ar-

chitecture relies on dynamic query refinement [27] to further reduce the load on the stream processor. By rewriting the input queries to start at a coarser level of granularity than specified in the original query, we can choose to process queries at finer granularities only after they have satisfied the same query at a coarser granularity. For example, consider the case where a query computes a statistic for individual IP Addresses. This query would require computing that statistic for up to 2^{32} addresses. If instead we computed the statistic for entire networks that share the same 8-bit prefix, we would only have to compute the statistic for 2^8 different networks. We then consider computing the statistic at finer granularities (e.g., /16, /24, /32) only within the /8 addresses that satisfy the first stage of analysis.

Early Sonata Prototype and Results

Prior work [27] focused their initial prototype and analysis with a single data-plane target in mind, i.e., OpenFlow [1] compatible switches. OpenFlow-compatible switches can be reprogrammed in a limited way, but they can only support filtering and sampling packets. This focus limited the set of queries that the early Sonata prototype could partition to the data plane. However, their analysis demonstrated that partitioning and refining queries with only these two operations available in the data plane could still substantially reduce the load on the stream processor.

2.1.3 Realizing the Sonata Architecture with PISA Switches

The Sonata architecture [27] first proposed the ideas of query partitioning and refinement but focused on a limited data-plane target. Later work [28], expanded on these ideas and proposed to incorporate PISA switches as candidate data-plane targets. We build upon both of these works to present Sonata: an instantiation of the original Sonata architecture with PISA switch targets. We present the following contributions.

Compiling Sonata Queries to PISA Primitives. (Section 2.3) The Sonata architecture allows queries to be expressed in a high-level dataflow programming language, similar to those used by stream processors. This dataflow programming paradigm is fundamentally similar to the PISA programming model. Sonata exploits this similarity by first demonstrating that several high-level dataflow operators can be compiled to PISA primitives expressed in the P4 [11] programming language. We then demonstrate how to combine these primitives into a single P4 program for executing on PISA switches.

Partitioning and Refining Sonata Queries for PISA Switches. (Section 2.4) While we may be able to compile certain Sonata queries to PISA primitives, we must still choose how to partition and dynamically refine input queries to reduce the load on the stream processor. Given the limited resources available in PISA switches, such as switch memory and processing stages, we model PISA resource consumption and constraints as an Integer Linear Program (ILP). Sonata's query planner uses this model to decide how to partition query execution between the switch and the stream processor. We then extend this model to incorporate dynamic query refinement and, using representative packet traces, we select the refinement plan that makes best use of the limited resources while optimally reducing the load on the stream processor.

Modular and Extensible Software Architecture. (Section 2.5) To support different types of data-plane and streaming targets, we expand upon the original design [27] for the Sonata architecture and add support for operations over arbitrary packet fields. The queries expressed using the Sonata interface are agnostic to the underlying switch and streaming targets. Our current prototype implements drivers for both hardware (e.g., Barefoot Tofino [74]) and software (e.g., BMV2 [75]) protocol-independent switches as well as the Spark Streaming [73] stream processor. The current prototype parses packet headers for several common protocols but can be extended to extract other information, such as queue size [32].

```
1 packetStream(W)
2 .filter(p => p.tcp.flags == 2)
3 .map(p => (p.dIP, 1))
4 .reduce(keys=(dIP,), f=sum)
5 .filter((dIP, count) => count > Th)
```

Query 2.1: Detect Newly Opened TCP Connections.

We use real packet traces from operational networks to demonstrate that Sonata's query planner reduces the load on the stream processor by as much as *seven orders of magnitude* over existing telemetry systems (Section 2.6). We also quantify how Sonata's performance gains depend on data-plane constraints and traffic dynamics. To date, our open-source software prototype has been used by both researchers at a large ISP and in a graduate networking course [7].

2.2 Example Sonata Telemetry Queries

Many network telemetry queries require computing aggregate statistics over a subset of traffic and joining the results from multiple queries, which can be expressed as a sequential composition of dataflow operators (e.g., filter, map, reduce). We now present three example queries: one that executes entirely in the data plane, a second that involves a join of two subqueries, and a third that requires parsing packet payloads. Table 2.4 summarizes the queries that we have implemented and released publicly along with the Sonata software [77].

Computing Aggregate Statistics on a Subset of Traffic Suppose that an operator wants to detect hosts that have recently opened too many TCP connections, as in a SYN flood attack. Detection requires parsing each packet's TCP flags and destination IP address, as well as computing a sum over the destination IP address field. Query 2.1 first applies a `filter` operation (line 2) over the entire packet stream to select TCP packets with just the `SYN` flag set. It then counts the number of packets it observed for each host (lines

```
1     packetStream
2     .filter(p => p.proto == TCP)
3     .map(p => (p.dIP,p.sIP,p.tcp.sPort))
4     .distinct()
5     .map((dIP,sIP,sPort) =>(dIP,1))
6     .reduce(keys=(dIP,), f=sum)
7     .join(keys=(dIP,), packetStream
8       .filter(p => p.proto == TCP)
9       .map(p => (p.dIP,p.pktlen))
10      .reduce(keys=(dIP,), f=sum)
11      .filter((dIP, bytes) => bytes > Th1) )
12    .map((dIP,(byte,con)) => (dIP,(con/byte))
13    .filter((dIP, con/byte) => (con/byte > Th2)
```

Query 2.2: Detect Slowloris Attacks.

3–4) and reports the hosts for which this count exceeds threshold `Th` at the end of the window (line 5). This query can be executed entirely on the switch, so existing systems (e.g., Marple [53]) can also execute this type of query at scale.

Joining the Results of Two Queries A more complex query involves joining the results from two subqueries. To detect a Slowloris attack [69], a network operator must identify hosts which use many TCP connections, each with low traffic volume. This query (Query 2.2) consists of two subqueries: the first subquery counts the number of unique connections by applying a `distinct`, followed by a `reduce` (lines 1–6). The second subquery counts the total bytes transferred for each host (lines 8–11). The query then joins the two results (line 7) to compute the average connections per byte (line 12) and reports hosts whose average number of connections per byte exceeds a threshold `Th2` (line 13). Marple [53] cannot support this query as it applies a join after an aggregation operation (reduce). Also, this query cannot be executed entirely in the data plane as computing an average requires performing a division operation. Even state-of-the-art programmable switches (e.g., Barefoot Tofino [74]) do not support the division operation in the data plane. In general, existing approaches that only use the data plane for query execution cannot support queries that require computation not available in the data plane. In contrast, Sonata's

```
1  packetStream
2  .filter(p => p.tcp.dPort == 23)
3  .join(keys=(dIP,), packetStream
4    .filter(p => p.tcp.dPort == 23)
5    .map(p => ((p.dIP,p.nBytes/N), 1))
6    .reduce(keys=(dIP, nBytes), f=sum)
7    .filter(((dIP,nBytes),cnt1) => cnt1 > Th1))
8  .filter(p => p.payload.contains('zorro'))
9  .map(p => (p.dIP,1))
10 .reduce(keys=(dIP,), f=sum)
11 .filter((dIP, count2) => count2 > Th2)
```

Query 2.3: Detect Zorro Attacks.

query planner partitions queries for partial execution on the switch and performs more complex computations at the stream processor.

Note that the second subquery is equivalent to detecting hosts for which the average bytes per connection is *less than* a threshold. While Sonata allows a user to express the query using both operators, the "greater than" condition allows a more efficient query execution which we explain in Section 2.4. Ideally, Sonata's runtime would identify this optimization and modify queries written in the less-efficient form when possible.

Processing Packet Payloads. Consider the problem of detecting the spread of malware via telnet [54], which is a common tactic when targeting IoT devices [2]. Here, miscreants use brute force to gain shell access to vulnerable Internet-connected devices. Upon successful login, they issue a sequence of shell commands, one of which contains the keyword "zorro". The query to detect these attacks first looks for hosts that receive many similar-size telnet packets followed by a telnet packet with a payload containing the keyword "zorro". The query (Query 2.3) for this task has two subqueries: the first part identifies hosts that receive more than `Th1` similar-size telnet packets rounded off by a factor of `N` (lines 4–7). The second part joins (line 3) the output of the first subquery with the other and reports hosts that receive more than `Th2` packets and contain the keyword "zorro" in the payload (lines 8–11). Since this query requires parsing packet payloads, existing data-plane-based approaches cannot support it. In contrast, Sonata can support and scale these queries by

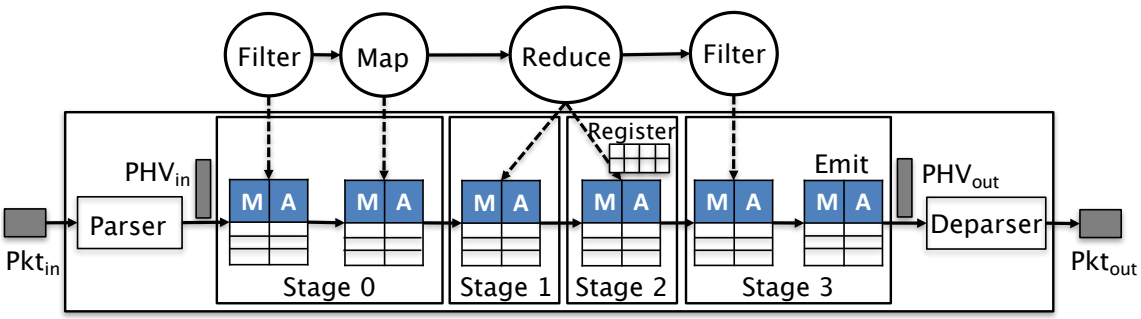


Figure 2.2: Compiling a dataflow query (Query 2.1) to a sequence of match-action tables for a PISA switch. Each query consists of an ordered sequence of dataflow operators, which are then mapped to match-action tables in the data plane.

performing as much computation as possible on the switch and then performing the rest at the stream processor.

2.3 Compiling Sonata Queries to PISA Switches

A central contribution of Sonata is to use the capabilities of programmable switches to reduce the load on the stream processor. In contrast to conventional switches, protocol-independent switch architecture (PISA) switches (e.g., RMT [12], Barefoot Tofino [34], Netronome [78]) offer programmable parsing and customizable packet-processing pipelines, as well as general-purpose registers for stateful operations. These features provide opportunities for Sonata to execute portions of queries on the switch, reducing the amount of data sent to the stream processor. Section 2.3.1 describes the PISA architecture and how it relates to Sonata's dataflow-like query interface. Section 2.3.2 describes how to compile individual operators to elements of a PISA program, and Section 2.3.3 describes how to compile an entire query to into a complete PISA program.

2.3.1 PISA Processing Model

Figure 2.2 shows how Query 2.1 naturally maps to the capabilities of the packet processing model of a PISA switch. On PISA switches, a reconfigurable parser constructs a packet

header vector (PHV) for each incoming packet. The PHV contains not only fixed-size standard packet headers but also custom metadata for additional information such as queue size. A fixed number of physical stages, each containing one match-action unit (MAU), then processes the PHVs. These MAUs each consist of some memory (e.g., SRAM, TCAM) and arithmetic-logic units (ALU) for consuming PHVs as input and emitting transformed PHVs as output. If fields in the PHV are matched by the MAU, then a set of custom actions are applied to the PHV. These actions can be stateless or stateful. Operations that maintain state across sequences of packets may only read and write back that state in a single physical stage. Other stages may not access that state with the exception that a copy may be stored in metadata for reading in subsequent stages. Finally, a deparser serializes the modified PHV and original payload into a packet before sending it to an output port.

Match-Action Abstraction. Atop this low-level architecture sits the *match-action table* abstraction. The match-action table is a logical table that consists of two kinds of columns: match columns and action columns. Each header field that the table *could* match on is a column and a single action column defines which actions could be taken, based on a corresponding match. At compile time, the structure of this table is known, i.e., which header fields are match columns, what kind of match they perform (e.g., exact or ternary), and which actions could be taken on a match. At run time, these tables are populated with entries, or rows, that contain specific values to match on and which action(s) to take upon a match. For example, consider a match-action table that performs IPv4 forwarding. Such a table might be called "Forwarding" and would consist of a single match column, `ipv4.dstIP`, and an action column that decides to forward a packet out a specific port or drop it. An entry in such a table would contain a specific IPv4 prefix to match on, and which port to send it out.

Match-action tables abstract from the programmer the physical processing stages in the PISA architecture. These tables represent the resources available in each physical stage,

i.e., MAUs, however, a logical match-action table may be mapped across several physical stages [12]. Match-action tables are the fundamental unit of processing in the PISA architecture. Whole data plane programs (i.e., packet processing pipelines) are expressed in the P4 language as a sequence of match-action tables, applied conditionally, in a user-defined control flow. A target-specific compiler maps this program consisting of logical match-action tables to the physical architecture described above.

Similarity to Dataflow. The PISA processing model aligns well with streaming analytics platforms, such as, Spark Streaming [86] or Apache Flink [59], that use a dataflow programming paradigm. The processing pipelines for both models can be represented as a directed, acyclic graph (DAG) where each node in the graph performs some computation on an incoming stream of structured data. For stream processors, the nodes in the DAG are dataflow operators and the stream of structured data consists of tuples. For PISA switches, the nodes in the DAG are match-action tables and the stream of structured data consists of packet header vectors. Given this inherent similarity, an ordered set of dataflow query operators could map to an ordered set of match-action tables in the data plane.

2.3.2 Compiling Individual Operators

Compiling dataflow queries to a PISA switch requires translating each operator in the query to a component of a P4 program, i.e., a match-action table or logic in the control flow. Rather than constraining the set of input queries to only those supported directly in the data plane, Sonata accepts the superset of queries executable both in the data plane and at a stream processor. Sonata then relies on its query planner to partition all input queries into a set of dataflow operators that can be executed on the switch and a set that must be executed at the stream processor. Before Sonata's query planner can make this partitioning decision, we must first quantify the resources required to compile individual dataflow operators. We now consider how each of the operators from Table 1.1 would be executed in the data plane.

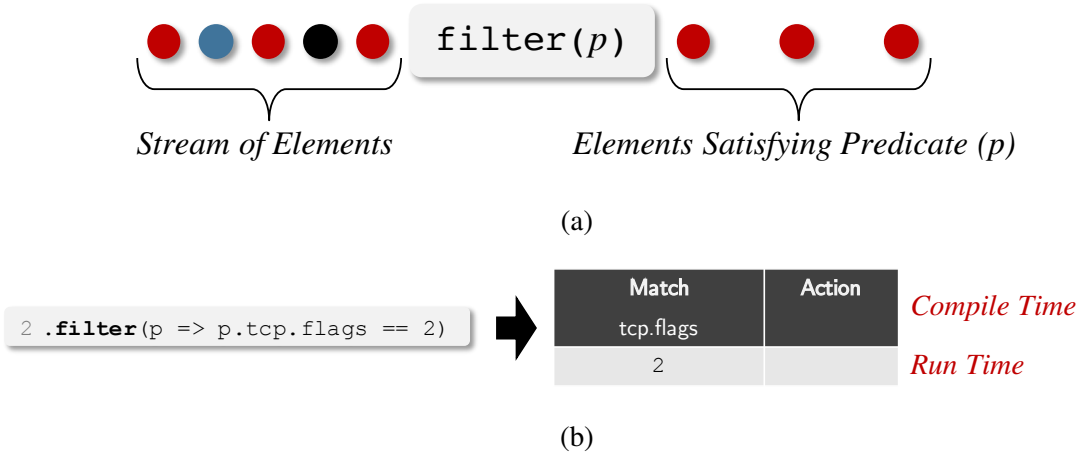
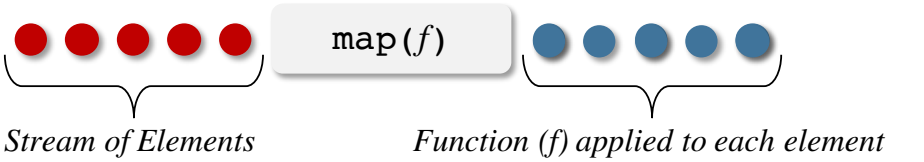


Figure 2.3: Filter. (a) The filter operator's high-level behavior. (b) Compiling the filter operator from Query 2.1 into a match-action table.

Filter

The `filter` operator takes a stream of elements as input and returns a set of elements that match some predicate (p), as shown in Figure 2.3a. In the figure, the predicate matches only red elements and all others are discarded. This operator only requires a single match-action table to match a set of fields in the PHV. For example, Figure 2.3b shows the match-action table we generate for line 2 of Query 2.1. The six-bit `tcp.flags` field becomes a match column and the value `2` is inserted into the table as a single entry or rule. In general, the match-action table for a `filter` operation has a column for each field in the predicate. A filter predicate with multiple clauses connected by the logical and operator corresponds to multiple match-columns, one per clause. A filter operator can also be implemented in the control flow of a P4 program using if-then-else logic. If the predicate (p) cannot be matched on a PISA switch, e.g., matching on a packet payload, then the operator cannot be implemented on the switch. However, these operators can still be executed at the stream processor.



(a)

```
3 .map(p => (p.dstIP, 1))
```

Match	Action
	meta.field1 = dstIP meta.field2 = 1

(b)

Figure 2.4: Map. (a) The map operator's high-level behavior. (b) Compiling the map operator from Query 2.1 into a match-action table.

Map

The map operator takes a stream of elements as input and returns the stream of elements after applying some function (f) to each element. As long as the function to be applied is executable in the data plane, then the map operator can be implemented with a single match-action table where the function is applied as an action. For example, line 3 of Query 2.1 transforms all incoming packets into a tuple consisting of the `ipv4.dIP` field from the packet's header and the value `1`. These values are stored in query-specific metadata for further processing. Although Sonata's query interface does not constrain the set of transformation functions that `map` might perform over a set of tuples, the operator cannot be compiled to the data plane if the switch cannot perform that function. Again, any operators that cannot be compiled to PISA primitives can still be executed at the stream processor.

Reduce

The reduce operator must maintain state across sequences of packets until a discrete window of time has elapsed; Sonata uses the PISA register primitive, which is simply an array of values indexed by some key, to do so. Query-specific metadata fields permit loading and

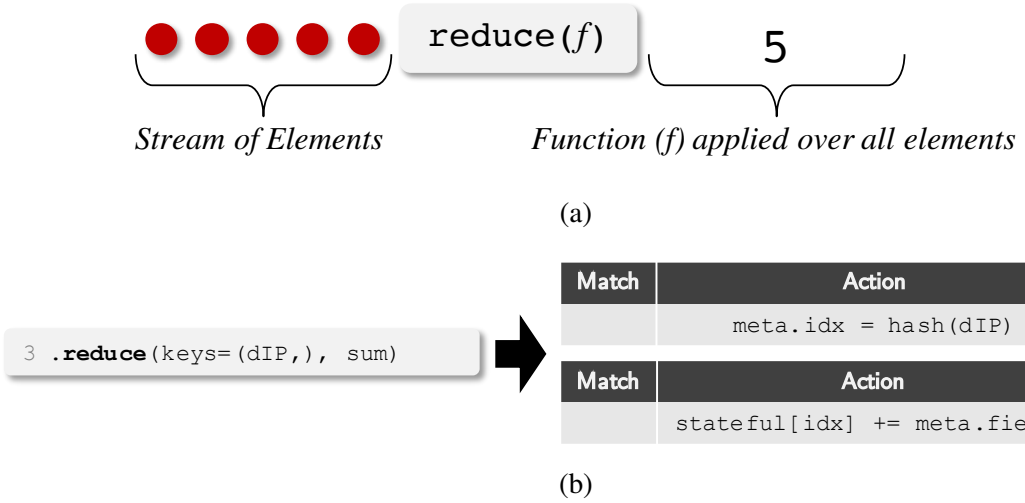


Figure 2.5: Reduce. (a) The reduce operator's high-level behavior. (b) Compiling the reduce operator from Query 2.1 into two match-action tables.

storing values from the registers. As a result, stateful operations require two match-action tables: one for computing the index of the value stored in the array and the other for updating state using arithmetic operators supported by the switch, such as `add` and `bit_or`. A corresponding metadata field carries the updated state after applying the arithmetic operation. For example, executing the `reduce` operator for Query 2.1 in Figure 2.2 requires a match-action table to compute an index into the register using the `dIP` header field. A second table performs the stateful action that increments the indexed value in the register and stores the updated value. In Section 2.4.2, we describe how Sonata's query planner uses representative training data to configure the number of entries for each register.

Distinct

Compiling the `distinct` operator is very similar to compiling the `reduce` operator where the function applied is `bit_or` with the argument 1. It likewise requires two tables, one for calculating an index into the stateful array and a second to read and update its value.

```
1  header_type app_metadata_t {
2      fields {
3          drop_q100: 1;
4          satisfied_q100: 1;
5          report: 1;
6      }
7  }
8
9  metadata app_metadata_t app_metadata;
10
11 header_type metadata_q100_t {
12     fields {
13         qid: 16;
14         count: 16;
15         dstIP: 32;
16         tcp_flags: 8;
17         index: 16;
18     }
19 }
20
21 metadata metadata_q100_t metadata_q100;
```

Figure 2.4: Additional Metadata for Query 2.1

Join

A join operation is costly to execute in the data plane. In the worst case, this operation maintains state that grows with the square of the number of packets. Sonata executes `join` operations at the stream processor by iteratively dividing the query into a set of subqueries. For example, Sonata divides Query 2.2 into two subqueries: one that computes the number of unique connections, and a second that computes the number of bytes transferred for each host. Sonata independently decides how to execute the two subqueries and ultimately joins their results at the stream processor.

2.3.3 Compiling Sequences of Operators

After compiling individual operators into match-action tables, Sonata must synthesize an entire P4 program from those tables. We describe some design considerations with respect to Query 2.1.

```
1 header_type out_header_q100_t {
2     fields {
3         qid : 16;
4         ipv4_dstIP : 32;
5         index : 16;
6     }
7 }
8
9 header out_header_q100_t out_header_q100;
```

Figure 2.5: Reporting Results for Query 2.1

Preserving packet forwarding decisions. Sonata preserves packet forwarding decisions by transforming only query-specific metadata fields rather than the actual packet contents that might affect forwarding decisions (e.g., destination address or port). The switch extracts values from the packets' original header fields and copies them to auxiliary metadata fields before performing any additional processing. This process leaves the original packet unmodified. Figure 2.4 shows the application level and query-specific metadata that Sonata's data-plane driver (Section 3.6) generates while implementing Query 2.1. The variable `app_metadata` has fields that track whether or not to continue processing a packet for a given query, in this case query `100`, and whether or not it has been satisfied to report to the stream processor. The variable `metadata_q100` stores a copy of the query-specific fields needed from the packet to process the query.

Reporting intermediate results to the stream processor. When a query is partitioned across the stream processor and the switch, the stream processor may need either the original packet or just an intermediate result from the switch so that it can perform its portion of the query. To facilitate this reporting, the switch maintains a one-bit `report` field in the metadata for each packet. Each subquery partitioned to the switch marks this field whenever a query-specific condition is met that requires the packet be sent to the stream processor. If this field is set at the conclusion of the entire processing pipeline, the switch sends to the stream processor all intermediate results needed to complete processing the query, including the original packet, if needed. If the last operator is stateful (e.g., `reduce`),

then the switch sends only one packet for each key to the stream processor. This informs the stream processor which register indices in the data plane must be polled at the end of each window to retrieve aggregated values stored in the switch (see Section 2.5 for details). Figure 2.5, shows the information needed by the stream processor for Query 2.1 stored in header `out_header_q100`.

Putting it All Together. Figure 2.6 shows the overall control flow that implements Query 2.1. In lines 2-4, the program initializes the application and query-specific metadata. In lines 6-18, tables that implement each operator in the query are applied in the same sequence as specified in the original query. At the conclusion of the query, lines 24-25 look to see if the query was marked as satisfied for reporting to the stream processor. On line 30, the program checks to determine if a given packet is an original packet to apply the desired forwarding logic. Otherwise, the packet is a clone that resulted from satisfying the query and must be reported to the stream processor. On lines 37-39, the data needed by the stream processor is appended to the packet and sent.

2.4 Executing Sonata Queries with PISA Switches

Now that we know we can implement Sonata queries on PISA switches, we must now choose which queries to partition across a stream processor and a PISA switch, given available switch resources, such that it minimizes the processing load on the stream processor. Section 2.4.1 discusses the constraints of PISA switches that Sonata's query planner considers. The planner then solves an optimization problem to partition the query in Section 2.4.2. We then describe how to extend our optimization problem to account for dynamic query refinement in Section 2.4.3

```
control ingress {
    apply(init_app_metadata);
    // query q100
    apply(mapinit_q100_1);
    // .filter(p => p.tcp.flags == 2)
    apply(filter_q100_2);
    if (app_metadata.drop_q100 != 1) {
            // .map(p => (p.dIP, 1))
            apply(map_q100_3);
            // .reduce(keys=(dIP,), f=sum)
            apply(init_reduce_q100_4);
            // .filter((dIP, count) => count > Th)
            if (reduce_q100_4.value == 40) {
                apply(continue_reduce_q100_4);
            }
            else {
                apply(drop_reduce_q100_4);
            }
            if (app_metadata.drop_q100 != 1) {
                    apply(mark_satisfied_q100);
            }
    }
    // clone packet for reporting
    if (app_metadata.report == 1) {
        apply(report_packet);
    }
}

control egress {
    if (standard_metadata.instance_type == 0) {
        // original packet, apply forwarding
    }

    else if (standard_metadata.instance_type == 1) {
        // cloned packet, report to stream processor
        if (app_metadata.satisfied_q100 == 1) {
            apply(add_out_header_q100);
        }
        apply(add_final_header);
    }
}
```

Figure 2.6: Control Flow for Query 2.1

2.4.1 Resource Constraints on PISA Switches

Sonata's query planner must consider the resource constraints of PISA switches for parsing packet header fields, performing actions on packets, executing stateful operations, and performing all of these operations in a limited number of stages.

Parser. The cost of parsing increases with the number of fields to extract from the packet. This cost is quantified as the number of bits to extract and the depth of the parsing tree. The size of the PHV limits the number of fields that can be extracted for processing. Typically, PISA switches have PHVs about 0.5–8 Kb [12] in size. Let M denote the maximum storage for metadata in the PHV.

Actions. Most stream processors execute multiple queries in parallel, where each query operates over its own logical copy of the input tuple. In contrast, PISA switches transform raw packets into PHVs and then concurrently apply multiple operations over the PHV in pipelined stages. These mechanisms suggest that PISA switches would be amenable to parallel query execution. In practice, there is a limit on how many actions can be applied over a PHV in one stage, which limits the number of queries that can be supported in the data plane. Typically, PISA switches support 100–200 stateless and 1–32 stateful actions per stage [12]; we denote the maximum number of stateful actions per stage as A.

Registers. The amount of memory required to perform stateful operations grows with the number of packets and the number of queries. Stream processors scale by adding more nodes for maintaining additional state. In contrast, operations that must maintain state across packets in PISA switches can be read and written back only in a single physical stage. The amount of memory for these operations is also bounded for each stage, which affects the switch's ability to handle both increased traffic loads and additional queries. Within a stage, the amount of memory available for a single operation is also bounded. Typically, PISA switches support 0.1–4 MB memory for each stage [12]. Let B denote the maximum number of bits available in each stage for stateful operations.

Stages. Match-action tables must be executed in physical stages. If a given stage lacks the resources to implement a match-action table, that table can be executed in a later stage.

Switch Constraints	
M	Amount of metadata stored in switch.
A	Number of stateful actions per stage.
B	Register memory (in bits) per stage.
S	Number of stages in match-action pipeline.
Input from Queries	
O_q	Ordered set of dataflow operators for query q.
T_q	Ordered set of match-action tables for query q.
M_q	Amount of metadata required to perform query q.
Z_t	Indicates whether table t performs a stateful operation.
Input from Workload	
$N_{q,t}$	Number of packets generated after table t of query q.
$B_{q,t}$	State (bits) required for executing table t of query q.
Output	
$P_{q,t}$	Indicates whether t is the last table partitioned to the switch for query q.
$X_{q,t,s}$	Indicates whether table t of query q executes at stage s in the switch.
$S_{q,t}$	Stage id for table t for query q.

Table 2.1: Summary of variables in the query planning problem.

It could also be split across physical stages. PISA switches typically support 1–32 physical stages [12]; we denote the maximum number of stages as S.

Effect of Constraints on Query Planning Consider a switch with $S = 4$ stages, $B = 3,000$ Kb, and $A = 4$ stateful actions per stage. These constraints are more strict than Barefoot's Tofino switch [74], but they illustrate how the data-plane resource constraints affect query planning. Sonata runs Query 2.1 over a one-minute packet trace from CAIDA [21] to compute that the switch requires $2,500$ Kb to count the number of TCP SYN packets per host (Figure 2.8). Since $2,500$ Kb $< B$, Sonata can execute the entire query on the switch, sending only the 77 tuples that satisfy the query to the stream processor. If B or S were smaller, Sonata could not execute the `reduce` operator on the switch and would need to execute the rest of the query at the stream processor. We now describe how Sonata considers these resource constraints to partition queries across the switch data plane and stream processor.

2.4.2 Query Partitioning as an Integer Linear Program

Sonata's query planner solves an Integer Linear Program (ILP) that minimizes the number of packet tuples processed by the stream processor, based on a partitioning plan and subject to switch constraints, as summarized in Table 2.2. Our approach is inspired by previous work on a different problem that partitions multiple logical tables across physical tables [37]. Table 2.1 summarizes the variables in the query planning problem. To select a partitioning plan, the query planner determines the capabilities of the underlying switch, estimates the data-plane resources needed to execute individual queries, and estimates the number of packets sent to the stream processor given a partitioning of operators on the switch.

Input. For the set of input queries (Q), Sonata interacts with the switch to compile the ordered set of dataflow operators in each query (O_q) to an ordered set of match-action tables (T_q) that implement the operators on the switch. In some cases, more than one dataflow operator can be compiled to the same table. For instance, the `filter` operator that checks the threshold after the reduce in Query 2.1 can be compiled to the same table as the reduce operator. Z_t indicates to the query planner whether a given table contains a stateful operator.

Using training data in the form of historical packet traces, the query planner estimates the number of packet tuples ($N_{q,t}$) sent to the stream processor and the amount of state ($B_{q,t}$) required to execute table t for query q on the switch. The planner applies all of the packets in the historical traces to each query q. After applying each table t that contains a stateful operator, the planner estimates the amount of state required to perform the stateful operation based on the total number of keys processed in the historical traces. It also estimates the number of packets sent to the stream processor ($N_{q,t}$) after table t processes the packets from the historical traces. The planner divides the historical traces into time

Goal
$min(N = \sum_{q}\sum_{t} P_{q,t} \cdot N_{q,t})$

Constraints	
$C1:$	$\forall s : \sum_{q}\sum_{T_q} X_{q,t,s} \cdot B_{q,t} \leq B$
$C2:$	$\forall s : \sum_{q}\sum_{T_q} Z_t \cdot X_{q,t,s} \leq A$
$C3:$	$\forall q,t : S_{q,t} < S$
$C4:$	$\forall q, i < j, i, j \in T_q : S_{q,j} > S_{q,i}$
$C5:$	$\forall q : \sum_{q} M_q \leq M$

Table 2.2: ILP formulation for the query partitioning problem.

windows of size W, computes $B_{q,t}$ and $N_{q,t}$ per window, and inputs the median value across all intervals to the ILP.

Objective. The objective of Sonata's query planning ILP is to minimize the number of tuples processed by the stream processor. The query planner models this objective by introducing a binary decision variable $P_{q,t}$ that captures the partitioning decision for each query; $P_{q,t} = 1$ if t is the last table for query q that is executed on the switch. For each query, only one table corresponding to one operator can be set as the last table on the switch: $\sum_{T_q} P_{q,t} \leq 1$. The total number of packets processed by the stream processor is then the sum of all packets emitted by the last table processed on the switch for all queries.

Switch constraints. To ensure that Sonata respects the constraints from Section 2.4.1, we introduce variables X and S. $X_{q,t,s}$ is a binary variable that reflects stage assignment: $X_{q,t,s} = 1$ only if table t for query q executes at stage s in the match-action pipeline. Similarly, $S_{q,t}$ returns the stage number where table t for query q is executed. These two variables are related: if $X_{q,t,s} = 1$, then $S_{q,t} = s$ for a given stage. We will now summarize how Sonata's query planner models various data-plane constraints.

C1: Register Memory per stage (B). For each stage, the amount of state allocated for Sonata's packet processing cannot exceed B. Since PISA targets can only configure tables with stateful operations in a single stage, the amount of state required to execute query q at stage s is $\sum_{T_q} X_{q,t,s} \cdot B_{q,t}$. This sum over all queries captures the total memory required for each stage s.

C2: Number of Actions per stage (A). For each stage, the total number of stateful operations cannot exceed A. We can again use the X variable to model this constraint. The expression $\sum_{T_q} Z_t \cdot X_{q,t,s}$ captures the number of stateful operations performed at stage s for query q. This sum over all queries captures the total number of stateful actions for each stage s.

C3: Number of Stages (S). The total number of stages required to execute a query in the data plane cannot exceed S. The variable $S_{q,t}$ represents the stage where table t for query q is executed. For every table of each query, this variable should always be less than S because the last stage is reserved to determine which packet needs to be reported to the stream processor.

C4: Intra-Query Ordering. We can also use S to express intra-query ordering constraints. For example, in the Slowloris query (Query 2.2), the tables for the `reduce` operator can only be executed after the `distinct` operator has been applied in a previous stage. For each query q and any two indices (i, j) in the ordered set of tables T_q where $(i < j)$, $S_{q,j}$ is always greater than $S_{q,i}$.

C5: Total Metadata (M). Finally, since the PHV consists of a fixed-size, (M) represents the maximum space available in the PHV to add query-specific metadata fields. The total metadata used for all queries must then be less than M, i.e., $\sum_q M_q \leq M$.

Accounting for Traffic Dynamics. The query planner uses training data to decide how to configure the number of entries (n) for each register, and how many registers (d) to use for each stateful operation. It is possible that the training data might underestimate the

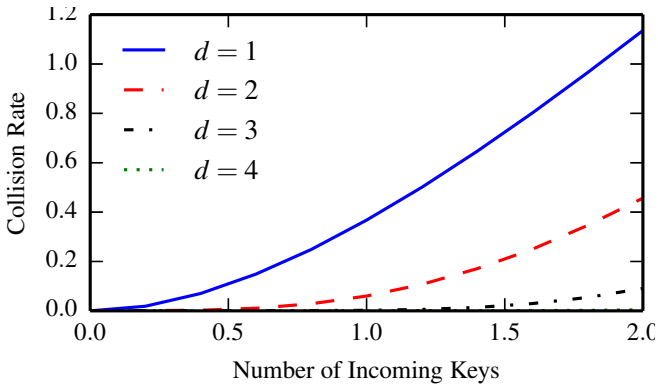


Figure 2.6: Relationship between collision rate and number of unique incoming keys.

number of expected keys (k) for a stateful operation due to variations in traffic patterns. In Figure 2.6, we show how the collision rates increase as the number of unique keys grows beyond the original estimated number of rows (n) in each of a sequence of (d) registers. Here, the x-axis is the number of incoming keys and y-axis is the collision rate—both normalized with respect to n. The collision rate increases as the number of incoming keys increases and decreases as d increases.

Since collision rates are predictable, we choose values of (n) and (d) to keep collision rates low but still high enough to send a signal to Sonata's runtime when the switch is storing many more unique keys than originally expected. Sonata's query planning ILP takes into consideration both the number of additional packets processed by the stream processor and the additional switch memory while computing the optimal query partitioning plans.

2.4.3 Dynamic Refinement as an Integer Linear Program

We now focus on extending our existing Integer Linear Program to also account for dyanmic query refinement. Sonata's query planner modifies the input queries to start at a coarser level of granularity than specified in the original query (Section 2.4.3). It then chooses a sequence of finer granularities that reduces the load on the stream processor. This process introduces additional delay in detecting the traffic that satisfies the input queries. The

specific levels of granularity chosen and the sequence in which they are applied constitute a *refinement plan*. To compute an optimal refinement plan for the set of input queries, Sonata's query planner estimates the cost of executing different refinement plans based on historical training data. Sonata's query planner then solves an extended version of the ILP from Section 2.4.2 that determines both partitioning as well as refinement plans to minimize the workload on the stream processor (Section 2.4.3).

Dynamic query refinement example. Sonata's query planner applies the augmented queries over the training data to generate Figure 2.8 for Query 2.1. This figure shows the costs to execute Query 2.1 with refinement key `dIP` and refinement levels $R = \{8, 16, 32\}$ over the training data. It shows the number of packets sent to the stream processor depending on which refinement level (r_{i+1}) is executed after level r_i. If only the `filter` operation is executed on the switch, then N_1 packets are sent to the stream processor. If the `reduce` operation is also executed on the switch, then N_2 packets are sent, but then B bits of state must also be maintained in the data plane. For simplicity of exposition, we assume that these counts remain the same for three consecutive windows.

Modifying Queries for Refinement

Identifying refinement keys. A refinement key is a field that has a hierarchical structure and is used as a key in a stateful dataflow operation. The hierarchical structure allows Sonata to replace a more specific key with a less specific version without missing any traffic that satisfies the original query. This applies to all queries that filter on aggregated counts greater than a threshold. For example, `dIP` has a hierarchical structure and is used as a key for aggregation in Query 2.1. As a result, the query planner selects `dIP` as a refinement key for this query. Other fields that have hierarchical structure can also serve as refinement keys, such as `dns.rr.name` and `ipv6.dIP`. For example, a query that detects malicious domains might count the number of unique IP addresses resolved for

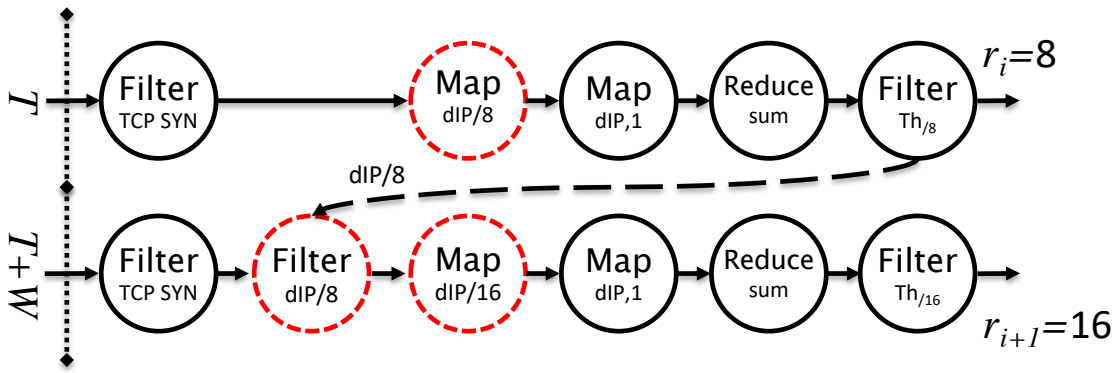


Figure 2.7: Query augmentation for Query 2.1. The query planner adds the operators shown in red to support refinement. Query 2.1 executes at refinement level $r_i =$ `/8` during window T and at level $r_{i+1} =$`/16` during window $(T+W)$. The dashed arrow shows the output from level r_i feeding a filter at level r_{i+1}.

each domain [9], and such a query can use the field `dns.rr.name` as a refinement key. In this case, a fully-qualified domain name is the finest refinement level and the root domain (`.`) is the coarsest. A query can contain multiple candidate refinement keys and Sonata independently selects refinement keys for each query. Also, note that expressing the second subquery in Query 2.2 as the one that reports flows for which the average connections per byte exceeds the threshold ensures that it can benefit from iterative refinement. Replacing a more specific key with a less specific one will not miss any traffic that satisfies the original query.

Enumerating refinement levels. After identifying candidate refinement keys, the query planner enumerates the possible levels of granularity for each key. Each refinement key consists of a set of levels $R = \{r_1 \ldots r_n\}$ where r_1 is the coarsest level and r_n is the finest. The inequality $r_1 > r_n$ means that r_1 is coarser than r_n. The semantics of the n^{th} refinement level is specific to each key; $n = 32$ would correspond to a `/32` IP prefix for the key `dIP` and $n = 2$ would correspond to second-level domain for the key `dns.rr.name`.

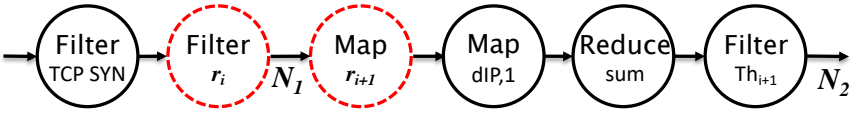
Augmenting input queries. To ensure that the finer refinement levels only consider the traffic that has already satisfied coarser ones, Sonata's query planner augments the input

queries. For example, Figure 2.7 shows how it augments Query 2.1 with refinement key `dIP` and $R = \{8, 16, 32\}$ to execute the query at level $r_{i+1} = 16$ after executing it at level $r_i = 8$. The query planner first adds a `map` at each level to transform the original reduction key into a count bucket for the current refinement level. For example, r_i and r_{i+1} rewrite `dIP` as `dIP/8` and `dIP/16`, respectively. By transforming the reduction key for each refinement level, the rest of the original query can remain unmodified. At refinement level r_{i+1}, the query planner also adds a `filter`. At the conclusion of the first time window, the runtime feeds as input to the `filter` operator the `dIP/8` addresses that satisfy the query at $r_i = 8$. This filtering ensures that refinement level r_{i+1} only considers traffic that satisfies the query at r_i.

Sonata's query planner also augments queries to increase the efficiency of executing refined queries. Since counting at coarser refinement levels (e.g., `/8`) will result in larger sums than at finer levels (e.g., `/32`), using the *original* query's threshold values at coarser refinement levels would still be correct but inefficient. Sonata's query planner instead uses training data to calculate relaxed threshold values for coarser refinement levels that do not sacrifice accuracy (e.g., $\texttt{Th}_{/8} > \texttt{Th}_{/16}$ in Figure 2.7). For each query and for each refinement level, the planner selects a relaxed threshold that is the minimum count for all keys satisfying the original query aggregated at that refinement level.

Note that by its very nature, dynamic refinement introduces additional delay (D) in detecting the traffic that satisfies the original input queries. In the worst case, Sonata can only identify network events lasting at least $W \times |R|$ seconds for each query. Here, W is the interval size and $|R|$ is the total number of refinement levels considered. However, by specifying an upper bound on the acceptable delay (D_q), the network operator can force Sonata to consider fewer refinement levels and reduce the delay to detect traffic that satisfies the original query.

Consider an approach, *Fixed-Refinement*, that applies a fixed refinement plan for all input queries. In this example, the query planner augments the original queries to always



$r_i \longrightarrow r_{i+1}$	N_1	B (Kb)	N_2
$* \longrightarrow 32$	570K	2,500	77
$* \longrightarrow 16$		180	99
$* \longrightarrow 8$		6	33
$8 \longrightarrow 32$	526K	1,900	77
$8 \longrightarrow 16$		50	98
$16 \longrightarrow 32$	450K	1,200	77

Figure 2.8: The N and B cost values for executing Query 2.1 at refinement level r_{i+1} after executing it at level r_i.

run at refinement levels 8, 16, and 32. The runtime updates the filter for the query at level 16 with the output from level 8 and the filter of level 32 with the output from 16. The costs of this plan are shown in rows $* \rightarrow 8$, $8 \rightarrow 16$, and $16 \rightarrow 32$ of Figure 2.8. If the switch only supported two stateful operations ($A = 2$), the `reduce` operator could only be performed on the switch for the first two refinement levels. This would result in sending 33 packets (N_2 for $* \rightarrow 8$) at the end of the first window, 98 packets (N_2 for $8 \rightarrow 16$) at the end of the second window, and 450,000 (N_1 for $16 \rightarrow 32$) packets at the end of the third window to the stream processor. Compared to the solution without any refinement from the beginning of Section 2.4.2, *Fixed-Refinement* reduces the number of tuples reported to the stream processor from 570 K to 450 K at the cost of delaying two additional time windows to detect traffic that satisfies the query.

In contrast, Sonata's query planner uses the costs in Figure 2.8 combined with the switch constraints to compute the refinement plan $* \rightarrow 8 \rightarrow 32$. Executing the query at refinement level $* \rightarrow 8$ requires only 6 Kb of state on the switch and sends 33 packet tuples to the stream processor at the end of the first window. Each packet represents an individual `dIP/8` prefix that satisfies the query in the first window. Sonata then applies the original input query (`dIP/32`) over these 33 `dIP/8` prefixes in the second window interval, processing 526,000 packets (N_1 for $8 \rightarrow 32$) and consuming only 1900 Kb on

Goal	
	$min(N = \sum_{q}\sum_{r_2} L_{q,t,r_2} \cdot N_{q,t,r_2})$
	$N_{q,t,r_2} = I_{q,r_2} \cdot \sum_{r_1} F_{q,r_1,r_2} \cdot N_{q,t,r_1,r_2}$
Constraints	
$C1:$	$\forall s : \sum_{q} X_{q,s,t} \cdot B_{q,t} \leq B_{max}$
	$B_{q,t} = \sum_{r_2} I_{q,r_2} \sum_{r_1} F_{q,r_1,r_2} \cdot B_{q,r_2,t}$
$C2:$	$\forall s : \sum_{q}\sum_{t} X_{q,t,s} \leq W_{max}$
	$X_{q,t,s} = \sum_{r} I_{q,r} \cdot X_{q,t,s,r}$
$C3:$	$\forall q,t,r : S_{q,t,r} \leq S_{max} - 1$
$C4:$	$\forall q,r,i < j : S_{q,j,r} < S_{q,i,r}$
$C5:$	$\forall q : \sum_{q}\sum_{r} I_{q,r} \cdot M_{q,r} \leq M$
$C6:$	$\forall q_i, q_j, r : I_{q_i,r} = I_{q_j,r}$
$C7:$	$\forall q : \sum_{r} I_{q,r} \leq D_q$

Table 2.3: Extended ILP to support dynamic refinement.

the switch. At the end of the second window, the switch reports 77 `dIP/32` addresses to the stream processor. This refinement plan sends 110 packet tuples to the stream processor over two window intervals, significantly reducing the workload on the stream processor while costing only one additional window of delay.

Extended ILP for Dynamic Refinement. The ILP for jointly computing partitioning and refinement plans is an extension of the ILP from Section 2.4.2. Table 2.3 presents the full version of the extended ILP, including these new constraints. The objective is the same, but the query planner must also compute the cost of executing combinations of refined queries (i.e., $N_{q,t,r}$ and $B_{q,t,r}$) to estimate the total cost of candidate query plans. We add new decision variables $I_{q,r}$ and F_{q,r_1,r_2} to model the workload on the stream processor in the presence of refined queries. $I_{q,r}$ is set to one if the refinement plan for query q includes level r. F_{q,r_1,r_2} is set to one if level r_2 is executed after r_1 for query q. These two variables are related by $\sum_{r_1} F_{q,r_1,r_2} = I_{q,r_2}$. We also augment X and S variables with subscripts to account for refinement levels.

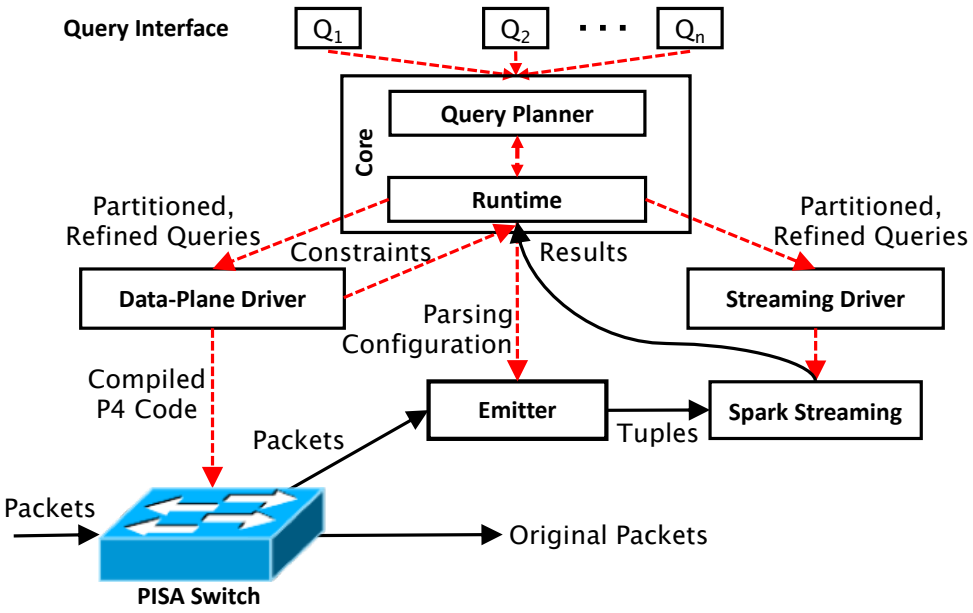


Figure 2.9: Sonata's Design and Implementation: red arrows show compilation control flow and black ones show packet/tuple data flow

Additional constraints. For queries containing `join` operators, the query planner can select refinement keys for each subquery separately, but it must ensure that both subqueries use the same refinement plan. We then add the constraint $\forall q, r$ and $\forall q_i, q_j \in q : I_{q_i,r} = I_{q_j,r}$. The variables q_i and q_j represent subqueries of query q containing a `join` operation. The query planner also limits the maximum detection delay for each query, $\forall q : \sum_r I_{q,r} \leq D_q$. Here, D_q is the maximum delay query q can tolerate expressed as a number of time windows.

2.5 Design and Implementation

Figure 2.9 illustrates the design for Sonata's implementation. For each query, the *core* generates partitioned and refined queries and *drivers* compile the parts of each query to the appropriate component. When packets arrive at the PISA switch, Sonata applies the packet-processing pipelines and mirrors the appropriate packets to a monitoring port, where a software *emitter* parses the packets and sends the corresponding tuples to the stream

processor. The stream processor reports the results of the queries to the runtime, which then updates the switch, via the data-plane driver, to perform dynamic refinement.

Core. The core has two modules: (1) the query planner and (2) the runtime. Upon initialization or re-training, the runtime polls the data-plane driver over a network socket to determine which dataflow operators the switch is capable of executing, as well as the values of the data-plane constraints (i.e., M, A, B, S). It then passes these values to the query planner which uses Gurobi [29] to solve the query planning ILP offline and to generate partitioned, refined queries. The runtime then sends partitioned and refined queries to the data-plane and streaming drivers. It also configures the emitter by specifying the fields to extract from each packet for each query, and each query is identified by a corresponding query identifier (`qid`). When the switch begins processing packets, the runtime receives query output from the stream processor at the end of every window. It then sends updates to the data-plane driver, which in turn updates table entries in the switch according to the dynamic refinement plan. When it detects too many hash collisions, the runtime triggers the query planner to re-run the ILP with new data.

Drivers. Data-plane and streaming drivers compile the queries from the runtime to target-specific code that can run on the switch and stream processor respectively. The data-plane drivers also interact with the switch to execute commands on behalf of the runtime, such as updating `filter` tables for iterative refinement at the end of every window. The Sonata implementation currently has drivers for two PISA switches: the BMV2 P4 software switch [75], which is the standard behavioral model for evaluating P4 code; and the Barefoot Wedge 100B-65X (Tofino) [74] which is a 6.5 Tbps hardware switch. The data-plane driver communicates with these switches using a Thrift API [3]. The current implementation also has a driver for the Apache Spark [73] streaming target for processing packet tuples in user-space and reporting the output of each query to Sonata's runtime.

#	Query	Lines of Code		
		Sonata	*P4*	*Spark*
1	Newly opened TCP Conns. [85]	6	367	4
2	SSH Brute Force [36]	7	561	14
3	Superspreader [83]	6	473	10
4	Port Scan [39]	6	714	8
5	DDoS [83]	9	691	8
6	TCP SYN Flood [85]	17	870	10
7	TCP Incomplete Flows [85]	12	633	4
8	Slowloris Attacks [85]	13	1,168	15
9	DNS Tunneling [10]	11	570	12
10	Zorro Attack [54]	13	561	14
11	DNS Reflection Attack [42]	14	773	12

Table 2.4: Implemented Sonata Queries. We report lines of code considering the same: (1) refinement plan; (2) partitioning plan, i.e., executing as many dataflow operators in the switch as possible.

Emitter. The emitter consumes raw packets sent to the data-plane monitoring port, parses the query-specific fields in the packet, and sends the corresponding tuples to the stream processor. The emitter uses Scapy [76] to extract the unique query identifier (`qid`) from packets. It uses this identifier to determine how to parse the remainder of the query-specific fields embedded in the packet based on the configuration provided by the runtime. As discussed in Section 2.3.3, the emitter immediately sends the output of stateless operators to the stream processor, but it stores the output of stateful operators in a local key-value data store. At the end of each window interval, it reads the aggregated value for each key in the local data store from the data-plane registers before sending the output tuples to the stream processor.

2.6 Evaluation

In this section, we first demonstrate that Sonata is expressive (Table 2.4). We then use real-world packet traces to show that it reduces the workload on the stream processor by

3–7 orders of magnitude (Figure 2.10) and that these results are robust to various switch resource constraints (Figure 2.11).

2.6.1 Setup

Telemetry applications. To demonstrate the expressiveness of Sonata's query interface, we implemented eleven different telemetry tasks, as shown in Table 2.4. We show how Sonata makes it easier to express queries for complex telemetry tasks by comparing the lines of code needed to express those tasks. For each query, Sonata required far fewer lines of code to express the same task than the code for the switch [11] and streaming [73] targets combined. Not only does Sonata reduce the lines of code, but also the queries expressed with Sonata are platform-agnostic and could execute unmodified with a different choice of hardware switch or stream processor, e.g., Apache Flink.

Packet traces. We use CAIDA's anonymized and unsampled packet traces [64], which were captured from a large ISP's backbone link between Seattle and Chicago. We evaluate over a subset of this data containing 600 million packets and transferring about 360 GB of data over 10 minutes. This data contains no layer-2 headers or packet payloads, and the layer-3 headers were anonymized with a prefix-preserving algorithm [24].

Query planning. For query planning, we consider a maximum of eight refinement levels for all queries (i.e., $R = \{4, 8, ..., 32\}$) because additional levels offered only marginal improvements. We replay the packet traces at $20x$ speed to evaluate Sonata on a simulated 100 Gbps workload (i.e., about 20 million packets per second) that might be experienced at a border switch in a large network. We use a time window (W) of three seconds. In general, selecting a shorter time interval is desirable; however, for very short time intervals the overhead of updating the filter rules in the data plane at the end of each window can introduce significant errors. Our choice of three seconds strikes a balance between achieving

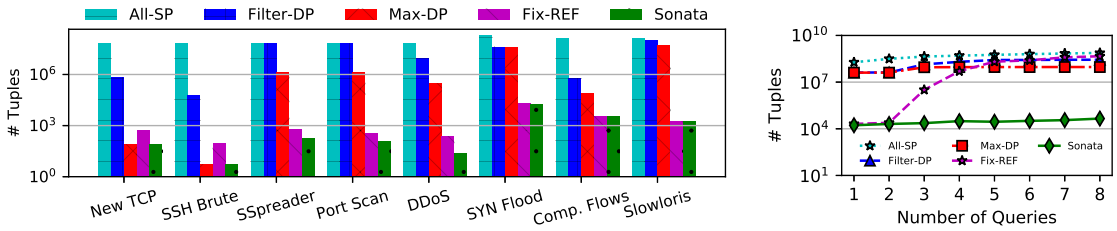
Query Plan	Description	Telemetry Systems
All-SP	Mirror all incoming packets to the stream processor	Gigascope[20], OpenSOC[62], NetQRE[85]
Filter-DP	Apply only filter operations on the switch	EverFlow[87]
Max-DP	Execute as many dataflow operations as possible on the switch	Univmon[44], OpenSketch[83]
Fix-REF	Iteratively zoom-in one refinement level at a time	DREAM[48]

Table 2.5: Telemetry systems emulated for evaluation.

a tolerable detection delay and minimizing the errors introduced by the data-plane update overhead. Sonata's query planner processed around 60 million packets for each time interval to estimate the number of packet tuples (N) and the register sizes (B). We observed that while the ILP solver was able to find near-optimal query plans in 10-20 minutes, it took the solver typically several hours to determine the optimal plans. Since running the ILP solver for longer durations had diminishing returns, we selected a time limit of 20 minutes for the ILP solver to report the best (possibly suboptimal) solution found in that period.

Targets. Since switches have fixed resource constraints, we choose to evaluate Sonata's performance with simulated PISA switches. This approach allows us to parameterize the various resource constraints and to evaluate Sonata's performance over a variety of potential PISA switches. Unless otherwise specified, we present results for a simulated PISA switch with sixteen stages ($S = 16$), eight stateful operators per stage ($A = 8$), and eight Mb of register memory per stage ($B = 8$ Mb). Within each stage, a single stateful operator can use up to four Mb.

Comparisons to existing systems. We compare Sonata's performance to that of four alternative query plans. Each plan is representative of groups of existing systems, such as Gigascope [20], OpenSOC [63], EverFlow [87], OpenSketch [83], and DREAM [48], as



(a) Single-query performance

(b) Multi-query performance

Figure 2.10: Reduction in workload on the stream processor running: (a) one query at a time, (b) concurrently running multiple queries.

shown in Table 2.5. Rather than instrumenting each of these systems, we emulate them by modifying the constraints on Sonata's query-planning ILP. For example to emulate the *Fix-REF* plan, we add the constraint $\forall q, r : I_{q,r} = 1$.

2.6.2 Load on the Stream Processor

We perform a trace-driven analysis to quantify how much Sonata reduces the workload on the stream processor. To enable comparisons with prior work, we evaluate the top eight queries from Table 2.4; these queries process only layer 3 and 4 header fields. *Fix-REF* queries use all eight refinement levels, while Sonata may select a subset of all eight levels in its query plans.

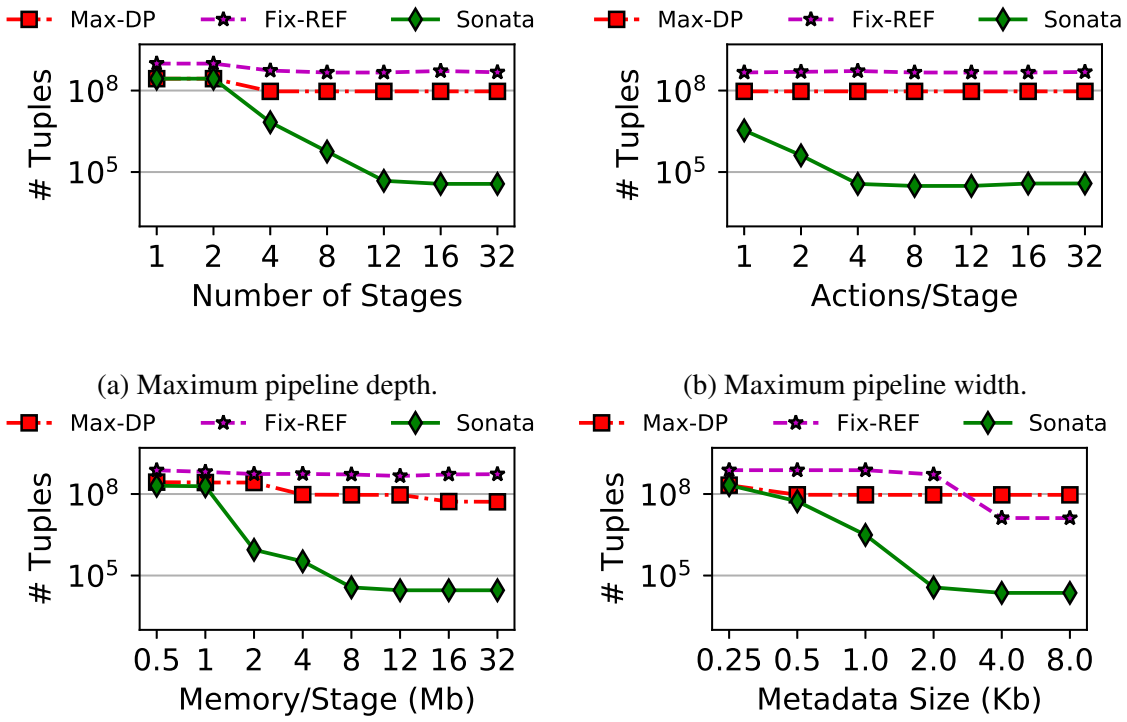
Single query performance. Figure 2.10a shows that Sonata reduces the workload on the stream processor by as much as seven orders of magnitude. *Filter-DP* is efficient for the SSH brute-force attack query, because this query examines such a small fraction of the traffic. *Filter-DP*'s performance is similar to *All-SP* for queries that must process a larger fraction of traffic, such as detecting Superspreaders [83]. For some queries, such as the SSH brute-force attack, *Max-DP* matches Sonata's performance. In many other cases, large amounts of traffic are sent to the stream processor due to a lack of resources. For example, the Superspreader query exhausts stateful processing resources. *Fix-REF*'s performance

matches Sonata's for most cases but uses up to seven additional windows to detect traffic that satisfies the query.

Multi-query performance. Figure 2.10b shows how the workload on the stream processor increases with the number of queries. When executing eight queries concurrently, Sonata reduces the workload by three orders of magnitude compared to other query plans. These gains come at the cost of up to three additional time windows to detect traffic that satisfies the query. The performance of *Fix-REF* degrades the most because the available switch resources, such as metadata and stages, are quickly exhausted when supporting a fixed refinement plan for several queries. We have also considered query plans with fewer refinement levels for *Fix-REF* and observed similar trends. For example, when considering just two refinement levels (`dIP/16` and `dIP/32`) for all eight queries, we observed that the load on the stream processor was two orders of magnitude greater than Sonata.

As the number of queries increases, the number of tuples will continue to increase and eventually be similar to *All-SP*. Although Sonata makes the best use of limited resources for a given target, its performance gains are bounded by the available switch resources. It is important to differentiate the limitations on Sonata's performance from the limitations imposed by existing hardware switches. While today's commodity hardware switches can support tens of network monitoring applications, we envision that the next-generation of hardware switches will enable Sonata to support hundreds of queries, if not more.

Effect of switch constraints. We study how switch constraints affect Sonata's ability to reduce the load on the stream processor. To quantify this relationship, we vary one switch constraint at a time for the simulated PISA switch. Figure 2.11a shows how the workload on the stream processor decreases as the number of stages increases. More stages allow Sonata to consider more levels for dynamic refinement. Additional stages slightly improve the performance of *Fix-REF* as it can now support stateful operations for the queries at finer refinement levels on the switch. We observe similar trends as the number of stateful



(a) Maximum pipeline depth.

(b) Maximum pipeline width.

(c) Maximum bits per stage.

(d) Metadata Size.

Figure 2.11: Effect of switch constraints.

actions per stage (Figure 2.11b), memory per stage (Figure 2.11c), and total metadata size (Figure 2.11d) increase. As expected, *Max-DP* slightly reduces the load on the stream processor when more memory per stage is available for stateful operations. Increasing the total metadata size also allows *Fix-REF* to execute more queries in the switch—reducing the load on the stream processor.

Overhead of dynamic refinement. When running all eight queries concurrently, as many as 200 `filter` table entries are updated after each time window during dynamic refinement. Micro-benchmarking experiments with the Tofino switch [74] show that updating 200 table entries takes about 127 ms, and resetting registers takes about 4 ms. The total update time took 131 ms which is about 5% of the specified window interval ($W = 3$s).

2.7 Related Work

Network telemetry. Existing telemetry systems that process all packets at the stream processor such as Chimera [10], Gigascope [20], OpenSOC [62], and NetQRE [85] can express a wide-range of queries but can only support lower packet rates because the stream processor ultimately processes all results. These systems also require deploying and configuring a collection infrastructure to capture packets from the data plane for analysis, incurring significant bandwidth overhead. These systems can benefit from horizontally scalable stream processors such as Spark Streaming [86] and Flink [59], but they also face scaling limitations due to packet parsing and cluster coordination [63].

Everflow [87], UnivMon [44], OpenSketch [83], and Marple [53] rely on programmable switches to execute queries entirely in the data plane. These systems can process queries at line rate but can only support queries that can be implemented on switches. Trumpet [50] and Pathdump [72] offload query processing to end-hosts (VMs in data center networks) but not to switches. Gupta et al. [27] proposed a telemetry system that can coordinate queries across a stream processor and switch, but the work considered only switches with fixed-function chipsets for single queries and required network operators to explicitly specify the refinement and partitioning plans. In contrast, Sonata supports programmable switches and employs a sophisticated query planner to automatically partition and refine multiple queries. We also quantify the performance gains and overhead with realistic packet traces and a programmable hardware switch.

Query planning. Database research has explored query planning and optimization extensively [57, 51, 5]. Gigascope performs query partitioning to minimize the data transfer from the capture card to the stream processor [20]. Sensor networks have explored the query partitioning problems that are similar to those that Sonata faces [57, 51, 5, 45, 46, 71]. However, these systems face different optimization problems because they typically involve lower traffic rates and involve special-purpose queries. Path Queries [52] and SNAP [4]

facilitate network-wide queries that execute across multiple switches; in contrast, Sonata currently only compiles queries to a single switch, but it addresses a complementary set of problems, such as unifying data-plane and stream processing platforms to support richer queries and partitioning sets of queries across a data-plane switch and a stream processor.

Query-driven dynamic refinement. Autofocus [22], ProgME [84], and DREAM [48], SCREAM [49], MULTOPS [26], and HHH [38] all iteratively zoom in on traffic of interest. These systems either do not apply to streaming data (e.g., ProgME requires multiple passes over the data [84]) they use a static refinement plan for all queries (e.g., HHH zooms in one bit at a time), or they do not satisfy general queries on network traffic (e.g., MULTOPS is specifically designed for bandwidth attack detection). These approaches all rely on general-purpose CPUs to process the data-plane output, but none of them permit additional parsing, joining, or aggregation at the stream processor, as Sonata does.

Chapter 3

Herd: Network-Wide, Continuous Telemetry

This chapter introduces the design, implementation, and evaluation of Herd, a system that allows operators to perform continuous, network-wide telemetry for a subset of queries that can be expressed with Sonata's query interface. While Sonata supports a wide-range of queries partitioned across a stream processor and a single switch, Herd enables executing queries that seek to monitor a global threshold distributed over several switches. While many queries (see Table 2.4) look for traffic of interest that exceeds a threshold, one specific example of this kind of telemetry query detects network-wide heavy-hitters. Herd combines an extension of existing theoretical work with a memory-efficient data structure to provide network-wide telemetry at a bandwidth cost that does not grow in proportion to the number of switches in the network.

3.1 Motivation and Overview

To effectively manage their networks, operators continuously monitor their traffic to detect attacks, performance bottlenecks, and failures—i.e., they continuously perform network telemetry. In many of these telemetry tasks, the operators seek to detect *heavy-hitters*

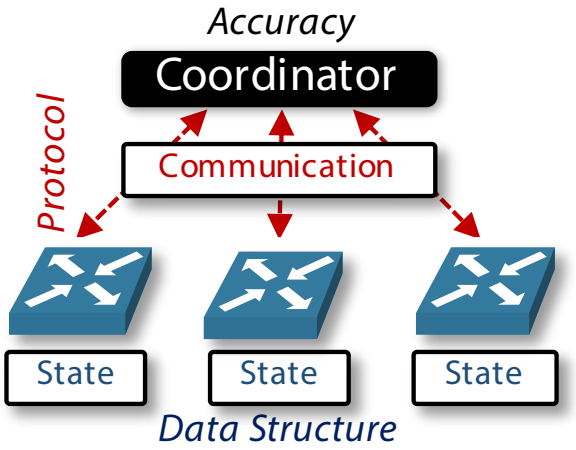


Figure 3.1: Herd Architecture. The coordinator aggregates the partial information observed at each switch to identify network-wide elephants.

by separating the *elephant* flows from the *mouse* flows. Elephants are the relatively few flows that significantly contribute to the overall traffic volume and mouse flows are the more numerous but smaller flows. To distinguish flows into these two categories, network operators must choose between measuring the flows in the network devices themselves, which have both limited memory and computational capacity, or sending a subset of traffic to general purpose CPUs for analysis. In the latter case, the operator is limited both in how much traffic can be sent across the network but also by the fact that general-purpose CPUs process data far slower than line rate. Existing solutions, such as NetFlow/sFlow [15, 56], seek to reduce the data required to detect heavy hitters using general-purpose CPUs by sampling packets at coarse-grained time scales. Other solutions [83, 44] use streaming algorithms and compact data structures, such as the count-min sketch [19], to detect heavy hitters directly on the resource-constrained network devices.

These existing techniques provide mechanisms to perform heavy-hitter detection from a single device with limited resources; however, they do not help us detect network-wide heavy hitters. Some flows generate a large volume of traffic for the network in total but are not heavy at any single ingress point [1]. For example, if a host inside the network is

[1]Here, we relax the definition of a flow to something more coarse-grained than a five-tuple, such as, source/destination IP address pair

the victim of a denial-of-service attack, and monitoring is performed at the network ingress switches, it is possible that each ingress switch will only observe moderate amounts of traffic to the victim host, yet aggregating the analysis over all switches will indicate that a high volume attack is occurring. Furthermore, when attacks of this magnitude begin to converge at a network choke point or at the victim, network devices can become unresponsive which may prevent further measurement from being performed and stymie root-cause analysis. Therefore, it is necessary and proper to place the monitoring upstream of convergence, at multiple locations which are able to handle fractions of the overall attack. Worse yet, existing techniques only provide reasonable accuracy for large timescales, leaving temporal "blind spots" which allow short-lived network conditions, e.g., TCP incast [14] or microbursts, to go undetected.

Detecting network-wide heavy-hitters reduces to a distributed monitoring problem among edge switches, i.e., the entry points of traffic into the network, and a centralized coordinator. The coordinator aggregates the partial information observed at each switch to identify flows whose aggregate count exceeds a global threshold. Deciding when a switch should report to the coordinator and what the switch should report determines how much communication is required and, ultimately, the accuracy of the results. For example, a strawman solution might "report" every packet header of a flow to the coordinator; such a solution would be highly accurate but could not scale to high load or large network size. One could reduce the data sent to the coordinator by instead sampling every n packets, but packet sampling is less accurate over short timescales or at low data rates [87]. Other solutions that aggregate information about many flows in a compact data structure and send the entire structure [43] to a central collector are inherently coupled to a fixed monitoring interval, and the shorter this interval is, the communication overhead will increase. The Continuous Distributed Monitoring (CDM) model provides a communication-efficient alternative for reporting local conditions, as-needed, to continuously track the heavy hitters without respect to a fixed interval. However, this model assumes the ability to store per-

flow state, which has hindered its adoption in real systems [17]. While this technique has provable accuracy bounds, we adapt the base algorithm to reflect the realities of modern networks and implement it in modern commodity switches.

We present Herd (depicted in Figure 3.1), a practical monitoring system for detecting network-wide heavy hitters in real time, with high accuracy, and under communication and state constraints. We extend the standard taxonomy of mice and elephant flows to more accurately describe the costs of performing heavy-hitter detection in a network-wide setting. Herd instructs each switch to probabilistically identify and report flows from this new taxonomy while accounting for the locality of flows to minimize communication. Our solution extends probabilistic reporting techniques presented in [81] and combines them with the sample-and-hold algorithm [23], to report measurements of individual flows in real time. Herd tunes the system parameters using representative traffic observed by the network to achieve the best accuracy possible within the available memory and bandwidth. We summarize our contributions as follows:

Communication-efficient coordination. We developed a new coordination protocol for detecting network-wide heavy hitters that uses adaptive thresholds to account for flow locality. This protocol probabilistically reports when switches observe a non-trivial contribution from a monitored flow and infers network-wide heavy hitters at the coordinator from these reports. Our analysis shows that this protocol reduces the communication cost by 17% for achieving 97% accuracy compared to sampling.

Memory-efficient switch data structure. We developed a data structure that efficiently stores locality parameters and counters for flows that show a non-trivial contribution to a network-wide count. This data structure probabilistically determines the subset of flows to monitor at the switch from a larger traffic stream. We demonstrate that this data structure can be implemented in modern programmable switches for line-rate execution. Our

evaluation shows that our solution requires 40% less switch memory at the expense of 3% degradation in detection accuracy when compared to counting all of the flows.

Parameter-tuning algorithm for high accuracy. While our solution consists of well-known algorithmic [18] and data-structure [23] building blocks, combining these building blocks to produce accurate results within resource constraints is challenging. We present an algorithm that relates the parameters of both the protocol and the data structure to the taxonomy of flows to be monitored and how those parameters affect Herd's performance in terms of accuracy, communication, and state. We describe a heuristic for achieving high accuracy under communication and state constraints.

To demonstrate the deployability of Herd, we present the implementation of both the coordination protocol and switch data structure on a Protocol Independent Switch Architecture (PISA) switch [11] in approximately 750 lines of P4 code.

In Section 3.2, we summarize Herd's architecture and describe a new taxonomy of flows for network-wide heavy hitter detection. We present the design of the coordination protocol in Section 3.3, and the switch data structure in Section 3.4. In Section 3.5, we present an algorithm for configuring various system parameters. We present our prototype in Section 3.6, evaluation in Section 3.7, and related work in Section 3.8.

3.2 Herd Architecture

We can reduce detecting network-wide heavy-hitters to a distributed monitoring problem among ingress switches and a centralized coordinator as shown in Figure 3.1. The coordinator aggregates the partial information observed at each switch to identify flows whose aggregate count exceeds a threshold. By counting locally at each switch and periodically reporting to the coordinator, we can reduce the communication cost, but the memory at switches is limited which also affects accuracy. Here we define accuracy in terms of both precision and recall to quantify false positives and false negatives in the results, respec-

tively. In this section, we first describe the taxonomy of flows in the network that affects how much state and communication is required to perform network-wide heavy-hitter detection and then we describe the mechanisms we use to distinguish flows in that taxonomy.

3.2.1 Who's Who in the Zoo

Classifying flows simply as mice or elephants at a single switch alone is insufficient for designing a system to detect network-wide heavy hitters within communication and state constraints. What might be classified as an elephant on one switch, might be classified as a mouse on another. We need a taxonomy that allows us to classify flows both *locally* and *globally*. Additionally, we need to be able to relate the sizes of these classes to the amount of memory and communication required to perform the network-wide detection.

Local moles and mules. Global elephants. We extend the traditional taxonomy of flows by introducing two new classes: *moles* and *mules* (see Figure 3.2). At each switch, a large number of flows will have no local or global significance which is the traditional class of mice; we seek to allocate no scarce resources for these flows. However, a smaller set of flows will have some significance locally, but a switch will not know if these flows matter globally. Switches will have to maintain state for these flows to determine whether or not they *might* have global significance; we call these flows moles. However, when a mole reaches a local threshold that could significantly impact a global count, a switch is obligated to inform the central coordinator; we call these flows mules.

Mules are, inherently, tracked both locally and globally. A single switch that determines a flow is a mule and reports it to the central coordinator, which then tracks the flow as a mule globally. However, one switch may observe a mule flow that another does not. Based on the reports sent by the switches for each of their mule flows, the central coordinator determines when a mule flow has become a network-wide elephant.

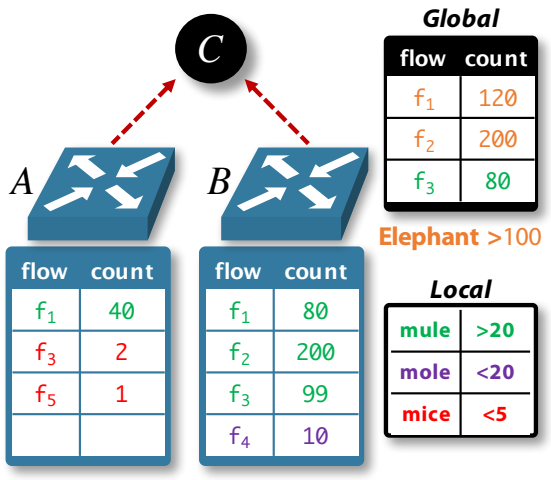


Figure 3.2: Example zoo. Switches A and B communicate with a coordinator C to determine the network-wide heavy hitters.

Who's in the zoo determines resource allocation. Naïvely maintaining state for all flows at the switch scales poorly for large networks and could require keeping counters for tens of millions of flows. For such large networks, the memory required ($O(GB)$) far exceeds the memory available in state-of-the-art programmable switches ($O(MB)$) [74]. This extended taxonomy allows us to better reason about the resources required to perform network-wide heavy-hitter detection. With a known upper bound on the state per switch, we focus our effort on ensuring that the number of moles at each switch does not exceed the upper bound. Similarly, if we know the allowed communication rate for each switch, we must ensure that the number of mules and their reporting frequency does not send too many reports to the coordinator. In the next section, we discuss the mechanisms Herd uses to distinguish among these classes of flows.

Example. Figure 3.2 shows a simple example where two switches (A and B) communicate with a central coordinator (C) to determine the network-wide elephants. In this example, we use thresholds of 5 and 20 to distinguish mole from mouse flows and mule from mole flows, respectively; we set the threshold for network-wide elephants at 100. The tables below each switch show the actual counts observed for flows f_1-f_5 at each switch.

At switch A, we avoid maintaining state for mouse flows f_3 and f_5, but store the counts for the local mule flow f_1. At switch B, we store counters for f_1-f_4, but we do not report f_4 to the coordinator because it is only a local mole. The coordinator is aware of all mules (f_1-f_3) from both switches, but determines that only f_1 and f_2 are global, network-wide elephants. In the case of f_3, notice how both switches and the coordinator all have *different* views of the total count for this flow. Since f_3 is a mouse at switch A, the switch actually has no information about the flow's count at all. At switch B, flow f_3 is a mule locally, but since the switch reports to the coordinator only once every 20 counts (the mule threshold), the coordinator believes the global count of f_3 is only 80. In fact, the global count of f_3 meets the network-wide threshold of 100, but our taxonomy of flows and their reporting requirements would not identify f_3 as a network-wide elephant; this is by design.

3.2.2 Probabilistic Counting and Reporting

Based on the above taxonomy, we must distinguish mice from moles and moles from mules. Once differentiated, we must also determine how frequently to update the coordinator with local information about the mules. In this section, we describe the techniques Herd uses for doing so.

Distinguishing moles from mice. Many existing techniques for storing flow counters with small state focus on accurately detecting only the local heaviest flows. For example, using a count-min sketch does not *eliminate* storing state for small flows, but it does provide bounds on the error incurred by doing so. Using a count-min sketch would both violate our goal of maintaining no state for mice, but we would also need a very large sketch to overcome the error incurred by storing the small but numerous mouse flows. Similarly, the space-saving algorithm [47] works well for storing local elephants, but that algorithm would allow local mice to evict moles from the data structure, which would lead to inaccurate results.

To avoid maintaining state for small flows, we use a data structure that relies on the sample-and-hold technique [23] to pick the flows that are moles. In this technique, the switch checks whether each incoming packet belongs to the set of moles. If so, it updates the counter; otherwise, the switch chooses to start counting the flow with some sampling probability (s). Effectively, this approach defines the set of moles as those whose count is greater than $(1/s)$, in expectation. We show how to choose s such that it reduces the memory footprint without compromising the detection accuracy in Section 3.4.1.

Distinguishing mules from moles. Only a subset of the mole flows sampled will ever become large enough to impact the global count for a given flow. We set a local threshold (τ) for local flow counts such that $1/s < \tau$. This ensures that the set of mule flows is strictly smaller than the set of mole flows. When a local mole flow's count reaches τ, the switch promotes the flow to a mule locally and reports to the central coordinator.

Reporting Mules. A network-wide, elephant flow might exceed this local threshold at multiple switches. Requiring all switches to send reports each time (τ) packets are observed for all mule flows to the coordinator would limit our system's ability to support large networks with many ingress switches. Rather than sending a report for each mule every time τ packets are observed, we build on the theoretical approach first described in [81] and report to the coordinator with probability (r) each time (τ) packets are observed for a mule. The coordinator identifies a mule as a network-wide heavy hitter if it receives R reports for this flow from any of the switches. We could choose to report very frequently (e.g., $r = 1$) for high accuracy, or we could choose a lower value of r to lower the coordination overhead. In Section 3.3.1, we show how to select values for τ, r, and R that strike a balance between detection accuracy and communication cost.

3.3 Coordination Protocol

The coordination protocol must allow the edge switches to efficiently communicate to the coordinator when they have observed counts that could be significant network-wide. The protocol must therefore determine *which* flows are mule flows and *when* to report them to the coordinator. In this section, we describe a protocol that sets the threshold for each mole flow to become a mule, and then uses the reports about the mules to determine the network-wide elephant flows. We then describe an extension to this protocol that leverages the spatial-locality of network traffic to reduce the communication cost of the protocol.

3.3.1 When to Report Which Flows

Separating the Mules from Moles

Because heavy hitters represent a tiny fraction of the total number of flows, reporting counters for all flows to the coordinator is wasteful. To reduce the communication cost, we design the coordination protocol such that switches can locally differentiate between moles and mule flows, and only report mule flows to the coordinator. In distinguishing mule from mole flows, the switch determines when the local contribution from a mole flow could significantly impact a global count—answering *which* flows to report. The switch can perform this discrimination by comparing the count of a mole to a local threshold (τ) set by the coordinator. Once a mule flow is identified, the switch reports to the coordinator, each time a *bundle* of τ packets is observed at the switch—answering *when* to report.

Scaling to Large Networks

However, the downside of the above technique is that it will not scale as the number of switches in the network grows. Because a switch only reports a flow once for every τ packets it observes, it will often have residual flow counts smaller than τ which have not

Algorithm 1: Switch Algorithm

Input: Local Threshold (τ), Report Probability (r)
Func `ProcessPacket` (*pkt*) **:**
    $f \leftarrow$ `ExtractFlow` (*pkt*)
    exceeds $\leftarrow$ `UpdateAndCheck` (*f,D*)
    if *exceeds*
        if `Flip` (r)
            `Report` (f)
        $D[f] \leftarrow 0$

yet been reported. In aggregate, these residual counts represent a "blind spot" for the coordinator and necessarily cause inaccuracies in the global count it maintains.

As τ increases or the number of switches grows, the inaccuracy of the final results will increase. One possible way to reduce the inaccuracy is to significantly lower τ. However, that would significantly increase communication, since the switch will produce many more reports for each mule flow. Prior work [18] proposed a probabilistic reporting approach that scales with the number of switches in the network and proved its efficiency. However, implementing this technique has proven to be challenging, and has yet to be implemented [17]. We adapt this technique to account for flow locality and enable execution on modern programmable switches.

Probabilistically Separating Elephants and Mules

Our algorithm for probabilistically reporting mule flows to the coordinator is described in Algorithm 1. The function `ProcessPacket` processes every packet received by the switch and `ExtractFlow` extracts from the packet the fields that identify flow f. The function `UpdateAndCheck` updates the counter for this flow and compares it with a local threshold (τ). In Section 3.4, we describe this function in more detail. If the updated count exceeds τ, then the switch reports the flow to the coordinator with probability r. Here D is just a simple key-value store and a single r and τ apply to all flows. By reporting with probability r, each bundle reported now represents a count of τ/r in expectation,

Algorithm 2: Coordinator Algorithm

Input: Reporting Threshold (R)
Output: Heavy Hitter Set (H)
Func `HandleReport` (f) **:**
    $Reports_f \leftarrow Reports_f + 1$
    if $Reports_f \geq R$
        $H \leftarrow H \cup \{f\}$

which reduces the total number of reports that must be sent. The coordinator executes Algorithm 2; after receiving a report for flow f, if the number of reports received for f exceeds threshold R, the coordinator determines that this mule flow is now an elephant.

Configuring Parameters

Configuring the parameters (τ, r, and R) to strike a balance between accuracy and communication cost is non-trivial. For example, we want to set τ high enough such that it can effectively differentiate between the mule and mole flows without affecting the detection accuracy, but low enough that it does not increase the number of flows classified as mules by the switch and, consequently, the communication cost, too significantly. Previous work [18] demonstrated tight bounds on communication and error by selecting specific values of $r = 1/k$ (k is the number of switches) and τ by introducing an approximation factor (ε). Their results show that this approach can achieve high accuracy with modest communication overhead that does not grow proportionally to the number of switches in the network. We generalize the results from prior work by setting $r = 1/k$, $\tau = \varepsilon T/k$, for $0 < \varepsilon < 1$; the coordinator then determines that a flow is an elephant after receiving $R = kr/\varepsilon$ reports. When $r = 1/k$ this threshold simplifies to $R = 1/\varepsilon$, but in Section 3.5 we describe how we might vary the value of r when tuning all of the Herd's parameters together.

Example. Let us consider an example topology with $k = 10$ switches, global threshold $T = 2500$ and we choose an approximation factor of $\varepsilon = 0.1$. In this case, switches would

report every $\tau = 25$ packets to the coordinator with probability $r = 0.1$. The coordinator would therefore declare any mule an elephant after receiving $R = 10$ reports. At this point, a single τ and r apply to all flows at all switches in the network.

3.3.2 Locality-aware Reporting Parameters

The protocol we described in the previous section implicitly assumes that all of the k switches in the network are equally likely to observe a portion of the traffic for a given flow. This assumption results in lower local thresholds (τ) as networks grow large. A smaller τ will result in the switch determining that more mole flows are mules, and, ultimately, increase the communication cost. However, in practice, most flows exhibit spatial locality, i.e., only a subset of edge switches observe traffic for a given flow. If a flow is only observed at $l << k$ locations, then we should configure the parameters based on this smaller number of switches. Accounting for this locality would increase τ, which, in turn, ensures that fewer moles are unnecessarily promoted to mules, thus reducing the communication cost.

Configuring Parameters

We now require an additional parameter l_f to account for the spatial locality. Here, l_f denotes the *number* of switches that observe flow f. For now, we can assume that we know the locality parameters for all flows *a priori* because forwarding state can be used to infer this information. Accounting for this locality parameter, we adjust the local threshold as $\tau = \varepsilon T / l_f$ and reporting probability as $r = 1/l_f$ for each flow at the switch. The coordinator reports flow f as a heavy hitter when it receives $R = 1/\varepsilon$ reports from the switches. Returning to Algorithm 1, we must augment the key-value store D to maintain l_f for each flow. The values for τ and r are then calculated based on looking up $D[f].l$.

Algorithm 3: Coordinator Algorithm for Learning l_f

```
Func HandleHello (hello):
    f, s ← ExtractFlow(hello)
    if s ∉ S_f
        S_f ← S_f ∪ s
        if |S_f| ≥ 2 l_f
            l_f ← |S_f|
            Send(x ∈ S_f, l_f)
        else
            Send(s, l_f)
```

Example. Let us return to our example with $k = 10$ switches, global threshold $T = 2500$, and an approximation factor of $\varepsilon = 0.1$. Let us now consider that a particular flow f is observed only at $l_f = 2$ switches in the network. We now can increase both our bundle size to $\tau_f = 125$ and reporting probability to $r_f = 0.5$. The coordinator would still declare a mule an elephant after receiving $R = 10$ reports for the flow, but there are now fewer mules sending reports to the coordinator due to the larger bundle size. Now, the threshold (τ) and reporting probability (r) can vary based on how many switches actually observe the flow.

Tracking Spatial Locality

In reality, the locality of flows in a network changes due to routing updates, misconfiguration, and failure; l_f must be tracked dynamically. Thus, Herd introduces a protocol that independently tracks changes in the spatial locality for flows directly in the data plane. At all times, a switch has knowledge of which flows it expects to observe (more on this in Section 3.4.2). When a switch receives a packet it is not expecting, the switch sends a `Hello` message to the coordinator. As shown in Algorithm 3, the coordinator extracts the flow (f) and switch identifier (s) from the `Hello` message and looks up the value of l_f. It also updates a data structure (S_f) that maintains a mapping of flows to switches. Finally, it sends the updated parameter to all switches in S_f. To avoid updating the parameter due to spurious or transient conditions, we choose to update the locality parameter for a flow only when the set of switches actually observing the flow (S_f) doubles in size.

3.4 Switch Data Structure

The coordination protocol described in the previous section assumed that switches could store a counter and l_f for each flow. Although modern programmable switches enable flexible packet processing directly in the data plane, the amount of memory available for stateful operations is orders of magnitude smaller than what the coordination protocol described earlier requires. In this section, we again exploit the observations that (1) only a few flows are heavy hitters, and (2) flows exhibit spatial locality to reduce the memory footprint on the switches. We first describe how we avoid maintaining state for the many mice flows, and then how we decouple storing the locality parameter from the flow counters,

3.4.1 Separating Mice from Moles

To reduce the memory footprint, we need a data structure that can avoid consuming resources for local mice flows, which are too small to significantly contribute to a network-wide elephant. This requires designing a data structure that can effectively and efficiently separate mole flows from mouse flows.

Why not just use sketches? In order to separate mice from moles, we could use a count-min sketch to estimate the size of all flows, and then only allocate an exact counter when a flow exceeds some minimum threshold ($\tau' < \tau$). Conventionally, approximate data structures, such as count-min sketches, have been used to monitor heavy hitters with bounded memory and error. [19, 83, 43, 44] Unfortunately, these data structures were designed for tracking single-site *elephants*, and are therefore normally used for identifying flows which take up a large portion of the traffic at a single switch. Using a count-min-sketch to accurately estimate both *mouse* and *mole* flows instead would require much larger sketches to achieve acceptable accuracy.

For example, a count-min sketch that uses b bits per row with r rows and processes N packets will produce an estimate that errs at most $2N/b$ with probability at least $1-(1/2)^r$

Algorithm 4: Sample and Hold Switch Algorithm

```
Func UpdateAndCheck(f, D):
    if f ∈ D
        l_f ← D[f].l
        τ_f ← εT / l_f
        D[f].count ← D[f].count + 1
        if D[f].count ≥ τ_f
            D[f].count ← 0
            return True
    else
        if Flip(s)
            D[f].count ← v
    return False
```

from the true count. Assuming $100M$ packets are processed by the switch in a monitoring interval, setting $b = 10K$ will result in an error of at most $20K$ and we would need to allocate $b = 100K$ to get an error of at most 2,000. For the task of counting small flows, count-min-sketches do not strike the right balance between state and accuracy.

Sample and Hold the Moles. Although mice flows comprise a large portion of the total flows in a network, these flows are few in total packet count. Therefore, we can use sampling to effectively filter out those flows whose count is less than the inverse of the sampling probability, in expectation. For flows that we do sample, however, we store an exact counter so that we can separate mules from moles as described in Section 3.3.1. While this technique will not prevent *all* mouse flows from erroneously being promoted to moles, it does eliminate enough mouse flows to store the sampled mole flows in the limited switch memory.

We now describe in more detail the `UpdateAndCheck` function in Algorithm 4 using a key-value store D of limited size. For each incoming packet belonging to a flow f, the switch first checks if f is currently in the key-value store D. If not, the switch inserts f into D with probability s. If the switch decides to insert the flow, it initializes the count in D to an initial value ($v = \frac{1}{s}$). We discuss how to set sampling rates and initial values in more

src	dst	l
10.0.0.0/8	20.0.0.0/8	3
5.0.0.0/8	6.0.0.0/8	2

flow	count
f_1	10

Figure 3.3: Herd Switch Data Structures for locality. We separate the groups for which we track locality parameters from the flows that we are counting.

detail in Section 3.5. If the flow is in D, the switch first looks up the l_f parameter stored in D to calculate τ_f and r_f. The switch then updates the count and checks to see if it will report this bundle of counts to the coordinator.

We can reduce the memory footprint of D by using a low sampling probability. However, we want a sampling probability high enough such that when sampled, the initial value does not exceed (τ) automatically promoting all moles to mules. By selecting a sampling probability greater than $\frac{1}{\tau}$ ensures that moles will not automatically be promoted to mules.

3.4.2 Locality-aware Data Structure

As we discussed in Section 3.3.2, a locality-aware coordination protocol ensures that switches can use a higher reporting threshold τ_f based on the locality parameter l_f. However, the forces that affect flow locality (e.g., Internet routing) can operate at a granularity independent of the granularity at which we may want to monitor flows. We must account for this disparity in the locality-aware data structure.

Storing Parameters at the Granularity of Locality

So far, we have assumed that switches can calculate per-flow parameters such as reporting probability (r_f), and reporting threshold (τ_f) based on the locality parameter l_f. The overhead of maintaining these parameters at the flow-level of granularity could outweigh the benefits of locality awareness both in terms of communication and memory costs. Fortunately, the granularity at which flows exhibit spatial locality is much coarser than that

Algorithm 5: Switch Algorithm

Input: See Table 3.1

Func `ProcessPacket` (*pkt*) **:**

$f \leftarrow$ `ExtractFlow` (*pkt*)

$l_g \leftarrow$ `GetLFromGroup` (f)

$\tau_g, r_g \leftarrow \tau = \varepsilon T / l_g, \frac{1}{l_g}$

exceeds $\leftarrow$ `UpdateAndCheck` (f,D,τ_g)

if *exceeds*

if `Flip` (r_g)

`Report` (f)

required for monitoring. For example, forwarding decisions are usually made at the granularity of source and/or destination IP prefixes, affecting where flows will display locality. On the other hand, network operators might be interested in detecting heavy hitters at the five-tuple or source-destination pair address granularity. We will now show how we leverage this observation to reduce the memory footprint and communication overhead for maintaining locality-aware parameters.

To leverage this observation, we define a group ($g_{src,dst}$) based on source-destination IP prefix pairs, such that $g_{src,dst} = \{f | f.srcIP \in src, f.dstIP \in dst\}$. As shown in Figure 3.3, rather than maintaining and updating the locality parameter on a per-flow basis, a switch maintains the locality parameter based on the group that displays this locality. We store a group (g) at a switch if at least one flow $f \in g_{src,dst}$ is observed at the switch. Algorithm 1 is modified so that the switch extracts the locality parameter l_g for a flow f based on the group to which the flow belongs. The switch then calculates the parameters τ_g and r_g and supplies τ_g as an additional parameter to `UpdateAndCheck`. The updated Algorithm 5 shows the small changes needed from the original switch algorithm. Algorithm 3 is also modified slightly such that all variables indexed by f are now indexed by g.

3.5 Tuning System Parameters

So far, we have discussed how we designed the coordination protocol and switch data structure to achieve high accuracy with limited communication and memory costs. However, when configuring the parameters for each (e.g., sampling and reporting probability, local threshold, etc.) in isolation, we may actually choose values for these parameters that worsen performance under resource constraints. Given the operational constraints on switch memory S and the communication overhead C, we now describe an algorithm for determining the optimal parameter configuration such that the system achieves high detection accuracy within the constraints. In this section, we use representative packet traces to first set the sampling rate based on the switch memory bounds (Section 3.5.1). We then show how to set reporting parameters (Section 3.5.2) based on bandwidth constraints. Finally, we describe a heuristic algorithm that maximizes the detection accuracy for a given bandwidth and memory budget (Section 3.5.3) by choosing "good" values of parameters based on fundamental bounds (Section 3.5.4). In this section, we assume a single flow group for the sake of clarity since the same process can be applied to multiple flow groups.

3.5.1 Sampling Based on State Constraints

If we had unlimited memory in the switch, we could maintain exact counters for all of the flows by setting the sampling probability to $s = 1$. However, the available memory is finite, and Herd needs to account for this limitation. Given the operational constraints on a switch's memory is S, our goal is to determine the highest sampling probability s that satisfies this constraint given the workload.

If P denotes the number of packets observed by the switch, then we expect the memory usage to be (sP). Therefore, the maximum sampling probability the switch data structure could support is $\frac{S}{P}$ for a given bound on switch memory S. However, if the flow size distribution is known a priori, we can select a higher sampling probability, based on the number

Symbol	Meaning
	Given
T	Global threshold
C	Communication budget per switch
S	Memory budget per switch (# counters)
k	Total number of ingress switches
l	Number of switches which observe a flow
D	Training Data
	Determine
ε	Approximation Factor
τ	Local (Mule) threshold
M	Set of moles observed at switch
U	Set of mules at a switch
r	Reporting probability to coordinator
s	Sampling probability at a switch

Table 3.1: Network-Wide Heavy Hitter Parameters

of moles that the switch can maintain. The `GetSampling` function in Algorithm 6 shows how we use the given data (D) to empirically determine the highest possible sampling probability, given the memory constraint S. Let M denote the set of mole flows observed at the switch. The `CalculateMoles` function is used to calculate M, given the workload D and sampling probability s; the function iteratively searches for the largest set M that the switch can support.

Since we want to ensure that the actual set of moles sampled at the switch contains true mules, we must carefully choose the sampling probability. Since mule flows have true counts strictly greater than mole flows, then we must ensure that we sample with probability greater than $\frac{1}{\tau}$ in order to ensure that the count of the sampled flows is strictly less than the mule threshold τ, in expectation. In summary, the local mule threshold determines the lower bound for sampling probability, and the available switch memory sets its upper bound.

3.5.2 Reporting Based on Communication Constraints

If we had unlimited bandwidth, we could set ε as small as needed to achieve the desired accuracy and incur the resulting communication overhead. However, communication resources are also constrained so we need to adjust the system parameters accordingly. Given the communication bound C, we must configure the reporting probability.

In section 3.3.1, we calculate the reporting probability as $\frac{1}{l}$ to achieve good accuracy and grow to large size networks. However, we must adjust the local threshold (τ), reporting probability (r), and global reporting threshold (R) for high accuracy given communication constraints. The function `DeriveReporting` configures these parameters based on a given value of (ε). A switch sends $\frac{T}{\tau}$ reports to the coordinator for each mule flow when $r = 1$. We denote U to be the set of mule flows observed at a switch. The `CalculateMules` function is used to determine U, given the set of moles and the local threshold as input. Finally, the algorithm uses the set of mule flows and the communication budget to calculate the reporting probability. The total number of reports sent to the coordinator is bound by the total number $\frac{T|U|r}{\tau}$, in expectation. For a given bound on communication overhead C, the upper bound on reporting probability is, therefore, $\frac{C \cdot \tau}{T|U|}$.

3.5.3 Tuning for High Accuracy

Once we have set the sampling rate and given a mechanism for determining the reporting probability, we can now find an optimal local threshold. Since the local threshold can be tuned with the approximation factor ε, we describe an algorithm that searches for an optimal value based on the given parameters of the system. In the `TuneAccuracy` function, the algorithm uses representative packet traces to empirically compute the moles and mules and set the parameters of the system as described above. After calculating all parameters, the algorithm calls the `GetAccuracy` function to determine the accuracy of the System using this parameter configuration. The algorithm then iteratively searches for a value of ε

Algorithm 6: Algorithm for tuning parameters.

Func `GetSampling(`S, D,*mole_tau*`)`**:**
  $s \leftarrow \frac{1}{mole\_tau}$
  $M \leftarrow$ `CalculateMoles(`*D, s*`)`
  `// Section 3.5.1`
  while $|M| < S$ **do**
    $mole\_tau \leftarrow mole\_tau - 1$
    $M \leftarrow$ `CalculateMoles(`*D, s*`)`
  end
  return *mole_tau*

Func `DeriveReporting(`C, ε, l, s`)`**:**
  $\tau \leftarrow \frac{\varepsilon T}{l}$ `// Section 3.5.2`
  $M \leftarrow$ `CalculateMoles(`*D, s*`)`
  $U \leftarrow$ `CalculateMules(`*M,* τ`)`
  $r \leftarrow \frac{C \cdot \tau}{T|U|}$
  $R \leftarrow \frac{l \cdot r}{T}$
  return R, U, r, τ

Func `TuneAccuracy(`T, S, C, D, l`)`**:**
  $A_{max} \leftarrow 0$ `// Section 3.5.3`
  $mole\_tau \leftarrow$ `GetSampling(`S, D, T`)`
  $s \leftarrow \frac{1}{mole\_tau}$
  while $\varepsilon \in [\varepsilon_{max} \ldots \varepsilon_{min}]$ **do**
    $R, U, r, \tau \leftarrow$ `DeriveReporting(`C, ε, l, s`)`
    $A \leftarrow$ `GetAccuracy(`*D, R, T, U, r, s,* τ,`)`
    if $A \geq A_{max}$
      $\varepsilon_{max} \leftarrow \varepsilon$
      $\varepsilon \leftarrow \varepsilon - \sigma$ `// Section 3.5.4`
      $A_{max} \leftarrow A$
    else
      break
  end

that is at least ε_{min} where the accuracy of a succeeding iteration is less than the preceding iteration and then terminates.

3.5.4 Selecting the Right Values of ε

Näively choosing values of ε without regard to the other parameters can actually result in poor system performance. By wisely selecting values of ε, we can both reduce the range

of possible values for ε that Algorithm 6 has to explore and avoid the detrimental effects that can occur when discretizing parameters. For example, although ε is a real number, it is used to calculate integer thresholds for the local switches (τ) and the global number of reports (R). When determining these discrete values based on a continuous calculation, rounding can significantly degrade System performance, which we describe and show later in Section 3.7.2. To reduce this error, we can select values of ε that result in whole integer values for other parameters. First, we begin by asserting that there are maximum (ε_{max}) and minimum values (ε_{min}) that we wish to explore shown here in Theorems 1, 2.

Theorem 1 (ε_{min}) *For all k,l,T, where $k \geq l$, there exists an ε_{min} such that $\tau \geq 1$.*

$$\frac{T\varepsilon}{max(k,l)} = \tau$$

$$\frac{T\varepsilon_{min}}{max(k,l)} \geq 1$$

$$\frac{T\varepsilon_{min}}{k} \geq 1$$

$$\varepsilon_{min} \geq \frac{k}{T}$$

Since we also know that as the local threshold on the switch grows too large, the accuracy of the system degrades. Consequently, we can ignore values of ε that would result in local thresholds larger than (T/k), shown here in Theorem 2.

Theorem 2 (ε_{max}) *For all k,l,T, where $k \geq l$ and $\tau \leq \frac{T}{k}$, there exists an ε_{max}.*

$$\frac{T\varepsilon_{max}}{min(k,l)} \leq \frac{T}{k}$$

$$\frac{\varepsilon_{max}}{l} \leq \frac{1}{k}$$

$$\varepsilon_{max} \leq \frac{l}{k}$$

Intuitively, we also know that, $\varepsilon_{max} \geq \varepsilon_{min}$ and whenever that condition does not hold, we cannot use our algorithm to find the best value of ε. We can use Theorems 1 and 2 to determine in what combination of *compatible parameters* this condition will hold.

Theorem 3 (Parameter Compatibility) *By Theorem 1,2, for all k,l,T, where $k \geq l$, parameters are compatible when $\varepsilon_{max} \geq \varepsilon_{min}$.*

$$\frac{l}{k} \geq \varepsilon_{max} \geq \varepsilon_{min} \geq \frac{k}{T}$$

$$\frac{l}{k} \geq \frac{k}{T}$$

$$1 \geq \frac{k^2}{T \cdot l}$$

$$T \geq \frac{k^2}{l}$$

To minimize quantization error when selecting values of ε, we should seek to ensure that the calculated local threshold is a whole integer without rounding. To achieve this, we select a *quantization step* ($\sigma = \frac{l}{T}$) and set ε as an integer factor of this step to ensure that all values of τ are whole integers.

Theorem 4 (Quantization Factor (σ)) *For all k,l,T,ε,τ, where $\sigma = \frac{l}{T}$ there exists an integer factor (n) that ensures τ is a whole integer.*

$$\varepsilon = \sigma n$$

$$\tau = \frac{T\varepsilon}{l}$$

$$\tau = n\frac{T\sigma}{l}$$

$$\tau = n\frac{T}{l} \cdot \frac{l}{T}$$

$$\tau = n$$

Finally, we can use Theorems 1, 2 and 4 to determine the values of n_{min} and n_{max}.

$$\sigma \cdot n_{min} \geq \varepsilon_{min} \geq \frac{k}{T}$$

$$\frac{l}{T} \cdot n_{min} \geq \frac{k}{T}$$

$$n_{min} \geq \frac{k}{l}$$

$$\sigma \cdot n_{max} \leq \varepsilon_{max} \leq \frac{l}{k}$$

$$\frac{l}{T} \cdot n_{max} \leq \frac{l}{k}$$

$$n_{max} \leq \frac{T}{k}$$

In this section, we have showed how to tune Herd's parameters for best accuracy given resource constraints. We used an offline algorithm to empirically determine the mole and mule flows for a combination of system parameters. The size of these sets determined the sampling probability (s) and reporting probability (r) based on resource constraints. We also analytically showed how to best configure the approximation parameter (ε) to reduce the search space of our algorithm and appropriately discretize system parameters to reduce quantization error.

3.6 P4 Prototype

We now describe our P4 prototype that is subject to the same constraints as described in Section 2.3.1 and implements both the coordination protocol and switch data structure previously described. In this section, we first describe the overall structure of our P4 prototype and then describe the challenges that we faced while implementing our prototype in this architecture.

3.6.1 The Life of a Packet

When a packet enters the PISA pipeline, we first determine to which group ($g_{src,dst}$) the packet belongs. The group identifier is then used to match a rule in a match-action table (Section 3.6.2), in order to determine the local threshold (τ_g), and reporting probability (r_g) for that group. Next, two independent but biased coin flips (Section 3.6.3) are performed; the first coin flip is based on the sampling probability (s) and the second based on the reporting probability (r). Both results are stored as packet metadata values $flip_1$ and $flip_2$, respectively, for use later in the pipeline. Finally, we must check if the flow is stored in the hash tables that implement the key-value store D (Section 3.6.4). If the flow is found, its counter is incremented. If the counter is greater than τ_g, the counter is reset to 0; if $flip_2 == \texttt{true}$ as well, a report is then sent for this flow. If the flow was not found and $flip_1 == \texttt{true}$, then the flow is sampled and stored in the hash table. If no empty space is found in the hash table, then the packets in the flow are sent to the coordinator—trading communication cost for accuracy.

3.6.2 Storing Locality Parameters with Match-Action Tables

So far, our presentation of the switch data structure assumed that the switch could compute the local threshold (τ_g), and reporting probability (r_g) by itself if it knew l_g. However in practice, such computations require floating point arithmetic that is currently not supported in PISA switches. Since both parameters are specific to each locality group, we can store both parameters in a match-action table as shown in Figure 3.3. However, instead of storing l_g itself, we can precompute the values of (τ_g) and (r_g) and store them instead. The function `GetLFromGroup` and the subsequent calculation using l_g in Algorithm 5 is actually implemented as a single lookup in the match-action table where τ_g and r_g are stored. These values are then copied to the packet's metadata field for use in the `UpdateAndCheck` and `Flip` functions. Our implementation intentionally deviates

from Algorithm 5 to demonstrate how seemingly simple algorithms must be altered in order implement them on (today's) PISA switches.

3.6.3 Flipping Coins with Hashes

The `Flip` function, used in Algorithms 2, 4, and 5, requires flipping a biased coin in the switch. A naïve implementation of `Flip` also requires support for floating point arithmetic on the switch. Instead, we represent floating point values ($0 < i \leq 1.0$) as unsigned integers, which is similar to how other works [8] have implemented probabilistic techniques with PISA switches. We then use a combination of a packet's timestamp and other header fields to compute a 32-bit hash value. A `Flip` operation returns `True` if the computed hash is less than $\lceil 2^{32} i \rceil$. This approach introduces a small quantization error since we can only represent probabilities as multiples of $\frac{1}{2^{32}}$.

3.6.4 Key-Value Store with Hash Tables

The `UpdateAndCheck` function (detailed in Algorithm 4) for updating the counters of mole flows requires implementing the key-value data store D. We can implement this data store as a hash-indexed register array within a single stage. However, these hash-indexed arrays will likely encounter collisions in a single stage. To address this problem, we implement D as a multi-stage hash-table. When inserting a value into D, we insert the value in the first hash table that contains no collision. Though this approach ensures that a switch can maintain counters for a large number of mole flows, the limited memory per stage and number of stages ensures that collisions are inevitable as the number of flows grows large.

3.7 Evaluation

In this section, we quantify how Herd makes efficient use of limited communication and state resources to detect network-wide heavy hitters with as high accuracy as possible. We use real-world packet traces to demonstrate how combining probabilistic counting with probabilistic reporting reduces Herd's memory footprint by 38% and bandwidth footprint by 17% to report network-wide heavy hitters with 97% accuracy.

3.7.1 Setup

To quantify Herd's performance, we run a simple network-wide heavy-hitter query to determine which flows (based on the standard five-tuple of source/destination IP address, source/destination port, and transport protocol) send a number of packets greater than a global threshold (T) during a rolling time window (W).

Simulation experiments. For our experiments, we monitor at the edge switches of the network where the number of edge switches (k) is 10 — representative of a wide-area network connecting multiple data centers for cloud providers [35]. For all experiments, each flow shows affinity for two ingress switches, i.e., $l = 2$, based on the source IP address. We choose a global threshold that corresponds to the 99.99^{th} percentile flow count in the packet trace.

Packet traces. To emulate real-world traffic distributions, we used CAIDA's anonymized Internet traces from 2016 [64]. These traces consist of all the traffic traversing a single OC-192 link between Seattle and Chicago within a major ISP's backbone network. Each minute of the trace consists of approximately 64 million packets. For our experiments, we use a time window (W) of five seconds resulting in around 5 million packets per window, which translates to around 270K unique flows per window.

Technique	Prob. Counting	Prob. Reporting	State Required
Strawman	✗	✗	345K
RLA	✗	✓	345K
Sampling	✗	✓	N/A
Herd	✓	✓	211K

Table 3.2: Comparison with other Heavy-Hitter detection techniques. Herd combines both probabilistic counting and reporting where other approaches use only one.

Since the packet traces are collected from a single link only, we associate packets from the trace with a given ingress switch based on a hash of the source IP address. For each source IP address, we assign an affinity for a specific ingress switch with probability p. Packets from a given source IP are, therefore, processed at a "preferred" switch with probability p and at the other switches with probability $(1-p)$. For the case of $l = 2$, this distribution simulates a primary/alternate relationship on ingress for a single source and $p = 0.95$.

3.7.2 Baseline Herd Performance

To demonstrate the benefits of combining probabilistic counting and reporting, we quantify the amount of state and communication overhead for Herd and compare it with the existing heavy-hitter detection techniques that either employ probabilistic counting or reporting, but not both.

Alternative Approaches. First, we consider a strawman solution that makes use of neither probabilistic counting or reporting; each switch maintains counters for every flow (all flows are moles) and reports all the counters to a central coordinator at the end of a window. Second, we consider a solution based on the randomized reporting technique [18]; where the switch still treats all flows as moles, but it probabilistically reports mules to the coordinator with parameters that ignore locality. Finally, we consider a solution based on packet

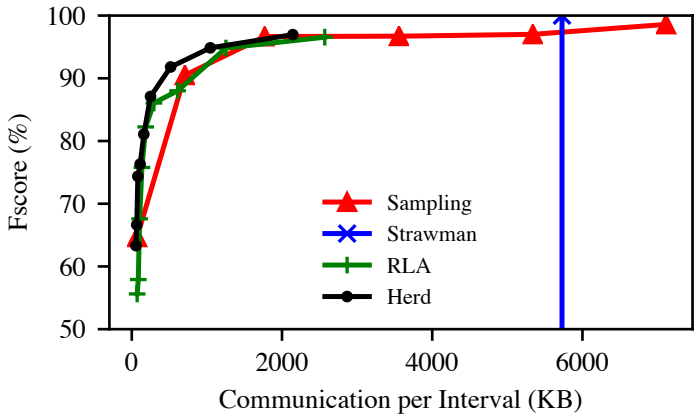
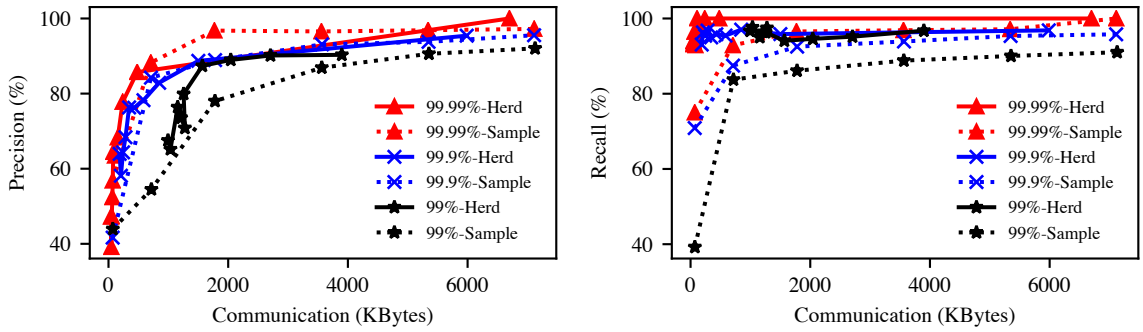


Figure 3.4: Communication vs. Accuracy. Herd can achieve accurate results with comparably lower communication overhead than existing approaches.

sampling technique [56], which probabilistically samples packets based on a sampling rate and reports all of those samples to the coordinator.

Communication and State Savings. We quantify the state overhead as the number of stateful counters required at the switch and the communication overhead as kilobytes sent to the coordinator for each window interval. We quantify accuracy in terms of both precision (PR) and recall (RE) and present them as a single F_1 score calculated as $\frac{2\times PR\times RE}{PR+RE}$. In Table 3.2, we see that Herd achieves 38% savings in the state required for alternate approaches. In Figure 3.4, we compare how much communication is needed to reach a certain level of accuracy. We see that to achieve an F_1 score of 97%, Herd communicates 17% less than sampling packets with a probability of 0.075.

Sensitivity to Heavy-Hitter Threshold. We see in Figure 3.4, that Herd can achieve higher accuracy for less bandwidth compared to existing approaches. As the threshold for heavy-hitters decreases, this advantage becomes more pronounced. In Figure 3.5, we show the communication/accuracy tradeoff compared with sampling for three different heavy-hitter thresholds ranging from the 99.99^{th} percentile to the 99^{th} percentile threshold. In each case, we see that Herd performs strictly better than the sampling approach except



(a) Precision

(b) Recall

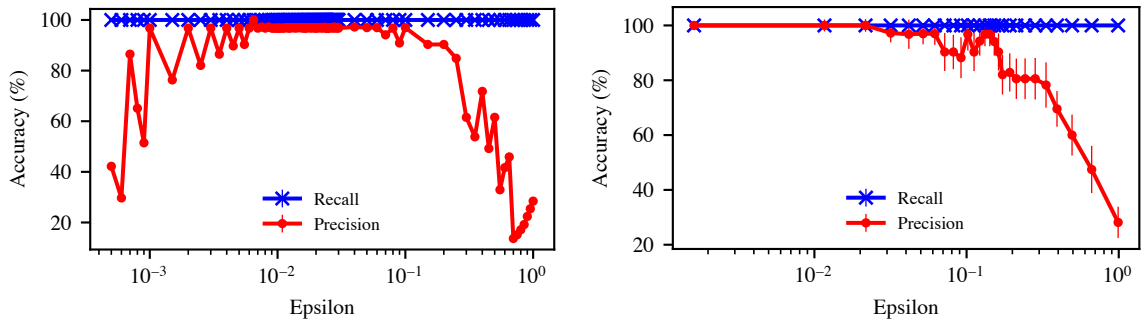
Figure 3.5: Accuracy vs. Communication Cost and Threshold. Each line shows the cost (communication) to achieve a given (a) precision/(b) recall for a given heavy-hitter threshold (99.99^{th} percentile to 99^{th} percentile). Each point on the line is a particular sampling rate or value of ε.

in the 99.99^{th} percentile threshold and the sampling probability is greater than 0.05 – an unrealistically-high sampling probability for modern data centers.

3.7.3 Tuning for Resource Constraints

So far, we have demonstrated that combining probabilistic counting with reporting reduces both the memory and bandwidth footprint for detecting network-wide heavy hitters. We will now show the relationship between Herd's performance (accuracy) and configuration parameters (ε) for different operational constraints. These relationships guide the design of our tuning algorithm.

Unconstrained Performance. We first show the relationship between accuracy and configuration parameters (derived from ε) for the unconstrained case. Figure 3.6, shows both precision and recall for Herd while varying ε without any resource constraints. As we choose a smaller epsilon, the accuracy of the results improves at the cost of additional communication. However, we do see that this relationship is not strictly monotonic. We observe the bands of decreasing precision when ε is very large or very small. These bands are caused by the the quantization errors introduced when discretizing system parameters



(a) Un-quantized Values of ε

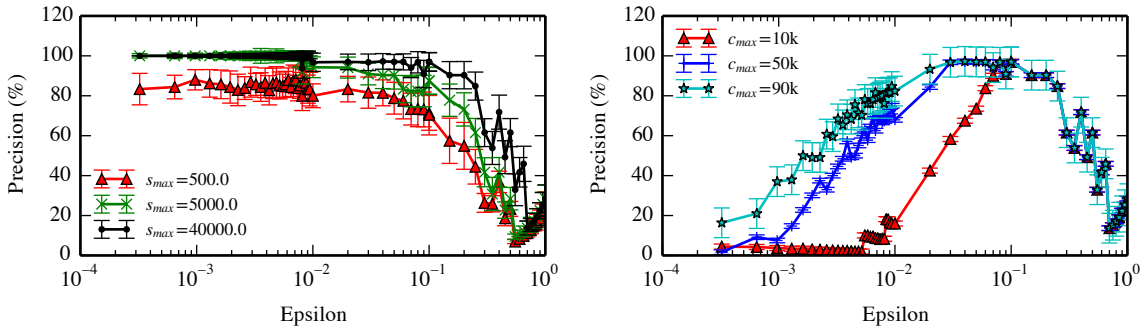
(b) Quantized Values of ε

Figure 3.6: Precision and recall as we vary ε without resource constraints, (a) with unquantized values of ε and (b) with properly quantized values of ε.

such as τ and R (discussed in Section 3.5.4). For example, in the bands of decreasing precision on the right of Figure 3.6a, each data point corresponds to a single global reporting threshold (R) but a range of local thresholds (τ) that can vary by up to a thousand. On the, left side of the graph, the opposite is true; a single local threshold uses a range of global reporting thresholds. By properly quantizing values of epsilon as shown in Section 3.5.4, we do not observe these artifacts as shown in Figure 3.6b.

Constrained State. Here, we limit the number of counters each switch can store. By choosing the sample-and-hold technique for storing counters at switches, we expect that for a given state capacity (S) and sampling probability (s), the data structure will contain a mixture of both small and heavy flows. However, as we increase S and s, we will count more *small flows* in expectation. As shown in Figure 3.7a, increasing S and s improves the precision of the results, but the improvement diminishes as S grows large.

Constrained Communication. We expect that as we decrease ε, we should increase communication and increase the accuracy of the results. Algorithm 6, shows us how to adjust the reporting probability based on the available communication capacity (C), however, as we decrease the reporting probability and the reporting threshold to cope with the com-



(a) (b)

Figure 3.7: Precision for various ε (a) constrained in state, but unconstrained by communication, and (b) constrained in communication, but unconstrained by state.

munication bound, too many false positives are generated as shown in Figure 3.7b. This trend is not reflected in the unconstrained case.

These results show that the relationship between accuracy and ε is non-monotonic, though communication and state costs do monotonically decrease as ε increases. These observations guided the design of our tuning algorithm that empirically tries to find the largest value of ε that achieves the highest detection accuracy.

Tuning Efficacy. To determine the effectiveness of our tuning algorithm, we varied both state and communication constraints and let Algorithm 6 find the best value of ε. In Figure 3.8a, we see the performance of our tuning algorithm when both state and communication constraints are imposed, shown on the x and y-axes, respectively. We see that as resource constraints are relaxed, the system finds a smaller value of ε. In Figure 3.8b, we see the system's accuracy under the same constraints using the best value of ε determined by the system. Herd is able to produce more accurate results as the constraints are relaxed, but Herd also provides good accuracy even under strict memory and bandwidth constraints.

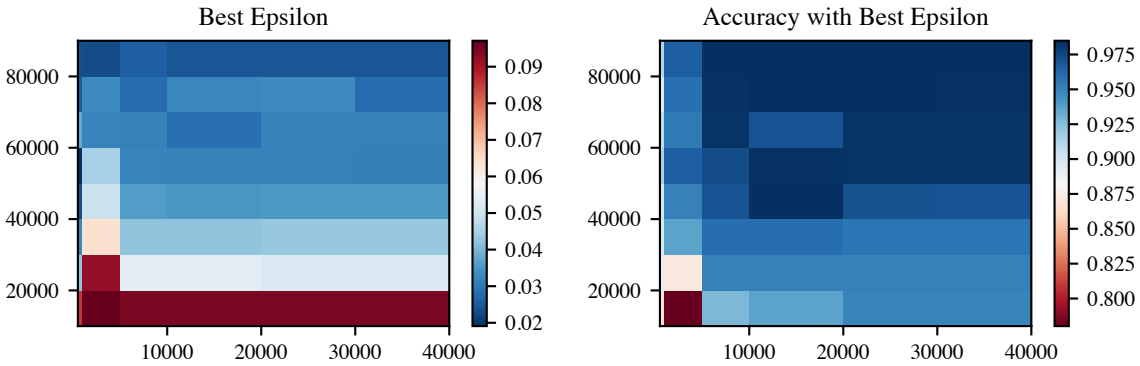


Figure 3.8: Efficacy of tuning. State constraints in (KB) are shown on the x-axes and communication constraints are on the y-axes. The value depicted in the heatmap reflects (a) the best value of ε after tuning, and (b) the best accuracy that is achieved using the best epsilon. Here, accuracy is the average of the precision and recall.

3.8 Related Work

Single-switch heavy hitters with limited state. Prior work showed how to use compact data structures (e.g., count-min sketch [19] and Space-Saving [47]) to compute heavy hitters on a single switch. However, Sivaraman et al. [68] showed that implementing such algorithms with state-of-the-art programmable switches is difficult. Similarly, other techniques such as Cuckoo [55] and d-left [79] hashing can detect heavy hitters with small state, but they prove to be impossible or impractical to implement in modern programmable switches [12]. Recent work, such as ElasticSketch [80], offers a technique to avoid maintaining state for mouse flows in the data plane by offloading the computation to the control plane. Our approach, based on the sample-and-hold [23] technique, uses sampling to filter out mice flows completely in the data plane and only maintains per-flow state for potential heavy hitters.

Scalable measurement techniques. NetFlow [15] was the first standardized approach for both storing flow counters on switches and communicating them to a collector from network switches. However, Netflow incurs significant CPU overhead or specialized hardware to run efficiently. Packet sampling [56] emerged as the de facto technique to cope

with both the memory and communication limitations for detecting heavy hitters. However, packet sampling can introduce significant inaccuracy to detecting heavy hitters, especially on small time scales [56]. FlowRadar [43] reduces the memory and communication overhead of implementing a NetFlow-like monitoring system, using a novel encoding of flow counters. However, it focuses on providing full visibility into all flows all the time rather than the heavy hitters. Similarly, CSamp [67] provides a sampling mechanism for network-wide measurements. While both of these works are general-purpose solutions for performing network-wide measurement of most flows, we offer a tailored solution for continuous, network-wide monitoring of a global threshold distributed across several switches. Furthermore, SketchVisor [33] provides a technique for local and network-wide measurements, yet they do so using *software* packet processing which limits its ability to handle very high data rates. Our work focuses on extending the formal definition of the problem and how to implement a solution within the constraints of programmable switches.

Network-wide heavy hitters with limited communication. Detecting network-wide heavy hitters is an instance of the continuous distributed monitoring (CDM) problem [17]. This formulation of the problem has enabled theoretical analyses that demonstrated upper and lower bounds on the communication complexity [18, 81] for both deterministic and randomized solutions. In our work, we extend the basic model from these theoretical works to account for the realities of flow affinity in modern networks [66, 65], as well as the capabilities of programmable switches to support these protocols. Recent work [31] showed that using adaptive local thresholds to account for flow locality could reduce communication overhead for computing network-wide heavy hitters exactly, but that solution does not scale as the number of nodes increases and requires much more communication overhead than our approach. In contrast, our work accounts for flow locality in the CDM model and the communication costs do not scale in proportion to the number of switches in the network. Our solution also offers tunable accuracy based on bandwidth constraints.

Chapter 4

Conclusion

4.1 Scalable, Network-Wide Telemetry

The need for scalable, network-wide telemetry exists now and will only intensify as the Internet-of-Things grows to over 100 billion connected devices in the coming years [61]. This dissertation has provided an architecture and two systems that help realize scalable, network-wide telemetry. The systems presented in this dissertation balanced the need to perform flexible and fine-grained telemetry queries with the memory, compute and bandwidth resources available to execute them with high accuracy. We leveraged the power of PISA switches both to achieve scale and to implement novel algorithms not previously implemented in hardware. Taken together, these systems can scalably execute a flexible range of queries on a single switch and provide network-wide view for a narrower range of queries in a scalable way.

4.1.1 Flexible and Scalable Telemetry with Sonata

Sonata combines the best of hardware and software query processing to achieve scalable execution for a high-level and flexible query language. Sonata models the resource constraints of modern PISA hardware, compiles high-level query operators to PISA primitives,

and solves an optimization problem that partitions high-level queries across the hardware and software resources given resource constraints and a representative workload. Our implementation and evaluation show that we can reduce the processing load on a software stream processor by up to seven orders of magnitude.

4.1.2 Scalable, Continuous, Network-wide Telemetry with Herd

While Sonata enables partitioning flexible queries across a single switch and stream processor cluster, Herd enables the accurate execution of a subset of these queries in a network-wide setting. Herd performs continuous distributed monitoring to enforce a global threshold on flow packet counts across a distributed set of switches with a cost that does not grow in proportion to the number of switches in the network. We use well-known algorithmic building blocks and model the relationship between their parameters to cope with memory and bandwidth constraints while providing high accuracy under those constraints.

4.2 Lessons Learned

Through the design and implementation of both Sonata and Herd, we make the following observations for building future network telemetry systems.

4.2.1 Federating Systems for Network Telemetry

In building Sonata, we faced a scalability challenge to answer packet-level queries about the state of the network. To handle this scalability challenge, we used PISA switches to partition queries and reduce the amount of processing by general-purpose CPUs (Section 2.4). Using PISA switches in this way is one example of *federating* all available computing resources, each having different strengths and weaknesses, to achieve scale. As we generate, process, and transmit more and more data each day, federating all available computing resources to scale network telemetry will become even more important. By fed-

erating disparate computing resources, telemetry systems will begin to look more and more like distributed systems. As these distributed systems become even more mission-critical to network management and network security, network operators will simultaneously demand that they also be highly available and fault tolerant. These new requirements will likely profoundly change the way we design and build networks to provide the computational capacity, in the right places, needed to make network telemetry a first-class citizen.

4.2.2 Reconciling Advances in Theory and Constraints in Practice

In addition to using heterogeneous compute resources to handle the scalability challenges inherent to network telemetry, we will continue to rely on algorithms that are amenable to distribution across a federation of devices. However, these algorithms must not only offer provable bounds on paper, they must be practically deployable on a range of different devices given the resources and constraints of those devices. In the design and implementation of Herd, we had to discount several candidate data structures because of the resource limitations on the PISA switches. Additionally, we had to modify existing algorithms in order to eliminate errors induced when translating theoretical results into practice (Section 3.5.4). Building network telemetry systems that can federate all available computational resources will rely on bridging the gap between the assumptions made in theoretical work with the practical constraints imposed by the myriad devices supporting the system.

4.3 Future Directions in Network Telemetry

While Sonata and Herd make important advances toward realizing a fully top-down network telemetry system, there are still substantial advances to be made in future work, especially if we seek to achieve intent-driven and automated network control.

4.3.1 Online Cost Modeling and Prediction

Both Sonata and Herd rely on offline mechanisms to estimate the costs of executing queries with a particular set of system parameters. While effective, this approach limits both systems by the availability, fidelity, and stationarity of the data supporting the offline analysis. Being able to model query costs in an online manner or by learning a model to predict query costs would eliminate this dependency and enhance both systems' effectiveness.

4.3.2 Supporting Additional Network Views

While Herd enables executing a particular subset of Sonata queries distributed across a collection of ingress switches, telemetry systems should also support distributing query execution for other kinds of network views, such as along a specific network path [52] or in arbitrary aggregates. However, some types of queries just cannot be executed in a distributed fashion, e.g., performing a `reduce` operation with a function that is not commutative or associative. Extending Sonata and Herd to support queries along a path or with arbitrary aggregation will require an extensive query planner that accounts for the distribution and coordination costs of these queries and incorporates them into our optimization problem along with partitioning and refinement.

4.3.3 Federating Additional Computing Resources

While Sonata itself already has a modular architecture for incorporating heterogeneous switch targets, additional processing resources that do not fit into the current optimization problem should be accounted for. Application specific integrated circuits (ASICs), field-programmable gate arrays (FPGAs), and Network Processing Units (NPUs), could all offer additional computational resources to which portions of queries could be partitioned. However, each of these devices has a slightly different processing and cost model that, to date, is not accounted for in Sonata's query planner. However, these additional computing re-

sources should fit within our optimization framework of compiling high-level operators into each architecture's low-level primitives while accounting for the costs of those primitives.

Bibliography

[1] OpenFlow Switch Specification. https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf, June 2012.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, and Michalis Kallitsis. Understanding the Mirai Botnet. In *USENIX Security Symposium*, 2017.

[3] Apache Thrift API. https://thrift.apache.org/.

[4] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *ACM SIGCOMM*, 2016.

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD International Conference on Management of Data*, 2015.

[6] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically Finding the Cause of Packet Drops. In *USENIX NSDI*, 2018.

[7] Assignment 3, COS 561, Princeton University. https://github.com/Sonata-Princeton/SONATA-DEV/tree/tutorial/sonata/tutorials/Tutorial-1.

[8] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE ICNP*, 2018.

[9] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *USENIX Network and Distributed System Security Symposium*, 2011.

[10] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security Symposium*, 2012.

[11] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, July 2014.

[12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[13] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1974.

[14] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *ACM SIGCOMM Workshop on Research on Enterprise Networking*, 2009.

[15] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. *RFC 3954*, 2004.

[16] Benoit Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. *RFC 7011*, 2013.

[17] Graham Cormode. Continuous Distributed Monitoring: A Short Survey. In *International Workshop on Algorithms and Models for Distributed Event Processing*, 2011.

[18] Graham Cormode, S Muthukrishnan, and Ke Yi. Algorithms for Distributed Functional Monitoring. *ACM Transactions on Algorithms*, 2011.

[19] Graham Cormode and Shan Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, 2005.

[20] Chuck Cranor, Theodore Johnson, Oliver Spatschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[21] The CAIDA UCSD Anonymized Internet Traces 2016-09. `http://www.caida.org/data/passive/passive_2016_dataset.xml`.

[22] Cristian Estan, Stefan Savage, and George Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *ACM SIGCOMM*, 2003.

[23] Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 2003.

[24] Jinliang Fan, Jun Xu, Mostafa H. Ammar, and Sue B. Moon. Prefix-Preserving IP Address Anonymization: Measurement-Based Security Evaluation and a New Cryptography-Based Scheme. *Computer Networks*, 2004.

[25] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN ICFP*, 2011.

[26] Thomer M Gil and Massimiliano Poletto. MULTOPS: A Data-Structure for Bandwidth Attack Detection. In *USENIX Security Symposium*, 2001.

[27] Arpit Gupta, Rudiger Birkner, Marco Canini, Nick Feamster, Chris MacStoker, and Walter Willinger. Network Monitoring as a Streaming Analytics Problem. In *ACM HotNets*, 2016.

[28] Arpit Gupta, Rob Harrison, Ankita Pawar, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Network Telemetry. *arXiv preprint arXiv:1705.01049*, 2017.

[29] Gurobi Solver. `http://www.gurobi.com/`.

[30] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*, 2014.

[31] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-Wide Heavy Hitter Detection with Commodity Switches. In *ACM SIGCOMM Symposium on SDN Research*, 2018.

[32] Mukesh Hira and L. J. Wobker. Improving Network Monitoring and Management with Programmable Data Planes. Blog posting, `http://p4.org/p4/inband-network-telemetry/`, 2015.

[33] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM*, 2017.

[34] Martin Izzard. The Programmable Switch Chip Consigns Legacy Fixed-Function Chips to the History Books. `https://www.barefootnetworks.com/blog/programmable-switch-chip-consigns-legacy-fixed-function-chips-history-books/`, September 2016.

[35] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. In *ACM SIGCOMM*, 2013.

[36] Mobin Javed and Vern Paxson. Detecting Stealthy, Distributed SSH Brute-Forcing. In *ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[37] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, 2015.

[38] Lavanya Jose, Minlan Yu, and Jennifer Rexford. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, 2011.

[39] Jaeyeon Jung, Vern Paxson, Arthur W Berger, and Hari Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *IEEE Symposium on Security and Privacy*, 2004.

[40] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic Load Balancing Without Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 2007.

[41] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" Abstraction in Software-Defined Networks. In *ACM SIGCOMM CoNEXT Conference*, 2013.

[42] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *USENIX Security Symposium*, 2014.

[43] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.

[44] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[45] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny Aggregation Service for Ad-hoc Sensor Networks. In *USENIX OSDI*, 2002.

[46] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database System*, 2005.

[47] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems*, 2006.

[48] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. *ACM SIGCOMM*, 2015.

[49] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *ACM SIGCOMM CoNEXT Conference*, 2015.

[50] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM*, 2016.

[51] James K. Mullin. Optimal Semijoins for Distributed Database Systems. *IEEE Transactions on Software Engineering*, 1990.

[52] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. Compiling Path Queries. In *USENIX NSDI*, 2016.

[53] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*, 2017.

[54] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. IoTPOT: Analysing the Rise of IoT Compromises. In *USENIX Workshop on Offensive Technology*, 2015.

[55] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *Journal of Algorithms*, 2004.

[56] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. *RFC3176*, 2001.

[57] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. Track join: Distributed joins with minimal network traffic. In *ACM SIGMOD International Conference on Management of Data*, 2014.

[58] An Update on the Memcached/Redis Benchmark. `http://oldblog.antirez.com/post/update-on-memcached-redis-benchmark.html`.

[59] Apache Flink. `http://flink.apache.org/`.

[60] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). `https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines`.

[61] Insight Into The Global Threat Landscape: NETSCOUT Arbor's 13th Annual Worldwide Infrastructure Security Report. `https://pages.arbornetworks.com/s/82-KNA-087/images/13th_Worldwide_Infrastructure_Security_Report.pdf`.

[62] OpenSOC. `http://opensoc.github.io/`.

[63] OpenSOC Scalability. `https://goo.gl/CX2jWr`.

[64] The CAIDA Anonymized Internet Traces 2016 Dataset. `https://www.caida.org/data/passive/passive_2016_dataset.xml`.

[65] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*, 2015.

[66] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *ACM SIGCOMM*, 2017.

[67] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. cSamp: A System for Network-Wide Flow Monitoring. In *USENIX NSDI*, 2008.

[68] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research*, 2017.

[69] Slowloris HTTP DoS. `https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris`, June 2009.

[70] Ed H. Song, T. Zhou, Z.B. Li, Z.Q. Li, P. Martinez-Julia, and L. Ciavaglia. Network Telemetry Framework. `https://tools.ietf.org/html/draft-song-opsawg-ntf-02`, December 2018.

[71] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator Placement for In-Network Stream Query Processing. In *Symposium on Principles of Database Systems*, 2005.

[72] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying Datacenter Network Debugging with PathDump. In *USENIX OSDI*, 2016.

[73] Apache Spark. `http://spark.apache.org/`.

[74] Barefoot's Tofino. `https://www.barefootnetworks.com/technology/`.

[75] P4 Behavioral Model Software Switch. `https://github.com/p4lang/behavioral-model`.

[76] Scapy: Python-based Interactive Packet Manipulation Program. `https://github.com/secdev/scapy/`.

[77] SONATA GitHub Repository. `https://github.com/Sonata-Princeton/SONATA-DEV`.

[78] Bapi Vinnakota. P4 with the Netronome Server Networking Platform. `https://www.netronome.com/blog/p4-with-the-netronome-server-networking-platform/`, May 2016.

[79] Berthold Vöcking. How Asymmetry Helps Load Balancing. In *IEEE Symposium on Foundations of Computer Science*, 1999.

[80] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *ACM SIGCOMM*, 2018.

[81] Ke Yi and Qin Zhang. Optimal Tracking of Distributed Heavy Hitters and Quantiles. In *ACM SIGMOD-SIGART-SIGACT Symposium on Principles of Database Systems*, 2009.

[82] Minlan Yu. Network Telemetry: Towards A Top-Down Approach. *ACM SIGCOMM Computer Communication Review*, 2019.

[83] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*, 2013.

[84] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. ProgME: Towards Programmable Network Measurement. In *ACM SIGCOMM*, 2007.

[85] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative Network Monitoring with NetQRE. In *ACM SIGCOMM*, 2017.

[86] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.

[87] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM*, 2015.