Exploring Network-Wide Flow Data with Flowyager

Said Jawad Saidi[1] Aniss Maghsoudlou[1] Damien Foucard[2]
Georgios Smaragdakis[2,1] Ingmar Poese[4] Anja Feldmann[1,3]
[1]Max Planck Institute for Informatics [2]TU Berlin [3]Saarland University [4]BENOCS GmbH

***Abstract*—Many network operations, ranging from attack investigation and mitigation to traffic management, require answering network-wide flow queries in seconds. Although flow records are collected at each router, using available traffic capture utilities, querying the resulting datasets from hundreds of routers across sites and over time, remains a significant challenge due to the sheer traffic volume and distributed nature of flow records.**

In this paper, we investigate how to improve the response time for *a priori unknown network-wide* queries. We present Flowyager, a system that is built on top of existing traffic capture utilities. Flowyager generates and analyzes tree data structures, that we call Flowtrees, which are succinct summaries of the raw flow data available by capture utilities. Flowtrees are self-adjusted data structures that drastically reduce space and transfer requirements, by 75% to 95%, compared to raw flow records. Flowyager manages the storage and transfers of Flowtrees, supports Flowtree operators, and provides a structured query language for answering flow queries across sites and time periods. By deploying a Flowyager prototype at both a large Internet Exchange Point and a Tier-1 Internet Service Provider, we showcase its capabilities for networks with hundreds of router interfaces. Our results show that the query response time can be reduced by an order of magnitude when compared with alternative data analytics platforms. Thus, Flowyager enables *interactive network-wide queries* and offers unprecedented drill-down capabilities to, e.g., identify DDoS culprits, pinpoint the involved sites, and determine the length of the attack.

***Index Terms*—Network Data Summarization, Network Monitoring, Network-Wide Traffic Analytics.**

I. Introduction

Network operators have to continuously keep track of the activity in their networks over both long and short time windows. Over long time windows, e.g., days or hours, network operators are interested in provisioning network capacity or making informed peering decisions. Over short time windows, e.g., minutes, network operators would like to identify and rectify unusual events, e.g., attacks or network disruptions. To that end, they typically rely on either flow-level or packet-level captures from routers within their network [1]. For a summary of tasks and how previous work tackled them see Table I.

Flow captures include 5-features: source (src) and destination (dst) IP addresses, port numbers, protocol ID–to summarize traffic information per flow–Packet and byte count [2], [3]. Packet captures gather packet headers [4], [5], [6], [7]. Unfortunately, gathering data for every packet is often too expensive at high-speed links. Thus, flow-level and packet-level capture tools rely on sampling packets, e.g., 1 of every 10k packets [8].

Among the most popular capture tools are NetFlow [9], IPFIX [10], sFlow [11], and libpcap [7]. All major router and high-end switch vendors (Cisco, Juniper, Alcatel-Lucent, and Huawei) offer flow capture capabilities [9], [10], [11][1] in their commodity as well as high-end products.[2]

Recently, query-driven solutions, e.g., Sonata [16], Stroboscope [17], and Marple [18], made it possible to compile specific queries into telemetry programs and collect data from all queried network nodes. These solutions provide exceptional flexibility, but they require the network operator to know *a priori* (i) the nature of the network problem, (ii) the network-related query that has to be compiled into telemetry programs, (iii) the network node where the telemetry capability is available, and (iv) the node where the query has to be executed. Unfortunately, in large networks with hundreds of interfaces, operational issues arise at different parts of the network and the queries that are required are not known in advance. In many cases, network engineers have to try different queries to locate the source and type of problem interactively. Thus, it takes a prohibitively large time to compile such queries into telemetry programs. Another obstacle toward adopting such solutions is that this requires hardware investments by the network operator. For example, Marple relies on P4-programmable software switches that are not yet widely adopted by Internet Exchange Points (IXP) operators and Internet Service Providers (ISP).

To the best of our knowledge, there is at this point in time no system that offers answers to *a priori unknown network-wide* queries in a scalable *interactive* manner, even though the necessary raw network data, e.g., via NetFlow [12], [5], sFlow [11], IPFIX [10], or libpcap [7] is collected by most

[1]NetFlow is a Cisco trademark, so other vendors market the NetFlow support with other names, e.g., Juniper Networks use the trademark Jflow or cflowd.

[2]NetFlow and IPFIX capabilities are available in router series, e.g., Cisco IOS-XR, IOS and Catalyst router [12], Juniper M-, T-, and MX-series routers [13], Alcatel-Lucent 7750SR [14], Huawei NE-series routers [15], and switches, e.g., Cisco (5600, 7000, 7700), Enterasysthese (S- and N-series), and servers, e.g., VMware (vSphere 5.x).

operators.

From an operational point of view, fast exploration of large volumes of network flows over time and across sites is useful to answer a range of operational queries (see Table I). Yet, network operators need to be able to tackle such tasks in a unified and systematic way with reliable and scalable tools. Existing data analytics systems, e.g., Spark [19], are not tailored to analyze network data when it comes to scalability, interactivity, handling of geo-distributed data, or answering a priori unknown network-wide queries.

In this paper, we design, implement and evaluate a system, Flowyager, that is able to answer *a priori unknown network-wide queries* with fast response, and, thus, enables *interactive* exploration of network data *across network sites* and *over time*. The architecture of our system is built around the following requirements:

(1) Scalability: The system should grow with the network size, the number of data sources, and the analysis requirements. Hereby, it should enable distributed deployment and not require all data to be transferred to a central location.

(2) Reuse of *existing* flow captures: As it takes significant effort to deploy novel network capture utilities, the system should work on top of existing, widely deployed, and supported flow capture capabilities, such as NetFlow, sFlow, IPFIX, or libpcap. In high-speed links, these tools typically sample packets [8] to provide summaries of flow activity.

(3) Support of *interactive* and *ad-hoc* queries: To easily explore network data, the system needs to offer an interface that is flexible and interactive (meaning response times in the order of seconds) so as to improve user productivity and enable drill-down capabilities. Possible queries vary and a system should not only focus on batch-style known queries but also enable quick *ad-hoc* exploration of the data, i.e., answer queries that *are not known in advance*, and allow for follow-up queries. Answering network-wide queries should not require custom code or scripting as network operators usually neither have the required time nor the resources (e.g., storage or computing). The goal is to reduce the response time of queries from hours or dozens of minutes to seconds and, thus, enable interactive and drill-down queries.

(4) Support of queries *across network sites* and *over time*: Most queries are not just for some specific time period or network site. Rather, they correlate data spanning multiple periods, across network sites, and at different granularities, e.g., per site, region, time of day, and event. The system should be able to collect, index, and store summary data across multiple sites and over time.

Although most networks gather raw flow data, answering network-wide queries is difficult due to: (a) the distributed nature of data collection (per interface and router) at different locations, i.e., at multiple border and/or backbone routers, (b) the massive and ever-increasing size of the flow data (despite sampling) incurring an excessive cost to store, transfer, and analyze flow data–indeed, it often has to be deleted after some time to be able to store more recent data, and (c) the international footprint with the requirement to comply with local legislation which may prohibit the transfer of raw data.

Application	Related Work
Aggregated flow statistics (range queries over IP/ports/time/location)	[20], [21], [22], [23], [24], [16], [18]
Counting traffic	[20], [25], [26], [27], [28], [29], [30], [24], [16], [23] [31], [32], [33], [17], [34], [35], [36], [37]
Traffic matrix	[27], [31]
DDoS diagnosis	[35], [25], [27], [38], [39], [40], [16], [31]
Super-spreaders Detection	[35], [25], [16]
top-K number of flows	[26], [36], [37], [41]
Flows above threshold T (Heavy Hitters)	[34], [33], [35], [25], [26], [22], [23], [42], [24], [16], [43]
Heavy Changers Detection	[34], [35], [25], [42], [44], [43]
Blackhole Detection	[45], [46], [47], [27]
Port-based / 4/5-tuple queries	[20], [34], [33], [32], [25], [26], [27], [16]

Table I: Typical network queries and systems to tackle them. Currently, no system addresses all of them.

To achieve the above, we need data structures that generate *succinct* and *space-efficient summaries*, as well as *indexing* of network flow captures that are light (easy to transfer), can be analyzed locally, and enable answering *interactive a priori unknown network-wide queries*. These data structures should be used to *accurately* and *quickly* answer queries and tackle network management tasks that involve *multiple sites* and/or span *multiple periods* in a *user-friendly* and *unified* way.

The contributions of our paper are:

- We design, deploy and evaluate Flowyager, a system built on top of existing voluminous network captures, that enables interactive data exploration. We show that with Flowyager the query response time for network-wide queries can be reduced from hours or minutes to seconds.
- We propose a lightweight self-adjusting data structure, Flowtree, that inherits the performance of previously proposed hierarchical heavy hitter structures for computing flow summaries. Flowtree summarizes elephants as well as mice flows and supports multiple operators, such as merge, compress, and diff, to summarize information across multiple sites and time periods.
- We propose an SQL-inspired language, FlowQL, which provides a unified interface to ask arbitrary ad-hoc queries about flow captures, including drill-down queries.
- We show that when answering a wide range of queries, Flowyager significantly outperforms the state of the art data analytics systems, namely, ClickHouse, and Spark.
- We share our experience of rolling out Flowyager at different operational environments, namely a large IXP and a tier-1 ISP, and showcase how to tackle various network management tasks. We will make Flowyager and its code available for non-commercial use under the following link [48].

II. State of the art

Existing network analytics systems, such as [49], [50], typically transfer the raw traces to a centralized data warehouse for archiving and processing. However, transferring the raw traces is increasingly expensive due to the data volume — e.g., Terabytes of flow data generated in a single day can be out of sync, and all need to be transferred. Moreover, additional constraints are posed by national regulations when networks operate at regions under different jurisdictions: for example, transferring data that includes user identifiers, e.g., IP addresses allocated to EU citizens, without their consent, violates the EU General Data Protection Regulation (GDPR) [51]. Fines are steep, namely up to 4% of worldwide turnover or 20 million Euros, whichever is higher.

Network monitoring systems: Alternative proposals suggest to enable powerful custom data collection per query and realize this by combining traffic mirroring and deterministic packet sampling. These include query-based monitoring such as Stroboscope [17], network troubleshooting using mirroring [52], [53], analysis of in-network packet traces [27], [54], as well as monitoring links on-demand as shown by Gigascope [20], pruning-based solutions such as Cheetah [55] or other SDN-based monitoring, such as [56] or PRECISION [57]. The main disadvantage of these systems is that the target flows, sites, and periods of interest need to be known in advance, which is often not the case in practice.

Streaming network telemetry systems, from more classic approaches such as A-GAP [37] to the numerous modern solutions, such as Sonata [16], FlowBlaze [58] or Poseidon [59], build on the same ideas but require programmability from network devices, e.g., P4 switches or FPGA. These systems assume that users can predefine what is relevant and optimize the monitoring accordingly, often following a top-down approach [60]. As a consequence, if, potentially, all flows are of interest, these systems can degrade to "standard" flow monitoring which for large networks is challenging. Marple [18] adds flexibility to network-wide monitoring but requires P4-programmable capabilities that have not been yet widely adopted in wide-area networks by ISP and IXP operators.

Big data analytics systems: Some operators directly feed their flow captures into state-of-the-art analytics systems, often based on the map-reduce principle, e.g., Spark [19] and Hadoop [61], or column-based databases, e.g., ClickHouse [62]. This has scalability issues. Thus, recently proposed big data analytic systems—see [63], [64], [65], [66], [67] as well as [68] and references within–suggest to use a distributed setup whereby data is locally preprocessed, e.g., by aggregation or sampling, and then centrally analyzed. This reduces the need to transfer the raw data. Note that none of the above focuses on network management tasks. Thus, their programming interface follows the map and reduce paradigm which differs from network operation tasks. Even though such systems can provide significant speedup for tasks that can be parallelized, not all network management tasks may benefit. Like Flowyager, such big data analytics systems are *flexible* w.r.t. the queries supported. Yet, unlike Flowyager, they typically are not *compatible* with existing network monitoring software, do not fully support principled *aggregation* (over time, space and flows), do not offer any *history*, and do not give any performance (accuracy or runtime) *guarantees*.

Data summaries–Heavy Hitters: Previous work on computing network summaries has focused on how to efficiently compute heavy hitters (HH) [5], [69], [4], [70] and hierarchical heavy hitters (HHH) [71], [23], [72] using minimal resources to be able to compute them on the router itself. These solutions provide an online summary of the (hierarchical) heavy hitters for a fixed observation window, at one location, and only on a given subset of the data. In contrast, to answer interactive network management queries (see Table I), we need summaries over different subsets of the data, per site/router and across sites/routers, and at many different time granularities, from minutes to days — or even months.

Heavy hitters change as data is aggregated: as more data comes in, popularities increase overall. Consequently, the threshold to be considered a heavy hitter should be raised. In contrast, some HHH data structures, e.g., [23] use a single manually defined absolute threshold (e.g., frequency above 1000) to characterize heavy hitters, resulting in a data structure unable to adapt its definition of heavy hitter as the underlying data changes. Flowyager builds upon heavy hitter data structures by adding support for *aggregation* (over time, location, and flows) and adding *flexibility* w.r.t. the supported queries.

Data summaries–Sketches: Another approach for computing network summaries are sketches, e.g., [73], [34], [74] as well as systems that utilize sketches for network monitoring and debugging [75], [25], [35], [76], [77]. The capabilities of sketches include counting, top-K, HH, as well as HHH. They are highly space-efficient data structures that support many types of queries. Yet, most do not support range queries, e.g., queries that involve a range of sites and/or time periods. Moreover, extracting an estimate from sketches is often not time-efficient. We note that the focus of sketches is similar to that of HHH, i.e., computing online summaries for a fixed observation window with minimal resources. Flowyager could be built upon sketches but we decided to build upon a HHH data structure.

III. Flowyager Architecture

To address the challenges outlined in the introduction, we build a scalable distributed network data analysis architecture, Flowyager. Its *input* is existing per-interface network *flow captures*, either flow summaries—reporting on packet, byte, or flow counts per 5-tuple (src/dst IP address, src/dst port, protocol)—or packet-level summaries (e.g., trace sample). We emphasize that we do *not* propose yet another NetFlow. Its *output* is network reports including packet, byte, or flow counts across network sites and time periods. Prime users, i.e., network operators, can access the data via FlowQL, an

	Net. Mon.	Analytics	HHH	Sketch	Flowyager
Input: Packets	✓	✗	✗	✗	✓
Input: Flows	✓	✗	✗	✗	✓
Distributed Queries	✓	✗	✗	✗	✓
Online	✗	✓	✗	✗	✓
Arbitrary Queries	✗	✓	✗	✗	✓
Query language	✗	✓	✗	✗	✓
Summarization	✓	✓	✓	✓	✓
Low Installation Cost	✓	✗	✓	✓	✓
Low Maintenance Cost	✓	✗	✓	✓	✓
Adaptivity to Data	✓	✗	✓	✓	✓

Table II: Comparison of systems w.r.t. functionality offered. ✓: full support, ✗: no support.

SQL-inspired query language that returns results in seconds and, thus, enables interactive ad-hoc queries with drill-down capabilities. For a comparison between Flowyager and other approaches, we refer to Table II.

To underline Flowyager's capabilities for exploring network data, we show in Fig 1 and Fig. 2 screenshots of Flowyager's Web interface. The Web interface highlights that searches are possible across time ranges, site sets, and feature sets. Moreover, it showcases Flowyager's drill-down capabilities that are also visually supported.

Flowyager is a modular system that consists of three main components:

1) *FlowAGG*, which takes existing flow (or packet) captures as input and computes flow summaries, using *Flowtrees* (see below), which it stores and exports. Besides, *FlowAGG* may, if it has enough storage, keep a local copy of the flow captures themselves.
2) *FlowDB*, which takes flow summaries as input, stores, and indexes them, while using them to answer FlowQL queries. It can use FlowAGG internally to compute further flow summaries.
3) *FlowQL*, which uses the flow summaries kept within FlowDB to answer interactive or batch-style queries including Hierarchical Heavy Hitter/top-K queries, Above-Thresh queries, or top-K heavy changer queries across time and sites.

To better understand the system architecture, Figure 3 gives an overview of the overall system, while Figure 4 presents Flowyager's processing pipeline. Each router sends its data to a NetFlow collector ①, which forwards it to one of potentially many distributed FlowAGG instances ②. Each FlowAGG instance computes summaries ③ and then uploads these either to another FlowAGG instance or directly to FlowDB ④[3]. FlowDB then processes the summaries ⑤ and uses them to answer user queries ⑥.

Flowtree is a data summary of a stream of raw flow data that supports efficient 1-d HHH extraction and other operators. Flowtrees are the data primitives of Flowyager.

[3]For simplicity we restrict our discussion to a centralized instance of FlowDB. However, it is possible to use a hierarchical design similar to what has been proposed for logs of distributed servers [78], [79]

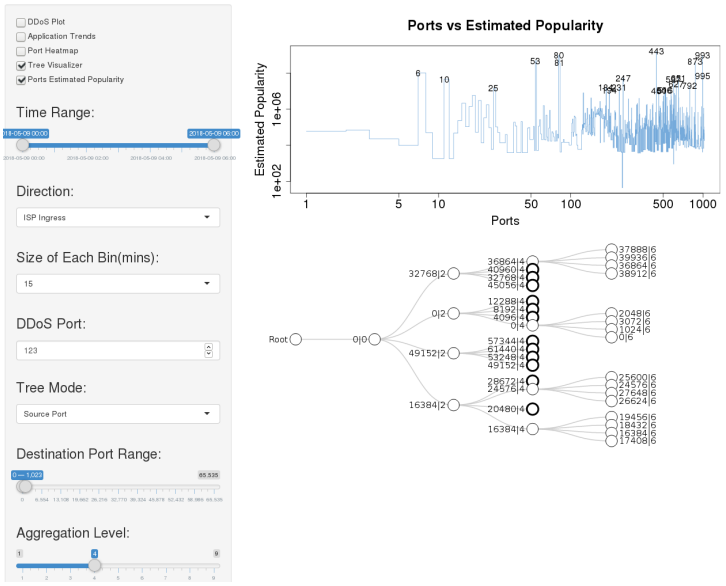


Figure 1: Flowyager: Interacting with 1-feature Flowtrees.

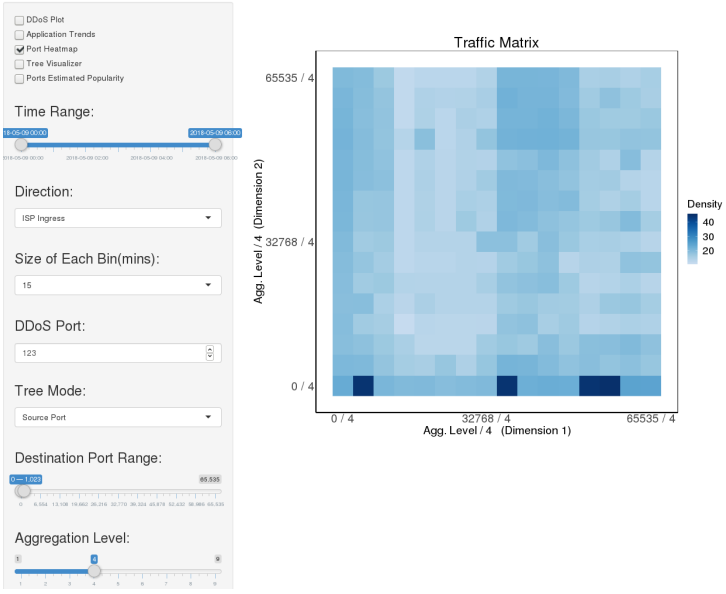


Figure 2: Flowyager: Interacting with 2-feature Flowtrees.

Details on the design and implementation of Flowtree data structure and Flowtree operators are presented in Section IV.

FlowAGG uses a separate plug-in, written in C, for each data source, including IPFIX, NetFlow, sFlow, and libpcap.

FlowDB is responsible for collecting and storing the Flowtrees. It also provides an interface that the user of the Flowyager can use to answer network-wide queries based on the stored Flowtrees, **FlowQL**, whose design is largely inspired by GSQL [20] which uses an SQL-like query language. Using GSQL directly does not suffice due to the unique capabilities of Flowyager. Details on the design and implementation of FlowDB are presented in Section V.

In total, it took approximately 21k lines of code (LoC) in C and C++ to realize Flowyager. About 16k LoCs are for

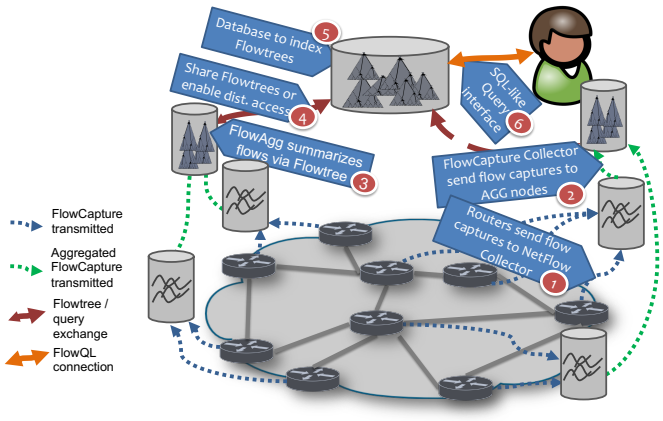


Figure 3: Flowyager architecture.

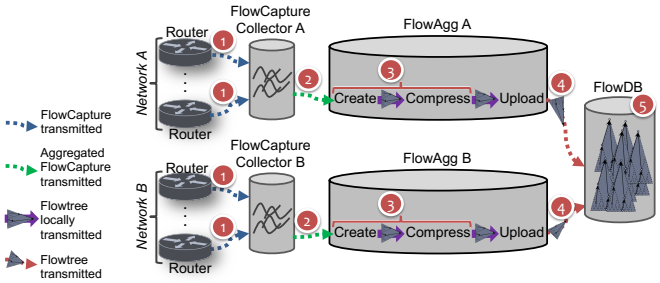


Figure 4: Flowyager Processing Pipeline.

FlowDB, 1.5k for FlowAGG, 2.5k for Flowtree library, and 1k for shared components.

IV. Flowtree

Flowtree is the data structure that is used as a data primitive in Flowyager. Before we dive into the details of Flowtree and its operators, we provide background on Hierarchical Heavy Hitter (HHH) data structures.

A. Hierarchical Heavy Hitters

To enable Flowyager we need succinct summaries from flow captures that are light to transfer, yet, allow for real-time, interactive queries using different flow feature sets. A *flow feature* refers to any of the components of a flow's 5-tuples, namely protocol, src and dst IP, src and dst port. A *feature set* includes a subset of the possible 5 flow features.

We take advantage of the fact that most of the data on the Internet is skewed in the sense that Zipf's law [80], [81], [82] typically applies. However, flat summaries, i.e., histograms, do not suffice. Rather, we need hierarchical heavy hitters (HHH) [4]. HHH utilize attribute hierarchies and identify the most popular elements across a hierarchy. For IPv4 prefixes, we use the network prefix length as an obvious feature hierarchy. As such, 10.1.2.0/23 is the parent of 10.1.2.0/24 and 10.1.3.0/24. For ports, we can use port ranges, e.g., 80/15 is the parent of 80/16 and 81/16. Each feature hierarchy, by default, uses a mask. An IP a.b.c.d is part of the prefix a.b.c.d—n_1 and a.b.c.d—n_1 is a more specific prefix and, thus, a child of a.b.c.d—n_2 if $n_1 > n_2$. The same applies to ports, whereby, e.g., 0—8 refers to the ports from $[0, 63]$. It is possible to define custom hierarchies, e.g., all Web ports, all DNS ports, or all well-known ports.

Ideally, one would use 5-dimensional hierarchical heavy hitters (5-d HHH), across all flow features. Unfortunately, this is infeasible due to its computational complexity [71], [83]. Rather, we use 1-d HHH which can be updated in amortized $O(1)$ time per entry while maintaining the accuracy for HHH and space efficiency of $O(H/\epsilon \log(\epsilon N))$, whereby N is the number of items processed, H is the number of hierarchy levels, and ϵ bounds the precision [71], [83].

Contrary to previous work, we do not restrict the 1-d HHH to a single flow feature. Our first key functionality is that we can generalize 1-d HHH by defining a *joined hierarchy* for a given feature set, e.g., a joined hierarchy for both dst IP and dst port, whereby, the parent of 10.1.2.0/24—80/16, as well as 10.1.3.0/24—81/16 (IP range—port range) is 10.1.2.0/23—80/15. The parent of 10.1.2.0/23—80/15 is 10.1.0.0/22—80/14 and its great-grandparent is 10.1.0.0/21—80/13. For visualization of a sample 2-f hierarchy see Figure 5. In effect, we rely on *generalized flows*: Flows summarize related packets over time at a specific aggregation level. Possible feature sets include "4-feature" flows (i.e., (src IP, dst IP, src port, dst port)), "2-feature" flows, e.g., (dst IP, dst port) (DIDP).

The joined hierarchy can capture the correlation of more than one dimension, e.g., the correlation between IP activity and port activity. It allows identifying heavy hitters on sets of features, and thus, investigating more complex use cases. For example, in an attack, both the target IP and port are important to investigate the type of attack. In general, any query that involves multiple features can be potentially benefited by this joined hierarchy.

Our second key functionality is that if the 1-d HHH data structure supports the operators *merge* ($\cup$) and *compress*, we can compute summaries across time and/or space. In effect, these two operators allow us to add the features *time* and *location*. Given two data structures, A_1 for time period t_1 (location l_1) and A_2 for t_2 (l_2), we get the joined data structure by $A_{12} = (A_1 \cup A_2)$. The *compress* operator is especially useful in reducing the memory footprint of the structure. This operator prunes the tree leaves, and if needed the internal nodes, whose contributions are less than some configurable thresholds, and summarizes their contribution to their parents.

Other operators are *diff*, *query*, *drill-down*, *HHH* resp. *TOP-k*, *Above-x* The diff operator is useful to identify changes, the drill-down operator to explore sub-regions. The HHH and Above-x operators allow us to find popular

[4]The set of HHH for a single hierarchical attribute with popularity counts and a threshold θ corresponds to finding all nodes in the hierarchy such that their HHH count exceeds $\theta * N$, whereby the HHH count is the sum of all descendant nodes which have no HHH ancestors.

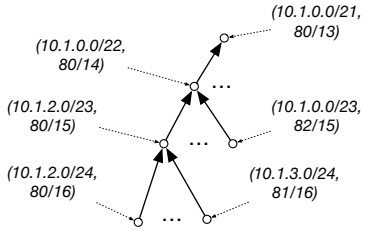


Figure 5: Example: 2-Feature flow hierarchy.

feature sets. The operators are used for interactive queries via FlowQL.

B. Flowtree Data Structure

After evaluating different 1-d HHH data structures, including those of Cormode et al. [71], [83], Basat et al. [23], and Mitzenmacher et al. [72], we decided to augment the structure by Cormode et al.: this data structure is self-adjusting and its entries can be easily extracted via enumeration; thus, it provides natively drill-down capabilities. Flowyager does *not* intrinsically depend on this data structure; rather, it can be built on top of any data structure that supports abstract hierarchies and the basic operators.

Flowtree data structure: Generalized flows form a tree via its hierarchy where each node corresponds to a flow. An edge exists between any two nodes a, b if a is a subnode of b in the feature hierarchy, i.e., if $a \subset b$ —see Figures 8(a) and 8(b). We annotate each node with its popularities, including packet count, flow count, and byte count for UDP and TCP. The popularity of a node is the sum of its own popularity and the popularity of the children—see Figure 7(c).

However, during the construction of the trees, we only keep the nodes' "complementary popularity", namely the popularity (pop) that is not covered by any of the children. Thus, it is possible to prune such a tree by pushing the contribution of the pruned nodes to their parent. This is a *key functionality* for efficiently updating our self-adjusting data structure. Flowtree keeps "popular" nodes and prunes "unpopular" ones by summarizing them at their parent. Flowtree inherits the insertion and self-adjusting strategy from Cormode et al. but rather than allowing the number of nodes to grow unlimited, we limit the maximum number of nodes that a tree can contain by repeatedly pruning (compressing) the tree when necessary. Still, Flowtree closely matches the excellent performance and accuracy bounds for 1-d HHH in terms of space efficiency and precision.

C. Flowtree: Visualizing the Concepts

We start with the visualization of the differences between popularities and complementary popularities in Figure 8. Next, we show the two different feature hierarchies, namely a 1-feature hierarchy on IP addresses, and a 4-feature hierarchy on src/dst IP addresses and src/dst ports with and without popularities, see Figures 7(a) and 8.

Initially, a Flowtree has exactly one entry—the root. When adding a node, we add a new leaf node if necessary and a subset of the nodes on the path to the first existing parent, (in the worst case the root) and update the statistics of the leaf node. We call these intermediate nodes as internal nodes. Thus, each node maintains the complementary popularity (comp_pop), the popularity (pop) that is not covered by any of the children, see Alg. 1. Popularities are computed from the complementary popularities by summing the complementary popularities of all nodes in its subtree including its own. This can be done via a depth first search in O(# nodes) time, see Alg. 2. This uses two functions for finding parents of a node. parent(node) refers to the direct parent in the feature hierarchy while find_parent(node) refers to the parent in the Flowtree.

Updating an existing node corresponds to finding it, which takes time O(1) using an appropriate hash-map. Adding a new node may take up to O(# hierarchy level) time (using an appropriate hash-map). Yet, the expected number of new nodes is small if the distribution of the data is skewed.

To limit Flowtree memory footprint, we periodically or on demand, delete nodes with low popularity. We first compute the popularities by using the stats function in Alg. 2 and then prune nodes whose complementary resp. absolute popularity are below an adjustable threshold. This ensures that at any time the number of nodes in a Flowtree is proportional to the number of processed flows resp. less than a predefined maximum. The complementary popularity of a deleted node as well as its children are pushed to its parent. The overall cost of such a compression step is O(# nodes). Note that since only nodes with small popularity are deleted, the complementary popularity of an interior node is a good estimate of the cardinality of the contributing flow set. Finally, to control the rate of the growth of the tree and preventing the frequent addition and deletion of internal nodes, we insert the internal nodes with a probability of p. The default value of p is 0.3.

D. Flowyager Operators

Query and drill-down: The base operators are *query* (see Fig. 6) and *drill-down*. If the feature f is a node in the Flowtree, the answer is computed from the node statistics. Otherwise, we find the potential node, q, that corresponds to f and estimate its popularity based on the popularity of the predecessor of q, p, and its children, C. We split the children into two subsets: C_f and $C_o = C - C_f$, whereby C_f includes those that are a subset of f in the hierarchy. Now, $\sum_{c \in C_f} \text{pop}(c)$ is a lower bound for the popularity of f and two estimates of f's popularity are $\text{pop}(p) - \sum_{c \in C_o} \text{pop(c)}$ or $\text{comp\_pop}(p) + \sum_{c \in C_f} \text{pop}(c)$, see Fig. 6. If the feature set does not correspond to a node p, the query is expanded to a tree-walk starting at the smallest possible parent of p. The output of the query are then all nodes and their popularities that match the input feature set. For example, src_ip = a.b.0.0—16 and src_port = 80—16

Algorithm 1 Flowtree: Creation/ update

Function: Build_Flowtree (pkts resp. flows)
1: Initialize Flowtree
2: **for** all pkts/flows **do**
3: **Extract_features**(pkt resp. flow).
4: **Construct** node from features.
5: **Add** (Flowtree, node, feature set).

Function: Add (Flowtree, node, features)
1: Add_node(Flowtree, node, features).
2: next = next_parent(node).
3: **while** next != parent(node) or (next ∈ tree). **do**
4: Add_node(Flowtree, next, NULL) with probability p.
5: next = next_parent(next).

Function: Add_node(Flowtree, node, features)
1: **if** node exists **then**
2: comp_pop[node] += stats(flow/pkt).
3: **else**
4: **Insert** node with comp_pop[node] = stats(flow/pkt).
5: parent(node) = find_parent(Flowtree, node).
6: **for** child in children(parent(node)) **do**
7: **if** child ∈ node **then**
8: parent(child) = node.

Algorithm 2 Flowtree: Stats and Compress operator

Function: Stats(Flowtree)
1: **Initialize** pop to comp_pop for all nodes
2: Node_list = nodes of Flowtree in DFS order
3: **for** node in Node_list **do**
4: pop[parent(node)] += pop[node]

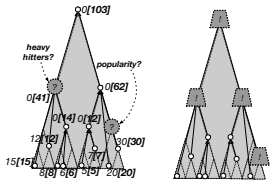
Function: Delete(Flowtree, node)
1: parent = find_parent(Flowtree, node).
2: comp_pop[parent] += comp_pop[node].
3: children(parent) += children(node).
4: **Free** node

Function: Compress(Flowtree, thresh_comp_pop, thresh_pop)
1: Stats(Flowtree).
2: **for each** node **do**
3: **if** (node is leaf **and** comp_pop[node] < thresh_comp_pop) **then**
4: Delete(Flowtree, node)
5: **else if** (comp_pop[node] < thresh_comp_pop **and** pop[node] < thresh_pop) **then**
6: Delete(Flowtree, node)

Algorithm 3 Flowtree: Operators

Function: Merge(Flowtree 1, Flowtree 2)
1: Flowtree = Flowtree 1
2: **for each** node in Flowtree 2 **do**
3: Add_node(Flowtree 1, node)

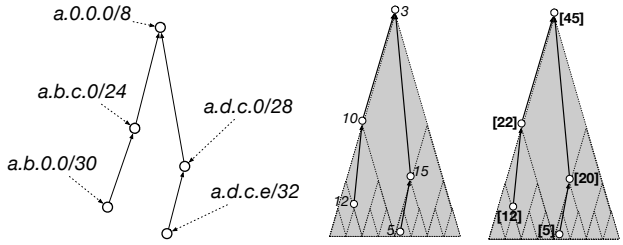
Function: Diff(Flowtree 1, Flowtree 2)
1: Flowtree = Merge(Flowtree 1,Flowtree 2)
2: **for each** node n in Flowtree 2 **do**
3: comp_pop(n) = abs(comp_pop(n) - 2*comp_pop2(n)).



(a) Node in Flowtree: 10. (b) Node not in Flowtree: 50 (c) Query: Above-29. Result see Boxes.

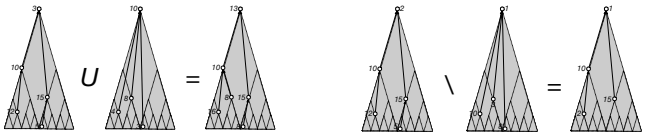
(a) Queries (b) Above-k

Figure 6: Flowtree queries.



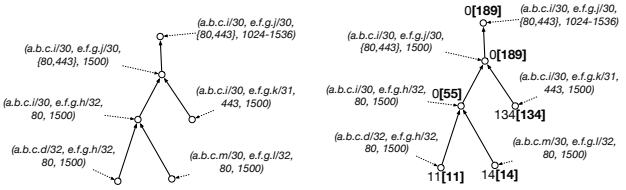
(a) 1-feature Flowtree: IP. (b) Comp_pop (c) Popularity

Figure 7: Flowtree concept.



(a) Merge (b) Diff

Figure 9: Flowtree Operators: Merge and Diff



(a) 4-feature Flowtree: src/dst IP and port.

(b) 4-feature Flowtree with complementary_pop. and pop.

Figure 8: 4-feature Flowtree.

start at node (a.b.0.0—16,80—8) and outputs only the nodes where src_port is 80 and src_ip is a subprefix of a.b/16. Drill-down queries retrieve the children of a node. Note that we can derive estimates for all flows, from mice to elephants: even for low-popularity nodes, the number of flows remains a good estimate for the number of contributing flows.

Above-t: Results in a tree-walk and all nodes whose popularity are above the threshold value are returned.

Top-k :

To compute the top-k, we identify the Flowtree entry with the largest popularity, delete its contribution, and then iterate. Hereby, we use a priority queue.

Merge: We merge two Flowtrees by adding the nodes of one to the other. Note that the update will only be done for the complementary popularities— see Alg. 3 and Fig. 9(a)—, with missing nodes being assigned a popularity of zero. The statistics have to be recomputed and, to reduce the memory footprint, we compress the joined tree. If the total absolute contributions of the two trees differ significantly, one should rescale the complementary popularities of the trees before merging.

Diff and HeavyChanger: Just as one can merge Flowtrees, one can also compute the difference between two trees. This is a merge operation with subtraction instead of addition— see Alg. 3 and Fig. 9(b). Heavy changers are detected by using Top-k on diff of the two trees.

Flowtrees maintain counters for various features of the flows. In the current implementation, we use counters for packet, byte and flow counts. This structure supports cardinality-based queries but is limited to the elements (features) already in the tree (nodes). It is possible to maintain additional counters and support additional cardinality-based queries, e.g., using counters for ports, but at the cost of requiring additional space. In some cases, this is necessary. For example, such cardinality-based queries will enable the detection of non-volumetric attacks, e.g., semantic attacks.

By allocating more space and maintaining more counters, it is possible to detect different types of attacks, e.g., "slow" DDoS attacks (Slowloris). We plan to explore the accuracy of cardinality based queries and the effect of allocating more space and maintaining more counters in Flowtrees as part of our future in future work.

V. FlowDB

FlowDB collects and stores Flowtree summaries computed by FlowAGG in persistent storage. Each Flowtree has a unique key that is made from its timestamp which along with its granularity reflects a time interval, the id of the site/location, and its feature-set. The values are the Flowtrees, which are stored as byte buffers. Figure 10 visualizes FlowDB's architecture.

A. FlowDB Implementation

Currently, our database of choice is MongoDB [84] because it is lightweight, although any other key-value datastore can be used. To accelerate query processing, we use an in-memory index and an in-memory cache. The in-memory index is a collection of T*-trees that track Flowtrees and enable range queries over different time periods. The in-memory cache uses a least recently used (LRU) policy to keep recently added or queried trees in memory. FlowDB is designed with parallelization in mind: it is capable of receiving multiple streams of Flowtrees from multiple FlowAGG daemons while answering queries to multiple users at the same time. Parallelization is employed in performing major tasks such as handling requests from FlowAGG daemons and remote API calls, storing Flowtrees in persistent storage, and query processing. Upon receiving a query, the system first checks whether the queried trees are in memory. In case of cache misses, it retrieves trees from storage.

The system is highly configurable in terms of memory usage, by setting a maximum number of Flowtrees in memory, cache eviction interval, degree of parallelization, etc. The maximum number of Flowtrees in memory controls the memory footprint of FlowDB. To access the database, FlowDB offers both an API with the services Add Flowtree and Get Flowtree and an interface for FlowQL. FlowAGG and other components of Flowyager use the Apache Thrift Remote Procedure Call (RPC) framework [85] for communication.

To enable *Geo-Distributed Query Execution*, the in-memory index keeps track of whether a Flowtree is stored locally or at a remote FlowDB. Thus, if necessary, all remote Flowtrees can be fetched via the FlowDB API to answer a FlowQL query. In our planned geo-distributed query execution, we partition site-IDs and map a site-ID to a FlowDB instance. Once a FlowDB instance receives a query, it will check whether the given site-ID is stored locally. If the required Flowtree is not stored locally, it can issue a request to the target FlowDB instance and retrieve the Flowtree. Once the Flowtree is retrieved, it will be merged with the Flowtrees that are already present and the intended query is fulfilled.

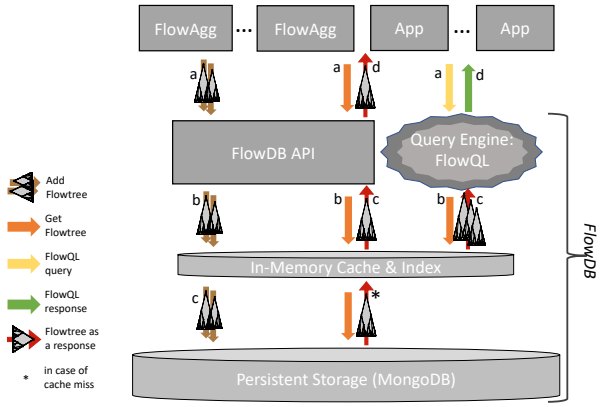


Figure 10: FlowDB overview.

The evaluation of this feature is beyond the scope of the current manuscript.

B. FlowQL Query Language

To realize FlowQL, we took inspiration from SQL keywords, yet we developed our own grammar. We used ANTLR [86] to generate the parser for the grammar. We offer an interactive command-line shell as well as a graphical user interface using R shiny [87]–cf. the screenshots from Fig 1 and Fig. 2. More specifically, with FlowQL the user chooses their operator via a `SELECT` clause, one or multiple time periods via a `FROM` clause, and the feature set via a `WHERE` clause.

SELECT: specifies the answer type. Allowed values include *'pop'* for popularity or flow/byte/packet count, *'top-K'* for the top-k most popular flows, *'HHH-P'* for the 1-d hierarchical heavy hitters with flow counts above P% of total traffic, *'hc-K'* for the top-k heavy changers, *'above-T'* for all flows with popularity above t, and '*' for all flows satisfying the WHERE clause.

FROM: specifies one or multiple time periods.

WHERE: selects the feature sets and one or multiple conditions. Possible feature elements are *site_id*, *src_ip*, *dst_ip*, *src_port*, *dst_port*, *proto*. Possible values are ANY or any region, IP prefix, or port range (using the IP—mask resp. the port—portmask syntax). Combinations are feasible via (AND, OR, and ()).

Thus, FlowQL queries have the following syntax:

```
SELECT [pop, top-k, hc-k, above-t, hhh-k, *]
FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm)+
WHERE ([Conditions via AND, OR, ), (, feature = value])+
```

Using FlowQL, we found that we often wanted to repeat the same query across multiple time bins or sites. Thus, we added two iterators: `answer-bin-x` that iterates across time bins of size x minutes and `site_id=ITR-x|n` that iterates across all sites within a site set, specified with an interval, e.g., $[x, x + 2^n - 1]$, or using a pattern.

To be able to drill-down and inspect a specific time-range in more detail, we additionally provide drill-down queries.

In a drill-down query, a particular granularity in which one desires to inspect the traffic should be specified. For instance, to see the result of a query in 15-minute time bins, one should specify *bin15* in the query.

C. Query Execution

Upon receiving a FlowQL query, first, the WHERE clause is converted into a Disjunctive Normal Form. This results in breaking down the current query into smaller queries, which we call *mini-queries*.

Each mini-query is then processed independently. For each mini-query, the corresponding trees are fetched considering the time-range, granularity, and feature sets. For instance, for a query requiring *src_port=X*, 1-feature trees, *SP* in this case, are fetched. In a non-drill-down query, trees with the highest granularity existing in FlowDB are fetched. For a drill-down query, trees with the granularity specified in the query are fetched. If the specified granularity does not exist in FlowDB, multiple lower-granularity trees are merged using the MERGE operator to build trees with the specified granularity. Consider the following query which asks for bin-30:

```
SELECT pop(any,byte,bin30) FROM (time
2018-05-09 00:00 to 2018-05-09 23:59) WHERE
site_id=ANY and src_port=X
```

This is a drill-down query to zoom into a full-day time-range in half-an-hour bins. Now assume that there are no 30-min granularity trees in FlowDB for the specified time-range, but there are 15-minute granularity trees. Then for each time-bin, two 15-minute trees will be merged to build the required granularity.

If the number of trees to be merged is large, the merge operation is performed in parallel to speed up the merge process. In a heavy changer query, two time-ranges should be provided and the trees fetched for each of these two time-ranges are diff'ed using the DIFF operator.

Then, the final trees are processed using different Flowtree operators to fulfill the query conditions, e.g. *src_port=X*. If the query is *pop*, knowing the popularity is as easy as finding the corresponding node in the tree and returning the popularity value. If the node is not in the tree, an estimation using the parent's popularity is returned as previously described in IV-D.

If the query is *above-T*, ABOVE-T operator with threshold T is used. For the *top-K* and *hhh-P*, the TOP-K operator will be used. In top-K, it should return the top K flows with any non-zero popularity. In hhh-P, P is the threshold for the fraction of total contributions.

VI. Experimental Deployments

We rolled out and tested Flowyager in three different types of networks, namely a large European IXP (IXP), a tier-1 ISP (ISP), and our testbed using a sample dataset (MAWI)—see Table III for an overview. In this paper, we report on experiments on stored data that we use for reproducibility. At two locations, the IXP and the ISP, we are in the process of moving towards live data import after extensive testing on site.

Dataset	Time range	#Interface	Input Size	Type	Time bin
IXP	Sep'19 1–7	≈ 1,250	≈ 10TB	Flow	15m
ISP	Apr'19 1–2	≈ 1,300	≈ 25TB	Flow	15m
MAWI	May'18 9–10	2	≈ 1TB	Packet	1m

Table III: Deployment overview: IXP, ISP, and MAWI.

Short Form	Meaning
SIDI	src IP and dst IP
SPDP	src port and dst port
SISP	src IP and src port
SIDP	src IP and dst port
DISP	dst IP and src port
DIDP	dst IP and dst port
SI	src IP
DI	dst IP
SP	src port
DP	dst port
FULL	src IP, dst IP, src port, and dst port

Table IV: Overview of the feature sets of Flowtree.

Ethical considerations: We are fully aware of the sensitivity of network data and, therefore, only work with a subset of the packet header information, namely src IP, dst IP, src port, dst port, protocol, whereby all IPs have been consistently anonymized per octet (bijective substitution using a hash function), even though this may negatively affect prefix aggregation. Note that the live operational deployment of Flowyager will not require such anonymization.

IXP Dataset: This dataset consists of IPFIX flow captures at one of the largest Internet Exchange Points (IXPs) in the world with more than 800 members and more than 8 Tbps peak traffic. The IPFIX flow captures are based on random sampling of 1 out of 10k packets that cross the IXP switching fabric. The anonymized capture includes information about the IP and transport layer headers, as well as packet and byte counts. To evaluate the system at real-world scales, we included all sites during the first week of September 2019. Each site corresponds to the router interface of an IXP member connected to the IXP's switching fabric.

We deployed Flowyager within a virtual machine (VM) on a server at the IXP's premises. The VM is assigned 400 GB of memory and 40 threads on a machine with two Intel-Xeon-gold 6148 CPUs each with 40 threads.

ISP Dataset: This dataset consists of approx. 1,300 NetFlow streams (one per interface) from a major tier-1 ISP. We receive NetFlow data from 40 routers located in 30 cities in 4 European countries, as well as the US. The ISP's internal systems preprocess the raw NetFlow streams into 26 separate ASCII data streams. The NetFlow packet sampling is identical across all the routers. We include all data from Apr. 01, 2019 (00:01:00 UTC) to Apr. 03, 2019 (02:01:00 UTC). We deployed Flowyager as a Docker container with 94 GB memory and 32 threads on a machine with two Intel

Xeon E5-2650 CPUs.

MAWI Dataset: This dataset consists of packet-level capture collected at the transit 1 Gbps link of the WIDE academic network to its upstream ISP on May 9-10, 2018. Each packet capture lasts for 15 mins and contains around 120 M packets. The anonymized trace is publicly available [88] and we use it to be able to release sample queries and results. We interpret each direction as a site. For this dataset, we deployed Flowyager on a testbed machine, with 128 AMD-EPYC 7601 CPUs and 1.5TB memory.

Flowyager setup: In terms of the basic setup for the Flowyager evaluation, we choose fixed time periods rather than a fixed number of flows. The advantage of the former is that we can easily summarize across time and that we can even look at coarser time granularities. The advantage of the latter is a constant number of entries to summarize. We choose the former rather than the latter as summarizing and investigating across time are typical network operator tasks. We keep Flowtrees for every 15 minutes for every site for the IXP and ISP datasets and 1 minute for the MAWI dataset. We generate 11 different feature trees, namely all four 1-feature trees, all six 2-feature trees, and a 4-feature tree, see Table IV for the details. By default, we limit each Flowtree to 40k nodes. 1-feature port Flowtrees are limited to 10k nodes. In addition, we generate aggregated trees for 15 minutes, 1 hour, 1 day, and 1-week time granularities, each with at most 40k nodes. This results in one tree per site for each time granularity and a single tree for all sites for each time granularity.

Big data analytics setup: We compare Flowyager's performance with *task-specific data-parallel Python scripts*, as well as installations of a prominent big data analytics platform, namely *Spark* [19], and a column-based state of the art database, namely *ClickHouse* [62]. Each installation was done on the same VM as Flowyager. Note that this implies that Spark was not deployed on a physical cluster of machines but in a multi-threaded environment.

VII. Flowyager Prototype Evaluation

Next, we describe our experience with deploying Flowyager, which we will make publicly available for non-commercial use. Our evaluation highlights the four main strengths of Flowyager: reduced storage footprint, low transfer cost, rapid response to a wide range of queries, and high accuracy. Since these characteristics are related to our choice of underlying data structure and its resp. parameters, we start by evaluating Flowtree— the current basis of Flowyager.

A. Flowtree Evaluation

Input data skewness: One motivation for using HHHs is to take advantage of the skewed input data. We indeed confirm that the flow captures are skewed in the sense that for all feature sets, all time periods, and all sites with enough traffic, the traffic volume follows a skewed distribution.

Next, the data structure should be able to summarize time periods with small as well as large numbers of flows as underlined by Figure 11, which shows the empirical cumulative distribution (ECDF) of the number of flow entries per 15-minute Flowtree for the IXP and the ISP datasets using a logarithmic x-axis. We find a huge skew. More than 37.5% of the time periods (per site) have less than 1,000 entries, yet more than 12.5% have more than 50k entries. This underlines that the data structure has to be very flexible to efficiently summarize time periods with many as well as few flows.

Flowtree creation time: Next, we focus on the worst-case runtime to generate Flowtrees, which, in part, depends on the deployed hardware[5]. We focus on one hour of data, the busy hour, for the largest site at the IXP and 15 minutes of data– again busy hour and largest site, for the ISP to get an upper bound on the runtime. Note that the data includes more than 6.5M flows that have to be processed. We compute Flowtrees for each 11 feature set while varying the maximum number of Flowtree nodes from 5k to 50k. We repeat the experiments 10 times and measure the runtime, in terms of wall time, for generating trees as reported by the C++ chrono library[6].

Figure 12 shows the 10th and 90th percentile of the tree creation times vs. the maximum number of Flowtree nodes. All runtimes are well below 15 seconds for 1-hour resp. 15 minutes input files; thus, even if we have to process flows from 1,000+ sites, the deployed hardware, with moderate parallelization, is sufficient for generating all 11-feature Flowtrees in real time. In the worst case we needed 20 min to process traces from all 1,000+ sites over one hour; that is, Flowtree would only not become a bottleneck if the throughput tripled and input from 1,000+ sites were to be processed. In that case, aggregating firs over different subsets of the flow space would be necessary. We notice different behavior for different features: The (destination IP, port) feature trees are very fast to compute, which can be explained by the fact that they exhibit the most skewed input distribution. The full (4-feature) trees take the longest—not surprising given that this feature combination potentially has the largest number of tree nodes.

We also notice that from one feature set to the next, the runtime sometimes decreases and sometimes increases as we increase the maximum number of tree nodes. The reasoning behind this surprising behavior is as follows. When the number of Flowtree nodes increases, while compressions happen less frequently, they take more time to run, given that they have to process a larger input. If the data is skewed, the increase of the compression runtime with the number of nodes is limited while the reduction in the average delay between two compressions is significant. Reversely, if the data is not less skewed, the increase in compression runtime outbalances the reduction in inter-compression delay.

[5]At the IXP we have Intel Xeon Gold 6148 CPUs; at the ISP we only have Xeon-E5-2650 CPUs

[6]We choose setup to similar to [23], [34] which also use wall time and preload the input data into memory.

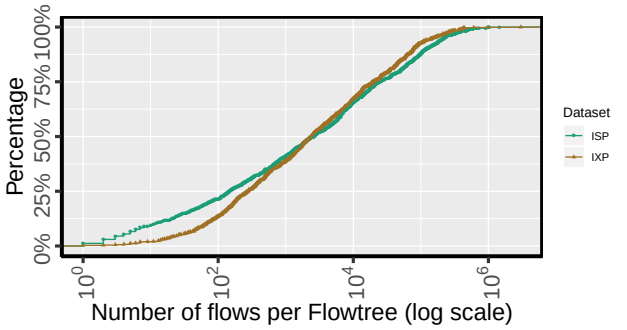


Figure 11: ECDF of # of entries–all sites (IXP and ISP).

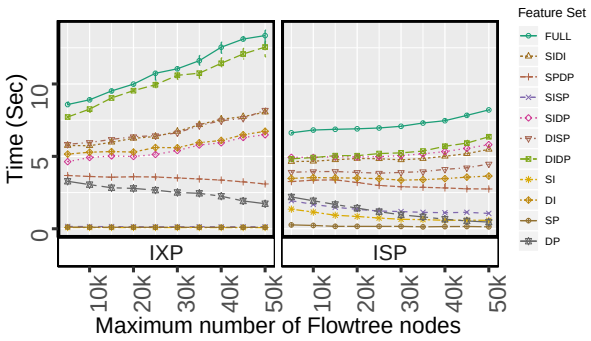
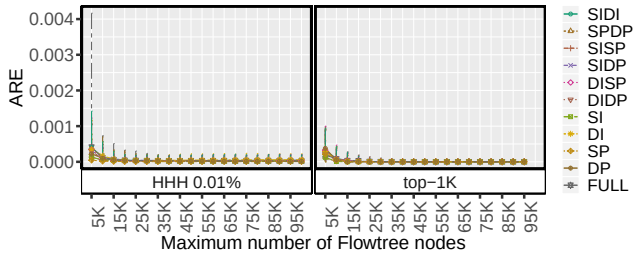
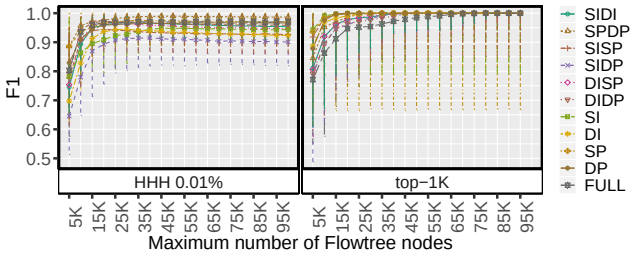


Figure 12: Flowtree build time (IXP/ISP: four/one 15-min. trees) vs. max. # of nodes per feature set.



(a) ARE vs. # of Flowtree nodes (all feature sets at IXP).



(b) $F1$-score vs. # of Flowtree nodes (all feature sets at IXP).

Figure 13: Accuracy of Flowtree for commonly-used queries (all feature sets at IXP).

Flowtree accuracy: Next, we look at the accuracy of the query results with focus on advanced queries, namely the 1-d HHH and top-K queries for Flowtrees with different featureset. Our metrics are the Average Relative Error, ARE, and the F_1 score. The ARE is the average of the ratios between the errors and the ground-truth values; that is, in our case, $\frac{1}{n}\sum_{i=1}^{n}\frac{|f_i-\hat{f}_i|}{f_i}$ with n the number of flows, f_i the flow popularity and $\hat{f}_i$ the estimated flow popularity. The F_1 score is the harmonic mean of precision and recall; accordingly, it accounts for both false positives and false negatives and ranges from 0 to 1—1 being the best value (perfect precision and recall) and 0 the worst. We calculate the ARE and the $F1$ score for the 1-d HHH and top-K queries, with thresholds of 0.01% and K=1000 respectively, for each 15-minute Flowtree and all sites over the IXP's busy hour, letting the maximum number of nodes in the Flowtrees vary from 5k to 100k. Note that we only evaluate the queries if a Flowtree summarizes at least 10k flows within the 15-minute time period since otherwise, the results would be a fraction of a flow, which does not exist. To generate the ground truth, we use a Flowtree with an unrestricted number of nodes. Finally, we only accept exact matches: in case of HHH, if a generalized flow f is in the actual heavy hitters it has to be returned by the HHH query; if the HHH query returns instead, a parent or child of f in the tree, this is a miss.

Figure 13(a) plots the median ARE values vs. the maximal number of nodes in the Flowtree and includes 10th and 90th percentiles as error bars in top-K and HHH queries. Our experiment shows that even for 10k Flowtrees the median ARE values are less than 0.0002 for all feature sets. Moreover, the main reason for ARE variations are flows with relatively small popularity.

The results for the $F1$ scores—see Figure 13(b) which shows the median together with the 10th and 90th percentile vs. the number of nodes per tree—confirm the excellent performance of Flowtree. Even for small trees, the median numbers are well above 0.9 for most feature sets. Moreover, the number of outliers is small.

Flowtree vs. RHHH: Next, we compare Flowtree to a state-of-the-art data structure, the constant time updates in hierarchical heavy hitters (RHHH) [23]. More precisely, RHHH is a randomized version of the deterministic HHH algorithm (dHHH) proposed by Mitzenmacher et al. [72]. RHHH has $O(1)$ update complexity, improving the $\Omega(H)$ update complexity of its deterministic counterpart, where H is the number of hierarchy levels.

While both Flowtree and RHHH take in the maximum node count as input, RHHH (and dHHH) have an additional input parameter: the HHH-threshold. The HHH-threshold determines if a frequent item is a heavy hitter, and, thus, if a node should be maintained in the tree. This complicates the usage of RHHH since neither the number of flows nor their popularity distribution is known in advance. Setting the threshold too high creates a very shallow tree with high aggregation, e.g., /16s and /8s, which does not keep enough

Data structure	1k	5k	10k	20k	40k
Flowtree	.19 (.31)	.77 (.92)	.92 (.99)	.98 (.99)	.99 (.99)
RHHH	.42 (.11)	.50 (.57)	.91 (.92)	. 92 (.94)	.93 (.95)

Table V: F1 score on top 1k src (dst) IPs for 1k, 5k, 10k, 20k, and 40k node Flowtree and RHHH trees.

detail. Setting the threshold too low may result in a tree with more nodes than the maximum node count. Indeed, we run into these limitations when executing the publicly available code [89] on the corresponding input. Hence, we evaluated the two systems under similar conditions, i.e., with the dataset that was used to evaluate RHHH [23] (CAIDA). The evaluation dataset comes from Equinix-Chicago trace of CAIDA [90]—this contains 20 Million packets (no sampling) from a 1Gbps link in the colocation facility in Chicago. In contrast, note that Flowtree is self-adjusting.

We used a number of metrics: (1) system runtime, (2) F1, on top 1k sources or destination of the input trace are present in the trees of 1k, 10k, 20k, and 40k nodes, (3) accuracy (ARE), i.e., how well the Flowtree or RHHH estimate the counters of the heavy hitters, either single IPs or aggregations.

The system runtime for creating RHHH trees is, as expected, quite constant: around 26 seconds. For Flowtree the time is higher, around 50 seconds, even as the number of nodes increases.

With regard to F1 score–identifying the correct set of heavy hitters–we find that if RHHH is not tuned, its performance is poor: very few of the IP heavy hitters are present and the trees are very small; the F1 of Flowtree is significantly better. Table V reports on the top-1k heavy hitter IPs indeed in the tree for Flowtree vs. RHHH with different total numbers of nodes(each time the threshold in RHHH is adjusted to produce 1k heavy hitters, i.e., the same output as Flowtree). For trees with up to 10k nodes, Flowtree includes a significantly larger number of heavy hitters than RHHH but beyond 10K nodes the differences get smaller.

Next, we turn our attention to the accuracy of the estimated values for each heavy hitter. We plot in Fig. 14 the estimated value using Flowtree (left) and RHHH (right) compared to the actual value for the 1k node trees—ARE on the top .1% IPs of 0.71 for Flowtree vs. 0.92 for RHHH.

The closer a point is to the diagonal the higher its accuracy. At first glance, RHHH might look better. However, it only contains a small subset of the relevant HHs as many top-1k entries are aggregated by RHHH. Thus, Flowtree again significantly outperforms RHHH.

Flowtree space saving: Given that we can compute Flowtrees efficiently and that they accurately answer 1-d HHH queries, we move on to study their space efficiency. Given the $F1$ scores and ARE values we, for the rest of this paper, choose 10k nodes for the 1-feature Flowtrees for src and dst ports and 40k nodes for all other feature combinations. (While 20k may be sufficient, using 40k does not increase the storage resp. communication overhead significantly, as we apply a final compress operation before using any Flowtree.)

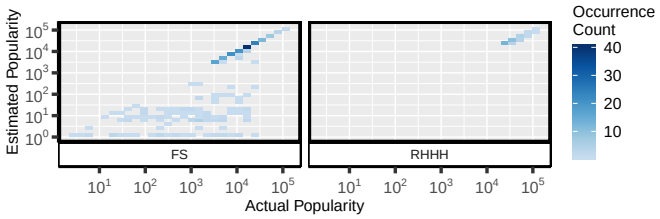


Figure 14: Comparison of estimated vs actual popularities using Flowtree (left) and RHHH (right).

To highlight the ability of Flowtree to compress its input, Figure 15(a) plots the ECDF of Flowtrees space saving ($1 - \frac{\texttt{\#nodes in tree}}{\texttt{\#input flows}}$) for all sites and all 15-minute time intervals. For almost all Flowtrees the space savings are well above 95%. This is also underlined by Figure 16(a) which shows the ECDF of the number of actual Flowtrees nodes. Note that a Flowtree will always contain less than 40k/10k nodes because we always run a final compression. Alternatively, it might simply happen that the data did not contain enough different feature combinations in the first place.

B. *Flowyager Evaluation*

Flowyager space efficiency: Given the above results regarding the capabilities of Flowtree, it is not surprising that Flowyager achieves excellent compression ratios. For the IXP (ISP), we see that compared to the original compressed IPFIX data (original compressed ASCII flow summaries), the single full-feature Flowtree in compressed binary format has a space saving of 97% resp. 99.5%. With additional feature sets, e.g., all 1-feature Flowtrees and three 2-feature Flowtrees, we still reach space saving of 92% resp. 97.5%. If we include all 11 possible feature combinations, the space saving is 89% resp. 96%. Even if we normalize not by the raw input data but only against the necessary features for the Flowtrees, the space savings are still excellent, e.g., more than 97% for the 1-feature Flowtree at the ISP. For a visualization of the space efficiency relative to the size of the raw compressed (gzip) input data, see Figure 17(a).

While 15-minute time granularity is excellent for answering detailed queries, many queries involve coarser time granularities. Thus, it can be useful to add time as another feature and add 1-hour as well as 1-day aggregated Flowtrees by merging (and then compressing) the smaller-time-granularity Flowtrees. Flowyager does so automatically. While this needs some extra memory, it adds less than 40% overhead—see Figure 18—while offering the potential to significantly reduce query response time. Moreover, should space become an issue, Flowyager may decide to permanently delete smaller-time aggregates while keeping higher-time aggregation summaries. This is one of the design features that enable resource management with Flowyager. It is always possible to still keep coarse grain summaries of

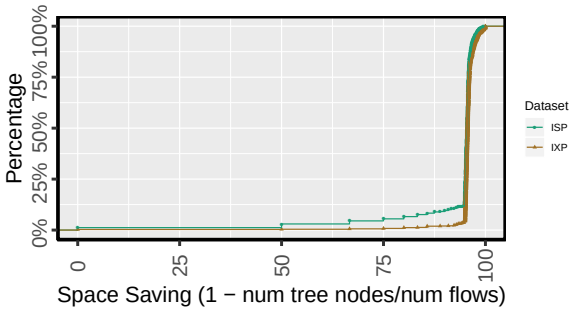


Figure 15: ECDF of space-saving for all Flowtrees (all time intervals/IXP sites)

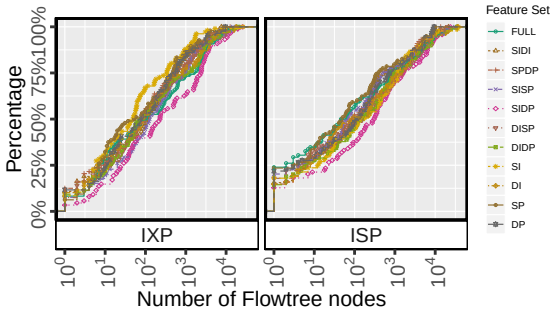


Figure 16: ECDF: # of Flowtree nodes (IXP and ISP).

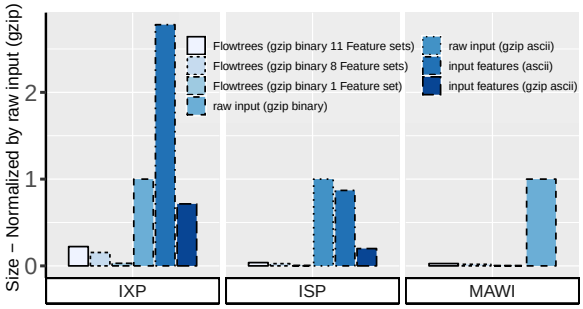
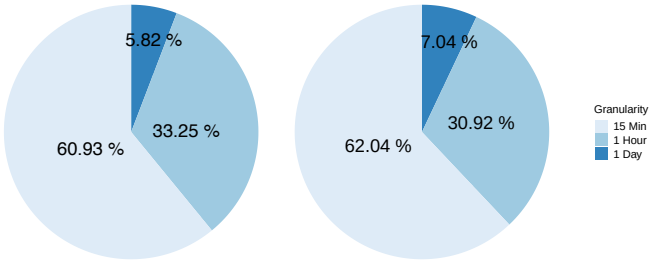


Figure 17: Space usage vs. raw compressed (gzip) input data.



(a) IXP (b) ISP

Figure 18: Pie Chart: MongoDB footprint.

previous time periods or site sets even if disk space is running out.

C. FlowQL Evaluation

Next, we focus on the performance (query response time) of the query capabilities and the query engine using a set of benchmark queries. In particular, we go back to the main tasks of a network manager—recall Table I—and pick a benchmark query for each of the identified tasks—note that the detection of one super-spreader requires two queries. These chosen queries are shown in Table VI, the table which thus contains queries for every single important network management task tackled by related work.

To challenge Flowyager, we task it to execute these queries for a full day for all sites in the IXP dataset. We evaluate three different ways of answering the queries using Flowtree, namely using FlowQL with Flowtrees and 15-minute, 1-hour, and 1-day aggregation. On the IXP machine, we execute each benchmark 10 times and measure, just as before, the wall time as reported by the C++ chrono library.

Figure 19 shows the resulting FlowQL query response times for each benchmark as boxplots. Hereby, we distinguish between cold and hot query response times. In the hot case, relevant Flowtrees may be retrieved from the in-memory cache. In the cold case, we restart the in-memory cache process for each benchmark. If we use the 1-day Flowtrees, see Figure 19(a), the answers are readily available and the response arrives in the blink of an eye (less than 1 second). By using the in-memory cache we speed up query response time by about 10 to 50%. We also check the accuracy of the results and find that the results are accurate[7].

With 1-hour trees, see Figure 19(b), the query response times typically increase by roughly a factor of 7, even though the number of Flowtrees that have to be processed increases by a factor of 24. This is possible as Flowyager takes advantage of parallelization. For Benchmark 5 the query response time is the worst as we have to execute an iterator across all 24 hours. Note, this is no principle limitation of the design of Flowyager but a limitation of the implementation which does not yet parallelize the iterators. If we move to 15-minute trees, see Figure 19(c), the query response time increases further up to a factor of eight. This highlights the efficiency obtained by using higher granularity trees in the design of Flowyager. Note, all benchmarks are executed using a research prototype rather than a production system.

Using an appropriate Flowtree granularity, we can answer all except one benchmark query in less than 5 seconds, underlining that Flowyager is indeed able to answer apriori unknown queries. This query response time enables interactive exploration of the data.

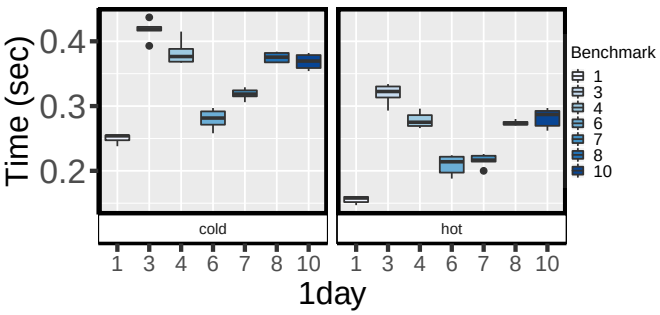
D. Flowyager vs. Possible Alternatives

Finally, we explore how well Flowyager performs compared to other systems. We picked three alternatives, namely, using (a) task-specific data-parallel Python scripts, (b) Spark [19]–a state of the art data analytics platform, and (c) ClickHouse [62]–a state of the art column database. We

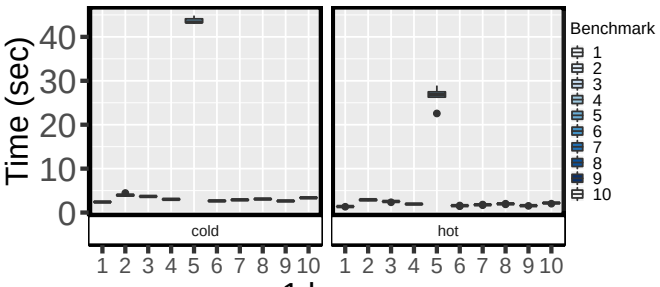
[7]We exclude Benchmarks 2, 5, and 9 as these benchmarks concern 60 min time-intervals and, thus, cannot be answered using data at 1-day granularity.

Table VI: Benchmark queries for Flowyager evaluation. Note that these queries correspond to those identified in Table I.

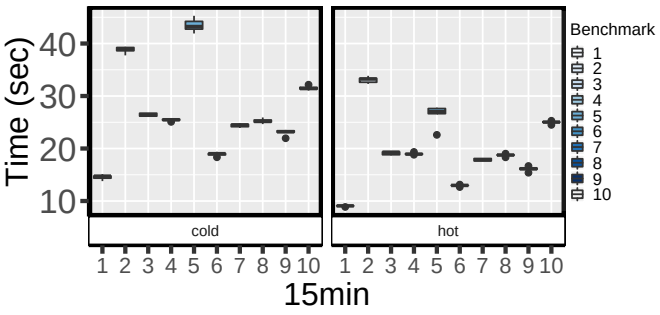
	Benchmark	Goal	Query
1	Aggregated flow statistics	Computing total traffic with specific features from IP/ports/time/location	SELECT pop(PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE (site_id = ANY and dst_ip = IP/mask and dst_port = port/portmask)
2	Counting traffic	Computing Traffic volume between given IP/Port subnet/addresses, for a specific site n	SELECT pop(PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE (site_id = n and src_ip = IP/mask)
3	Traffic flows	Displaying flows belonging to given subnets / IP addresses, passing through a specific site	SELECT *(PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE (site_id = n and src_ip = IP/mask)
4	Traffic matrix	Finding popular flows from a subnet to subnets for all sites	SELECT above(K,PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE (site_id = ANY and src_ip = ANY and dst_ip = ANY)
5	DDoS diagnosis	Finding the src IPs from which a dst IP (victim) has received abnormal traffic.	SELECT top(K,PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE site_id = ANY and dst_ip = [victim_ip]
6	Super-spreader Detection	Finding hosts that send packets to more than k unique dst during a time interval (requires multiple queries)	SELECT above(K,PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) where (site_id = ANY and src_ip = ANY) SELECT * FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) where (site_id = ANY and dst_ip = [pop_ip])
7	Top-k flows	Detect Top K flows in one or more sites , going to / coming from a specific subnet or IP address	SELECT top(K,PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE site_id = n and (src_ip = IP/mask or dst_ip = IP/mask)
8	Heavy Hitters	Detect all flows with popularity over threshold T, in one or more sites, going to / coming from a specific subnet or IP address	SELECT hhh(T,PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE (site_id = n and src_ip = IP/mask)
9	Heavy Changers Detection	Detect Top K heavily changed flows in one (or more) site(s).	SELECT hc(K,PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm)(time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE site_id = n
10	Full/4/5 tuple queries	Counting / Detecting flows belonging to a specific protocol/application	SELECT *(PROTO,COUNTMODE[,BIN]) FROM (time YYYY-MM-DD hh:mm to YYYY-MM-DD hh:mm) WHERE site_id = n



(a) 1-day Flowtrees (cold/hot cache)



(b) 1-hour Flowtrees (cold/hot cache)



(c) 15-minute Flowtrees (cold/hot cache)

Figure 19: IXP: Flowyager times (Table VI benchmarks).

evaluated all these systems on the same machine and dataset in IXP as previously described in VI.

First, we find that coding a custom python script for each benchmark takes a reasonably experienced programmer at least 2-3 hours for programming and debugging even if they can build upon a template from another benchmark. After all, it takes time to validate that the script is indeed doing what it is supposed to do. For some of the advanced tasks, e.g., the HHH, we did not start from scratch but rather included existing code. Nevertheless, this again did take additional time. Running the Python code on a day of data did take a mean of 39 minutes using a parallelization across 24 cores. Using 24 cores enables the script to parallelize the tasks by processing each hour of data in a separate process. Across all benchmarks, the Python code needed a minimum of 19 minutes and a maximum of 54 minutes.

Second, we find that setting up Spark and coding the queries require significant time. Indeed, it is necessary to first convert the data into a Spark-compatible format to get any reasonable performance (query response times less than 1 hour). This takes roughly 15.5 minutes per day of data for the IXP site. The resulting benchmark query response times are shown in Figure 20. Using this preprocessed data as input, the benchmark queries take a minimum of 20 seconds and up to 800 seconds. Note that for Benchmark 8 Spark only computes heavy hitters rather than HHH as implementing HHH on top of Spark is non-trivial. To measure the CPU usage and disk I/O usage of each Spark benchmark, we used the *iostat* command sampling every 5 seconds. In Figure 21, the x-axis shows the round, i.e. the 5-second period in which we

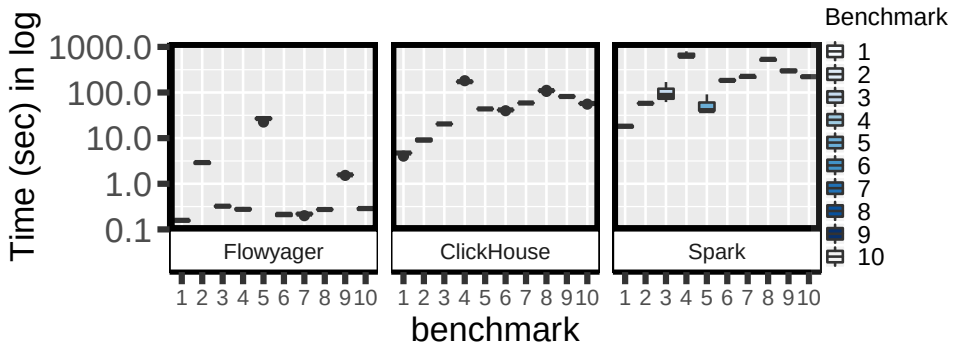


Figure 20: IXP: Query response time comparison: Flowyager vs. ClickHouse vs. Spark (Table VI benchmarks).

sample, and y-axis shows the utilization in percentage. CPU utilization is shown in square points, while the round points show the disk I/O. We observe that in the majority of the cases, Spark is bound by disk I/O rather than CPU. This holds for benchmarks 1, 4-10. However, benchmark 2 is a drill-down query and requires multiple GROUPBY statements. Also, benchmark 3 works with only two features. Hence, the intermediate results are not too big to require frequent disk access. Therefore, unlike other benchmarks, benchmark 2 and 3 are limited more by CPU capacity than disk I/O. Indeed, this figure highlights the significant overhead of query processing using only the raw data.

Third, we set up an instance of ClickHouse. Here, it is necessary to first load the data into the database. This takes roughly 45 minutes per day of IXP data. On the other hand, the resulting benchmark query response times are significantly smaller than those of Spark, see Figure 20. Again, ClickHouse only supports a limited version of the HHH query for Benchmark 8. Figure 20 also includes the Flowyager benchmark results from Section VII-C. Flowyager's benchmark performance supersedes all comparison systems.

E. Summary and Flowyager Limitations

Overall, Flowyager by far outperforms all three alternatives. Moreover, Flowyager is adaptive and supports HHH and physically distributed execution. We acknowledge that creating all Flowtrees does add some overhead–one day does take roughly 4 hours. However, this is a one-time operation, and overhead only matters if we consider archived data, but the Flowtrees can well be generated as the flow captures arrive, recall Section VII-A. Moreover, it is easy to do memory management within Flowyager; e.g., rather than purging older data, we can summarize it.

The limitation of Flowyager is that its answers are only estimates. However, these are accurate both for elephants and mice flows alike. Hereby, we want to point out that most network-wide systems anyhow rely on highly sampled flow captures. As such the fact that we "only" provide estimates does not increase the uncertainties dramatically. If higher accuracy is necessary, we recommend combining Flowyager for data exploration with ClickHouse for focused in-depth analysis. Moreover, the insights from Flowyager can be used to instantiate online non-sampled queries using streaming network telemetry systems, such as Sonata [16].

VIII. Use-Cases

In this section, we showcase how to use Flowyager for tackling typical network operator tasks.

Unveiling Application Trends: With Flowyager we can easily infer the 10 most popular applications within a time period using a top10 query with site_id=ANY and src_port=ANY:

```
SELECT top(10,any,byte) FROM (time
2018-05-09 00:00 to 2018-05-09 23:59) WHERE
site_id=ANY and src_port=ANY
```

To then see how the popularity of each top 10 port changed over time we use the query pop-bin60 for each port. Therefore, the query would be:

```
SELECT pop(any,byte,bin60) FROM (time
2018-05-09 00:00 to 2018-05-09 23:59) WHERE
site_id=ANY and src_port=X
```

See Figure 22(a) for the results for the MAWI dataset. We use the MAWI dataset for reproducibility as we will release the sample queries and their output along with the code. The query takes less than 1.4 seconds. Web and DNS related ports 80, 443, and 53 dominate. The same is true for the ISP. Still, during peak, other port numbers are prominent as well, e.g., port 3074. This port is used by Xbox LIVE and Games for Windows–Live. The peak traffic time also is the peak activity time for gaming, at least for residential customers of this Tier-1 ISP.

Traffic matrix: Computing a traffic matrix involves determining all src/dst pairs with a traffic volume larger than a value X. With Flowyager, one can use the above_t, for src_i=ANYp and dst_ip=ANY. Therefore, the following query can be used:

```
SELECT above(X,udp,byte) FROM (time
2018-05-09 00:00 to 2018-05-09 23:59) WHERE
site_id=ANY and src_ip=ANY and dst_ip=ANY
```

To highlight this capability we determine the src/dst traffic matrix for the MAWI data, see Figure 22(b). It shows

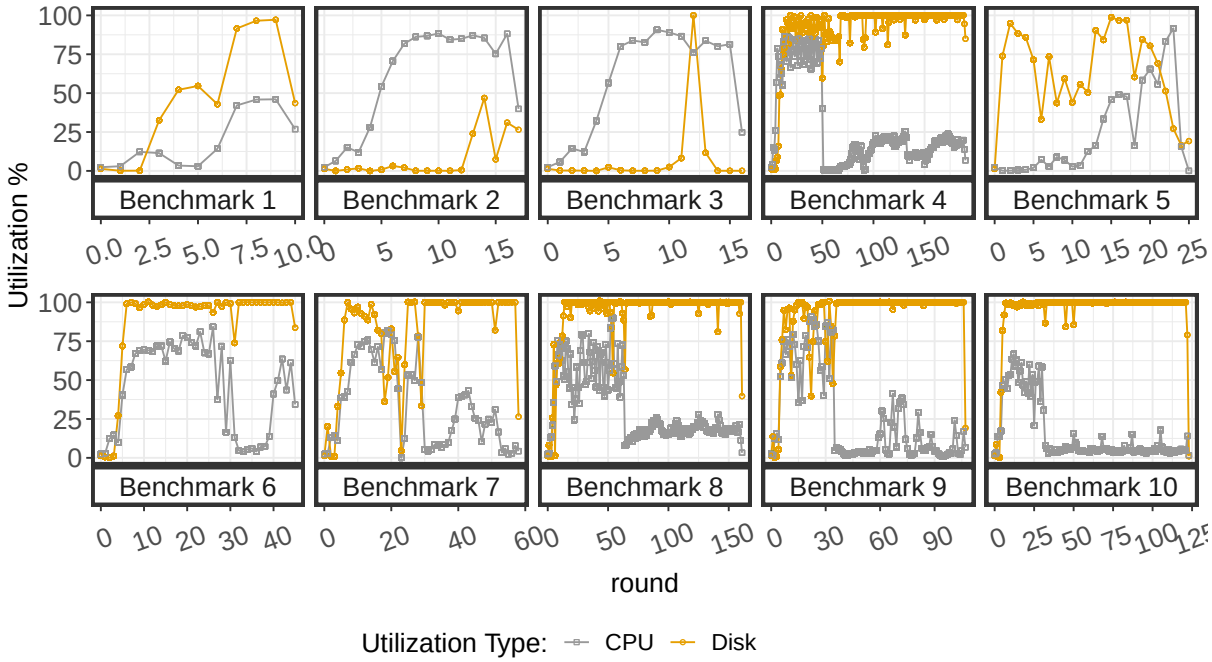
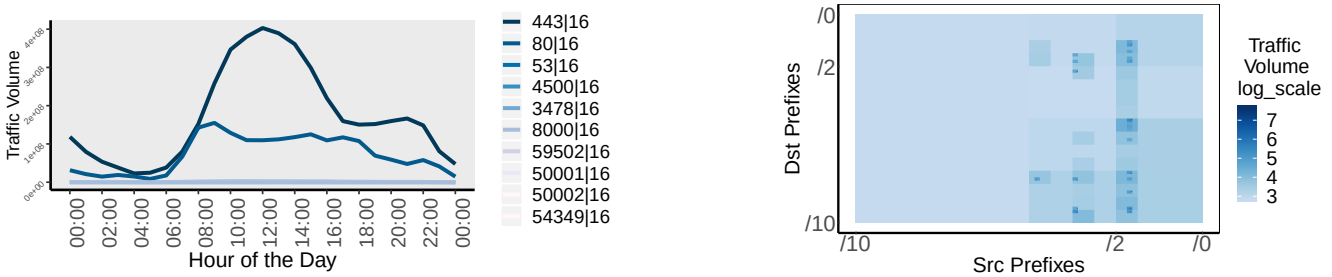


Figure 21: CPU and disk I/O usage in Spark experiments



(a) Application trends

(b) Traffic matrix.

Figure 22: MAWI: Data exploration.

(a) ISP: 123 port activity across 60 minute time bins

(b) ISP: 123 port activity across 15 minute time bins

Figure 23: ISP: DDoS NTP attack investigation.

the traffic matrix at different aggregation levels to detect which pairs of the source (src) and destination (dst) prefixes (at different granularity levels) are responsible for a large fraction of traffic exchange. For visualization, we use a two-dimensional heatmap where the x-axis corresponds to src IPs, the y-axis to dst IPs, and the color to the traffic volume normalized by the number of IPs within the area, i.e., traffic flowing from a src prefix to a dst prefix. This query took less than 13 seconds.

Investigating DDoS attacks: Network attacks, and in particular, distributed denial-of-service (DDoS) attacks are an ongoing nuisance for network operators as well as network users. A large body of research papers has focused on techniques for detecting DDoS attacks, see, e.g., [91], [92], [93], [94], including references and citations. Indeed, the multitude and the impact of DDoS attacks, see, e.g., [95], [96], have given rise to a variety of different mitigation techniques, see e.g., [97], [98]. Still, detecting DDoS attacks

reliably as well as diagnosing their root causes is critical for starting countermeasures or taking preventive future actions. Flowyager is an ideal system for tackling this challenge.

One of the most common signatures of DDoS attacks is a sudden rise in traffic for src/dst ports that are used within amplification attacks [99], [100], [95], [101]. Among such ports are 0, 123 (NTP), 11211 (memcached), 53 (DNS), and 1900 (SSDP), as discussed above. Potential DDoS attacks can be found by using the heavy changer query. It identifies time ranges during which they occurred. We execute these queries for each hour:

```
SELECT hc(100,any,byte) FROM (time
2019-04-01 00:00 to 2019-04-01 00:59)(time
2019-04-01 01:00 to 2019-04-01 01:59)
WHERE site_id=ITR and (dst_port=ANY or
src_port=ANY)
```

Per hour this takes less than 0.3 seconds. Among the heavy changers are high volume ports related to Web traffic, i.e., port 80, 443, as well as other ports where the volume can easily vary. But, we also find some unusual ports, i.e., 123 (NTP) which are known to be involved in DDoS attacks. Figure 23 shows a DDoS amplification attack in one of the sites of the ISP. This is a DDoS attack on NTP (port 123). Here, a very large number of src IPs scattered across multiple networks are involved but only a few dsts are targeted; namely two, whereby one of them receives more than 95% of the attack packets. It took us less than 5 minutes of human time and less than 1 minute computation time to find the attack for port 123, the site, the src of the attacks, and identify the start and the end of the attack. To illustrate the exploratory power of Flowyager, we identified the hours where the attack took place, see Figure 23(a), within a second. Then, we drill-down to the 15 minutes granularity to infer the start and end of the attack, see Figure 23(b), with a second query that took two seconds of execution time: `SELECT pop(any,byte,bin15) FROM (time 2019-04-01 01:00 to 2019-04-01 01:59) WHERE site_id=ITR and dst_port=123|16`
Note, detecting slowly increasing DDoS attacks needs a different approach. Here, a diff query to an earlier time period can be used as an indicator.

Towards real-time DDoS Mitigation: Using insights from historical analysis of DDoS attacks it is possible to use Flowyager also for near-live analysis if we keep recent Flowtrees at a shorter time granularity, e.g., 1-minute bins: we can then either use the above queries to monitor ports highly affected by DDoS attacks or we can use heavy-changer queries to look for ports with unusual activity. If we see such unusual activity, we can use the drill-down capabilities of Flowyager to check if, e.g., the traffic is targeted at specific IPs, i.e., only involves a small number of src or dst addresses, or involves spoofed addresses, i.e., a large number of IP addresses. If yes, Flowyager can be used to trigger an alarm which may then blackhole the attack traffic, e.g., using a system such as Stellar [97] or traffic scrubbing systems [96]. Recall that other techniques, e.g., telemetry, need to know a-priori the queries they have to execute. The power of Flowyager is that is can answer arbitrary queries that are not known in advance and using the already available network flow summaries supported by router vendors. Thus, Flowyager offers security capabilities that can help to identify arbitrary security issues. It can also help in generating the appropriate queries to execute them in real-time when, e.g., telemetry is used.

Lessons Learned: For our use cases neither the initial sampling in the flow captures nor the Flowyager estimates were detrimental to achieving the goal. However, we noticed some implementation challenges, e.g., handling flows from routers with unsynchronized clocks. We decided to use the timestamp when the flow is arriving at FlowAGG. Note that this may lead to some small amount of misbinning if the router is distant (in terms of network delay) from the aggregator. However, the impact is expected to be limited and probably well within the typical uncertainty of flow captures. Note that our approach even enables us to update Flowtrees of past time bins, should a significant number of flows arrive delayed.

Another observation is that one can tune Flowyager according to the needs of the users. Overall, we find that a query can be answered quickly if the aggregation level of the available (cached) Flowtrees matches the query granularity in terms of site sets and/or time granularity. The reason is that this avoids merging Flowtrees on the fly. Thus, if many queries involve the same subset of interfaces, e.g., per router, or all long-haul interfaces, it may make sense to store additional Flowtrees, if only temporarily. For example, keeping a Flowtree for all sites adds little overhead but speeds up queries significantly.

IX. Conclusion

Network flow captures are widely available and are essential for operators to monitor the health of their networks and steer their evolution. Yet, due to their ever-increasing size and complexity, their analysis is time-intensive and challenging. In the past, this has substantially hindered ad-hoc queries across multiple sites, for different time periods and over many network features. In this paper, we design, develop, and evaluate Flowyager, a system that allows exploration of network-wide data and answering ad-hoc a priori unknown queries within seconds. It achieves this using already existing network flow captures, without the need for specialized hardware, and without the need to compile specific queries into telemetry programs that should be known in advance and are slow to update.

Flowyager uses succinct summaries, Flowtrees, of raw flow captures and provides an SQL-like interface, FlowQL, that is easily usable by network engineers. We showcase the performance and accuracy of Flowyager in two operational settings: a large IXP and a tier-1 ISP. Our results show

that the query response time can be reduced by an order of magnitude, and, thus, Flowyager enables interactive network-wide queries and offers unprecedented drill-down capabilities to identify the culprits, pinpoint the involved sites, and determine the beginning and end of a network attack.

Acknowledgement

This work was supported in part by the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158) and by the German Ministry for Education and Research (BMBF) as BIFOLD - Berlin Institute for the Foundations of Learning and Data (01IS18025A, 01IS18037A).

References

[1] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow Monitoring Explained: From Packet Captureto Data Analysis With NetFlow and IPFIX," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, 2014.

[2] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft, "Structural Analysis of Network Traffic Flows," in *ACM SIGMETRICS*, 2005.

[3] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing Network-Wide Traffic Anomalies," in *ACM SIGCOMM*, 2004.

[4] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *ACM SIGCOMM*, 2002.

[5] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *ACM SIGCOMM*, 2004.

[6] C. Estan, S. Savage, and G. Varghese, "Automatically Inferring Patterns of Resource Consumption in Network Traffic," in *ACM SIGCOMM*, 2003.

[7] "TCPDUMP/LIBPCAP public repository," http://www.tcpdump.org/, 2019.

[8] N. Duffield, C. Lund, and M. Thorup, "Estimating Flow Distributions from Sampled Flow Statistics," in *ACM SIGCOMM*, 2003.

[9] B. Claise, "RFC 3954: Cisco Systems NetFlow Services Export Version 9," 2004.

[10] B. Claise, B. Trammell, and P. Aitken, "RFC 7011: Specification of the IPFIX Protocol for the Exchange of Flow Information," 2013.

[11] "InMon – bhuyan2015towards," 2019, http://sflow.org/.

[12] Cisco, "Introduction to Cisco IOS NetFlow - A Technical Overview," https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html, 2012.

[13] Juniper, "Exporting Flow Data Records (Juniper)," https://www.juniper.net/documentation/en_US/junos/topics/concept/services-exporting-version9-flow-data-collector.html, 2018.

[14] Alcater, "7750 SR," https://documentation.nokia.com/cgi-bin/dbaccessfilename.cgi/9300731102_V1_7750SROSRouteConfigurationGuide12.0.R4.pdf, 2014.

[15] Huawei, "Exporting Flow Data Records (Juniper)," https://actfornet.com/HUAWEI_ROUTER_DOCS/Router_All/Huawei_NE40_Product_Quick_Reference_Guide.pdf.

[16] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven Streaming Network Telemetry," in *ACM SIGCOMM*, 2018.

[17] O. Tilmans, T. Bühler, I. Poese, S. Vissicchio, and L. Vanbever, "Stroboscope: Declarative Network Monitoring on a Budget," *NSDI*, 2018.

[18] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-Directed Hardware Design forNetwork Performance Monitoring," in *ACM SIGCOMM*, 2017.

[19] M. Zaharia and M. Chowdhury and M. J. Franklin and S. Shenker and I. Stoica, "Spark: Cluster Computing with Working Sets," in *USENIX HotCloud*, 2010.

[20] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," in *ACM SIGMOD*, 2003.

[21] D. Sarlis, N. Papailiou, I. Konstantinou, G. Smaragdakis, and N. Koziris, "Datix: A system for scalable network analytics," *ACM CCR*, vol. 45, no. 5, 2015.

[22] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in data streams," in *VLDB*, 2003.

[23] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant Time Updates in Hierarchical Heavy Hitters," in *ACM SIGCOMM*, 2017.

[24] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM CCR*, vol. 41, no. 4, 2011.

[25] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y. C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[26] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, "Optimal elephant flow detection," in *IEEE INFOCOM*, 2017.

[27] Y. Da, Z. Yibo, A. Behnaz, F. Rodrigo, Z. Tianrong, D. Karl, and Y. Lihua, "dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces," in *NSDI*, 2019.

[28] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *Latin American Symposium on Theoretical Informatics*. Springer, 2004, pp. 29–38.

[29] G. Cormode and M. Hadjieleftheriou, "Finding Frequent Items in Data Streams," in *VLDB*, 2008.

[30] G. Cormode and S. Muthukrishnan, "Space efficient mining of multigraph streams," in *PODS*, 2005.

[31] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker, "Compiling Path Queries," in *NSDI*, 2016.

[32] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers." in *Nsdi*, 2016, pp. 311–324.

[33] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *ACM SIGCOMM*, 2018.

[34] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-wide Measurements," in *ACM SIGCOMM*, 2018.

[35] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *NSDI*, 2013.

[36] V. Bajpai and J. Schönwälder, "Network flow query language—design, implementation, performance, and applications," *IEEE TNSM*, vol. 14, no. 1, 2016.

[37] A. G. Prieto and R. Stadler, "A-GAP: An adaptive protocol for continuous network monitoring with accuracy objectives," *IEEE TNSM*, vol. 4, no. 1, 2007.

[38] J. Shuyuan and D. S. Yeung, "A covariance analysis model for DDoS attack detection," in *IEEE ICC*, 2004.

[39] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang, "Lads: Large-scale automated ddos detection system." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 171–184.

[40] S. M. Mousavi and M. St-Hilaire, "Early detection of DDoS attacks against SDN controllers," in *ICNC*, 2015.

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

[42] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M. Y. Kao, and G. Memik, "Reversible sketches: enabling monitoring and analysis over high-speed data streams," *IEEE/ACM Trans. Networking*, vol. 15, no. 5, 2007.

[43] L. Tang, Q. Huang, and P. P. Lee, "MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams," in *IEEE INFOCOM*, 2019.

[44] C. Graham and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," in *IEEE INFOCOM*, 2004.

[45] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *ACM OSDI*, 2016.

[46] ——, "Distributed network monitoring and debugging with switchpointer," in *NSDI*, 2018.

[47] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. Liu, J. Padhye, B. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *NSDI*, 2018.

[48] "Flowyager in Github," https://github.com/saidjawad/Flowyager.

[49] C. Labovitz, S. Lekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian, "Internet Inter-Domain Traffic," in *ACM SIGCOMM*, 2010.

[50] R. Caceres, N. Duffield, A. Feldmann, J. D. Friedmann, A. Greenberg, R. Greer, T. Johnson, C. R. Kalmanek, B. Krishnamurthy, D. Lavelle, P. P. Mishra, K. K. Ramakrishnan, J. Rexford, F. True, and J. E. van der Merwe, "Measurement and analysis of IP network usage and behavior," *IEEE Communications Magazine*, vol. 38, no. 5, 2000.

[51] European Union, "Data protection in the EU, The General Data Protection Regulation (GDPR); Regulation (EU) 2016/679," https://ec.europa.eu/info/law/law-topic/data-protection/, 2018.

[52] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *Usenix ATC*, 2011.

[53] A. Wundsam, A. Mehmood, A. Feldmann, and O. Maennel, "Network troubleshooting with mirror vnets," in *GLOBECOM Workshops*, 2010.

[54] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford, "Packetscope: Monitoring the packet lifecycle inside a switch," in *Proceedings of the Symposium on SDN Research*, 2020, pp. 76–82.

[55] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu, "Cheetah: Accelerating database queries with switch pruning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2407–2422.

[56] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "An incrementally-deployable P4-enabled architecture for network-wide heavy-hitter detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, 2020.

[57] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing heavy-hitter detection algorithms for programmable switches," *IEEE/ACM Transactions on Networking*, 2020.

[58] S. Pontarelli, R. Bifulco, M. B. C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Bianchi, "Flowblaze: Stateful packet processing in hardware," in *NSDI*, 2019.

[59] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *NDSS*, 2020.

[60] M. Yu, "Network telemetry: towards a top-down approach," *ACM CCR*, vol. 49, no. 1, 2019.

[61] Y. Lee and Y. Lee, "Toward Scalable Internet Traffic Measurement and Analysis with Hadoop," *ACM CCR*, vol. 43, no. 1, 2013.

[62] Yandex, "Clickhouse – open source distributed column-oriented dbms," https://clickhouse.yandex/, 2018.

[63] A. Vulimiri, C. Curino, B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *NSDI*, 2015.

[64] A. Vulimir, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a Geo-Distributed Data-Intensive Worl," in *CIDR*, 2015.

[65] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-Aware Optimization for Analytics Queries," in *NSDI*, 2016.

[66] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds," in *NSDI*, 2017.

[67] Y. Huang, Y. Shi, Z. Zhong, Y. Feng, J. Cheng, J. Li, H. Fan, C. Li, T. Guan, and J. Zhou, "Yugong: Geo-distributed data and job placement at scale," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2155–2169, 2019.

[68] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, "A survey on big data for network traffic monitoring and analysis," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 800–813, 2019.

[69] R. B. Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k, frequency, and volume estimation," *IEEE/ACM Transactions on Networking*, vol. 27, no. 4, pp. 1432–1445, 2019.

[70] R. Harrison, S. L. Feibish, A. Gupta, R. Teixeira, S. Muthukrishnan, and J. Rexford, "Carpe elephants: Seize the global heavy hitters," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 15–21.

[71] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-Dimensional Data," in *ACM SIGMOD*, 2004.

[72] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical Heavy Hitters with the Space Saving Algorithm," in *ALENEX*, 2012.

[73] G. Cormode and S. Muthukrishnan, "What's new: Finding significant differences in network data streams," *IEEE/ACM Trans. Networking*, vol. 13, no. 6, 2005.

[74] N. Ivkin, R. B. Basat, Z. Liu, G. Einziger, R. Friedman, and V. Braverman, "I know what you did last summer: Network monitoring using interval queries," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, 2019.

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

[76] T. Wellem, Y.-K. Lai, C.-Y. Huang, and W.-Y. Chung, "A flexible sketch-based network traffic monitoring infrastructure," *IEEE Access*, vol. 7, 2019.

[77] H. Wang, H. Xu, L. Huang, and Y. Zhai, "Fast and accurate traffic measurement with hierarchical filtering," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2360–2374, 2020.

[78] T. Repantis, J. Cohen, S. Smith, and J. Wein, "Scaling a Monitoring Infrastructure for the Akamai Network," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, 2010.

[79] J. Cohen, T. Repantis, S. McDermott, S. Smith, and J. Wein, "Keeping Track of 70,000+ Servers: The Akamai Query System," in *USENIX LISA*, 2010.

[80] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger, "A Methodology for Studying Persistency Aspects of Internet Flows," *ACM CCR*, vol. 35, no. 2, Apr 2005.

[81] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates," in *ACM CCR*, vol. 32, no. 4, 2002.

[82] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *IEEE INFOCOM*, 1999.

[83] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding Hierarchical Heavy Hitters in Streaming Data," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 4, 2008.

[84] "mongoDB: The database for modern applications," https://www.mongodb.com/, 2019.

[85] "Apache Thrift," https://thrift.apache.org/, 2019.

[86] "ANTLR (ANother Tool for Language Recognition)," https://www.antlr.org/, 2019.

[87] RStudio, Inc, *Easy Web applications in R*, http://www.rstudio.com/shiny/, 2013.

[88] "MAWI Working Group Traffic Archive," http://mawi.wide.ad.jp/mawi/, 2018.

[89] B. B. Ran, "Implementation of the Constant Time Updates in Hierarchical Heavy Hitters paper, ACM SIGCOMM 2017," https://github.com/ranbenbasat/RHHH, 2018.

[90] CAIDA, "The CAIDA UCSD Passive Monitor: Equinix-Chicago - 2016-02-18," https://www.caida.org/data/monitors/passive-equinix-chicago.xml, 2018.

[91] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *IEEE communications surveys & tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[92] G. Carl, G. Kesidis, R. R. Brooks, and S. Rai, "Denial-of-service attack-detection techniques," *IEEE Internet computing*, vol. 10, no. 1, pp. 82–89, 2006.

[93] C. Douligeris and A. Mitrokotsa, "Ddos attacks and defense mechanisms: classification and state-of-the-art," *Computer Networks*, vol. 44, no. 5, pp. 643–666, 2004.

[94] K. Lee, J. Kim, K. H. Kwon, Y. Han, and S. Kim, "Ddos attack detection method using cluster analysis," *Expert systems with applications*, vol. 34, no. 3, pp. 1659–1665, 2008.

[95] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir, "Taming the 800 Pound Gorilla: The Rise and Decline of NTP DDoS Attacks," in *ACM IMC*, 2014.

[96] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti, "Millions of Targets Under Attack: A Macroscopic Characterization of the DoS Ecosystem," in *ACM IMC*, 2017.

[97] C. Dietzel, G. Smaragdakis, M. Wichtlhuber, and A. Feldmann, "Stellar: network attack mitigation using advanced blackholing," in *ACM CoNEXT*, 2018.

[98] M. Jonker, A. Sperotto, R. van Rijswijk-Deij, R. Sadre, and A. Pras, "Measuring the Adoption of DDoS Protection Services," in *ACM IMC*, 2016.

[99] "US-Cert: Alert (TA14-017A), UDP-Based Amplification Attacks," https://www.us-cert.gov/ncas/alerts/TA14-017A, 2019.

[100] Akamai, "State of the Internet Security Report (Attack Spotlight: Memcached)," https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-summer-2018-attack-spotlight.pdf, 2018.
[101] C. Rossow, "Amplification Hell: Revisiting Network Protocols for DDoS Abuse," *NDSS*, 2014.

Said Jawad Saidi obtained his M.Sc. degree in computer science from Technische Universität Berlin in 2016. He is currently a Ph.D. student at Internet Architecture research group led by Prof. Anja Feldmann, at Max Planck Institute for Informatics, Saarbrücken, Germany. His research area involves Internet of Things(IoT), Internet Measurement as well as design and study of systems and measurement methodologies to enhance the observability of Internet.

Aniss Maghsoudlou is a Ph.D. student at Internet Architecture group at Max Planck Institute for Informatics. She has been working on several projects on network measurement, to investigate how internet traffic really looks like and to simplify management of the large scale traffic in ISPs and IXPs. She has also worked on Software-defined WLANs as her Master's thesis at Sharif University of Technology.

Damien Foucard is a Ph.D. student at TU Berlin, in the Open Distributed Systems (ODS) research group, led by Prof. Manfred Hauswirth, which he joined in 2018. From 2015-2018 Damien was a Ph.D. student in the Intelligent Network research group (INET), led by Prof. Anja Feldmann, during which he collaborated with EPFL (Switzerland) in 2016. He obtained a Master's diploma from Centrale-Supelec (France) in Engineering and a Master's diploma in Computer Science from TU Berlin (Germany), both in 2015. His research centers around online optimization of data structures by combining machine learning and statistical guarantees. As a teaching assistant, he has led the organization of and given lectures for classes of up to thousand students. He also supervised successfully dozens of theses and projects.

Georgios Smaragdakis is a Professor with Technical University (TU) Berlin, heading the Chair of Internet Measurement and Analysis. He is also a research affiliate with Max Planck Institute for Informatics and a research collaborator with Akamai Technologies. From 2014-2017 he was a Marie Curie fellow at the Massachusetts Institute of Technology (MIT) Computer Science and Artificial Intelligence Laboratory (CSAIL), and from 2015-2018 a research affiliate with the MIT Internet Policy Research Initiative (IPRI). From 2008-2014 he acted as Senior Researcher at Deutsche Telekom Laboratories and TU Berlin. In 2008 he was a research intern at Telefonica Research. He earned a Ph.D. degree in Computer Science from Boston University in 2009 and a Diploma in Electronic and Computer Engineering from the Technical University of Crete. His research brings a data- and measurement-driven approach to the study of the Internet's state, resilience, and performance, as well as to the enhancement of Web privacy. His research was recognized with a European Research Council (ERC) Starting Grant Award in 2015, a Marie Curie International Outgoing Fellowship in 2013, best paper awards at ACM Internet Measurement Conference (2011, 2016, and 2018), ACM CoNEXT (2015 and 2019), and IEEE INFOCOM in 2017, two IETF/IRTF Applied Networking Research Prizes (2019 and 2020), and was selected as best of ACM SIGCOMM Computer Communication Review in 2019.

Ingmar Poese is co-founder and CTO of BENOCS. He focuses his work on Large Scale Networks, System Design, Internet Measurement and Data Analytics. He obtained his Ph.D. from Technische Universität Berlin in 2013 and as well as his M.Sc. degree in 2009.

Anja Feldmann received her Master's at Universität Paderborn in Germany and her Ph.D. at Carnegie Mellon University. In the next four years she did research work at AT&T Labs Research, before taking professor positions at Saarland University, the TU Munich, and the TU Berlin. Since the beginning of 2018, Anja is a director at the Max Planck Institute for Informatics in Saarbrücken, Germany. Her current research interests include Internet measurement, traffic engineering and traffic characterization, network performance debugging, and network architecture. She was Co-Chair of ACM SIGCOMM 2003 and ACM IMC 2011 and Co-PC-Chair of ACM CoNext 2020, ACM SIGCOMM 2007, ACM IMC 2009, and ACM HotNets 2014.