SketchPolymer: Estimate Per-item Tail Quantile Using One Sketch

Jiarui Guo[*†]
Peking University
ntguojiarui@pku.edu.cn

Yisen Hong[‡*†]
Peking University
eason18@pku.edu.cn

Yuhan Wu[*†]
Peking University
yuhan.wu@pku.edu.cn

Yunfei Liu[§]
Peking University
yunfei_imne@pku.edu.cn

Tong Yang[*†]
Peking University
yangtongemail@gmail.com

Bin Cui[*]
Peking University
bin.cui@pku.edu.cn

ABSTRACT

[1] Estimating the quantile of distribution, especially tail distribution, is an interesting topic in data stream models, and has obtained extensive interest from many researchers. In this paper, we propose a novel sketch, namely SketchPolymer to accurately estimate per-item tail quantile. SketchPolymer uses a technique called **Early Filtration** to filter infrequent items, and another technique called **VSS** to reduce error. Our experimental results show that the accuracy of SketchPolymer is on average 32.67 times better than state-of-the-art techniques. We also implement our SketchPolymer on P4 and FPGA platforms to verify its deployment flexibility. All our codes are available at GitHub [1].

CCS CONCEPTS

• **Theory of computation** → **Sketching and sampling**; • **Information systems** → **Data stream mining**; • **Networks** → *Network measurement*.

KEYWORDS

Tail quantile estimation, Data stream, Sketch, Quantile estimation

ACM Reference Format:
Jiarui Guo, Yisen Hong, Yuhan Wu, Yunfei Liu, Tong Yang, and Bin Cui. 2023. SketchPolymer: Estimate Per-item Tail Quantile Using One Sketch. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23), August 6–10, 2023, Long Beach, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3580305.3599505

[*]National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China
[†]Peng Cheng Laboratory, Shenzhen, China
[‡]School of Software and Microelectronics, Peking University, Beijing, China
[§]School of Integrated Circuits, Peking University, Beijing, China
[1]The first three authors contribute equally. Tong Yang (yangtongemail@gmail.com) is the corresponding author.

KDD '23, August 6–10, 2023, Long Beach, CA, USA

ACM ISBN 979-8-4007-0103-0/23/08…$15.00
https://doi.org/10.1145/3580305.3599505

1 INTRODUCTION

1.1 Background and Motivation

It is vital to estimate the quantile of distribution, especially tail distribution (*e.g.*, 0.95 or 0.99 quantile) in practice. For example, in network scenarios, latency distribution is often heavy-tailed in reality [2–4], and users are usually sensitive to the worst-case latency, so tail latency is usually more representative than other usual latency. However, while most researchers focus on *aggregated* tail quantile estimation [5, 6], few of them have realized the importance of *per-item* tail quantile estimation, which means estimating tail quantile for every distinct item rather than aggregating all items as a whole. In fact, per-item tail quantile can be useful in many situations, and below we show some use cases.

Case 1: Website management. Users usually care about their personal experience on visiting the website. Even if the website achieves low latency for overall requests, the experience of waiting several seconds or even minutes for the website to respond can probably discourage the user from visiting it ever since [7, 8]. As a result, to obtain as many visitors as possible, website administrators shall pay attention to the tail latency of every visitor (*i.e.*, every item).

Case 2: Attack detection. The latency of a single item can suddenly increase due to cyber attacks or offending applications [9, 10]. However, the aggregated tail latency may remain unchanged, as the low latency of millions of packets can cover up this phenomenon. As a result, we can only detect the attack by per-item tail quantile measurement.

Per-item tail quantile estimation is fundamental and critical in these cases. However, few prior work focuses on estimating per-item tail quantile. Some techniques can estimate tail quantile [5, 6], but they are mainly concentrated on estimating aggregated tail quantile, and they do not distinguish different items in data streams. In theoretical computer science, state-of-the-art algorithms for quantile estimation achieve high accuracy in theory [11–16], but they are designed to estimate arbitrary quantile rather than tail quantile, which means they inevitably record much unnecessary information besides tail quantile, incurring a waste of time and memory.

Fortunately, approximate streaming algorithms, namely sketching algorithms, can be applied to solve this challenging problem in data stream models. Sketches can fulfill the need to filter infrequent items [17–19], and they can maintain the full information of frequent items within small memory [20–23], which is important in data stream models. Sketches have already been used to estimate

quantiles in data streams [24–26], and we design a new sketch to estimate per-item tail quantile while avoiding recording as much unnecessary information as possible.

1.2 Our Proposed Solution

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

SketchPolymer includes 4 stages. Filter Stage is used to filter infrequent items. Polymer Stage records the frequency and maximum value of every frequent item, and Splitting Stage and Verification Filter keep the detailed information of these items. The key techniques used in SketchPolymer are named **Early Filtration** and **VSS**. We show these techniques below.

Key Technique I: Early Filtration. Infrequent items account for the majority of data streams [27–29], but their tail quantile cannot be accurately estimated due to their low frequency. As a result, it is important to filter these infrequent items to make room for frequent items. Inspired by the Cold Filter [18], we propose Filter Stage to take on this role in SketchPolymer. For an incoming item e, we first query Filter Stage to check its frequency. If its frequency exceeds a predefined threshold $\mathcal{T}$, we start inserting it into the following stages; Otherwise, we simply insert it into Filter Stage and return. Our experimental results show that allocating a small proportion of memory for Filter Stage can significantly lower the error of query results (see Section 5 for more details).

Key Technique II: Value Splitting and Sharing (VSS). We use logarithm to split all positive numbers into several disjoint intervals. It works as follows: for an incoming item e with value t, we calculate $T = \lfloor \log_a t \rfloor$ as its logarithm value, where a is a predefined parameter for SketchPolymer, and record T in our data structure instead of t. By VSS, we split frequent items into several intervals, and items in the same interval share the same logarithm value. In this way, we convert the quantile estimation problem to the frequency estimation problem, and we record frequency rather than value in SketchPolymer, which is more efficient and accurate and can be solved using CMSketch [20] and Bloom Filter [30].

1.3 Key Contributions

This paper makes the following contributions:

- We propose a novel data structure, namely SketchPolymer, which can automatically separate frequent items from infrequent items and record the information of the former for accurate per-item tail quantile estimation.
- We provide rigorous mathematical analysis for SketchPolymer to theoretically derive its error bound and time complexity.
- We conduct extensive experiments on different datasets. The results show that SketchPolymer outperforms existing algorithms by 32.67 times in terms of error.
- We implement SketchPolymer on various platforms and verify its performance on both software and hardware platforms.

2 PROBLEM STATEMENT AND RELATED WORK

2.1 Problem Statement

The symbols frequently used in this paper and their meanings are shown in Table 1.

Table 1: Symbols frequently used in this paper.

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

Definition 1. ***Data Stream.*** *A data stream S is a series of items $\{e_1, e_2, \cdots, e_n, \cdots\}$ appearing in sequence. In this paper, every item has its value t.*

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

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

2.2 Related Work

2.2.1 *Quantile Estimation.*

Traced back to Munro and Paterson who first proposed the idea of quantile estimation [11], quantile estimation has become attractive to many researchers. Since any accurate quantile estimation requires at least $O(N)$ memory for a multiset with size N , many algorithms focus on returning an ε-approximate estimation, which means that it will return a number in $[r - \varepsilon N, r + \varepsilon N]$, where r is the real rank of the item. Manku, Rajagopalan and Lindsay proposed an algorithm with $O(\frac{1}{\varepsilon}\log^2(\varepsilon N))$ space complexity [12], and GK improved this complexity to $O(\frac{1}{\varepsilon}\log(\varepsilon N))$ [13]. However, all these algorithms have to know N in advance, and later, Felber and Ostrovsky improved this bound to $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ [14].

Other algorithms for quantile estimation use randomization, which probably saves memory but has a small probability to fall outside the interval. KLL algorithm [15] used this technique to obtain $O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta})$ memory complexity with failure rate at most δ for the first time, and further, KLL$^{\pm}$ [16] updated this bound to $O(\frac{1}{\varepsilon}\log^2\log\frac{1}{\varepsilon\delta})$ and supported delete operations.

DDSketch [25] is recently designed for estimating w-quantile distribution in data streams. It divides all positive numbers into several intervals by logarithm and uses buckets to record the frequency in each interval. Moreover, when too many intervals have frequency not equal to 0, DDSketch will merge adjacent buckets to save memory. SQUAD [31] is another algorithm which focuses on estimating per-item quantiles for heavy-hitters. It applies Reservoir

Sampling [32] to sample items from the data stream, and constructs sketches using Space Saving [33] to keep track of these items.

However, these algorithms are generally designed for estimating *full* quantiles, while *tail* quantiles are more important in many situations. Although they can be used to estimate per-item tail quantile, these algorithms keep the information of small values as well, which is unnecessary for tail quantile estimation.

2.2.2 *Frequency Estimation.*

CMSketch [20] is the simplest sketch for frequency estimation. It is composed of d arrays and each array has the same number of counters. For every incoming item, CMSketch uses a hash function to map it into a counter in every array, and increments the counter by 1. When querying an item, CMSketch similarly locates these d counters and returns the minimum of these values.

2.2.3 *Membership Query.*

Bloom Filter [30] is a compact data structure with high memory efficiency. It consists of arrays of bit groups, and is often used to judge whether an item belongs to a given set. For every coming item, Bloom Filter uses hash functions to map it into several bits and sets these bits to 1. When querying an item, Bloom Filter uses the same hash functions to check whether all these bits are 1.

3 SKETCHPOLYMER ALGORITHM

In this section, we first propose the baseline solution for per-item tail quantile estimation. Then we introduce the idea of VSS and propose the initial version of SketchPolymer. We optimize the SketchPolymer from two aspects and then present the final version of SketchPolymer.

3.1 Baseline Solution

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

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

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

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

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

Although the baseline solution can be used to estimate per-item tail quantile, it is inaccurate and memory-consuming, as it fails to filter infrequent items; Also, it keeps true values in the data structure, which is a waste of space in practice.

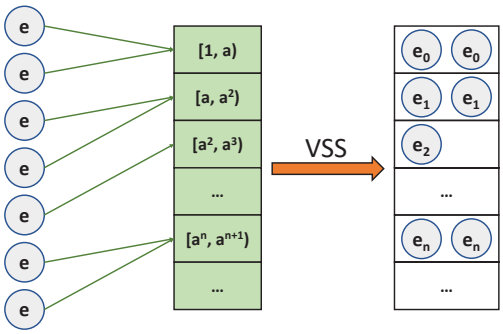


Figure 1: Idea of VSS

3.2 Idea of Value Splitting and Sharing

The value of every frequent item has to be recorded to accurately estimate per-item tail quantile. However, the naive idea of simply keeping its real value may not work, as the order of magnitude of value can vary from item to item. To avoid this problem, we propose **Value Splitting and Sharing (VSS)** technique (See in Figure 1): We use logarithm to split the value of all items into several intervals. We first choose a positive number a as the base of logarithm. To reduce error, a is usually greater than but very close to 1. Then, for every item e with its true value t, we calculate $T = \lfloor \log_a t \rfloor$ as its logarithm value. In this way, every frequent item is separated into several intervals, and items in the same interval share the same logarithm value. The item $e' = (e, T)$ can be viewed as a new item to be inserted into Splitting Stage and Verification Filter. After query process, suppose SketchPolymer returns integer T as the w-quantile of e, we use exponential to get the result a^T. In this way, SketchPolymer maintains the information of these items to the greatest extent without using too much memory.

3.3 The SketchPolymer Algorithm

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

3.3.1 *Idea of Early Filtration.*

A naive idea for per-item tail quantile estimation is to record the value of all items whether they are frequent items or infrequent items. However, tail quantile of infrequent items cannot be accurately estimated due to their low frequency. Also, the distribution of real datasets is generally skew [27–29], which means the majority of items in the data stream are infrequent items. To make full use of this prior distribution, we apply **Early Filtration** technique, which originates from the Cold Filter [18]: Filter Stage keeps the frequency of every distinct item, and only items with frequency exceeding the threshold are allowed to enter the following stages.

Filter Stage Data Structure: Filter Stage is a CMSketch consisting of $d^{(1)}$ arrays: $C_1^{(1)}, \cdots, C_{d^{(1)}}^{(1)}$. Each array consists of $n^{(1)}$ counters. There are $d^{(1)}$ hash functions $h_1^{(1)}, \cdots, h_{d^{(1)}}^{(1)}$ associating with $d^{(1)}$ arrays respectively. Each counter records the frequency of e.

Filter Stage Operation (Algorithm 1-2): To insert an item e, Filter Stage uses $d^{(1)}$ hash functions to map e into $d^{(1)}$ counters $C_1^{(1)}[h_1^{(1)}(e)], \cdots, C_{d^{(1)}}^{(1)}[h_{d^{(1)}}^{(1)}(e)]$ and increments these counters by 1. When querying e, Filter Stage will use the same hash functions to check $C_1^{(1)}[h_1^{(1)}(e)], \cdots, C_{d^{(1)}}^{(1)}[h_{d^{(1)}}^{(1)}(e)]$. Similarly to CMSketch, Filter Stage will return the minimum of these values as the frequency of e.

Algorithm 1: Filter Stage Insertion Procedure

Input: an item e;
for $1 \le i \le d^{(1)}$ **do**
  $C_i^{(1)}[h_i^{(1)}(e)] \leftarrow C_i^{(1)}[h_i^{(1)}(e)] + 1$;

Algorithm 2: Filter Stage Query Procedure

Input: an item e;
$f \leftarrow +\infty$
for $1 \le i \le d^{(1)}$ **do**
  $f \leftarrow \min\{f, C_i^{(1)}[h_i(e)]\}$;
return f;

3.3.2 Tail Quantile Estimation.

Rationale: Polymer Stage and Splitting Stage are designed for estimating tail quantile of frequent items. Polymer Stage records the frequency and maximum value of every frequent items, and Splitting Stage records the frequency of every item after VSS.

Algorithm 3: Polymer Stage Insertion Procedure

Input: an item (e, T);
for $1 \le i \le d^{(2)}$ **do**
  $C_i^{(2)}[h_i^{(2)}(e)].f \leftarrow C_i^{(2)}[h_i^{(2)}(e)].f + 1$;
  $C_i^{(2)}[h_i^{(2)}(e)].t \leftarrow \max\{C_i^{(2)}[h_i^{(2)}(e)].t, T\}$;

Algorithm 4: Polymer Stage Query Procedure

Input: an item e;
$f \leftarrow +\infty$
$t \leftarrow +\infty$
for $1 \le i \le d^{(2)}$ **do**
  $f \leftarrow \min\{f, C_i^{(2)}[h_i^{(2)}(e)].f\}$;
  $t \leftarrow \min\{t, C_i^{(2)}[h_i^{(2)}(e)].t\}$;
return f, t;

Polymer Stage Data Structure: Polymer Stage consists of $d^{(2)}$ arrays: $C_1^{(2)}, \cdots, C_{d^{(2)}}^{(2)}$. Each array consists of $n^{(2)}$ buckets and $d^{(2)}$ hash functions are associated with these arrays. What is different from traditional CMSketch is that each bucket has two fields: frequency field and value field. Frequency field is similar to Filter Stage, and value field records the **maximum** logarithm value.

Polymer Stage Operation (Algorithm 3-4): To insert an item e with logarithm value $T = \lfloor \log_a t \rfloor$, Polymer Stage uses $d^{(2)}$ hash functions to map e into $d^{(2)}$ buckets $C_1^{(2)}[h_1^{(2)}(e)], \cdots, C_{d^{(2)}}^{(2)}[h_{d^{(2)}}^{(2)}(e)]$. The frequency field of these buckets will be incremented by 1, and the value field of these buckets will be set to the maximum of its initial value and T. When querying e, Polymer Stage will similarly use hash functions to find these $d^{(2)}$ buckets. The frequency and maximum logarithm value will all be set to the minimum of relating fields.

Splitting Stage Data Structure: Splitting Stage is a CMSketch consisting of $d^{(3)}$ arrays: $C_1^{(3)}, \cdots, C_{d^{(3)}}^{(3)}$. Each array consists of $n^{(3)}$ counters and $d^{(3)}$ hash functions are associated with the arrays. However, in Splitting Stage, every hash function takes both the item e and the logarithm value T as arguments. Each counter records the frequency of every item with logarithm value equal to T.

Splitting Stage Operation (Algorithm 5-6): To insert an item e with logarithm value T, Splitting Stage maps e into $d^{(3)}$ counters $C_1^{(3)}[h_1^{(3)}(e,T)], \cdots, C_{d^{(3)}}^{(3)}[h_{d^{(3)}}^{(3)}(e,T)]$ and increments these counters by 1. To query e with logarithm value T, Splitting Stage will again check these $d^{(3)}$ counters and return the minimum of these values.

Algorithm 5: Splitting Stage Insertion Procedure

Input: an item (e, T);
for $1 \le i \le d^{(3)}$ **do**
  $C_i^{(3)}[h_i^{(3)}(e,T)] \leftarrow C_i^{(3)}[h_i^{(3)}(e,T)] + 1$;

Algorithm 6: Splitting Stage Query Procedure

Input: an item (e, T);
$f \leftarrow +\infty$;
for $1 \le i \le d^{(3)}$ **do**
  $f \leftarrow \min\{C_i^{(3)}[h_i^{(3)}(e,T)], f\}$;
return f;

3.4 Memory Optimization: Counter Truncation

After VSS, frequent items are usually split into many items according to their logarithm value. Hence, it will be memory-inefficient to still use 32 bits for a counter in Splitting Stage. Also, we are mainly interested in the tail distribution, and the frequency of these items are generally small. To tackle this problem, we propose **Counter Truncation**: Instead of using 32-bit counters, we only allocate 8-bit counters for Splitting Stage. When we are about to increment a counter, we first check whether the counter has achieve its maximum value (255 here). If so, we keep its value and do nothing. Our experiments show that instead of using 32-bit counters, using more small counters improves the accuracy of SketchPolymer (See in Section 5).

3.5 Accuracy Optimization: Overestimation Avoidance

The three stages above can fulfill the requirement of estimating per-item tail quantile. However, Splitting Stage is based on CMSketch, which suffers from overestimation error in practice. Inspired by the Bloom Filter, we propose Verification Filter in Stage 4 to compensate for Splitting Stage. When querying item e with logarithm value T, SketchPolymer will first check Verification Filter. If

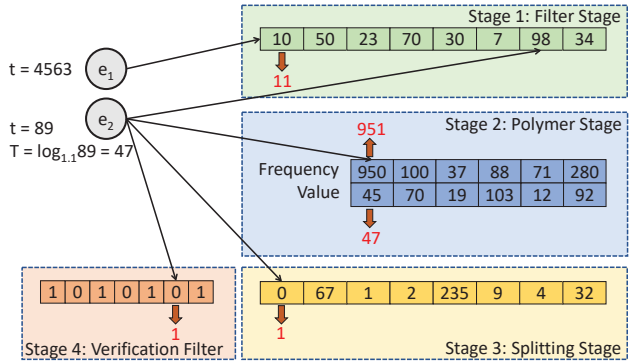


Figure 2: Example of Insertion

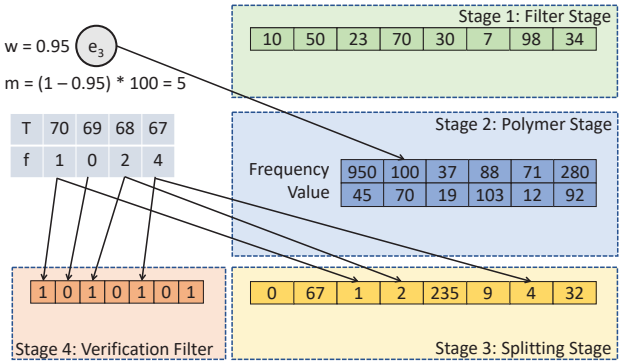


Figure 3: Example of Query

Verification Filter reports false for item (e, T), it will be considered to not have come before, so SketchPolymer will not query Splitting Stage and just return 0 as the frequency of item (e, T).

Verification Filter Data Structure: Verification Filter is a Bloom Filter consisting of $d^{(4)}$ arrays $C_1^{(4)}, \cdots, C_{d^{(4)}}^{(4)}$. Each array has $n^{(4)}$ bits. Similar to Splitting Stage, there are $d^{(4)}$ hash functions and each hash function takes the item e and its logarithm value T as arguments.

Verification Filter Operation (Algorithm 7-8): To insert an item e with logarithm value T, Verification Filter maps e into $d^{(4)}$ bits $C_1^{(4)}[h_1^{(4)}(e,T)], \cdots, C_{d^{(4)}}^{(4)}[h_{d^{(4)}}^{(4)}(e,T)]$ and sets all these bits to 1. When querying (e, T), Verification Filter will return the AND of these $d^{(4)}$ bits.

Algorithm 7: Verification Filter Insertion Procedure

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

Algorithm 8: Verification Filter Query Procedure

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

3.6 Our Final Version

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

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

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

Algorithm 9: SketchPolymer Insertion Procedure

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

Algorithm 10: SketchPolymer Query Procedure

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

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

951), and the value field is set to the maximum value of its initial value 45 and T. Finally, we map $(e_2, 47)$ into a counter in Splitting Stage and increment it by 1 and map it into a bit in Verification Filter and set it to 1. Figure 3 shows a running example of query procedure of SketchPolymer. To query the 0.95-quantile of e_3, SketchPolymer first queries Polymer Stage to get its frequency 100 and maximum logarithm value 70. Next we calculate $m = (1 - 0.95) \times 100 = 5$, which shows that 5 of e_3 have value greater than 0.95-quantile. Then we use $(e_3, 70)$ to query Splitting Stage and Verification Filter. Since Verification Filter returns true for $(e_3, 70)$, we reduce the query result of Splitting Stage (here is 1) from m, and m is now 4. Then we query Verification Filter with $(e_3, 69)$. Since Verification Filter returns false for these arguments, we do nothing and start query with $(e_3, 68)$. Verification Filter returns true and Splitting Stage returns 2 for 68, so m is reduced by 2 and is now 2. Finally we query Splitting Stage and Verification Filter with $(e_3, 67)$. As Verification Filter returns true and Splitting Stage returns 4, m will be reduced by 4 and will turn negative after this operation. Consequently, we stop recursively query and return $1.1^{67} \approx 593.35$ as the 0.95-quantile of e_3.

4 MATHEMATICAL ANALYSIS

In this section, we first provide error bounds for SketchPolymer. We derive the error bound of Polymer Stage and Splitting Stage in the first step, then we give an error bound for SketchPolymer. Finally we analyze the time complexity of SketchPolymer. Due to space constraint, we only list the conclusions in this section. Detailed proofs can be seen in Appendix A.

4.1 Error Bound

In this section, we assume that e_j is a frequent item, *i.e.*, e_j succeeds to enter Polymer Stage, Splitting Stage and Verification Filter in SketchPolymer. We obtain the error bound of SketchPolymer in the following theorems:

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

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

and

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

Proof. See in Appendix A.1. □

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

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

Proof. See in Appendix A.2. □

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

Proof. See in Appendix A.3. □

4.2 Time Complexity

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

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

5 EXPERIMENTAL RESULTS

In this section, we provide experimental results with SketchPolymer. First, we describe the experimental setup in Section 5.1. Then, we show how parameter settings affect SketchPolymer performance in Section 5.2. We compare the performance of SketchPolymer and other algorithms on different datasets in Section 5.3. Finally, we analyze the effects of two optimizations in Section 5.4.

5.1 Experimental Setup

Implementation: We implement SketchPolymer and all other algorithms in C++. In all experiments, we use Bob Hash [34] with different hash seeds to implement the hash functions.

Computation Platform: We conducted all the experiments on a server with one 18-core processor (36 threads, Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) and 128 GB DRAM memory. The processor has 64KB L1 cache, 1MB L2 cache for each core, and 24.75MB L3 cache shared by all cores.

Metrics:

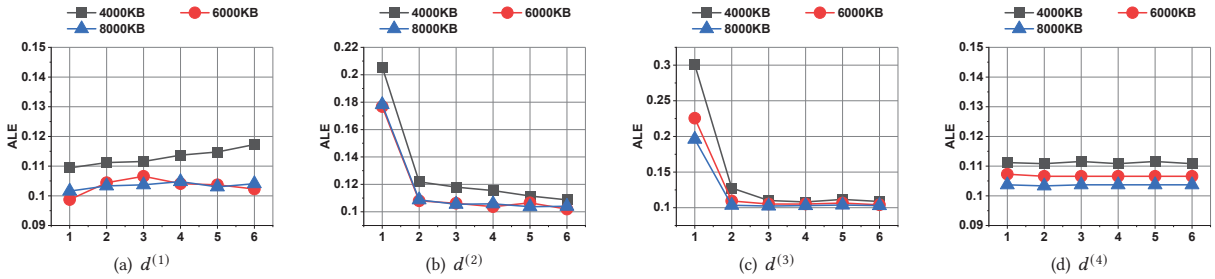
1) Average Logarithm Error (ALE): Since the orders of magnitude of value can vary significantly, it is unreasonable to simply measure the error by absolute value. Suppose $t_1, t_2, \cdots, t_n$ be the true quantile of all items, and $\hat{t}_1, \hat{t}_2, \cdots, \hat{t}_n$ be the estimated quantile, the average logarithm error (ALE) is defined as $\frac{1}{n}\sum_{i=1}^{n} |\log_2 t_i - \log_2 \hat{t}_i| = \frac{1}{n}\sum_{i=1}^{n} |\log_2 \frac{t_i}{\hat{t}_i}|$.

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

Datasets:

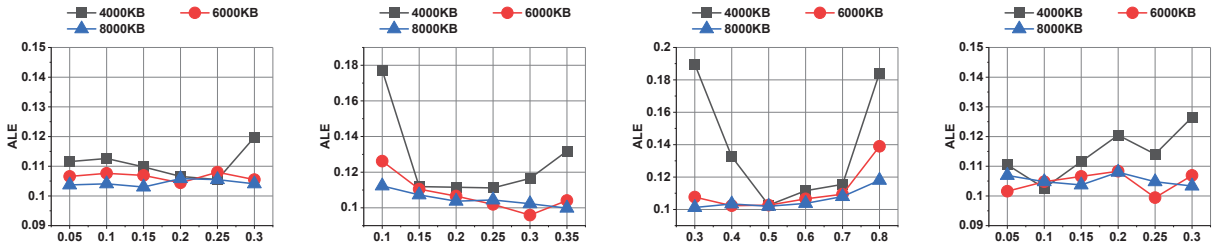
1) IP Trace: The IP Trace is streams of anonymous IP traces collected from 2016 by CAIDA [35]. We regard the interval of two consecutive packets as its value. We use 20 million items.

2) Seattle Dataset: The Seattle Dataset [36, 37] consists of round trip times (RTTs) between several nodes in the Seattle network. We treat RTTs between the same two nodes as the same item, and regard RTT as its value.



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

Figure 4: Effects of Numbers of Hash Functions



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

Figure 5: Effects of Memory Allocation Ratio

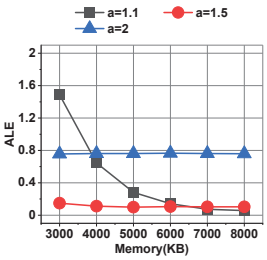


Figure 6: Effects of a

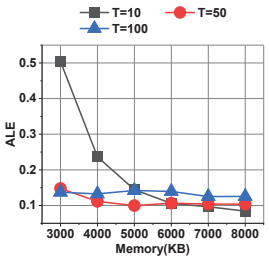


Figure 7: Effects of $\mathcal{T}$

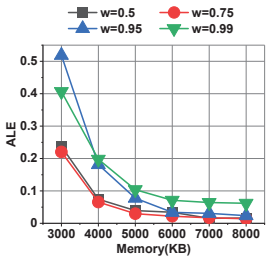


Figure 8: Effects of w

3) Web Latency Dataset: The Web Latency Dataset [38] is collected by Webget [39] on 182 probes distributed globally. We regard fetch time of each request as value.

5.2 Experiments on Parameter Settings

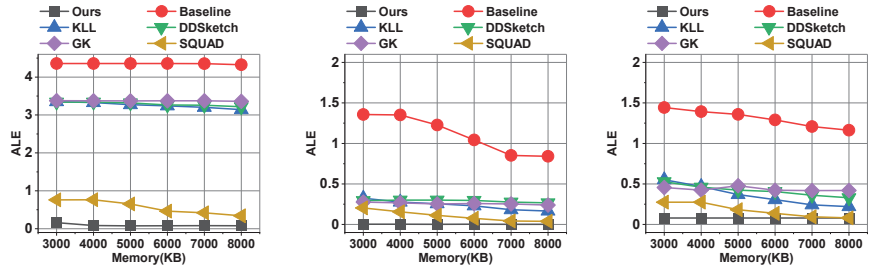
In this section, we measure the effects of some key parameters of SketchPolymer, namely, the number of hash functions in each stage, the memory allocation ratio, the choice of the base of logarithm a, the threshold $\mathcal{T}$, and the query quantile w. We use CAIDA dataset in these experiments, and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

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

Effects of Memory Allocation Ratio (Figure 5(a)-5(d)): *The experimental results show that most space shall be allocated to Splitting Stage.* In each experiment, we vary the ratio of one stage in a certain range, and keep the relative ratio of the other 3 stages unchanged. The results show that a small ratio of memory for Filter Stage, Polymer Stage and Verification Filter will be enough. Since Splitting Stage records the frequency of all items after VSS, we allocate 5% memory for Filter Stage, 30% memory for Polymer Stage, 50% memory for Splitting Stage, and 15% memory for Verification Filter in our experiments.

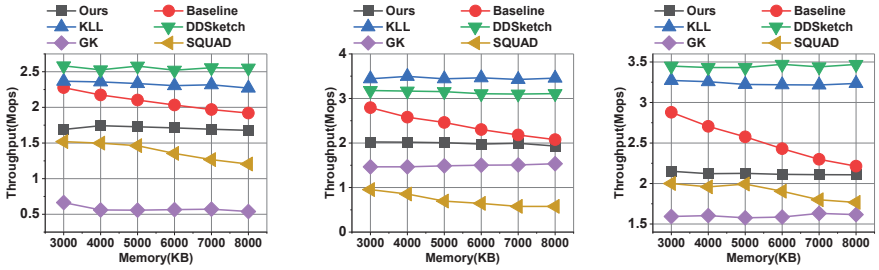
Effects of a (Figure 6): *The experimental results show that the choice of a is generally a trade-off between memory and accuracy.* We set a to 1.1, 1.5 and 2 respectively in each experiment, and results show that a larger a performs better within a smaller memory. However, if user allocates sufficient space to SketchPolymer, the ALE will be minimized when a is close to 1. To make balance between accuracy and memory, we set $a = 1.5$ in other experiments.

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



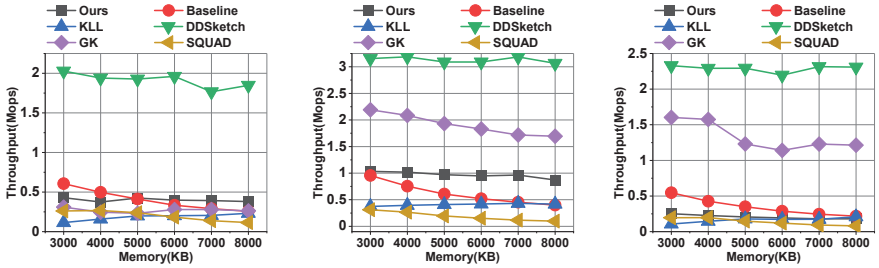
(a) CAIDA (b) Seattle (c) Web

Figure 9: ALE on Different Datasets



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

Figure 10: Insertion Throughput on Different Datasets



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

Figure 11: Query Throughput on Different Datasets

Effects of w (Figure 8): *The experimental results show that SketchPolymer performs well whatever quantile we set.* We query 0.5, 0.75, 0.95 and 0.99 quantile of frequent items respectively, and results show that ALE is close to 0 when memory is greater than 5000KB. In fact, users can set w to any quantile they are interested in, and we set $w = 0.95$ in other experiments.

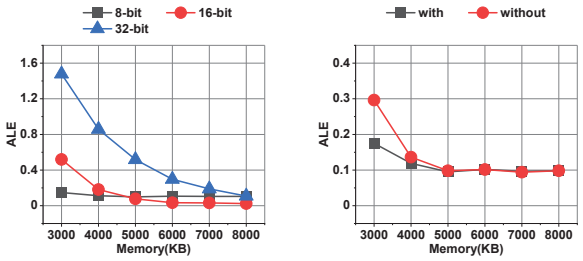
5.3 Comparison with Baseline Solution and Prior Work

In this section, we compare the performance of SketchPolymer with baseline solution and prior work (GK, KLL, DDSketch, SQUAD) on different datasets. Since some of these techniques (GK, KLL, DDSketch) are not designed for per-item quantile estimation, we allocate several buckets for these algorithms, which is similar to baseline solution: Before insertion procedure, we use a hash function to map the item e into a bucket, and insert e in this bucket. When querying an item, we use the same function to locate a bucket and return the quantile of all values in this bucket. Experimental results (Figure 9(a)-11(c)) show that SketchPolymer largely outperforms baseline solution and prior work on per-item tail quantile estimation. On CAIDA dataset, SketchPolymer performs 32.67 times better than other algorithms, and the ALE of SketchPolymer is less than 0.1 on Seattle and Web dataset. Although the insertion and query throughput of SketchPolymer is not the highest, it is close to baseline solution and significantly higher than SQUAD.

5.4 Analysis on Two Optimizations

In this section, we measure the effects of two optimizations, *i.e.*, memory optimization and accuracy optimization. We compare the performance of SketchPolymer with and without two optimizations respectively. We use CAIDA dataset in these experiments, and ALE to measure these effects.

Effects of Memory Optimization (Figure 12(a)): *The experimental results show that 8-bit counters for Splitting Stage is enough.* We compare the performance of using 8-bit, 16-bit and 32-bit counters in Splitting Stage. The results show that the performance of using



(a) Memory (b) Accuracy

Figure 12: Effects of Two Optimizations

three kinds of counters are close with more than 8000KB memory. However, the ALE of 8-bit counters is smaller than 0.2 within 3000KB memory, which is significantly lower than 16-bit and 32-bit counters.

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

6 P4 IMPLEMENTATION

We have fully built a P4 prototype of SketchPolymer on the Tofino switch [40]. Since the Tofino switch processes packets in a pipeline manner, SketchPolymer cannot support more than one hash function in Filter Stage, so we set $d^{(1)} = 1$. Also, the Tofino switch does not support logarithmic operations, so we choose $a = 2$ as the base of logarithm and SketchPolymer can calculate the logarithm value by prefix matching. We implement SketchPolymer using several registers and Stateful ALUs. We list the utilization of various hardware resources on the Tofino switch in Table 2 when $d^{(2)} = d^{(3)} = d^{(4)} = 1$, $n^{(1)} = n^{(2)} = 2^{15}$, $n^{(3)} = 2^{18}$ and $n^{(4)} = 2^{19}$.

Table 2: Hardware Resources Used by SketchPolymer

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

7 FPGA IMPLEMENTATION

We have implemented the SketchPolymer on the Altera FPGA. The model 5SEEBF45I2 in Stratix V family is used as the core chip. The architecture design diagram is illustrated in Figure 13. Two hash functions are used, and the results are stored in two separate RAMs. More hash functions can be supported if needed. FPGA does not support logarithmic operations, so $a = 2$ was chosen to simplify calculation.

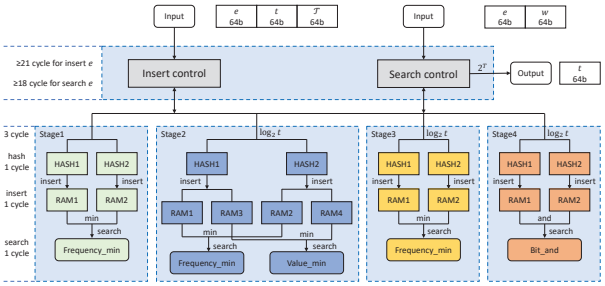


Figure 13: FPGA Implementation Architecture

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

Table 3: SketchPolymer Performance on FPGA Platform

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

8 CONCLUSION

Tail quantile estimation is important in many scenarios. In this paper, we propose SketchPolymer to estimate per-item tail quantile. SketchPolymer uses **Early Filtration** to filter infrequent items, and **VSS** to estimate tail quantile of frequent items efficiently. We implement SketchPolymer on three platforms: CPU, P4 switches and FPGA. Both theoretical and experimental results show that SketchPolymer is much more fast and accurate compared to the state-of-the-art. We have released our codes on GitHub [1].

ACKNOWLEDGEMENT

We would like to thank Kaicheng Yang and Wenrui Liu for their assistance on P4 implementation of SketchPolymer. We would like to thank Yuanpeng Li and Ruijie Miao for their valuable advice on the writing of our paper. We would like to thank the anonymous reviewers for their constructive comments. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, and National Natural Science Foundation of China (NSFC) (No. U20A20179).

REFERENCES

[1] The source codes related to SketchPolymer. https://github.com/SketchPolymer/SketchPolymer-code.

[2] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling tcp latency. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 3, pages 1742–1751. IEEE, 2000.

[3] Jörg Liebeherr, Almut Burchard, and Florin Ciucu. Delay bounds in communication networks with heavy-tailed and self-similar traffic. *IEEE Transactions on Information Theory*, 58(2):1010–1024, 2012.

[4] Pu Wang and Ian F Akyildiz. On the origins of heavy-tailed delay in dynamic spectrum access networks. *IEEE Transactions on Mobile Computing*, 11(2):204–217, 2011.

[5] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 7–16, 2015.

[6] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. Scalable tail latency estimation for data center networks. *arXiv preprint arXiv:2205.01234*, 2022.

[7] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[8] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019.

[9] Myungjin Lee, Nick Duffield, and Ramana Rao Kompella. Not all microseconds are equal: Fine-grained per-flow measurements with reference latency interpolation. In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 27–38, 2010.

[10] Muhammad Shahzad and Alex X Liu. Accurate and efficient per-flow latency measurement without probing and time stamping. *IEEE/ACM Transactions on Networking*, 24(6):3477–3492, 2016.

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

[12] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record*, 27(2):426–435, 1998.

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

[14] David Felber and Rafail Ostrovsky. A randomized online quantile summary in $O(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon})$ words. *Theory of Computing*, 13(1):1–17, 2017.

[15] 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, 2016.

[16] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. Kll±approximate quantile sketches over dynamic datasets. *Proceedings of the VLDB Endowment*, 14(7):1215–1227, 2021.

[17] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[18] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 741–756, 2018.

[19] Kaicheng Yang, Yuanpeng Li, Zirui Liu, Tong Yang, Yu Zhou, Jintao He, Tong Zhao, Zhengyi Jia, Yongqiang Yang, et al. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2021.

[20] 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.

[21] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[22] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proceedings of the VLDB Endowment*, 15(7):1426–1438, 2022.

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

[24] Ge Luo, Lu Wang, Ke Yi, and Graham Cormode. Quantiles over data streams: experimental comparisons, new analyses, and further improvements. *The VLDB Journal*, 25(4):449–472, 2016.

[25] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12).

[26] 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.

[27] Reinaldo Boris Arellano-Valle, Heleno Bolfarine, and Victor H Lachos. Skew-normal linear mixed models. *Journal of data science*, 3(4):415–438, 2005.

[28] Kai Cheng, Limin Xiang, and Mizuho Iwaihara. Time-decaying bloom filters for data streams with skewed distributions. In *15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA'05)*, pages 63–69. IEEE, 2005.

[29] Jing Gao, Wei Fan, Jiawei Han, and Philip S Yu. A general framework for mining concept-drifting data streams with skewed distributions. In *Proceedings of the 2007 siam international conference on data mining*, pages 3–14. SIAM, 2007.

[30] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[31] 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.

[32] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[33] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*, pages 398–412. Springer, 2005.

[34] The source code of Bob Hash. http://burtleburtle.net/bob/hash/evahash.html.

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

[36] Justin Cappos, Ivan Beschastnikh, Arvind Krishnamurthy, and Tom Anderson. Seattle: a platform for educational cloud computing. In *Proceedings of the 40th ACM technical symposium on Computer science education*, pages 111–115, 2009.

[37] 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, 2016.

[38] Alemnew Sheferaw Asrese, Steffie Jacob Eravuchira, Vaibhav Bajpai, Pasi Sarolahti, and Jörg Ott. Measuring web latency and rendering performance: Method, tools & longitudinal dataset. https://doi.org/10.5281/zenodo.2547512, January 2019.

[39] Alemnew Sheferaw Asrese, Steffie Jacob Eravuchira, Vaibhav Bajpai, Pasi Sarolahti, and Jörg Ott. Measuring web latency and rendering performance: Method, tools, and longitudinal dataset. *IEEE Transactions on Network and Service Management*, 16(2):535–549, 2019.

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

A PROOFS IN SECTION 4

A.1 Proof of Theorem 1

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

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

Due to the independence of hash functions, we get

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

Let us define another variable

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

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

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

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

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

Hence Equation 1 holds.

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

$$\begin{aligned} \mathbb{P}(\hat{T}_j = T_j) &\ge \frac{f_j - \mathcal{T}}{f_j} \times \mathbb{P}(\exists i, C_i^{(2)}[h_i^{(2)}(e_j)].T = T_j) \\ &= \frac{f_j - \mathcal{T}}{f_j} \times \left[1 - \mathbb{P}(\forall i, C_i^{(2)}[h_i^{(2)}(e_j)].T > T_j)\right] \\ &\ge \frac{f_j - \mathcal{T}}{f_j} \left[1 - d^{(2)}\mathbb{P}(C_i^{(2)}[h_i^{(2)}(e_j)].T > T_j)\right], \end{aligned}$$

where

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

Hence,

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

so Equation 2 also holds. □

A.2 Proof of Theorem 2

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

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

and

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

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

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

To prove the second part of the theorem, it is worth noticing that the only explanation to $\hat{f}_{j,T} < f_{j,T}$ is that items with logarithm value T come before e_j enters Splitting Stage, so the information *w.r.t.* (e_j, T) is lost. We define a variable X as the number of items with logarithm value T which come before e_j enters Splitting Stage, and suppose f_j is large enough, then X can be approximated to obey binomial distribution: $X \sim B(\mathcal{T}, \frac{f_{j,T}}{f_j})$. So $\mu = \mathbb{E}X = \frac{\mathcal{T}f_{j,T}}{f_j}$. Let $\delta = \frac{f_j}{\mathcal{T}f_{j,T}}\left(\varepsilon - \frac{\mathcal{T}f_{j,T}}{f_j}\right)$. Applying Chernoff Bound, we get

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

Applying both inequality, we get

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

So Equation 3 holds. □

A.3 Proof of Theorem 3

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

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

Equation 3 can also be simplified as

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

and we can prove

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

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

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

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

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

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

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

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

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

Hence

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

Applying Equation 5, we get

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

Finally, we get

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

which finishes our proof. □