PipHeap: Approximate Heap in the Pipeline Empowering Network Measurement

Yuhan Wu*
Peking University
Beijing, China
yuhan.wu@pku.edu.cn

Fenghao Dong*
Peking University
Beijing, China
dfh@pku.edu.cn

Qizhi Chen*
Peking University
Beijing, China
hzyoi@pku.edu.cn

Aomufei Yuan
Peking University
Beijing, China
yuanaomufei@gmail.com

Kaicheng Yang
Peking University
Beijing, China
ykc@pku.edu.cnn

Hanglong Lv
Peking University
Beijing, China
lyuhanglong@stu.pku.edu.cn

Tong Yang†
Peking University
Beijing, China
yangtongemail@gmail.com

Wenrui Liu
Peking University
Beijing, China
liuwenrui@pku.edu.cn

Gaogang Xie
Chinese Academy of Sciences
Computer Network Information Center
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
xie@cnic.cn

ABSTRACT

Network telemetry has seen an increasing trend of deploying approximate measurement algorithms (e.g., sketches) on programmable switches due to their ability to provide line-rate speed, high measurement accuracy, and low memory cost. Heap, a vital component of many of measurement algorithms, hinders their deployment because of the difficulties in incorporating it into pipelines. In this paper, we introduce PipHeap, a pipeline-friendly, binary-tree-based min heap that can enhance existing sketches without introducing additional errors. Through evaluation with real-world datasets, we demonstrate that PipHeap can reduce the error of these integrated algorithms by 33% to 97% (78% on average) under the same memory allocation. We have successfully implemented PipHeap and its combination with six different sketches in our testbed, and successfully extended other approximate algorithms (e.g. Space-Saving) onto programmable switch platforms. We have made all code associated available as open-source.

*Co-first author.
†Corresponding author: Tong Yang (yangtongemail@gmail.com)

IMC '25, October 28–31, 2025, Madison, WI, USA.

ACM ISBN 979-8-4007-1860-1/25/10
https://doi.org/10.1145/3730567.3764487

CCS CONCEPTS

• **Theory of computation** → *Sketching and sampling*; • **Information systems** → **Heap (data structure)**; • **Networks** → *Programmable networks.*

KEYWORDS

data streams; heaps; sketches; programmable switches

ACM Reference Format:
Yuhan Wu, Fenghao Dong, Qizhi Chen, Aomufei Yuan, Kaicheng Yang, Hanglong Lv, Tong Yang, Wenrui Liu, and Gaogang Xie. 2025. PipHeap: Approximate Heap in the Pipeline Empowering Network Measurement. In *Proceedings of the 2025 ACM Internet Measurement Conference (IMC '25), October 28–31, 2025, Madison, WI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3730567.3764487

1 INTRODUCTION

Network measurement, which estimates various traffic statistics, serves as a crucial initial step for numerous network functions including flow scheduling, congestion control, load balancing, traffic engineering, and anomaly detection [8, 11, 12, 16, 26, 41, 42, 47, 60, 69, 75]. Many of these functions rely heavily on the detection of heavy hitters. As such, many approximate measurement algorithms have been developed for traffic statistics, including per-flow size, heavy hitters, flow size distribution, and others [44, 51, 71]. These algorithms specialize in providing high-fidelity measurement results while efficiently processing high-speed network traffic with limited resources.

Existing algorithms can be broadly categorized into two types: Heap-based and Sketch-based. In both categories, the heap plays a vital role, either as an integral component of the algorithms or as a tool to facilitate measurement tasks.

- **Heap-based algorithms**, renowned for their ability to accurately report elephant flows, leverage a min-heap binary tree structure and include examples such as Space Saving [46], Frequent [22], Unbiased Space Saving [59], and others [48, 64, 67, 72]. These algorithms consist of a heap of ID-counter pairs that record flow IDs and flow sizes of elephant flows. Upon a packet's arrival, it will either update the pair with the identical flow ID or replace one ID-counter pair with a certain replacement strategy. Because these algorithms record the flow ID of elephant flows, they usually have favorable algorithmic properties (*e.g.*, high precision rate in estimating elephant flow sizes). However, the heap structure is challenging to implement in modern high-throughput network switches based on the pipeline architecture. We will discuss this issue in detail later and attempt to overcome it.
- **Sketches** have emerged as a promising solution for measurements on switches, offering line-rate performance while utilizing limited memory resources. Typical sketches include Count-Min (CM) [21], CU [25], Count [17], UnivMon [44], Nitro [43], and more [18, 31, 37, 45, 65, 68, 70, 74]. Sketches consist of multiple arrays of counters and do not record flow ID (*i.e.*, five-tuple) or fingerprint. This structure is inherently suited for deployment on pipelines. They support various measurements, including flow size, cardinality, entropy, etc. However, for tasks that involve reporting flow IDs, such as identifying heavy hitters, detecting packet losses, and measuring jitters, sketches necessitate an additional structure — ideally, a heap — to record those IDs. A substantial body of literature [17, 21, 53, 66] suggests utilizing a heap-sketch hybrid data structure comprising a heap (or a similar structure) followed by a sketch, where elephant flows and their flow IDs are stored in the heap, and smaller flows are inserted into the sketch. Thanks to the efficient collaboration between heap and sketch, this hybrid structure is capable of delivering higher measurement accuracy and accommodating various tasks.

Recently, with their deployment on the high-throughput pipeline-based programmable switches [13, 14, 51, 61], sketches can now measure Tbps-level network traffic. However, the more accurate heap-based algorithms and the heap-sketch hybrids cannot achieve such a high speed due to the difficulties of implementing heaps within the pipeline architecture. To be specific, the data plane of the programmable switch is a pipeline that each stage has disjoint memory space. Each packet can only pass through each stage sequentially and access a few memory address in each stage, *e.g.*, up to 4 in a Tofino [2] switch. The pipeline is naturally suitable for the implementation of sketches because they independently update multiple counter arrays upon a packet's arrival. However, it is challenging to implement a binary-tree-based heap that requires to exchange two nodes, because such exchange requires to write the nodes in the latter stage back to the former stage, violating the restriction of accessing stages in sequence. Our **design goal** is to develop a pipeline-friendly heap on programmable switches for tracking elephant flows, facilitating the deployment of heap-based algorithms and heap-sketch hybrids to boost high-accuracy and high-speed measurement.

To the best of our knowledge, no prior art can implement our desired heap or achieve similar functions with reasonable cost. Prior art bypasses the heap and tracks the elephant flows by other data structures. These solutions have an unfavorable trade-off in either degraded switch throughput, high error, or high network overhead. For example, Qpipe [33] and Precision [9] (A practical HashPipe [57]) rely on the recirculation, a technique that allows a packet to traverse the pipeline multiple times, which in turn leads to degraded throughput. We will discuss more about the limitations of recirculation methods later in Section 2.4. The hardware version of Elastic sketch [66] can track elephant flows in the data plane without recirculation, but it comes with substantial errors, *e.g.*, resetting the size of evicted flows to 1 and passing them to the following stages. The others rely on either control plane CPU [34, 42, 44] or end hosts [32].

As the summary of above, for network data stream measurement tasks, pure sketch and recirculation-based methods can be relatively easily implemented on programmable switch pipelines. However, they suffer from limitations such as limited functionality, low accuracy, and suboptimal throughput and latency performance. In contrast, other approaches, including heap-based and heap-sketch hybrid methods, can address these performance issues, but must confront another outstanding challenge: how to effectively implement a pipeline-friendly heap structure on the data plane.

In this paper, we present the first approximate heap (called PipHeap) for tracking elephant flows entirely on the data plane without requiring recirculation. We integrate PipHeap with six sketches and our evaluation on the real-world dataset reveals that errors can be reduced by 33% ~ 97% (78% on average) in the same memory.

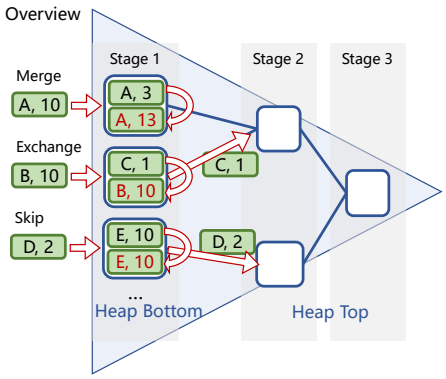


Figure 1: An overview of PipHeap. We show an example of PipHeap. When inserting a new flow to one stage, there can be three possible cases: (1) Merge. Flow A merges with the stored A. (2) Exchange. Flow B kicks out the Flow C, and C will go to the next stage. (3) Skip. Flow D skips the stage.

Overall Design: To approximate the min heap—a binary tree where each node stores one flow and its size, and larger flows reside closer to leaf nodes—we develop PipHeap as a binary tree structure as shown in Figure 1. Similar to the min heap that starts insertion from the bottom stage, insertion into PipHeap also begins from the bottom. As a result, we map all heap leaf nodes to the first pipeline stage, and nodes from each higher layer to the subsequent stage. To insert a new flow packet, we sequentially traverse a path from the leaf node (first pipeline stage) up to the root (final stage). In each accessed node, if the new flow is the same as the old stored one, we just merge their flow size and accomplish the insertion. Otherwise,

we should decide whether to exchange the two flows. If so, we will store the new flow in current node and try to insert the old one along the path later. Otherwise, we just skip this node and insert the new flow later.

We confront two technical challenges: *(1) Decision-making for exchange.* Due to the inability to perform reverse data movement within the pipeline, a decision to swap has to be made immediately after a new flow passes through a stage. The quality of this decision-making will ultimately affect the heap's ability to track elephant flows. To address this, we devise the asymmetric majority algorithm to effectively identify the most probable elephant flow. The core of this algorithm is to assign different accumulative weight for matched and not matched flows, which will be detailed in Section 3.2.1. *(2) Constraints for single node deployment.* On the Tofino switch, each pipeline stage (or more precisely, SALU) can only record two fields and read/output one field out at a time. Placing the flow key and size within the same stage prevents us from reading both fields when exchanging two flows. Conversely, placing them in distinct stages creates a prohibited data dependency, *i.e.*, the update result of any one field relies upon the other. We address the problem by incorporating a new field, `Assist`, in the first stage with the key, and place the value (flow size) in the subsequent stage. This arrangement ensures that only one field is read per stage while avoiding data dependency. We have two types of nodes in total, which will be detailed in Section 3.2 and Section 3.3, respectively.

We summarise our key contribution as follows: (1) We design the first approximate heap, PipHeap, entirely on the data plane. (2) We implement PipHeap combined with six sketch algorithms in real programmable switches. (3) Our evaluation shows that PipHeap can reduce the error of existing algorithms by 33% to 97% (78% on average) under the same memory allocation.

2 BACKGROUND AND MOTIVATION

We aim at an approximate heap that can benefit many sketches and solve many network measurement tasks. In this section, we first introduce the measurement tasks we focus on, followed by their solutions desiring for a heap. Then we discuss the challenges of implementing a heap on programmable switch. Finally, we take recirculation-based approaches as a counterpart of existing methods, to show the value and significance of implementing a heap on the data plane.

2.1 Measurement Tasks

We focus on common measurement tasks on the switch and we classify them into three types. (1) **Single key query** includes the flow size estimation and heavy hitter detection. Given a flow key (identified by the five-tuple or other fields), the flow size estimation queries the corresponding flow size that can be counted by packets or Bytes within a time window[1]. The heavy hitter detection is to find all flows whose size exceeds a threshold and report their flow keys. (2) **Hierarchical key query** refers to the Hierarchical heavy hitter detection [10], which finds the set of flows with a common IP address prefix that have a large total flow size (*i.e.*, greater than a threshold). (3) **Arbitrary partial key query [71].** Instead of defining the flow key before network traffic arrives, this query can define the flow key after the end of the traffic. The defined flow key can be any sub-part of a pre-defined broad key range. For example, when the pre-defined key range is the five-tuple, the user can query the total flow size from one subnet/IP-prefix to another subnet/IP-prefix.

2.2 Measurement Solutions and the Heap

At a high level, the measurement solutions using heaps are with the best theoretical properties, but the heap cannot be implemented on the data plane. The most recent example is that the CoCo [71] in SIGCOMM 2021 tried to approximately implement the heap-based Unbiased Space Saving [59] on the data plane, because USS is the ideal solution for arbitrary partial key query regardless of implementation challenges. Similar for single key query, heap-based Frequent and SpaceSaving[2] have the best fidelity on the results, but cannot be implemented on the data plane. For hierarchical key query, the milestone solution Randomized Hierarchical Heavy Hitters (RHHH) [10], which achieves constant update time first, relies on the SpaceSaving and heap. In addition, there are many solutions relying on heap have their own strengths, including Count-Min-Heap [21], Augmented [53], Heavy Keeper [67], Active Keeper [63] and more [56, 64].

In almost all the solutions previously discussed, a min-heap is used tailored for tracking heavy hitters. The data structure of the heap is an almost complete binary tree [3], and each node stores one key together with it value. In network measurement scenarios, the key is the flow key (*e.g.*, a five-tuple) and the value is the flow size. The value of a node must be less than or equal to the values of its children. The top of the heap (*i.e.*, the root of the tree) is with the smallest value. ***Basic insertion.*** When inserting a new element (*i.e.*, a flow with its size) whose key does not appear in the heap, we add the element and try to adjust the heap. Specifically, first, we add the new element as a new node to the last layer of the heap/tree (add a new layer if there is no space) as a leaf node. Then we compare the value of the element with the value of its parent node. If the parent node has a greater value, exchange the contents of the two nodes including keys and values. The exchange operation is repeated until the value of the parent is not larger than the new element. ***Index-and-Modify.*** It is possible that the key to be inserted already exists in the heap and we need to update its value. For example, a packet arrives and its flow size should be updated. However, the basic heap does not support indexing a key or modifying the value of any node, called Index-and-Modify. The desired practical heap in a network scenario requires Index-and-Modify. ***Find-min.*** Report the top of the heap as an element with the minimal value.

2.3 Challenges of Heap Implementation

We introduce the challenges of implementing heap on programmable switch, including a brief summary of the restrictions

[1]When a measurement scheme works on a switch, the timeline is divided into multiple fixed-sized time windows, and we measure the network traffic within each time window.

[2]When the flow size is counted by packets, SpaceSaving replace the heap by a linked list, which cannot be implemented on the data plane and cannot count flow sizes in Bytes.

[3]A tree where each node has at most two children, and except for the last/bottom layer of the tree, it is full of nodes.

of tofino programmable switches and the discussion of implementing one node of the heap. First, we introduce the restrictions of tofino programmable switches, which are inspired by existing work including Sketchlib [51], precision [9]. In a RMT programmable switches [14], when a packet arrives at the data plane, it successively goes through a header parser, ingress pipeline stages, traffic manager, egress pipeline stages, and a sparser finally. We only focus on those pipeline stages. Each stage has its own SRAM memory which does not intersect with other stages, and one stage can only process one packet at any time. A packet must pass through each stage one by one sequentially, and a packet can perform a few parallel and independent *Actions* in each stage. Take tofino switch as an example, one action can access at most one bucket (including two consecutive 32-bit fields) in the memory of the stage, do a few simple arithmetic operations and if-else branches, and read one 32-bit result out from the bucket.

A case study—one node to store the minimum. It is quite challenging to implement a heap, even one of its nodes. A node of the heap records one flow key along with its size. The node needs to support at least two functions when taking a new flow key with the size as input: (1) merge the sizes if the new flow key is the same as the recorded one. (2) Otherwise, if the new flow has a larger size, record the new flow in this node, read out the old recorded flow, and try to insert it into other nodes. However, the above two functions cannot be implemented simultaneously. We can either put the two fields (*i.e.*, the flow key and the size) of a node in one stage or separate them in two stages, but both solutions cannot work. If we put them in one stage, we can only put them together in one bucket, because we need to make a decision based on both (Condition A) whether the keys are the same and (Condition B) the comparison results of flow sizes. But putting them in one bucket hinders us from reading out the old recorded flow together with its size. If we put the two fields in two stages, the result (*i.e.*, Condition A or B) related to the field on the later stage cannot be passed back to the former stage, and the field in the former stage will not be updated correctly.

2.4 Limitations of Recirculation

In the data plane, recirculation is the process of sending a packet that has already been processed back to the ingress pipeline for a second (or third, etc.) round of processing. As in the case study in Section 2.3, this is an alternative solution when a single pass through the pipeline is insufficient to perform a complex operation that requires multistep state updates or conditional logic based on previously computed results. For example, Precision [9] uses probabilistic recirculation to find top flows on a programmable switch; PSMM [29] adopts in-switch recirculation technique to mitigate microbursts. However, recirculation-based methods have their inherent limitations, including Scalability, Latency, Stability and Resource Utilization, leading to a worse performance compared to recirculation-free approaches.

Scalability. Packet recirculation effectively reduces the system's overall throughput, as each recirculated packet consumes at least two packet processing cycles. For instance, if 10% of packets are recirculated, the maximum throughput is reduced by at least 10%, which is a significant penalty in high-speed networks (e.g., 100 Gbps+). For example, Precision [9] points out that its expected number of recirculations grows with the number of packets, which scales with $O(\sqrt{NC})$, where N is the number of packets, and C is the number of its internal counters. Under high load or with a large number of tracked flows, the switch pipeline may become a bottleneck, limiting the system's ability to scale.

Latency. Recirculated packets experience at least twice the full pipeline processing delay, including parsing, match-action stages, and ingress queueing. Under high load, recirculated packets may be queued behind new arrivals, further increasing latency. For example, PSMM [29] reports an average latency of around 970µs even under optimized settings. In addition, packet reordering introduced by recirculation may cause packets to be recirculated multiple times until their sequence number is matched, introducing both increased latency and jitter, which is undesirable for real-time monitoring and control.

Stability. Systems relying on recirculation face stability risks under certain failure scenarios. For example, PSMM's packet sequencer can fail if a packet is dropped during recirculation, causing all subsequent packets to be trapped in the recirculation queue. It takes time for the timers to reset the counters, which will disable the entire system for more than 1ms.

Resource Utilization. Each recirculated packet occupies pipeline resources that could otherwise be used for new packets, reducing the effective forwarding capacity. Precision notes that recirculation "reduces the rate that incoming packets can access the pipeline". Furthermore, implementing mechanisms such as packet sequencing or threshold monitoring requires additional pipeline stages and metadata (e.g., custom headers and state machines), which further constrains switch resources available for other network functions.

By implementing a pipeline friendly heap on the data plane, we could avoid all these issues caused by recirculation. This constitutes a key component of the motivation for our introduction of PipHeap.

3 PIPHEAP DESIGN

We introduce the design of PipHeap in this section. First, we show the overview of the whole data structure. Then we detail the design of the heap nodes. Table 1 shows the symbols that will be frequently used in the following.

Table 1: Symbols frequently used.

Symbol	Meaning
KEY	An arbitrary flow key, *e.g.*, the Source IP
L	PipHeap consists of L node layers
W	The number of activity scores increases when two flows merge [4]
m_i	Each layer consists of m_i nodes
N_i^j	The j-th node in the i-th node array
$N_i^j.key$	The field recording flow key in one node
$N_i^j.value$	The flow size of $N_i^j.key$ in this node
$N_i^j.assist$	The assistant counter
$H(\cdot)$	A hash function mapping the flow key to $\{0, 1, \cdots, m_i - 1\}$

3.1 Overview

Rationale. Towards a normal min heap, PipHeap tries to keep the elephant flows with larger values as close to the bottom layer as possible. As we insert flows starting from the bottom of the heap, we position the bottom layer in the initial stage and subsequently place the other heap layers—from the bottom to the top—into each successive stage. To insert a flow key with its size, we start from the bottom layer and pick a leaf node by hashing the flow key. If the node is empty, we just put the flow in and accomplish the insertion. And if the node records a same flow (*i.e.*, the stored flow key is the same as the new one), we merge them by adding the new flow size to the stored one and then accomplish the insertion. Otherwise, we meet the most challenging case that the stored flow is a different one. Since we cannot access current node again, we have to immediately make a decision between the following two choices: 1. Kick. Store the new flow in current node and move the old flow to the subsequent stages (*i.e.*, the upper layer of the heap). 2. Skip. Insert the new flow into subsequent stages.

Data structure. PipHeap has L layers of nodes, which are divided into two parts by their index (one has the first L_1 ($L_1 < L$) layers, and the other has the remaining), containing two different types of nodes, respectively. Regardless of the part, the i-th layer has m_i ($m_i = \lfloor m_{i-1}/2 \rfloor$) nodes and the j-th node in the i-th layer is denoted as N_i^j. The first layer has a hash function $H(\cdot)$ that maps the flow key to one of the m_1 leaf nodes in the first layer randomly, *i.e.*, $N_1^{H(KEY)}$. The parent node of N_i^j is $N_{i+1}^{j/2}$ and the children of N_i^j are $N_{i-1}^{j\times 2}$ and $N_{i-1}^{j\times 2+1}$. Regardless of node type, each node has three fields, including (1) $N_i^j.key$ that records a flow, (2) $N_i^j.value$ that records the size of $N_i^j.key$, and (3) $N_i^j.assist$ deciding whether to kick $N_i^j.key$ out. In the first L_1 layers, nodes are of the first type called **Agg**regation Nodes, AggNodes for short. AggNode leverages our asymmetric majority algorithm to aggregate the packets of elephant flows. In the other layers, nodes are of the second type called MaxNodes. MaxNode first realizes the basic functions of a heap node: Among two flows, store the larger flow and kick out the other.

Insert. The insert operation can insert a packet/flow $\langle KEY, VAL\rangle$ into PipHeap, where KEY is the flow key and VAL is 1 or the flow size. Starting from the first layer, we find the leaf node $N_i^{H(KEY)}$ by hashing and try to insert the flow into it. If the node is empty, we just store $\langle KEY, VAL\rangle$ in the node. Otherwise, there is already an old flow key with its size. We will choose one of the three possible operation: (1) Merge. If the old key is the same as the new one, we merge the sizes of both flows and finish the insertion without accessing following layers. (2) Skip. If the old key is a different one, we can skip the current node and try to insert the new flow to its parent node. (3) Kick. If the old key is a different one, we can kick out the old flow, store the new flow, and try to insert the old flow into the followed parent node. We repeat above operations in each layer until merge operation occurs or an empty node is encountered. In the last layer, if skip or kick occurs, the flow not stored will be discarded. How to choose between skip and kick depends on the type of the node that will be detailed later in Algorithm 1 and 3 and their explanations. A simple example is shown in Figure 1.

Query. Given a flow KEY, PipHeap reports an estimated flow size. Specifically, we access every layer, find the selected nodes, and check whether the stored flow key $N_i^j.key$ is the given KEY. For all nodes storing KEY, we sum up their $N_i^j.value$ field as our estimated flow size. If no node stores KEY, we report 0.

3.2 The Aggregation Node

3.2.1 Overview. The bottom nodes of the heap are of the AggNode type, which not only accumulates the size of packets with the same flow ID but also selects the most probable elephant flow among different flows for record, while transferring the remaining flows to subsequent stages. To select the elephant flow, we designed the asymmetric majority algorithm. Its key idea is the active score. Once a new flow comes to the node, we check whether it is the same as the old recorded flow. We will increase the active score by W if matches, or decrease by 1 otherwise. Once the active score decreases to zero or less, we think the recorded flow not active anymore, and an replacement will occur, *i.e.*, we will record the ID of the new flow in the node and transferring the old flows to the subsequent stage. Since W is adjustable variable, we examined the optimal value of W in Section 5.2.1. If we define the density of a flow by the probability of its occurrence in the incoming packet, and assume each packet is IID (Independent and Identically Distributed), then we can see that if we unfortunately record a mice flow whose density is less than $\frac{1}{1+W}$ in the AggNode, it will definitely be evicted sometime later. From a high-level view, we can consider the AggNode as a sieve with the size $\frac{1}{1+W}$ of its holes. It can ultimately sift away mice flows whose "size" smaller than that of its hole. Compared to the traditional majority algorithm [15] (*i.e.*, a special case where $W = 1$), we avoid excessive exchanges and achieves higher accuracy.

Algorithm 1: PipHeap Insertion in the first L_1 layers.

Input: A flow $\langle KEY, VAL\rangle$
$j \leftarrow H(KEY)$
for $i = 1, 2, \cdots, L_1$ **do**
  if $N_i^j.key = KEY$ **then**
    $N_i^j.assist \leftarrow N_i^j.assist + W$
    $N_i^j.value \leftarrow N_i^j.value + VAL$
    Break and Finish Insertion.
  else
    if $N_i^j.assist \neq 0$ **then**
      $N_i^j.assist \leftarrow N_i^j.assist - 1$
      Insert $\langle KEY, VAL\rangle$ to next level.
    else
      Insert $\langle KEY, VAL\rangle = \langle N_i^j.key, N_i^j.value\rangle$ to next layer.
      $N_i^j.key \leftarrow KEY$
      $N_i^j.assist \leftarrow W$
      $N_i^j.value \leftarrow VAL$
  $j \leftarrow j/2$
If the insertion is not finished, insert $\langle KEY, VAL\rangle$ **to next layers using Algorithm 3.**

For a further remark, careful readers may find that the calculation of the activity score does not leverage the flow size. The reason behind is that leveraging flow sizes for score calculation

will consume twice the number of stages, which hinders the stacking technique that we will introduce in 3.2.3. One may worry that a flow with a large size but inserted only once cannot be stored. Fortunately it will be addressed in Section 3.3 using the MaxNode.

3.2.2 Design and Functionalities. Each AggNode contains three fields $\{key, assist, value\}$ in its record, to track the flow ID, active score and value of the recorded flow respectively. It can support both insertion and query operations.

Insertion in the AggNode. To insert $\langle KEY, VAL\rangle$ to an AggNode N_i^j, we check the cases of merging, skipping, and kick one by one. If $N_i^j.key$ is KEY, we merge the new flow with the old flow including adding W to $N_i^j.assist$ and VAL to $N_i^j.value$, and then we finish the insertion. Otherwise, we check whether $N_i^j.assist$ is 0. If not, we decrease the $N_i^j.assist$ by one and skip this layer. Otherwise, when $N_i^j.assist$ is 0, we kick out the old flow $\langle N_i^j.key, N_i^j.value\rangle$, insert it to the next layer, and put the new flow into the node by setting $\langle N_i^j.key, N_i^j.assist, N_i^j.value\rangle$ to $\langle KEY, W, VAL\rangle$ respectively. In Algorithm 1, we show the pseudo-code of PipHeap insertion.

Example. In Table 2, we show how a node changes with many insertions. We sequentially insert the flows with keys: a, b, b, ... For simplicity, their sizes (VAL) are 1. Each insertion can take one action among K (Kick), S (Skip), and M (Merge). And we show the contents of the node before and after each action. For example, the node $\langle N_i^j.key, N_i^j.assist, N_i^j.value\rangle$ starts with $\langle 0, 0, 0\rangle$, and the insertion of flow a takes a kick operation setting the node to $\langle a, 2, 1\rangle$. Note that after insertion of the third b, there will be duplicate "b" nodes in the heap (previously Skipped "b"s may be stored in the subsequent level). This is acceptable and is actually a part of the heap implementation.

Table 2: An example of AggNode with $W = 2$.

Insertions	a,b,b,b,a,c,b,b,a,c,b,b
Actions	K,S,S,K,S,S,M,M,S,S,M,M
$N_i^j.key$	0,a,a,a,b,b,b,b,b,b,b,b
$N_i^j.assist$	0,2,1,0,2,1,0,2,4,3,2,4,6
$N_i^j.value$	0,1,1,1,1,1,1,2,3,3,3,4,5

Algorithm 2: Basic PipHeap Query.

Input: A flow KEY
$TotalValue \leftarrow 0$
$j \leftarrow H(KEY)$
for $i = 1, 2, \cdots, L$ **do**
  if $N_i^j.key = KEY$ **then**
    $TotalValue \leftarrow TotalValue + N_i^j.value$
  $j \leftarrow j/2$
Report $TotalValue$.

Query. In algorithm 2, we show how to estimate the flow size of a given flow key.

3.2.3 Deployment and Implementation. **Single AggNode deployment.** One AggNode requires two stages. $N_i^j.assist$ and $N_i^j.key$ are put in the first stage, and $N_i^j.value$ is put in the second stage. In the first stage, we check whether $N_i^j.key = KEY$ and $N_i^j.assist = 0$. When $N_i^j.assist = 0$, we set $N_i^j.key$ to KEY. When both $N_i^j.key$ not equals to KEY and $N_i^j.assist$ is not 0, we decrease $N_i^j.assist$ by one. Otherwise, we add $N_i^j.assist$ by W and read the $N_i^j.key$ out as $Outkey1$. The above operations are performed in parallel and the condition judgment is based on the variable content before the assignment operation.

In the second stage, we can take the action according to $Outkey1$ from the first stage, which is one of $\{empty, KEY, N_i^j.key\}$. If $Outkey1$ is empty, we know the skip happened ($N_i^j.key \neq KEY$ and $N_i^j.assist \neq 0$), do not take action in the second stage, and will move the new flow $\langle KEY, VAL\rangle$ to the next layer. If $Outkey1$ is KEY, we know the merge happened, increase the $N_i^j.value$ and accomplish the insertion. If $Outkey1$ is the old $N_i^j.key$, we know the kick happened, replace the $N_i^j.value$ by new VAL, read old $N_i^j.value$ out as $Outval2$, and finally move the old flow $\langle Outkey1, Outval2\rangle$ to the next layer.

Node stacking technique (Figure 2). Our stacking design can reduce the number of required stages. A straightforward implementation is to serialize each layer without overlapping each other. Since one node/later requires two stages, L layers of AggNodes requires $2 \times L$ stages, and therefore the pipeline including 12 stages can implement at most 6 layers. To efficiently utilize the limited number of stages, we devise the stacking optimization that reduces the required stages from $2 \times L$ to $L + 1$. Consider two nodes A and B in the first and the second layer respectively. As shown in the Figure 3, we can parallelize the second stage of node A and the first stage of node B. After getting $Outkey1$ from node A in stage 1, we know which action (Skip, Kick or Merge) to perform, and therefore we can determine whether the input KEY of the node B in stage 2 is the new key or the old key in stage 1. Similar for the input VAL of node B, we can determine whether the input VAL of the node B in stage 3 is the new value or the old one in stage 2. By the stacking design, the 12-stage pipeline can implement at most 11 layers of AggNodes.

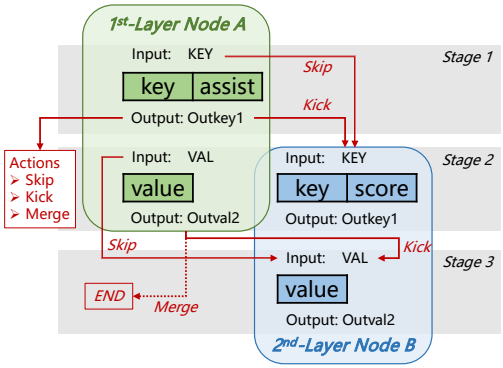


Figure 2: The stacking implementation of AggNodes.

Query entirely on the data plane. Most state-of-the-art solutions/sketches read the data structure from the data plane to the control plane, and uses the CPU to query. This prevents other functions deployed on the data plane (*i.e.*, load balancing or congestion control) from obtaining real-time measurement results. A practical solution should not only support queries on the control plane, but also be able to output the results directly on the data plane.

On the data plane, our PipHeap can either query the flow size of a given key (Query-Alone), or report the size of a flow while inserting it (Query-with-Insert). For Query-Alone, we can add a query action at each layer: compare whether the queried key is the same as the stored one in the selected node, and if they are the same, accumulate the value. For Query-with-Insert, we can add query action after the merge action. After the merge action occurs, the inert operation is completed. Then we can start the query action that reads out the value of the inserted flow, and accumulate the value in each following layers if the key matches.

3.3 The Max Node

The MaxNode aim at the basic function of a heap node: When inserting a new flow to a node storing an old flow, store the larger flow an kick out the smaller one to the next layer. A MaxNode has three fields, including $N_i^j.key$, $N_i^j.value$ and $N_i^j.assist$, where $N_i^j.assist$ is always equal to $N_i^j.value$ for the deployment.

Insertion in the MaxNode. To insert $\langle KEY, VAL\rangle$ to a MaxNode N_i^j, we check the cases of merging, skipping, and kick one by one. Firstly, if $N_i^j.key$ is KEY, we merge the new flow with the old flow including adding VAL to both $N_i^j.assist$ and $N_i^j.value$, and then we finish the insertion. Secondly, otherwise, we check whether $N_i^j.assist$ is larger than VAL. If not, we skip this layer and try to insert $\langle KEY, VAL\rangle$ into the parent node. Otherwise, when $N_i^j.assist$ is no larger than VAL, we kick out the old flow $\langle N_i^j.key, N_i^j.value\rangle$, insert it to the next layer, and put the new flow into the node by setting $\langle N_i^j.key, N_i^j.assist, N_i^j.value\rangle$ to $\langle KEY, VAL, VAL\rangle$ respectively. In Algorithm 1, we show the pseudo-code of PipHeap insertion.

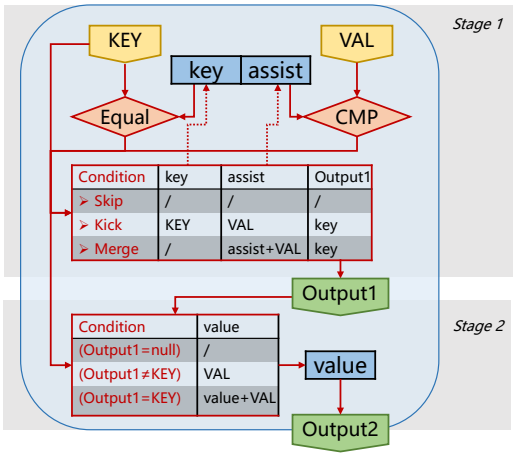


Figure 3: The implementation of MaxNodes.

Single MaxNode deployment. One MaxNode requires two stages. $N_i^j.key$ and $N_i^j.assist$ are put in the first stage, and $N_i^j.value$ is put in the second stage. In the first stage, we check whether $N_i^j.key = KEY$ and $N_i^j.assist \leqslant VAL$. For the $N_i^j.key$ field, only when the kick happens, *i.e.*, $N_i^j.assist \leqslant VAL$ and $N_i^j.key \neq KEY$, we set $N_i^j.key$ to KEY. For the $N_i^j.assist$ field, since the switch does not allow three branches, we use a two branch logic to update its value. First, when the merge happens (*i.e.*, $N_i^j.key = KEY$), we set $N_i^j.assist$ to $N_i^j.assist + VAL$. Second, when the skip or kick happens, we set $N_i^j.assist$ to $MAX(N_i^j.assist, VAL)$. Here we use a MAX function to avoid using the comparison unit to compare $N_i^j.assist$ and VAL. For the output of the first stage, we set $Output1$ to $N_i^j.key$ only when the kick or merge happens, *i.e.*, $N_i^j.key = KEY$ or $N_i^j.assist \leqslant VAL$.

In the second stage, we can take the action according to $Outkey1$ from the first stage, which is one of $\{empty, KEY, N_i^j.key\}$. If $Outkey1$ is empty, we know the skip happened, do not take action in the second stage, and will move the new flow $\langle KEY, VAL\rangle$ to the next layer. If $Outkey1$ is KEY, we know the merge happened, increase the $N_i^j.value$ and accomplish the insertion. If $Outkey1$ is the old $N_i^j.key$, we know the kick happened, replace the $N_i^j.value$ by new VAL, read old $N_i^j.value$ out as $Outval2$, and finally move the old flow $\langle Outkey1, Outval2\rangle$ to the next layer.

Algorithm 3: PipHeap Insertion in layers $L_1 + 1 \sim L$.

```
Input: A flow ⟨KEY, VAL⟩, and j
for i = L_1 + 1, ··· , L do
    if N_i^j.key = KEY then
        N_i^j.assist ← N_i^j.assist + VAL
        N_i^j.value ← N_i^j.value + VAL
        Break and Finish Insertion.
    else
        if N_i^j.assist > VAL then
            Insert ⟨KEY, VAL⟩ to next layer.
        else
            Insert ⟨KEY, VAL⟩ = ⟨N_i^j.key, N_i^j.value⟩ to next
              layer.
            N_i^j.key ← KEY
            N_i^j.value ← VAL
            N_i^j.assist ← VAL
    j ← j/2
Discard ⟨KEY, VAL⟩.
```

3.4 Summary of Design

The core challenge on a programmable switch pipeline is that data cannot flow backwards. Once a packet leaves a pipeline stage, it cannot go back to a previous stage. The traditional heap insertion's "bubble-up" process requires a node to be compared with its parent, which would necessitate moving data from a later stage back to an earlier one, which is prohibited.

PipHeap ingeniously reverses the process to work within the pipeline constraints. Looking back to Section 3.1, we stated that PipHeap tried to keep the elephant flows with larger values as close to the bottom layer as possible. This is achieved by making intelligent "Kick/Skip" decisions at the very first stage (and subsequent stages), where PipHeap actually acts as a filter. The bottom layers preferentially retain the larger, more frequent flows by kicking out smaller ones and skipping for larger ones. This filtering process ensures that the elephant flows are "trapped" in the bottom layers, achieving the stated goal. This is the fundamental approximation that allows a heap-like structure to function within the strict forward-only nature of a switch pipeline.

4 TASK APPLICATIONS

4.1 Single Key Query

4.1.1 Flow Size Query. Flow size estimation is to report the flow size of a user-given flow key, where the size can be either the number of packets or the Bytes. Our PipHeap can not only directly estimate the flow size accurately, but also optimize existing schemes (called collaborators) to obtain higher accuracy. The justification for this combination with collaborators lies in the segregation of elephant and mice flows, a strategy that has been proven effective in various studies [39, 66]. Specifically, we put PipHeap in the front and one collaborator in the following stages. When inserting flows, we insert all discarded flows of PipHeap, which were formerly discarded after the PipHeap insertion process, into the collaborator. To estimate the flow size, we let PipHeap and the collaborator estimate the flow size independently, and add up the results as the answer. Experiment results show that this combination works very well, and as a remarkable fact, PipHeap doesn't introduce any extra error at all to the collaborators, the overall error only depends on the combined algorithm. As a result, the combination can completely inherit the properties (such as one-side error, unbiased estimation, etc.) from collaborators.

Query (one-side error). To estimate the flow size with one-side error, *i.e.*, the reported answer is higher than or equal to the true flow size, we can select the collaborator from any solution with the one-side error, such as CM, CU, FCM and SuMax [73]. We recommend to select SuMax that is the state-of-the art solution, and answer the query in the way mentioned above.

Query (unbiased estimation). To given an unbiased estimation of the flow size, *i.e.*, the expect value of reported answer is exactly the true flow size, similarly, we can select the collaborator from any solution with the unbiased error, including Count, Univmon, Nitro and more [23].

4.1.2 Heavy Hitter Query. **Query heavy flows.** The heavy flows are flows whose sizes are greater than a threshold Λ. The query is to find all flow keys of heavy flows and their corresponding sizes. Similar to the query of flow sizes, PipHeap can either find heavy flows by itself, but also optimize existing schemes (called collaborators). When PipHeap works alone, we read it into the control plane and query the size of all keys stored in PipHeap through the standard query method. For all keys with an estimated size greater than Λ, we report them together with their estimated sizes. When PipHeap works with a collaborator, for all keys stored in PipHeap, we estimate the flow size by adding up the results from both PipHeap and the collaborator, and then report those keys with a size exceeds Λ.

Query top-k flows. The top-k flows are k flows with the largest sizes, which is similar to the heavy flows. When finding top-k flows, based the above scheme of finding heavy flows, we sort the heavy flows according to the estimated size and report the largest k flows.

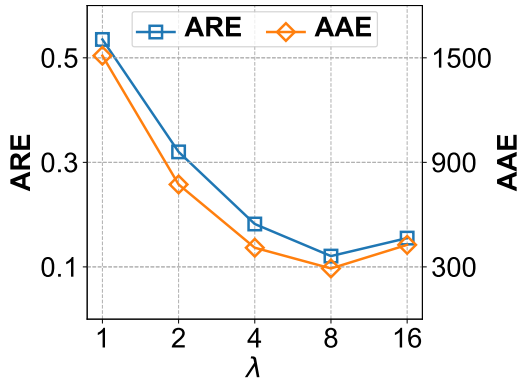
Query super-spreaders, DDoS victims and port scans. The super-spreader is a source IP sending packets to many (more than Λ) destination IP addresses in the measurement time interval. A common solution is to use an additional Bloom filter (BF) remove duplicate packets, and then count the destination IP for each source address: when a packet arrives, insert $\langle SrcIP, DestIP\rangle$ (possibly with a hash to fit in 32-bit limit) to BF and query whether the pair appears for the first time, If so, we insert $\langle SrcIP, 1\rangle$ to PipHeap, which means that $SrcIP$ has contacted a new $DestIP$. Otherwise, we ignore the packet. When a $SrcIP$ has a size greater than Λ, we report it as a super-spreader. In a similar way, we can find the DDoS victim (*i.e.*, a destination IP contacted by many sources) and port scans (*i.e.*, a source IP that accesses many destination IP addresses and ports).

4.1.3 Other tasks based on single key queries. Based on the results of single key queries, we can solve many tasks.

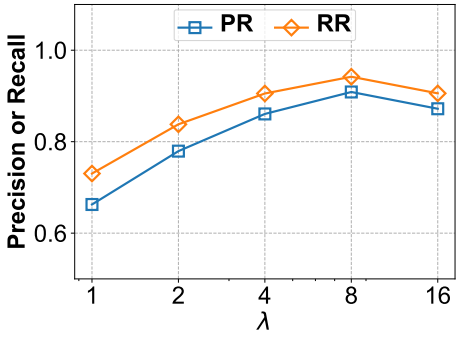
Flow size distribution estimation. To estimate the distribution of flow sizes, we combine PipHeap with a CM sketch. We use the MRAC [35] algorithm whose input is a CM sketch to calculate a distribution. Then, for all flow keys in PipHeap, we query their sizes in the CM sketch, remove the size from the distribution, and add the total size of each flow in both PipHeap and CM to the distribution.

Entropy estimation. We can compute the traffic entropy $-\sum n_i \cdot \frac{i}{m}\ln(\frac{i}{m})$ based on the flow size distribution, where m is the total number of distinct flows and there are n_i flows with size i.

Cardinality estimation. To estimate the cardinality, *i.e.*, the number of distinct flows, we combine PipHeap with a Linear Counting (LC) algorithm [62]. After inserting all packets, we insert all keys in PipHeap into the LC structure, and estimate the cardinality by the LC algorithm.

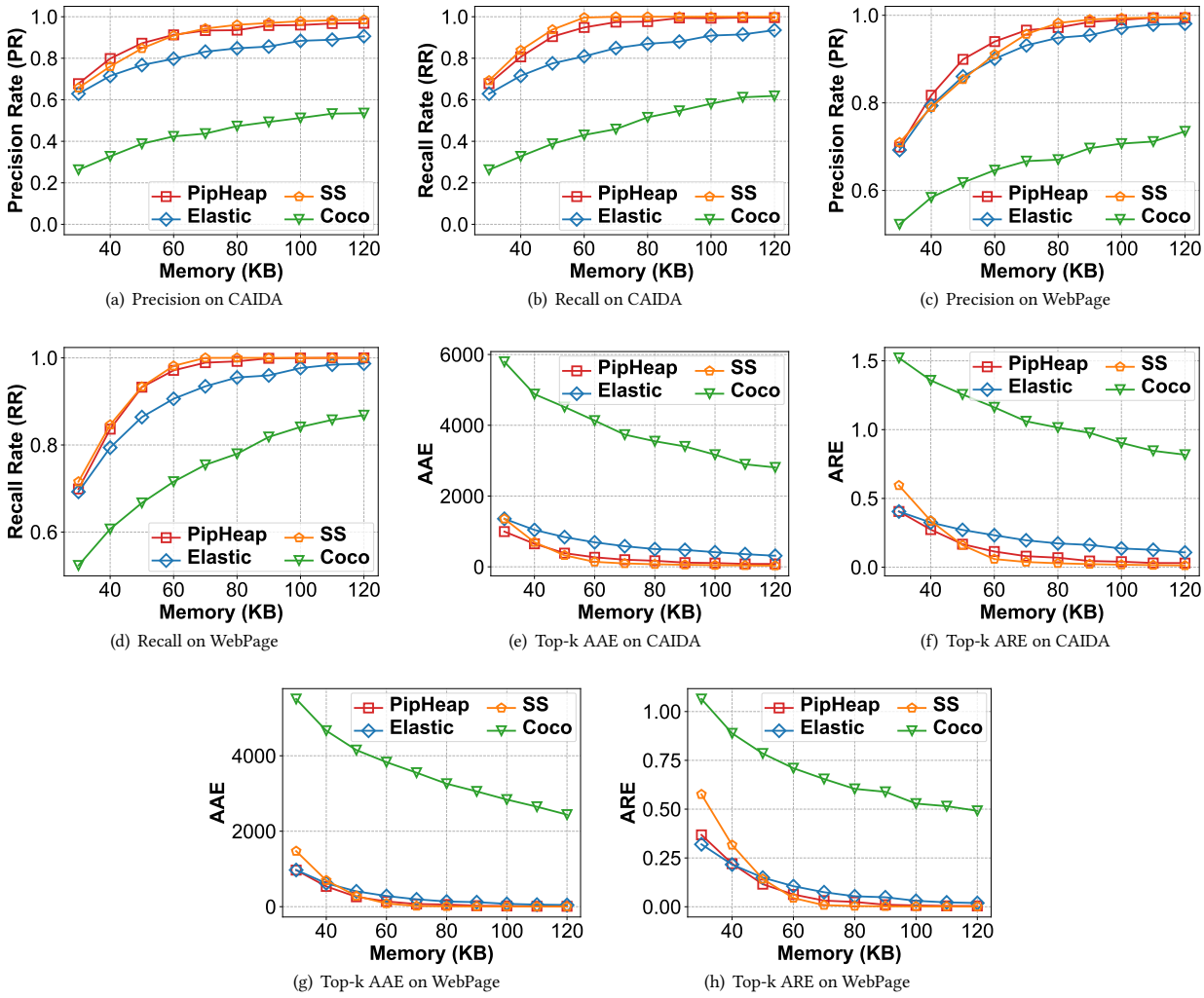


(a) ARE and AAE



(b) Precision and Recall

Figure 4: Impact of Parameter W on the Accuracy.



(a) Precision on CAIDA (b) Recall on CAIDA (c) Precision on WebPage

(d) Recall on WebPage (e) Top-k AAE on CAIDA (f) Top-k ARE on CAIDA

(g) Top-k AAE on WebPage (h) Top-k ARE on WebPage

Figure 5: Accuracy Comparison between PipHeap, the Ideal SS, and other Competing Methods.

4.2 Hierarchical Key Query

Query Hierarchical Heavy Hitters (HHH) is to find the set of flows with a common IP address prefix that have a large total flow size (*i.e.*, greater than a threshold). The milestone solution is Randomized Hierarchical Heavy Hitters (RHHH) that achieves constant update time using multiple SpaceSaving. We replace each SpaceSaving by our PipHeap in RHHH. After inserting all packets, we query each PipHeap to find HHH, which is the same as RHHH. Since SpaceSaving cannot be implemented on the programmable switch, our PipHeap will make RHHH practical in high speed traffic.

4.3 Arbitrary Partial Key Query

Given a full key range (*e.g.*, five-tuple) with N bits, the user can query the size of any flow key that is a part of the full range (*i.e.*, any number of bits in N bits). The Coco sketch is the first practical solution for arbitrary partial key query on the switch, and we can combine PipHeap with it by inserting all PipHeap's discarded flow into a Coco sketch. The combination will improve the accuracy of the Coco sketch.

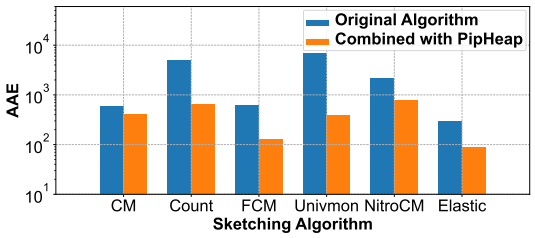
5 EXPERIMENTAL RESULTS

This section presents the experimental results of PipHeap. Firstly, we describe the experimental settings in section 5.1. Then, in section 5.2, we conduct experiments on various parameter settings to demonstrate how different parameters affect accuracy and to determine the best parameter to choose. Subsequently, in section 5.3, we carry out experiments on the combination of PipHeap with various types of sketching algorithms, demonstrating that the combination can yield a significant improvement in performance.

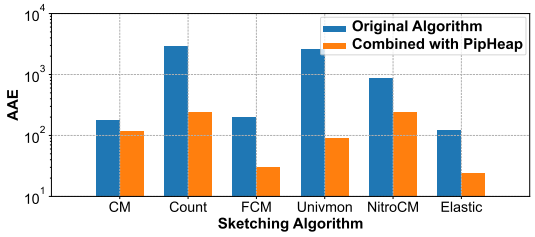
5.1 Experimental Setup

Implementation: We implement PipHeap and all other sketching algorithms in P4 and C++. Our source code is available at [5]. In order to evaluate the accuracy of PipHeap, we conduct experiments

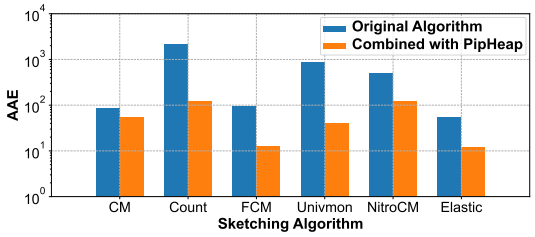
on CPU platform, using a server with a 36-core CPU (Intel i9-10980XE), 128GB DDR4 memory and 25.4MB L3 cache. We set the CPU frequency to 4.2GHz, and set the memory frequency to 3200MHz. Our accuracy results are robust to the hash functions used in the implementation, achieving nearly identical results on common CRC, Murmur [3], and Farm [1] hashes.



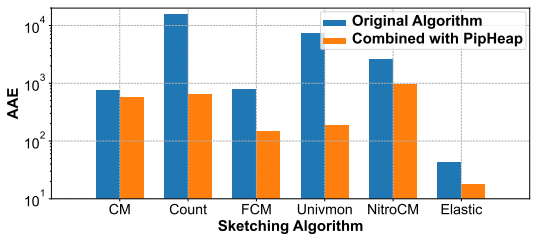
(a) CAIDA 120KB



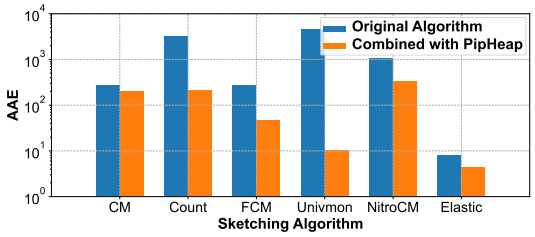
(b) CAIDA 240KB



(c) CAIDA 360KB



(d) WebPage 120KB



(e) WebPage 240KB

Figure 6: Accuracy Improvement Brought by PipHeap for Six Algorithms in Flow Size Estimation.

Datasets: We evaluate sketching algorithms using the following datasets: (1) **CAIDA:** The CAIDA is the IP trace containing anonymous network traces collected from high-speed monitors on backbone links in 2018 by CAIDA [6]. Due to the hardware constraint of the programming switch, we only use the source IP address (4 bytes) of each packet as its ID. We use one-minute trace of the dataset, containing approximately 27M packets belonging to 0.25M flows. (2) **WebPage:** The WebPage dataset is built from a collection of web pages [4]. Each item (4 bytes) represents the number of distinct items in a web page. We use a part of it, which contains approximately 64M items belonging to 0.94M distinct items.

Metrics: We evaluate the accuracy of sketching algorithms on the most frequent k flows using the following metrics:

- **Average Relative Error (ARE):** $\frac{1}{|\mathcal{S}_k|}\sum_{f_i\in\mathcal{S}_k}\frac{|n_i-\hat{n}_i|}{n_i}$, where n_i is the real frequency of flow f_i, $\hat{n}_i$ is the estimated frequency of flow f_i, and $\mathcal{S}_k$ is the set containing the most frequent k flows.
- **Average Absolute Error (AAE):** $\frac{1}{|\mathcal{S}_k|}\sum_{f_i\in\mathcal{S}_k}|n_i-\hat{n}_i|$, where $\mathcal{S}_k$, f_i, n_i, $\hat{n}_i$ are the same as those defined above.
- **Precision Rate (PR):** $|\hat{\mathcal{S}}_k\bigcap\mathcal{S}_k|/|\hat{\mathcal{S}}_k|$, where $\mathcal{S}_k$ is the set containing the most frequent k flows, $\hat{\mathcal{S}}_k$ is the set containing the the reported most frequent k flows by the sketching algorithm.
- **Recall Rate (RR):** $|\mathcal{S}_k\bigcap\hat{\mathcal{S}}|/|\mathcal{S}_k|$, where $\mathcal{S}_k$ is the same as that defined above, $\hat{\mathcal{S}}$ is the set contains all the flows reside in the data structure of the sketching algorithm.

5.2 Experiments on PipHeap

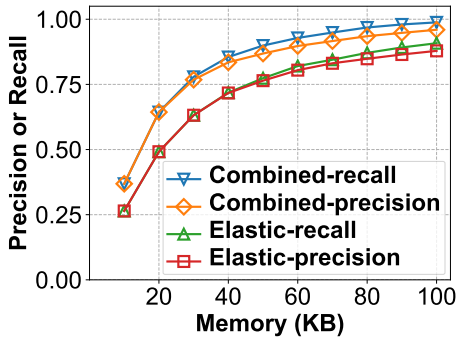
In this section, we first conduct experiment on PipHeap to show the influence of parameter W on the accuracy, which can help to decide the best choice of W. Then we compare PipHeap with Space Saving [46] and Elastic Sketch [66]. Through this comparison, we show that PipHeap exhibits traits similar to an ideal heap, performing favorably in tasks such as flow size estimation and identifying heavy hitters.

Rationale: Since Space Saving is an **ideal heap algorithm** which can be **only implemented on CPU**, the experimental result that PipHeap is close to Space Saving on accuracy indicates PipHeap is a pretty accurate heap implementation. Besides, for the P_4 version of Elastic Sketch is so far the only sketching algorithm that can be implemented on the data plane without recirculation, the result that PipHeap reaches much better accuracy than Elastic implies PipHeap is a fairly accurate sketching algorithm implemented on programming switch.

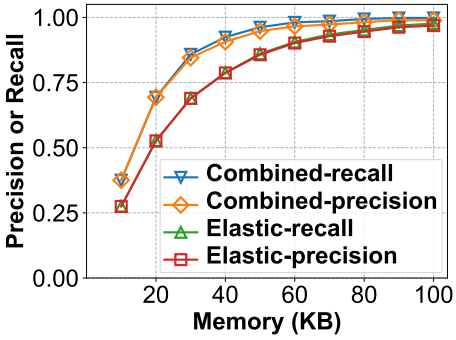
5.2.1 Impact of Parameters. We conduct experiments on PipHeap with W ranges from 1 to 16, and test accuracy on the most frequent 3000 flows. The result is shown in figure 4. We find that when $W = 1$, PipHeap doesn't perform well. As the W gets bigger, PipHeap gains better accuracy since the heavy flows become more difficult to be kicked out, which makes PipHeap easier to retain the heavy hitters. However, when W is greater than 8, the accuracy of PipHeap begins to deteriorate, since it is difficult to kick out the small flows reside in its aggregation nodes. Our experiments show that $W = 8$ is an optimal choice.

5.2.2 Accuracy Comparison.

Parameter Settings: We conduct experiments on PipHeap, Space Saving and Elastic Sketch on IP Trace and WebPage datasets with

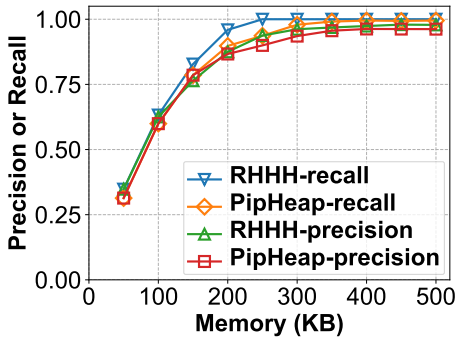


(a) CAIDA

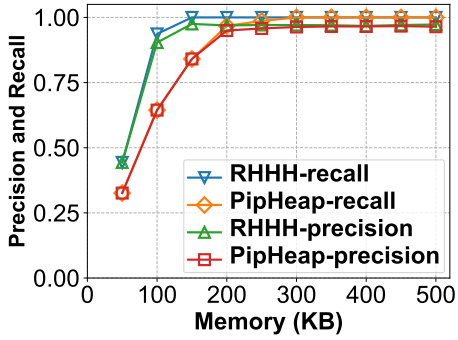


(b) WebPage

Figure 7: Accuracy Improvement Brought by PipHeap for Elastic in Heavy Hitter Detection.



(a) CAIDA



(b) WebPage

Figure 8: PipHeap makes the implementation of RHHH in the data plane possible without any significant loss in accuracy.

memory ranging from 30KB to 120KB. For PipHeap we take $W = 8$, for Elastic Sketch we use the P_4 version with 4 layers and take $\lambda = 32$ (as recommended in [66]). We test AAE, ARE, CR and PR on the most frequent 3000 flows.

Experimental Results (Figure 5): We find that PipHeap performs much better than Elastic Sketch, and it can reach almost the same accuracy as the ideal goal of Space Saving (SS). We calculated the accuracy of PipHeap, Elastic and Space Saving on CAIDA and WebPage datasets with memory ranging from 30 KB to 120 KB. From 5(a) and 5(b), we find that PipHeap is always better than Elastic clearly. PipHeap is only slightly worse than the ideal Space Saving on CAIDA dataset. As for the WebPage dataset in Figure 5(c) and 5(d) show that although the gap narrows, PipHeap still maintains its superiority. When calculating the flow sizes of top-k flows (Figure 5(e)-5(h)), we also find that PipHeap is dominant, and has always been more accurate than Elastic. When the memory is small, Space Saving does not perform well, although it gets better immediately, after more than 60KB, PipHeap and Space Saving are close again.

5.3 Experiments on PipHeap-based Solutions

In this subsection, we show the optimization of PipHeap for some existing problems, including heavy hitter detection, flow size estimation, hierarchical heavy hitter and arbitrary partial key query. These problems have been solved by many excellent algorithms. But because of the simplicity and generality of PipHeap, our experiments show that combining PipHeap with these algorithms can achieve better performance. Specifically, the combined algorithm we implemented uses half of the original memory for PipHeap, and the other half is reserved for other algorithms. For algorithms that can be implemented on programmable switches, we have the corresponding P4 version [5] .

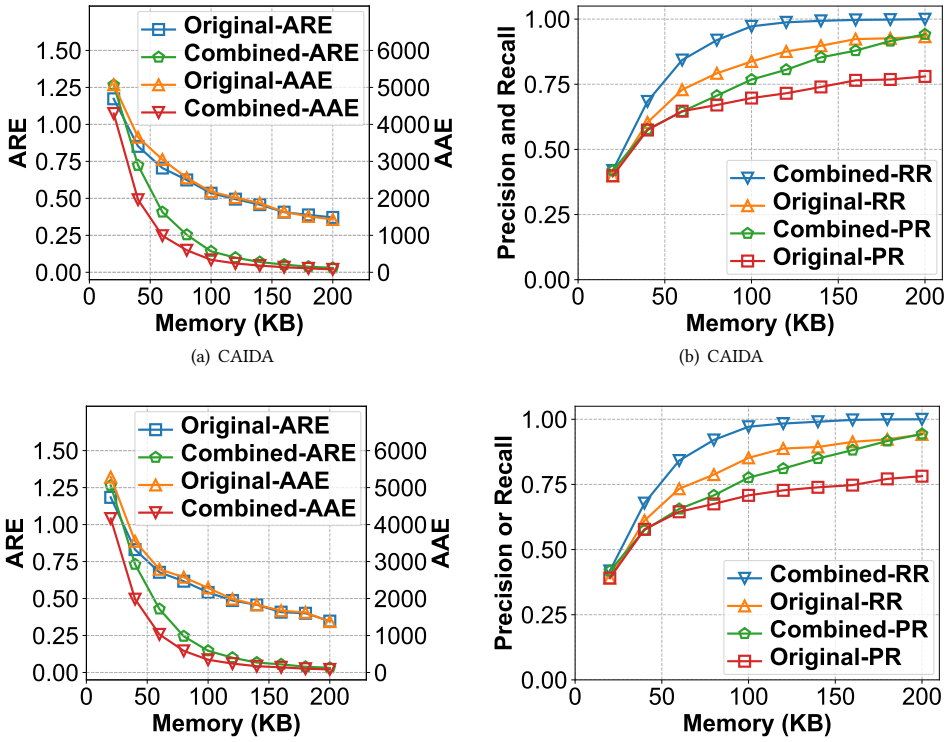
Key results. We combine PipHeap with eight sketches. For CAIDA dataset and the 240 KB memory, PipHeap reduces the error of seven sketches including CM, Count, FCM, Univmon, NitroCM, Elastic, and CoCo by 33%, 92%, 85%, 97%, 72%, 80%, and 87%, respectively. On average, PipHeap reduces the error by 78%. PipHeap make the implementation of RHHH possible on the data plane while only incurring 2% accuracy loss.

5.3.1 Single Key Heavy Hitter Detection. We conduct experiments on P_4 version of Elastic Sketch and the combination of PipHeap and Elastic Sketch to evaluate the precision rate and recall rate of heavy hitter (defined by most frequent 3000 flows) detection. The result is shown in figure 7. We find that in the memory ranges from 10KB to 100KB, the precision and recall of the combined algorithm in Heavy hitter detection are always better than Elastic. When the memory is 40KB, the combined algorithm can achieve a high accuracy rate of over 80% for both precision and recall. However, Elastic can not achieve the same performance until 100KB.

5.3.2 Single Key Flow Size Estimation. From the experiment results 6, we can find that after combining PipHeap, most algorithms can be significantly improved in the task of flow size estimation. For example, for CAIDA dataset and the same 240KB memory (Figure

Table 3: Resource used by PipHeap and Combined algorithms

Resource	PipHeap	+CM	+SuMax	+Coco	+Elastic	+Space Saving	+RHHH
SRAM	103(16.1%)	130(18.1%)	130(14.8%)	139(14.5%)	153(17.4%)	136(18.9%)	103(12.9%)
Map RAM	103(26.8%)	130(30.1%)	130(24.6%)	139(24.1%)	153(29.0%)	136(31.5%)	103(21.5%)
Hash Bits	255(7.7%)	303(8.1%)	303(6.6%)	319(6.4%)	313(6.8%)	271(7.2%)	255(6.1%)
TCAM	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)
Stateful ALU	10(31.3%)	13(36.1%)	13(29.6%)	13(27.1%)	12(27.3%)	11(30.6%)	10(25.0%)
Exact Match Xbar	16(12.5%)	19(13.2%)	19(10.8%)	19(9.9%)	19(10.8%)	17(11.8%)	17(10.6%)
VLIW Instr	12(4.7%)	13(4.5%)	15(4.3%)	15(4.4%)	17(4.3%)	13(4.5%)	15(4.7%)



(a) CAIDA (b) CAIDA

(c) WebPage (d) WebPage

Figure 9: Accuracy Improvement Brought by PipHeap for Coco in Arbitrary Partial Key Query.

6(b)), PipHeap reduces the error of CM, Count, FCM, Univmon, NitroCM, and Elastic by 33%, 92%, 85%, 97%, 72% and 80%, respectively. On average, PipHeap reduces the error by 76%. Here, we aim at estimating the size of all flows, not just heavy hitters. To do this, we combine some algorithms for flow size estimation with PipHeap. Specifically, when each item arrives, it is inserted into PipHeap first, and the item kicked out from the last node of PipHeap will be inserted into the subsequent algorithm. Note that PipHeap is an algorithm that simulates a heap, so the item kicked out from the last node is an approximate heap minimum. Therefore, the advantage of combining PipHeap with these flow size estimation algorithms is that the flows that will be inserted into the subsequent algorithms in this way are small flows with high probability, and most of the large flows will remain in PipHeap. This can bring new properties to the combined flow size algorithm. The distribution of the data flow becomes smooth. For example, CM sketches can rely on this property to reduce the number of bits per counter and reduce the total memory. Some algorithms do not work well because they are too simple or rely on the skewness of the flow distribution. And PipHeap can be implemented on programmable switches, which brings more possibilities for full-flow measurement on programmable switches.

5.3.3 Hierarchical Heavy Hitter Query. To find Hierarchical Heavy Hitters, we try to implement RHHH on the data plane by replacing its Space Saving structures by PipHeaps. In Figure 8, we show that PipHeap's accuracy is very close to the ideal RHHH in the CPU platform. As an approximation of the heap, PipHeap did not seriously affect the accuracy of RHHH. In the CAIDA dataset, we query the 16-bit prefix and the 24-bit prefix. PipHeap is consistently close to RHHH. In the WebPage dataset, we query the 24-bit prefix. Due to the format of the data, there are few types of prefixes, resulting in poor performance of PipHeap when the memory is small, but when there is enough memory, PipHeap can still get a good enough estimate. The experimental results prove that PipHeap makes finding hierarchical heavy hitters possible on programmable switches. When the total memory is 240KB with CAIDA dataset, PipHeap

only incurs additional 2% more AAE when querying the 24-bit prefix.

5.3.4 Arbitrary Partial Key Query. As shown in Figure 9, for arbitrary partial key query, we combine PipHeap with Coco and improve its accuracy significantly. We test them on the datasets of CAIDA and WebPage, and query the 16-bit prefix. PipHeap always improves the accuracy of CoCo in both AAE, ARE, Precision and Recall. For example, when the total memory is 240KB with CAIDA dataset, PipHeap reduces the error of CoCo by 87%.

5.4 Resources Usage on Programmable Switch Testbed

In the Tofino programmble switch, we implement a PipHeap with 131071 nodes (capable of tracking over 100K heavy flows) and its combination with six sketches inlcuding CM, SuMax, Coco, Elastic, Space Saving and RHHH. The measurement model we focus on is the classic fixed window, which divides the timeline into consecutive equal-sized fixed windows. Once each window ends, the sketch data from the data plane is sent to the control plane and reset to empty. We show the resource usage in Table 3. The two most used resources by PipHeap are Stateful ALU and Map RAM, which take up 31.25% and 26.82% of the total quota, respectively. These two are mainly used for the counter of heap node of PipHeap. The remaining resources of other items do not exceed 20%. After combining with other algorithms, these properties are still maintained, the highest Stateful ALU and Map RAM occupies reach 36.11% and 31.48%, and the others do not exceed 20%.

6 RELATED WORK

Heavy hitter detection schemes. Many work aim at finding heavy hitters on the data plane of programmable switches. Hashpipe [57] can track heavy flows using multiple tables of slots where each slot stores a flow ID and its size. However, it is based on the programming behavioral model and cannot be implemented [9] on real programmable switch product. By similar multiple tables, Elastic [66] can be implemented, but it incurs serious errors, *i.e.*, after moving a flow, its size will be reset to 1. Qpipe [33] and Precision [9] rely on the recirculation to update the desired position, but such recirculation will consume available bandwidth of the switch. Some solutions rely on the control plane CPU and the upload bandwidth, including Beaucoup [19], Netcache [34] and more [42, 44, 51]. The OmniMon [32] relys on the end hosts.

Sampling schemes. By sampling a portion of packets, the measurement scheme can monitor the network traffic with low overhead. The representative solutions include NetFlow [20], sFlow [52], Everflow [76], OpenSample [58], Nitro [43], and more [24, 36, 54, 55, 70]. However, the sampling solutions are not accurate enough because they cannot measure all packet and provide certain error bounds.

Sketch-based Solutions. The sketch [7, 28, 30, 37, 38, 49, 50, 71] is a series of approximate algorithms for stream data. Existing sketches can be classified into two types: counter-based sketches and heap-based sketches. Counter-based sketches consist of multiple arrays of counters and do not record flow ID (*e.g.*, five-tuple) or fingerprint. Typical sketches include Count-Min (CM) [21], CU [25], Count [17], UnivMon [44], Nitro [43], NZE [31], Tower [65] and more [68, 70, 74]. They are memory efficient for estimating per-flow size but do not record flow IDs, which is required in finding heavy hitters, packet loss, latency and more tasks. To record IDs, existing solutions either occupy the upload bandwidth to control plane, rely on the end hosts, or have low accuracy [32, 51]. The heap-based sketches, including Space Saving [46], Frequent [22], Unbiased Space Saving [59] and more [53, 64, 67], consist of many ID-counter pairs recording flow IDs and flow sizes, and these pairs are in a heap structure that can keep tracking pairs with large flow sizes. With recorded flow IDs, the heap-based sketches can provide more statistics such as heavy hitters and heavy changes. Some sketches (*e.g.*, Elastic [66] and Augment [53]) are the hybrid of the above two types: a heap followed by a counter-based sketch.

Decoding Schemes. Such schemes can achieve high accuracy when the resources are sufficient. However, they cannot provide measurement results at line rate, because they need recovery and aggregation using CPU. Typical schemes are CounterBraids [45], Omnimon [32], FlowRadar [39] and more [27, 40].

7 CONCLUSION

In this paper, we propose the first approximate heap (called PipHeap) for switch data plane, which firstly track heavy flows on the data plane without recirculation or incurring error on combined sketches. We combine PipHeap with eight sketches and our evaluation on the real-world dataset shows that their error can be reduced by 33% ∼ 97% (78% on average) in the same memory. In our testbed, we implement PipHeap and its combination with six sketches. Our source code is available at [5].

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No. 2024YFB2906602, and in part by the National Natural Science Foundation of China (NSFC) (No. 62372009, 624B2005).

A ETHICS

This work does not raise any ethical issues.

REFERENCES

[1] [n. d.]. FarmHash. https://github.com/google/farmhash.

[2] [n. d.]. Intel tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[3] [n. d.]. Murmur Hashing. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[4] [n. d.]. Real-Life Transactional Dataset. http://fimi.ua.ac.be/data/.

[5] [n. d.]. Source code of PipHeap. https://github.com/PipHeap/PipHeap.

[6] [n. d.]. The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[7] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. 2022. {HeteroSketch}: Coordinating Network-wide Monitoring in Heterogeneous and Dynamic Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 719–741.

[8] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 503–514.

[9] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[10] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. 2017. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 127–140.

[11] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 267–280.

[12] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. 1–12.

[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[14] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[15] Robert S Boyer and J Strother Moore. 1991. MJRTY: A Fast Majority Vote Algorithm. *Automated reasoning: essays in honor of Woody Bledsoe* 1 (1991), 105–117.

[16] Weibo CAI, Shulin YANG, Gang SUN, Qiming ZHANG, and Hongfang YU. 2023. Adaptive Load Balancing for Parameter Servers in Distributed Machine Learning over Heterogeneous Networks. *ZTE Communications* 21, 1, Article 72 (2023), 72-80 pages. https://doi.org/10.12142/ZTECOM.202301009

[17] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. *Automata, languages and programming* (2002).

[18] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. 2021. Out of many we are one: Measuring item batch with clock-sketch. In *Proceedings of the 2021 International Conference on Management of Data*. 261–273.

[19] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 226–239.

[20] Benoit Claise. 2004. *Cisco systems netflow version 9*. Technical Report.

[21] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005).

[22] Erik Demaine, Alejandro López-Ortiz, and J Munro. 2002. Frequency estimation of internet packet streams with limited space. *Algorithms—ESA 2002* (2002).

[23] Fan Deng and Davood Rafiei. 2007. New estimation algorithms for streaming data: Count-min can do more. *Webdocs. Cs. Ualberta. Ca* (2007).

[24] Nick G Duffield and Matthias Grossglauser. 2001. Trajectory sampling for direct traffic observation. *IEEE/ACM transactions on networking* 9, 3 (2001), 280–292.

[25] Cristian Estan and George Varghese. 2003. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)* 21, 3 (2003).

[26] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2019. {SIMON}: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 549–564.

[27] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on*. IEEE.

[28] Hui Han, Zheng Yan, Xuyang Jing, and Witold Pedrycz. 2022. Applications of sketches in network traffic measurement: A survey. *Information Fusion* 82 (2022), 58–85.

[29] Ping-Hsien Huang, Michael I.-C. Wang, Chi-Hsiang Hung, and Charles H.-P. Wen. 2024. Mitigating Microbursts by Packet Recirculation in Programmable Switch. *IEEE Access* 12 (2024), 183089–183102. https://doi.org/10.1109/ACCESS.2024.3510784

[30] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 113–126.

[31] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. 2021. Toward Nearly-Zero-Error Sketching via Compressive Sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/huang

[32] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 404–421.

[33] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 285–291.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[35] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. 2004. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review* 32, 1 (2004), 177–188.

[36] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. 2019. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication*. 130–144.

[37] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1574–1584.

[38] Shangsen Li, Lailong Luo, Deke Guo, Qianzhen Zhang, and Pengtao Fu. 2020. A survey of sketches in traffic measurement: Design, Optimization, Application and Implementation. *arXiv preprint arXiv:2012.07214* (2020).

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

[40] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 481–495.

[41] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM New York, NY, USA, 44–58.

[42] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 143–157.

[43] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 334–350.

[44] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM.

[45] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter braids: a novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (2008), 121–132.

[46] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proc. Springer ICDT*.

[47] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.

[48] Jayadev Misra and David Gries. 1982. Finding repeated elements. *Science of computer programming* 2, 2 (1982), 143–152.

[49] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[50] Michael Mitzenmacher. 2002. Compressed bloom filters. *IEEE/ACM transactions on networking* 10, 5 (2002), 604–612.

[51] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2022. {SketchLib}: Enabling Efficient Sketch-based Monitoring on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 743–759.

[52] Peter Phaal, Sonia Panchen, and Neil McKee. 2001. *InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks.* Technical Report.

[53] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*. 1449–1463.

[54] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. 2008. cSamp: A system for network-wide flow monitoring. (2008).

[55] Vyas Sekar, Michael K Reiter, and Hui Zhang. 2010. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 328–341.

[56] Devavrat Shah, Sundar Iyer, B Prahhakar, and Nick McKeown. 2002. Maintaining statistics counters in router line cards. *IEEE Micro* 22, 1 (2002), 76–81.

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

[58] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. 2014. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *2014 IEEE 34th International Conference on Distributed Computing Systems*. IEEE, 228–237.

[59] Daniel Ting. 2018. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*. 1129–1140.

[60] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 407–420.

[61] Qianglin WANG, Xiaoning ZHANG, Yi YANG, Chenyu FAN, Yangyang YUE, Wei WU, and Wei DUAN. 2025. VFabric: A Digital Twin Emulator for Core Switching Equipment. *ZTE Communications* 23, 1, Article 90 (2025), 90-100 pages. https://doi.org/10.12142/ZTECOM.202501012

[62] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. 1990. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990), 208–229.

[63] Mengkun Wu, He Huang, Yu-E Sun, Yang Du, Shigang Chen, and Guoju Gao. 2021. Activekeeper: An accurate and efficient algorithm for finding top-k elephant flows. *IEEE Communications Letters* 25, 8 (2021), 2545–2549.

[64] Qingjun Xiao, Zhiying Tang, and Shigang Chen. 2020. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 974–983.

[65] Kaicheng Yang, Yuanpeng Li, Zirui Liu, Tong Yang, Yu Zhou, Jintao He, Tong Zhao, Zhengyi Jia, Yongqiang Yang, et al. 2021. SketchINT: Empowering INT with TowerSketch for Per-flow Per-switch Measurement. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 1–12.

[66] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM SIGCOMM*. ACM, 561–575.

[67] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.

[68] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.

[69] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 207–220.

[70] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch.. In *NSDI*, Vol. 13.

[71] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 207–222.

[72] Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. 2021. SpaceSaving: An Optimal Algorithm for Frequency Estimation and Frequent items in the Bounded Deletion Model. *arXiv preprint arXiv:2112.03462* (2021).

[73] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. 2021. {LightGuardian}: A {Full-Visibility}, Lightweight, In-band Telemetry System Using Sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 991–1010.

[74] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. 2022. FlyMon: enabling on-the-fly task reconfiguration for network measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 486–502.

[75] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. 2020. Flow Event Telemetry on Programmable Data Plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 76–89.

[76] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 479–491.