DHS: Adaptive Memory Layout Organization of Sketch Slots for Fast and Accurate Data Stream Processing

Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu*
Tsinghua University

ABSTRACT

Data stream processing is a crucial computation task in data mining applications. The rigid and fixed data structures in existing solutions limit their accuracy, throughput, and generality in measurement tasks. We propose Dynamic Hierarchical Sketch (DHS), a sketch-based hybrid solution targeting these properties. During the online stream processing, DHS hashes items to buckets and organizes cells in each bucket dynamically; the size of all cells in a bucket is adjusted adaptively to the actual size and distribution of flows. Thus, memory is efficiently used to precisely record elephant flows and cover more mice flows. Implementation and evaluation show that DHS achieves high accuracy, high throughput, and high generality on five measurement tasks: flow size estimation, flow size distribution estimation, heavy hitter detection, heavy changer detection, and entropy estimation.

CCS CONCEPTS

• **Information systems → Data stream mining**.

KEYWORDS

Data stream processing; Approximate frequency estimation; Sketch

ACM Reference Format:
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 and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3447548.3467353

1 INTRODUCTION

Massive data stream processing is a basic computation scheme in various applications such as network monitoring [13, 16, 19], sensor management [1], stock tickers [4], recommendation systems [11] and anomaly detection [29]. A data stream consists of a sequence of data items, each of which has an ID. A *flow* represents items with the same ID. The data stream processing algorithm takes in the stream and outputs *statistical results* on flow level (e.g., distribution, entropy). In the applications above, data are generated at a high rate, and long-term and high-volume storage is not affordable or outweighs its benefits. Thus, data stream processing algorithms store data temporarily in limited memory, process each item in one pass, and keep the statistics in their data structures for periodical or final queries.

The data stream processing in various applications can be abstracted as five typical *measurement tasks*: (1) flow size estimation, (2) flow size distribution estimation, (3) heavy hitter detection, (4) heavy changer detection, and (5) entropy estimation. For example, in the software platform, electing the hottest data flow (e.g., web clickstream) can help recommendation systems make decision [11] and guarantee the quality of service [30]; filtering out heavy hitters or heavy changers contributes to identifying attackers in DDoS defense [6, 12]; the frequency distribution or entropy of data stream reflects the system state and can be applied to mine anomalies [29]. In the hardware platform such as programmable switches, the accuracy of finding top-k flows is critical to traffic offloading [15]; and flow counting is a common component for network functions and their hardware implementations [9].

The stream processing structure (i.e., data structure and its read/write methods) plays a key role for performance in these tasks. It should provide an estimation of *flow size*, which could be further used for the measurement tasks. Various solutions for stream processing have been proposed in the past two decades, and each of them focuses on specific measure tasks and has its performance emphasis (memory efficiency, accuracy, or throughput). Existing solutions can be classified into three categories: sketch-based solutions, counter-based solutions, and hybrid solutions. Sketch-based solutions provides fuzzy information of all flows with high throughput (count-min sketch [7], CU-sketch [10], reversible sketch [23] and Asketch [22]). Counter-based solutions record accurate information of elephant flows with poor throughput (space-saving [20], lossy counting [18] and unbiased space-saving [25]). And hybrid solutions combine key ideas of the above two solutions to make a trade-off between throughput and accuracy, and support more measurement tasks well (HeavyGuradian [26], Cold Filter [31], ElasticSketch [27], HeavyKeeper [28], WaveSketch [14]).

We observe that all existing solutions organize the basic counting unit — sketch/counter slots — in a rigid and fixed layout. We propose to organize the slot size *dynamically and adaptively* to the actual flow size and distribution. Thus, the limited memory can be used more efficiently, which further improves the quality of flow size measurement. Essentially, three factors affect the measurement quality — flow coverage (for tasks 2, 4, and 5), all flow size estimation (for tasks 2 and 5), and elephant flow size estimation (for tasks 1 and 3). If slot size can dynamically adapt to flow ID and its frequency, high-frequent flows can be ultimately preserved and the remaining memory can host more low-frequent flows. And three factors can be preserved and improved.

*Wenfei Wu is the corresponding author.

KDD '21, August 14–18,2021, Virtual Event, Singapore

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467353

Table 1: Notation and Their Meanings

Notation	Meaning
d	a data item
D	a data stream with multiple items
N	the size of data stream
e	a data flow
F	a flow set with multiple flows
n	the size of a flow set
f_i	the actual frequency of flow e_i
$\hat{f}_i$	the estimated frequency of flow e_i
θ	the preset threshold (task 3 and 4)
c_i	the actual number of flows of size N_i
$\hat{c}_i$	the estimated number of flows of size N_i
Δ	a flow's change degree between time windows
B	the number of buckets
W	the bucket size, i.e., the width of its array
b	the exponential decay parameter
l_k	bit length of a level-k cell's counter
λ_k	portion of flow size at level k
w_k	total number of flows at level k

We propose Dynamic Hierarchical Sketch (DHS), a new data structure for stream processing, which is orthogonal to existing solutions. DHS is a hybrid data stream processing solution, which inherits sketch's advantage of high throughput. The memory layout of each slot of DHS (also called a bucket) is organized dynamically and adaptively to the size of flows in it — large flows with more bits and small ones with fewer. The memory layout organization is *self-tuned*, where cells (i.e., counters) in each slot is initially assigned fewer bits and re-organized with more bits as flow frequency is accumulated. Moreover, DHS introduces a special query mechanism named "longest fingerprint first" to accelerate retrieval. With these designs, DHS can support data stream processing with high throughput, high accuracy, and high generality (supporting all five tasks). Our contributions can be concluded as below:

- We propose a data structure named Dynamic Hierarchical Sketch (DHS) for more accurate, generic, and faster data stream processing.
- We make the mathematical analysis of DHS's upper/lower error bounds.
- We build DHS and other four recent solutions: HeavyGuardian [26], ElasticSketch [27], HeavyKeeper [28], and WaveSketch [14] for performance evaluation in the five measurement tasks above. The implementation is in C++ and opensourced in Github [24].
- We use experiments to show that DHS outperforms other solutions: it has higher accuracy (e.g., 2.10x ∼ 392.47x lower flow size estimation ARE), better memory efficiency (e.g., one-third memory cost to achieve 0.9 F1-Score in heavy hitter detection, compared with the state-of-the-art solution), and higher throughput (faster item processing speed than most hybrid solutions).

2 BACKGROUND

We elaborate the five typical measurement tasks in stream processing, the performance requirements, and our intuition to improve the performance metrics.

2.1 Problem Formulation

Table 1 shows the definition of notations used in this paper. Let $D = \{d_1, d_2, ..., d_N\}$ be a data stream containing N items (e.g., a network stream with N packets). Each item belongs to only one flow, which is denoted as e. Items in a data stream D can be classified into n non-overlapping flows: $F = \{e_1, e_2, ..., e_n\}$. The number of items in each flow is called frequency $e_i.f$ (or f_i briefly), and so we have $\sum_{i=1}^{n} f_i = N$. A flow also has a unique ID to identify itself, represented as $e_i.id$ (e.g., the 5-tuple headers are usually used to identify a TCP flow in a network stream). All items of the same flow get the same IDs (i.e., $d_i.id = e_i.id$ if $d_i \in e_i$).

Interfaces. A data stream processing structure should provide two fundamental interfaces: **Insert()** and **Query()** to support measurement tasks in practice. Other more complicated data mining metrics can be achieved by the combination of these two primitive actions and some auxiliary memories when needed. The interfaces are defined as follows.

- **Insert**($d.id$): To insert an incoming item of the data stream, the processing structure would first retrieve its item ID; then depending on the available memory and the algorithm, the event of this item's arrival (i.e., frequency 1) would be recorded or discarded.
- **Query**($e.id$): Given a flow ID $e.id$, **Query**($e.id$) will return the frequency (which could be approximate) of the queried flow. It is worth noting that the processing structure does not record the complete flow IDs due to limited memory and thus can not support retrieval without any ID information (e.g., semantics like "list all flows' frequency" without giving the flow IDs).

2.2 Measurement Tasks

We abstract five typical measurement tasks from application scenarios.

Task 1: (Top-k) Flow Size Estimation. It provides the (approximate) frequency of each flow. In many practical cases like traffic offloading [15], elephant flow size estimation draws more attention, where the algorithm would make memory-efficient item counting at the cost of sacrificing mice flows' counting. Top-k accuracy is used more as the performance metric.

Task 2: Flow Size Distribution Estimation. It reports the number of flows belonging to each specific size. Flow size distribution estimation is widely used in database query load balancing [21] and anomaly detection by distribution [29].

Task 3: Heavy Hitter Detection. It aims to find a flow set F_{hh} satisfying that $\forall e \in F_{hh}, e.f > \theta_{hh} \cdot N$, where θ_{hh} is a predefined threshold. Heavy hitter detection is important in data mining applications like recommendation systems and information retrieval [8].

Task 4: Heavy Changer Detection. For each flow appearing in the time window T or $T-1$ (a measurement interval can be divided into several time windows), its change degree can be denoted as $\Delta = |e.f^T - e.f^{T-1}|$. Heavy changer detection reports the flows whose Δ is larger than $\theta_{hc} \cdot N$, where θ_{hc} is a predefined threshold.

Task 5: Entropy Estimation. It returns the entropy of flow sizes to describe its distribution and uncertainty. Flow entropy is applied in data mining works including clustering [5] and data quality evaluation [17].

2.3 Performance Requirements and Metrics

A stream processing structure needs to have high throughput, high accuracy, and high generality to support all tasks and applications above.

High Throughput. The high data generation rate and the limited storage space require the data stream to be processed in one pass, and the processing rate should be at least as fast as the data generation rate.

High Accuracy. Since the memory is limited, most existing solutions are approximate ones. The distance from the approximate results and its actual value would directly influence the quality of the measurement and subsequent application. Thus, the processing structure had better record item frequency accurately.

High Generality. Being generic means supporting all five tasks with practically acceptable accuracy and throughput. While various structures are proposed, each of them targets a specific measurement task above; deploying multiple stream processing structures simultaneously is not applicable due to the memory constraint. Therefore, a generic processing structure to support all five measurement tasks above is required.

2.4 Related Work

Existing data stream processing structures fall into three categories: sketch-based structures, counter-based structures, and hybrid structures. But they can hardly satisfy all the three requirements above.

2.4.1 Sketch-based Solutions. The basic sketch-based structure (sketch for short) — count sketch [3] — contains an array of B slots to count frequency. Each incoming item d's ID is $d.id$ hashed into $[0, B-1]$ to index a slot and increases the slot's frequency by one. The **Query()** action follows the same indexing process to get the slot's frequency. A more widely used variant of the sketch is count-min sketch [7]: it imports multiple pairs of the array and hash function to repeat the same insertion as count sketch for each pair. The **Query()** action will return the minimum result among all arrays. Subsequent researches keep improving sketches in the past twenty years. For example, CU-sketch [10] improves the accuracy by only increasing the frequency of the smallest cell among multiple arrays, reversible sketch [23] uses flow ID encoding and decoding to enable deletion of flows, and ASketch [22] uses an extra elephant-flow recording structure to improve accuracy.

As sketches use the hash value as the index, they can support $O(1)$ insertion and query, providing high throughput. Sketches store the elephant and mice flow information in the same space. Thus, flow information is mixed. Elephant flows are counted with slight over-estimation (supporting tasks 1 and 3), and mice flows would be falsely estimated as a large flow with a small probability (supporting task 2 and 5).

2.4.2 Counter-based Solutions. The basic counter-based structure (counter for short) — space-saving [20] — can be regarded as an array of W slots, where each slot stores a <key, value> pair. The key is the flow ID, and the value is the flow's frequency. When a new flow comes in, the flow of the smallest frequency in the store will be swapped out. The **Insert()** and **Query()** need to traverse all slots in $O(W)$ time complexity. Prior arts such as lossy counting [18] and unbiased space-saving [25] adjust the replacement strategy of counters to achieve higher accuracy.

Counters provide higher accuracy in elephant flow estimation because their replacement policy prefers to preserve large flows in the slots and each slot has a flow ID to exclude being falsely identified as a mice flow. As a cost, the counter stores no information of mice flows (and return 0 when queried) and thus is not friendly for tasks related to size distribution (task 2 and 5). Moreover, the overhead of traversing the array in insertion and query limits the counter's throughput.

Table 2: High-level Comparison among Schemes

Scheme	Accuracy	Generality[1]	Throughput
Sketch	Low	Moderate	High
Counter	Moderate	Low	Low
Hybrid Structure	Moderate	High	Moderate
DHS	High	High	High

2.4.3 Hybrid Solutions. Many recent priors [14, 26–28] are hybrid solutions combining the two schemes above. By replacing the single frequency number in sketch slots with key-value pairs, a hybrid solution can flexibly make a trade-off between throughput and accuracy. The memory space is organized as an array of buckets, and each bucket stores several key-value pairs (i.e., counters). Each flow would be hashed by its ID to a bucket, and its $<flowID, flowfrequency>$ is stored in a bucket. A representative work is HeavyKeeper [28]. Furthermore, to balance the information storage between elephant flows and mice flows for different tasks, each bucket's counter can be divided into the heavy part and the light part in HeavyGuardian [26]. The former is responsible for recording complete elephant flow information, while the latter acts like another sketch to store the fuzzy information of mice flows.

Hybrid solutions make $O(1)$ access to each bucket and linear traversal within a bucket, which has moderate throughput compared with pure sketches and counters; its careful organization of each bucket enables the structure to record more flow information, which inherits the same accuracy as counters (or better). In this paper, we mainly use hybrid solutions for comparison because they provide the best performances in the three dimensions.

2.5 Goal and Approach

Goal. Inspired by the observation that all existing solutions have rigid and fixed memory layout within their basic counting unit — slot/bucket, we propose to dynamically and adaptively organize the layout of each basic counting unit. Our goal is to build a new sketch-based hybrid solution, which has the advantages of high throughput (inheriting from sketch), high accuracy (efficiently using memory to record more information), and high generality (supporting all tasks with high accuracy). The position of our work and the comparison with the related work are shown in Table 2.

Approach. We propose Dynamic Hierarchical Sketch (DHS) for this goal. It is inspired by the intuition to dynamically organize

[1] In our experiments, basic Skecth can support task 1, 2, 3, and 5; basic Counter can support task 1, 3, and 4; hybrid solutions including DHS can support all kinds of measurement tasks.

Table 3: Frequency Field Space Utilization of Counter

Counter Size	No. of Flows	Mem. Used/Wasted	Mem. Util.
16 bits	13	0.28KB/0KB	100%
12 bits	1217	19.47KB/4.87KB	75.26%
8 bits	17016	0.27MB/0.13MB	51.70%
4 bits	147626	2.36MB/1.77MB	27.93%

Table 4: Probability of Fingerprint collision

Flows/Bucket	$l = 8$	$l = 12$	$l = 16$
n/B= 10^2	0.324	0.0241	0.00152
n/B= 10^3	0.980	0.217	0.0151
n/B= 10^4	≈ 1	0.913	0.142

the memory layout of sketch slots (buckets) adaptively to the flow characteristics (i.e., actual size and distribution). Overall, such a dynamic organization improves accuracy by preserving elephant flows precisely and covering more flows, and improves the throughput by inheriting the hash-based bucket access ($O(1)$) to buckets and priority-based in-bucket traversal (early termination). The three design points of dynamic allocation are elaborated as follows.

Design 1: Adaptively allocate memory for frequency counting. In counters and sketches, each slot has a fixed size. Storing mice flows' frequency in such a slot could waste memory space when the value is small. Table 3 shows the memory usage of storing CAIDA [2] by a 16-bit counter. We can observe severe memory wasting in measurement results: the space utilization decreases as more mice flows are recorded.

In the duration of stream processing, all frequencies are monotonically increasing. Thus, it is wise to use fewer bits to initially record a frequency; as a flow's size increases, DHS would allocate more bits to it (and evicts smaller ones for allocation if no space). This design makes more efficient usage of the limited memory, covering more flows and consequently improve the accuracy for tasks 2 and 5.

Design 2: Adaptively allocate memory for flow ID. Pure sketches do not store flow ID, which may introduce false estimation of mice flow size. Counters and hybrid solutions store flow IDs to exclude small flows falsely recognized as elephant flows. In some solutions [26], the flow ID can be compressed as a *fingerprint* (hashing the flow ID to a smaller range); this compression saves memory but introducing a probability of false recognition (hash collision between flows). If n flows are hashed to B buckets or slots, each of which has a fingerprint of l bits, the collision probability is

$$Pr\{fingerprint\ collision\} = 1 - (1 - 2^{-l})^{\frac{n}{B}}.$$

This probability varies with n, B, and l in Table 4.

However, the false recognition of mice flows does less harm to estimation accuracy. All existing solutions allocate the same rigid size of memory to each flow's ID, which is a waste when storing a mice flow ID/fingerprint. Therefore, we propose to classify flows within a bucket into groups according to their size and give different groups different per-flow space to store flow ID/fingerprint — the elephant-flow group allocates more bits for each flow's fingerprint, and the mice-flow group allocates fewer. This design makes each group have fewer flows, reducing the probability of hash collision; elephant flows with longer fingerprints would have a lower probability of being falsely recognized as other flows; mice flows with shorter fingerprints would have false flow recognition, but they influence the accuracy less significantly than elephant flows. The saved memory can be used to cover more flows.

During the online processing, when a flow's frequency is small, its fingerprint is assigned with fewer bits; as its frequency increases to a larger range, its fingerprint would be re-computed in a larger range.

Design 3: Adaptively allocate portion of memory to elephant/mice flows. At the beginning of the stream processing, all flows are small and are stored. As the data stream is processed, some flows' frequency would reach a threshold and become elephant flows. DHS would allocate more space to store the ID and frequency of these elephant flows; if there is no more space, it would evict small flows and reallocate the space to elephant flows. Thus, elephant flows can be ultimately preserved, and the remaining memory can be fully used to record mice flows.

3 DESIGN OF DYNAMIC HIERARCHICAL SKETCH

DHS is a hybrid solution, where the storage space within each bucket is organized in a hierarchical structure. Each bucket adaptively and automatically give suitable space for flows (fewer bits for a mice flow and more bits for an elephant flow). Among buckets, if the flow to bucket distribution is skewed, a bucket with more elephant flows would first store the elephant flows; and a bucket with fewer elephant flows would store more flows, each of which with fewer bits. This design inherits the advantage of high throughput from the sketch, increases the accuracy by storing elephant flows precisely and covering more mice flows, and is generic to support all five measurement tasks.

3.1 Data structure

The data structure of DHS is an array of B buckets. Each bucket consists of a circular array and metadata. The circular array is divided into several consecutive and logically hierarchical segments. Each segment has several basic cells — key-value pairs of $<fingerprint, frequency>$, but the pair's size is different and specific to its segment. The boundary between segments is adjustable and is stored in the bucket metadata (i.e., storing the starting index of the segment).

In DHS, each bucket is a three-level circular array. The fingerprint and frequency fields contain the same bits (8, 12, and 16 bits for level-1, level-2, and level-3 cells respectively). The total length of the circular array (bucket size) is fixed as W bits. The architecture of DHS is in Fig 1. There are four hash functions h_0, h_1, h_2 and h_3 in DHS. h_0 is used to compute the index of a flow in the bucket array. h_1, h_2, and h_3 return the fingerprint of each level. The hash value ranges of four hash functions are $[0, B-1]$, $[0, 255]$, $[0, 4095]$, $[0, 65535]$ respectively. During data processing, each segment's

boundary is adjusted automatically according to the flows inside, which consequently adjusts the portion of cells in each level.

It is worth noting that when a bucket is full of the highest-level cells, each bucket is more like a counter-based solution — with long fingerprints (less collision) storing elephant flow first. On the contrary, when a bucket is full of the lowest-level cells, each bucket is more like a sketch-based solution — with short fingerprints, each cell would accumulate the frequency of a group of flows (hash collided to the cell). DHS's self-tuning can make each bucket find the best trade-off between the two extremes.

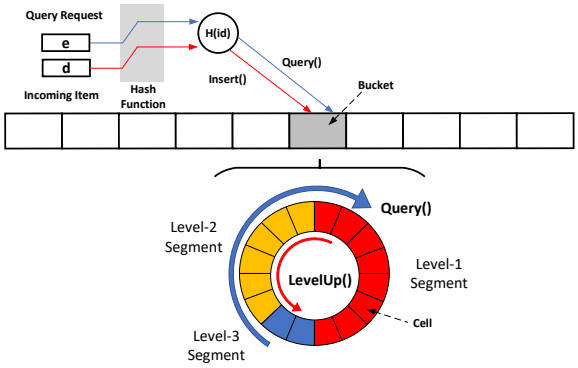


Figure 1: Architecture and corresponding workflow of Dynamic Hierarchical Sketch.

3.2 Workflow of DHS

The interfaces of DHS is also shown in Fig. 1. At the beginning of stream processing, all fields in DHS are initialized to zero, and each bucket only contains the lowest-level cells.

3.2.1 Operations. **Query.** The query process in DHS follows the principle of *longest fingerprint first*. When querying a flow, DHS first finds its bucket ($h_0(e.id)$), and then traverses the cells from the high level to the low level. In each level, the flow's fingerprint is computed ($h_k(e.id)$) and compared with each cell within that level. Once a match is found ($h_k(e.id) = fingerprint$), the cell's frequency is returned and the query terminates. Query() will return 0 if no cells match $e.id$.

This query principle is like the longest prefix first match in network switch rule lookup. Compared with other solutions, DHS query can traverse elephant flows first and early terminate once a match is found, reducing the lookup time and improving the throughput.

Insertion. When a new item d comes in, DHS first inserts it into the $h_0(d.id)^{th}$ bucket. Then in the bucket DHS executes a Query() action to check if the flow of d has been stored. Depending on the available memory space and the frequency value, there are four cases in executing the insertion, as shown in Fig 2. We use l_k to denote the field length in level k.

Case 1: Query() returns an existing record r at level k, and the frequency of r can be increased without overflow ($< 2^{l_k} - 1$). DHS would increase r's frequency by one directly.

Case 2: Query() returns an existing record r at level k, but increasing the frequency of r causes overflow ($= 2^{l_k} - 1$). Then DHS launches a "LevelUp()" action as follows.

It first attempts to find the space to put r in a higher-level cell ($k+1$): starting from level 0 to level k until one level has enough cells to deallocate (from the smallest frequencies) for new level-$k+1$ cells. There are three cases in DHS: two level-1 cells for a level-3 one, three level-1 cells for two level-2 ones, and four level-2 cells for three level-3 ones.

If the attempt succeeds, the deallocated cells are shifted to level $k+1$ to combine as new level-$k+1$ cells. The flow is put into one of new cells as $<h_{k+1}(d.id), 2^{l_k}>$ and its original cell at level k is cleared (kept as an empty cell).

If the attempt does not succeed, none of the deallocations above will happen, but an exponential decay will be applied on the smallest level-$k+1$ cell. The smallest cell's frequency f is decreased by one with a probability $Pr = b^{-f}$, where b is a preset parameter larger than one; if f is reduced to be smaller than r's frequency ($< 2^{l_k}$), the smallest cell is replaced by r with the pair $<h_{k+1}(d.id), 2^{l_k}>$ and r at level k will be cleared.

Case 3: Query() returns zero, and there remain empty level-1 cells. DHS would add a new level-1 cell $<h_1(d.id), 1>$.

Case 4: Query() returns zero, and there exist no empty level-1 cells. DHS will execute the exponential decay on the smallest level-1 cell and replace it with $<h_1(d.id), 1>$ once the smallest cell is decayed to 0.

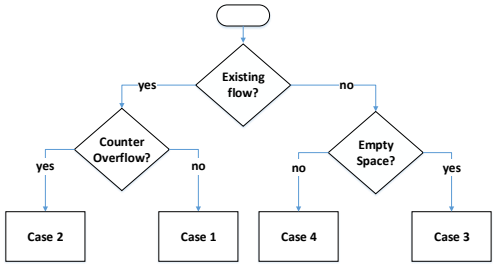


Figure 2: Execution Logic of Insert()

Examples. Fig. 3 shows two examples of insertion. For convenience, we display the circular array as a linear array. In the example of **Case 4**, item d_1 has not been recorded by DHS. And all level-1 cells are occupied by other flows. Therefore, the insertion of d_1 only causes an exponential decay on the smallest cell (i.e., e_b). While in the example of **Case 2**, item d_2 belongs to an existing flow in DHS. And after increasing the frequency by one, the level-2 cell overflows. Therefore, DHS launches LevelUp(). DHS traverses level-1 cells, deallocates the two smallest cells (i.e., e_d and e_e), shift cells to move the space to level 3, and allocates a new level-3 cell to record d_2's flow. Then the original level-2 cell is cleared for future use.

3.2.2 Throughput Analysis. The parameters of DHS include bucket number B, bucket size W, and exponential decay parameter b. We will discuss their influence on the system in Section 5.

DHS inherits the $O(1)$ complexity to access each bucket. Within each bucket, the optimization and the stream characteristics make

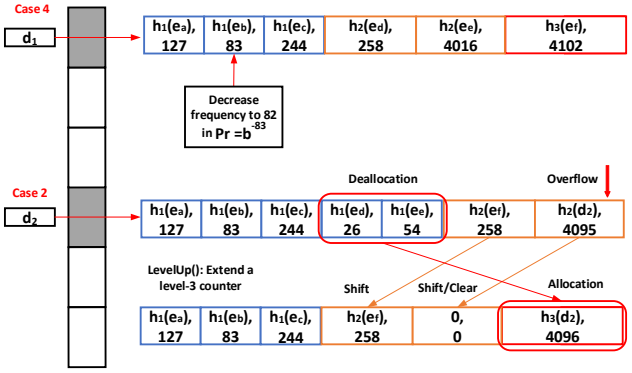


Figure 3: Insertion Examples of DHS

DHS faster than previous works: the longest fingerprint first improves the throughput by early termination (faster to access high-level cells), especially in the case where elephant flow dominates the total stream (about 41.61% in our test traces); LevelUp() is supposed to move $\theta(W)$ entries, but it is launched at most twice for each high-level flow and is amortized, and when there is a low portion of elephant flows, this extra complexity is negligible.

DHS has high accuracy for all five tasks (high generality). With its efficient memory usage, elephant flows are recorded precisely, and the remaining memory is used to record mice flows to the best extent. With a precise record of elephant flows and higher coverage of all flows, all five measurement tasks can be performed accurately.

4 ERROR BOUND ANALYSIS

We analyze the error bound of a DHS with B buckets, which records a traffic stream with N packets and n flows. Let e_i be the i^{th} largest flow, whose actual frequency is f_i. The bit length of level-k cells' fingerprint fields are l_k. We assume the average size of the level-k segment in the stable state is w_k, and the ratio of items falling in level-k segments across all buckets is λ_k. The final frequency estimation of flow e_i is

$$\hat{f}_i = f_i - X_i + Y_i, \tag{1}$$

where X_i is the decrement from exponential decay and Y_i is the increment from fingerprint collision. The two error bounds of DHS—the lower bound and the upper bound — result from X_i and Y_i respectively. $\hat{f}_i$ is also written as $\hat{f_{i,k}}$ to identify its corresponding level k.

4.1 Upper Bound Analysis

For any two flows e_i and e_j which are hashed into the same bucket and belong to the same cell at level k, we use $I_{i,j,k}$ to indicate that they has the same fingerprint. Then we have $E(I_{i,j,k}) = 2^{-l_k}$. Let Y_i denote the number of collisions occurring to e_i, so

$$E(Y_i) = E(\sum_{k=1}^{3}\sum_{j=1}^{n} I_{i,j,k} f_j) \le \sum_{k=1}^{3} min(\lambda_k \frac{N}{B} 2^{-l_k}, 2^{l_k} - 1), \tag{2}$$

where the first term is the expected number of collided items with e_i (average $N/B \times \lambda_k$ items in the bucket at level k, each with the probability 2^{-l_k} to collide) and the second term is the maximum value of a cell's frequency. Then we can use Markov inequality to transform the bound of expectation into the bound of possibility:

$$\begin{aligned} &Pr\{\hat{f}_i - f_i \ge \epsilon N\} = Pr\{Y_i - X_i \ge \epsilon N\} \\ &\le Pr\{Y_i \ge \epsilon N\} \le \frac{E(Y_i)}{\epsilon N} \le \sum_{k=1}^{3} min(\frac{\lambda_k}{\epsilon B} 2^{-l_k}, \frac{2^{l_k}-1}{\epsilon N}) \end{aligned} \tag{3}$$

4.2 Lower Bound Analysis

Here assumes that a flow finally stored in the level-k cell never encounters exponential decay at lower levels because it can not become the smallest one in lower levels. Exponential decay occurs to e_i only when e_i is the smallest one in its level. That means $w_k - 1$ among $i-1$ flows are hashed into the same bucket (each flow with probability $1/B$) when e_i is stored in the level-k counter, which follows a binomial distribution $B(i-1, 1/B)$. The probability of this event is

$$Pr_k = \binom{w_k - 1}{i-1} (\frac{1}{B})^{w_k - 1} (\frac{B-1}{B})^{i - w_k}. \tag{4}$$

This probability can be approximated by Poisson distribution $P(\frac{i-1}{B})$, therefore Pr_k can be rewritten as

$$Pr_k = e^{-\frac{i-1}{B}} \frac{\frac{i-1}{B}^{w_k - 1}}{(w_k - 1)!}. \tag{5}$$

The number of items $N_{i,k}$ that can apply exponential decay on e_i is the number of items at level k whose flow frequency is smaller than e_i. Its expectation is

$$E(N_{i,k}) = \frac{1}{B} \cdot (\sum_{j=i+1}^{n} f_j - \sum_{h=1}^{k-1} \lambda_h N), \tag{6}$$

where $1/B$ is the probability of an item in the same bucket with e_i, $\sum_{j=i+1}^{n} f_j - \sum_{h=1}^{k-1} \lambda_h N$ is total number of items whose flow size is smaller than e_i excluding the number of items at lower levels ($< k$).

Therefore, we get

$$\begin{aligned} E(X_{i,k}) &= Pr_k \times E(N_{i,k}) \times \frac{1}{E(\hat{f_{i,k}})} \sum_{j=1}^{E(\hat{f_{i,k}})} b^{-j} \\ &= Pr_k \frac{E(N_{i,k})}{E(\hat{f_{i,k}})} \frac{b^{-1}(1 - b^{-E(\hat{f_{i,k}})})}{1 - b^{-1}} \\ &\approx Pr_k \frac{E(N_{i,k})}{E(\hat{f_{i,k}})(b-1)}, \end{aligned} \tag{7}$$

where the third item is the expected probability when decay happens. So $E(\hat{f_{i,k}})$ can be written as

$$E(\hat{f_{i,k}}) = f_i - E(X_i) + E(Y_i) = f_i + E(Y_i) - E(X_{i,k}). \tag{8}$$

Replacing $E(X_{i,k})$ in the equation above and solving $E(\hat{f_{i,k}})$:

$$E(\hat{f_{i,k}}) = \frac{1}{2}(f_i + E(Y_i) + \sqrt{(f_i + E(Y_i))^2 - 4P_k \frac{E(N_{i,k})}{b-1}}). \tag{9}$$

And we use Markov inequality again to get the lower bound

$$\begin{aligned} &Pr\{f_i - \hat{f_{i,k}} \ge \epsilon N\} \le \frac{E(f_i - \hat{f_{i,k}})}{\epsilon N} \\ &= \frac{1}{2\epsilon N}(f_i - E(Y_i) - \sqrt{(f_i + E(Y_i))^2 - 4P_k \frac{E(N_{i,k})}{b-1}}). \end{aligned} \tag{10}$$

5 EVALUATION

5.1 Settings

Our testbed is a commercial off-the-shelf server with 2 Intel i9-7920X CPU, each with 12 cores running at 2.90GHz. All experiments of this paper are conducted in the testbed. Time measurement experiments are repeated ten times to smooth the system jitters.

The experimental dataset comes from CAIDA Anonymized Internet Trace [2], which is collected in the Equinix-Chicago monitor. In the traces, the data items are IP packets and the flow ID is identified by the 5-tuple headers (Source IP/Destination IP/Source Port/Destination Port/Protocol). The dataset contains 25M items and 1.6M unique flows.

We implement DHS and four typical hybrid solutions (baselines) for comparison: HeavyGuardian [26], ElasticSketch [27], HeavyKeeper [28], and WaveSketch [14]. They are all implemented in C++. We deploy the source code of previous works from authors and tune parameters for the best performance to guarantee a fair comparison (we do not use the SIMD version of ElasticSketch).

5.2 Metrics and Implementation

Different measurement tasks have different accuracy metrics. DHS is compared with the baselines in accuracy and throughput, and the generality is evaluated by how many measurement tasks can be accurately measured.

5.2.1 Accuracy metric for measurement tasks. **Flow Size Estimation** is implemented by using the **Query()** action to query each flow. We use the Average Relative Error (ARE): $\frac{1}{m}\sum_{i=1}^{m}\frac{|f_i-\hat{f}_i|}{f_i}$ of top-m flows to evaluate the accuracy of flow size estimation, where f_i is the actual frequency of flow e_i and $\hat{f}_i$ is the estimated frequency.

Flow Size Distribution Estimation is implemented by using auxiliary memory: using **Query()** to get the size of each flow and counting the number of flows of each size N_i ($e.f = N_i$) in the auxiliary memory. We use the Weighted Mean Relative Error (WMRE): $\frac{\sum_{i=1}^{z}|c_i-\hat{c_i}|}{\sum_{i=1}^{z}(c_i+\hat{c_i})/2}$ to evaluate the task, where z is the max flow size, c_i and $\hat{c_i}$ represent the actual and estimated flow number of size N_i.

Heavy Hitter Detection is implemented by using an auxiliary counter to record the total item number within a measurement interval and computing the threshold $\theta_{hh} \cdot N$, and storing and returning the flows whose frequency is larger than $\theta_{hh} \cdot N$ into a set $\hat{F_{hh}}$. This task can be regarded as a binary classification and evaluated by the F1-Score between the estimated $\hat{F_{hh}}$ and the actual F_{hh}.

Heavy Changer Detection needs two same data stream processing structures to store flows in two neighboring time windows $T-1$ and T respectively. For each target flow e, its change degree is $\Delta = |query^{T}(e.id) - query^{T-1}(e.id)|$. Flows whose change degrees are larger than a threshold will be identified as heavy changers and stored in a set $\hat{F_{hc}}$. Similarly, F1-Score between F_{hc} and $\hat{F_{hc}}$ is used to evaluate this task.

Entropy of a flow set F is defined as $H(F) = -\sum_{i=1}^{z} i\frac{c_i}{N}\log\frac{c_i}{N}$, where z is the max flow size. It is implemented using the flow size distribution estimation. We use the ARE $\frac{|H(F)-H(\hat{F})|}{H(F)}$ to evaluate its accuracy.

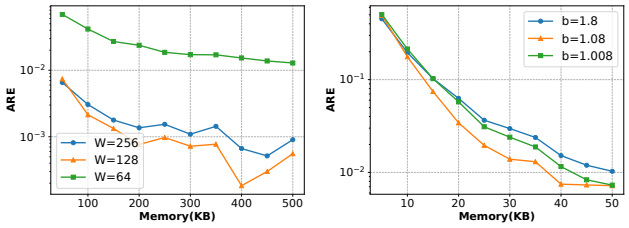
5.2.2 Throughput. There are two operations for data stream processing: Insert() is used to online process incoming items, and Query() is used to fetch the flow size periodically or finally (offline). Thus, we only care about throughput of **Insert()**. Suppose that a data stream processing structure costs t_N to insert N items; its throughput can be computed as $\frac{N}{t_N}$.

5.3 Parameter Tuning

DHS has three parameters: the number of buckets B, bucket size W, and exponential decay parameter b. We tune them to get the best performance. Given a fixed total memory in the experiment, B is decided by W (i.e., $W \cdot B$ equals the total memory). Thus, we only tune W and b. The tuning results are shown in Fig. 4.

Tuning W. We run top-k flow size estimation ($k = 1000$), with varying total memory and bucket size W, we show the absolute relative error (ARE) in Fig. 4(a). We observe that a too-large (256 bits) or too-small (64 bits) bucket size would hurt the accuracy, and an intermediate value of W (e.g., 128 bits) can contribute to good accuracy (low ARE). The reason is that a larger W could lead to less number of buckets and flow-to-bucket skewness, and a smaller W could cause each bucket unable to store elephant flows within the bucket.

Tuning b. The similar experiment of tuning b shows the same trend with W. An intermediate value of b could provide the optimal performance. In Fig. 4(b) we can observe that $b = 1.08$ outperforms other cases. Exponential decay parameter b controls the flow replacement of DHS. A larger b helps DHS eliminate mice flows from the system quickly, but is also likely to swap out elephant flows falsely; on the opposite, too small b makes DHS difficult to identify and keep elephant flows in the structure.



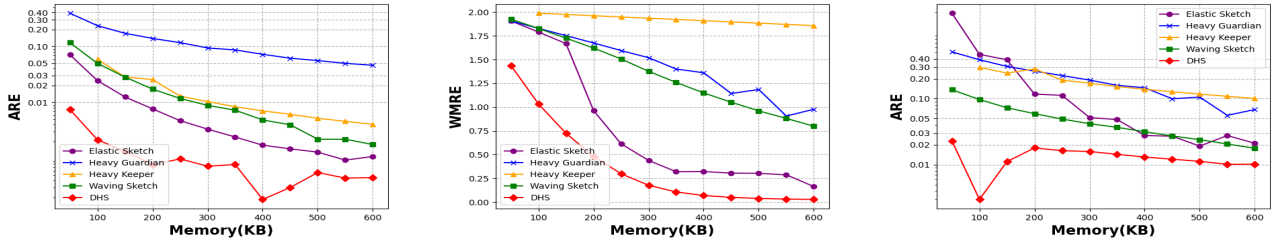
(a) Tuning on W (b) Tuning on b

Figure 4: Parameter Tuning Experiments

Parameter selection. Experiments on other measurement tasks show similar results. Due to space limitations, we do not provide the result. In the following experiments, we greedily select parameters of the best performance. Parameters are kept the same among all experiments.

5.4 Evaluation on Measurement Tasks

5.4.1 Accuracy. **Flow Size Estimation.** Fig. 5(a) shows experiments to evaluate five structures in top-k flow size estimation, where $k = 1000$ and the total memory varies from 50KB to 600KB. HeavyKeeper can not work with less than 100KB memory due to its independent heap structure. We observe that DHS outperforms other baselines: the ARE is 52.76x ~ 392.47x, 2.10x ~ 11.43x, 8.96x



(a) ARE of Flow Size Estimation (b) WMRE of Flow Size Distribution Estimation (c) ARE of Entropy Estimation

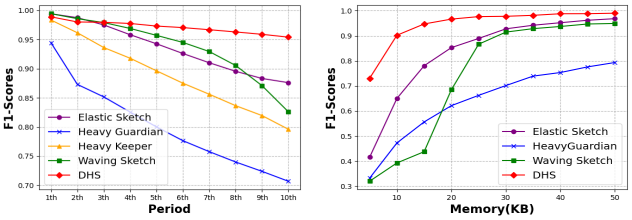
Figure 5: Measurement Tasks Evaluation under Variant Memories

~ 38.25x. 3.90x ~ 26.37x lower than that of HeavyGuardian, ElasticSketch, HeavyKeeper and WaveSketch. And the advantage of DHS is more significant when the memory is smaller.

DHS's auxiliary data structure is the index of segment boundary, which is negligible compared with the baselines, e.g., HeavyGuardian's light part (sketch), ElasticSketch's flags and $vote^-$, HeavyKeeper's duplicated counters, and WaveSketch' waving counters and flags. With more available memory and precise elephant flow recording, DHS is more accurate for top-k flow size estimation.

Flow Size Distribution Estimation. Fig.5(b) shows the WMRE of flow size distribution estimation in the same experiments above. DHS reports 1.32x ~ 35.19x, 1.33x ~ 9.07x, 1.92x ~ 66.89x, 1.34x ~ 28.73x lowers WMRE than that of HeavyGuardian, ElasticSketch, HeavyKeeper and WaveSketch respectively. We also observe that HeavyKeeper, which performs well in top-k flow size estimation above, shows a much worse accuracy in flow size distribution estimation. The reason is that HeavyKeeper's heap structure pays less attention to mice flows.

DHS persistently performs well in these two tasks (generality). Because in DHS, elephant flows are precisely recorded, and the remaining memory in each bucket is more efficiently allocated to more mice flows. And the flow coverage is essential for flow size distribution estimation.



(a) Periodic Detection under 150KB (b) Detection under Limited Memory

Figure 6: F1-Score of Heavy Hitter Detection

Heavy Hitter Detection. We divide the whole data stream into ten time windows of equal length. We set θ_{hh} as 0.02% as previous works do. We vary the memory size but only show the result of 150KB memory due to space limitations. Fig. 6(a) show the varying F1-Score at the end of each window. We observe a common trend where the F1-Score of all structures decreases as time elapses. The

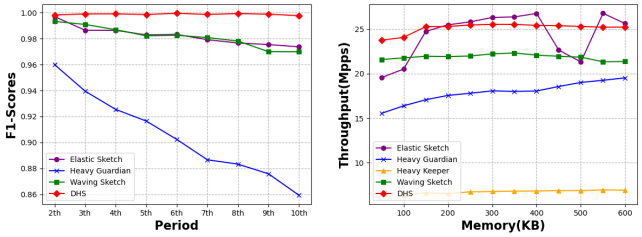


Figure 7: F1-Score of Heavy Changer Detection (150KB) **Figure 8: Throughput under Variant Memories**

reason is that a heavy hitter may need an exponential decay to evict the smallest flow and then get recorded. And as time elapses, more flows are recorded in the structure, causing new heavy hitters more difficult to be swapped in.

Nevertheless, we also observe that DHS decreases in the most graceful slope than baselines. It achieves 0.247, 0.078, 0.158, 0.128 higher F1-Score than HeavyGuardian, ElasticSketch, HeavyKeeper and WaveSketch respectively in the final time window. The reason is that DHS always give elephant flows priority than mice flows and can record more elephant flows than baselines.

Fig. 6(b) shows the final (the 10^{th} time window) F1-Score of each structure with varying size of memory, and the memory is limited in a small range (5 ~ 50KB). We adjust θ_{hh} to 0.2%. We observe that DHS performs extremely well under limited memory due to its highly efficient memory organization. To achieve a fixed F1-Score (0.9 as an example), DHS only costs one-third memory (10KB) than state-of-art solutions (30KB).

Heavy Changer Detection. We run the same experiment as above for heavy changer detection, where we set θ_{hc} as 0.05% and memory to be 150KB. The F1-Score is shown in Fig.7. We observe that DHS provides a persistent high F1-Score in the whole measurement duration (>0.99). But F1-Scores of HeavyGuradian, ElasticSketch, and WaveSketch decrease to 0.86, 0.97 and 0.97 after ten periods, and HeavyKeeper is too low (<0.5) to be shown in the range of the figure. DHS's accuracy in heavy changer detection is from its advantages of covering more flows and recording them accurately.

Entropy Estimation. We vary the available memory from 50KB to 600 KB to measure the ARE of entropy estimation. In Fig. 5(c), we

can observe that DHS reports 5.50x ~ 129.36x, 1.70x ~ 156.20x, 9.86x ~ 99.23x, 1.73x ~ 32.40x less entropy ARE than HeavyGuardian, ElasticSketch, HeavyKeeper and WaveSketch respectively. We also find that ARE of DHS's entropy decreases slowly with memory size increasing. DHS's good performance is still from its coverage of more mice flows.

5.4.2 Throughput and Generality. We measure the average throughput of structures with the growth of the total memory. The experiment results are in Fig 8. We have the following observations. First, DHS and ElasticSketch provide the top-2 highest throughput in most cases. ElasticSketch has a constant $O(1)$ access complexity (eight counters and a Count-Min Sketch), which makes it fast, but when available memory is limited, ElasticSketch will experience frequent entry swapping, causing throughput to decrease. While DHS always provides persistent high throughput with varying memory.

Second, DHS outperforms the other three baselines significantly. The reason is that the other three baselines contain a loop operation to traverse each cell, which is time-consuming. On the contrary, DHS's longest fingerprint first makes its looping able to terminate early, and the actual complexity is as low as that of ElasticSketch.

Considering the accuracy results of all five measurement tasks and the throughput, DHS satisfies the requirement of high generality for all five measurement tasks.

6 CONCLUSION

We designed Dynamic Hierarchical Sketch (DHS) to provide accurate, generic, and fast data stream processing for online stream measurement tasks. DHS is a hybrid structure with an array bucket and cells of key-value stores within each bucket. Orthogonal to existing solutions, DHS makes efficient use of per-bucket space by organizing cell size adaptively to the actual flow size and distribution. Implementation and evaluation show that the adaptive memory layout organization makes DHS achieve high throughput, high accuracy, and high generality for five measurement tasks. For the future work, we plan to implement DHS in more hardware platforms like P4 and FPGA. Using SIMD to accelerate DHS is also considered.

REFERENCES

[1] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. 2001. Towards sensor database systems. In *International Conference on mobile Data management*. Springer, 3–14.

[2] CAIDA 2008. Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml.

[3] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[4] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 379–390.

[5] Chun-Hung Cheng, Ada Waichee Fu, and Yi Zhang. 1999. Entropy-based subspace clustering for mining numerical data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 84–93.

[6] Graham Cormode, Flip Korn, Shanmugavelayutham Muthukrishnan, and Divesh Srivastava. 2003. Finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference*. Elsevier, 464–475.

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

[8] Josipa Crnic. 2011. Introduction to modern information retrieval. *Library Management* (2011).

[9] Marcus Vinicius Brito da Silva, Arthur Selle Jacobs, Ricardo José Pfitscher, and Lisandro Zambenedetti Granville. 2018. IDEAFIX: Identifying elephant flows in P4-based IXP networks. In *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–6.

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

[11] 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*. 535–540.

[12] He Huang, Yu-E Sun, Chaoyi Ma, Shigang Chen, You Zhou, Wenjian Yang, Shaojie Tang, Hongli Xu, and Yan Qiao. 2020. An Efficient K-Persistent Spread Estimator for Traffic Measurement in High-Speed Networks. *IEEE/ACM Transactions on Networking* (2020).

[13] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 113–126.

[14] 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*. 1574–1584.

[15] Xiuhua Li, Xiaofei Wang, and Victor CM Leung. 2016. Weighted network traffic offloading in cache-enabled heterogeneous networks. In *2016 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

[16] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 311–324.

[17] Zhetao Li, Fu Xiao, Shiguo Wang, Tingrui Pei, and Jie Li. 2018. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with AF-relays. *IEEE Journal on Selected Areas in Communications* 36, 2 (2018), 304–313.

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

[19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *Comput. Commun. Rev.* 38, 2 (2008), 69–74.

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

[21] Viswanath Poosala, Yannis E Ioannidis, et al. 1996. Estimation of query-result distribution and its application in parallel-join load balancing. In *VLDB*, Vol. 96. 3–6.

[22] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*. 1449–1463.

[23] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. 2004. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. 207–212.

[24] Source Code 2021. The source codes of DHS and baseline algorithms. https://github.com/ZeBraHack0/DHS.git.

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

[26] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. HeavyGuardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2584–2593.

[27] 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.

[28] 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.

[29] Shanshan Ying, Flip Korn, Barna Saha, and Divesh Srivastava. 2015. Treescope: finding structural anomalies in semi-structured data. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1904–1907.

[30] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 29–42.

[31] 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*. 741–756.