Geometric Sketch: The Inflatable-Shrinkable Sketch

Anonymous Author

***Abstract*—Stream frequency measurements are fundamental in many data stream applications such as financial data trackers, intrusion detection systems, and network monitoring. Count-Min Sketch and its variants have been widely adopted for this task due to their space efficiency and ability to provide approximate frequency estimates with bounded error guarantees. However, these sketches suffer from a limitation: they have a fixed memory usage determined by the desired accuracy, unable to adapt to changing memory availability or accuracy requirements during runtime. Some dynamic solutions exist, but they are constrained.**

In this paper, we introduce the *Geometric Sketch* (GS), a frequency estimation sketch that addresses the limitations of existing sketches by offering dynamic memory allocation. Unlike existing sketches with fixed memory footprints, the Geometric Sketch can dynamically grow or shrink its memory usage at an arbitrary size, down to a single-counter granularity, making it well-suited to systems with changing memory availability. When evaluated using real Internet packet traces, the Geometric Sketch achieves substantially higher accuracy compared to the state-of-the-art method, DCMS, by employing fine-grained expansion and compression. All our code is open sourced [2].

***Index Terms*—Sketches, Adaptivity, Stream Processing**

I. Introduction

Tracking items' frequency in data streams is a fundamental task in many applications, such as load balancing [9], caching [25], intrusion detection [18] and anomaly detection [13]. Such applications require time- and space-efficient algorithms to cope with high-speed data streams. To that end, data sketch algorithms are employed to analyze the data in a single pass and provide estimated statistical measures while using limited resources. These algorithms build a sketch, a memory-efficient data structure designed to capture statistical characteristics of data streams in one pass over the data stream.

Sketches trade off accuracy for memory efficiency, which is critical in stream processing applications. While some scenarios may prioritize high accuracy and allocate more memory to the sketch, others may operate under tight memory constraints, necessitating a compromise on accuracy. Various sketch algorithms have been proposed, each offering different accuracy-memory profiles tailored to specific use cases. Count-Min Sketch (CMS) [7] and Count Sketch [5] are popular examples used to estimate items' frequencies.

A common shortcoming of traditional frequency estimation sketches is that their memory allocation is statically defined during initialization time based on the desired accuracy parameters, requiring all decisions to be made a priori. Even if extra memory is available in the system [20], which may occur in software implementations of sketches found in SDNs and virtual switches, this memory cannot be used to improve the precision of the sketch. The Dynamic Count-Min Sketch (DCMS) [26] has been recently proposed to address this shortcoming. However, the granularity at which DCMS grows is coarse. Furthermore, it cannot free memory if the added memory is needed back for other purposes.

As another use case, in most sketches, empirically measured errors are usually significantly lower than the theoretical bounds. Hence, having the ability to dynamically resize the sketch during runtime enables to optimize the memory allocation based on the actual error growth and the possibly changing accuracy requirements of its applications.

In this paper, we introduce the *Geometric Sketch* (GS), a frequency estimation sketch that addresses the limitations of existing sketches by providing dynamic memory allocation. GS's accuracy remains proportional to its current memory allocation, offering similar accuracy to the widely-used CMS under the same memory budget. However, unlike traditional sketches with static memory footprints, GS's accuracy can be improved on the fly by allocating additional memory as it becomes available, without sacrificing existing accuracy guarantees. In contrary to DCMS, GS can grow and shrink at fine granularity, as small as a single counter, enabling superior control over its memory consumption and accuracy. This fine-grained elasticity allows GS to dynamically allocate more memory when available and to release memory back to the system when needed. That is, GS can seamlessly adapt to periods of varying memory availability, dynamically expanding when memory is available and contracting when memory is scarce to optimize resource utilization.

GS extends CMS by introducing an elastic memory allocation scheme. Instead of having a fixed number of counters, as is the case in CMS, GS can dynamically extend its width by adding "child counters" to existing counters, forming a logical tree structure. This enables allocating additional counters as more memory becomes available, which reduces collision rates and improves accuracy. Likewise, GS can free up memory by removing counters, either by undoing the allocation of child counters or compressing the sketch by removing old counters. We stress that GS does *not* employ pointers.

GS offers a similar error guarantee to that of CMS when utilizing the same amount of memory. When more memory becomes available, GS can improve its error bound. We show that when GS is expanded by a single counter in regular intervals, the probability of an estimation error larger than $M \log_B(N)$ is lower than δ, where B, M, and δ are constants that can be freely selected, while N is the total number of elements in the stream.

We evaluated the accuracy and throughput of GS in a diverse range of scenarios, using real- and synthetic-data traces. We compare its performance against CMS and DCMS. Our results indicate that GS can achieve better accuracy than DCMS when

operating within the same memory constraints. Additionally, GS can achieve substantially higher accuracy by utilizing its finer flexibility and support for on-the-fly lossless compression. Furthermore, GS supports undoing memory allocations, resulting in the state the sketch would be in had the expansion did not occur, yet updates did.

II. Related Work

Loosely speaking, we may categorize most frequency estimation data structures into *sketch based*, *counter based* or *hybrids* of both. We briefly survey them below:

A. Counter Based

Counter-based solutions explicitly store a mapping between keys and frequency counters, but typically only for a limited subset of keys. The counters might be modified to account for non-tracked flows. Many such implementations focus on identifying heavy hitters during runtime for explicit storage, while pruning mice flows, sometimes attempting to estimate them utilizing statistical data. For example, Lossy Counting [16] deletes infrequent keys. Frequent [8] decrements all counters whenever an untracked key is updated, until a counter reaches 0, at which point it is deleted. Space savings [17] replaces the least frequent key with an untracked key when updated.

RAP [3] improves space saving by adding a probabilistic admission filter. In this way, mice items are less likely to obtain a counter, which significantly improves overall accuracy.

Effective space savings [10] expand on this strategy by storing memory to gather statistical information on untracked flows, generating a better estimate of these keys.

B. Sketch Based: Classic

Count Sketch (CS) [5] and Count-Min Sketch (CMS) [7] are the most well-known sketches for frequency estimation. These sketches utilize a two-dimensional array of counters and pairwise independent hash functions, one for each row of the array. Keys are mapped to a counter in each row. To update a key's occurrence count, its counters are updated, with the exact details differing by the specific sketch type. CMS may only overestimate a counter and can support deletions (negative updates), while CS can also underestimate counters and does not support deletions. CMS's accuracy can be improved by utilizing conservative updates [4] (CUS), although this modification eliminates the ability to delete.

C. Sketch Based: Flexible

Elastic Sketch [23] includes a light part (classic sketch with small-sized counters) for recording mouse flows and a heavy part (a bucketized hash table) for more accurately recording elephant flows. Flows can be swapped between the heavy part and the light part during runtime. In case a flow turns out to have a larger number of heavy hitters than initially expected, the heavy part can grow by doubling its size. This solution is therefore a hybrid of the counter and sketch-based approaches.

Waving sketch [12] also implements a similar separation of flows into different parts, and many other sketches utilize a similar concept of flow separation. The heavy part can usually be expanded or shrunk easily, but the light part, a classic sketch, is more limited.

Pyramid Sketch [24] allocates small counters, a few bits wide, where an overflow expands into additional memory. A layered memory-sharing technique is used, utilizing the fact that only some counters overflow. This is useful since counters rarely utilize the 32 bits commonly allocated to them. This "counter-sharing" method is also present in Counter Braids [15] and FCM-Sketch [21].

More complex sketches utilize similar concepts of flow separation, expanding counters, and multiple sketches to great effect, often taking into account cache size, memory accesses, distributed systems, and the like. These include, e.g., Stingy Sketch [11], HeavyGuardian [22], and Tree sketch [14].

Dynamic Count-Min Sketch (DCMS) [26] uses a stack of sketches, only updating the top sketch. When the top sketch's error or cardinality (number of flows) grows beyond an allowed limit, an additional, larger sketch is appended. Multiple allocation strategies are presented for different accuracy requirements. DCMS is the most relevant to us since it tackles a similar problem. Unlike GS, DCMS cannot release back memory into the system. Also, its minimal expansion size is an entire CMS which grows bigger with each allocation, whereas GS can increase and decrease its memory utilization at fine granularity, by as little as a single counter at a time.

III. preliminaries

A. Count-Min Sketch

A Count-Min sketch (CMS) [7] data structure supports two methods: $\texttt{update}(k, v)$, which increments the value associated with key k by v, and $\texttt{query}(k)$, which returns an estimate for the sum of values added to key k. When used for frequency estimation, the update method is invoked with a value 1 on each occurrence of the key k in the stream, and then the query returns an estimate of k's frequency.

CMS consists of d arrays of w counters each, denoted CMS$[d, w]$, and d pairwise independent hash functions $\{h_1, \ldots, h_d\}$. In order to increment the frequency of an item x, all counters CMS$[i, h_i(x)]$ are incremented by v, for all $i \in [1, \ldots, d]$. In order to estimate the frequency of x, CMS returns $\min_{i\in[1,\ldots,d]}$ CMS$[i, h_i(x)]$. The use of pairwise independent hash functions d reduces the chances that two keys would repeatedly collide on multiple rows, thus reducing the estimation error. Denote the true frequency of x after processing N items by $f(x)$. Assigning $w = e\epsilon^{-1}$ and $d = \log(\delta^{-1})$ ensures that the frequency estimate $\hat{f}(x)$ obeys $0 \leq \hat{f}(x) - f(x) \leq \epsilon N$ with probability $1 - \delta$.

Note that the memory requirement of CMS is independent of the number of items that are inserted N (discounting the $O(\log N)$ bits per counter). However, the absolute error value does grow linearly with N. Also, CMS does not save any identifiers, which reduces its memory overhead, especially when identifiers are large.

B. Dynamic Count-Min Sketch

As was mentioned in the Introduction, Dynamic Count-Min Sketch (DCMS) is composed out of a stack of CMSs, such that only the top, widest CMS is updated [26]. On a query, all of the CMSs are queried and the sum is returned. The larger sketches are assigned according to an "expansion strategy".

Expansion strategies can be classified into two categories: *Linear* and *Exponential*, accordingly titled Strategy 2 and Strategy 3 in the DCMS paper. Both are defined by the constants K, N as follows:

Linear Strategy: $\varepsilon_j = \frac{\varepsilon'}{kj}$, $|a^j| = n * k * j$

Exponential Strategy: $\varepsilon_j = \frac{\varepsilon'}{k^j}$, $|a^j| = n * k^j$

where $|a^j|$ is the threshold of the j^{th} sketch. We may increment the sketch counters by that amount, after which an additional update would cause a larger sketch to be allocated, with its width determined by ε_j.

Since $W = \frac{e}{\varepsilon}$, the sketch widths series is $\{Wkj | j \in \mathbb{N}\}$ for the Linear Strategy and $\{Wk^j | j \in \mathbb{N}\}$ for the Exponential Strategy. For the linear case, for every N updates we allocate an additional CMS sized $k * i$, with i being the number of CMS in the DCMS, i.e., the sketches grow linearly. For the exponential case, we similarly allocate a k^i sized sketch following k^i updates. When we select $k = 1$, the linear strategy yields the same behavior as strategy 1 from the DCMS paper [26]. Hence, in this paper, we do not define a special distinction for Strategy 1.

Note that for both expansion strategies, the number of allocated counters follows a line passing through $(0, 0)$ with a slope of $\frac{WD}{N}$, with each counter requiring 4 bytes. The memory usage does not align exactly with the line, since additional memory is required for the administration of DCMS.

GS provides more flexibility than DCMS, allowing allocations and deallocations in arbitrary sizes. Therefore, we define the fine expansion strategy, with the parameters m, r - every m updates, r counters are added to GS. To match DCMS's strategies, the following equation must hold: $m * r = \frac{N}{WD}$.

IV. Geometric Sketch (GS)

A. Overview

GS operates similarly to CMS in that it updates and queries item frequencies. On update, the count of an item is modified, while on query an estimation of the item's count is returned.

GS improves upon CMS by utilizing trees of counters, each sprouting from an original CMS-like counter. The parent-child relationships between counters are stored implicitly, by the counters' ordering, thereby not wasting a single bit of memory.

This allows GS to introduce three additional operations: *Expand*, *UndoExpand*, and *Compress*. These operations enable dynamic adjustment of memory usage at arbitrary sizes, with a granularity of one counter. Users can invoke these operations at any time to suit specific use cases and memory constraints.

B. GS Internals

GS's fixed parameters include its width (W) and depth (D), similar to CMS. Additionally, it also has a branching factor (B), an integer determining the children per counter.

The GS in Figure 1 has the parameters $D = 4$ since it has 4 rows, $W = 2$ since layer 0 has 2 counters, and $B = 2$ since every counter can have 2 children.

GS also maintains some internal variables, which include an array of counters (Counters), similar to CMS, and also a compressed counters counter (O), incremented for every counter removed by compression.

Following the compression in Figure 1f, the value of O is increased by the number of counters compressed in the illustrated row, as well as the other rows.

Notably, B and O are the only additional fields present in GS's class while not being in CMS. Hence, the memory overhead of GS compared to CMS is just 2 additional integers.

1) Expand(v): Based on the parameter v, which indicates how many additional counters need to be allocated, *expand* increases the memory allocated to GS, where v, can range from a single counter to as many as the operating system can provide. This flexibility allows GS to adapt to varying accuracy requirements by dynamically expanding its memory footprint. Consider a scenario in which the accuracy requirement increases due to higher traffic volumes. By calling $Expand(100)$, for example, GS allocates 100 additional counters, thus reducing the build-up of errors.

2) UndoExpand(v): The *undoExpand* operation decreases memory usage by removing the v newest counters, that were previously added through expansion. This operation reverts GS to a state as if the undone counters were never added, yet updates were still applied. The value of n can be chosen arbitrarily, allowing partial or complete reversal of one or multiple expand operations.

After applying $Expand(100)$ on a GS, you may call $UndoExpand(40)$ to remove 40 counters, reducing memory usage. Following the undo, the state of GS is identical to its state had it been expanded by $Expand(60)$ and received the same update operations. This means we may expand to improve queries and undo the expansion later without paying a penalty. This is well suited when memory availability changes significantly during run-time, allowing us to utilize available memory and release it when required, all in arbitrary sizes.

3) Compress(v): The *compress* operation also reduces memory usage but does so without affecting the current accuracy of GS. It achieves this by removing the v oldest counters, which in turn limits the potential for future UndoExpand operations. *Compress* is ideal for scenarios that require memory reduction while maintaining the current level of accuracy and forgoing some of the flexibility of UndoExpand.

Consider a GS that has 60 additional counters and a branching factor IV-B of 2. When memory resources become constrained, yet we wish to retain our accuracy, we can use $Compress(10)$ to free up 10 counters. Following compression, we would only be able to undo $60 - 10 * 2 = 40$ counters (explained later in the paper). This means that we would be able to remove at most 50 counters in total instead of 60.

4) UndoExpand vs. Compress: As mentioned before, UndoExpand can free all previously expanded counters whose parents were not compressed, and thereby potentially return

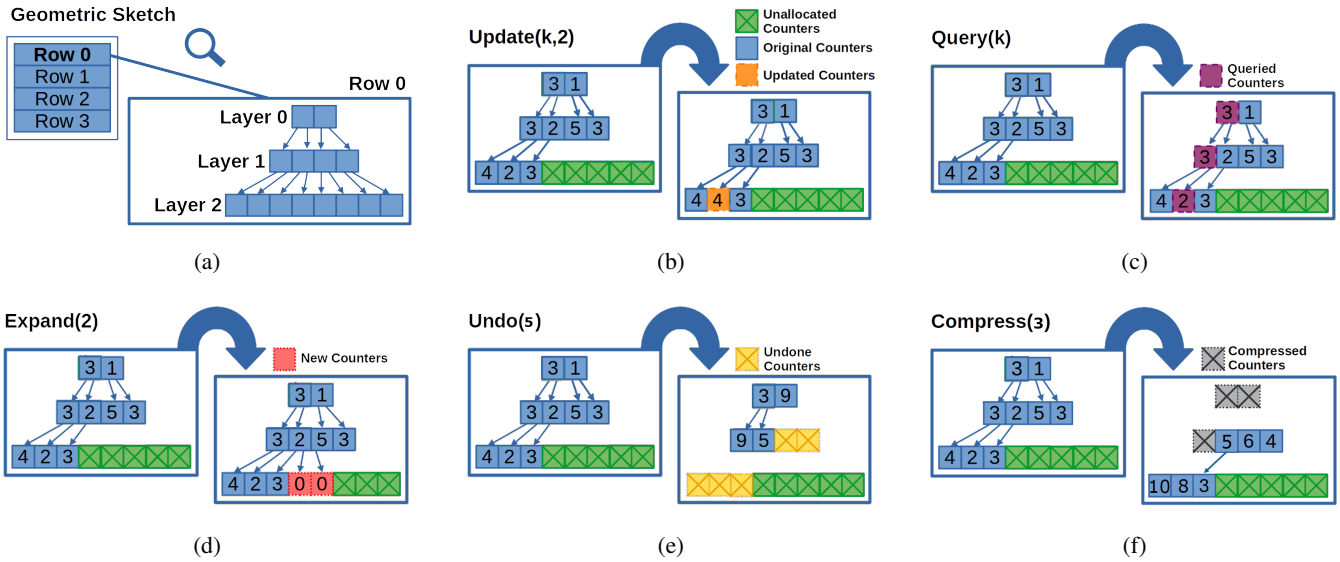

Fig. 1: (a) GS's simplified logical layout - note that we focus on a single row out of multiple, (b) GS update example: k is initially mapped to $C[0,0,0]$. layer 1's hash selects the left child $C[0,1,0]$ and finally layer 2's hash selects the right child $C[0,2,1]$, (c) GS query example: the sum $C[0,0,0]+C[0,1,0]+C[0,2,1]$ is row 0's estimation for k's counter, (d) GS expand example: expansion is done left to right, top down and adds zeroed counters, (e) GS undo example: undoing is done right to left, bottom up, (f) GS compress example: compression is done left to right, top down. The value of compressed counters is 0.

all added memory back to the system. However, UndoExpand reduces the accuracy, On the other hand, compress can free counters that are currently not being actively used, since they were expanded, without hurting the accuracy of the system. Yet, compressed counters prevent GS from returning their children counters' memory. This presents a flexibility vs. accuracy trade-off in how these two functions are being used.

Specifically, when the memory available for GS is known to be monotonically increasing, the user should compress GS immediately after each expansion to conserve memory. This ensures the best tradeoff between memory and accuracy. However, if GS may be required in the future to relinquish more memory than the amount that can be freed by compression, then some counters should not be compressed to allow UndoExpand to return their children's memory. Mark C the number of counters that GS can free with UndoExpand. Compressing one counter decreases C by B. Consequently, the user should aim to match C with the maximum amount of memory that could be claimed back from GS.

5) Expansion Strategies: When memory can be become available on demand, a common use case is to follow an expansion strategy, similar to DCMS, as already presented in Section III. With such a strategy, every set number of updates, GS is expanded by an amount dictated by a function. Immediately following the expansion, GS can be compressed as much as possible to save space while entirely retaining the benefit from the added counters. Using specific expansion strategies allows the user to bound the error by predefined conditions, as we explore in Section IV-F.

6) Expansion Strategies: Although users can freely utilize these functions, a common use case is following an expansion strategy, similar to DCMS, as already presented in Section III. Every set amount of updates, GS is expanded by an amount dictated by a function. Immediately following the expansion, GS can be compressed as much as possible to save space while entirely retaining the benefit from the added counters. Using specific expansion strategies allow the user to bound the error by predefined conditions, as we explore in Section IV-F. We expect this strategy to be employed since it is simple and is already employed in other sketches like DCMS. It also allows a comparison to existing sketches, although it does not utilize the full capabilities of GS.

C. Ordering of Counters

To enable efficient size modification and reduce memory fragmentation, GS stores all of its counters in a single array of integers named Counters. The role of each counter is determined by its index in Counters and O, the number of compressed counters. The counters are ordered such that expanding is implemented by appending zeros to the end of Counters. Decreasing memory usage is achieved by removing elements from either the beginning (compression) or the end (undoExpand) of the vector. The undoExpand and compress operations require updating the children or the parent of each removed counter accordingly.

The order is based on two abstractions: rows and layers. We denote $C[i]$ as the i^{th} counter in Counters. The Counters vector is logically partitioned into D rows, similarly to CMS. We denote the i^{th} counter in the r^{th} row as $C[r,i]$.

Layers are similar to rows, being themselves a partition of each row. Unlike CMS, the length of GS rows is not constant, with the l^{th} layer containing $W \cdot B^l$ counters. Layers are stored

in the order of their index, with the layers' counters in order. A logical schematic of the relationship between rows and layers can be seen in Figure 1a. On initialization, each row has a length of W, containing only layer 0.

An item is mapped to a single counter in each layer of each row. We denote the i^{th} counter in the l^{th} layer of the r^{th} row as $C[r, l, i]$. Each counter $C[r, l, i]$ has B child counters associated with it: $C[r, l+1, j] | i \cdot B \leq j < (i+1)B$.

Given the r, l and i values of a counter, we can calculate its children and parent. Logically, counters are ordered left to right, top to bottom; interlaced by row:

$$C[0,0,0], ..., C[D-1,0,0], C[0,0,1], ..., C[D-1,0,W-1],$$
$$C[0,1,0], ..., C[D-1,1,B \cdot W-1], C[0,2,0], \cdots$$

D. GS Operations and Time Complexity Analysis

We list the various operations supported by GS, along with their respective time complexities, assuming that the number of allocated counters is T.

1) Update: Update increments D counters. We update the lowest counter mapped to the item in each row, for which we need to calculate the hashes of each layer down to the lowest allocated one. There are $\log_B(\frac{(T+O)(B-1)}{BW}+1)$ layers and even non-allocated layers require hash calculations, like in Figure 1f. $O(D \log(T+O))$.

2) Query: Similarly to the update operation, for each of the D rows we calculate the hashes down to the lowest layer. However, during query we also read from every allocated layer along the way, noting that some of the relevant counters might be compressed, in which case their value is 0 and a memory access is avoided. $O(D \log(T+O))$.

3) Expand: In the expand operation, v zeroed counters are added to the end of the Counters array. While appending to an array may require reallocating the array, optimizations often avoid this costly operation. $O(v)$ or $O(T+v)$ on reallocation.

4) UndoExpand: This operation removes at most v counters from the end of the Counters array and adds their value to their parent counters. Only counters with an allocated (non compressed) parent can be undone. To prevent memory fragmentation, arrays might be moved after resizing, in which case we need to reallocate the entire Counters array. $O(v)$ or $O(T+v)$ on reallocation.

5) Compress: The compress operation removes at most v counters from the beginning of the Counters array and adds their value to vB child counters. Only counters with all children allocated can be compressed. Similarly to undoExpand operation, we might need to reallocate Counters. $O(vB)$ or $O(T+vB)$ on reallocation.

The pseudo-code of GS appears in Algorithm 1.

Algorithm 1 GS Procedures

```
procedure UPDATE(item, count)
    for iRow = 0, ..., D − 1 do
        iCounter ← getLowestCounter(iRow, item)
        Counters[iCounter] += count
procedure QUERY(item)
    firstLayer ← firstAllocatedLayerIndex()
    lastLayer ← lastAllocatedLayerIndex()
    estimate ← ∞
    for iRow = 0, ..., D − 1 do
        rowEstimate ← 0
        for iLayer = firstLayer, ..., lastLayer do
            iCounter ← getCounter(iRow, iLayer, item)
            if iCounter != −1 then
                rowEstimate += Counters[iCounter]
        estimate ← min(estimate, rowEstimate)
    return estimate
procedure UNDOEXPAND(v)
    numUndone ← 0
    cSize ← Counters.size()
    for iChild = cSize − 1, ..., cSize − 1 − v do
        iParent = parentIndex(iChild)
        if iParent == −1 then
            break
        Counters[iParent] += Counters[iChild]
        Counters.popBack()
        numUndone += 1
    return numUndone
procedure EXPAND(v)
    Counters.append(0, v)
procedure COMPRESS(v)
    numCompressed ← 0
    for iParent = 0, ..., v − 1 do
        iChildren = childrenIndice(iParent)
        if iChildren.size() < B then
            break
        for iChild in iChildren do
            Counters[iChild] += Counters[iParent]
        Counters.popFront()
        numCompressed += 1
    return numCompressed
```

E. Hash Functions

Each layer, in each row, is associated with a different hash function. We mark the hash function for $L_{r,l}$ with $H_{r,l}$. A straightforward method to implement these different hash functions is by using a hash function with a seed argument. The seed can be a 64-bit unsigned integer with the higher 32 bits being the layer index and the lower 32 bits being the row index. Note the range difference between the hash functions:

$$\forall H_{r,0} : \mathbb{N} \to [0, W-1], \qquad \forall H_{r,l\neq 0} : \mathbb{N} \to [0, B]$$

That is, layer 0's hash function maps an item to a CMS-like counter used as a root of our counter tree, while the hash functions of the lower layers map the item to a child counter in each layer. For convenience, we also define:

$H^*_{r,l} : \mathbb{N} \to [0, W * B^l]$. This composed hash function maps items to their counters in the l^{th} layer of the r^{th} row and can be derived from $\{H_{r,l'} | 0 \leq l' \leq l\}$.

F. GS Analysis

For lack of space, the proofs of the following theorems can be found in the supplementary material available at [2].

Theorem 1. *Compressing a sketch doesn't impact its queries.*

Theorem 2. *Expanding a sketch, updating it, followed by undoing the expand yields the same state as updating without performing the expand and undoExpand.*

Theorem 3. *Initialize GS with δ so that $D = \lceil ln(\frac{1}{\delta}) \rceil$, $W \in \mathbb{N}$. Additionally, select $M > 0$ - every time we increment GS's counters by a total of $u = \frac{M}{eD(B+1)}$, we expand GS by a single counter. After incrementing GS's counters by a total of N:*

- $actual(k) \leq GS.estimate(k)$
- $Pr[GS.estimate(k) > actual(k) + Mlog_B(N)] \leq \delta$

Corollary 3.1. *This accuracy guarantee is independent of W. We may select $W = 1$ for a minimal initial memory usage.*

V. Evaluation

Implementation: We have implemented GS, CMS, and DCMS, along with a CLI for bench-marking, in C++. An additional Python file is used to generate our graphs by communicating with our C++ CLI. The hash used is xxHash64 [6]. Our C++ sketch implementations are single-threaded.

Metrics:

Average Absolute Error (AAE): given a stream S and an item k with $count_k$ occurrences in S, the absolute error (AE) of $estimate_k$, the estimation of $count_k$, is $|estimate_k - count_k|$. Given $ES = \{estimation_k | k \in S\}$, AAE is the average of its estimations' absolute errors.

Average Relative Error (ARE): The relative error (RE) of $estimate_k$ is $\frac{|estimate_k - count_k|}{count_k}$. ES's ARE is the average of its estimations' relative errors.

Throughput: let g be the number of operations of type OP performed in time t, OP's throughput is $\frac{g}{t}$. We use the notation MOPS for "Million Operations per Second".

Zipfian Distribution: The probability density function of Zipfian distributions is $p(k) = \frac{k^{-a}}{\zeta(a)}$, with $\zeta(a)$ being the Riemann Zeta function and $a > 1$ being a constant.

Platform: We ran our benchmarks on a server running Ubuntu 18.04.6 LTS with 125GiB of RAM and an Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz, (8 Cores, 16 threads).

Parameters: We select W=5×10^4 ($\varepsilon = 5.44 \times 10^{-5}$) and D=5 ($\delta = 0.01$) for all of our sketches unless stated otherwise.

Dataset: We have used the 2019 New York network trace from CAIDA [1] as our main data stream. We interpret the source IP addresses of packets as 32-bit unsigned integers. IPv4 addresses are exactly 32 bits, while IPv6 addresses are 128 bits - the lower 32 bits are used. The size of our trace stream is $S_{real} \simeq 3.77 \times 10^7$. Curve fitting our trace to a Zipfian distribution yields $a = 1.02$. We have tested additional network traces from CAIDA and reached similar qualitative results, which can be found in the supplementary material at [2]. We also use synthetic traces of size $S_{syn} = 10^6$, generated according to a Zipfian distribution. The value of a differs for each synthetic trace, ranging from 1.01 to 2 in 0.01 increments, totaling 100 traces. The smaller a is, the more uniform our trace is, with keys having similar frequencies, and vice versa. When processing the stream, each packet with key k, corresponds to a single $update(k, 1)$ invocation.

A. DCMS vs GS

1) DCMS vs GS Expands: Figure 2 exhibits memory usage and ARE in GS-B2 (B marking a branching factor of 2) and DCMS when following DCMS's expansion strategy with constants $K = 2$ and $N = 10WD = 2.5 \times 10^6$. We have chosen N, K so that for every 32 updates, a single counter is added. Note that we have selected K such that DCMS allocates sketches as frequently as possible. The left sub-figure presents the memory usage of each sketch while processing the CAIDA trace one packet at a time. The right sub-figure presents the ARE of the same sketches.

We evaluate the linear (LIN), exponential (EXP) and fine (FIN) expansion strategies, introduced in section III. For each strategy, we evaluate GS without compression (U) and when compression is applied immediately after each expand (C). As both compressed and uncompressed GSs use the same amount of memory, the compressed GSs allocate more counters.

As can be observed, the size of the FIN GSs grows 6 times over while processing the trace, while LIN and EXP only allocate a single sketch, twice as large as the original. This highlights how DCMS requires large allocations, while GS is more flexible, able to utilize small amounts of available memory. Due to this, when utilizing the FIN strategy, both the uncompressed and compressed GS provide a better accuracy.

GS has an identical ARE with EXP and LIN (since both strategies have the same allocations), being more accurate when compressed. DCMS is more accurate than GS when they utilize EXP or LIN.

2) DCMS vs GS Heavy hitters error: Figure 2 compares the ARE of varied GSs and DCMSs configurations. In Figure 3 we plot the ARE and AAE of each sketch after processing the entire trace, when considering a subset of heavy hitters.

The Y axis displays the ARE or AAE of a subset of heavy hitters, while the X axis determines the number of heavy hitters included, ranging from the top 0.1% most frequent addresses to the top 10% most frequent. There are roughly $3.7 * 10^5$ unique addresses, therefore the subset size ranges from 377 to $3.7 * 10^4$. We can observe that the AAE of all sketches is constant regardless of the heavy hitter set size. The ARE maintains the same distinct ordering of sketches by accuracy, the same as in Figure 2. Since the AE of different flows is uniform, the larger the flow the smaller the RE. This occurs due to the AE being a small percentage of a large flow's size.

B. Branching Factor

1) CAIDA Branching Factor Comparison: Figure 4 illustrates 3 GSs, differing by their branching factor, marked accordingly, and a CMS (L0 marking the CMS width is the same GSs' layer 0, i.e., W). While processing the stream, we expand our GSs at regular intervals with minimal granularity,

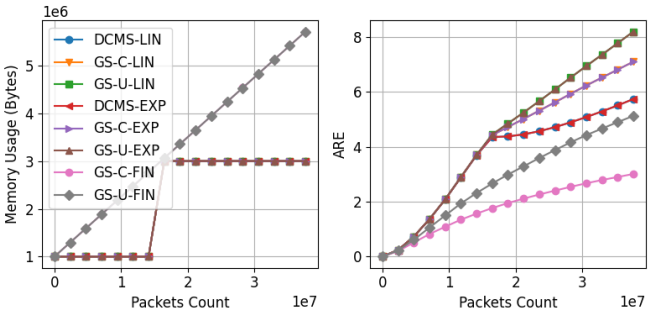

Fig. 2: ARE of varied GS and DCMS configurations

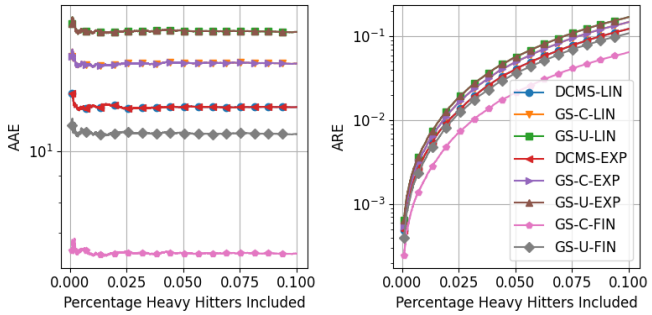

Fig. 3: GS vs DCMS heavy-hitters error

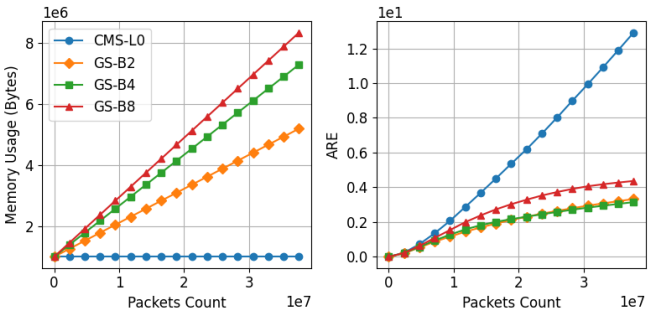

Fig. 4: GS fine expand and compress with differing B

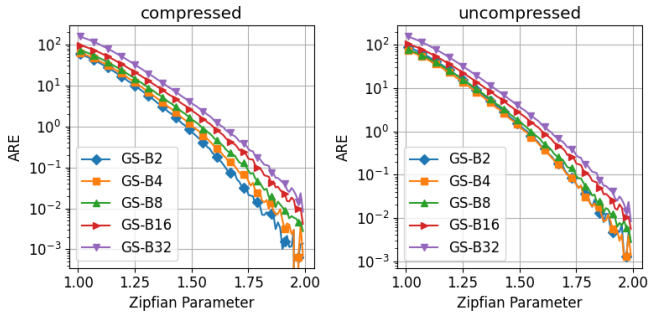

Fig. 5: ARE of GS with different branching factor, skew

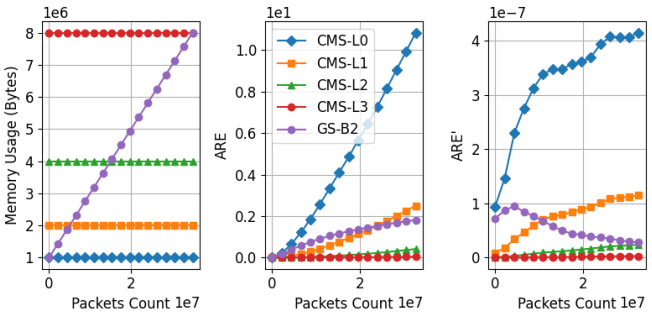

Fig. 6: GS fine expand and compress vs layer sized CMS

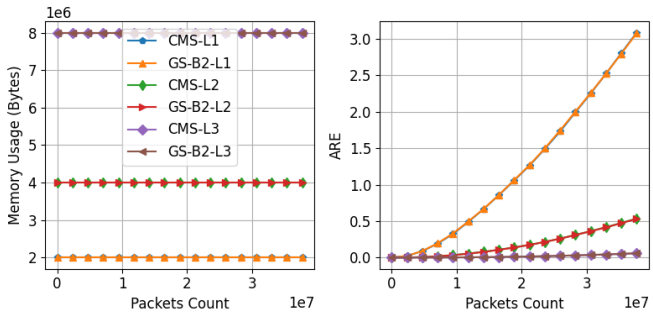

Fig. 7: GS expand, compress on creation compared to CMS

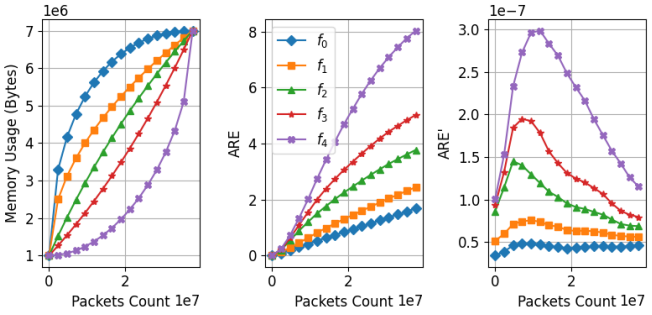

Fig. 8: GS different expansion strategies

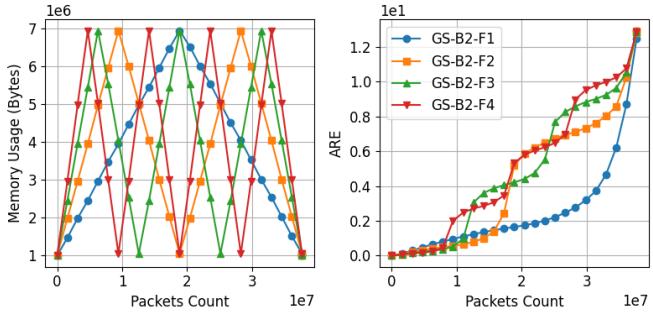

Fig. 9: GS cyclical expand and undoExpand

adding a single counter. Marking the maximum branching factor $B_{max} = 8$, we expand every $\frac{S_{real}}{W \cdot D \cdot B_{max}}$ updates, such that at the end of the stream we have filled layer 1 of the GS with the maximum branching factor. Every DB expands, we call compress(D) to remove the parents of added counters. This is the most aggressive compression possible for this expansion strategy. Compression does not affect accuracy.

We can observe that GS-B8 has a higher ARE throughout the trace when compared to GS-B4 and GS-B2, which have very similar ARE throughout the trace. Note that the lower the branching factor, the more aggressive compression can be applied. Therefore, GS-B2 uses the least memory, with a counter removed for each two added. Observe how initially the CMS's ARE is tangent to the GSs' ARE, yet their slope decreases with the increase in memory usage.

2) Synthetic Branching Factor Comparison: Figure 5 compares between the ARE of GSs with varying branching factors as the stream's skew increases. The horizontal axis represents the Zipfian parameter, and each ARE value is the final ARE after the entire stream is processed. Note that B does not have to be a power of 2. The different GSs have the same memory consumption while processing the trace, with expands at regular intervals such that at the end of the stream, layer 1 of the GS with the maximum branching factor (32) is filled. We provide both a compressed and an uncompressed comparison. Compression is done as early and as much as possible. In the compressed version, we can see an inverse correlation between the branching factor and the accuracy. Note that the GSs have the same memory usage during processing, and also that a GS with branching factor B can compress a counter every B expands. Thus, smaller branching factor implies more expands. For the uncompressed version, GS-B4 has the lowest ARE with GS-B2 as a close second. In general, accuracy improves with the skew, due to having fewer unique keys.

C. Expansion Strategy

1) Static GS vs CMS: Figure 7 exhibits 3 CMSs sized as a GS-B2's layer 1, 2 and 3. We compare these to 3 GS-B2, which immediately following initialization are expanded to fill their i^{th} layer, and immediately afterward completely compressed. Hence, GS-B2-Li contains just the i^{th} layer.

As can be observed, the accuracy of each GS-B2-Li and matching CMS-Li is identical. This is to be expected since a compressed GS containing a single layer functions like a CMS with a more complex hash function. Note that CMS-L0 and GS-B2-L0 are not included since they would return completely identical queries.

2) GS Layers and CMS Layers Comparison: Figure 6 compares the ARE of a GS-B2 and three CMSs, with CMS-Li having the same width as the i^{th} layer of a GS with $B = 2$ ($W \cdot B^i$). We expand GS-B2 by a single counter at regular intervals, such that at the end of our stream, GS's layer 3 is filled. We anticipate that the change in ARE for a compressed GS and a CMS of similar width would be similar.

To that end, we have added the derivatives of ARE, denoted ARE'. We use the most aggressive compression possible for GS-B2 (which does not affect accuracy) during this experiment. As shown, the ARE derivative of the expanding GS-B2 and those of the CMS layers are rather close. The difference from the static case could be explained by errors introduced while memory consumption was low.

3) Other Expansion Strategies: Figure 8 compares possible expansion strategies with the same GS-B2. We chose a selection of monotone rising functions, since we expect this to be a common use case. The expansion strategies match the following functions, ordered from most concave to least:

$$f_0(x) = sin(\frac{\pi}{2}\sqrt{x}) \qquad f_1(x) = \sqrt{x}$$
$$f_2(x) = log_2(x+1) \qquad f_3(x) = 2^x - 1$$
$$f_4(x) = \frac{2}{\pi} arcsin(x^2)$$

With x ranging from 0 to 1 and representing the portion of the stream already processed, while the function's value ranges from 0 to 1, representing the percentage of additional memory utilized out of a set constant chosen. The purpose is to evaluate the effect of the expansion strategy on the error. This constant is chosen such that layer 2 of a GS-B2 is filled when the function is equal to 1. We expand in 256 equally spaced allocations, whose size is determined by the respective expansion strategy. The earlier we allocate memory (the more concave the function is), the lower the ARE over time is, due to benefiting from the additional memory for a longer time. Hence, allocating memory should be done as early as possible.

4) GS Undo Expand Effect On ARE: For the experiment reported in Figure 9, we create multiple GS-B2, and while processing the stream, we expand and undoExpand them cyclically, with the integer following F in their names symbolizing the frequency. We expand the GS until layer 2 is fully allocated, and undo until it contains only layer 0. We can observe the relation between memory usage and the ARE and its derivative. Note that the sketches reach the same final ARE, as they are undone to their original size.

D. Throughput

To calculate update and query throughput, we measured the time duration elapsed while sequentially updating or querying all items in the CAIDA stream. Expands are performed beforehand, if necessary. We have selected W=200 and D=5, to enable us to test up to 8 layers, with a branching factor of 8.

1) DCMS Update and Query: Figure 10 presents the update and query throughput of DCMS as a function of the number of sketches (analogous to the number of layers in GS) and the value of the K parameter, relating to the exponential expansion strategy of DCMS and analogous to GS's branching factor.

Since a DCMS with 1 sketch is exactly a CMS, the first row of this table represents the throughput of CMS with width W and depth D. We have verified that a regular CMS of these dimensions indeed has a throughput of 16 MOPS.

A query in DCMS must access all stored CMSs. The rows in both tables for query have the same number of hash calculations and memory accesses. I.e., a DCMS with S CMSs

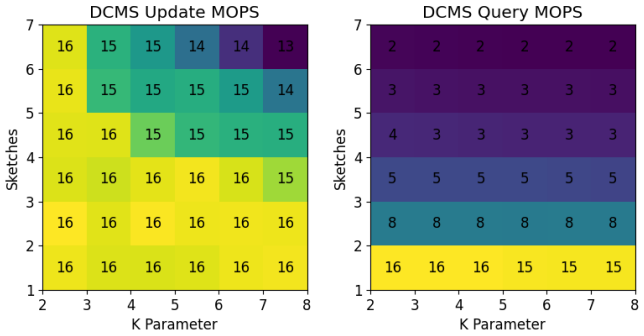

Fig. 10: DCMS update and query throughput

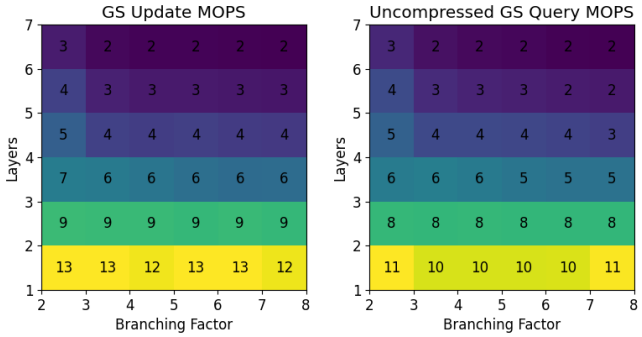

Fig. 11: GS update and uncompressed query throughput

requires $S \cdot D$ hashes and memory accesses per query. On the other hand, an update to DCMS only touches the top CMS, thus only D hashes and accesses are required. Hence, the update throughput is almost constant. Only when the memory usage becomes large enough does the throughput lowers.

2) GS Update and Query: Figure 11 presents the update and uncompressed query throughput of GS as a function of the branching factor and the index of the last full layer. Layers are either fully allocated or completely unallocated.

The number of hashes calculated for both update and query is $D \cdot B \cdot L$, for depth D, branching factor B, and L layers. Each update requires $D \cdot B$ memory accesses, while queries require $D \cdot B \cdot L$ memory accesses when uncompressed (compressed counters are not accessed, so we can reduce this to $D \cdot B$).

When utilizing a single layer, GS's query and update throughput are lower compared to DCMS and CMS, due to the added logic of GS. For update, GS requires more hash calculations, $D \cdot B \cdot L$, while DCMS requires only calculating the hashes of the top CMS, thus, DCMS is faster for updates. For querying, GS is slightly faster when using multiple layers, which could be due to the different memory access patterns.

VI. Optimizations and Extensions

A. Heavy Hitters

Elastic sketch [23] presents an elegant adaptation to handling heavy hitters, which can be employed alongside our implementation as well. The light part, implemented by a Count-Min Sketch in Elastic Sketch, can be replaced with our Geometric Sketch, which also supports the same operations as a Count-Min Sketch. The heavy hitters will be stored in the heavy part, as in [23]. This would allow Elastic Sketch to save memory while allowing more flexibility.

B. Optimizing Large Allocations

When allocating an entire layer for all rows at once, we can consider the set of the same depth layers as an actual CMS. On query, instead of summing the counter value for these layers with the rest of the row, we ignore the wholly allocated layers and select the minimum estimation from these layers, afterward adding this value, as in DCMS. Some of our layers can be atomically expanded, while the rest function normally. We have found that such large allocations improve our accuracy when uncompressed, behaving similarly to DCMS, which is more accurate when allocating in sketch sized chunks, but contradict the flexible nature we had in mind.

C. Reducing Hash Calculations

Updating and querying GS requires multiple hash calculations - one per layer, for all rows. Yet, each hash calculation produces much more bytes than what is needed for our use case. To reduce the number of hashes, we can split a single hash into multiple sub-hashes. This concept is not novel [19].

Selecting the first counter requires $\log_2(W)$ bits - 32 bits suffice. The branching factor is expected to be less than 16, so 4 bits are required for each additional layer. Therefore, a 64-bit hash yields enough sub-hashes to reach layer 8, even when we are extremely pessimistic. Since memory usage grows exponentially with the number of layers, a single hash calculation per row suffices, the same as CMS.

VII. Conclusions

In this paper, we have introduced the novel Geometric Sketch, a flexible sketch useful for flow frequency estimation in streams. It improves upon existing solutions' flexibility, as it is able to both allocate and free memory at very fine granularity. When freeing memory, it supports both lossless compression and undoing of expansions.

We have formally analyzed the estimation error of GS when a single counter is expanded at regular intervals. Our analysis method can be used to find similar bounds for other expansion strategies. We have shown that GS has a similar ARE buildup to CMS with the same memory usage, even if GS has reached this memory use following an expand or undo. We have shown that the branching factor has a small impact on accuracy when expanding, with a slight advantage to smaller branching factors.

A branching factor of 2 or 4 was found to be the best when uncompressed, while a branching factor of 2 for compressed - since the smaller the branching factor, the more aggressive compression can be. In general use cases, a branching factor of 2 is most practical. To improve accuracy, allocations should be done as early as possible. We have also shown that by using GS's increased flexibility (for finer expands) and its ability to compress itself, we can achieve a better accuracy than DCMS.

Code Availability: All code is available online [2].

REFERENCES

[1] Anonymized Internet Traces 2019. https://catalog.caida.org/dataset/passive_2019_pcap. Dates used: Equinix NYC, 20190117-30400UTC.

[2] Anonymous Anonymous. GS Implementation. https://github.com/ThrowAndAway/GS, May 2024.

[3] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized Admission Policy for Efficient Top-k, Frequency, and Volume Estimation. *IEEE/ACM Transactions on Networking*, 27(4):1432–1445, 2019.

[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. A Formal Analysis of the Count-Min Sketch with Conservative Updates. *Theoretical Computer Science*, 312(1):3–15, 2004.

[5] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding Frequent Items in Data Streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[6] Yann Collet. xxHash. https://xxhash.com/, 7 2023.

[7] Graham Cormode and Shan Muthukrishnan. An Improved Data Stream Summary: the Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[8] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *European Symposium on Algorithms (ESA)*, pages 348–360. Springer, 2002.

[9] Gero Dittmann and Andreas Herkersdorf. Network processor load balancing for high-speed links. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, volume 735. Citeseer, 2002.

[10] Roy Friedman, Or Goaz, and Ori Rottenstreich. Effective space saving. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking*, ICDCN '21, page 66–75, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Haoyu Li, Qizhi Chen, Yixin Zhang, Tong Yang, and Bin Cui. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proc. VLDB Endow.*, 15(7):1426–1438, mar 2022.

[12] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 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*, KDD '20, page 1574–1584, New York, NY, USA, 2020. Association for Computing Machinery.

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

[14] Lei Liu, Tong Ding, Hui Feng, Zhongmin Yan, and Xudong Lu. Tree sketch: An accurate and memory-efficient sketch for network-wide measurement. *Computer Communications*, 194:148–155, 2022.

[15] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter Braids: a Novel Counter Architecture for Per-Flow Measurement. In *ACM SIGMETRICS*, pages 1799–1807, 2008.

[16] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 346–357. Elsevier, 2002.

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

[18] Biswanath Mukherjee, L Todd Heberlein, and Karl N Levitt. Network intrusion detection. *IEEE network*, 8(3):26–41, 1994.

[19] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 743–759, April 2022.

[20] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application Level Ballooning for Efficient Server Consolidation. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys, page 337–350, 2013.

[21] Cha Hwan Song, Pravein Govindan Kannan, Bryan Kian Hsiang Low, and Mun Choon Chan. Fcm-sketch: generic network measurements with data plane support. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 78–92, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 2584–2593, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proc. of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 561–575, 2018.

[24] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. 10(11):1442–1453, aug 2017.

[25] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[26] Xiaobo Zhu, Guangjun Wu, Hong Zhang, Shupeng Wang, and Bingnan Ma. Dynamic count-min sketch for analytical queries over continuous data streams. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 225–234, 2018.