CounterSnake: A lossless and generalized compression framework for diverse sketches

Xunpeng Liu[†]
Peking University
liuxunpeng@stu.pku.edu.cn

Qun Huang[†‡]
Peking University
huangqun@pku.edu.cn

Yaojing Wang
Huawei
wangyaojing1@huawei.com

Lihua Miao
Huawei
miaolihua4@huawei.com

Chen Sun
Huawei
sunchen48@huawei.com

ABSTRACT

Sketches are vital for large-scale stream analytics. However, they often use fixed-size counters, which remain underutilized, especially under skewed data distributions. Prior solutions to address this inefficiency compromise on accuracy, real-time operations, or generality, which limits their applicability. In this paper, we propose CounterSnake, a novel hierarchical compression framework that reduces the memory consumption of sketch counters. Compared with existing efforts, CounterSnake is the first one to fulfill four key requirements: (1) zero counter error, (2) bounded latency, (3) full counter interfaces, and (4) efficient multi-sketch optimization. The key idea is to dynamically link overflowing counters across layers to form variable-size logical counters. Besides, we design techniques such as tag-based linking, d-way mapping, sign-bit encoding, and virtual-counter abstraction to address the four requirements. We also theoretically derive its memory and time complexities under justified assumptions. Experiments against six SOTA solutions demonstrate up to several orders of accuracy improvements and comparable operation throughput. We also thoroughly evaluate CounterSnake and other frameworks, showing that CounterSnake is the only one that fulfills all the requirements.

PVLDB Reference Format:
Xunpeng Liu, Qun Huang, Yaojing Wang, Lihua Miao, and Chen Sun. CounterSnake: A lossless and generalized compression framework for diverse sketches. PVLDB, 19(4): 780 - 793, 2025.
doi:10.14778/3785297.3785316

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/N2-Sys/CounterSnake.

[†]National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University.
[‡]Qun Huang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785316

1 INTRODUCTION

Sketches have emerged as a cornerstone of large-scale data stream processing, enabling efficient approximate analytics with sublinear memory footprints [36]. These probabilistic data structures are widely adopted in database systems and distributed architectures to tackle tasks such as frequency estimation [21, 26, 38, 48], top-k identification [8, 24, 55], heavy-hitter detection [19, 43], and heavy-changer tracking [12]. Their ability to process high-velocity, unbounded streams with rigorous error bounds makes them prevalent in applications such as network monitoring [26, 42, 53], anomaly detection [25], and streaming recommendation systems [17].

While sketches are widely adopted for their efficiency, they still face the fundamental trade-off between memory consumption and estimation accuracy [39]. To achieve bounded estimation errors, sketches require a theoretically determined minimum number of counters. This requirement escalates when multiple sketches are deployed simultaneously, as is common in distributed stream processing systems [19, 22]. For instance, monitoring network traffic patterns might demand dozens of sketches, each tracking distinct metrics [34] or sliding windows [46]. Thus, maximizing the number of allocatable counters per unit memory becomes critical to balancing accuracy with practical memory constraints.

To resolve this issue, we observe that the fixed-size counters used by conventional sketches [9, 11] face significant inefficiencies in skewed data distributions. Empirical studies [13, 57] reveal that many streams exhibit a *long-tail* pattern: a small fraction of items (hot keys) dominate the updates, while the majority (cold keys) occur infrequently. The fixed-size counters waste memory for cold keys, as their low-frequency values leave high-order bits unused. This inefficiency, amplified across millions of underutilized counters, motivates the need for compression techniques that reduce wasted bits while preserving the required number of counters.

Researchers have proposed various optimizations to adapt sketch counters to skewed data distributions. Existing approaches fall into three categories. The first one is counter encoding techniques [18, 41, 49], which encode potentially large values into small fixed-size counters. These methods achieve high compression rates, but only provide coarse-grained estimations. The second one is self-adjusting counters [5, 16, 39, 54], which dynamically adjust counter sizes when overflow occurs. However, adjusting one counter often leads to adjustments of many adjacent ones, introducing unbounded latency or sacrificing accuracy. The third one is hierarchical counter structures [10, 13, 23, 29, 47, 51], which partition counters into layers and record lower-layer overflows in higher layers. However, their static mapping rules between adjacent layers introduce tradeoffs between estimation accuracy and online processing.

In summary, existing solutions suffer from three limitations. First, they struggle to achieve full accuracy with real-time responsiveness, as many techniques require offline post-processing or introduce unbounded latency during updates. Second, they lack support for diverse counter operations by mainly focusing on insertions but failing to handle deletions and negative counters efficiently. Third, they are typically confined to individual sketches, ignoring opportunities for cross-instance resource sharing in multi-sketch deployments.

In this paper, we propose CounterSnake, a novel hierarchical framework that overcomes prior limitations. Unlike previous static designs prone to hash collisions, CounterSnake dynamically links overflowing counters across layers, ensuring zero error without post-processing. To enable real-time queries, we introduce d-way mapping, a lightweight indexing mechanism that bounds probe lengths during lookups. CounterSnake's structure inherently supports insertions and deletions, while a sign-bit scheme seamlessly accommodates negative counters with the least overhead. For multi-sketch environments, CounterSnake introduces a virtual counter abstraction that allows sketches to share unused counters, reducing memory waste and enabling elastic resource allocation. Our contributions are summarized as follows:

- We propose CounterSnake, the first framework to achieve online error-free sketch compression via a dynamical counter hierarchy.
- We design techniques such as d-way mapping and sign-bit encoding to ensure real-time insertions, deletions, and queries, even for negative counters. We use a virtual counter abstraction to enable cross-instance resource sharing, minimizing memory waste.
- We prove CounterSnake's zero counter error and derive its memory and time complexities under justified assumptions.
- We integrate CounterSnake with eight sketch variants and demonstrate its superiority over state-of-the-art frameworks in memory efficiency (up to 42% less memory overhead) and sketch performance (orders of accuracy improvements under the same memory budget) across diverse workloads.

2 BACKGROUND AND MOTIVATION

2.1 Preliminaries

Data Stream Model. A data stream is formally modeled as an unbounded sequence $S = \langle (e_1, \Delta_1), (e_2, \Delta_2), \ldots \rangle$, where each item e_i belongs to a universe U, and $\Delta_i \in \mathbb{Z}$ denotes the weight of e_i. The items may arrive incrementally ($\Delta_i \geq 0$) or decrementally ($\Delta_i < 0$), reflecting real-world scenarios like network traffic monitoring (packet counts) or dynamic database updates (row insertions/deletions). The frequency of an item e, denoted $f(e) = \sum_{i:e_i=e} \Delta_i$, represents its cumulative weight in the stream. The goal of data stream processing is to estimate these frequencies or derived statistical metrics, such as heavy hitters, entropy, or quantiles. Crucially, the stream's volume and velocity render exact storage infeasible, necessitating approximate, memory-efficient processing.

Sketch Types. We broadly classify sketches into two types, i.e., counter-based sketches and bitmap-based sketches, which mainly differ in their basic units. Counter-based sketches consist of an array of counters to record numerical values, while bitmap-based sketches consist of bitmaps to record existence information. CounterSnake focuses on counter-based ones because the high-order bits in their counters tend to be unused when the stream is skewed. These sketches, including Count-Min [11], HeavyKeeper [50], HyperLogLog [14], and MinHash [7], cover a wide range of tasks and are used in production databases like Redis [3]. Bitmap-based sketches such as BloomFilter [6] and LPC [45] exhibit a different type of sparsity under skewed data streams. There are other works [32, 44, 52] that focus on optimizing bitmap-based sketches, and our work is complementary to them.

Counter Interfaces. To efficiently support the general workflow of counter-based sketches, the underlying counter abstraction must support the following interfaces:

- **Add(i, v):** Increment the value of the i-th counter by v.
- **Sub(i, v):** Decrement the value of the i-th counter by v, which is critical for handling deletions or negative updates.
- **Query(i):** Return the value of the i-th counter.
- **Reset(i):** Reset the value of the i-th counter to 0.

It is straightforward to implement these interfaces with fixed-size counters as in traditional sketches, which incurs wasted bits under skewed data distribution. This inefficiency motivates the need for counter designs that preserve the interface's semantic requirements while minimizing memory overhead.

2.2 Related Work

Counter-encoding techniques. These solutions use fixed-size but short counters to reduce memory consumption. Small values are directly recorded, but large values are encoded to fit into short counters, albeit with some accuracy loss.

For example, Small Active Counter [41] uses an encoding method similar to floating-point representation. A counter is encoded with an estimation part A, an exponent part l, and a global scaling parameter r, so its value is $A \cdot 2^{rl}$. The counter increases A with probability 2^{-rl} to provide an unbiased estimator and increases l if A overflows. Self-Adaptive Counter [49] extends this idea by dynamically adjusting the length of the exponent part, which improves the accuracy for small values. DISCO [18] uses an elaborately designed sampling probability based on a concave increasing function.

Although these methods can achieve the lowest memory consumption, they only provide an unbiased estimator of the original value. For large values, the low sampling probability results in high variance and severe accuracy loss.

Self-adjusting counters. These solutions adjust the size of each counter dynamically to fit its value. For example, SALSA [5] starts with short counters and merges neighboring counters into larger ones later. ABC [16] borrows bits from adjacent counters when overflow occurs. DHS [54] and BitMatcher [39] employ both adjustments. They organize the data structure as an array of fixed-size buckets, each containing a few counters. DHS uses three levels of counters, each with 8, 12, and 16 bits. BitMatcher uses sophisticated transition rules to split a bucket into several counters. When a bucket has no spare counters, DHS clears the smallest one while BitMatcher kicks it out to another bucket by cuckoo hashing [35].

These methods can achieve fine-grained bit allocation for their counters. However, they also sacrifice accuracy by merging or dropping counters when counter sizes cannot be properly adjusted.

Hierarchical counter structures. These methods are the most relevant to CounterSnake. They contain layers of short counters. The carry-over value from a lower-layer counter is added to several mapped counters in the upper layer. However, existing solutions face the trade-off between accuracy and online queries.

For example, Counter Braids [29] and Coding sketch [10] rely on offline post-processing to decode accurate carry-over values, which prevents them from answering online queries. Bitsense [13] adds an extra Count-Min sketch for real-time queries, which comes at the expense of lower query accuracy and higher memory usage.

On the other hand, Diamond sketch [47], Pyramid sketch [51], and Stingy sketch [23] estimate the carry-over values online with simple calculations. They are prone to huge errors because of mapping collisions in higher-layer counters. Stingy sketch [23] handles collisions by kicking one node x to another place $p(x)$ but merges the two counters if the new place is already occupied.

Offline compression methods. These methods focus on sketch compression in the offline setting, where the whole stream has been absorbed into the sketches. Hokusai [31] divides every λ counters into a group and compresses them into one counter with the sum. Cluster Reduce [56] reduces the compression error by *nearness clustering*, where counters with similar values are rearranged into one group. TreeSensing [28] separates the sketch counters into small and large ones according to their values and compresses them separately with different techniques.

These compression frameworks aim to reduce the transmission or storage overhead after processing the data stream. In contrast, CounterSnake focuses on maximizing the number of allocatable sketch counters per memory unit in online stream processing.

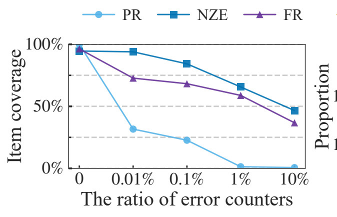
2.3 Design Motivation

To serve as a practical, general-purpose compression framework for sketches in online data stream processing, four key requirements emerge from real-world use cases:

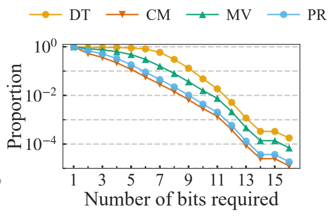
Requirement 1: Lossless Compression. Applications such as real-time service billing and precise failure localization [4] demand accurate frequency values for all items rather than only hot items. Accurate sketches such as FlowRadar (FR) [26], PR sketch (PR) [38] and NZE [21] are proposed for this purpose. These sketches involve intricate offline decoding algorithms to handle hash collisions during online processing and restore the accurate item frequencies. They are sensitive to errors in even a few counters.

We conduct empirical studies to justify this claim. We model errors commonly seen in previous solutions as two types. The first is to merge two counters and sum up their values. The second is to drop a counter and treat its value as zero. We introduce these errors by randomly choosing victim counters. We use *item coverage* as the performance metric, which is defined as the ratio of items with a relative error less than 1%. The results in Figure 1(a) show that even a small ratio of error counters would cause a catastrophe in the performance. For example, 0.1% of error counters can degrade the item coverage of PR sketch to 25% and FlowRadar to 60%.

Existing solutions only provide coarse-grained accuracy guarantees, such as the incorrect rate for an arbitrary counter. For these accurate sketches, these errors severely degrade their performance. For normal sketches, there is no general way to incorporate these



(a) Impact of counter errors



(b) Diverse counter distribution

Figure 1: Empirical studies on sketch counters

incorrect rates into the original error bound analysis. Thus, we need a lossless compression framework that provides correctness guarantees uniformly for all counters rather than for an arbitrary counter to preserve the performance of the original sketches.

Requirement 2: Constant-time Operations. Online processing tasks like network intrusion detection [15] or algorithmic trading are sensitive to processing latency. For example, in modern data centers where traffic exceeds 100 Gbps or even Tbps [37], packets arrive at sub-microsecond intervals. Network measurement tasks [42] in this setting demand *line-rate processing*, where insertions or queries triggered by one packet must resolve before the next packet arrives. Non-constant-time operations in line-rate processing can quickly overwhelm packet buffers, leading to drops and missed detections.

We find that many existing optimization frameworks fail in this regime in two ways. First, their operations can only achieve amortized $O(1)$ complexity, but may be non-constant in the worst cases. For example, Stingy sketch [23] kicks out a counter to another place to deal with collisions, which may cause cascading kick-outs. Second, they rely on offline post-processing to answer queries. For example, Coding sketch [10] needs to first decode counters with no collisions, and then decode other counters iteratively. In summary, a viable framework must guarantee strict $O(1)$ time complexity for all counter operations to fully support line-rate processing.

Requirement 3: Full Counter Interfaces. The general stream model allows arbitrary weights of items. Therefore, many sketches involve deletions or negative updates, and sometimes even require negative counters. For example, Count sketch [9] ensures unbiased estimation by storing dot products between item frequencies and random ± 1 vectors. This necessitates native support for arbitrary values (including negative ones) during both updates and queries.

We analyze existing solutions and find that counter-encoding techniques and self-adjusting counters can naturally support these interfaces. However, hierarchical solutions struggle with deletions and negative counters for two reasons. First, their many-to-one mapping rules introduce collisions, which may cancel out carry-over values from positive and negative counters, causing huge estimation errors. Second, naively using two's complement representation for negative values can cause excessive overflows in all layers when switching from positive to negative. In conclusion, we need a counter-optimization framework that can support full counter interfaces with minimum overhead.

Requirement 4: Efficient Handling of Multiple Sketches. Production systems often deploy multiple sketches concurrently. For example, a network monitoring service might use a Count-Min sketch for traffic analysis, a HeavyKeeper [50] for detecting DDoS

Table 1: Limitations of Existing Works

Types	Sketches	R1	R2	R3	R4
Encoding		✗	✓	✓	✗
Self-adjusting		✗	✓	✓	✗
Hierarchical	CounterBraids	✓	✗	✗	✗
	Coding, Bitsense	✓	✗	✓	✗
	Stingy, Pyramid	✗	✓	✗	✗
	Diamond	✗	✓	✓	✗
	CounterSnake	✓	✓	✓	✓

attackers, and a Burst sketch [33] for burst detection. Although the concept of universal sketches [20, 27, 48] has been proposed to estimate multiple metrics with one sketch, it does not eliminate the need for multiple sketches, especially in tasks involving multiple attributes [30] or sliding windows [46].

We observe that different sketches often have diverse counter distributions. Figure 1(b) shows the distributions of counter values in four sketches, which exhibit huge differences. For example, there are merely 1% counters that require 8 bits or more to store their values in Count-Min sketch (CM), while this number is 30% in Deltoid (DT). This leads to potentially more underutilized counters when optimizing the Count-Min sketch, which can be shared by Deltoid to enhance memory efficiency.

Existing solutions have two main limitations in supporting multiple sketches. First, some of them [5, 47] design different insertion or query strategies for different sketches. They do not provide a uniform view of sketch counters, failing to deal with heterogeneous sketches that are deployed simultaneously. Second, others [13] manage multiple sketch instances in isolation, ignoring opportunities for resource sharing. Therefore, it is highly motivated to design a framework that optimizes multiple sketches as a whole to provide a uniform view of counters and enable cross-sketch resource sharing.

Conclusion. We summarize in Table 1 the limitations of existing solutions based on the previous four requirements. In conclusion, CounterSnake is the only one that satisfies all the requirements to become a general-purpose compression framework.

3 THE COUNTERSNAKE FRAMEWORK

3.1 Overview

CounterSnake is designed to be a general-purpose optimization framework that adapts sketch counters to skewed data distributions. We have already seen in Section 2.3 that such a framework should satisfy four requirements (denoted as R1-R4) to apply to a wide range of data processing tasks. In this section, we present the design of CounterSnake and how it resolves the limitations of existing works by fulfilling these requirements. We list symbols that appear in this section and their definitions in Table 2 for reference.

The following subsections are organized as follows. In Section 3.2, we present the hierarchical data structure of CounterSnake and its two techniques to resolve the trade-off between accuracy and real-time queries. The first technique is *dynamic linking*, which erases mapping collisions in previous solutions and ensures lossless compression (R1). The second technique is *d-way mapping*, which limits the maximum number of lookups to find the mapped counter and thus ensures constant-time operations (R2). In Section 3.3, we introduce how to implement the counter interfaces in Section 2.1 with the data structure (R3). In Section 3.4, we introduce how to adapt CounterSnake to support negative counters (R3). Finally, in Section 3.5, we describe how CounterSnake manages counter resources across multiple sketches while providing a uniform virtual counter abstraction to conceal the underlying details (R4).

Table 2: Symbol Definitions

Symbol	Definition
l	The number of counter layers
n_i	The number of counters in layer i
w_i	The size of a counter in layer i
s_i	The number of counters in a group in layer i
d_i	The number of linking choices in layer i
id	The index of a counter
$tagId$	id modulo s
$groupId$	The integer division of id by s

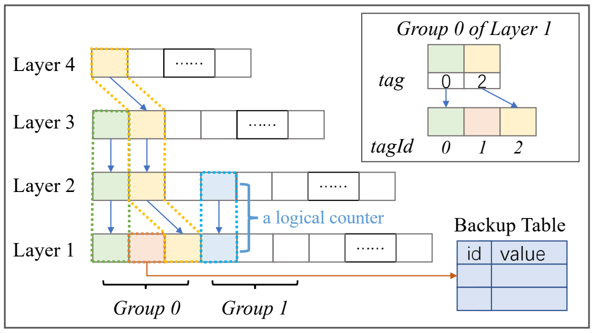


Figure 2: The hierarchical data structure of CounterSnake

3.2 Data Structure

Hierarchical data structure. As shown in Figure 2, the main structure consists of l layers of counters, labeled as $1 \sim l$ from bottom to top. Assume there are n_i counters in the i-th layer, and each one contains w_i bits. The basic idea is to record the carry-over value of the i-th layer to the $(i+1)$-th layer. Because of the highly skewed data distribution, only a small fraction of counters overflow at each layer. So we can set n_{i+1} to be significantly smaller than n_i to save memory. CounterSnake leverages two novel techniques to resolve the limitations of previous hierarchical frameworks:

Dynamic linking. When a counter at the i-th layer overflows, existing solutions map it to multiple counters in the $(i+1)$-th layer, leading to collisions and inaccurate estimation. In contrast, CounterSnake maps it to exactly one counter. By dynamically linking counters in adjacent layers, CounterSnake incrementally builds variable-size counters. We define these composed counters made of different layers of counters as *logical counters*. We mark four logical counters in Figure 2. This linking is realized through a pointer attached to each counter, which is called a *tag*. We initialize each tag as a reserved value *null*, indicating an unused counter. These tags point top-down, meaning a counter C_2 in the $(i+1)$-th layer holds the tag pointing to a counter C_1 in the i-th layer. However, operations on logical counters often rely on bottom-up search, i.e., we need to locate C_2 given C_1. This can be done by enumerating all tags in the $(i+1)$-th layer and finding which points to C_1.

D-way mapping. The brute-force method to do a bottom-up search does not satisfy the constant-time requirement. To ensure bounded search latency and reduce pointer sizes, CounterSnake limits the number of possible linking choices of each counter to a constant d. Specifically, we bundle every s_i counters in layer i into a *group*. The counter with index id is in the id/s_i-th group, which we define as *groupId*. This counter is the $id\%s_i$-th counter in that group, which we define as *tagId*. The k-th group is associated with d_i counters in layer $i+1$, ranging from kd_i to kd_i+d_i. If a counter C_1 overflows, we will link it to one of the d_i associated counters (C_2) and set C_2's tag to the *tagId* of C_1. In the top-right corner of Figure 2, we detail the mappings between *Group 0* in *Layer 1* and the associated counters in *Layer 2*. Here, each group in *Layer 1* contains three counters, and they are associated with two counters in *Layer 2*.

Excessive overflows. The d-way mapping technique can handle no more than d_i number of overflows in each group. To attain fully lossless compression, we add a small hash table called *Backup Table* to handle the excessive overflows. The key of the Backup Table is the index of the logical counter (i.e., that of its bottom-layer counter), and the value is a full-size counter. If we find no empty counter in the d_i associated ones, we search top-down to free occupied counters and gather their values to rebuild the logical counter. Then we insert it into the Backup Table. Later operations on this logical counter will be redirected to the Backup Table. In Section 4.2, we will discuss how to configure the table size to handle all excessive overflows and thus ensure zero error with high probability.

Discussion. There are two possible concerns about our design. The first one is why we organize pointers in a top-down manner, which complicates the bottom-up search. The main reason is that if we organize pointers in the other direction, then we need to attach tags to the lower-layer counters. This will incur more overhead because the lower layer contains many more counters. The second one is whether the multi-bit tags are more costly than the status bits used in prior methods. We find that prior methods attach one status bit for each counter in all layers, while CounterSnake does not attach tags to the bottom layer. As only a small number of counters overflow, the number of tags in CounterSnake is far less than the number of status bits in other solutions. This is especially true in our later experiments on real-world datasets with high skewness.

3.3 Counter Operations

Based on the hierarchical data structure, CounterSnake provides four counter interfaces. These interfaces are designed to operate in constant time (R2), which is in line with the online properties of data streams. For example, processing continuously-arrived items requires fast updates to sketch counters, while answering real-time queries on item frequencies demands quick counter lookups. In this subsection, we focus on non-negative logical counters and leave the discussion of negative counters to the next subsection.

Algorithm 1 Add Interface

```
1: procedure ADD($id, v$)                              ▷ Add value $v$ to counter $id$
2:     if $id$ is in the Backup Table then
3:         $BT[id] \leftarrow BT[id] + v$
4:     else
5:         for $i \leftarrow 1$ to $l$ do
6:             $u \leftarrow (D_i[id] + v)/2^{w_i}$
7:             $D_i[id] \leftarrow (D_i[id] + v)\%2^{w_i}$
8:             if $u == 0$ then break                      ▷ No overflow
9:             $v \leftarrow u$
10:            $setId, tagId \leftarrow id/s_i, id\%s_i$
11:            if $tagId$ in $T_{i+1}[setId * d_i : setId * d_i + d_i]$ then
12:                $id \leftarrow$ index of the matched tag
13:            else if $null$ in $T_{i+1}[setId * d_i : setId * d_i + d_i]$ then
14:                $id \leftarrow$ index of the null tag
15:                $T_{i+1}[id] \leftarrow tagId$
16:            else                                      ▷ No free counter
17:                Insert this logical counter into BT
18:                break
```

Add. The details of the Add interface are shown in Algorithm 1. We denote the counter in layer i as an array D_i, and the tags attached to them as an array T_i. We first check if the logical counter id is stored in the Backup Table and redirect the addition if so (line 2). If not, we update counters from the bottom layer to the top layer (line 5). In each layer, we assume that id is the index of the counter to be updated and that v is the value to be added. So we add the value v to the counter with index id (line 7). If there is an overflow, we update v as the carry-over value (line 9), which will be added in the next layer. Then we match $tagId$ with the d_i tags in the next layer, ranging from $setId*d_i$ to $setId*d_i+d_i$. There are three cases:

- There is a matched tag in this range (line 11). So we update id as the index of this tag and go to the next layer.
- There is a *null* tag (line 13). This means the counter overflows for the first time, so we update id, set this tag, and go ahead.
- No tag matches or is *null* (line 16). This means all the d_i counters are used up. So we search top-down, free the occupied counters, and insert the logical counter into the Backup Table (line 17).

The top-down search in the last case is straightforward. Suppose the counter index in the $(i+1)$-th layer is id, then the counter index in the i-th layer is $id/d_i \times s_i + T_{i+1}[id]$. We can assemble the counters along the way to get the logical counter and do the insertion.

Sub. The Sub interface is similar to Add, so we omit its code. The main difference is that a counter overflows if it becomes negative after subtraction, referred to as an underflow. To handle this underflow, we need to borrow values from the linked counter in the upper layer. Also, we can free an occupied counter for memory efficiency if (1) its value becomes zero after subtraction, (2) it is in the highest layer of the logical counter, and (3) it is not a bottom-layer counter. To free an occupied counter, we set its tag to *null*, enabling other counters in the lower layer to use this counter later. This procedure is conducted iteratively from the highest layer, because counters in other layers may be freed after we free the highest-layer one.

Algorithm 2 Query Interface

```
1:  procedure QUERY($id$)                    ▷ query the value of counter $id$
2:      if $id$ is in the Backup Table then
3:          $v \leftarrow \text{BT}[id]$
4:      else
5:          $v \leftarrow D_1[id]$
6:          $bits \leftarrow w_1$
7:          for $i \leftarrow 2$ to $l$ do
8:              $setId, tagId \leftarrow id/s_{i-1}, id\%s_{i-1}$
9:              if $tagId$ in $T_i[setId * d_{i-1} : setId * d_{i-1} + d_{i-1}]$ then
10:                 $id \leftarrow$ index of the matched tag
11:                 $v \leftarrow v + D_i[id] \times 2^{bits}$
12:                 $bits \leftarrow bits + w_i$
13:             else                             ▷ No linked counter
14:                 break
15:     return $v$
```

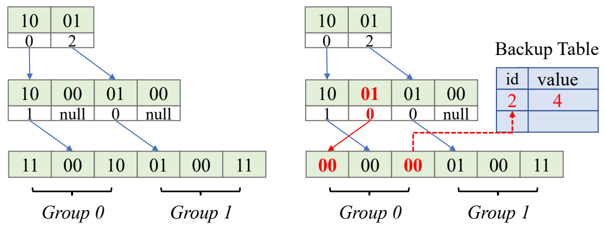
Query. Algorithm 2 shows how to answer a real-time query of a logical counter. We first check if the index is in the Backup Table, and return the value if so (line 2). Otherwise, we query counters from the bottom layer to the top layer. In each layer, we assume id to be the queried index, v to be the current result, and $bits$ to be the total number of bits up to this layer. We begin with layer 1 and get the value v from the counter id (line 5). To check if there is a linked counter in the next layer, we calculate $setId$ and $tagId$ and match $tagId$ as before (line 8). If there is a matched tag, we update id as the index, add the shifted counter value to v, update $bits$, and go to the next layer (line 10-12). Otherwise, we break from the loop (line 14) and return the value v as the final result.

Reset. The `Reset` operation is similar to `Query`. We first check if the logical counter is in the Backup Table. If so, we delete this entry and reset the bottom-layer counter of this logical counter to zero. Otherwise, we locate the components of this logical counter in a bottom-up manner as in `Query`, and reset their values and tags.

Examples. Figure 3 illustrates the details of the `Add` and `Query` operations. There are two groups in *Layer 1*, each associated with two counters in *Layer 2* ($s_1 = 3, d_1 = 2$). All counters in *Layer 2* are in the same group, sharing the two counters in *Layer 3* ($s_2 = 4, d_2 = 2$).

Suppose we first add logical counter 0 with value 1. The data structure before this operation is shown in Figure 3(a). We first add 1 to counter 0 in layer 1, which causes an overflow. Its $tagId$ is $0\%3 = 0$, so we check the first two tags in layer 2 and find that there is no match. So we choose counter 1, which has a *null* tag, update its tag, and record the carry-over value. Suppose we then add logical counter 2 with value 2. We first add 2 to counter 2 in layer 2, which also causes an overflow. By checking the first two tags, we find there is neither a matched one nor a free one. So we insert this logical counter into the Backup Table. The data structure after these two operations is shown in Figure 3(b).

Suppose we want to query logical counter 3 in Figure 3(a). We first query counter 3 in layer 1 and get value 1. Its $setId$ is $3/3 = 1$ and $tagId$ is $3\%3 = 0$, so we check the last two tags in layer 2. The result shows that counter 2 in layer 2 has the tag 0, whose value is 1. We calculate and match the $tagId$ similarly, and find counter 1 in layer 3 is linked. So the final query result is $1 * 16 + 1 * 4 + 1 = 21$.



(a) The initial state (b) The state after two Adds

Figure 3: An example of the Add interface

3.4 Extension to Negative Counters

In this subsection, we extend our data structure to support signed logical counters. Directly adopting the previous counter hierarchy with two's complement representation leads to inefficiency for small-magnitude negatives. This is because they are encoded with maximal leading 1s (e.g., -1 as 0xFF...FF), occupying counters in all layers. However, in real-world distributions, small-magnitude values dominate sketch counters. To solve this inefficiency, we propose to use a sign-bit representation that decouples sign handling from magnitude storage. Each counter in the bottom layer has a sign bit indicating positivity (0) or negativity (1). The hierarchical structure now stores the absolute value of the logical counters.

We want to efficiently extend the `Add` and `Sub` interfaces to handle possible sign-bit flips. Specifically, they should involve only one pass of bottom-up update as before when the sign bit is not changed. The main difficulty is that we cannot predict whether the sign bit will be flipped until we reach the highest layer. To solve this problem, we assume an unchanged sign bit while doing the bottom-up update, but also compute the results in the other branch. So if the sign bit is truly flipped later, we can set the counter hierarchy as the computed results in one pass of top-down update.

We use `Add` as an example, and the extension of `Sub` is similar. We first check the sign bit of the logical counter. If it is positive, then we just update the data structure as before. Otherwise, we should subtract v from the current magnitude. Assume that in the i-th layer, the counter value is c_i and the value to be subtracted is v_i. We update the counter as $c_i - v_i$, and record $v_i - c_i$ in a temporary register as the result for a flipped sign bit. If there is an underflow in the highest layer of the logical counter, we should flip the sign bit. Then we add the underflow value to higher layers and restore the lower layers to the recorded values in a top-down manner. This procedure ensures one pass of bottom-up update when the sign bit is not flipped and an additional pass of top-down update otherwise. As the sign bit is not changed in most cases, this extension adds little overhead to `Add` and `Sub` on average.

3.5 Multi-sketch Compression

While the hierarchical counter structure enables dynamic, lossless compression for individual sketches, real-world systems often deploy multiple heterogeneous sketches. The coexistence of multiple sketches introduces the following two critical challenges:

The first challenge is how to manage counters from different sketches in a uniform view. Each sketch expects its counters to

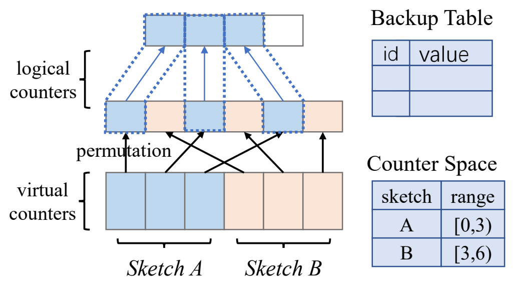


Figure 4: Mapping of virtual counters to logical ones

behave as a contiguous array, even though their physical storage is distributed across hierarchical layers. To distinguish from the physical counter layers, we refer to this contiguous view of counters as *virtual counters*. Each virtual counter corresponds to a logical counter in the hierarchical structure. However, naively partitioning the hierarchical structure into per-sketch regions would undermine compression efficiency. This is because different sketches exhibit distinct counter value distributions, leading to many excessive overflows in some regions, while many unused counters in others.

To balance compression efficiency and cross-sketch uniformity, we map the virtual counter space of each sketch to the physical hierarchy via a prime-based permutation. For the i-th sketch allocated with n_i virtual counters starting at offset o_i, its virtual counter id $\in [0, n_i)$ is mapped to physical address $p \times (o_i + \text{id}) \mod N$. Here N is the total number of logical counters (i.e., the number of bottom-layer counters). To ensure this mapping is indeed a permutation, N can be any value, but p should be a prime number not dividing N. This permutation ensures counters from different sketches are interleaved uniformly across the hierarchy to distribute overflows evenly. Sketches interact with their $[0, n_i)$ range as a virtual array, unaware of the underlying permutation. Figure 4 illustrates how this permutation enables better utilization of counters in the higher layers while preserving the contiguous view of virtual counters.

The second challenge is how to dynamically allocate and deallocate sketch instances. In windowing aggregation that involves data aging, sketches in outdated windows must be rapidly deallocated to free resources for new ones. Directly resetting all virtual counters in a sketch would incur $O(n_i)$ time, as their physical storage is scattered. To achieve constant-time deallocation, we introduce the *lazy reset* technique, which defers the actual reset cost.

Specifically, we divide the virtual counter space into fixed-size chunks. Each chunk is associated with a validity tag indicating which sketch currently owns it. When allocating a new sketch with index sk, we assign it a set of chunks from previous outdated sketches. Upon the first access to a chunk by sketch sk, if its validity tag does not match sk, we perform an $O(1)$ reset on this chunk and update the validity tag to sk. Therefore, the total cost is amortized to $O(1)$ per operation, rather than $O(n_i)$ upfront during deallocation.

Note that this *lazy reset* differs from the `Reset` operation in Section 3.3. The `Reset` operation targets a single logical counter by setting its value to zero. In contrast, *lazy reset* is a bulk operation that frees a range of counters instantly and defers the reset of individual counters until their reuse.

4 ANALYSIS

In this section, we analyze CounterSnake's memory consumption and time complexity to ensure zero error with high probability. Our analysis is based on two assumptions, which are stated and justified in Section 4.1. Then, we formally derive the memory consumption of CounterSnake in Section 4.2 and verify the constant-time complexity of its operations in Section 4.3.

4.1 Assumptions

Assumption 1. *The proportion p_i of counters that overflow in the i-th layer is pre-known.*

We define this proportion as *overflow rate*. This assumption states that we have some prior knowledge about the characteristics of the data stream distribution. We can further relax this assumption to knowing an upper bound of p_i. Note that all existing lossless compression frameworks [10, 13, 29] require similar knowledge in their analysis. These rates can either be obtained from historical data or be estimated by some reasonable upper bounds.

Assumption 2. *The counters in the same layer overflow independently with the same probability.*

There are two reasons to support this assumption. First, many sketches use hash functions to map one item to several counters, which brings randomness into the counting process. Second, for sketches without hashing, we distribute their neighboring virtual counters into different groups by a prime-based permutation as described in Section 3.5. This permutation also mixes counters from different sketches to ensure a uniform overflow rate in each group.

4.2 The Memory Consumption

The data structure of CounterSnake consists of the counters, the tags, and the Backup Table. To calculate their memory consumption, we need to analyze the number of excessive overflows. Excessive overflows arise when there are more than d overflowed counters in a group of s counters. In this case, only the first d overflowed counters are handled in the upper layer, and all later overflowed ones are inserted into the Backup Table. The following theorem gives the probability that excessive overflows occur in a group:

Theorem 1. *Suppose the overflow rate is p, then the probability that a group of s counters has more than d overflowed counters is*

$$\Pr[\exists \textit{ excessive overflow}] = \sum_{i=d+1}^{s} \binom{s}{i} p^i (1-p)^{s-i} \quad (1)$$

The proof of this theorem is by direct computation. According to our two assumptions, the number of overflowed counters in a group follows $\text{Binom}(s, p)$. For a large enough s, it is reasonable to approximate this binomial distribution by a normal distribution $N(sp, sp(1-p))$, which gives the following corollary:

Corollary 1. *To bound the probability that a group has excessive overflows under ϵ, the smallest d is given by*

$$d_{\min} = \min\left(sp + \sqrt{sp(1-p)}\Phi^{-1}(1-\epsilon), s\right) \quad (2)$$

, where $\Phi(x)$ is the CDF of the standard normal distribution.

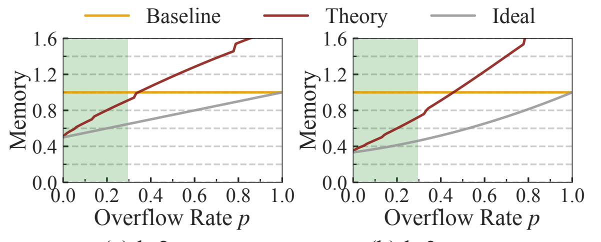


Figure 5: Memory comsumption results

We take the minimum here because excessive overflows can only occur when $d \geq s$. With these results, we can bound the total number of groups with excessive overflows using Chernoff Bounds:

Theorem 2. *Suppose the probability that a group has excessive overflows is under ϵ and there are k groups. Denote the number of group that has excessive overflows as X, then*

$$\Pr[X \geq (1+r)(\epsilon k)] \leq \exp\left(-\frac{r^2(\epsilon k)}{2+r}\right) \quad (3)$$

As one group has at most s excessive overflows, we can bound the total number of excessive overflows in a layer as follows:

Corollary 2. *Suppose there are N counters in a layer, and we set $d = \min\left(sp + \sqrt{sp(1-p)}\Phi^{-1}(1-\frac{cs}{N}), s\right)$. Then with probability at least $1-\delta$, the number of excessive overflows in this layer is $\leq s(3c - \ln\delta)$. Here c is an arbitrary positive constant.*

We can simply set $k = \frac{N}{s}$, $\epsilon = \frac{cs}{N}$ and $r = 2 - \frac{\ln\delta}{c}$ in Equation 3 to derive this corollary. Apply this result to all l layers, we have the following theorem:

Theorem 3. *Suppose there are $n_1 = N$ counters in the first layer, and we set $d_i = \min\left(s_i p_i + \sqrt{s_i p_i(1-p_i)}\Phi^{-1}(1-\frac{cs_i}{n_i}), s_i\right)$, $n_{i+1} = \frac{n_i d_i}{s_i}$. Then with probability at least $(1-\delta)^{l-1}$, the total number of excessive overflows in all layers is $\leq (3c-\ln\delta)\sum_{i=1}^{l-1} s_i$.*

In practice, we can fix the table size as $(3c-\ln\delta)\sum_{i=1}^{l-1} s_i$. If there are more excessive overflows, we simply drop their values. This static memory allocation ensures worst-case memory and time complexities, while also achieving lossless compression with high probability. In conclusion, we have the following result:

Theorem 4. *Suppose there are $n_1 = N$ counters in the first layer, and the counters in the i-th layer have w_i bits. If we set d_i, n_i and the table size as before, then CounterSnake achieves lossless compression with probability at least $(1-\delta)^{l-1}$, and its total number of bits is*

$$\text{Mem} = \sum_{i=1}^{l} n_i w_i + \sum_{i=2}^{l} n_i \lceil \log(d_i+1) \rceil + (3c-\ln\delta)\sum_{i=1}^{l-1} s_i \sum_{i=1}^{l} w_i \quad (4)$$

Discussion. In Equation 4, $l, N, c, \delta, w_i, s_i$ are pre-configured parameters and d_i, n_i are determined by overflow rate p_i. So it actually gives the relationship between memory consumption and p_i. To understand CounterSnake's memory saving under different overflow rates, we illustrate the theoretical results in Figure 5. We choose the same parameters as in later evaluations on real-world datasets, namely $s_i = 144$, $c = 16$, $\delta = 0.01$, $w_i = 6$, and $N = 1$M.

The two figures in Figure 5 demonstrate the results when $l = 2$ or 3 layers are used. Each figure contains three lines, indicating the baseline results of full-sized counters, the theoretical results in Equation 4, and the ideal results. The ideal case means we allocate just-enough counters in the $(i+1)$-th layer to handle overflows from the i-th layer without the overhead of extra data structures. For example, when $l = 2$, the ideal memory consumption is $w_1 N + w_2 N p$. The memory consumption is normalized against the baseline.

The results for $l = 2$ show that CounterSnake saves more memory than its overhead when $p < 0.34$. From empirical studies in Section 2.3, we find that the overflow rates are under 0.3 for all tested sketches (the green region). In practice, CounterSnake performs better than illustrated because we overestimate the size of the Backup Table in the previous analysis. The results for $l = 3$ show that CounterSnake performs better under the same overflow rates when more layers are used. This gives us heuristics about parameter choosing that we should use more layers while ensuring safe overflow rates (e.g., in the green region).

4.3 Constant-time Operations

In this subsection, we analyze the worst-case time complexities of the four operations described in Section 3.3. These operations mainly involve three routines, namely bottom-up search, top-down search, and searching the Backup Table. The bottom-up search requires at most l counter accesses, $\sum_{i=1}^{l-1} d_i$ tag checks and $l-1$ arithmetic operations. The top-down search is much easier, which requires at most l counter accesses, $l-1$ tag checks, and $l-1$ arithmetic operations. Locating an entry in the Backup Table costs at most $(3c-\ln\delta)\sum_{i=1}^{l-1} s_i$ steps by searching the whole table.

Add. For unsigned counters, the worst case of Add happens as follows. We do not find the logical counter in the Backup Table, do normal updates, encounter an excessive overflow, reset the hierarchy, and then insert the value into the Backup Table. Note that we have located the desired entry during the first search of the Backup Table, so later insertion does not require searching again.

For signed counters, the worst case depends on whether the sign bit is flipped. If not, the worst case is the same as before. Otherwise, the worst case happens when we record the final underflow value to higher layers and then encounter an excessive overflow. But the time complexities of these two worst cases are the same, both involving one pass of bottom-up search, one pass of top-down search, and one search of the Backup Table. This is $5l - 3 + \sum_{i=1}^{l-1} d_i + (3c-\ln\delta)\sum_{i=1}^{l-1} s_i$ steps in total. Note that we have $d_i \leq s_i$. As l, s_i, δ and c are all pre-configured parameters independent of N, we can conclude that Add is of $O(1)$ time in the worst case.

Sub. The procedure of the Sub operation is basically the same as that of Add on negative counters. So its worst-case time complexity is also $5l - 3 + \sum_{i=1}^{l-1} d_i + (3c-\ln\delta)\sum_{i=1}^{l-1} s_i$.

Query. To Query a counter, we need to first check whether it is in the Backup Table. If not, we then perform the bottom-up search. So the worst-case time complexity is $2l-1+\sum_{i=1}^{l-1} d_i + (3c-\ln\delta)\sum_{i=1}^{l-1} s_i$.

Reset. The procedure of the Reset operation is basically the same as that of Query. We need to check the Backup Table and reset the counters and tags through one pass of bottom-up search. So the worst-case time complexity is also $2l-1+\sum_{i=1}^{l-1} d_i + (3c-\ln\delta)\sum_{i=1}^{l-1} s_i$.

5 EVALUATION

In this section, we evaluate CounterSnake to demonstrate that it outperforms existing frameworks in terms of sketch performance and memory efficiency (§5.2). Furthermore, we demonstrate through a series of experiments that CounterSnake fulfills the four requirements R1-R4 in Section 2.3 (§5.3). We also provide some empirical results on CounterSnake's parameters and its lazy reset technique (§5.4). We summarize our findings of CounterSnake as follows:

- It outperforms six existing frameworks by achieving the best performance on various tasks and datasets, with competitive insertion speed and slightly degraded query speed as the cost of lossless compression (Exp#1).
- It adjusts the size of each counter close to the minimum size required by zero-error compression, and it also has negligible memory overhead in auxiliary data structures (Exp#2).
- Its performance remains stable across various datasets with different skewness, demonstrating its broad adaptivity to diverse data distributions (Exp#3).
- It is the only one to achieve zero counter error in all three sketches that cover all counter interfaces, thus fulfilling requirements R1 and R3 (Exp#4).
- Its worst-case number of memory accesses is comparable to that of the average case, thus fulfilling the requirement R2 (Exp#5).
- It achieves better memory efficiency in multi-sketch compression compared with per-sketch optimization, thus fulfilling the requirement R4 (Exp#6).
- We can carefully choose the parameters s_i and c to minimize CounterSnake's memory consumption (Exp#7).
- The lazy reset technique helps to amortize the cost of sketch deallocation in windowing processing tasks (Exp#8).

5.1 Experiment Setup

Implementation. We implement CounterSnake as a lightweight C++ library, exposing lossless counter interfaces through thread-safe APIs. For high-speed network scenarios, we also prototype CounterSnake in P4 language targeting Tofino switches.

Test platforms. We test the software prototype of CounterSnake and the compared frameworks on a server equipped with two 12-core Intel Xeon Gold 5118 @ 2.30 GHz CPUs and 128 GB memory. For the P4 implementation, we deploy the data-plane code on a BFN-T10-064Q Tofino switch. We also write a control-plane application on a server connected to the switch to communicate with the data-plane program to verify its functionalities.

Compared solutions. We compare CounterSnake (CSK) with SOTA solutions in the three categories mentioned in Section 2.2. For counter-encoding techniques and self-adjusting counters, we choose Self-Adaptive Counter (SAC) [41] and BitMatcher (BM) [39] as representatives. For hierarchical structures, we implement both lossless frameworks, such as Bitsense (BS) [13] and other frameworks without lossless guarantees, such as Pyramid (PY) [51], Stingy (ST) [23], and Diamond (DIA) [47]. We also include the baseline results of directly using fixed-size counters (FULL).

Datasets. We use various datasets in our evaluations, which include (i) Caida [1], a real IP backbone trace, which contains about 25.1M items (packets) and 1.0M distinct ones (network flows), (ii) Kosarak [2], a click-stream dataset, which contains about 8M items and 41K distinct ones, and (iii) Zipf, which is a series of synthetic datasets (each containing about 25M items) generated by ourselves according to Zipf distributions with different skewness (ranging from 0.0 to 2.0).

Sketches and tasks. We integrate eight sketches with CounterSnake and also test other frameworks with some of these sketches. We use these sketches to handle two types of data-stream processing tasks, namely frequency estimation and top-k item identification. Specifically, we use Count-Min (CM) [11], Elastic (ES) [48], FlowRadar (FR) [26], PR [38] to perform frequency estimation and Deltoid (DT) [12], HashPipe (HP) [40], MV [43], SketchLearn (SL) [20] to perform top-k identification. For top-k identification, we set k to be 5% of the number of distinct items.

Metrics. We use Average Absolute Error (AAE) and Average Relative Error (ARE) as the metrics of frequency estimation tasks. AAE is defined as $\frac{1}{n}\sum_{i=1}^{n}|\hat{f_i}-f_i|$, where n is the number of distinct items, $\hat{f_i}$ is the estimated frequency of the i-th item, and f_i is the true frequency of that item. ARE is similarly defined as $\frac{1}{n}\sum_{i=1}^{n}|\hat{f_i}-f_i|/|f_i|$. We use F1-score as the metric of top-k identification tasks, which is defined as the harmonic mean of the precision and recall. A lower AAE/ARE or a higher F1-score means better performance in corresponding tasks.

5.2 General Performance

(Exp#1) Sketch performance. We use two real-world datasets to test the performance of CounterSnake and existing frameworks on two tasks, namely frequency estimation and top-k identification. We use CM as the underlying sketch for frequency estimation and HP for top-k identification. For fair comparisons, we fix the memory budget of each framework, ranging from 20KB to 8 MB, in different tasks and datasets. Note that there are some missing points for ST on the Kosarak dataset (also on some of the Zipf datasets in later experiments). This is because it cannot function properly when the overflow rate is high, which causes too many kick-outs.

The results of task performance are illustrated in Figure 6(a)-(d). For frequency estimation, CounterSnake achieves the lowest AAE across all experiments on the two datasets. The most competitive framework is BS, as it also achieves lossless compression in this task. Other frameworks introduce errors in sketch counters. Therefore, their AAEs decrease slowly even when abundant memory is provided (8MB). For top-k identification, the objective is to identify the items with the top-5% highest frequency. In this task, small errors on sketch counters do not influence the F1-scores much, as long as small values are not mistakenly restored as large ones. However, the number of allocatable counters matters a lot because the underlying sketch can maintain more item frequencies with more counters, which improves the identification accuracy. We find that CounterSnake and SAC have the highest F1-scores because they achieve the lowest average bits per counter. Although SAC is not lossless, it only introduces errors on counters with large values, which does not impact top-k identification much. Other frameworks have lower F1-scores because they either introduce huge errors on small values or allocate more bits per counter. We also observe that the baseline solution tends to outperform some compression frameworks in both tasks, given enough memory. The reason is that the benefits from memory saving are insufficient to compensate for

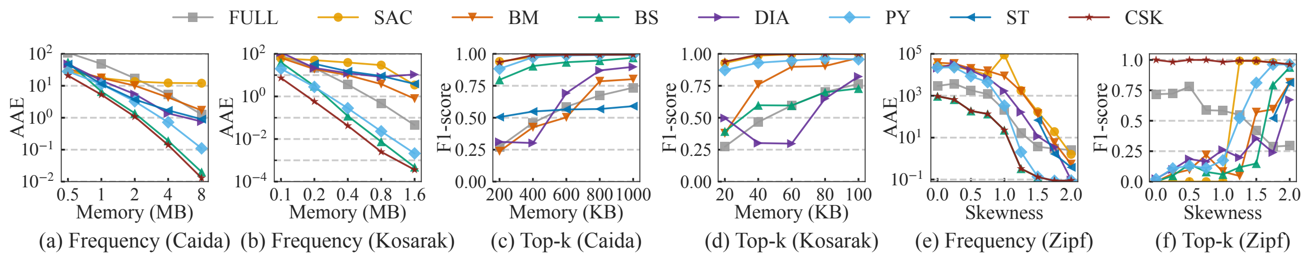


Figure 6: Performance of different compression frameworks

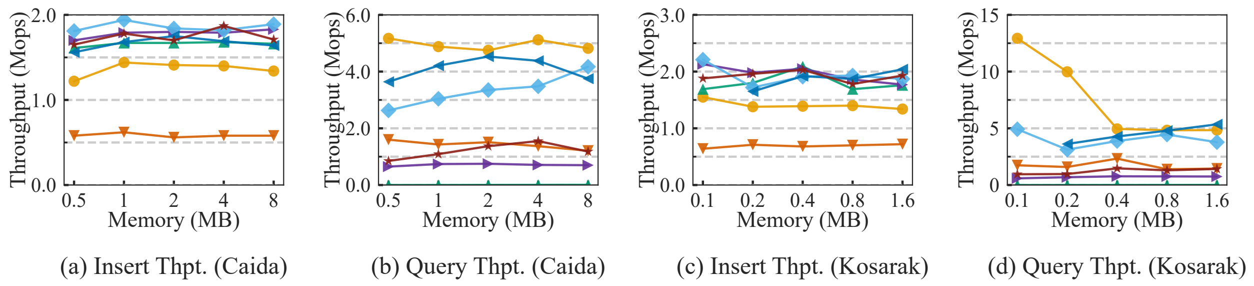


Figure 7: Insertion and query throughput on frequency estimation

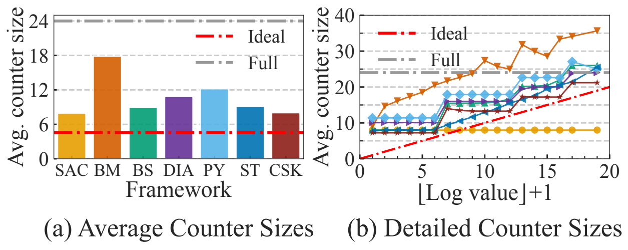


Figure 8: The counter sizes of different frameworks

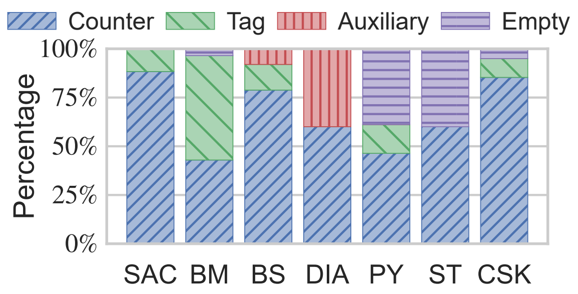


Figure 9: The memory breakdown of different frameworks

counter errors when the number of counters is already abundant. This also shows the necessity of a lossless compression framework to ensure better performance in all memory budgets.

The results of operation speed are illustrated in Figure 7. Here, we only list the results on frequency estimation because the trend in top-k identification is similar. To avoid CPU jitters, we run each experiment five times and report the median result. We can see CounterSnake's insertion throughput is among the highest, but the query speed is lower than the top ones. This is because CounterSnake needs to examine multiple tags to determine the counter in the next layer. PY and ST do not pay this effort owing to their binary tree structure, but their accuracy drops when both siblings overflow. We consider this cost acceptable, as other lossless frameworks (like BS) do not support lossless real-time queries.

(Exp#2) Memory efficiency. We also conduct experiments to compare the effectiveness of different compression frameworks on memory saving. First, we examine the number of bits that are allocated for each counter. For a counter with value $v > 0$, we need at least $\lfloor \log v \rfloor + 1$ bits to represent its value with no error. So we define $\lfloor \log v \rfloor + 1$ as the *ideal* size of this counter. Figure 8(a) shows the average bits allocated for a counter in each framework. We distribute the memory consumption of non-counter data structures evenly to each counter. For comparison, we include the averaged bits of ideal allocation and the fixed-size allocation. We observe that CounterSnake allocates the fewest bits to a counter. We further group counters by their ideal sizes and investigate the average size in each group. Figure 8(b) shows the results. The x-axis is the ideal size of a counter, while the y-axis is the average bits allocated for counters with this ideal size. We also include the ideal results and the baseline results. We can observe 'ladders' in the lines of PY, BS, DIA, and CSK because their hierarchical structures only allow four different sizes (as there are four layers). Among them, CSK remains the closest to the ideal line, demonstrating its high memory efficiency. BM and ST adjust counters in finer granularity, so their counter sizes increase steadily with the values. BM's line is far from optimal because it attaches an 8-bit fingerprint to each counter for cuckoo hashing. SAC allocates a constant number (eight) of bits to each counter, leading to errors for counters with large values.

We further break down the memory consumption into four parts, shown in Figure 9. *Counter* refers to the part that is for recording values. *Tag* refers to the part that indicates the status of each counter, such as the tags used in CounterSnake and the fingerprints used in BM. *Auxiliary* refers to the part that aims to support counter interfaces, such as the deletion part in DIA. *Empty* refers to counters that are not utilized, such as the unused counters in CounterSnake. The last three parts are considered overhead because they are not directly used to record values. We observe that DIA requires much memory to support deletion and negative values. PY and ST contain a significant number of unused counters because their rigid binary structures can not adapt to different overflow rates. We also find that SAC and CSK have the lowest overhead, so they can better utilize the available memory.

(Exp#3) Adaptivity. We test the performance of each framework on the two tasks using synthetic Zipf datasets with different skewness. In each experiment, the memory budget of each framework is also fixed at the same amount. Note that when skewness is zero, the item counts follow a uniform distribution. When the skewness is large, there are only a few counters with large values, so the overflow rate will be low. The results are illustrated in Figure 6(e)-(f). We can see that all frameworks tend to perform better with higher skewness. However, when the skewness is close to zero, many frameworks (especially those without lossless guarantees) show degraded performance, and sometimes even worse than the baseline solution that uses fixed-size counters. This is because low skewness means high overflow rates, leading to huge counter errors in these frameworks. CounterSnake is the only framework to maintain stable performance across all skewness, mainly due to its lossless compression and flexible configuration.

5.3 Requirements Fulfillment

(Exp#4) Counter errors. We evaluate the counter errors introduced by the frameworks on three sketches that cover all counter interfaces described in Section 2.1. Specifically, CM only involves adding the counters, HP involves adding, online queries, and resetting, while CS involves negative values. Figure 10 depicts the results on two datasets. CounterSnake and BS are the only two frameworks that aim for zero error. However, BS cannot achieve zero error on all sketches (e.g., HP), because its main structure does not support the online queries required in HP. BS uses an auxiliary CM sketch to assist online queries, leading to estimation errors. Other frameworks introduce counter errors in all three sketches. We also observe that PY shows high counter errors on the CS sketch that requires negative values. The reason is rooted in its binary tree structure. When one of the two siblings suffers from negative overflow and the other suffers from positive overflow, PY's no-under-estimation property fails, leading to huge errors. In summary, CounterSnake is the only one that achieves zero counter error on all types of counter interfaces, fulfilling requirements R1 and R3.

(Exp#5) Memory access. To analyze whether the frameworks achieve constant-time operations (R2), we further investigate the number of memory accesses during one insertion or query. As stated in Section 2.3, we aim for not only amortized $O(1)$ complexity but also worst-case $O(1)$ complexity. Therefore, we investigate both the average and maximum number of memory accesses per operation. We use the frequency estimation task as an example and present the results in Figure 11. The query accesses of BS are omitted because BS does not support online queries.

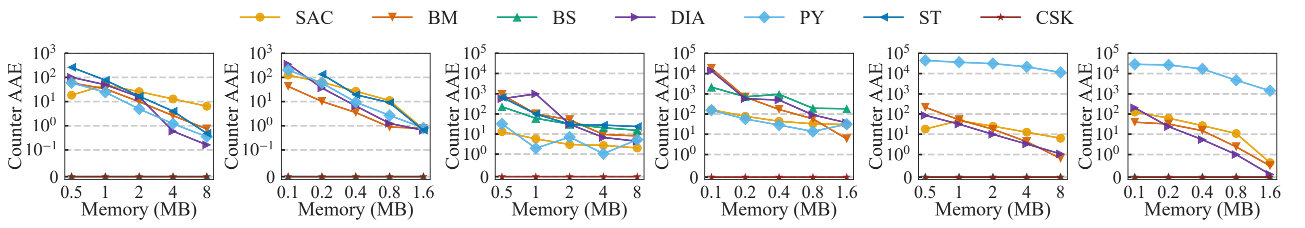
We find that the average number of accesses roughly corresponds to the throughput results in Exp#1. For example, CounterSnake involves similar insertion accesses as others, so they have similar insertion throughput. DIA involves more query accesses than others, so its query throughput is the lowest. However, the maximum number of memory accesses exhibits huge differences from the average number. We can see BS and ST involve many more insertion accesses in the worst case, while DIA and ST involve many query accesses in the worst case. This shows their worst-case complexity is high, failing to support tasks that are sensitive to processing latency. On the other hand, CounterSnake's maximum access numbers are not far from average ones, fulfilling requirement R2.

(Exp#6) Multiple sketches. We integrate CounterSnake with eight sketches to demonstrate its efficiency in dealing with multiple sketches. For comparison, we conduct two experiments. One is to compress the eight sketches as a whole, as described in Section 3.5. The other is to naively partition the available memory into per-sketch regions and compress them individually. In both experiments, we fix the same number of virtual counters for each sketch.

Figure 12 shows the percentage of used counters in layers ≥ 2. For per-sketch compression, the results are presented as the leftmost 8 columns. For compression as a whole, the result is shown as the rightmost column. We find that in per-sketch compression, the used percentage varies a lot among the eight sketches. For CM, ES, and FR, the percentages are below 25%, leading to a huge waste of counters in high layers. For DT, MV, and SL, they fully utilize high-layer counters but incur many excessive overflows, overwhelming the Backup Table. However, when we use CounterSnake to compress all sketches as a whole, the unused counters in CM, ES, and FR can be utilized by DT, MV, and SL. As a result, the used percentage remains high while the number of excessive overflows significantly reduced to only 18% in per-sketch compression. This demonstrates the resource-sharing efficiency of CounterSnake's multi-sketch compression, thus fulfilling requirement R4.

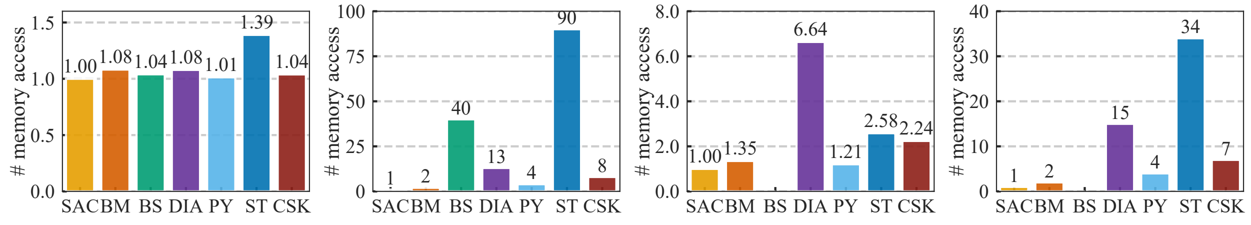
5.4 Microbenchmark

(Exp#7) The group size s_i and c. This experiment provides empirical insights on the influence of the group size s_i and the constant c on CounterSnake's memory consumption. Recall that c is introduced in Section 4.2 to control the size of the Backup Table. We set $l = 4$, $N = 1\text{M}$, $\delta = 0.01$ and estimate the overflow rates as $p_1 = 0.128$, $p_2 = 0.05$, $p_3 = 0.04$, which is gathered from **Exp#2** on CM sketch and Caida dataset. Then we try various s_i and c in $(0, 256]$ and test the average bits allocated for a counter. For simplicity, we set s_i in each layer to be the same s. The results are presented in Figure 13. We can see there are two regions that achieve the lowest average bits per counter. In practice, we choose $s = 144, c = 16$ (plotted as a red point) as the optimum values. There are two considerations for this choice. First, we want a small s_i (thus d_i is also small) to reduce the time complexity when searching for a matched tag. Second, the size of the Backup Table is proportional to s_i and c. Smaller s and c can reduce the frequency of dealing with time-consuming excessive overflows. We also vary p_i between $(0, 0.2)$ and find $s = 144, c = 16$ achieve good results overall.



(a) CM Error (Caida) (b) CM Error (Kosarak) (c) HP Error (Caida) (d) HP Error (Kosarak) (e) CS Error (Caida) (f) CS Error (Kosarak)

Figure 10: Counter AAE of different compression frameworks



(a) Avg. Insert Access (b) Max. Insert Access (c) Avg. Query Access (d) Max. Query Access

Figure 11: The average and maximum number of memory accesses during one insertion or query

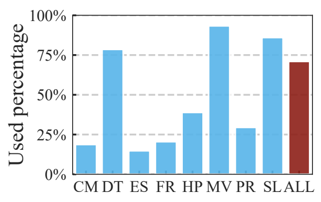


Figure 12: The usage ratio of high-layer counters in multi-sketch compression

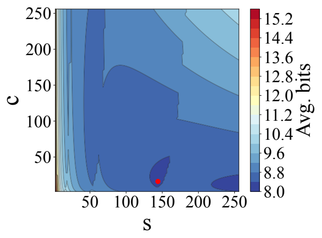


Figure 13: The influence of s and c on average counter bits

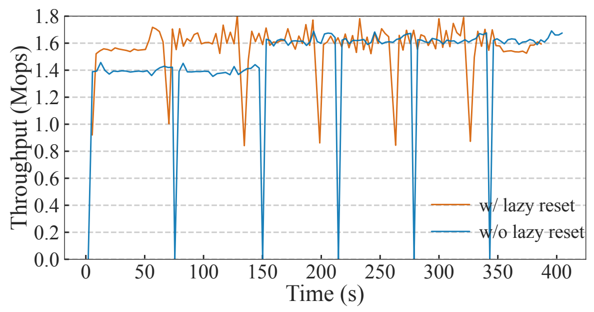


Figure 14: The effectiveness of the lazy reset technique

(Exp#8) The lazy reset technique. This experiment evaluates the effectiveness of the lazy reset technique introduced in Section 3.5. We use an online heavy-changer detection task as an example, which requires two sketches to alternately maintain the frequencies in the current and previous windows. We integrate CounterSnake with two CM sketches to perform this task on the Caida dataset. We set the window length as one minute, and the chunk size as 16 counters for the lazy reset technique. Figure 14 shows the insertion throughput with and without the lazy reset technique during a 6-minute data stream. Without the lazy reset technique, there is a two-second pause at the beginning of each window to reset the outdated sketch, which blocks the processing of the data stream. On the other hand, the lazy reset technique amortizes this cost to the first access of an outdated chunk. So the throughput is only partially dropped at the beginning of each window because of the resetting overhead, and the processing is not paused.

6 CONCLUSION

This paper addresses the memory inefficiency of fixed-size counters in sketches by proposing CounterSnake, a novel hierarchical compression framework. Through extensive theoretical analysis and experiments, we demonstrate that CounterSnake is the first to fulfill four vital requirements. However, like all previous solutions, CounterSnake still relies on some prior knowledge about the data stream distribution. Thus, we propose an open problem of how to design a compression framework that can ensure consistent good performance for arbitrary data stream distributions. In the future, we plan to improve CounterSnake by enabling dynamic assignment of the counter resources in each layer. This improvement adapts CounterSnake to different overflow rates, thus handling dynamic issues in data streams such as concept drift. In conclusion, CounterSnake bridges a critical gap between theoretical compression limits and practical demands for sketch-based stream processing.

ACKNOWLEDGMENTS

This work was supported by National Key Research and Development Program of China (2023YFB2904600), Joint Funds of the National Natural Science Foundation of China (U25B2038), and Research Grants Council of Hong Kong (GRF 14201523). We also thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] [n.d.]. The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019). https://www.caida.org/catalog/datasets/passive_dataset/. Accessed: 2025-06-15.

[2] [n.d.]. Frequent Itemset Mining Dataset Repository. http://fimi.uantwerpen.be/data/. Accessed: 2025-06-15.

[3] [n.d.]. Redis - The Real-time Data Platform. https://redis.io/. Accessed: 2025-09-10.

[4] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: democratically finding the cause of packet drops. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 419–435.

[5] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. SALSA: Self-Adjusting Lean Streaming Analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 864–875. https://doi.org/10.1109/ICDE51399.2021.00080

[6] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[7] A.Z. Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. 21–29. https://doi.org/10.1109/SEQUEN.1997.666900

[8] Lu Cao, Qilong Shi, Yuxi Liu, Hanyue Zheng, Yao Xin, Wenjun Li, Tong Yang, Yangyang Wang, Yang Xu, Weizhe Zhang, and Mingwei Xu. 2024. Bubble Sketch: A High-performance and Memory-efficient Sketch for Finding Top-k Items in Data Streams. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) *(CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 3653–3657. https://doi.org/10.1145/3627673.3679882

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15. https://doi.org/10.1016/S0304-3975(03)00400-6 Automata, Languages and Programming.

[10] Qizhi Chen, Yisen Hong, Yuhan Wu, Tong Yang, and Bin Cui. 2024. CodingSketch: A Hierarchical Sketch with Efficient Encoding and Recursive Decoding. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 1592–1605. https://doi.org/10.1109/ICDE60146.2024.00130

[11] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55, 1 (apr 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[12] G. Cormode and S. Muthukrishnan. 2005. What's new: finding significant differences in network data streams. *IEEE/ACM Transactions on Networking* 13, 6 (2005), 1219–1232. https://doi.org/10.1109/TNET.2005.860096

[13] Rui Ding, Shibo Yang, Xiang Chen, and Qun Huang. 2023. BitSense: Universal and Nearly Zero-Error Optimization for Sketch Counters with Compressive Sensing. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 220–238. https://doi.org/10.1145/3603269.3604865

[14] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07), Article 10 (Jan 2007). https://doi.org/10.46298/dmtcs.3545

[15] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Comput. Secur.* 28, 1–2 (Feb. 2009), 18–28. https://doi.org/10.1016/j.cose.2008.08.003

[16] Junzhi Gong, Tong Yang, Yang Zhou, Dongsheng Yang, Shigang Chen, Bin Cui, and Xiaoming Li. 2017. ABC: A practicable sketch framework for non-uniform multisets. In *2017 IEEE International Conference on Big Data (Big Data)*. 2380–2389. https://doi.org/10.1109/BigData.2017.8258193

[17] Şule Gündüz and M. Tamer Özsu. 2003. A Web page prediction model based on click-stream tree representation of user behavior. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, D.C.) *(KDD '03)*. Association for Computing Machinery, New York, NY, USA, 535–540. https://doi.org/10.1145/956750.956815

[18] Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Yu Cheng, and Hao Wu. 2014. Discount counting for fast flow statistics on flow size and flow volume. *IEEE/ACM Trans. Netw.* 22, 3 (jun 2014), 970–981. https://doi.org/10.1109/TNET.2013.2270439

[19] Qun Huang and Patrick P. C. Lee. 2014. LD-Sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 1420–1428. https://doi.org/10.1109/INFOCOM.2014.6848076

[20] Qun Huang, Patrick P. C. Lee, and Yungang Bao. 2018. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 576–590. https://doi.org/10.1145/3230543.3230559

[21] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. 2021. Toward Nearly-Zero-Error Sketching via Compressive Sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 1027–1044. https://www.usenix.org/conference/nsdi21/presentation/huang

[22] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. 2018. SketchML: Accelerating Distributed Machine Learning with Data Sketches. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1269–1284. https://doi.org/10.1145/3183713.3196894

[23] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. 2022. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proc. VLDB Endow.* 15, 7 (March 2022), 1426–1438. https://doi.org/10.14778/3523210.3523220

[24] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1574–1584. https://doi.org/10.1145/3394486.3403208

[25] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. 2006. Detection and identification of network anomalies using sketch subspaces. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement* (Rio de Janeriro, Brazil) *(IMC '06)*. Association for Computing Machinery, New York, NY, USA, 147–152. https://doi.org/10.1145/1177080.1177099

[26] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: a better NetFlow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (, Santa Clara, CA,) *(NSDI'16)*. USENIX Association, USA, 311–324.

[27] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[28] Zirui Liu, Yixin Zhang, Yifan Zhu, Ruwen Zhang, Tong Yang, Kun Xie, Sha Wang, Tao Li, and Bin Cui. 2023. TreeSensing: Linearly Compressing Sketches with Flexibility. *Proc. ACM Manag. Data* 1, 1, Article 56 (May 2023), 28 pages. https://doi.org/10.1145/3588910

[29] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter braids: a novel counter architecture for per-flow measurement. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Annapolis, MD, USA) *(SIGMETRICS '08)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/1375457.1375472

[30] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. 2022. Enabling efficient and general subpopulation analytics in multidimensional data streams. *Proc. VLDB Endow.* 15, 11 (July 2022), 3249–3262. https://doi.org/10.14778/3551793.3551867

[31] Sergiy Matusevych, Alexander J. Smola, and Amr Ahmed. 2012. Hokusai — sketching streams in real time. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence* (Catalina Island, CA) *(UAI'12)*. AUAI Press, Arlington, Virginia, USA, 594–603.

[32] Gabriel Mersy, Zhuo Wang, Stavros Sintos, and Sanjay Krishnan. 2024. Optimizing Collections of Bloom Filters within a Space Budget. *Proc. VLDB Endow.* 17, 11 (July 2024), 3551–3564. https://doi.org/10.14778/3681954.3682020

[33] Ruijie Miao, Zheng Zhong, Jiarui Guo, Zikun Li, Tong Yang, and Bin Cui. 2023. BurstSketch: Finding Bursts in Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11126–11140. https://doi.org/10.1109/TKDE.2022.3223686

[34] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2023. Sketchovsky: Enabling Ensembles of Sketches on Programmable Switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1273–1292. https://www.usenix.org/conference/nsdi23/presentation/namkung

[35] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (May 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[36] Wieger R. Punter, Odysseas Papapetrou, and Minos Garofalakis. 2023. OmniSketch: Efficient Multi-Dimensional High-Velocity Stream Analytics with Arbitrary Predicates. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 319–331. https://doi.org/10.14778/3632093.3632098

[37] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. 2023. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1237–1255. https://www.usenix.org/conference/nsdi23/presentation/scazzariello

[38] Siyuan Sheng, Qun Huang, Sa Wang, and Yungang Bao. 2021. PR-sketch: monitoring per-key aggregation of streaming data with nearly full accuracy. *Proc. VLDB Endow.* 14, 10 (jun 2021), 1783–1796. https://doi.org/10.14778/3467861.3467868

[39] Qilong Shi, Chengjun Jia, Wenjun Li, Zaoxing Liu, Tong Yang, Jianan Ji, Gaogang Xie, Weizhe Zhang, and Minlan Yu. 2024. BitMatcher: Bit-level Counter Adjustment for Sketches. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4815–4827. https://doi.org/10.1109/ICDE60146.2024.00366

[40] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. 164–176.

[41] R. Stanojevic. 2007. Small Active Counters. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. 2153–2161. https://doi.org/10.1109/INFCOM.2007.249

[42] Haifeng Sun, Qun Huang, Jinbo Sun, Wei Wang, Jiaheng Li, Fuliang Li, Yungang Bao, Xin Yao, and Gong Zhang. 2024. AutoSketch: Automatic Sketch-Oriented Compiler for Query-driven Network Telemetry. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1551–1572. https://www.usenix.org/conference/nsdi24/presentation/sun

[43] Lu Tang, Qun Huang, and Patrick P. C. Lee. 2020. A Fast and Compact Invertible Sketch for Network-Wide Heavy Flow Detection. *IEEE/ACM Transactions on Networking* 28, 5 (2020), 2350–2363. https://doi.org/10.1109/TNET.2020.3011798

[44] Pinghui Wang, Peng Jia, Xiangliang Zhang, Jing Tao, Xiaohong Guan, and Don Towsley. 2019. Utilizing Dynamic Properties of Sharing Bits and Registers to Estimate User Cardinalities Over Time. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1094–1105. https://doi.org/10.1109/ICDE.2019.00101

[45] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. 1990. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.* 15, 2 (June 1990), 208–229. https://doi.org/10.1145/78922.78925

[46] Yuhan Wu, Shiqi Jiang, Siyuan Dong, Zheng Zhong, Jiale Chen, Yutong Hu, Tong Yang, Steve Uhlig, and Bin Cui. 2023. MicroscopeSketch: Accurate Sliding Estimation Using Adaptive Zooming. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 2660–2671. https://doi.org/10.1145/3580305.3599432

[47] Tong Yang, Siang Gao, Zhouyi Sun, Yufei Wang, Yulong Shen, and Xiaoming Li. 2019. Diamond Sketch: Accurate Per-flow Measurement for Big Streaming Data. *IEEE Transactions on Parallel and Distributed Systems* 30, 12 (2019), 2650–2662. https://doi.org/10.1109/TPDS.2019.2923772

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

[49] Tong Yang, Jiaqi Xu, Xilai Liu, Peng Liu, Lun Wang, Jun Bi, and Xiaoming Li. 2019. A Generic Technique for Sketches to Adapt to Different Counting Ranges. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2017–2025. https://doi.org/10.1109/INFOCOM.2019.8737531

[50] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858. https://doi.org/10.1109/TNET.2019.2933868

[51] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1442–1453. https://doi.org/10.14778/3137628.3137652

[52] M. Yoon, T. Li, S. Chen, and J.-K. Peir. 2009. Fit a Spread Estimator in Small Memory. In *IEEE INFOCOM 2009*. 504–512. https://doi.org/10.1109/INFCOM.2009.5061956

[53] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software defined traffic measurement with OpenSketch. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(nsdi'13)*. USENIX Association, USA, 29–42.

[54] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. DHS: Adaptive Memory Layout Organization of Sketch Slots for Fast and Accurate Data Stream Processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 2285–2293. https://doi.org/10.1145/3447548.3467353

[55] Yikai Zhao, Wenchen Han, Zheng Zhong, Yinda Zhang, Tong Yang, and Bin Cui. 2023. Double-Anonymous Sketch: Achieving Top-K-fairness for Finding Global Top-K Frequent Items. *Proc. ACM Manag. Data* 1, 1, Article 79 (May 2023), 26 pages. https://doi.org/10.1145/3588933

[56] Yikai Zhao, Zheng Zhong, Yuanpeng Li, Yi Zhou, Yifan Zhu, Li Chen, Yi Wang, and Tong Yang. 2021. Cluster-Reduce: Compressing Sketches for Distributed Data Streams. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 2316–2326. https://doi.org/10.1145/3447548.3467217

[57] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold Filter: A Meta-Framework for Faster and More Accurate Stream Processing. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 741–756. https://doi.org/10.1145/3183713.3183726