One Sketch is Enough: Accurate Per-flow Tail Latency Estimation with SketchPolymer

Jiarui Guo, Yuqi Dong, Yuhan Wu, Yisen Hong, Yunfei Liu, Xiaolin Wang,
Yong Cui, *Member, IEEE*, Bin Cui, *Fellow, IEEE*, and Tong Yang, *Member, IEEE*

***Abstract*—Nowadays, network measurement has attracted a lot of attention for its importance in monitoring and improving the reliability of modern networks. In this paper, we focus on tail latency estimation, which captures the behavior of the latency distribution at its extremes. Real network systems require the ability to measure not only aggregate tail latency (*i.e.* the tail latency of the entire network), but also per-flow tail latency (*i.e.* the tail latency of a certain flow). However, existing algorithms are mostly designed for aggregate tail latency estimation, and they fail to address per-flow tail latency estimation efficiently. In this paper, we propose a novel sketch, namely SketchPolymer, to accurately estimate per-flow tail latency. SketchPolymer uses a technique called Early Filtration to filter small flows, and another technique called Latency Splitting and Sharing to reduce error. Our experimental results show that the accuracy of SketchPolymer is on average 1.91 times better than state-of-the-art techniques. We also implement SketchPolymer on P4 and FPGA platforms to verify its deployment flexibility. All our code is available at GitHub.**

***Index Terms*—Tail latency estimation, network measurement, data streams, sketch, per-flow measurement.**

I. Introduction

A. Background and Motivation

Network measurement plays a crucial role in modern web services. Many domains, including web service monitoring [2], [3], network streaming [4], [5], and CDN load balancing [6], [7] demand real-time network measurement results to improve task efficiency. Existing network measurement efforts primarily focus on estimating flow sizes [8], [9], detecting heavy hitters [10], [11], and identifying pattern-based flows [12], [13]. These measurement tasks have greatly advanced our ability to monitor, analyze, and optimize network behavior in real time. Thus, lightweight, high-speed telemetry is indispensable for ensuring responsiveness, scalability, and adaptability in dynamic network scenarios [14], [15].

Manuscript received May 12, 2025; accepted June 12, 2026. This work was supported by the National Key Research and Development Program of China under Grant 2024YFB2906603, and in part by the National Natural Science Foundation of China (NSFC) under Grant 62372009 and Grant 624B2005. (Corresponding author: Tong Yang)

Jiarui Guo, Yuqi Dong, Yuhan Wu, Yisen Hong, Yunfei Liu, Xiaolin Wang, Bin Cui, and Tong Yang are with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China. Email: {*ntguojiarui, yuhan.wu, eason18, yunfei_imne, wxl, bin.cui, yangtong*}@pku.edu.cn; 2200012812@stu.pku.edu.cn.

Yong Cui is with the Computer Science Department, Tsinghua University, Beijing 100084, China. Email: cuiyong@tsinghua.edu.cn.

The preliminary version of this paper, titled "SketchPolymer: Estimate Per-item Tail Quantile Using One Sketch", was published in Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23) [1].

With the continuous growth of users and the increasing complexity of modern network infrastructure, **latency** has become a critical metric in network performance evaluation [16]. A lot of network applications, such as video streaming, real-time communication, and collaborative services places high demands on latency [17]–[19]. **Tail latency**, which reflects the distribution of latency at its extremes, is an effective metric for monitoring and understanding network performance under different conditions [20], [21]. In this paper, we consider the **per-flow tail latency** estimation problem in data stream models, where each item is associated with a latency, and we aim to estimate the tail latency for every distinct flow.[1] To illustrate the significance of per-flow tail latency estimation, we now present several use cases to show the practical applications:

Case 1: Quality of Service Evaluation. Per-flow tail latency plays an important role in evaluating network quality of service (QoS) [22], especially in large-scale systems where latency varies across flows. For example, persistently high per-flow tail latency in a service region may indicate congestion, suggesting the need for more servers to improve per-client quality of service. Real-time estimation of per-flow tail latency allows providers to detect degraded user experience promptly and take corrective actions to meet QoS targets.

Case 2: SLA Compliance Monitoring. Service Level Agreements (SLAs) define performance guarantees made by service providers, and tail latency is one of the key metrics within these agreements [23], [24]. Service providers can ensure that the network complies with SLA requirements by monitoring per-flow tail latency, thereby delivering high-quality service to users.

Case 3: Anomaly Detection. The sudden increase in per-flow latency can imply an anomaly in networks, such as potential cyber attacks or offending activities [25]–[27]. For example, a web service under DDoS attack will suffer from a sudden increase in response time. However, the aggregate tail latency can remain unchanged, as millions of packets with low latency can conceal this phenomenon. As a result, we can detect the anomaly through per-flow tail latency estimation.

Case 4: Time-Sensitive Network Assurance. Time-sensitive network (TSN) is designed to support deterministic communication with strict latency and jitter guarantees, especially in industrial automation, aerospace, and automotive systems [28], [29]. In such environments, missing tight latency bounds even for a small proportion of packets can lead to control system

[1]The formal definition of per-flow tail latency estimation is given in Section II-A.

instability or safety risks. Per-flow tail latency estimation enables granular monitoring of worst-case delays for each flow, which is essential for ensuring latency compliance and facilitating timely interventions when deviations occur.

Per-flow tail latency estimation is fundamental and critical in these cases. However, little prior work focuses on estimating per-flow tail latency. Some techniques can estimate tail latency [30], [31], but they mainly focus on estimating aggregated tail latency, and they do not distinguish different flows in data streams. In theoretical computer science, state-of-the-art algorithms for quantile estimation achieve high accuracy in theory [32]–[37], but they are designed to estimate arbitrary quantiles (arbitrary latency) rather than tail quantile (tail latency). Therefore, they preserve information across the latency distribution, which is useful for general latency analysis but is not optimized for per-flow tail latency monitoring under tight memory and throughput constraints.

Fortunately, approximate streaming algorithms, namely sketching algorithms, can be applied to solve this challenging problem in data stream models. Sketches can fulfill the need to filter small flows [38]–[41], and they can maintain the full information of large flows with small memory [8], [42]–[44], which is important in data stream models. Sketches have already been used for quantile estimation in data streams [45]–[47], and we design a new sketch for high-accuracy per-flow tail latency estimation.

B. Our Proposed Solution

In this paper, we propose a new sketch, called SketchPolymer, for estimating per-flow tail latency. SketchPolymer is memory-efficient: It is compact enough to be placed in the L3 cache. SketchPolymer is accurate: Our experiments show that SketchPolymer achieves 0.1 Average Logarithm Error while serving 20 million items simultaneously with 5000KB memory. SketchPolymer is fast: It takes $O(1)$ time to process each item.

SketchPolymer includes 4 stages. Filter Stage is used to filter small flows. Polymer Stage records the frequency and maximum latency of every large flow, and Splitting Stage and Verification Filter keep the detailed information of these flows. The key techniques used in SketchPolymer are named **Early Filtration** and **Latency Splitting and Sharing**. We show these techniques below.

Key Technique I: Early Filtration. Small flows account for the majority of data streams [48]–[50], but their tail latency cannot be accurately estimated due to their low frequency. As a result, it is important to filter these small flows to make room for large flows. We design Filter Stage, a lightweight filtering module inspired by Cold Filter [39] but customized for the tail latency estimation setting. For an incoming item e, we first query Filter Stage to check its frequency. If its frequency exceeds a predefined threshold $\mathcal{T}$, we start inserting it into the following stages; otherwise, we simply insert it into Filter Stage and return. Our experimental results show that allocating a small proportion of memory for Filter Stage can significantly lower the error of query results (see Section V for more details).

Key Technique II: Latency Splitting and Sharing (LSS). A naive solution to per-flow tail latency estimation is to record the full latency value for each item, but this leads to high memory usage and poor generalization across flows. To reduce memory overhead while preserving the accuracy of tail latency estimation, we propose a logarithmic quantization approach that splits all positive latency values into several disjoint intervals. It works as follows: for an incoming item e with latency t, we calculate $T = \lfloor \log_a t \rfloor$ as its logarithm latency, where a is a predefined parameter for SketchPolymer, and record T in our data structure instead of t. By LSS, we split large flows into several small sub-flows, and items in the same sub-flow share the same logarithm latency. In this way, we convert the latency estimation problem to a frequency estimation problem, and we record frequency rather than latency in SketchPolymer, which is more efficient and accurate and can be solved using CMSketch [8] and Bloom Filter [51].

C. Key Contributions

This paper makes the following contributions:

- We propose a novel data structure, namely SketchPolymer, which can automatically separate large flows from small flows and record the information of the former for accurate per-flow tail latency estimation.
- We provide rigorous mathematical analysis for SketchPolymer to theoretically derive its error bound and time complexity.
- We conduct extensive experiments on different datasets. The results show that SketchPolymer outperforms existing algorithms by 91% in terms of error.
- We implement SketchPolymer on various platforms and verify its performance on both software and hardware platforms.

II. Problem Statement and Related Work

A. Problem Statement

The symbols frequently used in this paper and their meanings are shown in Table I.

Definition 1. ***Data Stream.*** *A data stream $\mathcal{S}$ is a series of items $\{e_1, e_2, \cdots, e_n, \cdots\}$ appearing in sequence. In this paper, every item has a latency t, which can be defined as the time from request to response, or the time interval between two consecutive arrivals, etc..*

Definition 2. ***Quantile.*** *Given a multiset of numbers $S = \{a_1, a_2, \cdots, a_n\}$ and a percentage w, where $a_1 \leq a_2 \leq \cdots \leq a_n$ and $0 \leq w \leq 1$, the w-quantile of multiset S is defined as $a_{\lfloor w(n-1) \rfloor + 1}$.*

Per-flow Tail Latency Estimation: Given an arbitrary item (flow ID) e and a quantile w (usually close to 1, *e.g.*, 0.95 or 0.99), the design goal of SketchPolymer is to estimate the w-quantile latency of flow e.

TABLE I. Symbols frequently used in this paper.

Notation	Meaning
e	A distinct item in data streams
t	Latency of a certain item
T	Logarithm latency
a	Base of logarithm in SketchPolymer
$\mathcal{T}$	Threshold for Filter Stage
$d^{(k)}$	Number of hash functions in Stage k
$n^{(k)}$	Number of counters in Stage k
$h_i^{(k)}(.)$	i^{th} hash function in Stage k
$\mathcal{C}_i^{(k)}$	i^{th} counter array in Stage k

B. Related Work

1) Latency Estimation:

Latency is usually defined as the time it takes for a packet to pass from one place to another in network scenarios [52]. Based on different situations, latency can also be used to measure the time from request to response. Due to the importance of latency estimation, recent studies have proposed many techniques to measure end-to-end latency. For instance, Pingmesh [53] is designed to measure network latency between any two servers. It leverages TCP or HTTP pings to provide the maximum latency coverage. Other work uses the matrix method to return an $n \times n$ network latency matrix among n nodes [54], [55], which can further enhance latency estimation.

However, these techniques are not appropriate for our application, mainly because they are designed to estimate latency rather than quantiles of latency. To detect per-flow tail latency in data streams, they have to carry out a large amount of computation, which is time-consuming and inefficient in reality. Although recent work like SketchFeature [56] extracts high-quality per-flow features in the data plane for security-aware measurement, it targets general feature collection rather than per-flow tail latency estimation, and it does not directly provide lightweight tail latency queries for each flow in data streams.

2) Quantile Estimation:

Traced back to Munro and Paterson who first proposed the idea of quantile estimation [32], quantile estimation has become attractive to many researchers. Since any accurate quantile estimation requires at least $O(N)$ memory for a multiset with size N , many algorithms focus on returning an ε-approximate estimation, which means that it will return a number in $[r - \varepsilon N, r + \varepsilon N]$, where r is the real rank of the item. Manku, Rajagopalan and Lindsay proposed an algorithm with $O(\frac{1}{\varepsilon}\log^2(\varepsilon N))$ space complexity [33], and GK improved this complexity to $O(\frac{1}{\varepsilon}\log(\varepsilon N))$ [34]. However, all these algorithms have to know N in advance, and later, Felber and Ostrovsky improved this bound to $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ [35].

Other algorithms for quantile estimation use randomization, which probably saves memory but has a small probability of falling outside the interval. The KLL algorithm [36] used this technique to obtain $O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta})$ memory complexity with failure rate at most δ for the first time, and further, KLL$^{\pm}$ [37] updated this bound to $O(\frac{1}{\varepsilon}\log^2\log\frac{1}{\varepsilon\delta})$ and supported delete operations.

DDSketch [46] was recently designed for estimating w-quantile distribution in data streams. It divides all positive numbers into several intervals by logarithm and uses buckets to record the frequency in each interval. Moreover, when too many intervals have frequency not equal to 0, DDSketch will merge adjacent buckets to save memory. SQUAD [57] is another algorithm which focuses on estimating per-flow quantiles for heavy-hitters. It applies Reservoir Sampling [58] to sample items from the data stream, and constructs sketches using Space Saving [59] to keep track of these items.

However, these algorithms are generally designed for estimating *full* quantiles, while *tail* quantiles are more important in many situations. Although they can be used to estimate per-flow tail latency, these algorithms keep the information of small latency values as well, which introduces extra overhead when the monitoring objective focuses on tail latency.

3) Frequency Estimation:

CMSketch [8] is the simplest sketch for frequency estimation. It is composed of d arrays and each array has the same number of counters. For every incoming item e, CMSketch uses a hash function to map it into a counter in every array, and increments the counter by 1. When querying an item, CMSketch similarly locates these d counters and returns the minimum of these values. CUSketch [9] shares the same data structure as CMSketch, except that it only increments the counter(s) with minimum value among the d counters.

SALSA [60] and TowerSketch [40] are two variants of CMSketch. SALSA is also composed of d arrays, but each counter in the array is small-sized. When the counter is about to overflow, it will merge adjacent counters to get a larger counter. In this way, SALSA maintains small-sized counters for most small flows, so SALSA significantly reduces memory consumption compared to CMSketch. TowerSketch [40] is based on the idea of different-sized counters for different arrays. Counters in the ith array use 2^i-bit counters. Since TowerSketch still allocates the same memory for different arrays, it has more small counters for small flows, which can record these small flows more accurately.

CSketch [61] is another sketch for frequency estimation. Its data structure is similar to CMSketch and CUSketch, with d arrays and m counters in each array. However, each array has two independent hash functions: f_i and g_i. The range of f_i is $\{0, 1, \cdots, m-1\}$, and the range of g_i is $\{1, -1\}$. To insert an incoming item e into each array, CSketch uses f_i to map it into a counter, and uses g_i to decide whether to increment (if $g_i(e) = 1$) or decrement (if $g_i(e) = -1$) the counter. To query its frequency, CSketch again locates one counter in each array. The value will be multiplied by $g_i(e)$ in each array, and CSketch will return the median among all results in d arrays.

More elegant sketches have also been designed ever since. Cold Filter [39] is a two-layer data structure with a CUSketch in the low layer and a CMSketch in the high layer. All items are inserted into the low layer, and if its frequency in the low layer exceeds the predefined threshold, it is inserted into the high layer instead. In this way, Cold Filter automatically separates small flows and large flows apart.

4) Membership Query:

Bloom Filter [51] is a compact data structure with high memory efficiency. It consists of arrays of bit groups, and is often used to judge whether an item belongs to a given set.

For every incoming item, Bloom Filter uses hash functions to map it into several bits and sets these bits to 1. When querying an item, Bloom Filter uses the same hash functions to check whether all these bits are 1.

III. SketchPolymer Algorithm

In this section, we first propose the baseline solution for per-flow tail latency estimation. Then we introduce the idea of LSS and propose the initial version of SketchPolymer. We optimize SketchPolymer from two aspects and then present the final version of SketchPolymer. Finally, we show that SketchPolymer is a framework by proposing three extensions of SketchPolymer.

A. Baseline Solution

Our baseline solution consists of p buckets $\mathcal{B}_1, \cdots, \mathcal{B}_p$, and each bucket has two fields: frequency field and latency field. The frequency field is just a counter recording the total number of items which have been mapped to this bucket. The latency field consists of q counters to record q maximum latency values in this bucket.

To insert an item e with latency t, we first use a hash function $h(.)$ to map e into bucket $\mathcal{B}_{h(e)}$. Then we update the frequency counter by incrementing it by 1, and try to insert t into the latency field. There are two cases:

Case 1: $\mathcal{B}_{h(e)}$ still has at least one empty counter. In this case, we just record t in one counter and return.

Case 2: $\mathcal{B}_{h(e)}$ does not have any empty counters. Since we focus on estimating tail latency, the baseline solution should mainly keep large latency rather than small latency. Consequently, we find the smallest latency among q counters in bucket $\mathcal{B}_{h(e)}$ (suppose it is $\tilde{t}$) and compare $\tilde{t}$ with t. If $t > \tilde{t}$, we evict $\tilde{t}$ and insert t into this counter; otherwise we do nothing and return.

The query operation is simple: to query the w-quantile latency of item e, we similarly map e into bucket $\mathcal{B}_{h(e)}$. We calculate $m = (1 - w) \times (\mathcal{B}_{h(e)}.frequency - 1) + 1$, which is the rank of w-quantile latency from large to small. Then if $m \leq q$, we return the mth-largest latency in bucket $\mathcal{B}_{h(e)}$; otherwise we return the smallest latency recorded in bucket $\mathcal{B}_{h(e)}$.

Although the baseline solution can be used to estimate per-flow tail latency, it is inaccurate and memory-consuming, as it fails to filter small flows; also, it keeps true latency in the data structure, which is a waste of space in practice.

B. Idea of Latency Splitting and Sharing

The latency of every large flow has to be recorded to accurately estimate per-flow tail latency. However, the naive idea of simply keeping its real latency may not work, as the order of magnitude of latency can vary from item to item. To avoid this problem, we propose the **Latency Splitting and Sharing (LSS)** technique (See Figure 1): We use logarithm to split the latency of all items into several intervals. We first choose a positive number a as the base of the logarithm. To reduce error, a is usually greater than but very close to 1.

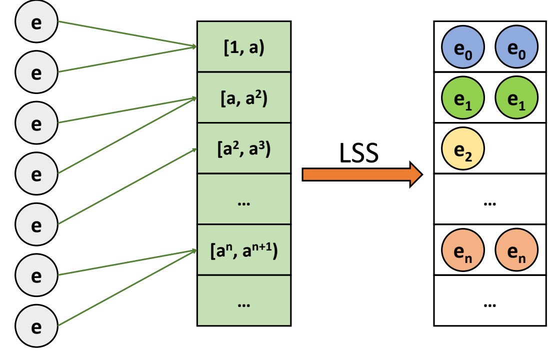


Fig. 1. Idea of LSS

Then, for every item e with its true latency t, we calculate $T = \lfloor \log_a t \rfloor$ as its logarithm latency. In this way, every large flow is separated into several sub-flows, and items in the same sub-flow share the same logarithm latency. The item $e' = (e, T)$ can be viewed as a new item to be inserted into Splitting Stage and Verification Filter. After the query process, suppose SketchPolymer returns an integer T as the w-quantile latency of e, we use exponentiation to get the result a^T. In this way, SketchPolymer maintains the information of these flows to the greatest extent without using too much memory.

C. The SketchPolymer Algorithm

Overview: In this paper, we propose a novel sketch, namely SketchPolymer, to accurately estimate per-flow tail latency. The initial version of SketchPolymer consists of three stages. To avoid recording unnecessary information from small flows, Stage 1 (Filter Stage) uses Early Filtration to separate large flows and send these flows to the following stages. Stage 2 (Polymer Stage) and Stage 3 (Splitting Stage) are designed for estimating tail latency.

1) Idea of Early Filtration:

A naive idea for per-flow tail latency estimation is to record the latency of all flows whether they are large flows or small flows. However, the tail latency of small flows cannot be accurately estimated due to their low frequency. Also, the distribution of real datasets is generally skewed [48]–[50], which means the majority of flows in the data stream are small flows. To make full use of this prior distribution, we apply the **Early Filtration** technique, which originates from the Cold Filter [39]: Filter Stage keeps the frequency of every flow, and only flows with frequency exceeding the threshold are allowed to enter the following stages.

Filter Stage Data Structure: Filter Stage is a CMSketch consisting of $d^{(1)}$ arrays: $\mathcal{C}_1^{(1)}, \cdots, \mathcal{C}_{d^{(1)}}^{(1)}$. Each array consists of $n^{(1)}$ counters. There are $d^{(1)}$ hash functions $h_1^{(1)}, \cdots, h_{d^{(1)}}^{(1)}$ associated with $d^{(1)}$ arrays respectively. Each counter records the frequency of e.

Filter Stage Operation (Algorithm 1-2): To insert an item e, Filter Stage uses $d^{(1)}$ hash functions to map e into $d^{(1)}$ counters $\mathcal{C}_1^{(1)}[h_1^{(1)}(e)]$, $\cdots$, $\mathcal{C}_{d^{(1)}}^{(1)}[h_{d^{(1)}}^{(1)}(e)]$ and increments these counters by 1. When querying e, Filter Stage will use the same hash functions to check $\mathcal{C}_1^{(1)}[h_1^{(1)}(e)]$, $\cdots$, $\mathcal{C}_{d^{(1)}}^{(1)}[h_{d^{(1)}}^{(1)}(e)]$. Similar to CMSketch, Filter Stage will return the minimum of these values as the frequency of e.

Algorithm 1: Filter Stage Insertion Procedure

Input: an item e;

1 **for** $1 \leq i \leq d^{(1)}$ **do**

2 $\quad \mathcal{C}_i^{(1)}[h_i^{(1)}(e)] \leftarrow \mathcal{C}_i^{(1)}[h_i^{(1)}(e)] + 1;$

Algorithm 2: Filter Stage Query Procedure

Input: an item e;

1 $f \leftarrow +\infty$

2 **for** $1 \leq i \leq d^{(1)}$ **do**

3 $\quad f \leftarrow \min\{f, \mathcal{C}_i^{(1)}[h_i(e)]\};$

4 **return** f;

It is worth noticing that filtering small flows does not mean that their latency is unimportant. Instead, high-percentile latency estimation for small flows is statistically unstable with only a few samples available, and the estimated tail latency can be dominated by a single delayed item. Therefore, SketchPolymer prioritizes large flows, while small flows can still be monitored in a best-effort manner (discussed later in Section III-G).

2) Tail Latency Estimation:

Rationale: Polymer Stage and Splitting Stage are designed for estimating the tail latency of large flows. Polymer Stage records the frequency and maximum latency of every large flow, and Splitting Stage records the frequency of every subflow after LSS.

Algorithm 3: Polymer Stage Insertion Procedure

Input: an item (e, T);

1 **for** $1 \leq i \leq d^{(2)}$ **do**

2 $\quad \mathcal{C}_i^{(2)}[h_i^{(2)}(e)].f \leftarrow \mathcal{C}_i^{(2)}[h_i^{(2)}(e)].f + 1;$

3 $\quad \mathcal{C}_i^{(2)}[h_i^{(2)}(e)].t \leftarrow \max\{\mathcal{C}_i^{(2)}[h_i^{(2)}(e)].t, T\};$

Algorithm 4: Polymer Stage Query Procedure

Input: an item e;

1 $f \leftarrow +\infty$

2 $t \leftarrow +\infty$

3 **for** $1 \leq i \leq d^{(2)}$ **do**

4 $\quad f \leftarrow \min\{f, \mathcal{C}_i^{(2)}[h_i^{(2)}(e)].f\};$

5 $\quad t \leftarrow \min\{t, \mathcal{C}_i^{(2)}[h_i^{(2)}(e)].t\};$

6 **return** f, t;

Polymer Stage Data Structure: Polymer Stage consists of $d^{(2)}$ arrays: $\mathcal{C}_1^{(2)}, \cdots, \mathcal{C}_{d^{(2)}}^{(2)}$. Each array consists of $n^{(2)}$ buckets and $d^{(2)}$ hash functions are associated with these arrays. The difference from traditional CMSketch is that each bucket has two fields: frequency field and latency field. The frequency field is similar to Filter Stage, and the latency field records the **maximum** logarithm latency.

Polymer Stage Operation (Algorithm 3-4): To insert an item e with logarithm latency $T = \lfloor \log_a t \rfloor$, Polymer Stage uses $d^{(2)}$ hash functions to map e into $d^{(2)}$ buckets $\mathcal{C}_1^{(2)}[h_1^{(2)}(e)]$, $\cdots$, $\mathcal{C}_{d^{(2)}}^{(2)}[h_{d^{(2)}}^{(2)}(e)]$. The frequency field of these buckets will be incremented by 1, and the latency field of these buckets will be set to the maximum of their initial values and T. When querying e, Polymer Stage will similarly use hash functions to find these $d^{(2)}$ buckets. The frequency and maximum logarithm latency will both be set to the minimum of corresponding fields.

Splitting Stage Data Structure: Splitting Stage is a CMSketch consisting of $d^{(3)}$ arrays: $\mathcal{C}_1^{(3)}, \cdots, \mathcal{C}_{d^{(3)}}^{(3)}$. Each array consists of $n^{(3)}$ counters and $d^{(3)}$ hash functions are associated with the arrays. However, in Splitting Stage, every hash function takes both the item e and the logarithm latency T as arguments. Each counter records the frequency of every item with logarithm latency equal to T.

Splitting Stage Operation (Algorithm 5-6): To insert an item e with logarithm latency T, Splitting Stage maps e into $d^{(3)}$ counters $\mathcal{C}_1^{(3)}[h_1^{(3)}(e,T)]$, $\cdots$, $\mathcal{C}_{d^{(3)}}^{(3)}[h_{d^{(3)}}^{(3)}(e,T)]$ and increments these counters by 1. To query e with logarithm latency T, Splitting Stage will again check these $d^{(3)}$ counters and return the minimum of these values.

Algorithm 5: Splitting Stage Insertion Procedure

Input: an item (e, T);

1 **for** $1 \leq i \leq d^{(3)}$ **do**

2 $\quad \mathcal{C}_i^{(3)}[h_i^{(3)}(e,T)] \leftarrow \mathcal{C}_i^{(3)}[h_i^{(3)}(e,T)] + 1;$

Algorithm 6: Splitting Stage Query Procedure

Input: an item (e, T);

1 $f \leftarrow +\infty$;

2 **for** $1 \leq i \leq d^{(3)}$ **do**

3 $\quad f \leftarrow \min\{\mathcal{C}_i^{(3)}[h_i^{(3)}(e,T)], f\};$

4 **return** f;

D. Memory Optimization: Counter Truncation

After LSS, large flows are usually split into many small subflows according to their logarithm latency. Hence, it will be memory-inefficient to still use 32 bits for a counter in Splitting Stage. Also, we are mainly interested in the tail distribution, and the frequency of these items is generally small. To tackle this problem, we propose **Counter Truncation**: Instead of using 32-bit counters, we only allocate 8-bit counters for Splitting Stage. When we are about to increment a counter, we first check whether the counter has achieved its maximum value (255 here). If so, we keep its value and do nothing. Our experiments show that instead of using 32-bit counters, using more smaller counters improves the accuracy of SketchPolymer (See Section V).

E. Accuracy Optimization: Overestimation Avoidance

The three stages above can fulfill the requirement of estimating per-flow tail latency. However, Splitting Stage is based on CMSketch, which suffers from overestimation error in practice. Inspired by the Bloom Filter, we propose Verification Filter in Stage 4 to compensate for Splitting Stage. When querying an item e with logarithm latency T, SketchPolymer will first check Verification Filter. If Verification Filter reports false for item (e, T), it will be considered to not have appeared before, so SketchPolymer will not query Splitting Stage and just return 0 as the frequency of item (e, T).

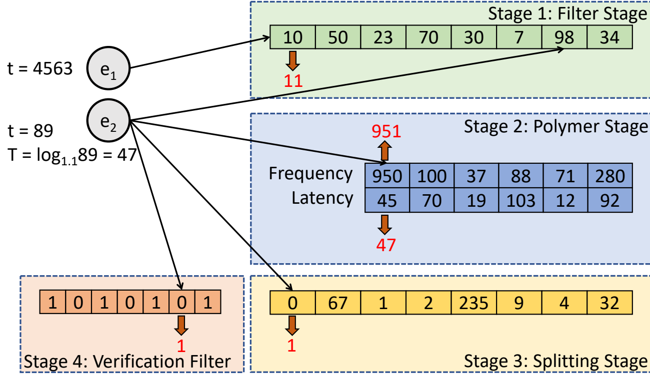


Fig. 2. Example of Insertion

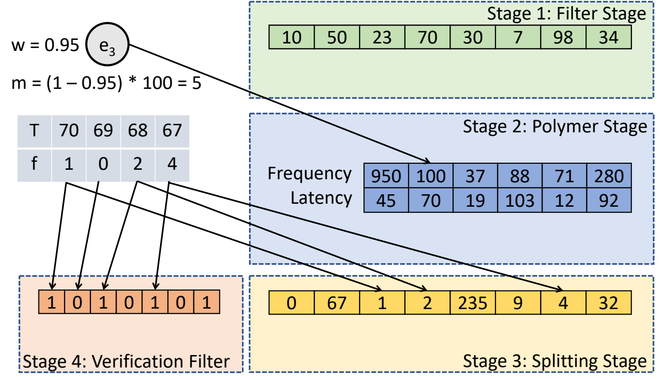


Fig. 3. Example of Query

Verification Filter Data Structure: Verification Filter is a Bloom Filter consisting of $d^{(4)}$ arrays $\mathcal{C}_1^{(4)}, \cdots, \mathcal{C}_{d^{(4)}}^{(4)}$. Each array has $n^{(4)}$ bits. Similar to Splitting Stage, there are $d^{(4)}$ hash functions and each hash function takes the item e and its logarithm latency T as arguments.

Verification Filter Operation (Algorithm 7-8): To insert an item e with logarithm latency T, Verification Filter maps e into $d^{(4)}$ bits $\mathcal{C}_1^{(4)}[h_1^{(4)}(e,T)], \cdots, \mathcal{C}_{d^{(4)}}^{(4)}[h_{d^{(4)}}^{(4)}(e,T)]$ and sets all these bits to 1. When querying (e,T), Verification Filter will return the AND of these $d^{(4)}$ bits.

Algorithm 7: Verification Filter Insertion Procedure

Input: an item (e, T);
1 **for** $1 \le i \le d^{(4)}$ **do**
2 $\quad \mathcal{C}_i^{(4)}[h_i^{(4)}(e,T)] \leftarrow 1$;

Algorithm 8: Verification Filter Query Procedure

Input: an item (e, T);
1 **for** $1 \le i \le d^{(4)}$ **do**
2 $\quad$ **if** $\mathcal{C}_i^{(4)}[h_i^{(4)}(e,T)] = 0$ **then**
3 $\quad\quad$ **return false**;
4 **return true**;

F. Our Final Version

Our final version of SketchPolymer consists of four stages as above. The first three stages are all based on CMSketch, and the Verification Filter is added in Stage 4 for overestimation avoidance. The full operation of SketchPolymer can be summarized as follows:

Insertion (Algorithm 9): Given an item e with latency t, we first query Filter Stage to get its frequency. If its frequency reported by Filter Stage does not exceed the predefined threshold $\mathcal{T}$, we simply insert e into Filter Stage and finish the insertion procedure. Otherwise, we regard e as a large flow, calculate its logarithm latency $T = \lfloor \log_a t \rfloor$ and insert (e, T) into Polymer Stage, Splitting Stage and Verification Filter respectively.

Query (Algorithm 10): Given an item e and a quantile w, we first query Polymer Stage to get its frequency f and maximum logarithm latency T. Then we calculate $m = (1-w) \times f$, which is the number of items of e with latency greater than the w-quantile. Then, we recursively query Splitting Stage and Verification Filter with item e while decreasing the logarithm latency from T until the sum of query results exceeds m. In this way, SketchPolymer finds the w-quantile latency of e.

Algorithm 9: SketchPolymer Insertion Procedure

Input: an item (e, t)
1 **if** `FilterStage.query(e)` $< \mathcal{T}$ **then**
2 $\quad$ `FilterStage.insert(e)`;
3 $\quad$ **return**;
4 $T \leftarrow \lfloor \log_a t \rfloor$;
5 `PolymerStage.insert(e,T)`;
6 `SplittingStage.insert(e,T)`;
7 `VerificationFilter.insert(e,T)`;

Algorithm 10: SketchPolymer Query Procedure

Input: an item e, a quantile w
1 $f, T \leftarrow$ `PolymerStage.query(e)`;
2 $m \leftarrow (1-w)f$;
3 **while** $m > 0$ **do**
4 $\quad$ **if** `VerificationFilter.query(e,T)` **then**
5 $\quad\quad$ $m \leftarrow m -$ `SplittingStage.query(e,T)`;
6 $\quad$ $T \leftarrow T - 1$;
7 **return** a^{T+1};

A running example: For simplicity, we choose $d^{(1)} = d^{(2)} = d^{(3)} = d^{(4)} = 1$, $\mathcal{T} = 50$ and $a = 1.1$. Figure 2 shows a running example of the insertion procedure of SketchPolymer. When inserting e_1 with latency 4563, we first use a hash function to map e_1 into a counter in Filter Stage. Since Filter Stage returns 10 as its frequency, which is smaller than the threshold, SketchPolymer just increments this counter by 1 and finishes the inserting procedure. When inserting e_2 with latency 89, we again query Filter Stage and Filter Stage returns 98. Since 98 is greater than the threshold, e_2 is allowed to enter Polymer Stage, Splitting Stage and Verification Filter. We get its logarithm latency by $T = \lfloor \log_{1.1} 89 \rfloor = 47$. The frequency field of Polymer Stage associated with e_2 is incremented by 1 (from 950 to 951), and the latency field is set to the maximum of its initial value 45 and T. Finally, we map $(e_2, 47)$ into a counter in Splitting Stage and increment it by 1 and map it into a bit in Verification Filter and set it to 1. Figure 3 shows a running example of query procedure of SketchPolymer. To query the 0.95 tail latency

of e_3, SketchPolymer first queries Polymer Stage to get its frequency 100 and maximum logarithm latency 70. Next we calculate $m = (1 - 0.95) \times 100 = 5$, which shows that 5 of e_3 have latency greater than the 0.95-quantile. Then we use $(e_3, 70)$ to query Splitting Stage and Verification Filter. Since Verification Filter returns true for $(e_3, 70)$, we reduce the query result of Splitting Stage (which is 1 here) from m, and m is now 4. Then we query Verification Filter with $(e_3, 69)$. Since Verification Filter returns false for these arguments, we do nothing and start the query with $(e_3, 68)$. Verification Filter returns true and Splitting Stage returns 2 for 68, so m is reduced by 2 and is now 2. Finally we query Splitting Stage and Verification Filter with $(e_3, 67)$. As Verification Filter returns true and Splitting Stage returns 4, m will be reduced by 4 and will turn negative after this operation. Consequently, we stop recursively querying and return $1.1^{67} \approx 593.35$ as the 0.95 tail latency of e_3.

G. SketchPolymer is a Framework

We have proposed SketchPolymer for per-flow tail latency estimation in data stream models. However, this design still has room for improvement. Firstly, we can use more advanced sketches to replace CMSketch in Polymer Stage and Splitting Stage. Secondly, SketchPolymer only estimates tail latency for large flows, but tail latency estimation for small flows can also be needed for applications. Finally, to make SketchPolymer more general, we extend SketchPolymer to arbitrary latency quantile estimation.

1) Using Different Sketches in SketchPolymer:

SketchPolymer uses LSS to estimate per-flow tail latency more efficiently. For every item e, Polymer Stage records its frequency and maximum latency, and Splitting Stage records its frequency after LSS. To make the SketchPolymer design more generic, we can use other sketches to replace the CMSketch in Polymer Stage and Splitting Stage. Below we show how to extend SketchPolymer to these sketches.

Data Structure: Polymer Stage and Splitting Stage are the same as mentioned before, except that these two stages now follow the rationale of the extended sketch instead of CMSketch. Polymer Stage is a $d^{(2)}$-array data structure, and each array consists of several buckets. Each bucket has two fields: frequency field and latency field. Splitting Stage consists of $d^{(3)}$ arrays, and each array is consists of several counters.

Insertion and Query Operation: To insert an item e with logarithm latency T into Polymer Stage, we use $d^{(2)}$ hash functions to locate $d^{(2)}$ buckets. However, to update the frequency field of these buckets, we follow the insertion algorithm of the extended sketch, rather than simply incrementing each counter by 1 in the CMSketch version. The latency field of these buckets will again be set to the maximum of its initial value and T, just as above. When querying e, its frequency will be calculated according to the query algorithm of the extended sketch, and the maximum latency will be set to the minimum of $d^{(2)}$ latency fields. The insertion and query operations of Splitting Stage are similar to those of the extended sketch: we use hash functions to locate one counter in each array, and follow its insertion and query algorithm to insert and query (e, T).

Examples: Now we show how to extend SketchPolymer from CMSketch to some popular sketches, *i.e.* CUSketch, CSketch, SALSA and TowerSketch. In SketchPolymer-CU, we only increment the counter(s) with the minimum value in Polymer Stage and Splitting Stage; In SketchPolymer-C, we use additional $d^{(2)}$ and $d^{(3)}$ hash functions to decide whether to increment or decrement the counter in the insertion process, and we return the median rather than the minimum value among these counters as the estimated frequency; In SketchPolymer-SALSA, we initialize Polymer Stage and Splitting Stage with small counters (*e.g.* 4-bit counters). If the counter is about to overflow, we merge it with adjacent counters to get a larger counter; In SketchPolymer-Tower, different arrays in Polymer Stage and Splitting Stage use different-sized counters. A counter will be marked as overflowed if it reaches its maximum value. We will not increment overflowed counters, and the query operation will return the minimum value among non-overflowed counters.

Guidelines for Choosing Underlying Sketches: The choice of the underlying sketch depends on the monitoring goal and resource budget. SketchPolymer-CM is a simple default choice with high throughput and easy hardware implementation. SketchPolymer-CU is preferable for skewed workloads, where reducing overestimation caused by frequent flows is important. SketchPolymer-C can be used when one wants to reduce the one-sided overestimation bias of CMSketch, but it introduces additional hash computations. SketchPolymer-SALSA and SketchPolymer-Tower are more suitable for memory-constrained scenarios, since they use compact or heterogeneous counters to improve memory efficiency.

2) Handling Small Flows:

SketchPolymer achieves accurate tail latency estimation for large flows but sacrifices the accuracy of small flows. However, tail latency estimation for small flows still has a wide range of applications in network scenarios, *e.g.* malicious small-flow detection [62], [63], cache optimization for small flows [64] and flow priority management [65]. To make SketchPolymer more elastic, we make a small modification to Filter Stage to cater for small flows.

Filter Stage Data Structure: To realize tail latency estimation for small flows, the modified Filter Stage uses the same data structure as Polymer Stage. Filter Stage still consists of $d^{(1)}$ arrays, and each array still consists of $n^{(1)}$ buckets. Each bucket consists of two fields: frequency field and latency field, which is similar to Polymer Stage.

Filter Stage Insertion and Query Operation: To insert an item e with logarithm latency T, we still use $d^{(1)}$ hash functions to map e into one bucket in each array. The frequency field of all buckets will be incremented by 1, and the latency field will be set to the maximum of its initial value and T. To query the tail latency of e, we just use the maximum latency to approximate tail latency: we again locate $d^{(1)}$ buckets, find the minimum logarithm latency, and use an exponential operation to calculate the maximum latency as its tail latency.

It should be noted that this extension is intended for best-effort monitoring rather than precise tail latency estimation. Since small flows contain only a few samples, their tail latency is inherently unstable and can be dominated by a

single delayed item. Therefore, this extension is mainly useful for coarse-grained triage, such as identifying suspicious small flows and triggering additional sampling or fine-grained monitoring, rather than strict SLA enforcement or precise root-cause analysis.

3) Arbitrary Latency Quantile Estimation:

SketchPolymer is mainly designed for tail latency estimation, but users may sometimes be interested in arbitrary latency quantiles. To broaden the applicability of SketchPolymer, we replace 8-bit counters with 16-bit counters in Splitting Stage to support arbitrary latency quantile estimation. In this way, most counters in Splitting Stage will not overflow, so SketchPolymer can estimate arbitrary latency quantiles for large flows accurately.

IV. Mathematical Analysis

In general, SketchPolymer can be extended to many popular sketches. But for convenience, we only conduct theoretical analysis for SketchPolymer-CM. We first provide error bounds for SketchPolymer. We derive the error bound for Polymer Stage and Splitting Stage in the first step, then we give an error bound for SketchPolymer. Finally, we analyze the time complexity of SketchPolymer.

A. Error Bound

In this section, we assume that e_j is a large flow, *i.e.*, e_j successfully enters Polymer Stage, Splitting Stage and Verification Filter in SketchPolymer. We obtain the error bounds for SketchPolymer in the following theorems:

Theorem 1. *Let f_j, T_j be the real frequency and the real maximum logarithm latency of e_j, and $\hat{f}_j, \hat{T}_j$ be the estimated results reported by Polymer Stage. Let N denote the number of items in the data stream, and suppose M flows have maximum latency greater than that of e_j. Then, given a small positive number ε, the estimation error of f_j and T_j is bounded by*

$$\mathbb{P}(\hat{f}_j \geq f_j - \mathcal{T} + \varepsilon) \leq \left(\frac{N}{\varepsilon n^{(2)}}\right)^{d^{(2)}}, \tag{1}$$

and

$$\mathbb{P}(\hat{T}_j \neq T_j) \leq \frac{\mathcal{T}}{f_j} + \frac{d^{(2)}M}{n^{(2)}}. \tag{2}$$

Proof. We define an indicator variable $I_{j,k,l}$ as

$$I_{j,k,l} = \begin{cases} 1, & \text{if } l \neq j \wedge h_k^{(2)}(l) = h_k^{(2)}(j); \\ 0, & \text{others.} \end{cases}$$

Due to the independence of hash functions, we get

$$\mathbb{E}I_{j,k,l} = \mathbb{P}(h_k^{(2)}(j) = h_k^{(2)}(l)) = \frac{1}{n^{(2)}}.$$

Let us define another variable

$$X_{j,k} = \sum_l f_l I_{j,k,l}$$

indicating the overestimation error caused by hash collisions. By the linearity of expectation,

$$\mathbb{E}X_{j,k} = \sum_l f_l \mathbb{E}I_{j,k,l} = \frac{N}{n^{(2)}}.$$

Notice that $\hat{f}_j = f_j - \mathcal{T} + \min\{X_{j,k} : 1 \leq k \leq d^{(2)}\}$. According to the Markov Inequality, we get

$$\mathbb{P}(\hat{f}_j \geq f_j - \mathcal{T} + \varepsilon) = \mathbb{P}(\forall 1 \leq k \leq d^{(2)}, X_{j,k} \geq \varepsilon)$$
$$= [\mathbb{P}(X_{j,k} \geq \varepsilon)]^{d^{(2)}} \leq \left[\frac{\mathbb{E}(X_{j,k})}{\varepsilon}\right]^{d^{(2)}} = \left(\frac{N}{\varepsilon n^{(2)}}\right)^{d^{(2)}}.$$

Hence Equation 1 holds.

As to the second part of the theorem, note that if the maximum latency of e_j appears after e_j enters Polymer Stage, and no items with larger latency collide with e_j, then the recorded maximum logarithm latency must be accurate. Let $e_{p_1}, \cdots, e_{p_M}$ denote all flows with maximum latency larger than that e_j, then

$$\mathbb{P}(\hat{T}_j = T_j) \geq \frac{f_j - \mathcal{T}}{f_j} \times \mathbb{P}(\exists i, \mathcal{C}_i^{(2)}[h_i^{(2)}(e_j)].T = T_j)$$
$$= \frac{f_j - \mathcal{T}}{f_j} \times \left[1 - \mathbb{P}(\forall i, \mathcal{C}_i^{(2)}[h_i^{(2)}(e_j)].T > T_j)\right]$$
$$\geq \frac{f_j - \mathcal{T}}{f_j}\left[1 - d^{(2)}\mathbb{P}(\mathcal{C}_i^{(2)}[h_i^{(2)}(e_j)].T > T_j)\right],$$

where

$$\mathbb{P}(\mathcal{C}_i^{(2)}[h_i^{(2)}(e_j)].T > T_j) = 1 - \left(1 - \frac{1}{n^{(2)}}\right)^M \leq \frac{M}{n^{(2)}}.$$

Hence,

$$\mathbb{P}(\hat{T}_j = T_j) \geq \left(1 - \frac{\mathcal{T}}{f_j}\right)\left(1 - \frac{d^{(2)}M}{n^{(2)}}\right) \geq 1 - \frac{\mathcal{T}}{f_j} - \frac{d^{(2)}M}{n^{(2)}},$$

so Equation 2 also holds. □

Theorem 2. *For an integer T, let $f_{j,T}$ be the real number of e_j with logarithm latency equal to T, and $\hat{f}_{j,T}$ be the result reported by Splitting Stage. f_j and N are defined as above. Given a small positive number ε, if $\hat{f}_{j,T} < 255$ (to ensure that the 8-bit counter does not overflow), then the estimation error of $f_{j,T}$ is bounded by*

$$\mathbb{P}(|\hat{f}_{j,T} - f_{j,T}| \geq \varepsilon) \leq e^{-\frac{f_j}{2\mathcal{T}f_{j,T}}\left(\varepsilon - \frac{\mathcal{T}f_{j,T}}{f_j}\right)^2} + \left(\frac{N}{\varepsilon n^{(3)}}\right)^{d^{(3)}}. \tag{3}$$

Proof. We prove this theorem in two parts. The first part is similar to Equation 1: Define

$$I_{j,T,k,l,S} = \begin{cases} 1, & \text{if } (j,T) \neq (l,S) \wedge h_k^{(3)}(j,T) = h_k^{(3)}(l,S); \\ 0, & \text{others.} \end{cases}$$

and

$$X_{j,T,k} = \sum_{(l,S)} f_{l,S} I_{j,T,k,l,S},$$

then $\mathbb{E}X_{j,T,k} = \frac{N}{n^{(3)}}$.

$$\mathbb{P}(\hat{f}_{j,T} - f_{j,T} \geq \varepsilon) = \mathbb{P}(\forall 1 \leq k \leq d^{(3)}, X_{j,T,k} \geq \varepsilon)$$
$$= [\mathbb{P}(X_{j,T,k} \geq \varepsilon)]^{d^{(3)}} \leq \left[\frac{\mathbb{E}X_{j,T,k}}{\varepsilon}\right]^{d^{(3)}} = \left(\frac{N}{\varepsilon n^{(3)}}\right)^{d^{(3)}}.$$

To prove the second part of the theorem, it is worth noticing that the only explanation for $\hat{f}_{j,T} < f_{j,T}$ is that items with logarithm latency T come before e_j enters Splitting Stage,

so the information *w.r.t.* (e_j, T) is lost. We define a variable X as the number of items with logarithm latency T which arrive before e_j enters Splitting Stage, and suppose f_j is large enough, then X can be approximated by a binomial distribution: $X \sim B(\mathcal{T}, \frac{f_{j,T}}{f_j})$. So $\mu = \mathbb{E}X = \frac{\mathcal{T} f_{j,T}}{f_j}$. Let $\delta = \frac{f_j}{\mathcal{T} f_{j,T}}\left(\varepsilon - \frac{\mathcal{T} f_{j,T}}{f_j}\right)$. Applying Chernoff Bound, we get

$$\begin{aligned}&\mathbb{P}(\hat{f}_{j,T} - f_{j,T} \le -\varepsilon) \le \mathbb{P}(X \ge \varepsilon)\\&= \mathbb{P}(X \ge (1+\delta)\mu) \le e^{-\frac{\mu\delta^2}{2}} = e^{-\frac{f_j}{2\mathcal{T} f_{j,T}}\left(\varepsilon - \frac{\mathcal{T} f_{j,T}}{f_j}\right)^2}.\end{aligned}$$

Applying both inequalities, we get

$$\begin{aligned}&\mathbb{P}(|\hat{f}_{j,T} - f_{j,T}| \ge \varepsilon)\\&= \mathbb{P}(\hat{f}_{j,T} - f_{j,T} \ge \varepsilon) + \mathbb{P}(\hat{f}_{j,T} - f_{j,T} \le -\varepsilon)\\&\le e^{-\frac{f_j}{2\mathcal{T} f_{j,T}}\left(\varepsilon - \frac{\mathcal{T} f_{j,T}}{f_j}\right)^2} + \left(\frac{N}{\varepsilon n^{(3)}}\right)^{d^{(3)}}.\end{aligned}$$

So Equation 3 holds. □

Theorem 3. *Assume $f_j >> \mathcal{T}$ and $T_j = \hat{T}_j$. Let t_j be the real w-quantile latency of e_j, and $\hat{t}_j$ be the w-quantile latency reported by SketchPolymer. Suppose the real quantile function at $\hat{t}_j$ is $\hat{w}$, then $|\hat{w} - w| < \varepsilon$ with probability at least $1 - O(\varepsilon^{-d})$, where $d = \min\{d^{(2)}, d^{(3)}\}$.*

Proof. Since $f_j >> \mathcal{T}$, we approximate Equation 1 as

$$\mathbb{P}(\hat{f}_j \ge f_j + \varepsilon) \le \left(\frac{N}{\varepsilon n^{(2)}}\right)^{d^{(2)}}. \tag{4}$$

Equation 3 can also be simplified as

$$\mathbb{P}(\hat{f}_{j,T} \ge f_{j,T} + \varepsilon) \le \left(\frac{N}{\varepsilon n^{(3)}}\right)^{d^{(3)}},$$

and we can prove

$$\mathbb{P}\left(\sum_{s=T}^{T+R-1} \hat{f}_{j,T} \ge \sum_{s=T}^{T+R-1} f_{j,T} + \varepsilon\right) \le \left(\frac{NR}{\varepsilon n^{(3)}}\right)^{d^{(3)}}. \tag{5}$$

Let $l = \lfloor \log_a t_j \rfloor$ and $\hat{l} = \lfloor \log_a \hat{t}_j \rfloor$. Both $\hat{f}_j$ and $\hat{f}_{j,T}$ suffer from overestimation errors. The former will lead to underestimation of t_j, and the latter will lead to overestimation of t_j. By the definition of w and $\hat{w}$, we know that

$$\begin{cases}(1-w)f_j \approx f_{j,l} + \cdots + f_{j,T_j},\\(1-w)\hat{f}_j \approx \hat{f}_{j,\hat{l}} + \cdots + \hat{f}_{j,T_j},\\(1-\hat{w})f_j \approx f_{j,\hat{l}} + \cdots + f_{j,T_j}.\end{cases}$$

If $w - \hat{w} \ge \varepsilon$, then

$$(1-w)\hat{f}_j \approx \hat{f}_{j,\hat{l}} + \cdots + \hat{f}_{j,T_j} \ge f_{j,\hat{l}} + \cdots + f_{j,T_j} \approx (1-\hat{w})f_j,$$

which means $w - \hat{w} \le \frac{\hat{f}_j - f_j}{f_j}(1-w)$. Applying Equation 4, we get

$$\mathbb{P}(w - \hat{w} \ge \varepsilon) \le \mathbb{P}\left(\hat{f}_j - f_j \ge \frac{\varepsilon f_j}{1-w}\right) \le \left(\frac{N(1-w)}{\varepsilon f_j n^{(2)}}\right)^{d^{(2)}}.$$

Similarly, if $\hat{w} - w \ge \varepsilon$, then

$$\begin{aligned}&\hat{f}_{j,\hat{l}} + \cdots + \hat{f}_{j,T_j} \approx (1-w)\hat{f}_j \ge (1-w)f_j\\&\approx (\hat{w} - w)f_j + (f_{j,\hat{l}} + \cdots + f_{j,T_j}).\end{aligned}$$

Hence

$$\sum_{s=\hat{l}}^{T_j} \hat{f}_{j,s} - \sum_{s=\hat{l}}^{T_j} f_{j,s} \ge (\hat{w} - w)f_j.$$

Applying Equation 5, we get

$$\begin{aligned}&\mathbb{P}(\hat{w} - w \ge \varepsilon) \le \mathbb{P}\left(\sum_{s=\hat{l}}^{T_j}(\hat{f}_{j,s} - f_{j,s}) \ge \varepsilon f_j\right)\\&\le \left(\frac{N(T_j - \hat{l} + 1)}{\varepsilon f_j n^{(3)}}\right)^{d^{(3)}} \le \left(\frac{N(T_j - l + 1)}{\varepsilon f_j n^{(3)}}\right)^{d^{(3)}}.\end{aligned}$$

Finally, we get

$$\begin{aligned}&\mathbb{P}(|\hat{w} - w| < \varepsilon) = 1 - \mathbb{P}(|\hat{w} - w| \ge \varepsilon)\\&\ge 1 - \left(\frac{N(1-w)}{\varepsilon f_j n^{(2)}}\right)^{d^{(2)}} - \left(\frac{N(T_j - l + 1)}{\varepsilon f_j n^{(3)}}\right)^{d^{(3)}}\\&= 1 - O(\varepsilon^{-d}),\end{aligned}$$

which finishes our proof. □

B. Time Complexity

Theorem 4. *Assume $d^{(1)}, d^{(2)}, d^{(3)}$ and $d^{(4)}$ are all very small. The insertion time complexity of SketchPolymer for an arbitrary item e is $O(1)$.*

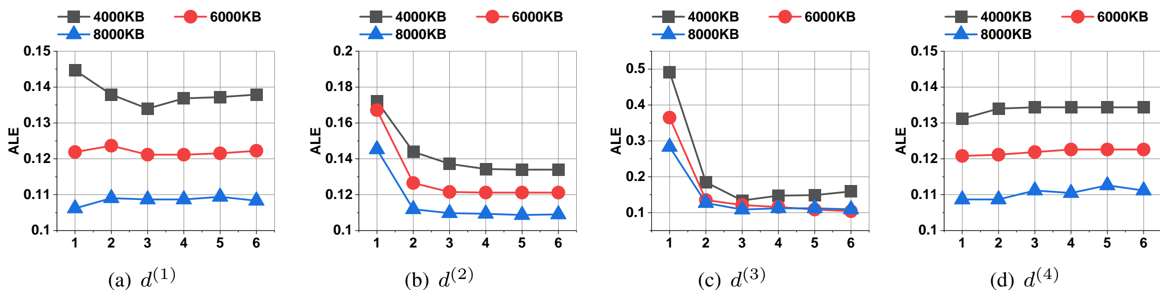
Proof. To insert an arbitrary item e, SketchPolymer first queries Filter Stage to get its frequency. Then, either e is inserted into Filter Stage, or e is inserted into Polymer Stage, Splitting Stage and Verification Filter. All of these processes can be done in $O(1)$ time. □

V. Experimental Results

In this section, we provide experimental results for SketchPolymer. First, we describe the experimental setup in Section V-A. Then, we show how parameter settings affect SketchPolymer's performance in Section V-B. We compare the performance of SketchPolymer and other algorithms on different datasets in Section V-C. We provide an analysis of SketchPolymer in Section V-D. Finally, we demonstrate the broader applicability of SketchPolymer by integrating it into the RocksDB database to accelerate tail quantile estimation in Section V-E.

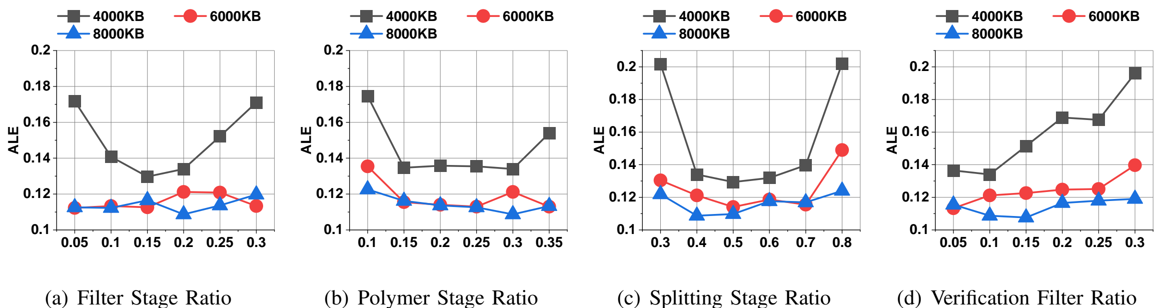
A. Experimental Setup

Implementation: We implement SketchPolymer and all other algorithms in C++. In all experiments, we use Murmur3 Hash [66] with different hash seeds to implement the hash functions.
Computation Platform: We conducted all the experiments on a server with one 18-core processor (36 threads, Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) and 128 GB DRAM memory. The processor has 64KB L1 cache, 1MB L2 cache for each core, and 24.75MB L3 cache shared by all cores.
Metrics:
1) Average Logarithm Error (ALE): Since the orders of magnitude of latency can vary significantly, it is unreasonable to simply measure the error by absolute value. Suppose $t_1, t_2, \cdots, t_n$ are the true tail latency of all flows, and



(a) $d^{(1)}$ (b) $d^{(2)}$ (c) $d^{(3)}$ (d) $d^{(4)}$

Fig. 4. Effects of Numbers of Hash Functions



(a) Filter Stage Ratio (b) Polymer Stage Ratio (c) Splitting Stage Ratio (d) Verification Filter Ratio

Fig. 5. Effects of Memory Allocation Ratio

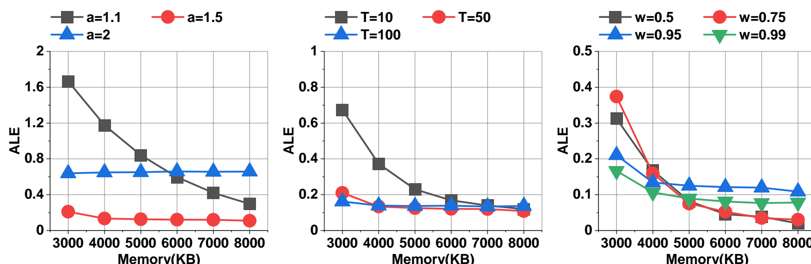


Fig. 6. Effects of a Fig. 7. Effects of $\mathcal{T}$ Fig. 8. Effects of w

$\hat{t}_1, \hat{t}_2, \cdots, \hat{t}_n$ be the estimated tail latency. The average logarithm error (ALE) is defined as $\frac{1}{n}\sum_{i=1}^{n}|\log_2 t_i - \log_2 \hat{t}_i| = \frac{1}{n}\sum_{i=1}^{n}|\log_2 \frac{t_i}{\hat{t}_i}|$.

2) Throughput: We use millions of operations (insertions and queries) per second (Mops) to measure the throughput. We repeat the experiment 10 times and calculate the average results as our throughput.

Datasets:

1) IP Trace: The IP Trace consists of streams of anonymous IP traces collected from 2016 by CAIDA [67]. We regard the interval between two consecutive packets as its latency. We use 20 million items.

2) Seattle Dataset: The Seattle Dataset [68], [69] consists of round trip times (RTTs) between several nodes in the Seattle network. We treat RTTs between the same two nodes as the same flow, and regard RTT as its latency.

3) Web Latency Dataset: The Web Latency Dataset [70] is collected by Webget [17] on 182 probes distributed globally. We regard the fetch time of each request as latency.

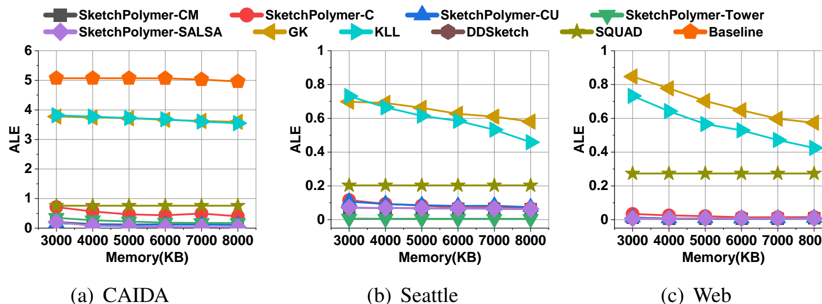
B. Experiments on Parameter Settings

In this section, we measure the effects of some key parameters of SketchPolymer-CM, namely, the number of hash functions in each stage, the memory allocation ratio, the choice of the base of logarithm a, the threshold $\mathcal{T}$, and the query latency quantile w. We use the CAIDA dataset in these experiments, and use ALE to measure these effects.

Effects of Numbers of Hash Functions (Figure 4): *The experimental results show that when allocated sufficient memory, more hash functions usually work better.* In this experiment, we vary each $d^{(k)}$ from 1 to 6. The results show that more hash functions generally lead to smaller ALE, but it can also perform worse with smaller space. Taking into consideration the fact that more hash functions can have a detrimental influence on overall throughput, we set $(d^{(1)}, d^{(2)}, d^{(3)}, d^{(4)}) = (3, 5, 3, 2)$ by default.

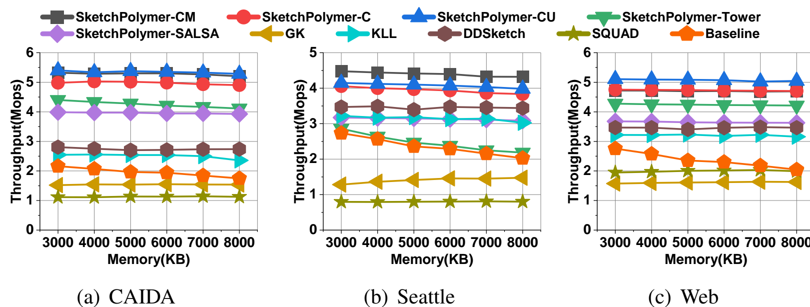
Effects of Memory Allocation Ratio (Figure 5): *The experimental results show that most space should be allocated to Splitting Stage.* In each experiment, we vary the ratio of one stage in a certain range, and keep the relative ratio of the other 3 stages unchanged. The results show that a small ratio of memory for Filter Stage, Polymer Stage and Verification Filter will be enough. Since Splitting Stage records the frequency of all flows after LSS, we allocate 20% memory for Filter Stage, 30% memory for Polymer Stage, 40% memory for Splitting Stage, and 10% memory for Verification Filter in our experiments.

Effects of a (Figure 6): *The experimental results show that*



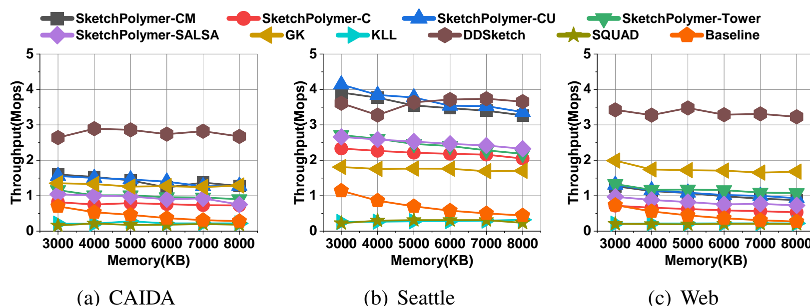
(a) CAIDA (b) Seattle (c) Web

Fig. 9. ALE on Different Datasets



(a) CAIDA (b) Seattle (c) Web

Fig. 10. Insertion Throughput on Different Datasets



(a) CAIDA (b) Seattle (c) Web

Fig. 11. Query Throughput on Different Datasets

the choice of a is generally a trade-off between memory and accuracy. We set a to 1.1, 1.5 and 2 respectively in each experiment, and results show that a larger a performs better within a smaller memory. However, if a user allocates sufficient space to SketchPolymer, the ALE will be minimized when a is close to 1. To strike a balance between accuracy and memory, we set $a = 1.5$ in other experiments.

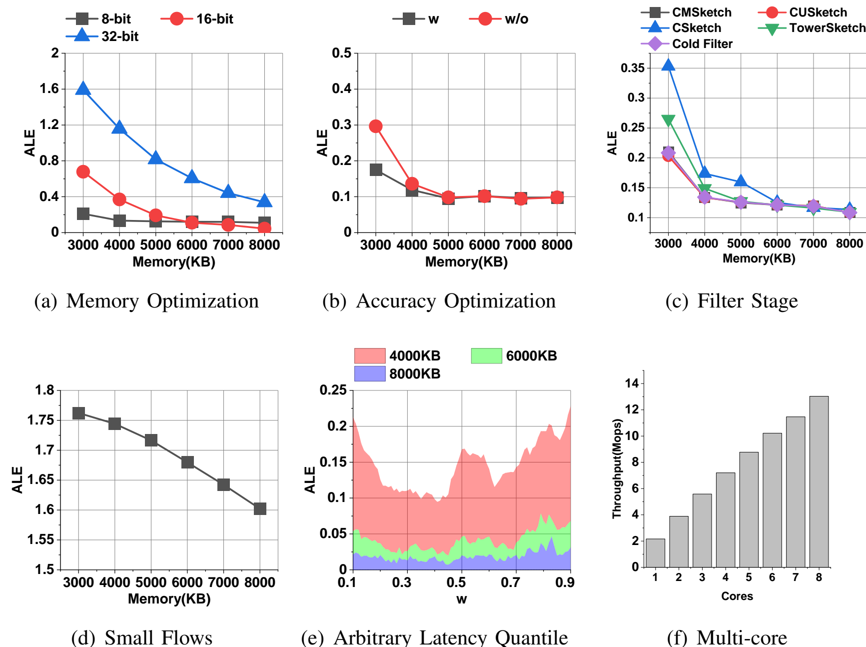
Effects of $\mathcal{T}$ (Figure 7): *The experimental results show that the best option for $\mathcal{T}$ is between 50 and 100.* We try different values of $\mathcal{T}$, and find that a larger $\mathcal{T}$ is better when memory is small, as it can filter as many small flows as possible in Filter Stage. However, a larger $\mathcal{T}$ also risks losing the information of large flows, as the first $\mathcal{T}$ latency values of every flow will not be recorded by SketchPolymer. Taking both cases into account, we choose $\mathcal{T} = 50$.

Effects of w (Figure 8): *The experimental results show that SketchPolymer performs well regardless of which latency quantile we set.* We query the 0.5, 0.75, 0.95 and 0.99 tail latency of large flows respectively, and the results show that ALE is close to 0 when memory is greater than 8000KB. In fact, users can set w to any quantile they are interested in, and we set $w = 0.95$ in other experiments.

Guidelines for Parameter Configuration: The parameter settings of SketchPolymer should be configured according to the workload characteristics and memory budget. In general, more hash functions can reduce hash collisions and improve accuracy, while fewer hash functions are preferable for high-speed scenarios. Most memory should be allocated to Splitting Stage, since it records the frequencies of sub-flows after LSS and directly affects estimation accuracy. The logarithm base a controls the trade-off between memory and accuracy: a smaller a gives finer-grained latency intervals, while a larger a is more suitable when memory is limited. The threshold $\mathcal{T}$ should be chosen according to the flow-size distribution: a larger $\mathcal{T}$ filters more infrequent flows, while a smaller $\mathcal{T}$ allows more medium-sized flows to be monitored. For new workloads, operators can use a short warm-up sample to estimate the flow-size distribution and latency range before setting these parameters.

C. Comparison with Baseline Solution and Prior Work

In this section, we compare different versions of SketchPolymer with prior work (GK, KLL, DDSketch, SQUAD) on different datasets. We measure the ALE and throughput (insertion and query) of different algorithms on three real-world datasets.



(a) Memory Optimization (b) Accuracy Optimization (c) Filter Stage

(d) Small Flows (e) Arbitrary Latency Quantile (f) Multi-core

Fig. 12. Analyses on SketchPolymer

Implementation: Since some of these techniques (GK, KLL, DDSketch) are not designed for per-flow quantile estimation, we allocate several buckets for these algorithms. Before the insertion procedure, we use a hash function to map the item e into a bucket, and insert e into this bucket. When querying an item, we use the same function to locate a bucket and return the quantile of all latency in this bucket.

ALE (Figure 9): *The experimental results show that different versions of SketchPolymer all outperform prior work on three datasets.* On the CAIDA dataset, the ALE of SketchPolymer-CM, SketchPolymer-CU, SketchPolymer-Tower and SketchPolymer-SALSA is smaller than 0.4 within 3000KB memory, and the ALE is close to 0 for all five versions of SketchPolymer within 8000KB memory, which is 91% better than SQUAD and 800% better than GK, KLL and DDSketch. The ALE of SketchPolymer is all smaller than 0.1 on Seattle dataset, and smaller than 0.01 on Web dataset (except SketchPolymer-Tower), while the ALE of other algorithms is all greater than 0.2 on the two datasets. [2]

Insertion Throughput (Figure 10): *The experimental results show that different versions of SketchPolymer all insert items faster than prior work.* The insertion throughput of SketchPolymer is higher than 4 Mops on CAIDA dataset. In contrast, the insertion throughput of GK, KLL, DDSketch and SQUAD is lower than 3Mops, because their insertion operation involves complex calculations. The insertion throughput of SketchPolymer is still higher than that of prior work on the Seattle and Web datasets, and it is 2 times higher than SQUAD, another per-flow quantile estimation algorithm.

Query Throughput (Figure 11): *The experimental results show that SketchPolymer queries items faster than most prior work.* Even if SketchPolymer involves a complex and iterative query process, it can answer per-flow tail latency queries efficiently. The query throughput of SketchPolymer is only lower than DDSketch on CAIDA dataset, and it is higher than 2 Mops on Seattle dataset, which GK, KLL and SQUAD fail to achieve.

[2]On the CAIDA dataset, the ALE of DDSketch exceeds 13, and both the baseline solution and DDSketch have ALE greater than 1 on Seattle and Web datasets. Therefore, we omit their results from Figure 9 to preserve visual clarity.

D. Analysis of SketchPolymer

In this section, we analyze SketchPolymer-CM from several aspects. Firstly, we measure the effects of two optimizations, *i.e.* memory optimization and accuracy optimization. Secondly, we try different data structures for Early Filtration in Filter Stage. Thirdly, we evaluate the effects of tail latency estimation for small flows. We also evaluate how SketchPolymer performs under arbitrary latency quantile queries. Finally, we implement SketchPolymer on a multi-core CPU to accelerate its insertion throughput. We use the CAIDA dataset in these experiments, and use ALE to measure these effects.

Effects of Memory Optimization (Figure 12(a)): *The experimental results show that 8-bit counters for Splitting Stage are enough.* We compare the performance of using 8-bit, 16-bit and 32-bit counters in Splitting Stage. The results show that the performance of using three kinds of counters is close with more than 8000KB memory. However, the ALE of 8-bit counters is smaller than 0.2 within 3000KB memory, which is significantly lower than 16-bit and 32-bit counters.

Effects of Accuracy Optimization (Figure 12(b)): *The experimental results show that Verification Filter can slightly improve accuracy.* We compare the performance of SketchPolymer with and without Verification Filter. The results show that Verification Filter is important for SketchPolymer even if it will take up part of the memory of Splitting Stage. Verification Filter plays a remarkable role in SketchPolymer when memory is between 3000KB and 4000KB. It still slightly lowers the ALE when memory is large.

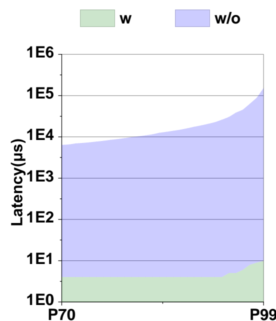


Fig. 13. Tail Latency of Quantile Query

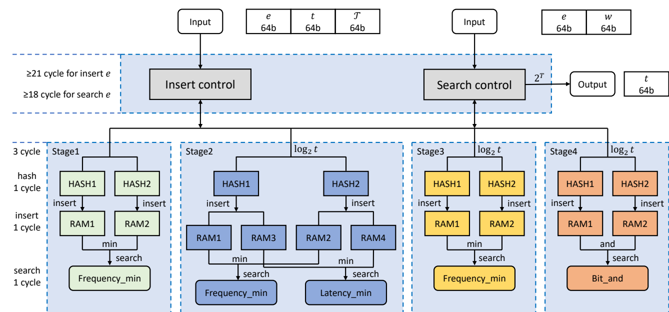


Fig. 14. FPGA Implementation Architecture

Effects of Filter Stage Data Structure (Figure 12(c)): *The experimental results show that any data structure for filtering small flows can greatly lower the error of SketchPolymer.* We replace CMSketch with CUSketch, CSketch, TowerSketch and Cold Filter in Filter Stage, and we find that the ALE of SketchPolymer is always below 0.2 within 4000KB memory, and close to 0.1 within 8000KB memory. However, the performance of using different sketches in Filter Stage does not vary significantly, so we just use CMSketch in Filter Stage.

Effects of Tail Latency Estimation for Small Flows (Figure 12(d)): *The experimental results show that SketchPolymer provides a rough result for small flows.* We add a latency field for each bucket in Filter Stage, and results show that the error of tail latency estimation for small flows is within two orders of magnitude when 8000KB memory is allocated. Since tail latency estimation is difficult for small flows due to a lack of information, users should collect more samples to obtain more accurate tail latency estimates for these small flows.

Effects of Arbitrary Latency Quantile Estimation (Figure 12(e)): *The experimental results show that SketchPolymer is adaptable to arbitrary latency quantile estimation.* We implement Splitting Stage with 16-bit counters, and vary the query quantile from 0.1 to 0.9 in a step of 0.01. The results show that SketchPolymer still maintains a low error rate in these scenarios. When 4000KB memory is allocated, the ALE is always smaller than 0.25, and it is lower than 0.05 within 8000KB memory. The application of SketchPolymer is not limited to tail latency estimation, and users can set w to any value they are interested in.

Effects of Multi-core Implementation (Figure 12(f)): *The experimental results show that SketchPolymer is scalable when running on multiple CPU cores.* We build a SketchPolymer instance shared by all cores and use the lock mechanism for synchronization. The results show that SketchPolymer achieves much higher insertion throughput on a multi-core CPU. The insertion throughput of SketchPolymer is over 13 Mops on an 8-core CPU, which is much higher than that on a single-core CPU.

E. Experiments on RocksDB Database

In general, SketchPolymer is specifically designed for per-flow tail latency estimation in network scenarios. However, since quantile estimation is also a core operation in many database systems, we explore the potential of applying SketchPolymer in a broader context. Here, we utilize SketchPolymer to reduce the quantile query latency in real databases.

Background: RocksDB [71] is a high-performance embedded database for key-value data storage. It utilizes a data structure known as the log-structured merge tree (LSM-tree for short) to provide fast writes. LSM-tree first logs all write operations in an in-memory structure named MemTable. Once the MemTable reaches a predetermined size, it is transformed into an immutable sorted string table (SSTable for short) and asynchronously written to disk. To speed up point queries and reduce disk access, each SSTable applies a Bloom Filter to skip most non-existent items. However, Bloom Filter does not support quantile queries, and to query the per-flow tail quantile, we have to query all SSTables to find items with the given key, which is time-consuming in reality. Particularly for frequent keys, collecting all their scattered records from the database incurs massive read amplification and overhead.

Implementation: We generate 1M items, each with a 64-bit key and a 64-bit value following a Zipfian distribution with $\alpha = 1$. Here, items with the same key are considered as belonging to the same flow, and the associated value is interpreted as the latency of that item. This interpretation allows us to naturally formulate per-flow tail latency estimation in the database setting, where we query the tail quantile for each distinct key. To handle items with the same key, we add a distinct 16-bit suffix to the key to distinguish them, so RocksDB will not report two records with the same key. We add an extra SketchPolymer to the RocksDB database for per-flow tail quantile estimation. Before inserting an item into the RocksDB database, we insert it into SketchPolymer with its original 64-bit key and 64-bit value. To query the w-quantile of item e, we directly query SketchPolymer to get the result, while the scan-based exact baseline (RocksDB without SketchPolymer) has to select all items from the database with that key and calculate the result since RocksDB does not directly support quantile estimation.

Experiments (Figure 13): *The experimental results show that SketchPolymer can substantially reduce the latency of quantile queries compared with the scan-based exact baseline.* We measure the tail quantile query latency with and without SketchPolymer, and the results show that the P99 tail latency with SketchPolymer is smaller than $10\mu s$. In contrast, the scan-based exact baseline is much slower because it needs to retrieve all records with the queried key before computing the quantile. Its P70 query latency is close to 1ms, and its P99 query latency is around 100ms. Therefore, integrating a compact in-memory sketch into RocksDB avoids expensive scan-based quantile queries and achieves smaller latency.

VI. Hardware Implementation

In this section, we implement SketchPolymer on two hardware platforms, *i.e.* P4 tofinos and FPGA.

A. P4 Implementation

We have fully built a P4 prototype of SketchPolymer-CM on a Tofino switch [72]. Since the Tofino switch processes packets in a pipelined manner, SketchPolymer cannot support more than one hash function in Filter Stage, so we set $d^{(1)} = 1$. Also, the Tofino switch does not support logarithmic operations, so we choose $a = 2$ and SketchPolymer can calculate the logarithm value by prefix matching. We implement SketchPolymer using several registers and Stateful ALUs. We list the utilization of various hardware resources on the Tofino switch in Table II when $d^{(2)} = d^{(3)} = d^{(4)} = 1$, $n^{(1)} = n^{(2)} = 2^{15}$, $n^{(3)} = 2^{18}$ and $n^{(4)} = 2^{19}$.

TABLE II. SketchPolymer Performance on P4 Tofinos

Resource	Usage	Percentage
Hash bits	149	5.97%
Exact Xbar	54	7.03%
Ternary Xbar	8	2.02%
Stateful ALU	5	20.83%
SRAM	19	3.96%
TCAM	2	1.39%
Map RAM	17	5.9%

B. FPGA Implementation

We have implemented SketchPolymer-CM on an Altera FPGA. The model 5SEEBF45I2 in the Stratix V family is used as the core chip. The architecture design diagram is illustrated in Figure 14. Two hash functions are used, and the results are stored in two separate RAMs. More hash functions can be supported if needed. FPGA does not support logarithmic operations, so $a = 2$ is chosen to simplify calculation.

The resource usage is listed in Table III with $d^{(1)} = d^{(2)} = d^{(3)} = d^{(4)} = 2$, $n^{(1)} = n^{(2)} = n^{(3)} = n^{(4)} = 2^{10}$. 1) We use 1,560 logics, less than 1% of the 359,200 total available. 2) We use 390 pins, 46% of the 840 total pins, and 98.5% of used pins are for 64-bit data input and output. 3) We use 526,336 bits of Block RAM, less than 1% of the 54,067,200 total on-chip RAM. 4) We use 8 DSP Blocks, 2% of the 352 total DSP Blocks. The clock frequency of our implemented FPGA is 143.23 MHz, meaning a processing speed of 143.23 Mops.

TABLE III. SketchPolymer Performance on FPGA Platform

Resource	Usage	Percentage
Logics	1,560	<1%
Pins	390	46%
Block memory bits	526,336	<1%
DSP Blocks	8	2%

VII. Conclusion

Tail latency estimation is important in many scenarios. In this paper, we propose SketchPolymer to estimate per-flow tail latency. SketchPolymer uses **Early Filtration** to filter small flows, and **Latency Splitting and Sharing** to estimate tail latency of large flows efficiently. We implement SketchPolymer on three platforms: CPU, P4 switches, and FPGA. Both theoretical and experimental results show that SketchPolymer is much faster and more accurate than the state-of-the-art. We have released our code on GitHub [73].

References

[1] J. Guo, Y. Hong, Y. Wu, Y. Liu, T. Yang, and B. Cui, "Sketchpolymer: Estimate per-item tail quantile using one sketch," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 590–601.

[2] R. B. Halima, E. Fki, K. Drira, and M. Jmaiel, "A large-scale monitoring and measurement campaign for web services-based applications," *Concurrency and computation: practice and experience*, vol. 22, no. 10, pp. 1207–1222, 2010.

[3] S. Abbaspour Asadollah and T. Kian Chiew, "Web service response time monitoring: architecture and validation," in *International Conference on Theoretical and Mathematical Foundations of Computer Science*. Springer, 2011, pp. 276–282.

[4] Y. Liu, F. Li, L. Guo, B. Shen, S. Chen, and Y. Lan, "Measurement and analysis of an internet streaming service to mobile devices," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2240–2250, 2012.

[5] M. Chesire, A. Wolman, G. M. Voelker, and H. M. Levy, "Measurement and analysis of a streaming media workload," in *3rd USENIX Symposium on Internet Technologies and Systems (USITS 01)*, 2001.

[6] Y. Bai, B. Jia, J. Zhang, and Q. Pu, "An efficient load balancing technology in cdn," in *2009 Sixth International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 7. IEEE, 2009, pp. 510–514.

[7] H. He, Y. Feng, Z. Li, Z. Zhu, W. Zhang, and A. Cheng, "Dynamic load balancing technology for cloud-oriented cdn," *Computer Science and Information Systems*, vol. 12, no. 2, pp. 765–786, 2015.

[8] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[9] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, 2002, pp. 323–336.

[10] J. Huang, W. Zhang, Y. Li, L. Li, Z. Li, J. Ye, and J. Wang, "Chainsketch: An efficient and accurate sketch for heavy flow detection," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 738–753, 2022.

[11] Q. Xiao, X. Cai, Y. Qin, Z. Tang, S. Chen, and Y. Liu, "Universal and accurate sketch for estimating heavy hitters and moments in data streams," *IEEE/ACM Transactions on Networking*, vol. 31, no. 5, pp. 1919–1934, 2023.

[12] G. Gao, T. Ma, H. Huang, Y.-E. Sun, H. Wang, Y. Du, and S. Chen, "Scout sketch +: Finding both promising and damping items simultaneously in data streams," *IEEE/ACM Transactions on Networking*, 2024.

[13] Z. Fan, X. Wang, X. Li, J. Guo, W. Liu, H. Li, S. Long, Z. Zhong, T. Yang, X. Chen *et al.*, "Steadysketch: A high-performance algorithm for finding steady flows in data streams," *IEEE/ACM Transactions on Networking*, 2024.

[14] C. Misa, R. Durairajan, R. Rejaie, and W. Willinger, "Dynatos +: A network telemetry system for dynamic traffic and query workloads," *IEEE/ACM Transactions on Networking*, 2024.

[15] J. Cai, H. Lin, T. Sun, Z. Zhou, L. Zhu, H. Chen, J. Zhou, D. Zhang, and C. Wu, "Openint: Dynamic in-band network telemetry with lightweight deployment and flexible planning," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 2488–2497.

[16] X. Jiang, H. Shokri-Ghadikolaei, G. Fodor, E. Modiano, Z. Pang, M. Zorzi, and C. Fischione, "Low-latency networking: Where latency lurks and how to tame it," *Proceedings of the IEEE*, vol. 107, no. 2, pp. 280–306, 2018.

[17] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. Sarolahti, and J. Ott, "Measuring web latency and rendering performance: Method, tools, and longitudinal dataset," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 535–549, 2019.

[18] S. Wei and V. Swaminathan, "Low latency live video streaming over http 2.0," in *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, 2014, pp. 37–42.

[19] M. Iorio, F. Risso, and C. Casetti, "When latency matters: measurements and lessons learned," *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 4, pp. 2–13, 2021.

[20] U. Fiedler and B. Plattner, "Using latency quantiles to engineer qos guarantees for web services," in *Quality of Service—IWQoS 2003: 11th International Workshop Berkeley, CA, USA, June 2–4, 2003 Proceedings 11*. Springer, 2003, pp. 345–362.

[21] G. Lim, M. S. Hassan, Z. Jin, S. Volos, and M. Jeon, "Approximate quantiles for datacenter telemetry monitoring," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1914–1917.

[22] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 325–341.

[23] B. Cai, B. Wang, M. Yang, and Q. Guo, "Automan: Resource-efficient provisioning with tail latency guarantees for microservices," *Future Generation Computer Systems*, vol. 143, pp. 61–75, 2023.

[24] J. Sommers, P. Barford, N. Duffield, and A. Ron, "Accurate and efficient sla compliance monitoring," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007, pp. 109–120.

[25] C. Chen, Y. Chen, K. Zhang, M. Ni, S. Wang, and R. Liang, "System redundancy enhancement of secondary frequency control under latency attacks," *IEEE Transactions on Smart Grid*, vol. 12, no. 1, pp. 647–658, 2020.

[26] M. Shahzad and A. X. Liu, "Accurate and efficient per-flow latency measurement without probing and time stamping," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3477–3492, 2016.

[27] M. Lee, N. Duffield, and R. R. Kompella, "Not all microseconds are equal: Fine-grained per-flow measurements with reference latency interpolation," in *Proceedings of the ACM SIGCOMM 2010 conference*, 2010, pp. 27–38.

[28] L. Zhao, P. Pop, and S. S. Craciunas, "Worst-case latency analysis for ieee 802.1 qbv time sensitive networks using network calculus," *Ieee Access*, vol. 6, pp. 41 803–41 815, 2018.

[29] N. Finn, "Introduction to time-sensitive networking," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 22–28, 2018.

[30] S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi, "Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, 2015, pp. 7–16.

[31] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson, "Scalable tail latency estimation for data center networks," *arXiv preprint arXiv:2205.01234*, 2022.

[32] J. I. Munro and M. S. Paterson, "Selection and sorting with limited storage," *Theoretical computer science*, vol. 12, no. 3, pp. 315–323, 1980.

[33] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 426–435, 1998.

[34] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 58–66, 2001.

[35] D. Felber and R. Ostrovsky, "A randomized online quantile summary in $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ words," *Theory of Computing*, vol. 13, no. 1, pp. 1–17, 2017.

[36] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, 2016, pp. 71–78.

[37] F. Zhao, S. Maiyya, R. Wiener, D. Agrawal, and A. E. Abbadi, "Kll±approximate quantile sketches over dynamic datasets," *Proceedings of the VLDB Endowment*, vol. 14, no. 7, pp. 1215–1227, 2021.

[38] Y. Liu and X. Xie, "A probabilistic sketch for summarizing cold items of data streams," *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, pp. 1287–1302, 2023.

[39] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 741–756.

[40] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, T. Zhao, Z. Jia, Y. Yang *et al.*, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021, pp. 1–12.

[41] Z. Fan, J. Guo, X. Li, T. Yang, Y. Zhao, Y. Wu, B. Cui, Y. Xu, S. Uhlig, and G. Zhang, "Finding simplex items in data streams," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 1953–1966.

[42] L. Gu, Y. Tian, W. Chen, Z. Wei, C. Wang, and X. Zhang, "Per-flow network measurement with distributed sketch," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 411–426, 2023.

[43] H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui, "Stingy sketch: a sketch framework for accurate and fast frequency estimation," *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1426–1438, 2022.

[44] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[45] G. Luo, L. Wang, K. Yi, and G. Cormode, "Quantiles over data streams: experimental comparisons, new analyses, and further improvements," *The VLDB Journal*, vol. 25, no. 4, pp. 449–472, 2016.

[46] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *arXiv preprint arXiv:1908.10693*, 2019.

[47] J. He, J. Zhu, and Q. Huang, "Histsketch: A compact data structure for accurate per-key distribution monitoring," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023.

[48] R. B. Arellano-Valle, H. Bolfarine, and V. H. Lachos, "Skew-normal linear mixed models," *Journal of data science*, vol. 3, no. 4, pp. 415–438, 2005.

[49] K. Cheng, L. Xiang, and M. Iwaihara, "Time-decaying bloom filters for data streams with skewed distributions," in *15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA'05)*. IEEE, 2005, pp. 63–69.

[50] J. Gao, W. Fan, J. Han, and P. S. Yu, "A general framework for mining concept-drifting data streams with skewed distributions," in *Proceedings of the 2007 siam international conference on data mining*. SIAM, 2007, pp. 3–14.

[51] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[52] J. Kurose and K. Ross, "Computer networks: A top down approach featuring the internet," 2010.

[53] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 139–152.

[54] T. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1. IEEE, 2002, pp. 170–179.

[55] Y. Liao, W. Du, P. Geurts, and G. Leduc, "Dmfsgd: A decentralized matrix factorization algorithm for network distance prediction," *IEEE/ACM Transactions on Networking*, vol. 21, no. 5, pp. 1511–1524, 2012.

[56] S. Kim, S. M. M. Mirnajafizadeh, B. Kim, R. Jang, and D. Nyang, "Sketchfeature: High-quality per-flow feature extractor towards security-aware data plane." in *NDSS*, 2025.

[57] R. Shahout, R. Friedman, and R. Ben Basat, "Together is better: Heavy hitters quantile estimation," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–25, 2023.

[58] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[59] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005, pp. 398–412.

[60] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "Salsa: self-adjusting lean streaming analytics," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 864–875.

[61] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[62] C. Xu and H. Chen, "A hybrid data mining approach for anomaly detection and evaluation in residential buildings energy data," *Energy and Buildings*, vol. 215, p. 109864, 2020.

[63] J. Cao, M. Xu, Q. Li, K. Sun, and Y. Yang, "The loft attack: Overflowing sdn flow tables at a low rate," *IEEE/ACM Transactions on Networking*, vol. 31, no. 3, pp. 1416–1431, 2022.

[64] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "{RobinHood}: Tail latency aware caching–dynamic reallocation from {Cache-Rich} to {Cache-Poor}," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 195–212.

[65] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, "Small is better: Avoiding latency traps in virtualized data centers," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[66] "Murmurhash," https://github.com/aappleby/smhasher.

[67] "The CAIDA Anonymized Internet Traces." http://www.caida.org/data/overview/.

[68] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: a platform for educational cloud computing," in *Proceedings of the 40th ACM technical symposium on Computer science education*, 2009, pp. 111–115.

[69] R. Zhu, B. Liu, D. Niu, Z. Li, and H. V. Zhao, "Network latency estimation for personal devices: A matrix completion approach," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 724–737, 2016.

[70] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. Sarolahti, and J. Ott, "Measuring web latency and rendering performance: Method, tools & longitudinal dataset," https://doi.org/10.5281/zenodo.2547512, Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2547512

[71] "Facebook RocksDB," http://rocksdb.org/.

[72] "Barefoot tofino: World's fastest p4-programmable ethernet switch asics." https://barefootnetworks.com/products/brief-tofino/.

[73] "The source codes related to SketchPolymer." https://github.com/ntguojiarui/SketchPolymer-new.

Jiarui Guo received his bachelor degree in the School of Electronic Engineering and Computer Science at Peking University in 2023. He is currently pursuing the Ph.D. degree with the School of Computer Science, Peking University. His research focuses on the design of high-performance data structures and algorithms, with a dedicated effort to bridge algorithmic theory and system performance in LLM inference acceleration, network telemetry, and database kernels.

Yuqi Dong is an undergraduate student of Peking University majoring in Computer Science. His research interests include network measurements and sketches.

Yuhan Wu received his bachelor degree in the School of Electrical Engineering and Computer Science at Peking University in 2021. Currently he is a CS Ph.D. student in School of Computer Science at Peking University, advised by Tong Yang. His research interests lie in the fields of computer network and database, including key-value stores, network measurement, and sketches.

Yisen Hong received the B.S. degree in Computer Science from Peking University in 2022. He is currently pursuing the master degree of software engineering with Peking University, advised by Tong Yang and Huiping Sun. He has contributed to several publications in the field of networking. His research focuses primarily on network measurements and sketch algorithms.

Yunfei Liu received the Ph.D. degree in microelectronics from Peking University in 2023. His research interests include digital circuit and digital signal processing.

Xiaolin Wang received the BS and PhD degrees from Peking University, in 1996 and 2001, respectively. He is an associate professor with Peking University. His research interests include system software, virtualization technologies and distributed computing, etc.

Yong Cui (Member, IEEE) received the B.E. and Ph.D. degrees in computer science and engineering from Tsinghua University, China, in 1999 and 2004, respectively. He is currently a Full Professor with the Computer Science Department, Tsinghua University. His research interests include mobile cloud computing and network architecture. He serves on the Editorial Boards for IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON CLOUD COMPUTING, and the IEEE INTERNET COMPUTING.

Bin Cui (Fellow, IEEE) is a professor in School of Computer Science and Director of Institute of Network Computing and Information Systems at Peking Univer sity. His research interests include database system architectures, query and index techniques, big data management and mining. He is a fellow of IEEE, member of ACM and distinguished member of CCF.

Tong Yang (Member, IEEE) received the PhD degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with School of Computer Science, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, and KV stores. He has published more than ten papers in SIGCOMM, SIGKDD, SIGMOD, NSDI, etc.