Per-Flow Quantile Estimation Using M4 Framework

Zhuochen Fan, *Member, IEEE*, Yalun Cai, Siyuan Dong, *Member, IEEE*, Qiuheng Yin, Tianyu Bai, Hanyu Xue, Peiqing Chen, Yuhan Wu, Tong Yang, *Member, IEEE*, and Bin Cui, *Fellow, IEEE*

***Abstract*—This paper introduces a novel framework, M4, designed to estimate per-flow quantiles in data streams accurately. M4 is a versatile framework that can be integrated with a wide array of single-flow quantile estimation algorithms, thereby enabling them to perform per-flow estimation. The framework employs a sketch-based approach to provide a space-efficient method for recording and extracting distribution information. M4 incorporates two techniques: *MINIMUM* and *SUM*. The *MINIMUM* technique minimizes the noise on a flow from other flows caused by hash collisions, while the *SUM* technique efficiently categorizes flows based on their sizes and customizes treatment strategies accordingly. We demonstrate the application of M4 on three single-flow quantile estimation algorithms (DDSketch, *t*-digest, and ReqSketch), detailing the specific implementation of the *MINIMUM* and *SUM* techniques. We provide theoretical proof that M4 delivers high accuracy while utilizing limited memory. Additionally, we conduct extensive experiments to evaluate the performance of M4 regarding accuracy and speed. The experimental results indicate that across all three example algorithms, M4 significantly outperforms two comparison frameworks in terms of accuracy for per-flow quantile estimation while maintaining comparable speed.**

***Index Terms*—Per-flow, quantile estimation, data streams.**

I. Introduction

A. Background and Motivation

With the development of data stream processing, accurate, real-time extraction of required information from a large volume of high-speed data streams is attracting increasing attention [2], [3], [4], [5], [6]. Among the various types of information, quantile information, which requires distribution statistics for data streams, has become a focal point of numerous studies [7], [8], [9], [10], [11], [12], [13]. Recent research trends have evolved from estimating the quantile for a single data stream to developing data structures that can concurrently estimate quantiles for multiple sub-streams, also known as flows. In practical scenarios, many metrics necessitate per-flow granularity estimation of distribution, such as Latency [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], Inter-Arrival Time [24], [25], [26], Packet Size [27], [28], [29], [30], and TTL (Time to Live) Value [31], [32]. Accurate estimation of per-flow distribution has wide-ranging practical applications and significant potential in distributed network scenarios, including improving the quality of service (QoS) for users [33], [34], [35], enhancing network anomaly detection [36], [37], [38], and boosting the performance of Content Delivery Networks (CDNs) [39]. Consequently, the primary objective of this article is to perform quantile estimation for each individual flow in the data stream.

A data stream is a sequence of items, each represented as a *key-value* pair. Items sharing the same *key* compose a *flow*. The shared *key* serves as the *flow* ID.[1] The *value* is the metric that needs processing. All items from different flows are intermixed in a data stream ($e.g.$, $DS = \{\langle a,3\rangle, \langle a,2\rangle, \langle b,5\rangle, \langle d,1\rangle, \langle a,4\rangle, \ldots\}$). This paper uses *quantile* to demonstrate per-flow *value* distribution. The items in a flow can be represented by a multiset $\mathcal{F} = \{\langle a,x_1\rangle, \langle a,x_2\rangle, \ldots, \langle a,x_n\rangle\}$ of size n, where a is the *key*, x_i is the *value* and $x_1 \le x_2 \le \cdots \le x_n$. Given a percentage p $(0 \le p \le 1)$, the p-quantile of *value* is x^* $s.t.$ the percentage of $x_i \le x^*$ in the multiset $\mathcal{F}$ equals to p. With the above preliminaries, we give the problem definitions:

- *Per-Flow Quantile Estimation:*
  SELECT *key, p-quantile(value)*
  FROM $DataStream$
  GROUP BY key
- *Single-Flow Quantile Estimation:*
  SELECT *p-quantile(value)*
  FROM $DataStream$

Accurately estimating the per-flow distribution is of wide practical usage and has many important potential applications in distributed scenarios. We provide three use cases as follows:

1) Improving of the quality of service (QoS) for online app users: In the digital era, online applications such as real-time

Received 29 May 2024; revised 10 April 2025; accepted 24 April 2025. Date of publication 26 May 2025; date of current version 7 July 2025. This work was supported by the National Key R&D Program of China under Grant 2024YFB2906603, in part by the Project of the Department of Strategic and Advanced Interdisciplinary Research of Pengcheng Laboratory under Grant 2025QYB004, in part by the Major Key Project of Pengcheng Laboratory under Grant PCL2023A06, and in part by the National Natural Science Foundation of China (NSFC) under Grant 62402012, Grant 62372009, Grant 624B2005. Recommended for acceptance by Reza Akbarinia. *(Corresponding author: Tong Yang.)*

Zhuochen Fan is with the Department of Strategic and Advanced Interdisciplinary Research, Pengcheng Laboratory, Shenzhen 518055, China, and also with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: fanzhch@pcl.ac.cn).

Yalun Cai, Siyuan Dong, Qiuheng Yin, Tianyu Bai, Yuhan Wu, and Tong Yang are with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: caiyalun@pku.edu.cn; dongsiyuan@pku.edu.cn; yinqiuheng@stu.pku.edu.cn; tianyubai@pku.edu.cn; yuhan.wu@pku.edu.cn; yangtong@pku.edu.cn).

Hanyu Xue is with Yuanpei College, Peking University, Beijing 100871, China (e-mail: xuehy2002@stu.pku.edu.cn).

Peiqing Chen is with the Department of Computer Science, University of Maryland, College Park, MD 20742 USA (e-mail: pqchen99@umd.edu).

Bin Cui is with the Key Laboratory of High Confidence Software Technologies (MOE) & School of Computer Science, Peking University, Beijing 100871, China (e-mail: bin.cui@pku.edu.cn).

Digital Object Identifier 10.1109/TKDE.2025.3573812

[1]A flow ID is typically defined as a part of the five tuples: source IP address, destination IP address, source port, destination port, and protocol. This paper considers the number of items in a flow as the flow size, also referred to as the item frequency.

video communications, online gaming, and streaming services have become integral to daily life [33], [34], [35]. These applications demand high-quality network services to ensure seamless user experience. Any degradation in network performance, particularly in terms of latency, can significantly affect the Quality of Service (QoS). This is especially true for applications requiring real-time interactions, such as remote control systems and remote sensing applications, where delays can compromise operational integrity and user experience. The challenge for network managers in maintaining optimal QoS lies in the ability to precisely identify and rectify latency issues at a per-user granularity. By pinpointing the exact source and user with latency issues, network managers can implement targeted interventions to resolve these issues swiftly and efficiently.

2) More effective network anomaly detection: In this scenario, a flow refers to packets going through a network link with the same five-tuple ID. Network anomalies, such as congestion or the presence of malware, often manifest as abrupt increases in the latency of several flows [36], [37], [38] within a network link. These anomalies are critical to identify and mitigate, as they can severely impact network performance and security. Traditional single-flow quantile estimation approaches aggregate data across all flows, which can dilute the impact of anomalies present in a small subset of flows. This aggregation effect can lead to a failure in detecting subtle yet critical anomalies, allowing them to persist undetected and potentially cause extensive damage to the network infrastructure or compromise sensitive data. Our algorithm, by contrast, ensures that even minor deviations in latency are detected, thereby significantly enhancing the sensitivity of anomaly detection.

3) Cache performance optimization: A core challenge in this domain [40], [41], [42] is the diverse latency requirements and usage patterns across different flows, where a flow may represent a distinct user or a specific service. High latency in a flow often signals suboptimal data placement within the cache, necessitating strategic adjustments to either the data's location or its storage modality to enhance access speed. Our approach effectively evaluates cache strategy impacts on different flows, identifying when optimizations reduces average latency but inadvertently disadvantage high-priority users. It also identifies strategies that benefit latency-insensitive services at the cost of sensitive ones, allowing the development of cache strategies that enhance system efficiency without compromising critical service performance.

Numerous studies have significantly advanced the field of single-flow quantile estimation [16], [17], [18], [19], [43], [44], [45], [46]. Approximate algorithms—often referred to as sketches—have predominantly excelled in scenarios requiring low memory overhead and rapid processing, with only minimal accuracy trade-offs. However, a critical limitation inherent in these approaches is their inability to discern among multiple flow IDs. They are designed either to estimate quantiles for individual keys in isolation or to aggregate across all data stream values without key differentiation. This results in a binary choice: a focused but isolated single-flow estimation or a comprehensive yet undifferentiated analysis across all flows. It fails to address the nuanced requirements of modern networked systems, where identifying and analyzing per-flow metrics is crucial for optimizing performance and detecting anomalies.

The evolution of applications and the diversity of their requirements necessitate a more granular approach to quantile estimation—one that can accurately measure and adapt to the unique characteristics of each flow. To bridge this gap, there are two strategic paths: designing a brand new algorithm tailored for per-flow estimation [47], [48] or a versatile framework capable of enabling existing single-flow algorithms to handle per-flow queries. We advocate for the latter, recognizing its potential for broad applicability and design simplicity. This approach leverages the strengths of existing algorithms to cater to scenarios with varying priorities like throughput, accuracy, and memory efficiency.

By opting for a framework that transforms single-flow algorithms into their per-flow counterparts, we present a solution that is both adaptable and scalable. This framework not only retains the inherent advantages of the original algorithms but also expands their utility to support per-flow estimation, thereby addressing a critical need in network management and analysis. In doing so, our approach provides a comprehensive toolset for network administrators and researchers, enabling them to tackle a wide array of challenges with unprecedented precision and flexibility.

There are several challenges when designing such a framework. **(1) Algorithmic Compatibility.** Different single-flow algorithms may have unique characteristics, optimizations, and assumptions that may not directly translate to a per-flow context. The framework must provide a flexible architecture that can adapt various single-flow algorithms to per-flow requirements without compromising their inherent advantages. **(2) Scalability.** One of the foremost challenges is ensuring that the framework scales efficiently with the number of flows. Single-flow algorithms are typically optimized for performance with a single data stream. Extending these to accommodate multiple, potentially thousands or millions of flows, can introduce significant computational and memory overhead. The framework must efficiently manage resources to maintain high performance and accuracy across all flows. **(3) Data Skew and Flow Variability.** In real-world networks, some flows may be more active or larger than others, leading to data skew. The framework needs to handle such variability, ensuring that large or high-volume flows do not overshadow smaller ones.

B. Our Solution and Contributions

To achieve our design goal, we propose a novel framework named MINIMUM-SUM (M4). M4 is a framework that can be applied to an extensive range of single-flow quantile estimation algorithms, enabling them to perform per-flow quantile estimation. For simplicity, we refer to the single-flow algorithm on which we employ M4 as *META*. As depicted in Fig. 1, M4 uses limited memory to construct several layers of buckets. Each bucket contains a *META* to record distribution. Every *META* treats all incoming items identically. We use hash functions to map flows to buckets for recording. A single flow can be mapped to multiple buckets. If a hash collision occurs, the distribution of the collided flows will sum up in the bucket. Each bucket has a load capacity. The insertion of a flow starts at a lower layer. When the buckets in the lower layer overflow, we insert the subsequent items into the upper layers. Buckets in higher layers have larger capacity and finer granularity. M4 comprises two techniques: *MINIMUM* and *SUM*. As long as the *META* can generate the *value* distribution for a single flow, we can use *MINIMUM* and *SUM* to transform it into an efficient per-flow quantile estimation algorithm.

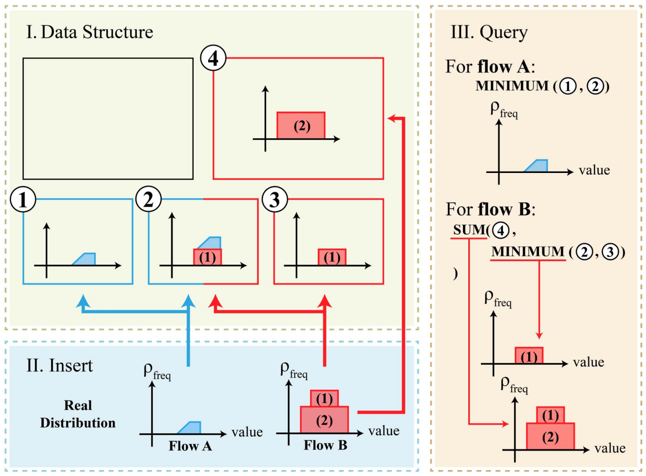


Fig. 1. **Illustration of M4.** Arrows pointing from II to I represent the hashing operation. Flow A is represented by blue. Flow B is represented by red.

MINIMUM resolves hash collisions by randomly selecting several buckets for each flow in each layer using multi-hashes and extracting the real distribution from these selected buckets. To address hash collisions, a straightforward solution is to use a hash table of buckets, each containing a *META* to record the *value* distribution of a flow. However, this approach requires recording IDs and executing complex operations (typically dependent on the number of flows) to locate another available bucket during a hash collision, which is both memory and time consuming. A more efficient solution is *Memory Sharing*. We use w hash functions to map a flow to w buckets of *META* for recording, providing us with w records for every flow. Due to hash collisions, more than one flow may be mapped into one bucket. Thus, every flow record may contain some noise from other flows, and we need to compare and analyze the distributions given by all the records to restore the real distribution. For instance, as shown in Fig. 1, a small flow A and a large flow B are inserted into M4.[2] Flow A can be contained in the first layer, while flow B is segmented into parts I and II[3] due to its large size. Flow A and B encounter a hash collision at bucket ② in the first layer, so their distribution information is mixed in the bucket. We need to use bucket ① and ② to restore the real distribution of flow A and use bucket ② and ③ to restore the real distribution of part I of flow B. Because noise at one *value* point only increases the density at that point, by selecting the lowest density at each *value* point, the estimated *value* distribution bears the slightest noise possible. As we take the intersection of distributions, we call it the *MINIMUM* technique. We can see in Fig. 1 that the *MINIMUM* technique allows us to restore the real distribution of flow A and part I of flow B. It should be noted that if *META* can give an error-free result when estimating single-flow distribution, our *MINIMUM* is optimal and can guarantee a unilateral error of over-estimation when estimating per-flow distribution.

SUM categorizes flows based on their sizes through segmentation. It uses multiple layers to segment every flow into multiple parts and tailor the treatment in each layer. Then it aggregates these parts to generate the full distribution. There are two reasons why we need flow categorization. First, the flow size distribution in a data stream is usually highly skewed. For example, in CAIDA [49] dataset, about 40% flows only contain ≤ 3 items. It would be a colossal waste to allocate equal resources to a small flow and a large one. Second, large flows get more attention in practical applications[4]. Hence, the larger a flow is, the more resources should be allocated to achieve a fair or better accuracy than small flows. Thus, we want to categorize flows according to their sizes efficiently. However, we do not know if a flow is large or small in advance. To solve this, M4 has a layered structure, where lower layers are for coarse-grained recording of small flows and higher layers are for fine-grained recording of large flows. Each flow is considered a small flow in the beginning. When the recorded flow size exceeds the capacity in lower layers, we know it is a large flow, so we insert the subsequent items into higher layers. In this way, the distribution information of a large flow is scattered at multiple layers. We need to aggregate all parts together to get the entire distribution. As shown in Fig. 1, we need to use bucket ④ and (②*MINIMUM*③) to restore the real distribution of flow B. Because the distribution in two layers corresponds to two disjoint parts of flow B, we should sum up the density at each *value* point to construct the overall distribution information. Therefore, we call it the *SUM* technique. We can see in Fig. 1 that the *SUM* technique allows us to restore the real distribution of flow B.

We apply our M4 to three *META*s (DDSketch [16], t-digest [17], [19], and modified ReqSketch [18], [43]). The three *META*s each have their emphasis. DDSketch allows us to focus on the tail value distribution and bounds the relative error of quantile estimation to a constant at different percentages. t-digest allows us to tailor the relative accuracy of quantile estimation at different percentages. ReqSketch provides a relative guarantee on the error of rank estimation. We design the *MINIMUM* and *SUM* techniques for them according to their features. See Section III for more details. Further, we provide theoretical proof that M4 delivers high accuracy while utilizing limited memory in Section IV.

We conduct extensive experiments to evaluate our performance regarding accuracy and speed in Section V. We devise two comparison frameworks for better comparison. CPU experimental results indicate that M4 is per-flow friendly and accurate. For tiny flows, maximum value estimation is on average 90.6% error-free, while comparison frameworks offer almost no error-free estimates. For larger flows, the Average Logarithm Error (ALE) of M4 reach 2.26× lower than comparison frameworks. M4 is memory-efficient. It only needs 6 MB to handle 27M items. Finally, we implement M4 entirely on the Tofino platform. All codes are available on GitHub [50].

Key contributions:

- We introduce M4, the first general framework that can be applied to a wide range of single-flow quantile estimation algorithms to accomplish per-flow quantile estimation, filling a gap in the research field.
- We propose the *MINIMUM* and *SUM* techniques. Together, they reduce the error from hash collisions and allow us to tailor treatment strategies for flows of different sizes.

[2]There is a predefined threshold that classifies flows based on their sizes. Flows with sizes below this threshold are called small flows, otherwise they are called large flows.

[3]We cut flow B into top and bottom halves just for simplicity.

[4]Large flows tend to represent critical or high-priority data. By paying more attention to large flows, administrators can optimize the delivery of important information and enhance overall system performance.

- We apply M4 to DDSketch, t-digest, and modified ReqSketch and implement them on a CPU platform. Compared to two comparison frameworks, M4 achieves significantly better accuracy with a comparable speed across all three algorithms. M4 is also implemented in a P4 version.

II. Related Work

A. Sketch

A sketch is a type of probabilistic data structure designed to process data with small and controllable errors. One of the most classic sketches is CM Sketch [51], designed for estimating item frequency. CM Sketch consists of d arrays, each array $A_i (1 \leq i \leq d)$ has w counters and is associated with a hash function $h_i(\cdot)$. When an incoming item e is inserted, we increase the counter $A_i[h_i(e)\%w]$ by 1 for all $i \in \{1, 2, \ldots, d\}$. To query the item e, CM Sketch reports the minimum counter among all the d mapped counters determined by hash functions. Other classic sketches include Flajolet-Martin (FM) Sketch [52], CU Sketch [53], Count Sketch [54], CSM Sketch [55] and CMM Sketch [56].

B. Single-Flow Quantile Estimation

Quantile estimation approximates the distribution of metrics, essential for analyzing large or streaming datasets. While previous studies have advanced single-flow quantile estimation [16], [17], [18], [19], [43], [44], [45], [46], [57], [58], they fall short in multi-flow scenarios typical in data streams from diverse sources. Our framework, M4, extends these methods to estimate distributions per flow. We illustrate its application using three algorithms: DDSketch, t-digest, and ReqSketch, each with unique advantages. DDSketch allows us to focus on the tail *value* distribution and bounds the relative error of quantile estimation to a constant at different percentages. t-digest allows us to tailor the relative accuracy of quantile estimation at different percentages. ReqSketch provides a relative guarantee on the error of rank estimation.

1) DDSketch: DDSketch [16] is designed for estimating value distributions by partitioning the value range into segments, each monitored by a counter for values within the segment. The segment boundaries are determined by $\gamma := (1+\alpha)/(1-\alpha)$, with each segment's counter, C_i, tallying values x in the range $\gamma^{i-1} < x \leq \gamma^i$. The insertion of a value x is indexed by $\lceil log_\gamma(x) \rceil$. DDSketch approximates values in a segment by $\hat{x} = \frac{2\gamma^i}{\gamma+1}$, maintaining a relative error within α. The trade-off between the range coverage and accuracy is governed by α: a higher α extends the range but reduces accuracy. To estimate the value at a certain percentile $p \in [0, 1]$, we sum counters up to the relevant segment i and use $\hat{x} = \frac{2\gamma^i}{\gamma+1}$ as the quantile estimate.

2) t-Digest: The t-digest [17], [19] algorithm clusters real-valued samples to approximate value distributions, grouping items by similarity. Each cluster records the mean *value* and total count of its items. Items are added to the closest cluster, updating its statistics. t-digest controls cluster sizes to balance precision and memory use, adjusting cluster counts through a scale function k and a compression parameter δ. The function k ensures uniform cluster weight growth, allowing more clusters where data is denser. For quantile queries, weights are summed until the target cluster is identified, assuming uniform distribution within clusters to estimate the queried value.

t-digest [17], [19] is designed to estimate *value* distributions by clustering real-valued samples. t-digest uses clusters to group items with near *values*. Each cluster contains an $average$ cell recording the mean *value* of absorbed items, and a $weight$ cell recording the total number of absorbed items. Each incoming item is assigned to the cluster with the nearest average *value*, after which the average and weight cells of that cluster are updated. The key idea of t-digest is to confine the weight of each cluster to an appropriate level, being small enough to record the distribution accurately, while large enough to avoid unacceptable memory costs. Accurate confinement is achieved by constantly monitoring all clusters' weights and keeping them at the same level. There are a non-decreasing $scale\ function\ k : [0,1] \rightarrow \mathbb{R}$ describing the weight restriction and a $compression\ parameter\ \delta$ bounding the number of clusters used. We define w_i as cluster C_i's $weight$ value, and N as $\sum_i w_i$. Each w_i must satisfy: $k(\frac{w_{<i}+w_i}{N}) - k(\frac{w_{<i}}{N}) \leq \frac{1}{\delta}$. As a result, t-digest allocates more clusters to the segment of *value* with more items. Besides, we can tailor the relative accuracy of quantile estimation at different percentages by changing the *scale function* k. To query for the *value* at percentage $p \in [0, 1]$, we accumulate cluster weights until finding the cluster that p falls in. Deeming that items are uniformly distributed in each cluster, we can get the estimated *value* according to the position of p in that cluster.

3) ReqSketch: ReqSketch [18], [43] employs multiple levels of *compactors* to store item *values*, utilizing $O(\log N)$ *compactors* for a flow size N, each with a buffer of similar size. Items enter at level 0 and, upon a compactor's capacity being reached, a sorted even-sized subset is compacted and half its elements are elevated to the next level, preserving total item weight due to the weighting scheme where items at level h are assigned a weight of 2^h. This mechanism, known as compaction, ensures the integrity of distribution estimates. To estimate a value at a certain percentile p, item weights are cumulatively tallied from the smallest value until reaching p, with the corresponding item's value serving as the estimate.

III. M4 Design

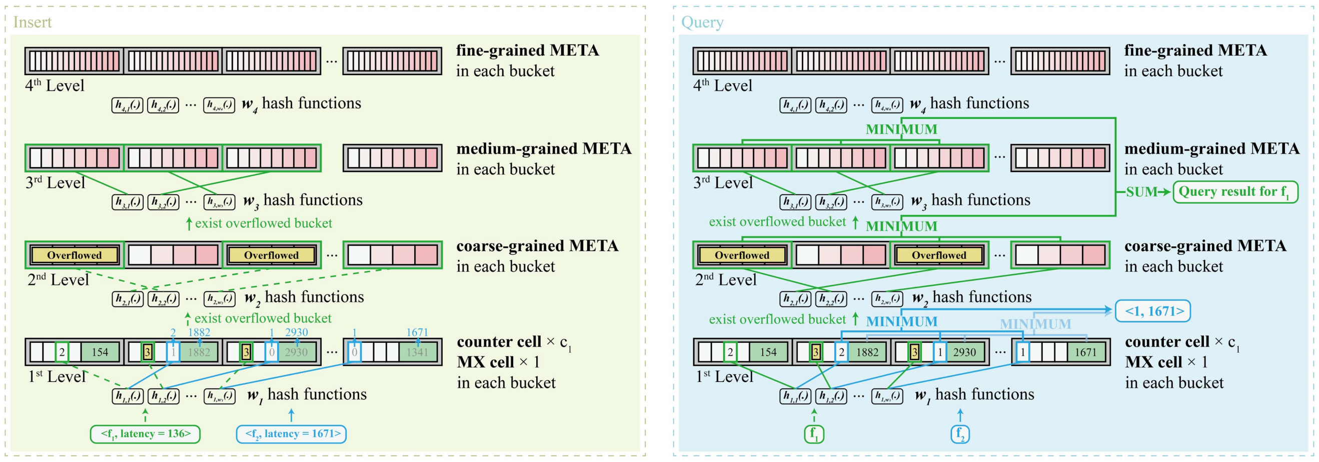
A. Problem Statement

Definition 1 (Data Stream): A data stream is a series of items appearing in sequence. Each item e_i is a *key-value* pair. The *key* serves as an ID, while the *value* represents the metric we aim to process. An example of a data stream is $DS = \{\langle a, 3\rangle, \langle a, 2\rangle, \langle b, 5\rangle, \langle d, 1\rangle, \langle a, 4\rangle, \ldots\}$.

Definition 2 (Flow): Items sharing the same *key* compose a *flow*, and the shared *key* is their *flow* ID. The number of items in a flow is the flow size, also called the item frequency. An example of a flow is $\mathcal{F} = \{\langle a, 3\rangle, \langle a, 2\rangle, \langle a, 4\rangle, \ldots\}$.

Definition 3 (Quantile): Given a numerical multiset $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$ of size n, where $x_1 \leq x_2 \leq \cdots \leq x_n$, and a percentage p $(0 \leq p \leq 1)$, the p-quantile of multiset $\mathcal{S}$ is defined as $x_{\lfloor p(n-1) \rfloor + 1}$.

Per-Flow Quantile Estimation: Given an arbitrary flow f in a data stream of *key-value* pairs and a percentage p, we need to estimate the p-quantile of *value* in f. To express it in SQL:



(a) Insert (b) Query

Fig. 2. Framework structure of M4.

```
CREATE TABLE DataStream (
key   int,
value  int
)
/∗ Insert items into DataStream. ∗/
SELECT key, p-quantile(value)
FROM DataStream
GROUP BY key
```

TABLE I
NOTATIONS

Symbol	Meaning
e	an item in the data stream
f	the *key* (flow ID) of a certain item
v	the *value* of a certain item
L_i	the i^{th} level of bucket array
b_i	the number of buckets in L_i
w_i	the number of hash functions associated with L_i
$h_{i,j}(\cdot)$	the j^{th} hash function in L_i
l_i	the capacity parameter for buckets in L_i
c_i	the granularity parameter for buckets in L_i

B. Framework Description

This section describes the structure and operations of M4. Frequently used notations are outlined in Table I.

1) Framework Structure: In Fig. 2, M4 is presented as a four-tiered bucket array, chosen for its balance between precision and efficiency. Each level L_i comprises b_i buckets and employs w_i hash functions ($h_{i,1}(\cdot)$ to $h_{i,w_i}(\cdot)$). L_1 captures the size and maximum *value* of small flows, L_2 the *value* distribution of moderate flows, and L_3 and L_4 the distribution of large flows. We classify tiny, medium, and huge flows as having sizes in [1,3), [3,255), and [255, $+\infty$), respectively. The algorithm M4, when integrated with *META* (DDSketch, t-digest, or modified ReqSketch), is referred to as M4-*META*.

Each L_1 bucket in any M4-*META* contains c_1 counters and an MX cell for flow size and maximum *value*, respectively, with a counter cell of l_1 bits. Overflows occur when a cell's count maxes out. L_1 is designed for scenarios where small flows are less critical.

For levels L_i where $i \geq 2$, each bucket holds a *META*, with uniform capacity and granularity within the level but varying across levels to accommodate the significance and size of the flows. A bucket overflows once its *META* reaches capacity. The details for different *META*s on $L_i (i \geq 2)$ are as follows:

DDSketch: Referencing Section II-B1, DDSketches in levels $L_i (i \geq 2)$ consist of c_i counter cells ($c_2 \leq c_3 \leq c_4$), each tracking the frequency of values within distinct segments. The bit length of counters in L_i is l_i ($l_2 \leq l_3 \leq l_4$). A bucket in L_i overflows when any counter cell's frequency hits its maximum l_i-bit value.

DDSketch(C): DDSketch with collapsing strategy [DDSketch(C)] is an optimized variant of the original data structure, designed to efficiently manage memory usage while maintaining accurate quantile estimates. DDSketch(C) implements a specific collapsing mechanism that works as follows: When the number of buckets exceeds the maximum limit, we remove the smallest bucket (index 0) and merge its count into the next smallest bucket. This merging process effectively collapses adjacent buckets at the lower end of the distribution. Technically, the collapsing happens by: 1) Removing the bucket at position 0 from our ordered vector of $\langle position, count \rangle$ pairs; 2) Adding its count value to the new smallest bucket (which becomes the new position 0); 3) Maintaining the ordered structure of buckets by position. This approach creates a variable-width binning where smaller values have wider buckets (reduced precision) while larger values maintain narrow buckets (higher precision). The key insight is that we selectively sacrifice precision for smaller values to save memory while retaining accuracy for higher quantiles at the tail of the distribution that are typically more important.

t-digest: As per Section II-B2, a t-digest in levels $L_i (i \geq 2)$ includes c_i clusters ($c_2 \leq c_3 \leq c_4$), with each cluster holding an *average* and a *weight* cell for the *mean* value and item count, respectively. The weight cell length in L_i is l_i ($l_2 \leq l_3 \leq l_4$). Overflow occurs when a bucket's weight cell frequency in L_i maxes out its l_i-bit capacity.

mReqSketch: Mentioned in Section II-B3, mReqSketches in $L_i (i \geq 2)$ comprise l_i compactors ($l_2 \leq l_3 \leq l_4$) with c_i cells each ($c_2 \leq c_3 \leq c_4$) for value storage. The maximum weight for

Algorithm 1: Framework Insertion.

```
Function Insert-To-Framework($e$):
    Input: Item : $e = \langle f, v \rangle$
    for $i = 1, 2, 3, 4$ do
        if not $L_i$.isOverflowed($f$) then
            Insert-To-Level($e, i$)
            return
        end
    end
    /* control never reaches here */
End Function
Function Insert-To-Level($e, i$):
    Input: Item: $e = \langle f, v \rangle$, level: $i$ in $\{1, 2, 3, 4\}$
    if $i == 1$ then
        for $j = 1, \cdots, w_1$ do
            $buc\_idx$ = Calc-Buc-Idx-In-Level($f, 1, j$)
            $cnt\_idx = h_{1,j}(f)\%c_1$
            $L_1[buc\_idx]$.counters$[cnt\_idx] += 1$
            $L_1[buc\_idx]$.MX =
              $\max(L_1[buc\_idx].MX, v)$
        end
    else
        for $j = 1, \cdots, w_i$ do
            $buc\_idx$ = Calc-Buc-Idx-In-Level($f, i, j$)
            $L_i[buc\_idx]$.insert($e$)
        end
    end
End Function
Function Calc-Buc-Idx-In-Level($f, i, j$):
    Input: Flow ID: $f$, level: $i$ in $\{1, 2, 3, 4\}$, index of
      hash function: $j$
    Output: Index of the bucket to which flow $f$ is
      mapped in $L_i$ under the $j^{th}$ hash function
    return $i == 1 \ ? \ \lfloor \frac{h_{1,j}(f)\%(b_1 c_1)}{c_1} \rfloor : h_{i,j}(f)\%b_i$
End Function
```

Algorithm 2: Framework Query.

```
Function Query-In-Framework($f$):
    Input: Flow ID: $f$
    Output: The distribution of flow $f$
    for $i = 1, 2, 3, 4$ do
        if not $L_i$.isOverflowed($f$) then
            return Query-Till-Level($f, i$)
        end
    end
    /* control never reaches here */
End Function
Function Query-Till-Level($f, top$):
    Input: Flow ID: $f$, level: $top$ in $\{1, 2, 3, 4\}$
    Output: The distribution of flow $f$, using
      information up to level $top$
    $res\_sum$ = MINIMUM-In-Level($f, top$)
    for
      $i = 2, 3, \cdots, top - 1$ /* empty set if $top \leq 2$ */
      do
        $res\_min$ = MINIMUM-In-Level($f, i$)
        $res\_sum$ = SUM($res\_min, res\_sum$)
    end
    return $res\_sum$
End Function
Function MINIMUM-In-Level($f, i$):
    Input: Flow ID: $f$, level: $i$ in $\{1, 2, 3, 4\}$
    Output: The result of MINIMUM operation of
      flow $f$ in level $i$
    $buc\_idx$ = Calc-Buc-Idx-In-Level($f, i, 1$)
    $res\_min = L_i[buc\_idx]$
    for $j = 2, \cdots, w_i$ do
        $buc\_idx$ = Calc-Buc-Idx-In-Level($f, i, j$)
        $res\_min$ =
          MINIMUM($res\_min, L_i[buc\_idx]$)
    end
    return $res\_min$
End Function
```

an mReqSketch in L_i is $(2^{l_i} - 1) \times c_i$. Overflow is determined when a bucket's recorded frequency in L_i reaches this limit.

2) Framework Insertion Operation: To insert an item $e = \langle f, v \rangle$ into M4-*META*, we target the lowest non-overflowed level, denoted as L_{top}. First, e is mapped to w_1 buckets in L_1 using the index $\lfloor \frac{h_{1,j}(f)\%(b_1 c_1)}{c_1} \rfloor$ for $j \in 1, 2, \ldots, w_1$. The $(h_{1,j}(f)\%c_1)^{th}$ counter cell in each mapped bucket is incremented, and the MX cell is updated to $\max MX, v$, unless overflow occurs, prompting an attempt to insert e into L_2.

For levels L_i where $i \geq 2$, e is mapped to w_i buckets using $h_{i,j}(f)\%b_i$. If no overflow occurs in the mapped buckets, e is inserted into the *META* of each, concluding the insertion.

The pseudo-code of the insertion operation is shown in Algorithm 1.

3) Framework Query Operation: Querying a flow with ID f involves mapping it to corresponding buckets across levels, starting from L_1 up to L_{top}, the highest non-overflowed level for f. Unlike insertion, querying aggregates results from all relevant levels up to L_{top}.

At each level L_i, we obtain w_i records for f, which may be affected by hash collisions. To mitigate this, we employ the *MINIMUM* technique to derive the least polluted distribution by selecting the minimum values from counter and MX cells for L_1, representing the size and maximum value of f. For levels L_i where $i \geq 2$, the approach adjusts based on *META*'s structure, detailed in Section III-C.

The output for f depends on top. For $top = 1$, the result is based solely on L_1 data using the *MINIMUM* method. For $top \geq 2$, it is a *SUM*-merged aggregation from L_2 to L_{top}, with specifics on the *SUM* technique in Section III-C.

The pseudo-code of the query operation is shown in Algorithm 2

4) Example: Our example uses parameters $\langle c_1 = 4, l_1 = 2, w_1 = 3 \rangle$, $\langle w_2 = 3 \rangle$, $\langle w_3 = 3 \rangle$, and $\langle w_4 = 3 \rangle$.

In the insertion operation illustrated in Fig. 2(a), item $e_1 = \langle f_1, v = 136 \rangle$ is first mapped to L_1's three buckets, but due to counter overflows, it's redirected and successfully

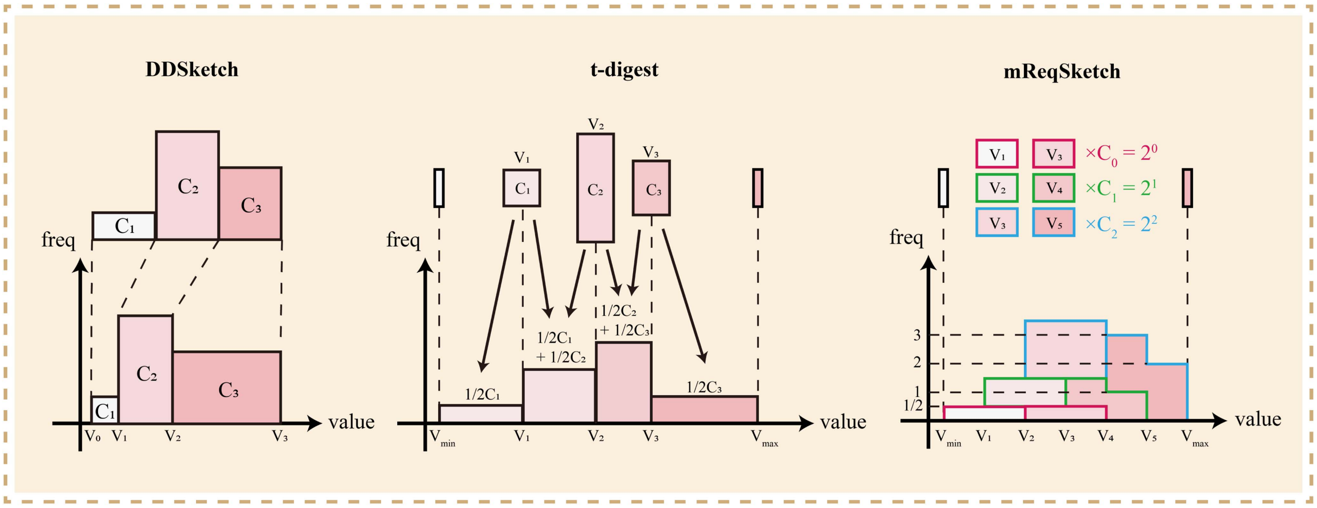


Fig. 3. From *META* to histogram.

inserted into L_3. For item $e_2 = \langle f_2, v = 1671\rangle$, mapping to L_1 reveals no overflow, so counters are incremented, and 1341 is updated to 1671.

Fig. 2(b) shows the query operation. For f_1, overflow at L_1 and L_2 leads to $top = 3$, with results merged using the *SUM* and *MINIMUM* techniques from L_2 and L_3. For f_2, L_1 provides the flow size as the minimum counter value (1) and the maximum value as the smallest *MX* cell (1671).

C. MINIMUM & SUM

In this section, we expound on applying *MINIMUM* and *SUM* to an arbitrary *META* and illustrate the workflow on three example *META*s, DDSketch, t-digest, and mReqSketch. First, we present the distribution stored in the *META* as histograms (Section III-C2). Subsequently, we perform *MINIMUM* and *SUM* operations on the histograms (Section III-C3).

*1) Rationale: MINIMUM***.** The *MINIMUM* technique mitigates hash collision effects without tracking IDs, facilitating $O(1)$ time complexity for both insertion and query operations. Recognizing that accurate distribution estimation equates to frequency estimation at each value point, this method leverages the fact that lower densities in flow buckets, resulting from hash collisions, provide a more accurate density at any given value point. Thus, selecting the minimum density from all mapped buckets yields the most reliable value distribution estimate.

SUM: The *SUM* technique efficiently categorizes flows based on their sizes and tailors treatment strategies accordingly, maximizing the overall accuracy. To achieve so, we use multiple levels to divide a flow's distribution information into various fractions. Since the information in these fractions is disjointed, we need to sum up the density at each *value* point to construct the overall distribution, akin to piecing together a jigsaw puzzle.

2) From META to Histogram: Histograms are a widely used method for representing distributions. Consequently, every *META* can transform the distribution information stored with its data structure into a histogram. In this subsection, we illustrate how DDSketch, t-digest, and mReqSketch are transformed into histograms.

DDSketch: As discussed in Section II-B1, DDSketch divides the entire range of *value* into fixed segments, each tracked by a counter cell that records the number of *values* that fall into that segment. If we index each segment by $i \in \mathbb{Z}$, then the counter C_i records the number of *value* x that falls between $V_{i-1} = \gamma^{i-1} < x \leq \gamma^i = V_i$.

The process of transforming a DDSketch into a histogram is illustrated on the left side of Fig. 3. The range of each segment on the horizontal axis is determined by V_i $(i \in \{0, 1, 2, 3\})$, and the frequency of each segment is the corresponding C_i.

DDSketch(C): The process of converting a DDSketch(C) to a histogram involves dynamically calculating bin edges based on position values and a scaling factor γ, populating the histogram with corresponding frequencies, and inserting zero-height bins to account for any gaps, ensuring a continuous and accurate representation of the frequency distribution.

t-digest: As discussed in Section II-B2, t-digest uses clusters to group items with near *values*. Each cluster contains an $average$ cell V_i recording the mean *value* of absorbed items, and a $weight$ cell C_i recording the total number of absorbed items. Each incoming item is assigned to the cluster with the nearest average *value*, after which the V_i and C_i of that cluster are updated. Besides, t-digest records the minimum *value* $V_{\min}$ and the maximum *value* $V_{\max}$.

The process of transforming a t-digest into a histogram is illustrated in the middle of Fig. 3. The range of each segment on the horizontal axis is determined by $V_{\min}$, $V_{\max}$, and V_i $(i \in \{1, 2, 3\})$. Since V_i is an average *value*, we divide C_i into two halves and distribute them to adjacent segments.

mReqSketch: As discussed in Section II-B3, ReqSketch consists of several levels of $compactor$ serving as buffers. Each level comprises multiple cells storing the *value* V_i of items. Each cell in level i carries a weight $C_i = 2^i$ $(i \in \{0, 1, 2, \ldots\})$. Besides, ReqSketch also records the minimum *value* $V_{\min}$ and the maximum *value* $V_{\max}$.

The original design of $compaction\ operation$ in ReqSketch is memory-intensive and slow, making it unsuitable for estimating per-flow *value* distribution in data streams. We introduce minor modifications to the original design while maintaining its quintessence and rename it mReqSketch. According to the new design, the incoming items are always inserted into level 0. Whenever a level h becomes full, we sort the items in level

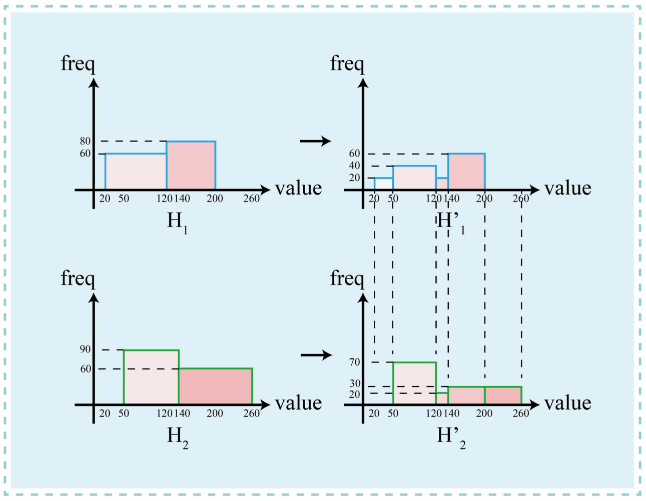


Fig. 4. Segment alignment.

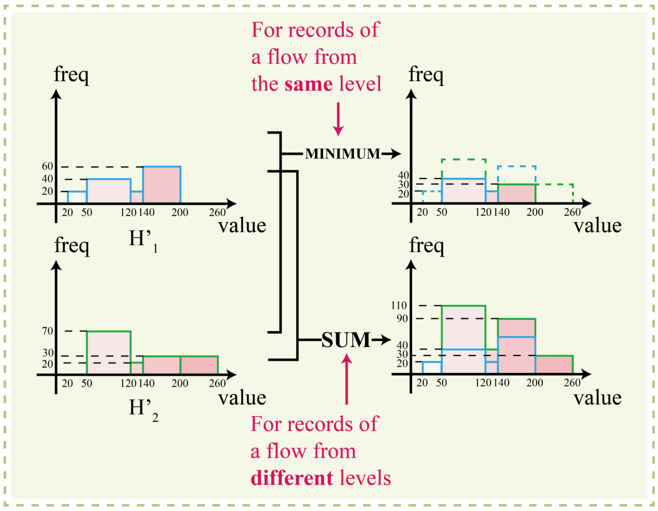

For records of a flow from the **same** level
MINIMUM
SUM
For records of a flow from **different** levels
freq
value
H'_1
H'_2

Fig. 5. *MINIMUM* & *SUM* on aligned Histograms.

h, remove them from this level, and randomly select half of the items (either odd or even indexed) to be inserted into level $h+1$.

The process of transforming a mReqSketch into a histogram is illustrated on the right side of Fig. 3. The range of each segment on the horizontal axis is determined by $V_{\min}$, $V_{\max}$, and V_i $(i \in \{1,2,3,4,5\})$. Since V_i is a randomly selected *value*, we divide C_i into two halves and distribute them to adjacent segments.

3) MINIMUM & SUM on Histogram: After transforming *META* into histograms, we can perform the *MINIMUM* and *SUM* operations. These operations necessitate a prerequisite known as *Segment Alignment*. As illustrated in Fig. 4, suppose we have two histograms H_1 and H_2, which require alignment. We first need to obtain the union boundary set on the horizontal axis. The boundary set in H_1 is $S_1 = \{20, 120, 200\}$, and that of H_2 is $S_2 = \{50, 140, 260\}$. Thus, the union boundary set is $U = \{20, 50, 120, 140, 200, 260\}$. Next, for both histograms, we scatter the frequency recorded in each segment determined by S_i into the new segments determined by U, following a uniform distribution. The rationale here is that focusing on a minimal interval allows us to approximate any distribution with a uniform one, mirroring the central concept in calculus.

After *Segment Alignment*, we can operate on the frequency in each segment of H'_1 and H'_2. As shown in Fig. 5, if H'_1 and H'_2 originate from the same level of the same flow, we will need to *MINIMUM*-merge them. We select the minimum frequency in each segment to construct the *MINIMUM*-merged distribution. If H'_1 and H'_2 originate from different levels of the same flow, we will need to *SUM*-merge them. We add the frequencies in each segment to construct the *SUM*-merged distribution.

To query for the *value* at percentage $p \in [0,1]$, we accumulate segment frequencies until finding the segment that p falls in. Then we calculate a quantile according to the uniform distribution and report it as the estimation result.

D. Discussion

1) Using Histogram as an Intermediate Step: A critical aspect of developing a universally applicable framework is ensuring the adaptability of core operations, such as *MINIMUM* and *SUM*, to accommodate a wide range of single-flow algorithms. The inherent challenge lies in the diversity of distribution outputs that quantile estimation algorithms can produce. Our solution is normalizing these distributions into histograms, which serve as a standardized intermediary representation. It simplifies the implementation of the *MINIMUM* and *SUM* operations by reducing them to operations on histogram bins, thereby enhancing the framework's efficiency and scalability.

This design choice underscores the flexibility of the M4 framework, allowing it to seamlessly incorporate a vast array of quantile estimation algorithms without necessitating extensive customization or reconfiguration. This adaptability is crucial for a general framework aimed at broad applicability across diverse network environments and applications.

2) Data Aging: One potential area for improvement in our framework is enhancing the data aging process to better align with real-time data distribution changes. Currently, the mechanism for phasing out outdated data is clearing the data structure for every time window. It may not be sufficiently prompt, creating a gap between the stored data distribution and the actual current distribution. This discrepancy becomes particularly problematic when data distribution shifts rapidly, potentially compromising the method's effectiveness. Improving the data aging process to more accurately reflect immediate distribution changes is essential for maintaining M4's accuracy in dynamic data environments.

3) Flow Size Distribution: Our methodology, including the underlying data structures and operations, is specifically optimized for data streams characterized by a long-tailed distribution of flow sizes, a common phenomenon in real-world applications where the principle of *the few governing the many* often applies. This long-tailed distribution pattern ensures that a small fraction of flows carries a significant portion of data.

It is important to note, however, that in environments where the flow size distribution is uniform, our approach may not be the most efficient. We acknowledge that this specificity may limit the universal applicability of our approach but we believe that its optimized performance in its intended context represents a significant contribution to the field.

IV. Mathematical Analysis

There are two sources of error for M4: (1) The accuracy of the recording algorithm in each bucket and (2) hash collision. The error introduced by (1) is due to inaccuracies in *META*.

We will present an error bound for mReqSketch in this section as we modified the original design of ReqSketch. As for (2), which is hash collision related, we will present an error bound for M4-DDSketch. The analysis is conducted at one specific level at once, assuming that there are n buckets and m flows arriving at the level we are examining, with each flow mapped to w buckets.

A. Analysis of M4-DDSketch

We begin by defining some frequently used notations. $freq$ represents the total number of items at a level. $freq_i$ represents the number of items in flow i at this level. $freq_i^T$ represents the number of items in flow i within a segment T at this level. $freq^T$ represents the total number of items within the segment T at this level.

1) Huge and Medium Flows: Theorem 1: Let $\hat{freq_i}$ denote the estimation of $freq_i$. Then

$$P\left(\hat{freq_i} > freq_i + \epsilon\right) < (\frac{freq}{n\epsilon})^w \tag{1}$$

Proof: Let $\hat{freq}_{i,k}$ $(k \in \{1, 2, \ldots, w\})$ denote the k^{th} record of $freq_i$. They are independent from each other because corresponding hash functions are independent. Since $\hat{freq_i} = \min\{\hat{freq}_{i,k}\}$, we obtain that

$$P(\hat{freq_i} > freq_i + \epsilon) = P(\hat{freq}_{i,k} > freq_i + \epsilon)^w \tag{2}$$

Now we analyze the situation where $w = 1$. Let A_j denote the event that flow j $(j \neq i)$ is mapped to the same bucket as flow i. Then we have $\hat{freq}_{i,1} = freq_i + \sum_{j\neq i} freq_j 1_{A_j}$ (1_{A_j} is the characteristic function of the event A_j) and $P(A_j) = \frac{1}{n}$. Hence, $E(\hat{freq}_{i,1} - freq_i) = \sum_{j\neq i} freq_j \frac{1}{n} < \frac{freq}{n}$. Because $\hat{freq}_{i,1} - freq_i \geq 0$, we use the Markov inequality to obtain

$$P(\hat{freq}_{i,1} - freq_i > \epsilon) \leq \frac{E(\hat{freq}_{i,1} - freq_i)}{\epsilon} < \frac{freq}{n\epsilon} \tag{3}$$

Therefore,

$$\begin{aligned} P(\hat{freq_i} > freq_i + \epsilon) &= P(\hat{freq}_{i,1} - freq_i > \epsilon)^w \\ &< \left(\frac{freq}{n\epsilon}\right)^w \end{aligned} \tag{4}$$

□

Let $\hat{freq}_i^T$ denote the estimation of $freq_i^T$. Then we similarly have[5]

$$P(\hat{freq}_i^T > freq_i^T + \epsilon) < (\frac{freq^T}{n\epsilon})^w. \tag{5}$$

Theorem 2. Let $\hat{t_p}$ denote the estimated quantile of percentage p. Then

$$P(|\hat{t_p} - t_p| < \alpha t_p) \geq 1 - (1 - e^{-\frac{m}{n}})^w \tag{6}$$

Proof: The probability of hash collision happening in all mapped buckets of a flow is $P_C = [1 - (\frac{n-1}{n})^m]^w \approx (1 - e^{-\frac{m}{n}})^w$. So the probability that there is at least one bucket where no hash collision occurs is $1 - P_C = 1 - (1 - e^{-\frac{m}{n}})^w$. In this case, the error is bounded by $|\hat{t_p} - t_p| < \alpha t_p$, as proved in DDSketch. Therefore, $P(|\hat{t_p} - t_p| < \alpha t_p) \geq 1 - (1 - e^{-\frac{m}{n}})^w$ □

Hash collisions may have a significant impact on quantile t_p. Unless strong assumptions are made on the *value* distributions of flows, giving an error bound of $\hat{t_p}$ is impossible when hash collisions happen in all mapped buckets.

Comparison with prior work: We choose to compare M4-DDSketch with SketchPolymer [48], a per-flow quantile estimation algorithm which has a similar algorithm principle with M4-DDSketch. The main difference between M4-DDSketch and SketchPolymer is that M4-DDSketch uses multiple levels for huge and medium flows, while SketchPolymer only uses one level/layer after the filtration of tiny flows. When the memory is limited, our algorithm is usually more accurate for huge flows. The reason is that by using less bits storing medium flows, we can allocate more memory for huge flows to avoid hash collisions.

First, let us consider M4-DDSketch. We divide the flows to medium and huge flows. The DDSketch for medium and huge flows has the same α[6], but we use fewer bits for the counters of medium flows. We denote the memory cost of one bucket for medium and huge flows by b_1 and b_2, respectively. We set $w = 1$ and the amount of buckets for medium and huge flows to be n_1 and n_2. We denote the total memory by C. Then, we have

$$n_1b_1 + n_2b_2 = C \tag{7}$$

We can choose the proportion of n_1 and n_2 to make the expected value of hash collision probability (denoted by $E(\text{HC})$) the same for medium and huge levels. For a huge flow i, the error of its frequency (sum of medium and huge parts) $freq_i$ has the expectation value:

$$E(\hat{freq_i} - freq_i) = E(\text{HC})F, \tag{8}$$

where F is the total number of items. We denote the number of medium flows by X_1 and the number of huge flows by X_2, then we have

$$\begin{aligned} X_1 + X_2 &= E(\text{HC})n_1 \\ X_2 &= E(\text{HC})n_2 \end{aligned} \tag{9}$$

Combining (7), (8) and (9), we get

$$E(\hat{freq_i} - freq_i) = \frac{F}{C}(X_1b_1 + X_2b_1 + X_2b_2) \tag{10}$$

Therefore, we have the error bound

$$\begin{aligned} P(\hat{freq_i} - freq_i > \epsilon) &\leq \frac{E(\hat{freq_i} - freq_i)}{\epsilon} \\ &= \frac{F}{C\epsilon}(X_1b_1 + X_2b_1 + X_2b_2) \end{aligned} \tag{11}$$

Next, let us consider SketchPolymer. We have total memory$=$ C, bucket size$=$ b_2 (all the buckets should be of the size for huge flows), X_1 medium flows and X_2 huge flows. The corresponding error bound is

$$P(\hat{freq}'_i - freq_i > \epsilon) \leq \frac{F}{C\epsilon}(X_1b_2 + X_2b_2) \tag{12}$$

[5]Strictly speaking, bucket collapsing in DDSketch(C) algorithm will influence $\hat{freq}_i^T$ but only buckets with shortest latency. Since we focus on buckets with long latency, we ignore this effect.

[6]For the definition of α, please refer to Section II-B1

The difference of the error bound is

$$\frac{F}{C\epsilon}(X_2 b_1 - X_1(b_2 - b_1)) \tag{13}$$

In real situations, X_1 is much larger than X_2. So the error bound of M4-DDSketch is smaller than that of SketchPolymer.

2) Tiny Flows: Theorem 3: Let $x = \frac{wm}{n}$, then the probability of getting a wrong maximum *value* is $(\frac{e^{-x}+x-1}{x})^w$.

Proof: Suppose that a flow f is mapped to w buckets $a_1, a_2, \ldots, a_w$. The probability that there are k_i other flows in bucket a_i $(i \in \{1, 2, \ldots, w\})$ is

$$P(\{k_i\}) = \frac{\binom{wm-w}{k_1}\binom{wm-w-k_1}{k_2}\cdots\binom{wm-w-k_1-k_2-\ldots-k_{w-1}}{k_w}}{n^{wm-w}} \cdot (n-w)^{wm-k_1-k_2-\ldots-k_w-w}. \tag{14}$$

The equation above is based on the assumption that the w hash values of a flow are independent. Due to the factor $(n-w)^{-k_1-k_2-\ldots-k_w}$, the probability of k_i being large is low. So we can assume that $k_i << m$ and arrive at the approximation $P(\{k_i\}) \approx [\prod_i \binom{wm-w}{k_i}(n-w)^{-k_i}](1-\frac{w}{n})^{wm-w} \approx [\prod_i \binom{wm}{k_i} n^{-k_i}]e^{-\frac{w^2 m}{n}}$. In this situation, the probability of obtaining an incorrect maximum *value* is $\prod_i \frac{k_i}{k_i+1}$. Hence, the probability of obtaining an incorrect maximum *value* is

$$\begin{aligned}
&\left[\prod_i \sum_{k_i=1}^{wm} \binom{wm}{k_i} n^{-k_i} \frac{k_i}{k_i+1}\right] e^{-\frac{w^2 m}{n}} \\
&= \left[\sum_{k=1}^{wm} \binom{wm}{k} n^{-k} \frac{k}{k+1}\right]^w e^{-\frac{w^2 m}{n}} \\
&= \left[\frac{n - \left(1+\frac{1}{n}\right)^{wm}(-wm+n)}{wm+1} e^{-\frac{wm}{n}}\right]^w \\
&\overset{x=\frac{wm}{n}}{\approx} \left(\frac{e^{-x}+x-1}{x}\right)^w
\end{aligned} \tag{15}$$

□

B. Analysis of mReqSketch

In this section, we conduct the error analysis for mReqSketch. $R(y)$ represents the count of *values* that is less than or equal to *value* y across all items recorded at a level. $R_i(y)$ represents the count of *value* that is less than or equal to *value* y in flow i at this level.

Error bound of mReqSketch: Consider the following setting. There are M compactors $C_0, C_2, \ldots, C_{M-1}$ in the mReqSketch. The buffer size in each compactor is $b = 2^a$. The number of items in this mReqSketch is at maximum capacity $N = 2^a(2^M - 1) \approx 2^{a+M}$. Let p denote the real value of the fraction of *values* less than y. Consider the estimation for $R(y)$.

Theorem 4: Let $\hat{R(y)}$ denote the estimation of $R(y)$. Then

$$P(|\hat{R}(y) - R(y)| > \epsilon N) < 2e^{-\frac{4b^2}{1-(1-2p)^b}\epsilon^2} \tag{16}$$

Each time we conduct the $compaction\ operation$, we operate on b *values*. Among these, the probability that the number of *value* less than y is odd is given by

$$\begin{aligned}
P &= \frac{\sum_{i=1}^{b}[\binom{b}{i}p^i(1-p)^{b-i}] - \sum_{i=1}^{b}[\binom{b}{i}(-p)^i(1-p)^{b-i}]}{2} \\
&= \frac{1-(1-2p)^b}{2}
\end{aligned} \tag{17}$$

We must perform the $compaction\ operation$ on C_0 for 2^M times. If the selected items have odd indices, the probability of having an error of $+1$ on $R(y)$ is P, and the probability of error-free is $(1-P)$. If items with even indices are selected, the probability of having an error of -1 on $R(y)$ is P, and the probability of error-free is $(1-P)$. Therefore, the expected error is 0 each time, and the error variance is $P(1-P)$. The overall expected error is 0, and the overall error variance is $2^M P(1-P)$.

We need to perform the $compaction\ operation$ on C_1 for 2^{M-1} times. If items with odd indices are selected, the probability of having an error of $+2$ on $R(y)$ is P, and the probability of error-free is $(1-P)$. If items with even indices are selected, the probability of having an error of -2 on $R(y)$ is P, and the probability of error-free is $(1-P)$. Therefore, the expected error is 0 each time, and the error variance is $4P(1-P)$. The overall expected error is 0, and the overall error variance is $2^{M+1}P(1-P)$.

Performing the above analysis for all the compactors, we find that the overall expected error in $C_i (i \in \{0, 1, 2, \ldots, M-1\})$ is 0. The overall variance of the error in C_i is $2^{M+i}P(1-P)$. Therefore, the variance of the error across the whole mReqSketch is

$$\begin{aligned}
\sigma^2 &= P(1-P)(2^M + 2^{M+1} + \ldots + 2^{2M-1}) \\
&= P(1-P)2^M(2^M - 1) \\
&\approx \frac{1-(1-2p)^{2b}}{4} 2^{2M} \\
&\approx \frac{1-(1-2p)^b}{4}\frac{N^2}{b^2}
\end{aligned} \tag{18}$$

Note that the variance from C_i increases with i, so the Lindeberg-Feller condition is not satisfied. We cannot apply the central limit theorem. However, we can employ the sub-Gaussian distribution estimation and find

$$P(|\hat{R}(y) - R(y)| > \epsilon N) < 2e^{-\left(\frac{\epsilon N}{\sigma}\right)^2} = 2e^{-\frac{4b^2}{1-(1-2p)^b}\epsilon^2} \tag{19}$$

V. Experimental Results

A. Experiment Setup

1) Implementation: We implement M4 and all related *META* data structures (DDSketch, t-digest, and mReqSketch) in C++. The hash functions used uniformly are the 32-bit Bob Hash (sourced from an open-source website [59]) with different initial seeds. All the experiments are executed on an 18-core CPU server (Intel i9-10980XE) with 128 GB memory and 24.75 MB L3 cache. Each experiment is repeated ten times to compute an average result.

2) Straw-Man Solution: To compare effectively, we develop a straw-man solution using a Dleft-like approach for each *META* (DDSketch, t-digest, and mReqSketch), featuring three bucket arrays for an optimal balance between accuracy and speed. Each bucket records a flow ID key and the distribution of *values* using a *META*. We use three distinct hash functions, $h_1'(\cdot)$, $h_2'(\cdot)$, and $h_3'(\cdot)$, for the arrays. Additionally, a global *META* aggregates the value distribution for all flows, serving as a fallback for queries on unrecorded flows.

Insertion: When a new item $e = \langle f, v \rangle$ arrives, it's first added to the global *META*. We then try to insert e to the $h'_1(f)^{th}$ bucket of the first array. If this bucket is free or already contains a flow with $key = f$, e is added to this bucket's META, setting key to f. Failing that, we move to the $h'_2(f)^{th}$ bucket of the second array, and if necessary, to the $h'_3(f)^{th}$ bucket of the third array. Should all arrays reject e, it is discarded.

Query: To find a flow f, we check the $h'_1(f)^{th}$, $h'_2(f)^{th}$, and $h'_3(f)^{th}$ bucket of the respective arrays for a $key = f$. A match returns the query from that bucket's *META*; otherwise, the global *META* provides the result.

3) Cuckoo Filter: We devise another comparison framework based on Cuckoo Filter [60]. Cuckoo Filter is an efficient hash table implementation based on cuckoo hashing [61], which can achieve both high utilization and compactness. It records the fingerprint instead of the flow ID to improve space efficiency.

The structure employs a table with buckets and three hash functions, $h_1(\cdot)$, $h_2(\cdot)$, and $h_f(\cdot)$, where each bucket records a fingerprint fp^c and the distribution of *values* using a *META*. For an item $e = \langle f, v \rangle$, we determine $fp = h_f(f)$ and map e to the $[h_1(fp)]^{th}$ and $[h_2(fp)]^{th}$ buckets. We then try to locate one of these two buckets where $fp^c = fp$ and insert v to the *META* in that bucket. If no matching bucket is found but an empty bucket exists, we insert e to it by setting its $fp^c = fp$ and inserting v to the *META*. If both buckets are full, then one of the two flows in them will be evicted to its alternate bucket (because each flow has two mapped buckets). This random eviction continues until an empty bucket is found or a preset maximum number of attempts, $MAX\_NUMBER\_OF\_TURNS$, is reached. If that happens, we discard this item. Like what we do in the straw-man solution, a global *META* is maintained as a fallback for queries. We choose $MAX\_NUMBER\_OF\_TURNS = 8$ and 32-bit fp to minimize collisions, as increasing $MAX\_NUMBER\_OF\_TURNS$ beyond this point does not significantly enhance accuracy but does reduce throughput.

4) Datasets:

1) *CAIDA Dataset:* This dataset comprises streams of anonymized IP items collected from high-speed monitors by CAIDA in 2018 [49]. We use the trace with a monitoring interval of 60 s. Each item consists of a 5-tuple (13 bytes). There are around 27 M items and 1.3 M flows in this dataset.
2) *MAWI Dataset:* This dataset contains real traffic trace data maintained by the MAWI Working Group [62]. Similar to CAIDA, each item in the dataset is a 5-tuple. There are around 9 M items and 13 K flows in the MAWI dataset.
3) *IMC Dataset:* This dataset comes from one of the data centers studied in [63]. Each item also consists of a 5-tuple. There are around 14 M items and 5 K flows in this dataset.
4) *Web Latency Dataset:* The Web Latency Dataset is collected by Webget via 182 probes distributed around the world [64]. We consider the fetch time of each request as value.
5) *Seattle Dataset:* The Seattle Dataset consists of round trip times (RTTs) between many nodes [22] in the constructed network [65]. We consider RTTs between the same two nodes as the same item and the RTT as its value. Note that this dataset has no tiny flows and the experiments in Section V-D will not be run on it.

5) Metrics:

1) *ALE (Average Logarithm Error):* We employ *ALE* to evaluate the accuracy of quantile estimation for huge and medium flows. Since the order magnitude of latency may vary significantly, it is unreasonable to measure the error by absolute value alone. We define *ALE* as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}|log_2 t_i - log_2\hat{t_i}| = \frac{1}{|\Psi|}\sum_{f_i\in\Psi}|log_2\frac{t_i}{\hat{t_i}}|$, where Ψ represents the set of all huge and medium flows in the data stream. t_i and $\hat{t_i}$ denote the real and estimated quantile at a given percentage p.
2) *APE (Average Percentage Error):* We employ *APE* to evaluate the accuracy of quantile estimation for huge and medium flows. We define *APE* as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}|p - \hat{p_i}|$, where Ψ represents the set of all the huge and medium flows in the data stream. Here, p is a given percentage we need to query, returning a value t. Then, we query t in the real distribution to get the percentage $\hat{p}$. The difference between p and $\hat{p}$ represents the error of our query.
3) *AAE (Average Absolute Error):* We employ *AAE* to evaluate the accuracy of quantile estimation for large and medium flows from another perspective. We define *AAE* as $\frac{1}{|\Psi|}\sum_{f_i\in\Psi}|t - \hat{t_i}|$, where Ψ, t_i, and $\hat{t_i}$ are consistent with the above ALE's.
4) *RE (Relative Error):* We employ *RE* to evaluate the accuracy of quantile estimation for tiny flows. We define *RE* as $\frac{|\hat{x}_{max} - x_{max}|}{x_{max}}$, where x_{max} and $\hat{x}_{max}$ are the real and estimated maximum *value* in a flow.
5) *Throughput (Insertion and Query):* We use million operations (insert an item or query a flow) per second (Mops) to measure the throughput.

TABLE II
DEFAULT PARAMETER SETTINGS

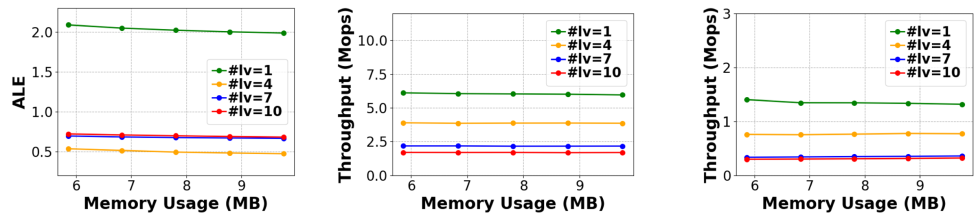
Algorithm	c			l		
	c_2	c_3	c_4	l_2	l_3	l_4
M4-DDSketch	20	20	35	8	16	32
M4-t-digest	4	8	16	8	16	32
M4-mReqSketch	2	2	4	8	16	32
Strawman-DDSketch	35			32		
Strawman-t-digest	16			32		
Strawman-mReqSketch	4			32		
CuckooFilter-DDSketch	68			32		
CuckooFilter-t-digest	32			32		
CuckooFilter-mReqSketch	8			32		

6) Default Settings: In our experiment, we set $p = 0.5$ as the default setting. For M4, the memory ratio of L_1, L_2, L_3, L_4 is 3%, 60%, 35%, 2%, respectively. Each arriving item at every level is mapped to 3 buckets ($w_i = 3, i \in \{1,2,3,4\}$). Each bucket in L_1 has 4 counter cells ($c_1 = 4$) and 1 MX cell, and each counter cell consists of 2 bits ($l_1 = 2$). The parameter settings for DDSketch, t-digest, and mReqSketch on different frameworks are shown in Table II.

B. Experiments on Parameter Setting

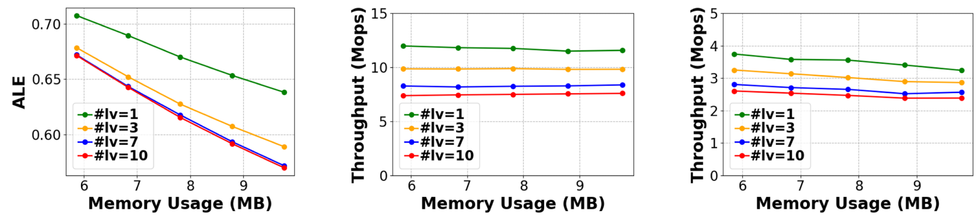
Effect of # levels: We conduct experiments on the choice of the number of levels for M4 (represented by M4-DDSketch) and the straw-man solution. As shown in Fig. 6, choosing $\# \ lv = 4$ gives the best accuracy and a comparatively good speed for M4, so we design the M4 to have 4 levels. As shown in Fig. 7, choosing $\# \ lv = 3$ gives a close to the best accuracy and a comparatively good speed for the straw-man solution, so we design the straw-man solution to have 3 levels.

Effect of the hash number w: We conduct w tuning experiments on different M4-*META*s on the CAIDA dataset. As



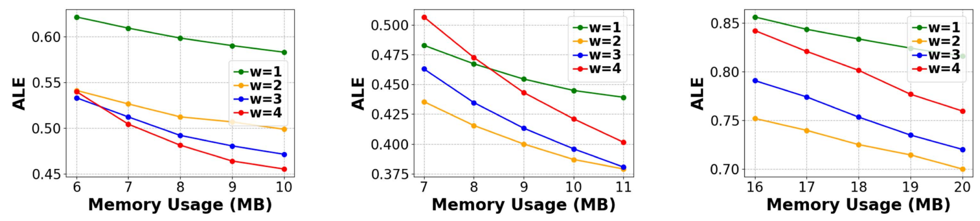
(a) ALE (b) Insertion throughput (c) Query throughput

Fig. 6. Effect of the number of levels on M4.



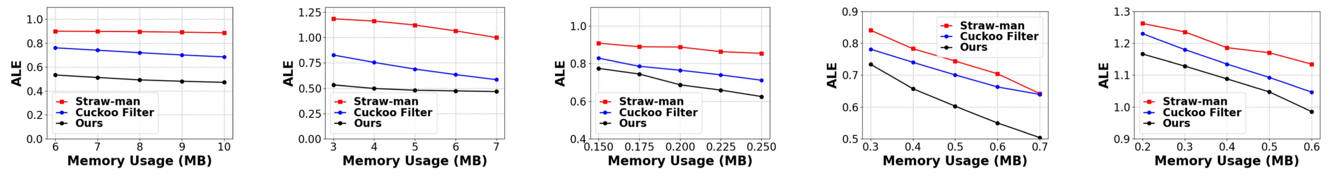
(a) ALE (b) Insertion throughput (c) Query throughput

Fig. 7. Effect of the number of levels on Straw-man.



(a) M4-DDSketch (b) M4-t-digest (c) M4-mReqSketch

Fig. 8. Effect of w on accuracy.



(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

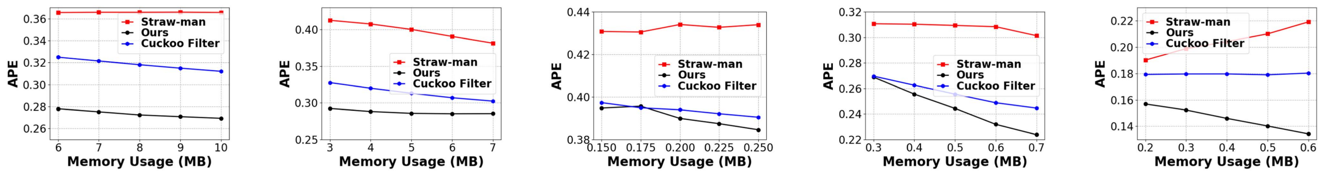
Fig. 9. ALE of M4-DDSketch.

shown in Fig. 8, we find that the three versions of M4 all reach sub-optimal ALE and the gap between it and the optimal is small at $w = 3$, so the overall performance of M4 at $w = 3$ is the best among different w values. In light of this, we select $w = 3$ as our default setting.

C. Experiments of Huge and Medium Flows on Accuracy

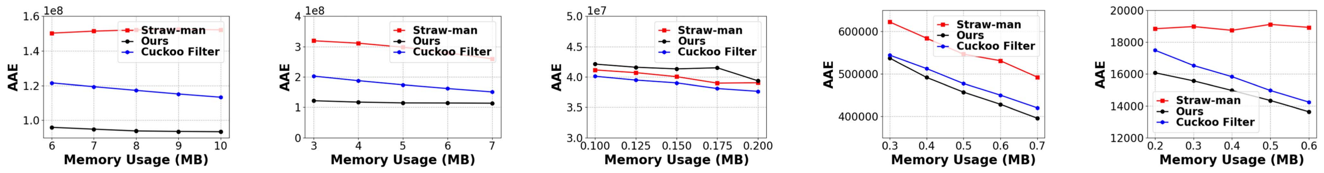
Experiments on M4-DDSketch: As shown in Figs. 9, 10, and 11, the results show that the ALEs (APEs) [AAEs] of M4-DDSketch are lower than those of the comparison frameworks. Specifically on the five real-world datasets, the ALEs (APEs) [AAEs] of M4-DDSketch are on average 1.80× (1.34×) [1.61×], 2.26× (1.39×) [2.53×], 1.27× (1.11×) [0.97×], 1.22× (1.26×) [1.21×], and 1.11× (1.40×) [1.27×] lower than those of the straw-man solution, and 1.45× (1.17×) [1.24×], 1.42× (1.09×) [1.51×], 1.10× (1.01×) [0.94×], 1.16× (1.05×) [1.04×], and 1.05× (1.23×) [1.06×] lower than those of Cuckoo Filter.

Experiments on M4-DDSketch(C): As shown in Figs. 12, 13, and 14, the results show that the ALEs (APEs) [AAEs] of M4-DDSketch(C) are always lower than those of the comparison frameworks. Specifically on the five real-world datasets, the ALEs (APEs) [AAEs] of M4-DDSketch are on average 1.71× (1.36×) [2.19×], 1.51× (1.23×) [1.88×], 1.66× (1.16×) [1.88×], 1.22× (1.20×) [1.47×], and 1.12×



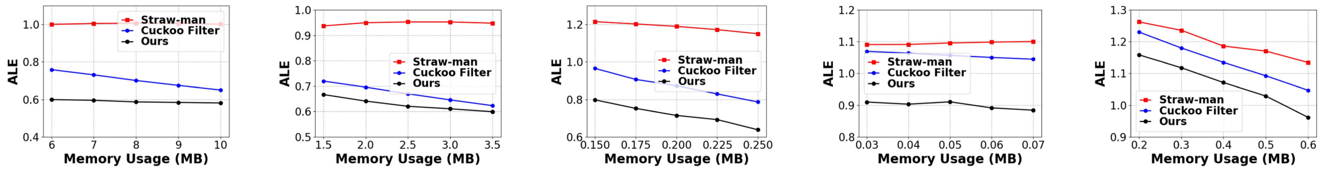
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 10. APE of M4-DDSketch.



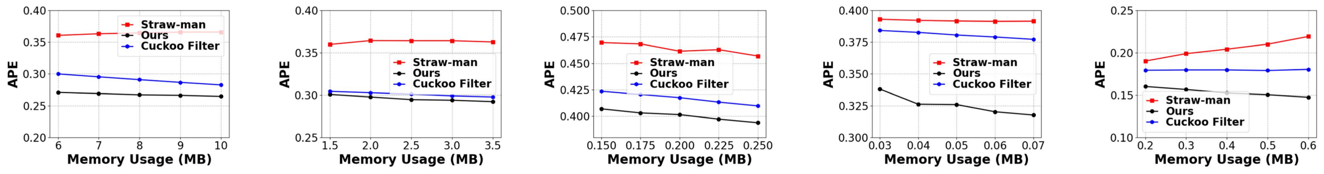
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 11. AAE of M4-DDSketch.



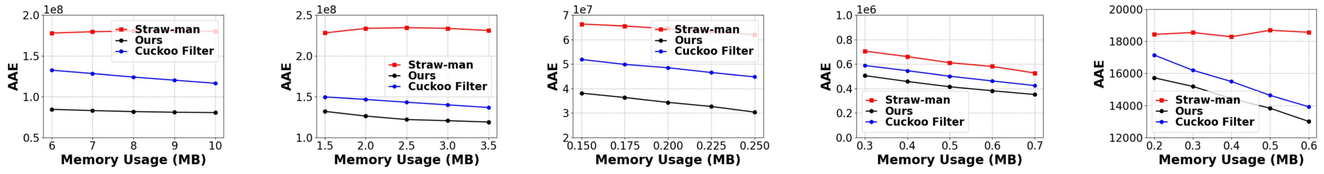
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 12. ALE of M4-DDSketch(C).



(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 13. APE of M4-DDSketch(C).



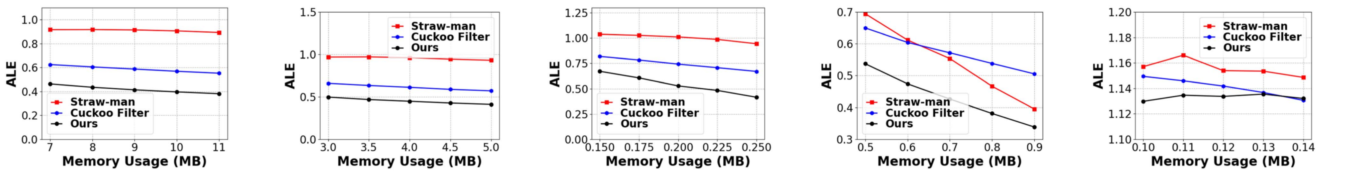
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 14. AAE of M4-DDSketch(C).

(1.34×) [1.29×] lower than those of the straw-man solution, and 1.19× (1.09×) [1.51×], 1.07× (1.02×) [1.16×], 1.21× (1.04×) [1.41×], 1.17× (1.17×) [1.20×], and 1.07× (1.17×) [1.07×] lower than those of Cuckoo Filter.

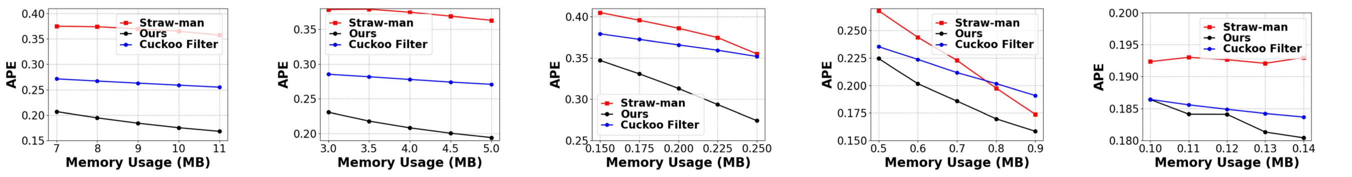
Experiments on M4-t-digest: As shown in Figs. 15, 16, and 17, the results demonstrate that the ALEs (APEs) [AAEs] of M4-t-digest are always lower than those of the comparison frameworks. Specifically on the five real-world datasets, the ALEs (APEs) [AAEs] of M4-t-digest are on average 2.19× (1.99×) [2.89×], 2.13× (1.77×) [2.74×], 1.90× (1.23×) [2.13×], 1.26× (1.18×) [1.37×], and 1.02× (1.05×) [1.07×] lower than those of the straw-man solution, and 1.41× (1.42×) [1.71×], 1.36× (1.32×) [1.65×], 1.40× (1.17×) [1.72×], 1.33× (1.13×) [1.22×], and 1.01× (1.01×) [1.02×] lower than those of Cuckoo Filter.

Experiments on M4-mReqSketch: As shown in Figs. 18, 19, and 20, the results demonstrate that the ALEs (APEs) [AAEs] of M4-mReqSketch are lower than those of the comparison frameworks. Specifically on the five real-world datasets, the ALEs (APEs) [AAEs] of M4-mReqSketch are on average 1.94×



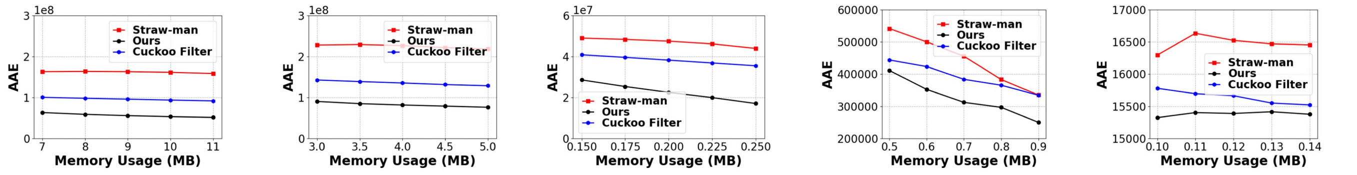
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 15. ALE of M4-t-digest.



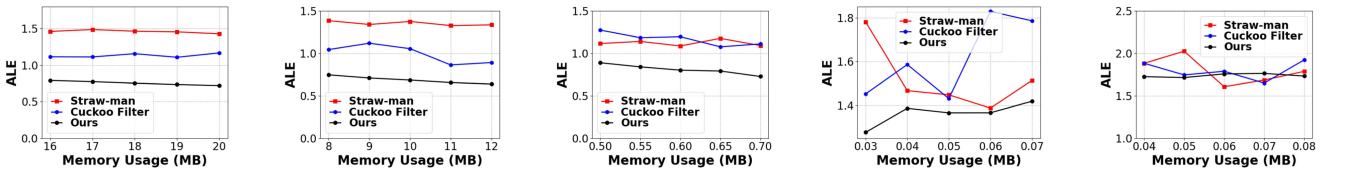
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 16. APE of M4-t-digest.



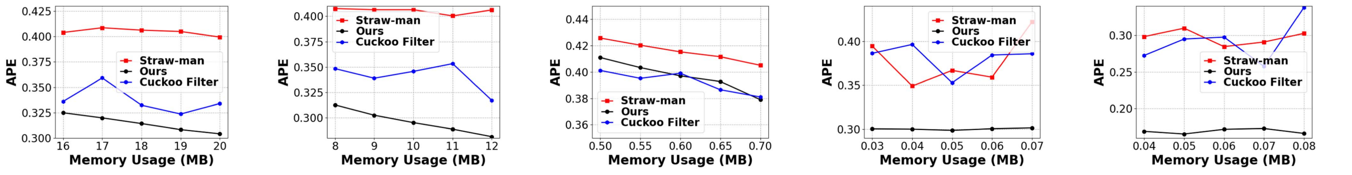
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 17. AAE of M4-t-digest.



(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 18. ALE of M4-mReqSketch.

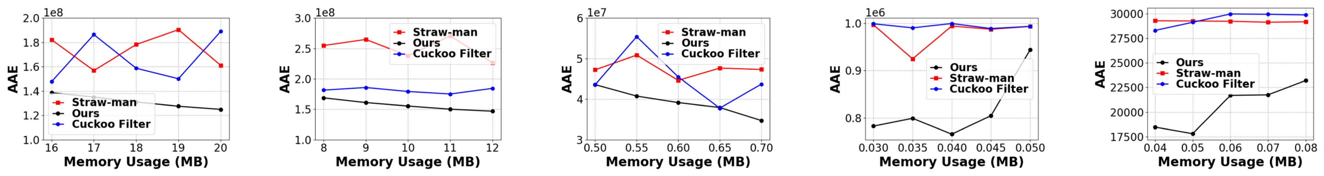


(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 19. APE of M4-mReqSketch.

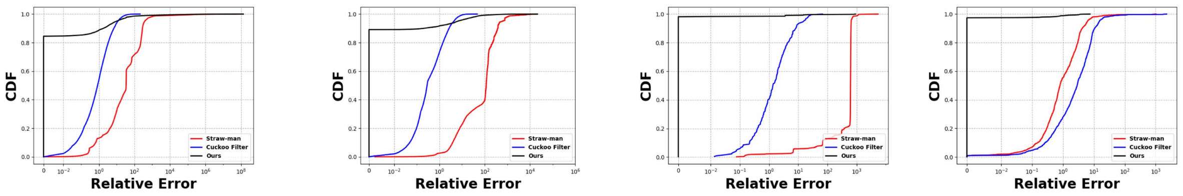
(1.29×) [1.32×], 1.97× (1.37×) [1.60×], 1.39× (1.05×) [1.22×], 1.11× (1.26×) [1.20×], and 1.03× (1.76×) [1.43×] lower than those of the straw-man solution, and 1.50× (1.07×) [1.27×], 1.45× (1.15×) [1.28×], 1.44× (0.99×) [1.16×], 1.19× (1.27×) [1.22×], and 1.03× (1.73×) [1.44×] lower than those of Cuckoo Filter.

Analysis: The accuracy advantage in Figs. 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20 is established by making better



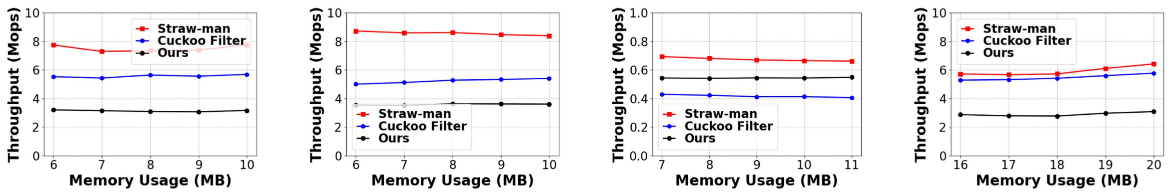
(a) CAIDA (b) MAWI (c) IMC (d) Web (e) Seattle

Fig. 20. AAE of M4-mReqSketch.



(a) CAIDA (b) MAWI (c) IMC (d) Web

Fig. 21. RE distribution of tiny flows on different datasets.



(a) DDSketch (b) DDSketch(C) (c) t-digest (d) mReqSketch

Fig. 22. Insertion throughput on the CAIDA dataset.

use of resources. First, the separation of medium and huge flows brought by the *SUM* technique allows for allocating fewer bits for medium flows. Furthermore, the multi-layer structure prevents huge and medium flows from contaminating each other. Otherwise, once a medium flow collides with a huge one, its distribution will be utterly covered up by the huge flow because of its size. Second, the *MINIMUM* technique improves the algorithm's robustness against hash collisions.

D. Experiments of Tiny Flows on Accuracy

As shown in Fig. 21, the experimental results demonstrate that for most tiny flows, the maximum *value* can be well estimated by M4, which significantly outperforms the two comparison frameworks. Specifically on the four real-world datasets (except Seattle), M4 attains an error-free (i.e., $RE = 0$) rate of 84.5%, 89.1%, 98.2%, and 97.4%, while the comparison frameworks offer almost no error-free estimates.

Analysis: Tiny flows are too small to well-define a distribution. Using *METAs* to record tiny flows would be inaccurate and memory-consuming. Hence, only recording the maximum *value* gives us significantly better results than two comparison frameworks.

E. Experiments on Speed

We conduct experiments on the speed (insertion and query throughput) of different M4-*METAs* and two comparison frameworks on the CAIDA dataset.

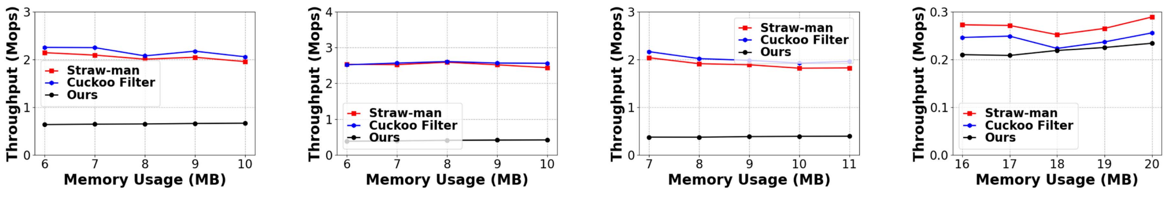
As shown in Figs. 22 and 23, the results demonstrate that the insertion (query) throughput of M4-*METAs* is slower than or at the same level with respect to the comparison frameworks. Specifically on the CAIDA dataset, the insertion (query) throughput of M4-(DDSketch, DDSketch(C), t-digest, and mReqSketch) is on average $2.40\times$ $(3.15\times)$, $2.39\times$ $(6.28\times)$, $1.24\times$ $(4.98\times)$, and $2.05\times$ $(1.24\times)$ lower than those of the straw-man solution, and $1.78\times$ $(3.33\times)$, $1.46\times$ $(6.38\times)$, $0.77\times$ $(5.28\times)$ and $1.89\times$ $(1.09\times)$ lower than those of Cuckoo Filter.

Analysis: The experiments show that the throughput of M4 in insertion and query is slightly lower than that of the comparison frameworks. This is because our solution contains more layers. Besides, *MINIMUM* and *SUM* operations take extra time.

F. Evaluation on Tofino Platform

We fully build a P4 prototype of M4 on the Tofino switch [66] using P4 language.

Challenges and Design: The architectural design is shown in Fig. 24. Since the P4 pipeline can only access each register



(a) DDSketch (b) DDSketch(C) (c) *t*-digest (d) mReqSketch

Fig. 23. Query throughput on the CAIDA dataset.

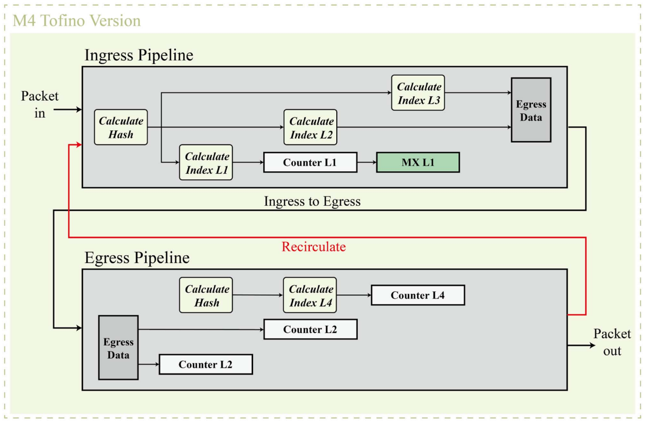


Fig. 24. Architecture design of the Tofino version of M4.

TABLE III
H/W RESOURCES USED BY M4

Resource	Usage	Percentage
Hash Bits	432	8.65%
SRAM	32	3.33%
Map RAM	30	5.21%
TCAM	2	0.69%
Stateful ALU	15	31.25%
VLIW Instr	27	7.03%
Exact Xbar	79	5.14%
Ternary Xbar	4	0.51%

once, the Tofino version of M4 uses recirculate, $i.e.$, each packet enters the pipeline twice. When it enters the pipeline for the first time, each layer is checked for overflow and the final layer to be inserted is calculated, while the insertion is completed the second time it enters the pipeline. Of course, this results in half the efficiency of the switch. We add a custom header to the measured packets, including a 32-bit flow ID and 16-bit delay. The second to fourth level *META* uses DDSketch, and its base is 2. For simplicity, the number of segments of DDSketch at each level is the same, which is 16 segments: segment 1 corresponds to $[0, 2)$, segment i corresponds to $[2^{i-1}, 2^i), i = 2, \ldots, 16$. We directly use longest prefix matching (LPM) on 16-bit delay to complete the segment number query.

Results: We list the utilization of various hardware resources on the Tofino switch in Table III. We find that Stateful ALU is the most used resources of M4, accounting for 31.25% of the total quota, while Map RAM accounts for 5.21% .

VI. CONCLUSION

This paper introduces the M4 framework designed to enable per-flow quantile estimation using single-flow estimation algorithms. The key techniques of M4 are *MINIMUM*, employed for minimization of the noise caused by hash collisions, and *SUM*, employed for efficient flow categorization based on their sizes and customized treatment strategies. M4 is implemented on CPU and Tofino platforms. CPU experimental results indicate that M4 outperforms two comparison frameworks in estimating the *value* distribution of huge, medium, and tiny flows. We have made our code publicly available on GitHub [50] to facilitate further research and application in this field.

ACKNOWLEDGMENT

The authors sincerely thank Xiaokun Yuan from Peking University for his participation in the experiments of the paper.

REFERENCES

[1] S. Dong et al., “M4: A framework for per-flow quantile estimation,” in *Proc. IEEE 40th Int. Conf. Data Eng.*, 2024, pp. 4787–4800.
[2] D. Nguyen, G. Memik, S. Memik, and A. Choudhary, “Real-time feature extraction for high speed networks,” in *Proc. Int. Conf. Field Program. Log. Appl.*, 2005, pp. 438–443.
[3] A. Bifet, “Mining Big Data in real time,” *Informatica*, vol. 37, no. 1, pp. 15–20, 2013.
[4] E. Viegas, A. Santin, A. Bessani, and N. Neves, “BigFlow: Real-time and reliable anomaly-based intrusion detection for high-speed networks,” *Future Gener. Comput. Syst.*, vol. 93, pp. 473–485, 2019.
[5] M. M. Rathore, H. Son, A. Ahmad, A. Paul, and G. Jeon, “Real-time Big Data stream processing using GPU with spark over hadoop ecosystem,” *Int. J. Parallel Program.*, vol. 46, pp. 630–646, 2018.
[6] Z. Fan et al., “Onesketch: A generic and accurate sketch for data streams,” *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12887–12901, Dec. 2023.
[7] T. Akidau et al., “Millwheel: Fault-tolerant stream processing at internet scale,” *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, 2013.
[8] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, “Continuous pattern detection over billion-edge graph using distributed framework,” in *Proc. Int. Conf. Data Eng.*, 2014, pp. 556–567.
[9] J. Kuhlenkamp, M. Klems, and O. Röss, “Benchmarking scalability and elasticity of distributed database systems,” *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1219–1230, 2014.
[10] Y. Zeng, Y. Tong, and L. Chen, “Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees,” *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 320–333, 2019.
[11] C. Yue et al., “Analysis of indexing structures for immutable data,” in *Proc. SIGMOD*, 2020, pp. 925–935.
[12] J. Wang, C. Chai, J. Liu, and G. Li, “Face: A normalizing flow based cardinality estimator,” *Proc. VLDB Endow.*, vol. 15, no. 1, pp. 72–84, 2021.
[13] B. Wang, R. Chen, and L. Tang, “Easyquantile: Efficient quantile tracking in the data plane,” in *Proc. 7th Asia-Pacific Workshop Netw.*, 2023, pp. 123–129.
[14] Y. Zhang, W. Zhang, J. Pei, X. Lin, Q. Lin, and A. Li, “Consensus-based ranking of multivalued objects: A generalized Borda count approach,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 1, pp. 83–96, Jan. 2014.

[15] Z. Shah, A. N. Mahmood, Z. Tari, and A. Y. Zomaya, "A technique for efficient query estimation over distributed data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2770–2783, Oct. 2017.

[16] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2195–2205, 2019.

[17] T. Dunning, "The size of a t-digest," 2019, *arXiv: 1903.09921*.

[18] G. Cormode, A. Mishra, J. Ross, and P. Vesely, "Theory meets practice at the median: A worst case comparison of relative error quantile algorithms," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2021, pp. 2722–2731.

[19] T. Dunning, "The T-digest: Efficient estimates of distributions," *Softw. Impacts*, vol. 7, 2021, Art. no. 100049.

[20] M. Szymaniak, D. Presotto, G. Pierre, and M. van Steen, "Practical large-scale latency estimation," *Comput. Netw.*, vol. 52, no. 7, pp. 1343–1364, 2008.

[21] T. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *Proc. 21st Annu. Joint Conf. IEEE Comput. Commun. Soc.*, 2002, pp. 170–179.

[22] R. Zhu, B. Liu, D. Niu, Z. Li, and H. V. Zhao, "Network latency estimation for personal devices: A matrix completion approach," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 724–737, Apr. 2017.

[23] Y. Liao, W. Du, P. Geurts, and G. Leduc, "DMFSGD: A decentralized matrix factorization algorithm for network distance prediction," *IEEE/ACM Trans. Netw.*, vol. 21, no. 5, pp. 1511–1524, 2013.

[24] G. Horváth, P. Buchholz, and M. Telek, "A MAP fitting approach with independent approximation of the inter-arrival time distribution and the lag correlation," in *Proc. 2nd Int. Conf. Quantitative Eval. Syst.*, 2005, pp. 124–133.

[25] D. J. Bertsimas and G. V. Ryzin, "Stochastic and dynamic vehicle routing with general demand and interarrival time distributions," *Adv. Appl. Probability*, vol. 25, no. 4, pp. 947–978, 1993.

[26] B. McShane, M. Adrian, E. T. Bradlow, and P. S. Fader, "Count models based on weibull interarrival times," *J. Bus. Econ. Statist.*, vol. 26, no. 3, pp. 369–378, 2008.

[27] R. Sinha, C. Papadopoulos, and J. Heidemann, "Internet packet size distributions: Some observations," USC/Information Sciences Institute, Marina Del Rey, CA, USA, Tech. Rep. ISI-TR-2007-643, pp. 1536–1276, 2007.

[28] P. Du and S. Abe, "Detecting dos attacks using packet size distribution," in *Proc. 2nd Bio-Inspired Models Netw. Inf. Comput. Syst.*, 2007, pp. 93–96.

[29] C.-N. Lu, C.-Y. Huang, Y.-D. Lin, and Y.-C. Lai, "Session level flow classification by packet size distribution and session grouping," *Comput. Netw.*, vol. 56, no. 1, pp. 260–272, 2012.

[30] D. Parish, K. Bharadia, A. Larkum, I. Phillips, and M. Oliver, "Using packet size distributions to identify real-time networked applications," *IEE Proc.- Commun.*, vol. 150, no. 4, pp. 221–227, 2003.

[31] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Comput. Netw.*, vol. 65, pp. 212–231, 2014.

[32] E. Cohen, E. Halperin, and H. Kaplan, "Performance aspects of distributed caches using TTL-based consistency," *Theor. Comput. Sci.*, vol. 331, no. 1, pp. 73–96, 2005.

[33] B. Wang, G. Chen, L. Fu, L. Song, and X. Wang, "DRIMUX: Dynamic rumor influence minimization with user experience in social networks," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 10, pp. 2168–2181, Oct. 2017.

[34] D. Kaur, G. S. Aujla, N. Kumar, A. Y. Zomaya, C. Perera, and R. Ranjan, "Tensor-based Big Data management scheme for dimensionality reduction problem in smart grid systems: SDN perspective," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 10, pp. 1985–1998, Oct. 2018.

[35] J. Li, W. Liang, W. Xu, Z. Xu, and J. Zhao, "Maximizing the quality of user experience of using services in edge computing for delay-sensitive IoT applications," in *Proc. 23rd Int. ACM Conf. Model. Anal. Simul. Wireless Mobile Syst.*, 2020, pp. 113–121.

[36] I. Alsmadi and D. Xu, "Security of software defined networks: A survey," *Comput. Secur.*, vol. 53, pp. 79–108, 2015.

[37] Y. Chen, J. Pei, and D. Li, "DETPro: A high-efficiency and low-latency system against ddos attacks in SDN based on decision tree," in *Proc. IEEE Int. Conf. Commun.*, 2019, pp. 1–6.

[38] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical cache attacks from the network," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 20–38.

[39] M. Shahzad and A. X. Liu, "Noise can help: Accurate and efficient per-flow latency measurement without packet probing and time stamping," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 207–219, 2014.

[40] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J. Schmitz-Hermes, "Adaptive caching in big SQL using the HDFS cache," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 321–333.

[41] Y. Zhang and R. Gupta, "Enabling partial-cache line prefetching through data compression," *High- Perform. Comput.: Paradigm Infrastructure*, pp. 183–201, 2005.

[42] P. Chen, D. Chen, L. Zheng, J. Li, and T. Yang, "Out of many we are one: Measuring item batch with clock-sketch," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 261–273.

[43] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Vesely, "Relative error streaming quantiles," in *Proc. PODS Conf.*, 2021, pp. 96–108.

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

[45] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *Proc. Annu. Symp. Foundations Comput. Sci.*, 2016, pp. 71–78.

[46] R. Shahout, R. Friedman, and R. B. Basat, "Squad: Combining sketching and sampling is better than either for per-item quantile estimation," 2022, *arXiv:2201.01958*.

[47] J. He, J. Zhu, and Q. Huang, "HistSketch: A compact data structure for accurate per-key distribution monitoring," in *Proc. Int. Conf. Data Eng.*, 2023, pp. 2008–2021.

[48] J. Guo, Y. Hong, Y. Wu, Y. Liu, T. Yang, and B. Cui, "SketchPolymer: Estimate per-item tail quantile using one sketch," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2023, pp. 590–601.

[49] "CAIDA Anonymized Internet Traces 2018 Dataset," 2018/. [Online]. Available: https://www.caida.org/catalog/datasets/request_user_info_forms/passive_dataset_request/

[50] "The source codes of ours along with other algorithms," 2025/. [Online]. Available: https://github.com/pkufzc/M4

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

[52] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. System Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[53] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.

[54] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. Int. Colloq. Automata Lang. Program.*, 2002, pp. 693–703.

[55] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 5, pp. 1622–1634, Oct. 2012.

[56] F. Deng and D. Rafiei, "New estimation algorithms for streaming data: Count-min can do more," 2007. [Online]. Available: https://webdocs.cs.ualberta.ca/~drafiei/papers/cmm.pdf

[57] I. Epicoco, C. Melle, M. Cafaro, M. Pulimeno, and G. Morleo, "UDDSketch: Accurate tracking of quantiles in data streams," *IEEE Access*, vol. 8, pp. 147604–147617, 2020.

[58] E. Gribelyuk, P. Sawettamalya, H. Wu, and H. Yu, "Simple & optimal quantile sketch: Combining Greenwald-Khanna with Khanna-Greenwald," *Proc. ACM Manage. Data*, vol. 2, no. 2, pp. 1–25, 2024.

[59] "Bob jenkins' hash function web page, paper published in DR dobb's journal," 1997. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[60] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. ACM Int. Conf. Emerg. Netw. Experiments Technol.*, 2014, pp. 75–88.

[61] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[62] "MAWIWorking group traffic archive," 2010. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[63] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.

[64] 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 Trans. Netw. Service Manag.*, vol. 16, no. 2, pp. 535–549, Jun. 2019.

[65] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: A platform for educational cloud computing," in *Proc. 40th ACM Tech. Symp. Comput. Sci. Educ.*, 2009, pp. 111–115.

[66] "Barefoot tofino: World's fastest P4-programmable ethernet switch asics," 2022. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

Zhuochen Fan (Member, IEEE) received the PhD degree in computer science from Peking University, in 2023, advised by Professor Tong Yang. He is currently an associate researcher with Pengcheng Laboratory. Before joining Pengcheng Laboratory, he was a Boya postdoctoral research fellow with Peking University. His research interests include approximation algorithms in computer networks and databases. He had articles published by *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, *The VLDB Journal*, ICDE, RTSS, ICPP, ICNP, USENIX ATC, *etc*.

Yalun Cai received the BS degree from the Department of Electrical Engineering and Computer Science, Peking University, in 2023. His research interests include network measurements and sketches.

Siyuan Dong (Member, IEEE) is currently working toward the undergraduate degree majoring in computer science with the School of Electronics Engineering and Computer Science, Peking University, advised by Professor Tong Yang. His research interests include cover network measurements, sketches, bloom filters, and hash tables.

Qiuheng Yin is currently working toward the undergraduate degree majoring in computer science with the School of Electrical Engineering and Computer Science, Peking University. His research interests include network measurements, programmeable switch, and network system.

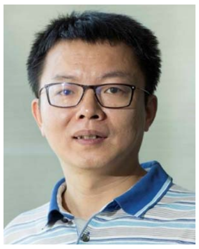
Tianyu Bai is currently working toward the undergraduate degree majoring in computer science with the School of Electrical Engineering and Computer Science, Peking University. He is now exploring network measurements and in-network computing.

Hanyu Xue is currently working toward the undergraduate degree majoring in mathematics with the Yuanpei College, Peking University. His current research interest includes algebraic topology and mathematical physics.

Peiqing Chen is a PhD student at the Department of Computer Science, University of Maryland, College Park, advised by Professor Zaoxing Liu. His current research interest includes system and networking.

Yuhan Wu received the bachelor degree from the Department of Electrical Engineering and Computer Science, Peking University, in 2021. He is currently working toward the PhD degree in computer science with the School of Computer Science, Peking University, advised by Tong Yang. His research interests include the fields of computer network and database, including key-value stores, network measurement, and sketches.

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). Now he is 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 served as a TPC Member for several premier conferences such as ICDE, INFOCOM, IMC, ICNP, *etc*.. He is currently an associate editor for *Knowledge and Information Systems*. He published dozens of papers in *IEEE/ACM Transactions on Networking*, *IEEE Journal on Selected Areas in Communications*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Knowledge and Data Engineering*, SIGCOMM, SIGKDD, SIGMOD, NSDI, USENIX ATC, ICDE, VLDB, INFOCOM, *etc*.



Bin Cui (Fellow, IEEE) received the PhD degree from the National University of Singapore, in 2004. He is a professor and vice dean with the School of Computer Science at Peking University. His research interests include database system architectures, query and index techniques, Big Data management and mining. He is editor-in-chief of *Data Science and Engineering*, in the Editorial Board of *Distributed and Parallel Databases*, *Journal of Computer Science and Technology*, and *Science China Information Sciences*, and was an associate editor of *IEEE Transactions on Knowledge and Data Engineering (TKDE)* and *VLDB Journa*l. He is serving as vice chair of Techical Commettee on Database CCF (2020-2027), and had served as Trustee Board Member of VLDB Endowment (2016-2021). He was awarded Microsoft Young Professorship award (MSRA 2008), CCF Young Scientist award (2009), and Second Prize of Natural Science Award of MOE China (2014), *etc*.