# Scalable Overspeed Item Detection in Streams

Yuhan Wu*‡, Hanbo Wu*, Chengjun Jia†, Bo Peng*, Ziyun Zhang*,
Tong Yang*‡, Peiqing Chen§, Kaicheng Yang*, and Bin Cui*
*Peking University, Beijing, China, †Tsinghua University, Beijing, China,
‡Peng Cheng Laboratory, Shenzhen, China, §University of Maryland College Park, Maryland, United States

***Abstract*—In data stream mining, monitoring high-speed users and segregating their excessive use, known as "Overspeed items," is crucial for preventing system overload and maintaining fairness in messaging and network systems. Current approaches, however, face scalability challenges with large user bases, primarily due to increasing memory requirements proportional to user numbers. We have pinpointed the inefficiency in allocating memory for all users, recognizing that only a small fraction exhibit overspeed behavior at any given time. Addressing this, we employed the sketching technique, a type of approximate algorithm, and designed the first sketch algorithm for finding Overspeed items, named SpeedSketch: (1) Scalability. SpeedSketch can scale user numbers (saving memory space) to a factor of 6430 while maintaining a low average error rate of 0.1% in real-world datasets. (2) Accuracy. In theory, SpeedSketch stands out as the only sketch algorithm offering a per-user relative error bound. (3) Speed. SpeedSketch is implemented on a high-speed programmable switch with a throughput capacity of 4.8 billion items per second. All codes are available on GitHub for reference.**

## I. Introduction

In data streams, monitoring high-speed users, such as those who use substantial network bandwidth [1] or exceed the processing capability by sending an excessive number of service requests quickly [2], is crucial for preventing system overload and maintaining fairness [3]. In this context, detecting their excessive usage (requests), labeled as "Overspeed items", is of significant importance. Overspeed items occur when a user's items, such as packets or message requests, arrive faster than it can be processed. However, the challenge in defining overspeed items arises from the fact that items arrive at discrete, non-continuous intervals. To address this, a feasible definition for overspeed items can be derived from the producer-consumer or bounded buffer model:

**Overspeed items.** Under the parameters $V$ for average speed limit and $B$ for burst capacity, each user is assigned an individual buffer that holds up to $B$ items and processes them at speed $V$. A data item, upon arrival, is directed to the buffer of the user who owns it. If this buffer is full, the item is defined as an overspeed item; otherwise, it's added to the buffer and not considered overspeed. We illustrate the overspeed item in Figure 1. The aim of detecting overspeed items is to categorize each incoming item into one of these two categories. This method is more detailed than existing binary classifications of user keys, like identifying frequent items, which simply categorizes all keys as either frequent or infrequent [4], [5]. In actual implementation, to conserve resources, these buffers are usually not physically constructed.

Co-primary authors: Yuhan Wu, Hanbo Wu, and Chengjun Jia. Corresponding author: Tong Yang (yangtongemail@gmail.com).

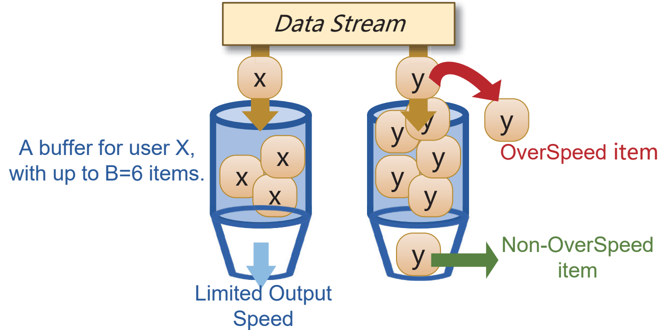



Figure 1: Illustration of OverSpeed Items. In a data stream containing items from two users, x and y, each entering their respective buffers. User x's items are within speed limits, whereas user y's buffer is full, resulting in newly arriving items being marked as overspeed.

**Use case — Network Rate Control:** Network operators restrict users from using the network beyond their paid rates to protect spare bandwidth and prevent congestion [1]. Overspeed items are the target of such restriction, with the V and B parameters matching the SLA-defined Committed Information Rate and Peak Burst Size [1]. Specifically, modern cloud data centers usually uses a rate control system to provide performance isolation among multiple simultaneous applications. Deploying it on the gateway offers several advantages such as interoperability, which places high demands on scalability for hosting more than hundreds of applications and millions of application instances [6]. However, the capacity of a single machine is limited. A switch with 15MB of SRAM can, at most, create entries of 8 bytes for fewer than 2 million users. In the process of scaling, more advanced hardware switches and distributed devices are introduced. However, such rate control may not be 100% accurate, with errors arising from the computational limitations of hardware switches [6] or synchronization issues in distributed software gateways [7]. As the actual workings of network devices are quite complex, the overspeed item concept has not been officially defined until our research. Within network devices, where high-speed storage comes at a high cost, the detection of overspeed items requires maximal space efficiency to accommodate more users.

Current methods [6], [8], [9] for detecting overspeed items struggle to scale up because they require memory that increases linearly with the number of users; our goal is to overcome this limitation. These methods use individual counters

along with user keys and timestamps to monitor the remaining space in each key's buffer, so as to identify overspeed items. This becomes problematic with a growing number of users, as the memory required can quickly exceed what is available, especially on specialized high-speed hardware. For example, the Intel Tofino V2 switch, which is commonly used for processing data streams, can handle billions of items every second but only has a small amount of high speed SRAM (15 MB), making it difficult to support large-scale users [10]–[12].

Our key observation is that it's not efficient to allocate memory for all possible users when only a small number of them have overspeed items. In practice, the total number of potential users typically exceeds the number of active [1] users ($N_A$) using the service at the same time by more than a hundredfold. Of these active users, less than 5% ($N_O$) are overspeed at any given moment [13]. This situation indicates that we only need to focus on and allocate resources for those overspeed users, suggesting a significant opportunity for optimizing the existing linear memory usage approach.

Guided by our observation, we adopt the sketching technique as our approach. Sketching [14]–[18] has demonstrated the potential to break away from linear memory usage while achieving high speed and controllable accuracy in other tasks, such as finding frequent keys. However, no existing sketch techniques are capable of detecting overspeed items, necessitating our own design.

In this paper, we propose the first sketch (named SpeedSketch) for finding the overspeed items. It is fast, memory-efficient, and accurate: (1) Fast: SpeedSketch algorithm is deployed on a real programmable switch and achieves the processing speed of 4.8 billion items per second. (2) Memory-efficient: We reduced the memory cost from $O(N)$ of existing solutions to $O(N'_O)$, where $N'_O$ is the largest-possible number of concurrent overspeed users ($N_A \geqslant N'_O \geqslant N_O$). SpeedSketch supports 10 million users with 1 MB memory in real datasets. (3) Accurate: SpeedSketch provides tight theory guarantees and achieves an average error rate of less than $0.1\%$ easily.

To tackle the problem that existing sketches cannot detect overspeed items, we developed Basic SpeedSketch. This approach includes buckets that track each buffer's status and calculate the item consumption rate using timestamps. We drew inspiration from current sketch and Bloom filter designs for this setup. We enable several users to simultaneously share a single bucket, optimizing memory use and maintaining high accuracy by dynamically changing the composition of the shared users and finely excluding the influence of inactive users. This design initially demonstrated potential but required enhancements in memory efficiency. Thereafter, we designed the Global-Clock, whose core idea is that having everyone hold a stopwatch is not the most cost-effective method. Instead, providing a global clock visible to all, where each individual only needs to record their timing start point, allows for the calculation of elapsed time when needed. This approach significantly improves memory efficiency by eliminating the space occupied by timestamps. Furthermore, our CounterFlip technique replaces the high bits of the counter with 1 bit, compressing the counter and further enhancing memory efficiency.

Our theoretical analysis revealed a unique aspect: while existing sketches haven't been able to provide high-accuracy per-user relative error bounds for other problems, our design uniquely achieves this for detecting overspeed items, marking a significant advancement in sketching techniques.

Finally, we list our key contributions as follows:

1) We formally define the concept of overspeed items in data streams, inspired by the producer-consumer model.
2) We develop SpeedSketch for detecting overspeed items, effectively leveraging the core principles of sketch methods and introducing new designs such as the Global-Clock Technique and the Counter-Flip Technique.
3) We perform a rigorous mathematical analysis of our algorithm, reducing the memory cost from $O(N)$ in existing solutions to $O(N'_O)$, where $N'_O$ is the maximum possible number of concurrent overspeed users ($N_A \geqslant N'_O \geqslant N_O$). Additionally, our algorithm is the first sketch to provide a per-user relative error bound.
4) We evaluate SpeedSketch on four datasets, and the results show that SpeedSketch can save memory by up to $6430$ times while only incurring $0.1\%$ average error rate (§ V-C1).
5) We prototype SpeedSketch on a switch that can process 4.8 billion items per second. All codes are open-sourced [19].

## II. PRELIMINARY

In this section, we provide a formal definition of the overspeed item (§ II-A), followed by the real-world application (§ II-B) and the related algorithms (§ II-C). We also introduce our target processing platforms (§ II-D).

Furthermore, since there are no existing approximate algorithms or sketches specifically addressing our problem, we introduce two baseline solutions. The two baselines are proposed in order to highlight the limitations of existing solutions in solving our problem, as well as to emphasize the challenges in designing new algorithms. The first one (§ II-C1) is a linear method for exact answers while the second one (§ II-C2) utilizes the the state-of-the-art (SOTA) sketch called Stingy [20]. While reading the two baselines may cost some time, readers can choose to skip them and proceed directly to our solution (§ III).

### A. Problem Definition

**Definition 1.** (Data Stream [21], [22]) The data stream $\mathcal{S}$ is a sequence of items, *i.e.*, $\mathcal{S} = \{\langle e_1, t_1\rangle, \langle e_2, t_2\rangle, \dots\}$. The $x$-th arriving item has a user key $e_x$ ($e_x \in U, |U| = N$) and a timestamp $t_x$.

**Definition 2.** (Overspeed items.) Under the parameters $V$ for processing speed limit and $B$ for burst capacity, each user is

[1] Active users refer to those whose buffers are not empty. Note that here, the buffer is virtually existent, merely used to define which items are overspeed.

**Algorithm 1:** The linear solution.

```
   Input: An item with user key e and arriving time t.
   Output: Answer
 1 Function UpdateConsumer(Bucket A):
 2 |   A.C ← max(0, A.C − V × (t − A.T))
 3 |   A.T ← t
 4 return
 5 UpdateConsumer(A^e)
 6 if A^e.C + 1 ⩽ B then
 7 |   Answer ← NOS
 8 |   A^e.C ← A^e.C + 1
 9 else
10 |   Answer ← OS
11 end
```

assigned an individual buffer that can hold up to $B$ items and processes them at a rate of $V$ items per second. When a data item arrives, it is directed to the buffer of its corresponding user. If this buffer is full, the item is classified as an OverSpeed (OS) item; otherwise, it is added to the buffer and categorized as Not OverSpeed (NOS). Every $1/V$ seconds, the buffer processes and removes one item from the buffer.

**Example.** Consider that user A's items $\{a, a, a, a, a, a, a, a\}$ arrive uniformly over seconds 1-8. The PC model has a buffer capacity of $B = 2$ and a consumption speed of $V = 0.5/s$. According to the definition, the capitalized and bolded items in the stream $\{a, a, a, \mathbf{A}, a, \mathbf{A}, a, \mathbf{A}\}$ are marked as OS. This is because the first two items of "a" consume the buffer capacity, and then the items are consumed at a rate of 0.5 item per second. Hence, within every 2-second interval, one "a" item is labeled as NOS, with the remainder **A** labeled as OS.

### *B. Applications*

Beyond the previously mentioned use case of network rate control, we present two additional use cases.

**AWS Message System Fairness:** In AWS messaging systems SQS [23], users send out streams of messages that are queued for processing. Ensuring every user gets fair service is essential. Following AWS's approach to avoiding queue backlogs, queues are rate-limited to identify excess messages (overspeed items), which are then moved to spillover queues for later processing. This system, however, can become resource-intensive when more users are added [3]. Thus, a more space-efficient and scalable method for detecting overspeed items can maintain fairness even as user numbers grow. Similar situations occur in messaging systems like Kafka [2] and RabbitMQ [24].

**Multi-sensor Rate Adjustment:** In Robotic Operating System (ROS), topic is a major communication method where a publisher node sends messages to a subscriber node. Messages not processed in a timely manner by the subscriber node are stored in a queue for later processing. In large-scale SLAM tasks [25], sensor fusion [26]–[28] is a common approach where multiple heterogeneous sensors (e.g. cameras, lidars, etc.) are used together to enhance feature detection and optimize path planning. In this setting, multiple sensors (each deployed as a ROS node) are connected with one unique mapping system running on a ROS node. Any message sent from the sensor node that cannot be handled by the mapping system in a certain amount of time is an overspeed item. These messages have lost their information timeliness as they spend too long time queuing, which cannot serve as a useful source of data for the mapping system. Detecting overspeed items in real-time can help discard outdated sensing data and adjust the rates from each sensor.

### *C. Prior Art*

We introduce two types of related algorithms: exact algorithms with linear space complexity, and approximate algorithms with sub-linear space complexity.

- Exact algorithms do not allow for any errors in algorithm design. The existing solutions for finding OS items are all exact algorithms. They allocate separate counting resources for each user, ensuring zero-error but requiring linearly proportional space with the number of users. However, this approach becomes impractical when supporting a large number of users.
- Approximate algorithms allow for controllable errors during their design and have smaller space requirements compared to exact algorithms. The size of the space needed for approximate algorithms depends on the allowed error. This article aims to propose the first approximate algorithm for finding OS items.

We will now proceed to introduce these two types in detail:

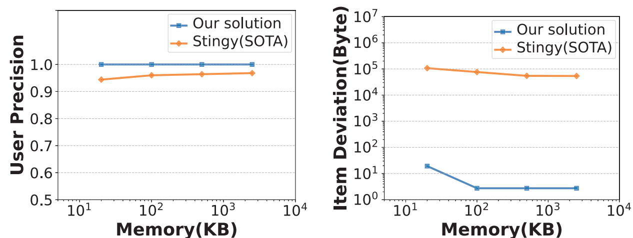
#### *1) **Exact Algorithms (Linear)**:*

The existing work [6], [8], [9] capable of finding overspeed items are linear solutions (Pseudocode in Algorithm 1): In a hash table, each user is allocated a bucket denoted by $A^e$, where $e$ represents the user key. Each bucket $A^e$ consists of a counter $A^e.C$ to record the usage and a timestamp $A^e.T$ to track the latest update time. When an item $e$ arrives at time $t$, the following steps are performed in its bucket $A^e$:

1) Remove items (update bucket): The update interval is calculated as $t - A^e.T$, and $V \times (t - A^e.T)$ items are consumed. If the remaining item quantity $A^e.C$ decreases to 0, the consumption will stop. Consequently, $A^e.C$ is updated as $max(0, A^e.C - V \times (t - A^e.T))$, where $max(0, x)$ sets any real number $x$ to a non-negative value. Additionally, the timestamp $A^e.T$ is updated to the current time $t$.
2) Insert items: Check whether there are enough space, *i.e.*, $A^e.C + 1 \leqslant B$. If so, $A^e.C$ is incremented by 1. Otherwise, the item is marked as an overspeed item and $A^e.C$ remains unchanged.

The linear solution incurs a memory cost that scales linearly with the number of users $N$, limiting its ability to support a large number of users ($N > 10^7$) on high-speed hardware platforms (§ II-D).

Despite being considered exact algorithms from the perspective of algorithm design, these solutions introduce a small degree of error when deployed on production equipment.




(a) Top 500 Users Frequently Reporting Overspeed Items (b) Average Per-User Overspeed Item Mark Deviation

Figure 2: Comparison between Stingy (SOTA Sketch) and Our Solution: SOTA can roughly locate users with many OS Items, but cannot accurately mark OS items.

This is because compromises are necessary to accommodate hardware limitations, such as floating-point calculations [6], or to address issues in distributed systems, such as real-time synchronization problems [7], as detailed in Section VII. Therefore, it can be demonstrated that adopting approximation algorithms with controllable errors is acceptable, especially if they bring significant scalability and cost benefits.

*2) **Approximate Algorithms (Sub-linear)**:*

In the processing of streaming data, approximate algorithms have gained widespread attention and recognition due to their small memory overhead, fast processing speed, and controllable accuracy. For convenience, we collectively refer to these algorithms as "sketches." Existing sketches have proven effective in tackling numerous critical patterns, including frequent items, heavy changes, bursts, persistent items, etc. Despite the abundance of sketch schemes, none of them can be readily adapted to efficiently detect OS items.

When considering patterns comparable to the OS item, the frequent item (also referred to as frequent user or heavy hitter) stands out as it denotes the user key whose frequency surpasses a pre-defined threshold $F$ within a specified time window, with options including either fixed or sliding windows. For example, considering the aforementioned stream $\{a, a, a, a, a, a, a, a\}$ in time $[1, 8]$, "a" with a frequency of 8 would be the frequent user under a threshold of $F = 2$.

Although a large number of sketches can estimate the user's frequency and identify frequent users, they struggle to pinpoint which specific items are overspeed. To illustrate this, consider a straightforward solution that uses a sketch (*e.g.*, the state-of-the-art (SOTA) sketch called Stingy [20]) to continuously record the frequency of each user. Subsequently, any subsequent items from a user are labeled as overspeed once their frequency exceeds a specified threshold, such as 2. To ensure long-term smooth processing, we clear the sketch and restarts counting every $W$ (*e.g.*, $W = 4$) seconds, where each $W$-second interval is referred to as a fixed window. In our example, the capitalized and bolded segments in $[a, a, \mathbf{A}, \mathbf{A}]$, $[a, a, \mathbf{A}, \mathbf{A}]$ would be marked as OS. With this approach, OS items cluster towards the end of each time window, leading to detrimental periodic network traffic fluctuations and other potential issues.

We further propose a smarter improvement using the idea of sliding windows, but its performance remains unsatisfactory. To enhance accuracy, we reduce the magnitude of the window size $W$ and maintain $M$ stingy sketches to track the number of NOS items attributed to each user in the most recent $M$ windows. When a new item arrives, we query the frequency of NOS items for its user key within the previous $M$ windows. Based on the buffer capacity $B$ and consumption speed $V$ of the PC model, we simulate whether the current buffer would be empty after processing these $M$ windows. This simulation is used to label the new item. However, as depicted in the experiment results, this approach only approximates which users are with the most OS items (Figure 2b) and fails to accurately identify the specific OS items (Figure 2b). The total count of OS items labeled for each user significantly deviates from the correct answer. Despite extensive attempts to utilize existing sketches as tools to address the problem of OS items, the results have been unsatisfactory. Therefore, we decide to design a new algorithm.

### *D. Processing Platforms*

Our goal is to design a common algorithm for multiple platforms including both CPU software and high-speed hardware platforms (*i.e.*, FPGA and programmable switches). Among them, programmable switches have the highest processing speed and the strongest design constraints. In other words, as long as the algorithm we designed can be implemented on a switch, it can be implemented on other platforms. Intel Tofino V2 switch is a popular programmable switch with total 15MB of memory. Any algorithm successfully deployed on it, including ours, can achieve a processing speed of 4.8 billion items per second.

Table I: Key symbols in this paper.

| Symbol | Meaning |
|---|---|
| $N$ | Number of distinct users potentially present |
| $B$ | Capacity of a single user buffer |
| $V$ | Rate at which items are consumed from a buffer |
| $C$ | Usage of a single buffer |
| $G$ | Global counter |
| $D$ | Total number of items in the data stream |
| $S$ | Speed of the data stream (items per second) |

## III. SpeedSketch Design

To find overspeed items under strict memory constraints, we propose two solutions as follows:

1) Basic SpeedSketch, a memory-efficient solution inspired by sketches and the Bloom filters [29], [30], greatly reduces the memory cost with bounded accuracy loss.
2) Advanced SpeedSketch further saves memory utilizing the techniques of *Global-Clock* and *Thrift*. We also propose another two extension techniques that can be added to SpeedSketch when necessary.

### *A. The Basic SpeedSketch*

**Methodology.** In the linear solution, one bucket can only serve one user at any time, which has become the bottleneck of its performance. Our approach is allowing multiple users to share one bucket at the same time. At a high level, when all users

**Algorithm 2:** The basic SpeedSketch.

**Input:** An item with user key $e$ and arriving time $t$.
**Output:** $Answer$

```
1  Initially, MinUsage is MAXVALUE.
2  for i = 1, 2, ..., k do
3      UpdateConsumer( A_i^{h_i(e)} )
4      MinUsage ← min(MinUsage, A_i^{h_i(e)}.C)
5  end
6  if MinUsage + 1 ⩽ B then
7      Answer ← NOS
8      for i = 1, 2, ..., k do
9          A^{h_i(e)}.C ← min(B, A^{h_i(e)}.C + 1)
10     end
11 else
12     Answer ← OS
13 end
```

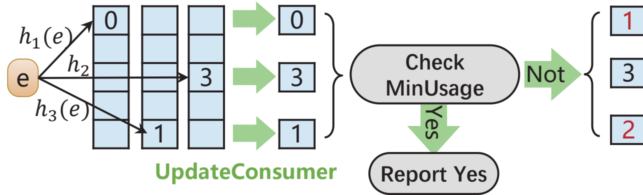



Figure 3: An example of the Basic SpeedSketch with $n = 5$, $k = 3$ and $B = 3$. For an arriving item $e$, we access all its mapped buckets and update their usage to the latest state, *i.e.*, $0, 3, 1$. The minimum usage is $0$, which is less than $B$. Thus the item is not an overspeed one and we increment all its usages by one (but the usage should be no more than $B$).

sharing a bucket are slow, they can perfectly use a bucket at the same time. Otherwise, we can repeatedly change the composition of the shared users until the error is small and controllable.

**Data Structure.** The basic version of SpeedSketch consists of $k$ arrays and each array contains $n$ buckets. The $j$-th bucket in the $i$-th array is denoted as $A_i^j, i \in [1, k], j \in [1, n]$. There are $nk$ buckets in total, which is much smaller than the number of buckets in the linear solution. Each bucket has a counter $A_i^{h_i(e)}.C$ to record the usage, and a timestamp $A_i^{h_i(e)}.t$ to record the latest update time. There are $k$ independent hash functions: $h_1(\cdot), \dots, h_k(\cdot)$. For the arriving item with user key $e$, the $i$-th function maps the user key to a random *mapped bucket* in the $i$-th array, *i.e.*, $A_i^j, j = h_k(e)$.

**Insert.** For each arriving item, we first calculate the hash to find the $k$ mapped buckets. Then, for all mapped buckets, we update the usage after the consumption in the same way as introduced in Straw-man, and find the minimum usage $MinUsage = min_{\forall i}(A_i^{h_i(e)}.C)$. Among all recorded usages in the mapped buckets, $MinUsage$ is the most accurate one in depicting the real usage of user $e$, which is explained later. Then, we identify the overspeed item by judging whether $MinUsage + 1$ reaches capacity $B$. If $MinUsage + 1 \leqslant B$, the arriving item is not an overspeed one and we increment all usages in its mapped buckets by one. Otherwise, the arriving item is reported as an overspeed one. We show an example in

**Algorithm 3:** The advanced SpeedSketch.

**Input:** An item with user key $e$ and arriving time $t$.
**Output:** $Answer$

```
1  G ← ⌊V × t⌋
2  MinUsage ← MAXVALUE
3  for i = 1, 2, ..., k do
4      C ← max(0, A_i^{h_i(e)}.cnt − G)
5      MinUsage ← min(MinUsage, C)
6      if C + 1 ⩽ B and C ⩽ MinUsage then
7          C ← C + 1
8      end
9      A_i^{h_i(e)}.cnt ← C + G
10 end
11 Answer ← { NOS   MinUsage + 1 ⩽ B
            { OS    Otherwise.
```

Figure 3.

From what is stated above, we can see that the usage recorded in one bucket is the total usage of all users hashed to it. As an increasing number of other users share the same bucket with $e$, the recorded usage will become larger with time. Any usage in the mapped buckets of $e$ is no less than the real usage of $e$ (*i.e.*, the usage of $e$ in the linear solution). Therefore, the minimum usage is the most accurate one and thus we use it to identify the overspeed item. At a high level, SpeedSketch has the following properties:

1) Inactive users who have no items do not incur overhead. Inactive users neither increment any counter nor are allocated with any fixed resources.
2) Slow users can share one bucket with no error. When only slow users enter a bucket and their total speed and burst size are less than $V$ and $B$ respectively, no overspeed item will be reported, which is perfectly exact.
3) One-side and small error. SpeedSketch has a one-side error that it never miss any overspeed item, and we mathematically prove that the error is small in § IV.

### B. The Advanced SpeedSketch

The advanced SpeedSketch is the optimized version of the basic SpeedSketch featuring two techniques: *the Global-Clock* and *the Thrift*. It is worth mentioning that the utilization of these two techniques does not introduce additional drawbacks. Global-Clock reduces the memory cost of every single bucket without incurring additional errors, while Thrift reduces the error without changing the data structure.

**The Global-Clock Technique.** Reducing the cost of the bucket can always benefit SpeedSketch as a whole. The Global-Clock technique tries to remove the timestamp and to record only one counter in each bucket. The timestamp serves for consuming items, *i.e.*, the consumed amount is determined by how long ago it was last updated. After the removal of timestamp, we need an alternative way to consume items. A straightforward way is to use an additional background thread to scan all buckets and consume items periodically, but it will increase the computing cost and reduce the throughput. Our approach is to maintain a global number of consumed items that can update all buckets simultaneously. The details are as follows.

We add a new global counter $G$ that grows at the speed of $V$ and has no upper limit, and let each bucket only record a counter $A_i^j.cnt$ ($= A_i^j.C + G$) for the number of consumed items, rather than the usage $A_i^j.C$ (*i.e.*, the number of items not consumed). Therefore, $C = A_i^j.cnt - G$ is the usage in bucket $A_i^j$ as before. Besides, $C$ should not be less than 0, so we calculate the usage $C$ by $max(0, A_i^j.cnt - G)$.

As shown in Algorithm 3, for the arriving item $e$, we access each mapped bucket in sequence and work as follows: First, we calculate $C$ by $max(0, A_i^{h_i(e)}.cnt - G)$. Then, we update the $MinUsage$ by $C$ as before. Next, we try to increment $C$ to record the arriving item and ensure that it does not exceed $B$. Note that $C$ should be no greater than $MinUsage$, which is the Thrift-Update Technique detailed later. So far, the calculation of $C$ in this bucket has been completed. Since $C$ is a temporary variable that is not stored, we save it through $A_i^{h_i(e)}.cnt = C + G$. Finally, in the same way as the basic version, we report $NOS$ if $MinUsage + 1 \leqslant B$. Otherwise, we report $OS$.

**The Thrift Technique.** The Thrift technique can reduce the repeated increment in the mapped buckets, so as to improve the accuracy. The operation of Thrift is quite simple (Line 6 in Algorithm 3): Do not increment $C$ when $C > MinUsage$.

The justifications are as follows: Recall our goal of approximating the real usage of $e$ by $C$. Define $u$ to denote the real usage of $e$ before inserting the arriving item, *i.e.*, $u = A^e.C$, where $A^e.C$ is the counter in the linear solution. $C$ should be made as small as possible while being no less than $u$. If so, we have $u \leqslant C_i \forall i$, where $C_i$ is the $C$ in the $i$-th mapped bucket[2], and thus $u \leqslant MinUsage$. The new real usage after the arrival of the new item is $u' = u + 1$. If $C$ is greater than $MinUsage$, we have $u' = u + 1 \leqslant MinUsage + 1 < C + 1$. Since both $u$ and $C$ are integers, we have $u' \leqslant C$. Therefore, when $C > MinUsage$, we do not increment $C$ just to ensure that $C$ is no less than the real usage and as small as possible.

### C. Extensions

We propose two extension techniques of SpeedSketch: 1. The Weight Technique, which aims at supporting customized size of item, instead of the traditional "1". 2. The Counter-Flip Technique, which can solve the integer overflow problem that may be encountered when running the sketch for a long time. The pseudo-code of SpeedSketch with two extensions is shown in Algorithm 4.

**The Weight Technique.** In some scenarios, each item may have a different weight (*e.g.*, the size of the data packet in the network is between 64 Bytes and 1500 Bytes), making it meaningful to consider the overspeed item of the weighted version: The $x$-th arriving item has a user key $e_x$, the current timestamp $t_x$, and its weight $w_x$, where $w_x \in [0, W]$ and $W$ is the maximum possible weight. The total weight of all items in the same buffer should not exceed the capacity $B$. The item incapable of entering the buffer is defined as an overspeed item.

[2] $C_i = max(0, A_i^{h_i(e)}.cnt - G)$

**Algorithm 4:** The advanced SpeedSketch with two extensions.

**Input:** An item with user key $e$, time $t$, and weight $w$.
**Output:** $Answer$

```
1  G ← ⌊V × t⌋
2  G' ← G mod MaxG
3  Flag = ⌊G / MaxG⌋ mod 2
4  P' ← { 1  w.p. w/W ; 0  Otherwise }
5  MinUsage ← MAXVALUE
6  for i = 1, 2, ··· , k do
7      if A_i^{h_i(e)}.F ≠ Flag then
8          A_i^{h_i(e)}.F ← Flag
9          A_i^{h_i(e)}.cnt ← A_i^{h_i(e)}.cnt − MaxG
10     end
11     C ← min(B, max(0, A_i^{h_i(e)}.cnt − G))
12     MinUsage ← min(MinUsage, C)
13     if C + 1 ⩽ B and C ⩽ MinUsage then
14         C ← C + P
15     end
16     A_i^{h_i(e)}.cnt ← C + G
17 end
18 Answer ← { NOS  MinUsage + 1 ⩽ B ; OS  Otherwise. }
```

We solve the weighted version by sampling technique. Let each unit of the usage counter $C$ represent a weight of $W$. For each arriving item with weight $w$, increment the usage by one with a probability of $\frac{w}{W}$, and otherwise leave it as it is. In other words, we just replace "$C \leftarrow C + 1$" in line 6 of Algorithm 3 with "$C \leftarrow C + P$", where $P = \begin{cases} 1 & w.p.\ \frac{w}{W} \\ 0 & Otherwise. \end{cases}$ The sampling introduces additional errors, but considering the huge number of items, the central limit theorem tells us that the error is small and tolerable.

**The Counter-Flip Technique.** If SpeedSketch has been running for a long time, $G$ and $A_i^{h_i(e)}.cnt$ may grow until the integer overflow. The Counter-Flip Technique can solve the potential error of the overflow, and reduce the size of $A_i^{h_i(e)}.cnt$ to further save memory. Specifically, we limit $G$ (denoted as $G'$) to grow circularly in the interval $[0, MaxG)$ (*e.g.*, $MaxG = 2^{16} = 65536$), and use a global one-bit $Flag$ (initially 0) to record the parity (odd/even) of the number of cycles $G's$. Specifically, every time $G'$ exceeds $MaxG$, we reset it to 0 and $Flag$ will be flipped ($Flag = 1 - Flag$). In every bucket, we append a local one-bit flag $A_i^{h_i(e)}.F$ to record the value of $Flag$ when last accessed. Every time an item accesses its mapped bucket, we first check whether the global flag is the same as the local flag. If so, we just insert the item in the same manner as before (Line 4-9 in Algorithm 3). Otherwise, we know that after undergoing an extra cycle, $G'$ has become a smaller value, whilst $A_i^{h_i(e)}.cnt$ has not. In this case, we perform additional operations of updating the local flag to the global one and decreasing $A_i^{h_i(e)}.cnt$ by $MaxG$. Then we insert the item as before. Only in rare cases, when there are two $G$s, namely $G1$ and $G2$, that corresponds to two consecutive updates of a bucket, and their difference $G2$-$G1$ is close enough to a multiple of $2 \cdot MaxG$, will the counter-flip

be unable to reduce the overflow error to zero. Even in such cases, the errors are strictly no greater than $B$, making them almost negligible.

Counter-Flip limits both $G$ and $A_i^{h_i(e)}.cnt$ to less than $MaxG + B$, which not only eliminates most overflow errors, but also further reduces the size of the counter $A_i^{h_i(e)}.cnt$ to $log_2(MaxG + B)$ bits.

## IV. Mathematical Analysis

In this section, we comprehensively show the theoretical features of the SpeedSketch, including accuracy, space complexity and time complexity. Our analysis is based on the basic version of SpeedSketch, but since the advanced version of SpeedSketch (Algorithm 3) outperforms the basic one in all aspects, all theoretical results are applicable to both versions.

We define some symbols as follows. For the data stream with $N$ users and $D$ items, the set of user keys is $\mathcal{S} = \{e_1, e_2, \ldots, e_N\}$. Define $f_i$ that denotes the number of items with $e_i$, *i.e.*, the size of $e_i$. Define $g_i$ that denotes the number of items that are not overspeed among all $e_i$'s items, *i.e.*, the ground-truth. Define $\hat{g}_i$ that denotes the number of items marked as not overspeed in the basic SpeedSketch, *i.e.*, our estimation.

We first give a simple error bound for a single user. The derivation process is simple and radical, and the optimization related to arrival time is not used, so the result is relatively weak.

**Lemma 1.** *(Weak Per-user Absolute Error Bound) Given a basic SpeedSketch with $n = \lceil e/\epsilon \rceil, k = \lceil ln(\frac{1}{\delta}) \rceil$, with probability at least $1 - \delta$, for any user,*

$$\textit{AbsoluteError} = |g_i - \hat{g}_i| \leqslant \epsilon D, \ \textit{and} \ \hat{g}_i \leqslant g_i \tag{1}$$

*Proof.* Give an indicator function

$$I(i,j,k) = \begin{cases} 1 & i \neq k \wedge h_j(e_i) = h_j(e_k) \\ 0 & Otherwise \end{cases}$$

which outputs 1 if user $e_i$ and $e_k$ collide at the $j$-th array. Let $X_{i,j}$ be the total size of users collided with user $e_i$ in the $j$-th bucket, $X_{i,j} = \sum_{k=1}^{n} I(i,j,k) f_k$. Let $\hat{g}_{i,j}$ be the estimated number of $e_i$'s non-overspeed items in the $j$-th bucket. In the best case, no collision occurs (*i.e.*, $\hat{g}_{i,j} = g_i$). If collision occurs, $\hat{g}_{i,j}$ decreases, but the decrement will not exceed the total collisions, *i.e.*, $g_i - X_{i,j} \leqslant \hat{g}_{i,j} \leqslant g_i$. Since the estimation depends on the $MinUsage$, $\hat{g}_i$ must be no smaller than any $\hat{g}_{i,j}, \forall j$. Therefore, we have

$$g_i - \hat{g}_i \leqslant \min_{j=1\ldots k} X_{i,j}, \quad X_{i,j} = \sum_{k=1}^{n} I(i,j,k) f_k$$

For $X_{i,j}$, we have:

$$E[I(i,j,k)] = \Pr[h_j(i) = h_j(k) \wedge i \neq k] \leqslant \frac{1}{n} = \frac{\epsilon}{e}$$

$$E[X_{i,j}] = E\left(\sum_{k=1}^{n} I(i,j,k) f_k\right) = \sum_{k=1}^{n} E[I(i,j,k)] f_k \leqslant \frac{\epsilon}{e} D$$

By the Markov inequality, $\Pr[X_{i,j} > \epsilon D] < \frac{E[X_{i,j}]}{\epsilon D} = \frac{1}{e}$. Therefore, the error is bounded with high probability:

$$\begin{aligned} \Pr[g_i - \hat{g}_i > \epsilon D] &\leqslant \Pr\left[\left(\min_{j=1\ldots k} X_{i,j}\right) > \epsilon D\right] \\ &\leqslant (\Pr[X_{i,j} > \epsilon D])^k = e^{-k} \leqslant \delta \end{aligned}$$

□

Lemma 1 does not take into account the time characteristics of item arrival, and thus only derives the bound from the total size of each user. We further propose these observations:

1) In the proof of Lemma 1, the ground-truth overspeed items from other users will not cause additional errors to the observed user $e_i$, so they can be excluded from the bound.
2) For a user $e_i$ that is not active for a long time, we only need to focus on the nearby items that arrive at the same time or a little earlier (the time difference is no more than $\frac{B}{V}$). Compared to the large number of total items $D$ in Lemma 1, we only need a small number of nearby items to safely bound the error of $e_i$.
3) For user $e_i$ with $g_i$ non-overspeed items, the worst pattern which results in most error is as follows: Fill the buffer with $B$ items instantaneously, and then continue to insert items at the speed of $V$ until a total of $g_i$, in the time interval of $\frac{g_i - B}{V}$. This pattern ensures that the buffer is always full and therefore escalates every collision with other users' items to an error of 1.

Based on the above three observations, we can replace the $D$ in Lemma 1 by a much smaller $D_i$ as follows: Let $S$ be the speed of the data stream including all users. The length of time interval that will cause the most errors is $T_i = \frac{g_i - B}{V} + \frac{B}{V} = \frac{g_i}{V}$. Thus $D_i = ST_i = \frac{Sg_i}{V}$. We obtain the following theorem.

**Theorem 1.** *(Per-user Error Bound) Given a basic SpeedSketch with $n = \left\lceil \frac{e}{\gamma} \cdot \frac{S}{V} \right\rceil, k = \lceil ln(\frac{1}{\delta}) \rceil$, with probability at least $1 - \delta$, for user $e_i$,*

$$\textit{Absolute Error} = |g_i - \hat{g}_i| \leqslant \gamma g_i \tag{2}$$

$$\textit{Relative Error} = \left| \frac{g_i - \hat{g}_i}{g_i} \right| \leqslant \gamma \tag{3}$$

According to the central limit theory, for $N > 100$ users, we can use the expectation of relative error to approximate the average relative error. Besides, we get the total absolute error by accumulating the absolute error of each user.

**Theorem 2.** *(The Overall Error) Given a basic SpeedSketch with $n = \left\lceil \frac{e}{\gamma} \cdot \frac{S}{V} \right\rceil, k = \lceil ln(\frac{1}{\delta}) \rceil$, with probability at least $1 - \delta - o(N^{-0.5})$,*

$$\textit{Average Relative Error} = \frac{1}{N} \sum_i \left| \frac{g_i - \hat{g}_i}{g_i} \right| \leqslant \frac{\gamma}{e} \tag{4}$$

$$\textit{Total Absolute Error} = \sum_i |g_i - \hat{g}_i| \leqslant \frac{\gamma}{e} \cdot D \tag{5}$$

**Complexity:** Our space complexity is $O(nk) = O(\frac{S \ln(\delta^{-1})}{\gamma V})$, in which $\gamma$ and $\delta$ are accuracy parameters. Since a total speed

of $S$ can support up to $N'_O = \frac{S}{V}$ overspeeding users (each of which must have a speed no less than $V$), the space complexity can also be written as $O(\gamma^{-1}\ln(\delta^{-1})N'_O)$, where $N'_O$ is the largest-possible number of concurrent overspeeding users. The time complexity of inserting any item is $O(k) = O(\ln(\delta^{-1}))$.

Existing sketches could only guarantee absolute error, not relative error. Guaranteeing relative error is more challenging. This is because when the absolute error is very small, the relative error can be large due to the base being small. We can provide bounds for both relative and absolute errors on a per-user basis.

## V. Evaluation

In this section, we conduct experiments for SpeedSketch and present the results. Firstly, we describe the experiment setup and introduce the stra-wman solution. Secondly, we apply SpeedSketch to detect overspeed items, evaluating its accuracy and throughput. Codes of SpeedSketch are open-source on Github [19] anonymously.

### A. Experiment Setup

**Evaluation Metrics.**

- **Absolute Error:** $\sum_{e_i \in U} |f(e_i) - \hat{f}(e_i)|$, where $U$ is the set of overspeeding users, $f(e_i)$ is the total size of user $e_i$'s OS item (ground truth), and $\hat{f}(e_i)$ is the total size of all reported OS items. And the Average Absolute Error (AAE) is $\frac{1}{|U|}\sum_{e_i \in U} |f(e_i) - \hat{f}(e_i)|$.
- **Relative Error:** $\sum_{e_i \in U} |f(e_i) - \hat{f}(e_i)|/f(e_i)$. And the Average Relative Error (ARE) is $\frac{1}{|U|}\sum_{e_i \in U} |f(e_i) - \hat{f}(e_i)|/f(e_i)$.
- **False Positive Rate:** $\frac{|\Omega - \Psi|}{|U|}$, where $\Psi$ is the set of overspeed users (*i.e.*, users with OS items) and $\Omega$ is the set of reported overspeed users. It refers to the ratio of the number of correctly reported answers to that of all users.
- **Throughput:** $N/T$, where $N$ is the number of operations and $T$ is the time consumption. Mips is an acronym for Million Instructions Per Second.

**Datasets.**

- **IP Trace:** An anonymized dataset comprised of IP packets collected from [31]. We identify a user by its source and destination IP addresses. The dataset is a 23-second-long stream with 26.6M items and about 1.3M users
- **University Data Center:** An anonymized packet trace from university data center [32]. It is a one-hour-long stream with 19.8M items and about 0.7M users. To standardize the number of concurrent users, we compress the timeline of the dataset by $1000\times$.
- **Synthetic Dataset I (Zipf):** We generate one synthetic packet trace dataset, the sizes and speeds of users are empirically generated according to Zipf distribution. The arrival times of items are generated according to $b$-model [33]. The dataset is a 8-second-long stream with 10M items and about 0.45M users.
- **Synthetic Dataset II (Burst):** We generate an additional synthetic dataset that only includes burst users. The dataset is a 4-second-long stream with 10M items and about 12K users.

**Implementation(Software).** We implement SpeedSketch and related algorithms in C++. Each bucket of SpeedSketch consists of a 32-bit counter. According to the study in § V-C, we set $k$=3 by default for the accuracy-throughput tradeoff. For all algorithms, we set $V$=10Mbps by default, and $B$=25KB to make sure the tolerance interval of a bucket $B/V$=0.02s. All the software experiments are conducted on our server with Intel CPU i9-10980XE CPU (18 cores, 36 threads, 3.00 GHz, 1MB L1 cache, 16MB L2 cache, 25MB L3 Cache) and 128GB DRAM. Executable files are compiled with O2 optimization on.

### B. Straw-man Solution

Since the aforementioned baseline (§ II-C2) showed unsatisfactory results, we made some simple improvements to the linear solution using the ideas of sketches. This was undertaken to provide a point of reference highlighting the superiority of our final proposal.

Our straw-man solution, an intuitive improvement of the linear solution, makes better use of memory by allowing active users to preempt storage space for statistics, but loses all statistics on users who fail in the preemption.

**Data Structure.** Straw-man has an array of $n$ buckets, where the $j$-th bucket is denoted as $A^j, j \in [1, n]$. Each bucket $A^j$ has three fields: a key field $A^j.Key$ recording which user occupies this bucket, a counter $A^j.C$ recording the usage, and a timestamp $A^j.t$ recording the latest update time. A hash function $h(\cdot)$ maps each user key $e$ to a bucket (called the mapped bucket), *i.e.*, $A^{h(e)}$.

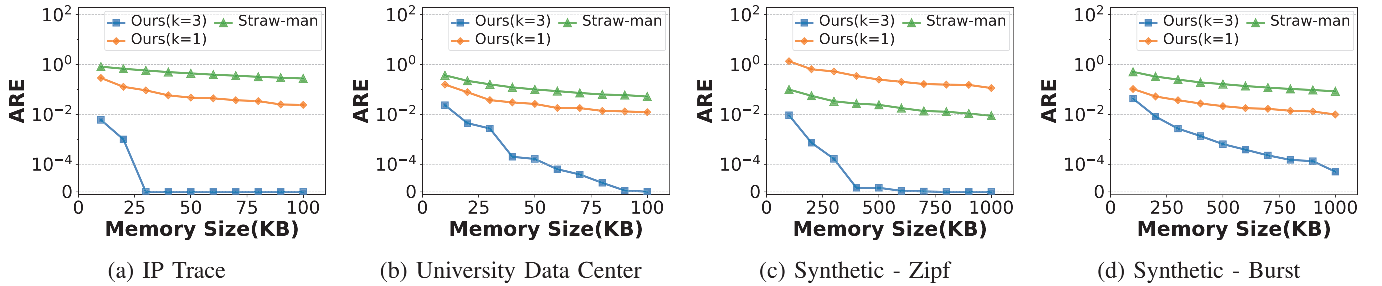
**Insert.** For each arriving item with user key $e$, we insert it as follows: We calculate the hash to find the mapped bucket $A^{h(e)}$, and update the usage after the consumption to the latest state in the same manner as the linear solution. Then, for the most critical step of Straw-man, we check whether the bucket can be cleared and preempted by new user $e$. The criterion is whether the usage $A^{h(e)}.C$ is 0. If so, we know that all items have been consumed, so we assign it to the new user. Otherwise, we remain the old one. Finally, we try to record the arriving item. If the bucket is not occupied by this user ($A^{h(e)}.Key \neq e$), we have no information and thus report $NOS$ for the arriving item. If the bucket is occupied, on the other hand, we check whether the usage will reach the capacity ($A^{h(e)}.C + 1 \leqslant B$). If not, we report $NOS$ and increment $A^{h(e)}.C$. Otherwise, we report $OS$.

### C. Experiments on Accuracy

In this section, we apply SpeedSketch to overspeed item detection and compare SpeedSketch with other solutions.

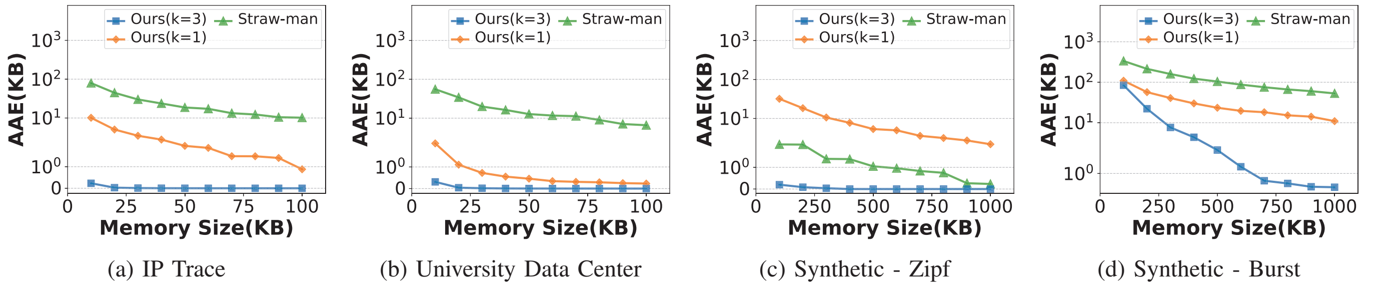
*1)* ***Impact of Memory****:*

We find that SpeedSketch achieves high accuracy and memory-efficiency in overspeed item detection, compared with other solutions. Specially, the linear solution in § II-C1 has no




(a) IP Trace (b) University Data Center (c) Synthetic - Zipf (d) Synthetic - Burst

Figure 4: ARE of Overspeed Item Detection.




(a) IP Trace (b) University Data Center (c) Synthetic - Zipf (d) Synthetic - Burst

Figure 5: AAE of Overspeed Item Detection.

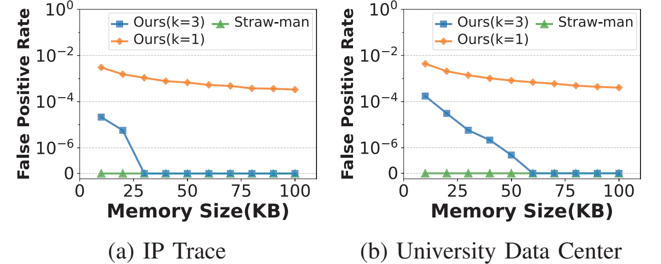
estimation error, while the memory requirement of it is associated with the number of total users. It requires about 14.72MB, 7.86MB, 5.04MB, and 0.14MB memory respectively for the four datasets IP Trace, University Data Center, Synthetic-Zipf, and Synthetic-Burst (the load rate of hash table estimated at 70%), which is far higher than other solutions.

**ARE vs. Memory Size (Figure 4a, 4b, 4c, 4d):** It is shown that SpeedSketch(k=3) keeps ARE below $10^{-4}$ when the memory sizes are 30KB, 50KB, 300KB, and 900KB on the four datasets respectively. With the same memory size, AREs of SpeedSketch(k=1) and the straw-man solution are about more than $10^2\times$ higher.

**AAE vs. Memory Size (Figure 5a, 5b, 5c, 5d):** It is shown that SpeedSketch also controls AAE to a satisfying level. For example, when the memory size is 10KB, AAE of SpeedSketch(k=3) is far lower than an MTU (1500Bytes by default) on the two real world datasets, while the AAEs of others are approximately $10^1 \sim 10^2\times$ higher.

**False Positive Rate vs. Memory Size (Figure 6a, 6b):** Once an item from a user without OS items is mistakenly reported as an OS item, we call it a false positive. It shows that false positive rate is also controlled well with restricted memory. When the memory sizes are 30KB and 40KB on IP Trace and University Data Center datasets respectively, SpeedSketch(k=3) offers no error estimation. The straw-man solution has no false positive error, but it reports false negative instead.

**Minimal Memory Usage (Figure 7a, 7b):** We also examine the minimal memory cost required to achieve different target error rates. On real datasets, SpeedSketch achieves memory savings of **6430 and 1503 times compared to the existing linear solution**, with only a 0.1% reduction in accuracy (AAE). This is primarily due to the fact that the number




(a) IP Trace (b) University Data Center

Figure 6: False Positive Rate of Overspeeding User Detection.

of active users in real datasets is relatively small (less than 1%) compared to the total number of users. In synthetic datasets, where the proportion of active users is higher, the advantage of SpeedSketch diminishes and can even fall behind the linear method. Furthermore, as the tolerance for reduction in accuracy decreases, the advantages of SpeedSketch become less pronounced (Figure 7b). For a detailed analysis regarding the proportion of active users, please refer to § V-C4.

*2) Impact of Hash Function:*

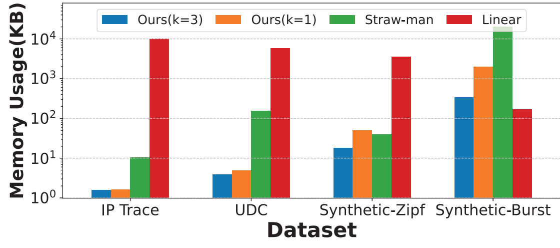
We implement three common hash functions (Bob Hash [34] , Murmur Hash [35], and xxHash [36]) to SpeedSketch, vary the memory size, and compare the accuracy on IP Trace dataset and Zipf synthetic dataset.

**AAE vs. Memory Size (Figure 8a, 8b):** It shows that the selection of hash functions has no significant influence on accuracy, and different hash functions affect throughput more than accuracy. Finally, we choose 32-bit Murmur Hashing for overall performance.

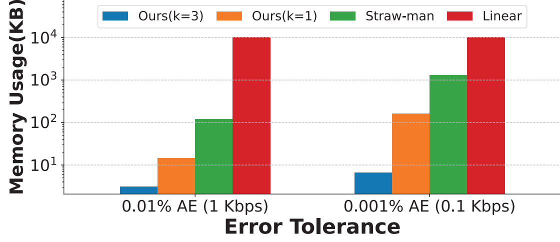
*3) Error Distribution:*

We further conduct several experiments on SpeedSketch to illustrate the factors related to the error.

**Error vs. User Size (Figure 9):** We count AAE and ARE of users with similar size on IP Trace dataset. The figures show that as the user size gradually becomes larger, the size of the

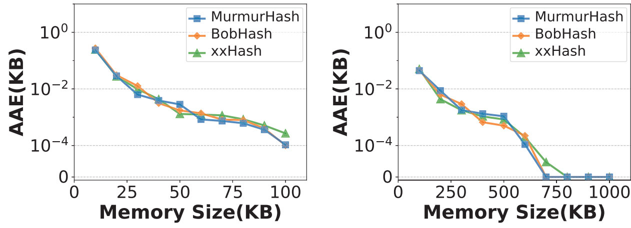



(a) Impact of Datasets (with 10Kbps (0.1% AE) Tolerance).




(b) Impact of Error Tolerance (with IP Trace).

Figure 7: Minimal Memory Requirement.




(a) IP Trace (b) Synthetic - Zipf

Figure 8: AAE of SpeedSketch with Different Hash Functions.

overspeed part also increases, so the AAE rises accordingly. No significant change in ARE is observed, which indicates the relative error of SpeedSketch is insensitive to user size.

**Error vs. # Concurrent Users (Figure 10):** We generate several independent Zipf synthetic datasets in the similar way, altering the number of concurrent users only. The figures show that AAE and ARE are both positively correlated with the number of concurrent users. Indeed, the errors are mainly

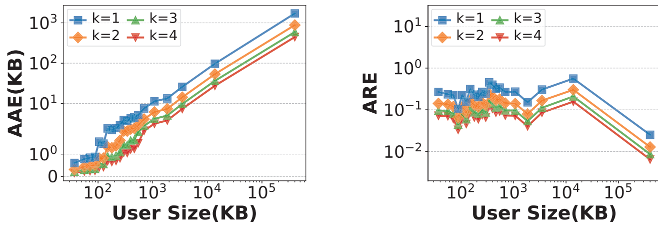



Figure 9: Error Distribution by Item Size.

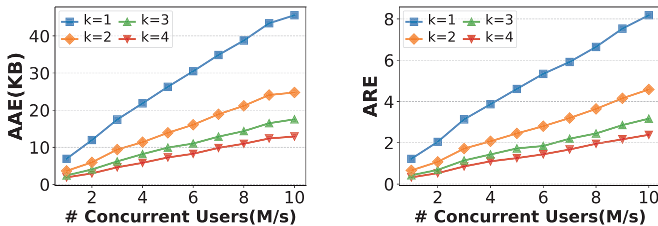



Figure 10: Error Variation by User Density.

from the hash collisions between the overspeeding users. As the number of concurrent users increases, the overspeeding users increase subsequently, thus the collisions and errors. Therefore, in real deployment of SpeedSketch, it is prudent to pay attention to the user density beforehand.

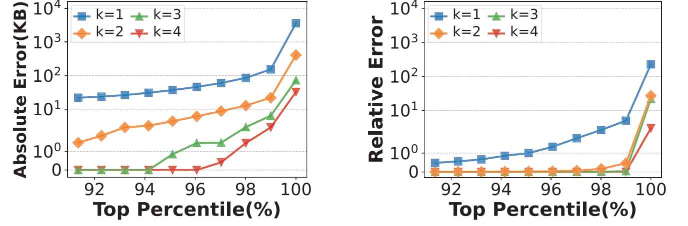



Figure 11: Error Distribution in Ascending Order.

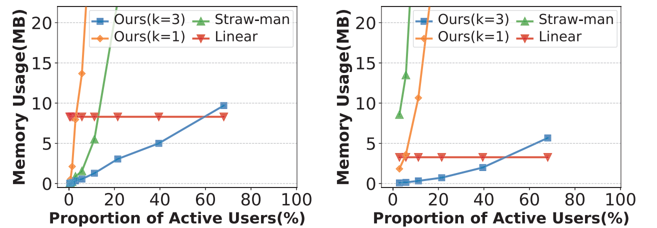
**Error per User (Figure 11):** We count absolute errors and relative errors of overspeeding users on IP Trace dataset and sort them in ascending order. The memory usage is fixed to 10KB. The figures indicate that only a very small number of users suffer from large errors. On the other hand, when $k = 1 \sim 3$, the increase in $k$ can significantly reduce hash collisions and thus the error. Therefore, it is recommended to set $k = 3$ for the accuracy-throughput tradeoff.

*4) Impact of Dataset:*

We alter the synthetic datasets, change the user density, and try to estimate the minimal memory usage to keep error lower than $0.1\%$ of speed limit $V$. Specially, the linear solution is not affected by the number of concurrent users and only concentrate on the number of total users.

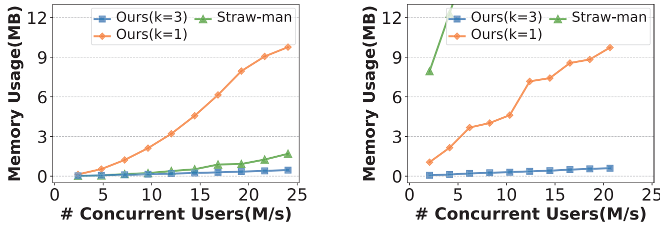
**Memory Usage vs. Proportion of Active Users (Figure 12a, 12b):** We find that the minimal memory usage of SpeedSketch($k = 3$) is far less than others when the proportion of active users is not extremely high. Notice that in most application scenarios, active users in recent 20ms will be a tiny fraction of overall users. When 5% of the users are recently active, for Zipf synthetic dataset, SpeedSketch($k = 3$) requires about 0.56MB memory, about $14.8\times$ less than others. For Burst synthetic dataset, SpeedSketch($k = 3$) requires 0.41MB memory, about $7.9\times$ less than others.

**Memory Usage vs. # Concurrent Users (Figure 13a, 13b):** We find that the minimal memory usage is essentially linearly correlated with the number of concurrent users, and SpeedSketchwith $k = 3$ requires far less memory than the others. For Zipf synthetic dataset, SpeedSketch($k$=3) requires 0.39MB memory when the number of concurrent users is 20M/s, about $3.21\times$ less than the straw-man solution. For Burst synthetic




(a) Synthetic - Zipf (b) Synthetic - Burst

Figure 12: Result under Different User Activity.




(a) Synthetic - Zipf (b) Synthetic - Burst

Figure 13: Result under Different User Densities.

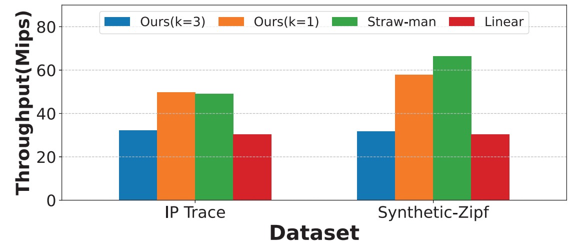



Figure 14: Throughput Evaluation.

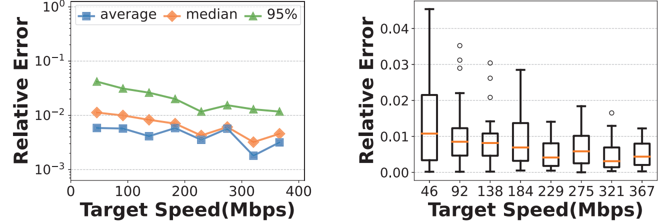



Figure 15: Per-user Error on Tofino Switch.

dataset, SpeedSketch($k = 3$) requires 0.60MB memory, while the others requires more than 10MB.

### D. *Experiments on Throughput*

In this section, we conduct experiments with two different datasets to illustrate the processing speed of SpeedSketch. We find that SpeedSketch is more memory efficient while achieving a throughput comparable to that of other solutions. **Process Throughput (Figure 14):** The figure shows that the process throughput of SpeedSketch($k$=3) is 32.3Mips on IP Trace dataset, which is comparable with the linear solution. Thanks to fewer hash function calls, SpeedSketch($k$=1) achieves 49.9Mips throughput, which is close to the performance of the straw-man solution. The throughput of SpeedSketch($k$=3) on Zipf synthetic dataset is 31.7Mips, the conclusion is the same as above and does not change significantly.

## VI. A Case Study: Network Rate Control

In this section, we implement SpeedSketch on the Tofino switch to demonstrate its effectiveness in Network Rate Control applications and present the experiment results.

The experiments on the Tofino switch can also demonstrate the wide applicability of SpeedSketch on various platforms. Among the four common platforms: 1) CPU, 2) GPU, 3) FPGA, and 4) Programmable switches, it is the programmable switches that impose the strongest restrictions. An algorithm that can be deployed on programmable switches is considered to be applicable for deployment on other platforms as well.

In the experiments, the Tofino switch and servers are connected with 40GbE links. Each server is equipped with two Xeon(R) Silver 4116 CPUs (12 cores, 2.1GHZ, and 16.5MB L3 cache), 256 GB RAM (2400MHZ), and a Mellanox Connectx-3 NIC (40GbE).

### A. ***Implementation***

As a specific programming hardware, the Intel Tofino switch has many limitations for a program to run on its platform.

- The built-in multiplication unit has limited precision, which only guarantees the unbiasedness for the multiplication/division of one 4b variable and one constant. It would introduce errors for other cases, which pose a severe impact on the correctness of the $Answer$ in Algorithm 4.
- The Stateful ALU only supports the update of a pair of up to 32b registers with two conditions, and there are restrictions on the expression of the new register values. It limits the number of operations that can be performed between the reading and writing to the same register, making it difficult to express the update of $A_i^{h_i(e)}.cnt$.
- There are only 12 stages in our Tofino switch, each supporting the add/subtract/mask/shift triple operations for only up to 32b integers. We must design the parallelism of the operations to limit the diameter of the computational graph.

We implement the advanced SpeedSketch with ~500 lines of $P4_{16}$ language and compile it on the ASIC within 10 stages. To improve the accuracy of the calculation, we use a pre-configured lookup table, which has $2^{16}$ entries, to replace the multiplication unit. To meet all above-mentioned constraints, we record a counter $A_i^{h_i(e)}.C2 = A_i^{h_i(e)}.cnt - B$ in every bucket $A_i^{h_i(e)}$, instead of recording $A_i^{h_i(e)}.cnt$ directly. Then the update of one bucket is approximate to

$$C2 \leftarrow \begin{cases} C2+1 & \hat{V} \leqslant C2 < MinUsage+1 \\ \max\{\hat{V}, C2\} & Otherwise. \end{cases}$$
$$MinUsage \leftarrow min(MinUsage, C2)$$

where $\hat{V} = G - B + 1$. We still identify the overspeed item according to $MinUsage$ as before (Algorithm 4). There is a little error in the approximate calculation, but the error is negligible according to our following experiments.

### B. ***Results***

**Resource Usage (Table II):** It shows the occupancy of critical hardware resources on the Tofino ASIC. Including the ability to distinguish different users, the whole system takes 10 stages and only 25 actions. It uses 6.5% SRAM and 32.5% ALU (Arithmetic Logic Unit) of the ASIC. PHV(Packet Header Vector) is used to pass variable values between stages. The system only needs 396B PHV and it accounts for 9.67%. It demonstrates the feasibility of SpeedSketch on the Tofino ASIC and actually leaves a lot of resources free.

Table II: The Tofino switch resource usage (Percentages in brackets are fractions of the total resource.)

| #Stages | #Actions | #SRAM | #Meter ALUs | PHV/Bytes |
|---|---|---|---|---|
| 10 | 25 | 52 (6.50%) | 13 (32.50%) | 396 (9.67%) |

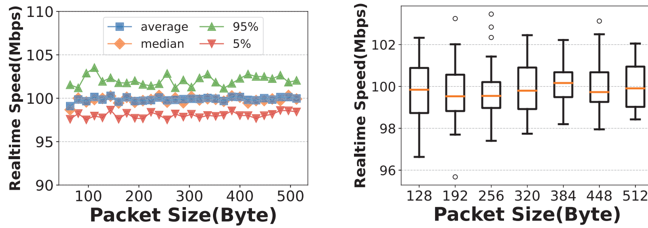



Figure 16: Per-user Speed on Tofino Switch.

**Relative Error vs. Target Speed (Figure 15):** It shows the average, median and 95-ile relative error for different target speeds. We range the speed from about 45Mbps to 400Mbps. The average and median relative errors are below 1% for almost all cases, and 95% is still not and than 5%. Such errors are negligible in the cloud network [6], [37]. It demonstrates the effectiveness of Algorithm 4 under different target speeds.
**Packet Size vs. Realtime Speed (Figure 16):** It shows the actual throughput for different packet sizes when we configure the Tofino to drop the overspeed packets if the incoming iperf UDP traffic is over 100Mbps. The packet size ranges from 128 to 512 bytes to test the effectiveness of the weight technique. The throughput jitter is always within the 1% of the target in most cases, and the maximum deviation does not exceed 4%. Because 30s traffic data are collected in the experiments, $G$ must have underwent overflow. As a result, the results also demonstrate the effectiveness of the Counter-Flip Technique on the testbed.

## VII. Related Work

In this section, we introduce each specific related work.
**Exact solutions.** As we summarized in § II-C, exact solutions allocate separate counting resources for each user, requiring linearly proportional space with the number of users. The Token Bucket (TB) [38] and the Hierarchical Token Bucket (HTB) [39] are two classic algorithms for finding OS items in packet-switched and telecommunications networks. Each TB or HTB bucket can only limit the speed of one or a group of users, hence they are generally organized into bucket lists composed of multiple buckets. The operation of the bucket in our basic SpeedSketch is isomorphic to TB, but the similarity ends there. SwRL [6], the SOTA technique, minimizes memory overhead for token bucket (TB) algorithms within the constraints of programmable switches, thereby supporting up to 1 million users. However, due to hardware limitations, it cannot achieve 100% accuracy. Specifically, current hardware programmable switches do not support high-precision floating-point multiplication and division operations. To navigate this limitation, SwRL employs an Approximate Division Table (ADT), which allows for an approximation of TB functionality but introduces a 1.98% error rate. On the other hand, alternatives using CPU servers for precise computations face challenges due to their lower processing power compared to hardware switches. Although software solutions on a single CPU server can perform precise calculations, their limited throughput necessitates a distributed setup. In such environments, synchronization issues prevent achieving 100% accurate rate control, illustrating the inherent trade-offs between accuracy, processing capacity, and the scalability of network rate control solutions [7].
**Sketches and approximate solutions** have proven effective in tackling numerous critical tasks. For frequency estimation and finding frequent items in the fixed window, works range from schemes like SpaceSaving [40], Misra-Gries [41], Count-Min [14], and Count [42] schemes to the latest USS [43], SS± [44], and Stingy [20], offering a variety of options. For the same task in the sliding window, typical solutions include Histograms [45], Exponential Count-Min [46], Proportional [47], Sliding [48], and more [49], [50]. However, as we explained in detail in § II-C2, finding frequent items cannot solve our problem.
**Sampling solutions.** The sampling solutions include two types, user sampling (randomly sampling a portion of users) [51] and item sampling (randomly sampling a certain percentage of packets, such as one percent) [52], [53]. The former doesn't work in this case, as we cannot mark OS for only a small subset of users. The latter is often used to sacrifice accuracy for improved processing speed (throughput) or to comply with hardware computation limits. We employed this method in § III-C, using one unit in the counter to represent a data volume of W (*e.g.*, 1500KB), to efficiently accomplish weighted statistics.

## VIII. Conclusion

In this paper, we formally define the overspeed item and propose SpeedSketch as a solution to address this challenge. Our experiments across four datasets demonstrate that SpeedSketch is highly scalable and memory-efficient, achieving up to a 6430-fold reduction in memory cost while maintaining a low average relative error of just 0.1%. The high memory efficiency of SpeedSketch enables it to support a large number of users within limited space and also facilitates its implementation on the Tofino switch, which offers a throughput capacity of 4.8 billion items per second. The comprehensive mathematical analysis we conducted further confirmed the efficiency of the algorithm. All our codes are available on GitHub for easy replication and improvement.

## Acknowledgment

We would like to thank the anonymous reviewers for their constructive feedback. Thanks to Yuxuan Tian for his help during the revision stage. This work is supported by National Key R&D Program of China (No. 2022YFB2901504), and National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009).

## REFERENCES


[1] Cisco switch document of traffic policing. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_plcshp/configuration/xe-3s/qos-plcshp-xe-3s-book/qos-plcshp-trfc-plc.html.

[2] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[3] Amazon Builders' Library. https://aws.amazon.com/builders-library/avoiding-insurmountable-queue-backlogs/?nc1=h_ls.

[4] G. Lukasz, D. David, D. Erik D, and etal. Identifying frequent items in sliding windows over on-line packet streams. In *ACM IMC*, 2003.

[5] Yuanpeng Li, Feiyu Wang, Xiang Yu, Yilong Yang, Kaicheng Yang, Tong Yang, Zhuo Ma, Bin Cui, and Steve Uhlig. Ladderfilter: Filtering infrequent items with small memory and time overhead. *Proceedings of the ACM on Management of Data*, 1(1):1–21, 2023.

[6] Yongchao He, Wenfei Wu, Xuemin Wen, Haifeng Li, and Yongqiang Yang. Scalable on-switch rate limiters for the cloud. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[7] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review*, 37(4):337–348, 2007.

[8] Yongchao He and Wenfei Wu. Fully functional rate limiter design on programmable hardware switches. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 159–160, 2019.

[9] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 468–482, 2016.

[10] Intel Tofino 2 Switch. https://www.intel.com/content/www/us/en/products/sku/218643/intel-tofino-6-4-tbps-4-pipelines/specifications.html.

[11] Tong Yang, Jie Jiang, Peng Liu, and etal. Elastic sketch: Adaptive and fast network-wide measurements. In *SIGCOMM Conference*, 2018.

[12] Shangsen Li, Lailong Luo, and Deke Guo. Sketch for traffic measurement: design optimization application and implementation. *arXiv preprint*, 2020.

[13] The CAIDA UCSD Anonymized Internet Traces Dataset - 2018.03.15. http://www.caida.org/data/passive/passive_dataset.xml.

[14] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[15] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *SIGMOD Conference*, 2016.

[16] Shangsen Li, Lailong Luo, Deke Guo, Qianzhen Zhang, and Pengtao Fu. A survey of sketches in traffic measurement: Design, optimization, application and implementation. *arXiv preprint arXiv:2012.07214*, 2020.

[17] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, page 15, 2011.

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

[19] https://github.com/Speed-Sketch/Speed-Sketch.

[20] 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.

[21] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, 2002.

[22] Georg Krempl, Indre Žliobaite, Dariusz Brzeziński, Eyke Hüllermeier, Mark Last, Vincent Lemaire, Tino Noack, Ammar Shaker, Sonja Sievi, Myra Spiliopoulou, et al. Open challenges for data stream mining research. *ACM SIGKDD explorations newsletter*, 16(1):1–10, 2014.

[23] Amazon Simple Queue Service. https://aws.amazon.com/sqs/.

[24] RabbitMQ. https://www.rabbitmq.com/.

[25] Seyed Ali Miratabzadeh, Nicolas Gallardo, Nicholas Gamez, Karthikpai Haradi, Abhijith R Puthussery, Paul Rad, and Mo Jamshidi. Cloud robotics: A software architecture: For heterogeneous large-scale autonomous robots. In *2016 world automation congress (WAC)*, pages 1–6. IEEE, 2016.

[26] Xiaobin Xu, Lei Zhang, Jian Yang, Chenfei Cao, Wen Wang, Yingying Ran, Zhiying Tan, and Minzhou Luo. A review of multi-sensor fusion slam systems based on 3d lidar. *Remote Sensing*, 14(12):2835, 2022.

[27] Chengjun Tian, Haobo Liu, Zhe Liu, Hongyang Li, and Yuyu Wang. Research on multi-sensor fusion slam algorithm based on improved gmapping. *IEEE Access*, 11:13690–13703, 2023.

[28] Kai Dai, Bohua Sun, Guanpu Wu, Shuai Zhao, Fangwu Ma, Yufei Zhang, and Jian Wu. Lidar-based sensor fusion slam and localization for autonomous driving vehicles in complex scenarios. *Journal of Imaging*, 9(2):52, 2023.

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

[30] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*, pages 684–695. Springer, 2006.

[31] The caida anonymized 2016 internet traces. http://www.caida.org/data/overview/.

[32] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *SIGCOMM conference*, 2010.

[33] M. Wang, T. Madhyastha, N. H. Chan, S. Papadimitriou, and C. Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *International Conference on Data Engineering*, pages 507–516, 2002.

[34] Hash website. http://burtleburtle.net/bob/hash/evahash.html.

[35] Murmur hashing source codes. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[36] Hash website. https://xxhash.com/.

[37] Keqiang He, Weite Qin, Qiwei Zhang, Wenfei Wu, Junjie Yang, Tian Pan, Chengchen Hu, Jiao Zhang, Brent Stephens, Aditya Akella, et al. Low latency software rate limiters for cloud networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 78–84, 2017.

[38] Jonathan Turner. New directions in communications (or which way to the information age?). *IEEE communications Magazine*, 24(10):8–15, 1986.

[39] HTB Linux queuing discipline manual. http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm.

[40] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[41] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[42] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.

[43] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1129–1140, 2018.

[44] Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. Spacesaving±: an optimal algorithm for frequency estimation and frequent items in the bounded-deletion model. *Proceedings of the VLDB Endowment*, 15(6):1215–1227, 2022.

[45] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[46] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proceedings of the VLDB Endowment*, 5(10), 2012.

[47] Nicoló Rivetti, Yann Busnel, and Achour Mostefaoui. Efficiently summarizing data streams over sliding windows. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 151–158. IEEE, 2015.

[48] Xiangyang Gou, Long He, Yinda Zhang, and etal. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *SIGKDD Conference*, 2020.

[49] Zijie Zeng, Lin Cui, Mimi Qian, Zhen Zhang, and Kaimin Wei. A survey on sliding window sketch for network measurement. *Computer Networks*, 226:109696, 2023.

[50] Yuhan Wu, Zhuochen Fan, Qilong Shi, Yixin Zhang, Tong Yang, Cheng Chen, Zheng Zhong, Junnan Li, Ariel Shtul, and Yaofeng Tu. She: A generic framework for data stream mining over sliding windows. In

*Proceedings of the 51st International Conference on Parallel Processing*, pages 1–12, 2022.

[51] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):177–188, 2004.

[52] Mea Wang, Baochun Li, and Zongpeng Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 628–635. IEEE, 2004.

[53] Cisco Systems, Cisco CNS NetFlow Collection Engine. https://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_user_guide_chapter09186a00801ed569.html, 2005.