QPipe: Quantiles Sketch Fully in the Data Plane

Nikita Ivkin*[†]
Amazon

Zhuolong Yu*
Johns Hopkins University

Vladimir Braverman
Johns Hopkins University

Xin Jin
Johns Hopkins University

ABSTRACT

Efficient network management requires collecting a variety of statistics over the packet flows. Monitoring the flows directly in the data plane allows the system to detect anomalies faster. However, monitoring algorithms have to handle a throughput of 10^9 packets per second and to maintain a very low memory footprint. Widely adopted sampling-based approaches suffer from low accuracy in estimations. Thus, it is natural to ask: "Is it possible to maintain important statistics in the data plane using small memory footprint?". In this paper, we answer this question in affirmative for an important case of quantiles. We introduce QPipe, the first quantiles sketching algorithm that can be implemented entirely in the data plane. Our main technical contribution is an on-the-plane implementation of a variant of SweepKLL [27] algorithm. Specifically, we give novel implementations of argmin(), the major building block of SweepKLL which are usually not supported in the data plane of the commodity switch. We prototype QPipe in P4 and compare its performance with a sampling-based baseline. Our evaluations demonstrate 10× memory reduction for a fixed approximation error and 90× error improvement for a fixed amount of memory. We conclude that QPipe can be an attractive alternative to sampling-based methods.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Network monitoring**.

KEYWORDS

Quantile, Sketch, Data plane, Programmable networks, Network monitoring

ACM Reference Format:
Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. QPipe: Quantiles Sketch Fully in the Data Plane. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3359989.3365433

*Equal contribution.
[†]This work was done when the author was at Johns Hopkins University.

1 INTRODUCTION

Collecting and monitoring essential statistics of network traffic drives the efficiency of the traffic management and adaptive control. Catching heavy flows [17], evaluating cardinality[21, 24], entropy [41] and other problems are at the core of dynamic flow scheduling [2, 30], attack detection [32], link congestion resolving [3] and a variety of other applications. Aggregating and processing these statistics directly in the data plane of the switch is an ultimate goal, that leads to a shorter response time and as a consequence more efficient use of limited networking infrastructure. However, widely adopted sampling-based solutions, though simple in implementation and logically straightforward, suffer from unacceptably low accuracy. Therefore in search of better memory/precision trade-offs networking community adopted a variety of sketching techniques from the field of streaming algorithms [25, 34]. However, due to the switch hardware limitations, sketch-based solutions stayed outside of the data plane for a while.

The recent introduction of programmable switches [4, 7, 14] paved a path to a new generation of algorithms standing beyond "hash and count" framework. Multistage architecture with each packet pipe-lined through all stages operates at the line rate of billions of packets per second. Such a throughput unavoidably enforces a constraint of limited operations on each stage per packet [45], in addition with a few megabytes of memory per stage is available [45], which is also shared with other infrastructural needs. In announced constraints, *out-of-the-box* use of sketching algorithms on programmable switches is not feasible. To the best of our knowledge, only a few sketches were adopted to the programmable switch architecture, among them HashPipe [47] designed to find "heavy hitters" flows. OpenSketch [49] provides a three-stage pipeline (hashing, filtering, and counting), which can support several measurement tasks is feasible to be implemented in the switch.

In the current manuscript, we challenge the problem of finding order statistics of the traffic in the programmable switch. Quantiles (median, 99th percentile, and others) and cumulative distribution function (CDF) help to understand the underlying structure of the traffic and detect both sharp short-term anomalies and long term change in the distribution. We target quantiles computation over a fixed number of updates, i.e. epoch, and adopt SweepKLL sketch [27] (close to optimal in both memory and update time). We implement our algorithm QPipe in P4, show in practice that it works at the line rate. In addition, we benchmark our solution with packet-sampling approach in terms of memory vs precision tradeoff and show how QPipe can be utilized to find heavy hitters with a significantly lower rate of false positives, compared to widely adopted Count Min Sketch algorithm [18]. We emphasize the challenge of implementing SweepKLL on the commodity switch, as it heavily utilizes such basic routines as sort() and argmin() which are not supported by the hardware of the switch. To overcome this challenge we utilize the design specifics of both switch and sketch. SweepKLL subsamples the stream before feeding it into the data structure, and only performs one memory access per packet. We take the best of both worlds by "*employing*" the unsampled packets to do the infrastructural work of maintaining the data structure.

2 STREAMING QUANTILES

Finding order statistics such as median, 95-th percentile, full CDF and others is crucial in many big data applications[15, 19, 42–44]. No surprise that finding quantiles in the streaming model is a well-studied problem with more than two decades of research since the pioneering work by Munro and Paterson [40]. In formal terms, the problem of streaming quantiles can be stated as follows. Algorithm observes a stream of n updates $s_1, \dots, s_n$ one at a time, then upon observance of value q at the end of the stream, it returns the rank $r(q)$ among items s_i, i.e. number of items s_i smaller than q. Similarly, if queried with the rank r, it returns r-th smallest item. However, any exact algorithm would require memory $poly(n)$ [40], therefore the main interest is in approximate version of the problem, that allows to return the rank $r(q)$ with additive error of εn.

The uniform sample preserves all the ranks with approximation $\pm\varepsilon n$ given the memory budget of $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$. Widely adopted sketches, such as GK [22], KLL [31] and Q-digest [46], showed a significant improvement in this trade-off, pushing space complexity down to the optimal $O(\frac{1}{\varepsilon}\log\log\frac{1}{\varepsilon})$. While many of them advantage from constant amortized update time, all suffer from poor worst case update time of $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$, which made it impossible to neither run at the line rate nor implement it fully in the data plane of the switch. Recently Ivkin *et al.* [27] presented a series of improvements to KLL sketch [31], among them is sweeping technique, that pushed the worst case update time for KLL down to $O(\log\frac{1}{\varepsilon})$. In the current manuscript we refer to this modification as SweepKLL and address the challenges of implementing it fully in the data plane of the switch. But first, we cover the concept of the compactor, a building block behind a series of sketches [1, 31, 35, 48]. For more details on the broader topic of streaming quantiles we refer reader to [23, 27, 48].

Consider a set of k numbers, rank function $r(x)$ is a step function making a step of 1 at every number in the set. Figure 1 depicts the example of the rank function $r(x)$ for the set of $k = 6$ numbers: $\{1, 2, 4, 6, 7, 9\}$. Note that one can compress k numbers into $k/2$ by: 1) sorting all numbers; 2) deleting all odd (or all even) positions in the sorted order; 3) assigning weight 2 to the rest. Given that every item left has weight 2, rank function $r'(x)$ makes a step size 2 at every number left in the set. On Figure 1, dashed line shows the rank function $r'(x)$ in case if all even positions deleted. Note that $\forall x : |r(x) - r'(x)| \leq 1$, i.e. the compression procedure introduced the rank error at most 1. The building block that inputs k items and outputs $k/2$ introducing the rank error of at most 1, called compactor. Note that if all input items have weight w, the compactor will output items of weight $2w$ and introduce rank error at most w.

The compactor can also be considered as a stream processor: it inputs a stream length n with item's weight w [1], and outputs a stream length $n/2$ with item's weight $2w$. It is basically a buffer that collects

[1]in the original stream all updates have weight 1

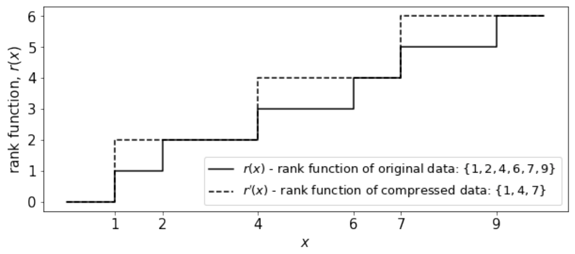


Figure 1: Example of the compaction procedure on the set $\{1, 2, 4, 6, 7, 9\}$: solid line shows the rank function for original set, dashed line shows the rank function after compression (deleting all even positions).

k items from the stream, sort them, delete every other, outputs $k/2$ of those which left with a double weight, then collects new k items from the stream and repeats. Manku *et al.* [35] suggested to use a stack of $O(\log\frac{n}{k})$ compactors each of size k: i-th compactor inputs the stream of length $\frac{n}{2^i}$ with items weight 2^i from the output of $(i-1)$-th compactor (or from the original stream if $i - 1 = 0$) and outputs the the stream of length $\frac{n}{2^{i+1}}$ with items weight 2^{i+1} into the $(i+1)$-th compactor. Note, that i-th compactor of size k while processing a stream of length $\frac{n}{2^i}$ performs $\frac{n}{k2^i}$ compressions (or compactions), each compaction introduces rank error of at most $w = 2^i$, therefore total rank error introduced by i-th compactor is at most $\frac{n}{k}$. And total rank error introduced by all compactors is $O(\frac{n}{k}\log\frac{n}{k})$. Setting $k = \frac{1}{\varepsilon}\log\varepsilon n$ brings the desired approximation of $\pm\varepsilon n$ with the space needed to store all the compactors $O(k\log\frac{n}{k}) = O(\varepsilon^{-1}\log^2\varepsilon n)$. Agarwal *et al.* [1] showed how method can be combined with sampling and suggested to flip a coin when choosing which (odd or even) positions to delete in the compactor. New algorithm required only $O(\frac{1}{\varepsilon}\log^{3/2}\frac{1}{\varepsilon})$ of space. Karnin *et al.* [31] pointed out the higher importance of the top compactors, and suggested the size of compactor to decrease exponentially: for the top compactor $k_H = O(\frac{1}{\varepsilon}\log^{1/2}\frac{1}{\varepsilon})$ and for the compactor $i : k_i = c^{H-i}k_H$, where $c \in (0.5, 1)$. All compactors of the size less than 2 can be dropped and replaced with a sampler. Karnin *et al.*[31] show that this approach (KLL sketch) drops the space complexity down to $O(\frac{1}{\varepsilon}\log^{1/2}\frac{1}{\varepsilon})$[2].

All aforementioned sketches inherit a poor worst-case update time of $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$, caused by: 1) necessity to sort the content of the compactor prior compressing; 2) updating weight for the half and deleting the other half of the items in the compactor, all at once. Recently, Ivkin *et al.*[27] suggested a series of improvements to KLL, among which is a sweeping technique, that de-amortizes the running time of KLL, pushing the worst-case update time down to $O(\log\frac{1}{\varepsilon})$. To explain the idea behind it, we recall that a compactor of size k can be considered as $k/2$ pairs in sorted order, such that in each pair only one item survives the compaction procedure. It is crucial that pairs do not intersect, i.e. given set $\{1, 2, 4, 6\}$ we need to break down into pairs as $\{(1, 2), (4, 6)\}$ rather than $\{(1, 4), (2, 6)\}$, as former case introduces rank error 1 and latter case rank 2. That is the sole reason, why sorting and compressing all k at once is required, however [27] suggested the way around it: compress one

[2][31] also present stronger result with $O(\frac{1}{\varepsilon}\log\log\frac{1}{\varepsilon})$, however, it is theory driven and more complicated implementation-wise

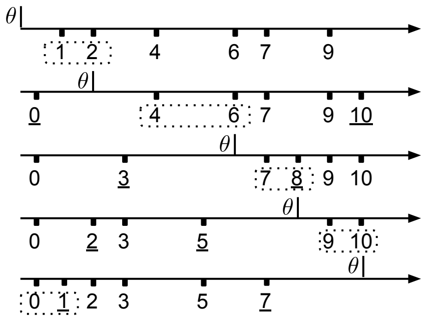


Figure 2: High level idea behind SweepKLL: dashed box indicates the pair chosen for the compression at current moment, underlining indicates the new items just appeared in the compactor.

Algorithm 1 SweepKLL

```
1: function SweepKLL.update($key$)
2:     if Sampler() then $A[0]$.append($key$)
3:     for $h = 0 \dots H-2$ do
4:         if len($A[h]$) > $kc^{H-h}$ then $A[h]$.compact()
5: function $A[h]$.compact( )
6:     $A[h]$.sort();   $i$ = argmin$_i$($A[h][i] \geq A[h].\theta$);
7:     if $i$ == None then
8:         $i$ = 0; random_bit = random([0, 1]);
9:     $A[h].\theta = A[h][i+1]$
10:    pair = $A[h]$.pop([$i$, $i+1$])
11:    $A[h+1]$.append(pair[random_bit])
```

pair at a time, at the same time keep track that every new pair does not intersect with already compressed ones. SweepKLL maintains the compactor's content in a sorted heap, i.e. adding new item costs at most $O(\log \frac{1}{\varepsilon})$. To avoid intersection problem, it keeps track of the largest element in all compressed pairs so far (call it θ), and chooses the smallest pair above that threshold. Figure 2 depicts the idea on the set $\{1, 2, 4, 6, 7, 9\}$, pair compression is called only when compactor is full, newly arrived items are underlined. θ is initialized with $-\infty$ and first pair to compact is $(1, 2)$, which updates $\theta = 2$. New items 0 and 10 arrived, but note that SweepKLL chooses $(4, 6)$ rather than $(0, 4)$ because of $0 < \theta = 2$. This procedure repeats until no items larger than θ left, then current *sweep* is finalized and θ is reset to $-\infty$. See Algorithm 1 for detailed pseudocode for sweepKLL. As mentioned earlier, SweepKLL maintains the content of the compactor in the heap, therefore finding the next pair to compress takes at most $O(\log \frac{1}{\varepsilon})$ time.

3 QUANTILES IN DATA PLANE

3.1 Switch Architecture

Networking switch typically consists of two logical parts: data plane and control plane. Packets stream and get processed entirely in the data plane at the line rate of 10^9 packets per second. To maintain the line rate data plane has very restrictive architecture, limiting the number of memory accesses per packet and the number of arithmetic operations. On the contrary, the control plane can be considered as a regular server, most streaming solutions can be implemented out of the box there. However, the communication channel between the control plane and the data plane is very limited and ranges at

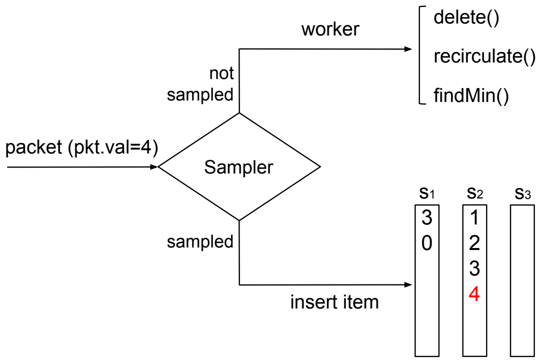


Figure 3: QPipe samples packets and insert sampled packets into the array a_0 (stored in stage S_2). Unsampled packets will work as "workers" to do certain operations.

10^5 packets per second. As a consequence, all monitoring software implemented on the control plane is forced to operate with heavily subsampled data, which leads to poor accuracy and delayed reaction in case of attack or anomaly in the traffic flow. This drives the main motivation for implementing monitoring tasks entirely on the data plane. Further, we go over the high-level architecture and main restrictions in the data plane of the networking switch.

In recent years, the emergence of programmable switches [4, 14, 26] enables programmability and enriches the operations on the data plane. Based on the Protocol Independent Switch Architecture (PISA), most commodity programmable switches (Barefoot Tofino [4], Cavium XPliant [14], Intel Flexpipe [26]) inherit a similar pipeline, with the packet going via a certain number of stages (match-action tables, memory, arithmetic logic units). Each stage has independent memory and due to the necessity of maintaining the line rate, it restricts each packet to access (read/update) only a limited number of memory registers. On the brighter side, a stage can attach some values to the packet metadata, and any following stages in the pipeline can access it. In other terms, a stage can send signals down the pipeline in the metadata of the packet. In addition, any stage can request to recirculate the packet, i.e. send it to the first stage again. This operation is the only way to send signals to earlier stages, however, it creates an additional traffic load and its usage should be minimized.

3.2 Data Plane Design

Vanilla version SweepKLL maintains the content of each compactor in a sorted heap. There are two natural ways to implement the heap entirely in the data plane: store the binary heap on the same stage or store one level of the binary heap per stage. The former method is infeasible, as swapping the items requires access to two memory registers at the same time, while the latter method is prohibitively expensive due to the high number of recirculations involved for each swap. The alternative route of sorting compactor's content on-demand follows the same issues. Without sorted order finding a new pair to compress is challenging. In the current section, we address these challenges and implement QPipe, algorithm finding order statistics in the data plane of the switch.

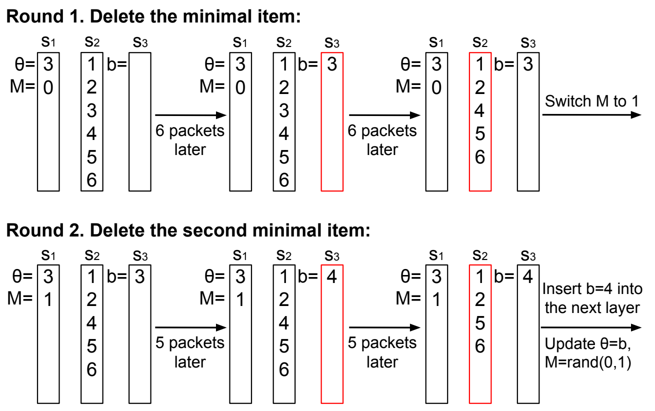


Figure 4: An example of QPipe deleting and moving items. The first round uses 12 "worker" packets to delete the minimal item. The second round uses 10 packets to delete the second minimal item and insert the deleted item into the next layer.

Algorithm 2 QPipe ()

```
1: Sample the packet with probability $1/K$.
2: if Packet is sampled then
3:     $a_0$.insert($pkt.v$)
4: else
5:     if $a_i$ is full then
6:         findMin(), delete().
7:         findMin(), delete().
8:         Randomly pick one deleted item $v'$, $a_{i+1}$.insert($v'$).
```

Algorithm 3 findMin()

```
1: //* Find the minimal value larger than $\theta$
2: $pkt.\theta \leftarrow \theta$                          ▷ Stage 1
3: if $a[i] \geq pkt.\theta$ then                     ▷ Stage 2
4:     $pkt.v \leftarrow a[i]$
5: if $pkt.v < b$ then                             ▷ Stage 3
6:     $b \leftarrow pkt.v$
7: if finish then
8:     recirculate($pkt.b \leftarrow b$)
```

As already mentioned above SweepKLL requires either sorting a register array or getting the minimal value of a register array. Although both operations are not supported in commodity switches, we circumvent the need of these primitives with a pragmatic choice: employing "*worker*" packets to help to maintain the data structure.

QPipe repeats the logic of SweepKLL and subsample the packets before adding them to the first compactor. However, it also employs the unsampled packets (or "*workers*") to maintain the data structure properties. As shown in Figure 3, QPipe samples the packets, and feeds the value of the sampled packets into the data structure. Since the packets that are not sampled will still go through all the stages in a pipeline to be routed, we use these unsampled packets as "workers" to carry some values and finish certain operations. With the help of these "workers", we are able to implement QPipe in the data plane in a suitable manner.

Programmable switches allow users to develop custom data plane modules, which can parse custom packet headers, perform user-defined actions, and access the switch on-chip memory for stateful operations. Based on PISA pipeline of Barefoot Tofino Switch [4], we use a number of stages to store values and perform certain operations. We leverage a primitive action *modify_field_rng_uniform* in P4 to generate a random number as a sampling indicator to sample packets. The sampled items are inserted into the array in the first layer (a_0). When the array (a_i) of layer i becomes full, QPipe will select two minimal items larger than a predefined value θ. Among the two items, QPipe randomly deletes one item and moves the other item to the next layer (a_{i+1}). We denote the process of deleting one item or moving one item as one *round*. QPipe leverages a set of successive "worker" packets to finish one *round*.

Figure 4 shows a concrete example of deleting and moving items. Note that, we only show the three main stages here for simplicity, but the real QPipe system requires more stages (12 stages used in our implementation) due to the constraints of the switch pipeline. Stage S_1 stores the value of θ and M. M is an indicator indicating whether an item will be inserted into the next layer. After the removal of the minimal item in the current layer (a_i), the removed item will be inserted into the next layer (a_{i+1}) if and only if M equals 1. M will switch from 1 to 0 or from 0 to 1 upon the completion of each round. θ and M are updated every two rounds: θ is set as the second minimal item (which is just deleted or moved) and M is reset randomly from $\{0, 1\}$. Stage S_2 stores the arrays. Stage S_3 maintains the minimal value of the current array which is larger than θ, *i.e.*$min\{a_i[j] \mid a_i[j] \leq \theta, j \in [0, a_i.len)\}$.

In the example, the value θ is set as 3, and the random indicator M is set as 0. We are going to find two minimal items larger than 3, delete the first (smallest) one ($M = 0$ in the first round) and move the second one to the next layer ($M = 1$ in the second round). The array of the current layer becomes full because of the insertion of the six items: (1, 2, 3, 4, 5, 6). QPipe uses a set of "worker" packets to maintain the data structure. In the first round, QPipe deletes the minimal item larger than θ of the array. Firstly, 6 "worker" packets are used to scan every item in the array and compare it with θ and b. The minimal value ($b = 3$) larger than θ is stored in stage S_3. Then, QPipe uses another 6 packets to check every item and delete the one which matches value b. Note that: (1) In our real implementation, we use one resubmit action to carry the final value of b to the first stage and store it there; (2) To delete one item, QPipe swap it with the item at the head of the array and move the head pointer forward. For simplicity, we omit these details in this example.

Upon the deletion of the item, indicator M is switched to 1 and the second round begins. In the second round, QPipe deletes the next minimal item larger than θ from the current layer and insert it into the next layer. Similar to the procedure of the first round, QPipe uses 5 "worker" packets to get the minimal value larger than θ ($b = 4$). QPipe uses the next 5 "worker" packets to delete the corresponding item. As M equals to 1 here, QPipe will insert the deleted item to the next layer (The same deletion process will be triggered if the next layer gets full). At last, QPipe will update θ to value b and set M as 0 or 1 randomly. Note that, if b reaches the maximum of the array, value θ will be reset to the initial value (*i.e.*lower bound) instead of b. More details are provided in Algorithm 2–5.

Algorithm 4 delete()

```
$pkt.b \leftarrow b'$                              ▷ Stage 1
if $a[i] == pkt.b$ then                            ▷ Stage 2
    $pkt.i \leftarrow i$
if finish then                                     ▷ Stage 3
    remove($pkt.i$)
```

Algorithm 5 recirculate()

```
$b' \leftarrow pkt.b$                              ▷ Stage 1
$\theta \leftarrow pkt.b$
```

4 EVALUATION

In this section, we evaluate QPipe. We compare the performance of QPipe with a sampling based baseline solution. In addition, we show how QPipe can be used to identify "heavy hitters" flows, and benchmark its performance with sampling and widely adopted Count-Min sketch [18].

Experiment setup. We evaluate QPipe's performance using three sets of traces. The first trace, denoted by (a), comes from a large-scale traceroute-based measurement on the Archipelago (Ark) measurement infrastructure [12]. We use the second trace [11] as an application study, which contains DNS round-trip time (RTT) information, and is denoted by (b). The third trace, denoted by (c), is from a monitoring work on high-speed Internet backbone links [13]. While we implement QPipe in P4 on a Barefoot Tofino switch (code for BMV2 available at https://github.com/netx-repo/QPipe), the experiments are conducted as simulations in python on a server with 8-core CPU (Intel Xeon E5-2620 2.1GHz) for ease of comparison. QPipe maintains a memory-efficient data structure in the data-plane. The switch is able to run QPipe at line rate as long as QPipe can be compiled and fit the switch resources. We wrap the registers and operations of each stage from P4 into a class in our python simulation ensuring that each register is accessed only at its own stage. Metadatas (e.g. meta and recirculate_hdr in Algorithm. 6) in P4 are treated as variables visible to all the classes in the simulation. We conduct the simulation strictly follow the constraint of switch's pipeline. A simple example of a class of setting and getting the value of the register is shown in Algorithm. 6. We focus on evaluating the accuracy of QPipe.

We investigate the accuracy of QPipe by calculating the average and maximum approximation error [3] by using trace (a) in Section. 4.1 and trace (b) in Section. 4.2. We also show the performance of QPipe on finding the heavy hitters in Section. 4.3 by using trace (c).

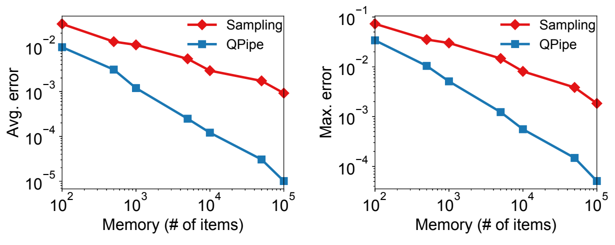
4.1 Accuracy of QPipe

We show the performance improvement of QPipe over sampling-based baseline solution. We use source IP address as the key and identify the quantiles. Figure 5 shows the space versus approximation error trade-off, for both average absolute error (Figure 5(a)) and maximum absolute error (Figure 5(b)). Since the programmable switch can provide approximately a few MB memory size per stage [45], we evaluate the solutions with the array size from 100 to 100K. Both the vanilla sampling solution and QPipe can take advantage of larger

[3]approximation error is defined as the absolute difference between the estimated quantile and the real quantile

Algorithm 6

```
class stage_x:
    def __init__(self, len):
        self.a_register = [0 for i in range(len)]
        return
    def _get_head_value(self):
        return self.a_register[meta.head]
    def _push_value(self):
        self.a_register[meta.tail] = meta.value
        return
    def _set_value_beta(self):
        self.a_register[recirculate_hdr.index_beta_ing] = meta.value
        return
    def _set_value_gamma(self):
        self.a_register[recirculate_hdr.index_gamma_ing] = meta.value
        return
    def _fetch_item(self):
        if (self.a_register[meta.filter_index] > meta.theta):
            return self.a_register[meta.filter_index]
```



(a) Avg. error. (b) Max. error.

Figure 5: Performance comparison of QPipe and Sampling under different memory size in trace (a) with source IP address as the key.

memory size to reduce the approximation error. However, from the figure, we can observe that the approximation error of QPipe follows more favorable asymptotics and provide up to 91.09× better approximation compared to the sampling-based solution under the same memory constraints. When the memory is able to store 100 (100K) items, QPipe outperforms basic sampling by 3.47× (91.09×) and 2.15× (36.15×) on average approximation error and maximum approximation error respectively. Meanwhile, to achieve the same level of approximation error, QPipe can save about 90% of memory compared with sampling. Current evaluation aims to compare QPipe with the only available alternative implementable fully in the data plane, i.e. sampling. Though there exist other quantiles sketches, their efficient implementation in the data plane is a subject of future research. For a detailed evaluation of those sketches on a regular server, we refer reader to [27, 48]. In addition, we emphasize that the sampling rate within QPipe (as well as KLL) is driven by memory constraints: increasing sampling rate under fixed memory would not influence theoretical guarantees.

4.2 Application Study: Round-trip time

End-to-end delay captures network service degradations caused by various reasons. Therefore it is crucial to efficiently monitor this vital network performance factor [16, 20]. Here, we present the QPipe's performance on analyzing round-trip time (RTT) as an application

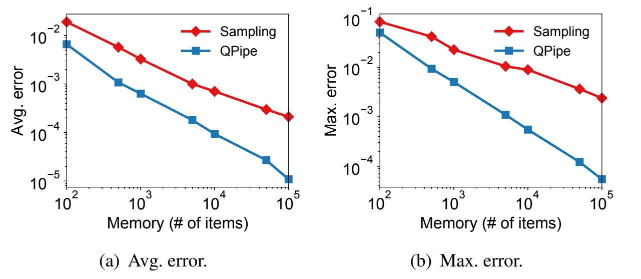


Figure 6: Performance comparison of QPipe and Sampling under different memory size in trace (b) with RTT as the key.

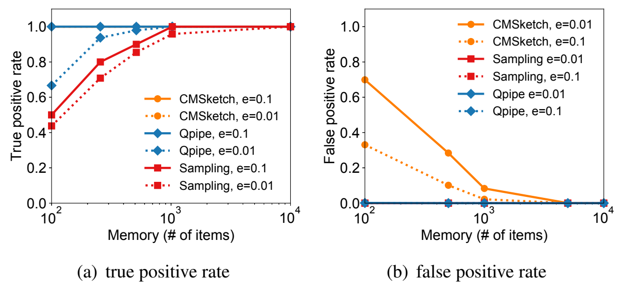


Figure 7: Performance comparison of QPipe, Sampling and Count-Min Sketch under different memory size for finding heavy hitters in trace (c) with source IP address as the key.

study, and compare it with sampling. Figure 6 shows approximation error vs. memory size. Figure 6(a) and Figure 6(b) present the average approximation error and maximum approximation error respectively. With the help of the efficient data structure, QPipe reduces the average approximation error and maximum approximation error by 2.89×-19.33× and 1.65×-43.89× respectively compared to sampling. We can see that in practice the approximation error of QPipe follows theoretical guarantees and significantly outperforms sampling on the given range and asymptotically.

4.3 Finding Heavy Hitters

We evaluate the performance of QPipe on finding the heavy hitters. QPipe finds heavy hitters by calculating each item's approximate proportion. More specifically, QPipe queries items at $x\%$ and $(x\%+\epsilon)$ respectively. If the two items are equal, denoted as v, then v is a ϵ-heavy hitter.

We compare QPipe with sampling and Count-Min sketch on trace (c) using source IP address as the key, and show the true positive rate (TPR) and false positive rate (FPR). Figure 7(a) shows that QPipe is more accurate at finding heavy hitters than sampling, especially when the memory size is small. When the memory can store 100 items, the TPR of QPipe is about 66.67% (100.0%) for 0.01-heavy hitter (0.1-heavy hitter) while the TPR of sampling is about 43.75% (50.0%). When given enough memory (100K items), QPipe will successfully find all the 0.01-heavy hitters and 0.1-heavy hitters, while sampling will still miss items for 0.01-heavy hitters.

Among the three methods, Count-Min sketch has the highest TPR as it won't miss heavy items (The yellow lines <CMSketch, e=0.1> and <CMSketch, e=0.01> are both at line <TPR = 1.0>, hidden by the blue solid line <QPipe, e=0.1>). However, Count-Min sketch allows a high false positive rate. As figure 7(b) shows, Count-Min sketch has the false positive rate up to 69.8%, while the FPR of QPipe is only up to 0.11%.

5 RELATED WORK

Network measurement has been a crucial area in network research for a long time [5, 36, 37]. With the emergence of software-defined networking and programmable data-plane [4, 7, 14], there has been a lot of work about monitoring jobs [25, 47, 49] and other applications [28, 29, 33, 38] in the data plane. OpenSketch [49] defines several APIs for general sketch-based measurement tasks running in commodity switches. Liu *et al.*[34] proposed a "one-big switch" abstraction for monitoring UnivMon for the management applications to run atop of. UnivMon was based on the concept of "universal sketches" [6, 8–10] and was presented with a proof-of-concept using P4. SCREAM [39] dynamically allocates resources to many sketch-based measurement tasks and ensures a user-specified minimum accuracy. Huang *et al.*[25] designed and implemented SketchVisor on top of Open vSwitch. SketchVisor augments sketch-based measurement in the data plane with a fast path and recovers accurate network-wide measurement results via compressive sensing. Hashpipe [47] was a prototype in P4 to detect heavy hitters entirely in data plane. To the best of our knowledge, QPipe is the first prototype that can efficiently report quantiles in data plane.

6 CONCLUSION

In this paper, we present QPipe, to the best of our knowledge, the first quantiles sketching algorithm implemented entirely in the data plane. We properly address the challenge of implementing sophisticated operations in data plane by using "worker" packets. We show 90× improvement in precision under a fixed memory budget, compared with sampling-based baseline.

ACKNOWLEDGMENTS

We thank our shepherd Ramakrishnan Durairajan and the anonymous reviewers for their valuable feedback. This work is supported in part by NSF grants CRII-1755646, CNS-1813487 , CCF-1918757 and CAREER-1652257, ONR Award N00014-18-1-2364, Lifelong Learning Machines program from DARPA/MTO, Facebook Communications Networking Research Award, and Amazon AWS Cloud Credits for Research Program.

REFERENCES

[1] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. 2013. Mergeable summaries. *ACM Transactions on Database Systems (TODS)* 38, 4 (2013), 26.
[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. 2010. Hedera: dynamic flow scheduling for data center networks.. In *Nsdi*, Vol. 10.
[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 503–514.

[4] Barefoot. 2019. Barefoot Tofino. (2019). https://www.barefootnetworks.com/technology/#tofino.

[5] Steven Bauer, Robert Beverly, and Arthur Berger. 2011. Measuring the state of ECN readiness in servers, clients, and routers. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 171–180.

[6] Jaroslaw Blasiok, Vladimir Braverman, Stephen R. Chestnut, Robert Krauthgamer, and Lin F. Yang. 2017. Streaming Symmetric Norms via Measure Concentration. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2017)*. ACM, New York, NY, USA, 716–729. https://doi.org/10.1145/3055399.3055424

[7] 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.

[8] Vladimir Braverman and Stephen R Chestnut. 2014. Universal sketches for the frequency negative moments and other decreasing streaming sums. *arXiv preprint arXiv:1408.5096* (2014).

[9] Vladimir Braverman, Stephen R Chestnut, David P Woodruff, and Lin F Yang. 2016. Streaming space complexity of nearly all functions of one variable on frequency vectors. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 261–276.

[10] Vladimir Braverman, Rafail Ostrovsky, and Alan Roytman. 2015. Zero-One Laws for Sliding Windows and Universal Sketches. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Naveen Garg, Klaus Jansen, Anup Rao, and José D. P. Rolim (Eds.), Vol. 40. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 573–590. https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2015.573

[11] CAIDA. 2009. The CAIDA DNS root/gTLD RTT Dataset. (2009). https://www.caida.org/data/passive/dns_root_gtld_rtt_dataset.xml.

[12] CAIDA. 2014. IPv4 Routed /24 DNS Names Dataset. (2014). https://www.caida.org/data/active/ipv4_dnsnames_dataset.xml.

[13] CAIDA. 2016. The CAIDA Anonymized Internet Traces 2016 Dataset. (2016). http://www.caida.org/data/passive/passive_2016_dataset.xml.

[14] cavium. 2019. Cavium XPliant. (2019). https://www.cavium.com/.

[15] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[16] Baek-Young Choi, Sue Moon, Rene Cruz, Zhi-Li Zhang, and Christophe Diot. 2007. Quantile sampling for practical delay monitoring in Internet backbone networks. *Computer Networks* 51, 10 (2007), 2701–2716.

[17] Graham Cormode and Marios Hadjieleftheriou. 2010. Methods for finding frequent items in data streams. *The VLDB Journal* 19, 1 (2010), 3–20.

[18] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[19] David J DeWitt, Jeffrey F Naughton, and Donovan A Schneider. 1991. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and distributed information systems, 1991., proceedings of the first international conference on*. IEEE, 280–291.

[20] Kelvin Ross Edmison, Hans Frederick Johnsen, and Walter Joseph Carpini. 2006. Method and system of measuring latency and packet loss in a network by using probe packets. (October 24 2006). US Patent 7,127,508.

[21] Éric Fusy and Frécéric Giroire. 2007. Estimating the number of active flows in a data stream over a sliding window. In *Proceedings of the Meeting on Analytic Algorithmics and Combinatorics*. Society for Industrial and Applied Mathematics, 223–231.

[22] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, Vol. 30. ACM, 58–66.

[23] Michael B Greenwald and Sanjeev Khanna. 2016. Quantiles and equi-depth histograms over streams. In *Data Stream Management*. Springer, 45–86.

[24] Hazar Harmouch and Felix Naumann. 2017. Cardinality estimation: An experimental survey. *Proceedings of the VLDB Endowment* 11, 4 (2017), 499–512.

[25] 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.

[26] Intel. 2019. Intel Flexpipe. (2019). https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[27] Liberty Edo Lang Kevin Karnin Zohar Ivkin, Nikita and Vladimir Braverman. 2019. Streaming Quantiles Algorithms with Small Space and Update Time. *arXiv preprint* (2019).

[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*.

[29] 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 *ACM SOSP*.

[30] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic scheduling of network updates. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 539–550.

[31] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*. IEEE, 71–78.

[32] Ramana Rao Kompella, Sumeet Singh, and George Varghese. 2004. On scalable attack detection in the network. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 187–200.

[33] 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 *USENIX FAST*.

[34] 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 ACM SIGCOMM Conference*. ACM, 101–114.

[35] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. 1998. Approximate medians and other quantiles in one pass and with limited memory. In *ACM SIGMOD Record*, Vol. 27. ACM, 426–435.

[36] Peter V Marsden. 1990. Network data and measurement. *Annual review of sociology* 16, 1 (1990), 435–463.

[37] Peter V Marsden. 2005. Recent developments in network measurement. *Models and methods in social network analysis* 8 (2005), 30.

[38] 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 *ACM SIGCOMM*.

[39] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2015. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 14.

[40] J Ian Munro and Mike S Paterson. 1980. Selection and sorting with limited storage. *Theoretical computer science* 12, 3 (1980), 315–323.

[41] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. 2008. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. ACM, 151–156.

[42] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* 13, 4 (2005), 277–298.

[43] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. 1996. Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, Vol. 25. ACM, 294–305.

[44] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 23–34.

[45] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI*.

[46] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. 2004. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 239–249.

[47] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. 2016. Smoking out the heavy-hitter flows with hashpipe. *arXiv preprint arXiv:1611.04825* (2016).

[48] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. 2013. Quantiles over data streams: an experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 737–748.

[49] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 29–42.