M4: A Framework for Per-Flow Quantile Estimation

Siyuan Dong
Peking University
dongsiyuan@pku.edu.cn

Zhuochen Fan
Peking University
fanzc@pku.edu.cn

Tianyu Bai
Peking University
tianyubai@stu.pku.edu.cn

Tong Yang
Peking University
yangtong@pku.edu.cn

Hanyu Xue
Peking University
xuehanyu02@gmail.com

Peiqing Chen
Peking University
chenpeiqing@pku.edu.cn

Yuhan Wu
Peking University
yuhan.wu@pku.edu.cn

ABSTRACT

The field of quantile estimation has grown in importance due to its myriad practical applications. 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. 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 the straw-man solution in terms of accuracy for per-flow quantile estimation while maintaining comparable speed.

1 INTRODUCTION

1.1 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 [1–4]. Among the various types of information, quantile information, which requires distribution statistics for data streams, has become a focal point of numerous studies [5–11]. 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 [12–21], Inter-Arrival Time [22–24], Packet Size [25–28], and TTL (Time to Live) Value [29, 30]. 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 [31–33], enhancing network attack detection [34–36], and boosting the performance of Content Delivery Networks (CDNs) [37]. 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 $\mathcal{F} = \{\langle a, x_1\rangle, \langle a, x_2\rangle, \cdots, \langle a, x_n\rangle\}$ of size n, where a is the *key*, x_i is the *value* and $x_1 \leq x_2 \leq \cdots \leq x_n$. Given a percentage p ($0 \leq p \leq 1$), the p-quantile of *value* is x^* $s.t.$ the percentage of $x_i \leq x^*$ 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*

Numerous prior works have made significant contributions to single-flow quantile estimation [14–17, 38–41]. Among these, approximate algorithms (often referred to as sketches) have demonstrated excellent performance due to their low memory overhead and fast processing speed, with only minor compromises in accuracy. However, these single-flow quantile estimation algorithms can only calculate the quantile of a *value* for a single *key* or disregard *keys* altogether to estimate the quantile for all data stream *values*. They do not differentiate between different flow IDs, leading to either a single-flow estimation or a broad calculation for all flows. To achieve our design goal of per-flow quantile estimation, these algorithms must be adapted to support arbitrary flow queries since we cannot predict which specific flows require measurement in advance.

To address the challenge of per-flow quantile estimation, two approaches can be considered: design a new algorithm [42, 43] from scratch, or develop a framework that transforms existing single-flow algorithms into corresponding per-flow algorithms. We

[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.

opt for the latter because it provides broad applicability and simplifies design. Different application scenarios prioritize different aspects of algorithms, such as high throughput, high accuracy, or low memory overhead. Therefore, there is a need for a diverse range of algorithms with varying strengths. Designing a new per-flow algorithm often cannot simultaneously meet the requirements of various scenarios. By developing a framework, we can create a reusable solution that not only preserves the unique strengths of existing single-flow algorithms for different scenarios, but also enhances their capacity to support more advanced properties, such as per-flow estimation. Thus, a framework allows us to handle a wide range of scenarios in the most efficient manner.

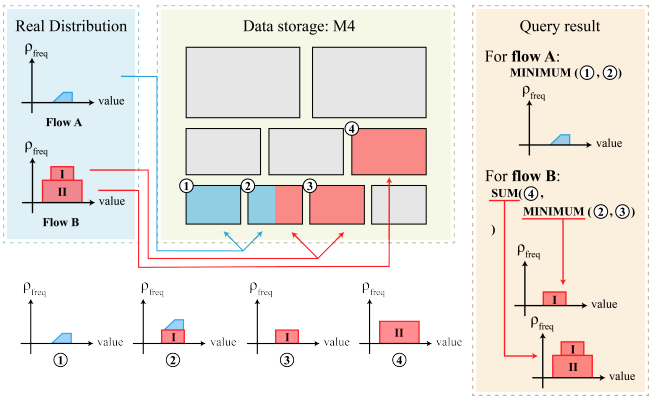


Figure 1: Illustration of M4.

1.2 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 Figure 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.

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 Figure 1, a small flow A and a large flow B are inserted into M4. Flow A can be contained in the first layer, while flow B is segmented into parts I and II[2] 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 Figure 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. 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[3]. 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 Figure 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 Figure 1 that the *SUM* technique allows us to restore the real distribution of flow B.

We apply our M4 to three *META*s (DDSketch [14], t-digest [15, 17], and modified ReqSketch [16, 38]). The three *META*s each

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

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.

We devise a straw-man solution (Section 5.1.2) for better comparison. We perform extensive experiments to evaluate our performance regarding accuracy and speed. Experiment results indicate that *M4 is per-flow friendly and accurate.* For tiny flows, maximum *value* estimation is on average 90.6% error-free, while the straw-man solution offers almost no error-free estimates. For larger flows, the Average Logarithm Error (ALE) and Average Percentage Error (APE) of M4 reach 2.26× and 1.99× lower than the straw-man solution. *M4 is memory-efficient.* It only needs 6MB to handle 27M items. We also provide theoretical proof that M4 delivers high accuracy while utilizing limited memory.

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.
- We apply M4 to DDSketch, t-digest, and modified ReqSketch and implement them on a CPU platform. Compared to the straw-man solution, M4 achieves significantly better accuracy with a comparable speed across all three algorithms. All codes are available on GitHub [44].

2 RELATED WORK

2.1 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 [45], designed for estimating item frequency. CM Sketch consists of d arrays, each array $A_i (1 \le i \le d)$ 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 [46], CU Sketch [47], Count Sketch [48], CSM Sketch [49] and CMM Sketch [50].

2.2 Single-Flow Quantile Estimation

Quantile estimation is a statistical method used to approximate answers about the distribution of certain metrics. These methods are crucial for large or streaming datasets in query optimization and data analysis. Many prior works have contributed to single-flow quantile estimation [14–17, 38–41]. However, these works focus on accurately estimating distribution for a single flow, while data streams often consist of multiple flows (belonging to different users or services). We need to efficiently store and analyze the distribution information of each flow separately, which cannot be achieved by single-flow distribution estimation solutions. Our proposed framework, M4, can be applied to single-flow distribution estimation algorithms, enabling per-flow distribution estimation. Next, we will introduce three examples of single-flow distribution estimation algorithms DDSketch, t-digest, and ReqSketch, as they will be used as meta-algorithms to show how M4 works. These three algorithms have different emphases and strengths. 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.

2.2.1 *DDSketch.*

DDSketch [14] is designed to estimate *value* distributions by dividing 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. We define $\gamma := (1+\alpha)/(1-\alpha)$ to determine the boundaries of every segment. Specifically, if we index each segment by $i \in \mathbb{Z}$, then the counter C_i records the number of *value* x that falls between $\gamma^{i-1} < x \le \gamma^i$. Thus, we insert x into the segment indexed by $\lceil log_\gamma(x) \rceil$. The key idea of DDSketch is to devise an approximation strategy to represent all the items in each segment with the same estimation *value* $\hat{x} = \frac{2\gamma^i}{\gamma+1}$, and the relative error of this estimation is bounded by α, i.e. $\frac{|x-\hat{x}|}{x} \le \alpha$. A larger α means longer segments and lower accuracy. In other words, given a fixed number of segments, a larger α allows us to cover a wider range at the cost of accuracy. To query for the *value* x at percentage $p \in [0, 1]$,we accumulate counter values until finding the segment i that p falls in. Then we report $\hat{x} = \frac{2\gamma^i}{\gamma+1}$ as the quantile estimation result.

2.2.2 *t-digest.*

t-digest [15, 17] 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* $k : [0, 1] \to \mathbb{R}$ describing the weight restriction and a *compression parameter* δ 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: $k(\frac{w_{<i}+w_i}{N}) - k(\frac{w_{<i}}{N}) \le \frac{1}{\delta}$. 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.

2.2.3 ReqSketch.

ReqSketch [16, 38] consists of several levels of *compactor* serving as buffers to store the *values* of items. To record a flow of size N, ReqSketch requires $O(log_2 N)$ *compactor*s, each with a buffer sized $O(log_2 N)$. The incoming items are always inserted into level 0. Whenever the *compactor* of level h becomes full, we sort the items in this level and choose an appropriate even-sized prefix for compaction. Then we remove the prefix from level h and randomly select half[4] of the prefix to be inserted into the *compactor* of level $h+1$. This process is called *compaction operation*. Each item stored in level h carries a weight of 2^h, which means the total weight of the stored items remains unchanged after the *compaction operation*. This is because when we remove $w \times 2^h$ weights from level h, the weight gain in level $h+1$ is $\frac{w}{2} \times 2^{h+1}$. The key idea of ReqSketch is to sample items according to the need without causing much influence on the distribution estimation. To query for the *value* at percentage $p \in [0, 1]$, we accumulate item weights from the item with the smallest *value* until the sum reaches p. Then, the *value* of the item we arrive at is reported as the estimation result.

3 M4 DESIGN

3.1 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 $\mathcal{S} = \{x_1, x_2, \cdots, x_n\}$ of size n, where $x_1 \le x_2 \le \cdots \le x_n$, and a percentage p $(0 \le p \le 1)$, 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:

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

3.2 Framework Description

This section describes the structure and operations of M4. Frequently used notations are outlined in Table 1.

3.2.1 Framework Structure.

As depicted in Figure 2, M4 is a four-level bucket array, where the i^{th} level, denoted by L_i, has b_i buckets and is associated with w_i hash functions $(h_{i,1}(\cdot), h_{i,2}(\cdot), \ldots, h_{i,w_i}(\cdot))$. L_1 records the size and maximum *value* of tiny flows. L_2 records the *value* distribution of medium flows. L_3 and L_4 record the *value* distribution of huge flows. Hereafter in this article, tiny flows are defined as flows with

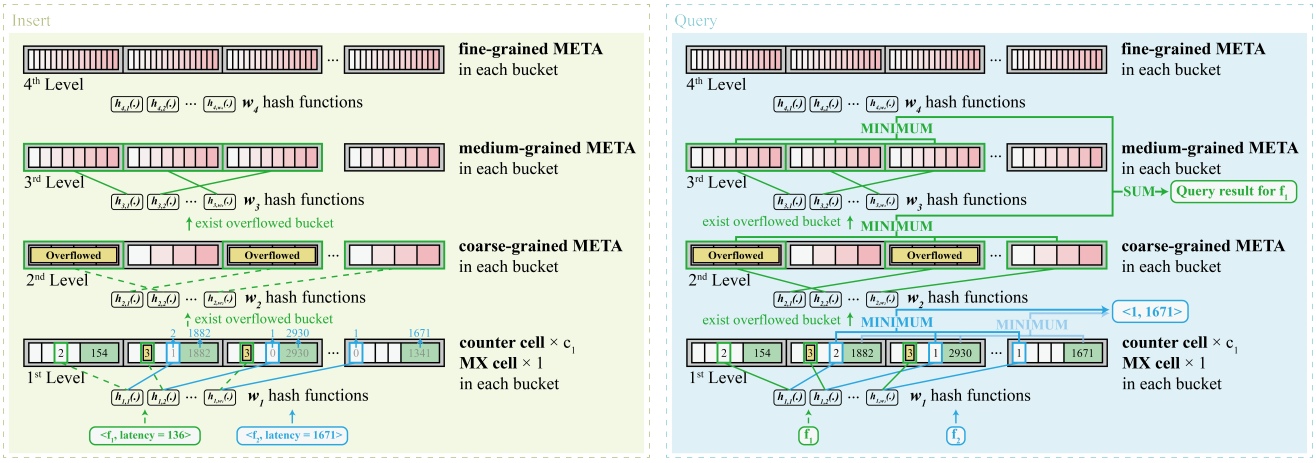
[4] either the odd-indexed half or the even-indexed half

Table 1: 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

Algorithm 1: Framework Insertion

```
1  Function Insert-To-Framework(e):
2      Input: Item : e = ⟨f, v⟩
3      for i = 1, 2, 3, 4 do
4          if not L_i.isOverflowed(f) then
5              Insert-To-Level(e, i)
6              return
7          end
8      end
9      /* control never reaches here */
10 End Function
11 Function Insert-To-Level(e, i):
12     Input: Item: e = ⟨f, v⟩, level: i in {1, 2, 3, 4}
13     if i == 1 then
14         for j = 1, ··· , w_1 do
15             buc_idx = Calc-Buc-Idx-In-Level(f, 1, j)
16             cnt_idx = h_{1,j}(f)%c_1
17             L_1[buc_idx].counters[cnt_idx]+ = 1
18             L_1[buc_idx].MX = max(L_1[buc_idx].MX, v)
19         end
20     else
21         for j = 1, ··· , w_i do
22             buc_idx = Calc-Buc-Idx-In-Level(f, i, j)
23             L_i[buc_idx].insert(e)
24         end
25     end
26 End Function
27 Function Calc-Buc-Idx-In-Level(f, i, j):
28     Input: Flow ID: f, level: i in {1, 2, 3, 4},
29            index of hash function: j
30     Output: Index of the bucket to which flow f is mapped
31             in L_i under the j^th hash function
32     return i == 1 ? ⌊h_{1,j}(f)%(b_1 c_1) / c_1⌋ : h_{i,j}(f)%b_i
33 End Function
```



(a) Insert (b) Query

Figure 2: Framework structure of M4.

a size $\in [1, 3)$, medium flows as flows with a size $\in [3, 255)$, and huge flows as flows with a size $\in [255, +\infty)$. For simplicity, we call the algorithm on which we employ M4 as *META*, and the M4 loaded with *META* as M4-*META*. We use DDSketch, *t*-digest, and modified ReqSketch (mReqSketch) as *META*s.

The configuration of L_1 for each M4-*META* is identical. Each bucket consists of c_1 counter cells and one MX cell, recording the flow size and maximum *value* respectively. Each bucket can store the size of c_1 flows, which share one maximum *value*. The number of bits in a counter cell (counter size) is l_1. A counter cell in L_1 is considered overflowed when the recorded flow size reaches the largest number that l_1 bits can represent. L_1 is designed this way because in most scenarios, tiny flows are not important. If we need to improve the accuracy for tiny flows, we can increase the memory allocated for L_1 and decrease c_1.

For $L_i (i \geq 2)$, each bucket consists of a *META*. All the *META*s at the same level have the same capacity and granularity. Lower-level *META*s have smaller capacities and coarser granularity, while higher-level *META*s have larger capacities and finer granularity. This is because smaller capacities are required to record smaller flows, which are also less important than larger ones. A bucket in $L_i (i \geq 2)$ is deemed overflowed when the number of recorded items reaches the *META*'s capacity within the bucket. The details for different *META*s on $L_i (i \geq 2)$ are as follows:

DDSketch. As mentioned in Section 2.2.1, a DDSketch in $L_i (i \geq 2)$ contains c_i ($c_2 \leq c_3 \leq c_4$) counter cells recording the frequency of *value* falling into each of the c_i segments. Also, the counter size in L_i is l_i ($l_2 \leq l_3 \leq l_4$). A bucket in $L_i (i \geq 2)$ is deemed overflowed when the frequency recorded in any counter cell reaches the largest number that l_i bits can represent.

***t*-digest.** As mentioned in Section 2.2.2, a *t*-digest in $L_i (i \geq 2)$ contains c_i ($c_2 \leq c_3 \leq c_4$) clusters. Each cluster maintains an *average* cell and a *weight* cell recording the mean *value* and the number of absorbed items. Also, the length of each *weight* cell in L_i is l_i ($l_2 \leq l_3 \leq l_4$). A bucket in $L_i (i \geq 2)$ is deemed overflowed when the frequency recorded in any *weight* cell reaches the largest number that l_i bits can represent.

mReqSketch. As mentioned in Section 2.2.3, an mReqSketch in $L_i (i \geq 2)$ contains l_i ($l_2 \leq l_3 \leq l_4$) compactors. Each compactor in L_i maintains c_i ($c_2 \leq c_3 \leq c_4$) cells for recording *values*. The maximum total weight of a mReqSketch in L_i is $(2^{l_i} - 1) * c_i$. A bucket in $L_i (i \geq 2)$ is deemed overflowed when the frequency recorded in it reaches this maximum total weight.

3.2.2 *Framework Insertion Operation.*

To insert an item $e = \langle f, v \rangle$ into M4-*META*, we aim to insert e into the lowest level that is not overflowed (we call it L_{top}). We first attempt to insert it into L_1 by mapping it to w_1 buckets with an index of $\lfloor \frac{h_{1,j}(f)\%(b_1 c_1)}{c_1} \rfloor$, where $j \in \{1, 2, 3, \ldots, w_1\}$. The $(h_{1,j}(f)\%c_1)^{th}$ counter cell in each mapped bucket is the mapped counter. If any of the w_1 mapped counters in L_1 overflows, we consider L_1 overflowed for flow f and try to insert e into L_2. Otherwise, we increase all the w_1 mapped counters by one and update the *MX* cell in each mapped bucket by setting $MX = \max\{MX, v\}$, ending the insertion.

For L_i ($i \geq 2$), we map e to w_i buckets with an index of $h_{i,j}(f)\%b_i$, where $j \in \{1, 2, 3, \ldots, w_i\}$. If any of the w_i mapped buckets overflows, we consider L_i overflowed for flow f and try to insert e into L_{i+1}. Otherwise, we insert e into each *META* in the w_i mapped buckets, ending the insertion.

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

3.2.3 *Framework Query Operation.*

When querying a flow with ID f, we must find all levels containing its information. Following the same procedure as the insertion operation, we map the flow to their corresponding buckets at each level. The query process starts at L_1 and stops at L_{top} (the lowest level that is not overflowed for f). The difference from the insertion operation is that we need to return the aggregated result of all levels with index $\leq top$, not just L_{top}.

At each L_i ($i \geq 1$), we get w_i distribution records of flow f. All the w_i records are potentially polluted because of hash collisions. We need to use them to generate the least polluted distribution, which is why we need the *MINIMUM* technique. The *MINIMUM* technique can minimize the pollution brought by hash collisions. For L_1, the *MINIMUM* technique returns the minimum

of all mapped counter cells and the minimum of all mapped MX cells, respectively, as the size and the maximum *value* of flow f. For each L_i ($i \geq 2$), the *MINIMUM* technique differs slightly according to the design of *META*. We will go into detail in Section 3.3.

Every level from L_1 to L_{top} contains part of the distribution information of flow f. If $top = 1$, we only return the size and the maximum *value* of flow f, generated with *MINIMUM* on L_1. If $top \geq 2$, we return the *SUM*-merged result of levels from L_2 to L_{top}. The *SUM* technique differs slightly according to the design of *META* and will be elaborated upon in Section 3.3.

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

3.2.4 Example.

The chosen sample for our explanation consists of $\langle c_1 = 4, l_1 = 2, w_1 = 3\rangle$, $\langle w_2 = 3\rangle$, $\langle w_3 = 3\rangle$, $\langle w_4 = 3\rangle$.

Figure 2(a) depicts an instance of the insertion operation in M4. When item $e_1 = \langle f_1, v = 136\rangle$ arrives, we first map it to three buckets in L_1. We notice that two of the three mapped counters are 3, which is the largest number that $l_1 = 2$ bits can represent. So L_1 is deemed overflowed for flow f_1. We then map e_1 to three buckets in L_2. Two mapped buckets have overflowed, so L_2 is also deemed overflowed for flow f_1. We proceed to map e_1 to three buckets in L_3. Since none of the mapped buckets are overflowed, we insert e_1 into them, ending the insertion operation. Upon the arrival of item $e_2 = \langle f_2, v = 1671\rangle$, we map it to three buckets in L_1, where we find the three mapped counters are 1, 0, and 1. As L_1 is not overflowed for flow f_2, we increment all the counters by 1. For the MX cells, because 1341<1671<1882<2930, we update 1341 to 1671 and leave the other two *MX* cells unchanged.

Figure 2(b) depicts an instance of the query operation in M4. We query for the flow f_1, f_2 mentioned in the insertion example above. When querying f_1, mapping to L_1 and L_2 reveals overflow. L_3 is the lowest level that is not overflowed for f_1, so we construct the query result for f_1 using the *SUM* technique to merge the *MINIMUM*-merged results in L_2 and L_3. When querying for f_2, we map it to three buckets in L_1 and find the three counters are 2, 1, and 2, so L_1 is not overflowed for f_2. As a result, we return flow size = $\min\{2, 1, 2\} = 1$ and maximum *value* = $\min\{1671, 1882, 2930\} = 1671$.

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 ··· , top − 1/* empty set if top ≤ 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, ··· , 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
```

3.3 *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 3.3.2). Subsequently, we perform *MINIMUM* and *SUM* operations on the histograms (Section 3.3.3).

3.3.1 Rationale.

MINIMUM. The *MINIMUM* technique addresses the problem of hash collisions. Conventional methods of resolving hash collisions involve recording ID and executing complex operations (typically dependent on the number of flows) to locate another available bucket during a hash collision, which is both memory consuming and time consuming. In contrast, our solution eliminates the need for ID recording and executes both the insertion and query operations in $O(1)$ time. The *MINIMUM* technique, based on the observation that distribution estimation can be regarded as frequency estimation at each *value* point (which is density), helps in this regard. Pollution on a flow bucket only increases the density at some *value* points. Hence, if two buckets recording the same flow differ in density at some *value* points, the lower density is always the more accurate. By selecting the lowest density at each *value* point recorded in all mapped buckets, we obtain the best possible estimation of the *value* distribution.

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 layers 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.

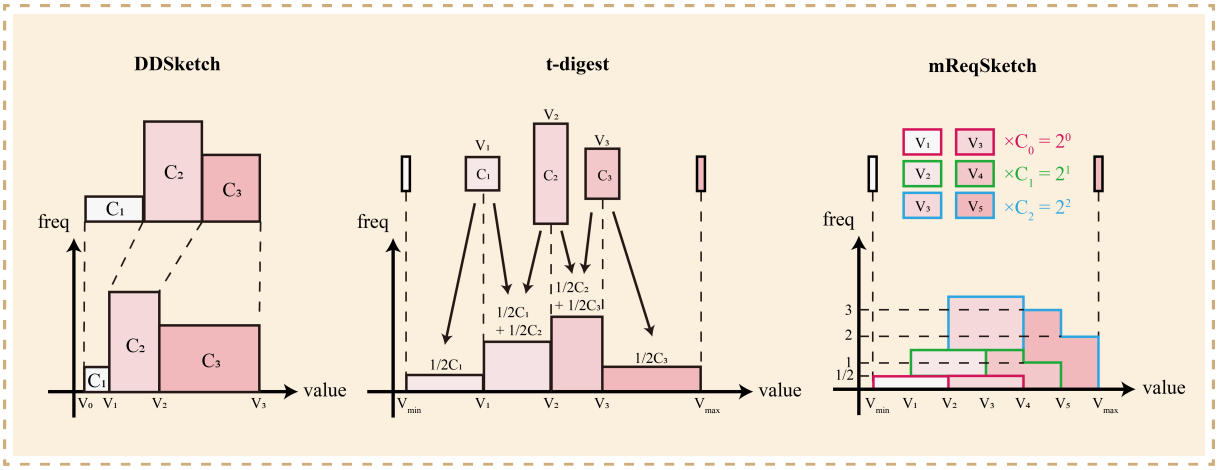


Figure 3: From *META* to Histogram.

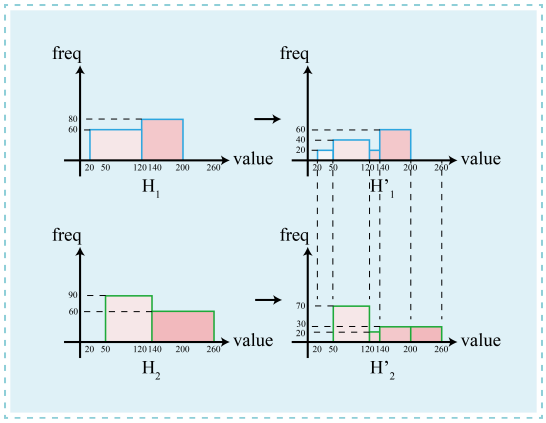


Figure 4: Segment Alignment.

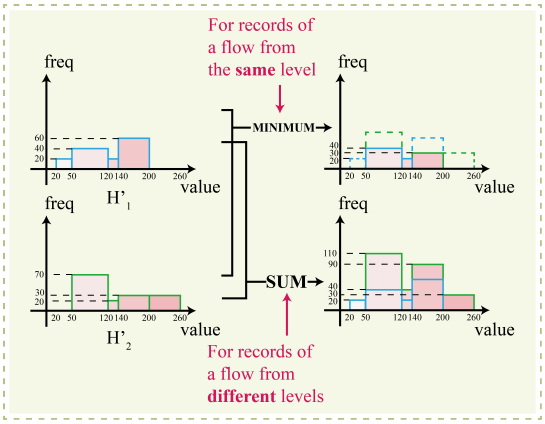


Figure 5: *MINIMUM & SUM* on aligned Histograms.

3.3.2 *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 2.2.1, 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 Figure 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.

***t*-digest.** As discussed in Section 2.2.2, *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 Figure 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 2.2.3, 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 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 Figure 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.

3.3.3 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 Figure 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 Figure 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.

4 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.

4.1 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.

4.1.1 Huge and Medium Flows.

Theorem 1. *Let $\hat{freq}_i$ denote the estimation of $freq_i$. Then*

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

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

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

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 $\hat{freq}_{i,1} = freq_i + \sum_{j \neq i} freq_j 1_{A_j}$ 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 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,

$$P(\hat{freq}_i > freq_i + \epsilon) = P^w(\hat{freq}_{i,1} - freq_i > \epsilon) < (\frac{freq}{n\epsilon})^w \tag{4}$$

□

Theorem 2. *Let $\hat{freq}_i^T$ denote the estimation of $freq_i^T$. Then*

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

Proof. Let $\hat{freq}_{i,k}^T$ ($k \in \{1, 2, \ldots, w\}$) denote the k^{th} record of $freq_i^T$, independent from each other. Since $\hat{freq}_i^T = \min\{\hat{freq}_{i,k}^T\}$, we obtain

$$P(\hat{freq}_i^T > freq_i^T + \epsilon) = P^w(\hat{freq}_{i,k}^T > freq_i^T + \epsilon) \tag{6}$$

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 $\hat{freq}_{i,1}^T = freq_i^T + \sum_{j \neq i} freq_j^T 1_{A_j}$ and $P(A_j) = \frac{1}{n}$. Hence, $E(\hat{freq}_{i,1}^T - freq_i^T) = \sum_{j \neq i} freq_j^T \frac{1}{n} < \frac{freq^T}{n}$. Because $\hat{freq}_{i,1}^T - freq_i^T \geq 0$, we obtain

$$P(\hat{freq}_{i,1}^T - freq_i^T > \epsilon) \leq \frac{E(\hat{freq}_i^T - freq_i^T)}{\epsilon} < \frac{freq^T}{n\epsilon} \tag{7}$$

Therefore,

$$P(\hat{freq}_i^T > freq_i^T + \epsilon) = P^w(\hat{freq}_{i,1}^T - freq_i^T > \epsilon) < (\frac{freq^T}{n\epsilon})^w \tag{8}$$

□

Theorem 3. *Let $\hat{t_p}$ denote the estimated quantile of percentage p. Then*

$$P(|\hat{t_p} - t_p| < \alpha t_p) \geq 1 - (1 - e^{-\frac{m}{n}})^w \tag{9}$$

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.

4.1.2 Tiny Flows.

Theorem 4. *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

$$P_{\{k_i\}} = \frac{\binom{wm-w}{k_1}\binom{wm-w-k_1}{k_2}\ldots\binom{wm-w-k_1-k_2-\ldots-k_{w-1}}{k_w}}{n^{wm-w}} \cdot (n-w)^{wm-k_1-k_2-\ldots-k_w-w}. \quad (10)$$

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

$$\begin{aligned} &[\prod_i \sum_{k_i=1}^{wm} \binom{wm}{k_i} n^{-k_i} \frac{k_i}{k_i+1}] e^{-\frac{w^2 m}{n}} \\ &= [\sum_{k=1}^{wm} \binom{wm}{k} n^{-k} \frac{k}{k+1}]^w e^{-\frac{w^2 m}{n}} \\ &= [\frac{n-(1+\frac{1}{n})^{wm}(-wm+n)}{wm+1} e^{-\frac{wm}{n}}]^w \\ &\overset{x=\frac{wm}{n}}{\approx} (\frac{e^{-x}+x-1}{x})^w \end{aligned} \quad (11)$$

□

4.2 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 5. *Let $\hat{R(y)}$ denote the estimation of $R(y)$. Then*

$$P(|\hat{R}(y) - R(y)| > \epsilon N) < 2e^{-\frac{4b^2}{1-(1-2p)^b}\epsilon^2} \quad (12)$$

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

$$\begin{aligned} P &= \frac{\sum_{i=1}^{b}[\binom{b}{i}p^i(1-p)^{b-i}] - \sum_{i=1}^{b}[\binom{b}{i}(-p)^i(1-p)^{b-i}]}{2} \\ &= \frac{1-(1-2p)^b}{2} \end{aligned} \quad (13)$$

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

$$\begin{aligned} \sigma^2 &= P(1-P)(2^M + 2^{M+1} + \ldots + 2^{2M-1}) \\ &= P(1-P)2^M(2^M-1) \\ &\approx \frac{1-(1-2p)^{2b}}{4} 2^{2M} \\ &\approx \frac{1-(1-2p)^b}{4} \frac{N^2}{b^2} \end{aligned} \quad (14)$$

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

$$P(|\hat{R}(y) - R(y)| > \epsilon N) < 2e^{-(\frac{\epsilon N}{\sigma})^2} = 2e^{-\frac{4b^2}{1-(1-2p)^b}\epsilon^2} \quad (15)$$

5 EXPERIMENTAL RESULTS

5.1 Experiment Setup

5.1.1 Implementation.

We implemented M4 and all related *META* data structures (DDSketch, t-digest, and mReqSketch) in C++. The hash functions were devised using the 32-bit Bob Hash (sourced from an open-source website [51]), each with different initial seeds. All the experiments were executed on an 18-core CPU server (Intel i9-10980XE) with 128GB memory and 24.75MB L3 cache. Each experiment was repeated ten times to compute an average result.

5.1.2 Algorithm Comparison.

For a comprehensive comparison, we devised a straw-man solution applied to every *META* (DDSketch, t-digest, and mReqSketch). The straw-man solution incorporates the idea of Dleft, consisting

Table 2: 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		

of three bucket arrays[5]. Every bucket stores a key field to record the flow ID and a $META$ to record the *value* distribution. There are three corresponding hash functions, $h_1'(\cdot), h_2'(\cdot)$, and $h_3'(\cdot)$, for the three bucket arrays. Furthermore, the straw-man solution maintains a global $META$ to track the *value* distribution of items across all flows. If the queried flow's information is not present in the data structure, we default to the query result from the global $META$.

Insertion. Upon the arrival of an item $e = \langle f, v \rangle$, it is initially inserted into the global $META$. We then attempt to insert e into the $h_1'(f)^{th}$ bucket of the first array. If the bucket is unoccupied, or if the flow in the bucket has $key = f$, we insert e into the bucket's $META$ and set key to f. If not, we attempt to insert e into the $h_2'(f)^{th}$ bucket of the second array. The process is repeated in the third array if the second one cannot accept e. If the third array also rejects e, we discard e.

Query. To query for a flow f, we seek a mapped bucket with $key = f$ from the $h_1'(f)^{th}$, $h_2'(f)^{th}$, or $h_3'(f)^{th}$ bucket of the respective arrays. If a matching bucket is found, we return the query result from the $META$ of that bucket. Otherwise, we return the query result from the global $META$.

Possible alternative. Another alternative straw-man solution is Cuckoo hashing. It uses two hash functions. If one position is occupied for an item, it can go to the other one. When an item arrives, we check the first position (using the first hash function). If it's empty, we insert it there. If it's occupied, we check the second position (using the second hash function). If it's empty, we insert it there. If it's also occupied, we insert the item into one of the two positions, and "kick out" the item that was already there. So we need to rehash the kicked-out item and try to place it in its alternate position. Then we continue this process until every item finds a position or until a certain number of displacements have been attempted (this can be a predefined threshold).

We choose Dleft rather than Cuckoo hashing as the straw-man solution because memory is limited ($\#flow > \#bucket$). Therefore, all the buckets will eventually be occupied for both Dleft and Cuckoo hashing, but Cuckoo hashing is much slower than Dleft because of its "kick out" operation.

5.1.3 Datasets.

(1) *CAIDA Dataset.* This dataset comprises streams of anonymized IP items collected from high-speed monitors by CAIDA in 2018 [52]. We use the trace with a monitoring interval of 60s. Each item consists of a 5-tuple (13 bytes). There are around 27M items and 1.3M flows in this dataset.

(2) *MAWI Dataset.* This dataset contains real traffic trace data maintained by the MAWI Working Group [53]. Similar to CAIDA, each item in the dataset is a 5-tuple. There are around 9M items and 13K flows in the MAWI dataset.

(3) *IMC Dataset.* This dataset comes from one of the data centers studied in [54]. Each item also consists of a 5-tuple. There are around 14M items and 5K flows in this dataset.

5.1.4 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 quantile 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) *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.

(4) *Throughput (Insertion and Query).* We use million operations (insert an item or query a flow) per second (Mops) to measure the throughput.

5.1.5 Default settings.

In our experiment, we set $p = 0.5$ as the default setting[6]. 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 2.

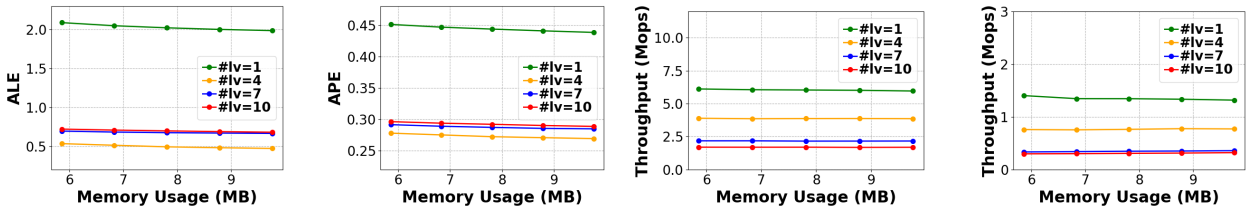
5.2 Experiments on Parameter Setting

Effect of # levels. We conduct experiments on the choice of the number of levels for M4 and the straw-man solution. As shown in Figure 6, choosing # $lv = 4$ gives the best accuracy and a comparatively good speed. So we design the M4 to have four levels. As shown in Figure 7, choosing # $lv = 3$ gives a close to the best accuracy and a comparatively good speed. So we design the straw-man solution to have three levels.

Effect of w. As shown in Figure 8, the optimal performance is achieved when $w = 3$. We conduct experiments on different M4-$META$s on various datasets. Altering the hash number w, we find

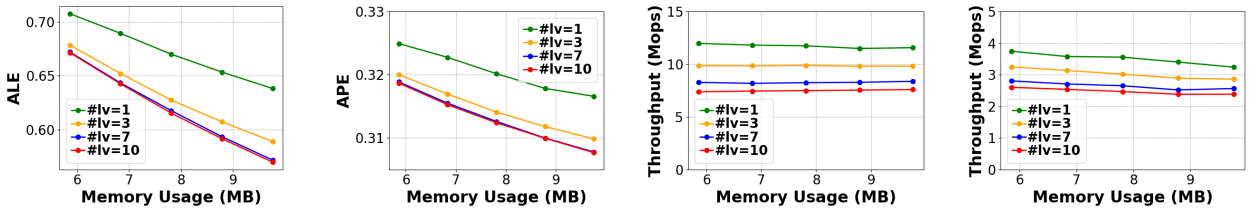
[5]We choose to have three bucket arrays because it is a gold trade-off between accuracy and speed. For more details, please refer to Section 5.2

[6]Experiment results under different settings of p are similar. For more details, please refer to Section 5.2



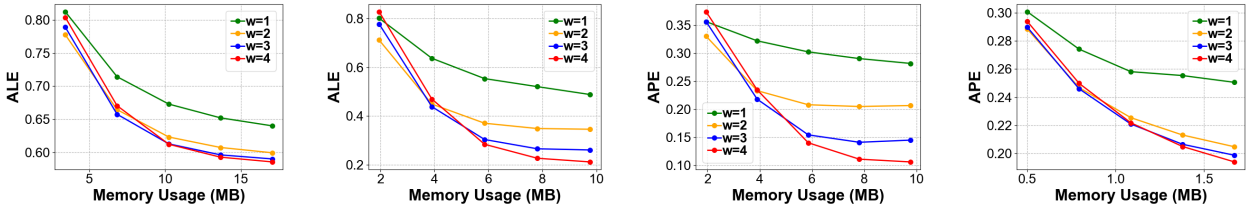
(a) ALE. (b) APE. (c) Insertion throughput. (d) Query throughput.

Figure 6: Effect of the number of levels on accuracy of M4.



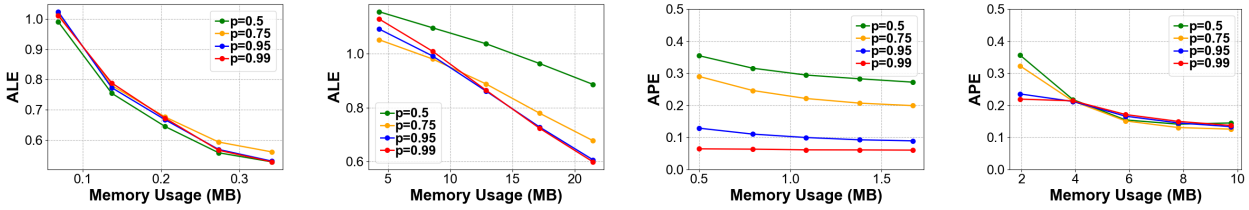
(a) ALE. (b) APE. (c) Insertion throughput. (d) Query throughput.

Figure 7: Effect of the number of levels on accuracy of Strawman.



(a) M4-DDSketch on CAIDA. (b) M4-t-digest on MAWI. (c) M4-t-digest on MAWI. (d) M4-mReqSketch on IMC.

Figure 8: Effect of w on accuracy.



(a) M4-DDSketch on IMC. (b) M4-mReqSketch on CAIDA. (c) M4-mReqSketch on IMC. (d) M4-t-digest on MAWI.

Figure 9: Effect of p on accuracy.

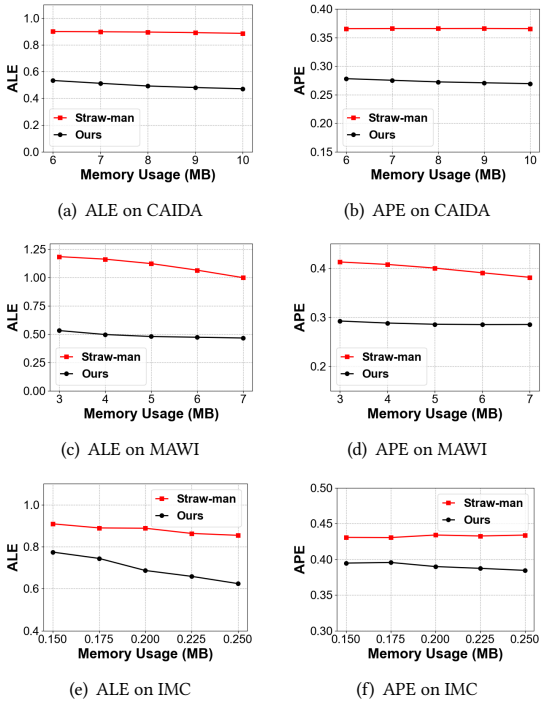
that $w = 3$ and $w = 4$ yield similar accuracy. Since M4 with $w = 3$ runs faster than $w = 4$, we select $w = 3$ as our default setting.

Effect of p. As shown in Figure 9, M4 performs consistently regardless of the value of p. We query $p = 0.5, 0.75, 0.95$, and 0.99 respectively on different M4-*META* on various datasets. Results indicate that the accuracy under different settings of p is similar and follows a consistent trend. Users can set p to an arbitrary value. We default $p = 0.5$ for conducting other experiments.

5.3 Experiments of Huge and Medium Flows on Accuracy

Experiments on M4-DDSketch. As shown in Figure 10, the experimental results show that both ALE and APE of M4-DDSketch are significantly lower than those of the straw-man solution. Specifically on the three real-world datasets, the ALEs of M4-DDSketch are 1.80×, 2.26×, and 1.27× lower on average than those of the straw-man solution. Similarly, the APEs of M4-DDSketch are 1.34×, 1.39×, and 1.11× lower on average than those of the straw-man solution.

Experiments on M4-t-digest. As shown in Figure 11, the experimental results demonstrate that both ALE and APE of M4-t-digest are significantly lower than that of the straw-man solution. Specifically on the three real-world datasets, the ALEs of M4-t-digest are 2.19×, 2.13× and 1.90× lower on average than those of the straw-man solution. Similarly, the APEs of M4-t-digest are 1.99×, 1.78×, and 1.23× lower on average than those of the straw-man solution.

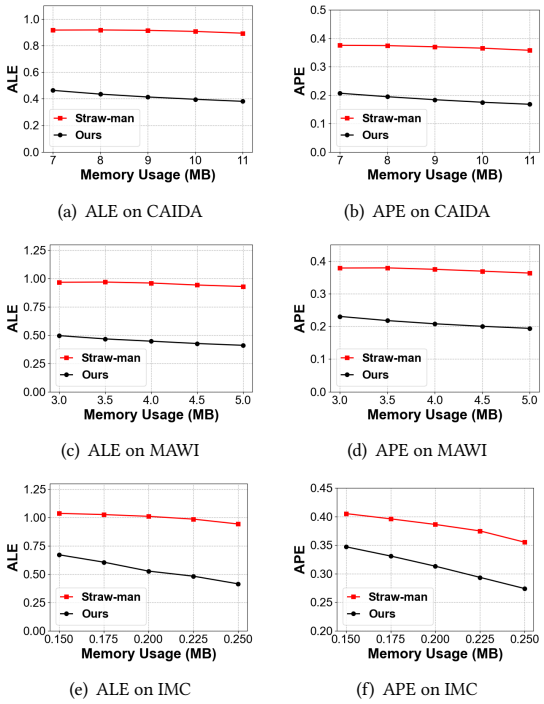


(a) ALE on CAIDA (b) APE on CAIDA

(c) ALE on MAWI (d) APE on MAWI

(e) ALE on IMC (f) APE on IMC

Figure 10: Accuracy of M4-DDSketch.



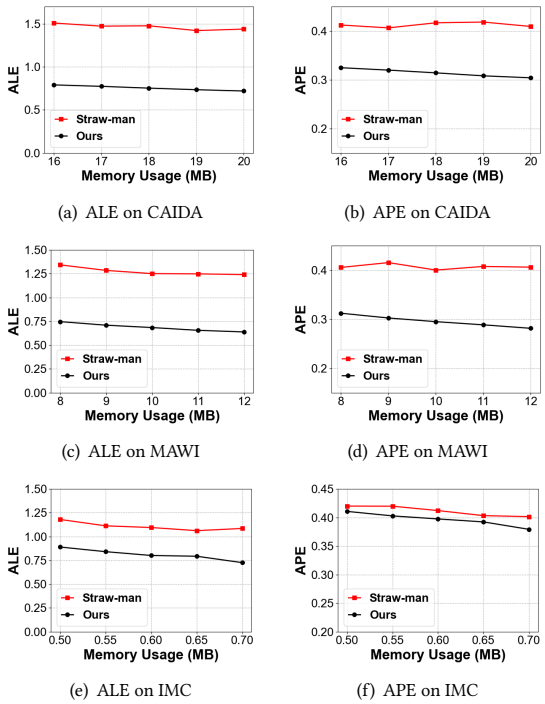
(a) ALE on CAIDA (b) APE on CAIDA

(c) ALE on MAWI (d) APE on MAWI

(e) ALE on IMC (f) APE on IMC

Figure 11: Accuracy of M4-*t*-digest.

Experiments on M4-mReqSketch. As shown in Figure 12, the experimental results demonstrate that both ALE and APE of M4-mReqSketch are significantly lower than those of the straw-man solution. Specifically on the three real-world datasets, the ALEs of M4-mReqSketch are 1.94×, 1.86×, and 1.37× lower on average



(a) ALE on CAIDA (b) APE on CAIDA

(c) ALE on MAWI (d) APE on MAWI

(e) ALE on IMC (f) APE on IMC

Figure 12: Accuracy of M4-mReqSketch.

than those of the straw-man solution. Similarly, the APEs of M4-mReqSketch are 1.31×, 1.38×, and 1.04× lower on average than those of the straw-man solution.

Analysis. The accuracy advantage in Figure 10-12 is established by making better 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.

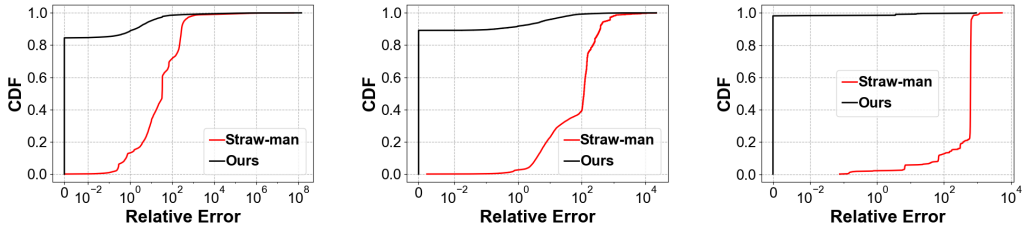
5.4 Experiments of Tiny Flows on Accuracy

As shown in Figure 13, the experimental results demonstrate that for most tiny flows, the maximum *value* can be well estimated by M4, which significantly outperforms the straw-man solution. Specifically on the three real-world datasets, M4 attains an error-free (i.e., $RE = 0$) rate of 84.5%, 89.1%, and 98.2%, while the straw-man solution offers almost no error-free estimates.

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 the straw-man solution.

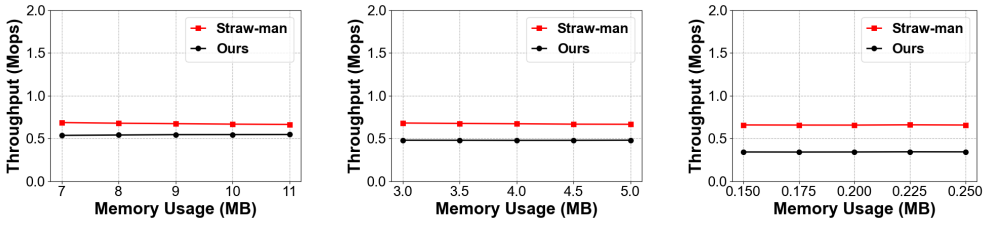
5.5 Experiments on Speed

Insertion Throughput. As shown in Figure 14, the experimental results demonstrate that the insertion throughput of M4 is lower than the straw-man solution. Specifically on the three real-world



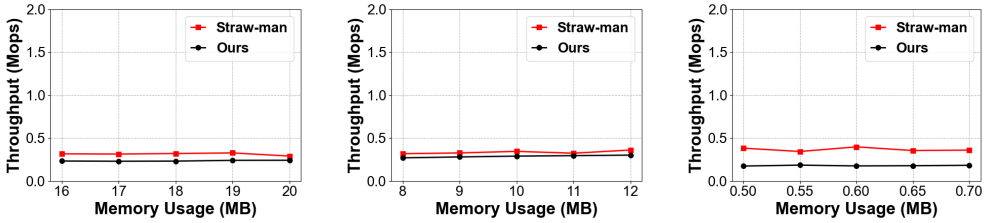
(a) CAIDA (b) MAWI (c) IMC

Figure 13: RE distribution of tiny flows on different datasets.



(a) CAIDA. (b) MAWI (c) IMC

Figure 14: Insertion throughput on different datasets.



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

Figure 15: Query throughput on different datasets.

datasets, the insertion throughput of M4 is 1.24×, 1.40×, and 1.92× lower on average than those of the straw-man solution.

Query Throughput. As shown in Figure 15, the experimental results demonstrate that the query throughput of M4 is also lower than the straw-man solution. Specifically on the three real-world datasets, the query throughput of M4 is 1.33×, 1.17×, and 2.05× lower on average than those of the straw-man solution.

Analysis. The throughput of M4 in insertion and query is slightly lower because we have one more layer than the straw-man solution. Besides, *MINIMUM* and *SUM* operations take extra time.

6 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. The experimental results indicate that M4 outperforms the straw-man solution in estimating the *value* distribution of huge, medium, and tiny flows. For huge and medium flows, the ALE and APE are 2.26× and 1.99× lower than the straw-man solution. For tiny flows, the maximum *value* estimation attains an error-free rate of 98.0%, which is a stark improvement over the virtually nonexistent error-free estimates offered by the straw-man solution. We have made our code publicly available on GitHub [44] to facilitate further research and application in this field.

REFERENCES

[1] D. Nguyen, G. Memik, S.O. Memik, and A. Choudhary. Real-time feature extraction for high speed networks. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 438–443, 2005.

[2] Albert Bifet. Mining big data in real time. *informatica*, 37(1), 2013.

[3] Eduardo Viegas, Altair Santin, Alysson Bessani, and Nuno Neves. Bigflow: Real-time and reliable anomaly-based intrusion detection for high-speed networks. *Future Generation Computer Systems*, 93:473–485, 2019.

[4] M Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwanggil Jeon. Real-time big data stream processing using gpu with spark over hadoop ecosystem. *International Journal of Parallel Programming*, 46:630–646, 2018.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, 2013.

[6] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *Proc. ICDE*, pages 556–567, 2014.

[7] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *Proc. VLDB Endow.*, 7(12):1219–1230, 2014.

[8] Yuxiang Zeng, Yongxin Tong, and Lei Chen. Last-mile delivery made practical: An efficient route planning framework with theoretical guarantees. *Proc. VLDB Endow.*, 13(3):320–333, 2019.

[9] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. Analysis of indexing structures for immutable data. In *Proc. SIGMOD*, pages 925–935, 2020.

[10] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. Face: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.

[11] Bo Wang, Rongqiang Chen, and Lu Tang. Easyquantile: Efficient quantile tracking in the data plane. In *Proceedings of the 7th Asia-Pacific Workshop on Networking*, pages 123–129, 2023.

[12] Ying Zhang, Wenjie Zhang, Jian Pei, Xuemin Lin, Qianlu Lin, and Aiping Li. Consensus-based ranking of multivalued objects: A generalized borda count approach. *IEEE Trans. Knowl. Data Eng.*, 26(1):83–96, 2012.

[13] Zubair Shah, Abdun Naser Mahmood, Zahir Tari, and Albert Y Zomaya. A technique for efficient query estimation over distributed data streams. *IEEE Trans. Parallel Distrib. Syst.*, 28(10):2770–2783, 2017.

[14] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proc. VLDB*, 12(12):2195–2205, 2019.

[15] Ted Dunning. The size of a t-digest. *arXiv preprint arXiv:1903.09921*, 2019.

[16] Graham Cormode, Abhinav Mishra, Joseph Ross, and Pavel Veselỳ. Theory meets practice at the median: a worst case comparison of relative error quantile algorithms. In *Proc. KDD*, pages 2722–2731, 2021.

[17] Ted Dunning. The t-digest: Efficient estimates of distributions. *Software Impacts*, 7:100049, 2021.

[18] Michal Szymaniak, David Presotto, Guillaume Pierre, and Maarten van Steen. Practical large-scale latency estimation. *Computer Networks*, 52(7):1343–1364, 2008.

[19] T.S.E. Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 170–179 vol.1, 2002.

[20] Rui Zhu, Bang Liu, Di Niu, Zongpeng Li, and Hong Vicky Zhao. Network latency estimation for personal devices: A matrix completion approach. *IEEE/ACM Transactions on Networking*, 25(2):724–737, 2017.

[21] Yongjun Liao, Wei Du, Pierre Geurts, and Guy Leduc. Dmfsgd: A decentralized matrix factorization algorithm for network distance prediction. *IEEE/ACM Transactions on Networking*, 21(5):1511–1524, 2013.

[22] Gábor Horváth, Peter Buchholz, and Miklós Telek. A map fitting approach with independent approximation of the inter-arrival time distribution and the lag correlation. In *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 124–133. IEEE, 2005.

[23] Dimitris J Bertsimas and Garrett Van Ryzin. Stochastic and dynamic vehicle routing with general demand and interarrival time distributions. *Advances in Applied Probability*, 25(4):947–978, 1993.

[24] Blake McShane, Moshe Adrian, Eric T Bradlow, and Peter S Fader. Count models based on weibull interarrival times. *Journal of Business & Economic Statistics*, 26(3):369–378, 2008.

[25] Rishi Sinha, Christos Papadopoulos, and John Heidemann. Internet packet size distributions: Some observations. *USC/Information Sciences Institute, Tech. Rep. ISI-TR-2007-643*, pages 1536–1276, 2007.

[26] Ping Du and Shunji Abe. Detecting dos attacks using packet size distribution. In *2007 2nd Bio-Inspired Models of Network, Information and Computing Systems*, pages 93–96. IEEE, 2007.

[27] Chun-Nan Lu, Chun-Ying Huang, Ying-Dar Lin, and Yuan-Cheng Lai. Session level flow classification by packet size distribution and session grouping. *Computer Networks*, 56(1):260–272, 2012.

[28] DJ Parish, K Bharadia, A Larkum, IW Phillips, and MA Oliver. Using packet size distributions to identify real-time networked applications. *IEE Proceedings-Communications*, 150(4):221–227, 2003.

[29] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Performance evaluation of hierarchical ttl-based cache networks. *Computer Networks*, 65:212–231, 2014.

[30] Edith Cohen, Eran Halperin, and Haim Kaplan. Performance aspects of distributed caches using ttl-based consistency. *Theoretical computer science*, 331(1):73–96, 2005.

[31] Biao Wang, Ge Chen, Luoyi Fu, Li Song, and Xinbing Wang. Drimux: Dynamic rumor influence minimization with user experience in social networks. *IEEE Trans. Knowl. Data Eng.*, 29(10):2168–2181, 2017.

[32] Devinder Kaur, Gagangeet Singh Aujla, Neeraj Kumar, Albert Y Zomaya, Charith Perera, and Rajiv Ranjan. Tensor-based big data management scheme for dimensionality reduction problem in smart grid systems: Sdn perspective. *IEEE Trans. Knowl. Data Eng.*, 30(10):1985–1998, 2018.

[33] Jing Li, Weifa Liang, Wenzheng Xu, Zichuan Xu, and Jin Zhao. Maximizing the quality of user experience of using services in edge computing for delay-sensitive iot applications. In *Proc. MSWiM*, pages 113–121, 2020.

[34] Izzat Alsmadi and Dianxiang Xu. Security of software defined networks: A survey. *Comput. Secur.*, 53:79–108, 2015.

[35] Yixin Chen, Jianing Pei, and Defang Li. Detpro: a high-efficiency and low-latency system against ddos attacks in sdn based on decision tree. In *Proc. ICC*, pages 1–6, 2019.

[36] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *Proc. SP*, pages 20–38, 2020.

[37] Muhammad Shahzad and Alex X Liu. Noise can help: Accurate and efficient per-flow latency measurement without packet probing and time stamping. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):207–219, 2014.

[38] Graham Cormode, Zohar Karnin, Edo Liberty, Justin Thaler, and Pavel Veselỳ. Relative error streaming quantiles. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 96–108, 2021.

[39] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

[40] Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*, pages 71–78. IEEE, 2016.

[41] Rana Shahout, Roy Friedman, and Ran Ben Basat. Squad: Combining sketching and sampling is better than either for per-item quantile estimation. *arXiv preprint arXiv:2201.01958*, 2022.

[42] Jintao He, Jiaqi Zhu, and Qun Huang. Histsketch: A compact data structure for accurate per-key distribution monitoring. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023.

[43] Jiarui Guo, Yisen Hong, Yuhan Wu, Yunfei Liu, Tong Yang, and Bin Cui. Sketchpolymer: Estimate per-item tail quantile using one sketch. 2023.

[44] The source codes of ours and other related algorithms. https://github.com/M4Framework/M4.

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

[46] Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[47] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems (TOCS)*, 21(3):270–313, 2003.

[48] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming: 29th International Colloquium, ICALP 2002 Málaga, Spain, July 8–13, 2002 Proceedings 29*, pages 693–703. Springer, 2002.

[49] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking*, 20(5):1622–1634, 2012.

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

[51] Bob jenkins' hash function web page, paper published in dr dobb's journal. http://burtleburtle.net/bob/hash/evahash.html.

[52] The CAIDA Anonymized Internet Traces. https://www.caida.org/catalog/datasets/overview/.

[53] MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.

[54] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proc. IMC*, pages 267–280, 2010.