DaVinci Sketch: A Versatile Sketch for Efficient and Comprehensive Set Measurements

Yanshu Wang[†], Jianan Ji[‡], Chao-Hsuan Liu[§], Hengyang Zhou[¶], Tong Yang[†]
[†]Peking University, China [‡]Carnegie Mellon University, USA [§]East China Normal University, China
[¶]China University of Petroleum, China
{yanshuwang, yangtong}@pku.edu.cn, {jji2}@andrew.cmu.edu, {10215501450}@stu.ecnu.edu.cn, {2021015426}@st.cupk.edu.cn

***Abstract*—Set measurements are fundamental in numerous areas including network measurement, database queries, and data mining. These measurements are typically executed on multisets. Existing algorithms optimize a specific set measurement task, leading to sophisticated but narrowly focused solutions. This specialization often results in inefficiencies when multiple set measurement tasks are required simultaneously, consuming excessive computational and storage resources.**

This paper introduces *DaVinci Sketch*, a versatile sketch designed to efficiently handle various set measurement tasks using a single unified data structure. *DaVinci Sketch* employs a novel approach by utilizing a dedicated structure to store frequent elements, thereby reducing collisions among flows that have the most significant impact on results. Remarkably, *DaVinci Sketch* can simultaneously perform up to nine different measurement tasks with a single data structure and a unified operation, whereas other approaches typically support fewer tasks.

The experimental results demonstrate that *DaVinci Sketch* achieves high accuracy across 9 measurement tasks. Furthermore, in multi-task scenarios, *DaVinci Sketch* significantly reduces the memory usage (by more than 59%) and achieves high throughput (more than 23 times faster than other methods).

***Index Terms*—set measurement, database query, network measurement, sketch algorithm**

I. Introduction

Set measurements play a crucial role in data processing and analysis. Whether it's in database query [1], data mining [2], [3], information retrieval [4], or network measurement [5]–[8] fields, set measurements are an essential foundational tool. In practical applications, sets are categorized into two main types: single sets and multisets. A single set is a collection of distinct elements where order does not matter and duplicates are not allowed. For example, the set {a,b,c} is identical to {c,a,b} because both contain the same elements regardless of the order. On the other hand, a multiset extends the idea of a set by allowing multiple occurrences of its elements, essentially acknowledging the presence of duplicates. In a multiset, while the order of elements still does not matter, the count of each element is crucial. For instance, the multiset {a,a,b,c} is different from {a,b,c} because the former includes two occurrences of "a".

This paper focuses on the measurements of multisets, as single sets can be considered a special case of multisets where the count of each element is limited to one. The measurements applicable to single sets are also applicable to multisets. The set measurements encompass heavy-hitter detection, element frequency measurement, heavy-changer detection, element frequency distribution measurement, entropy measurement, cardinality measurement, difference, union, the cardinality of the intersection, the cardinality of the inner join, and so on.

Previous advancements in set measurement algorithms have been primarily centered on optimizing a specific type of set measurement task, motivated by the demand for the efficiency and precision in specific data processing scenarios. This specialization has led to the development of highly sophisticated algorithms at specific tasks. For instance, JoinSketch [9] focuses on only the cardinality of the inner join. CM Sketch (Count-Min Sketch) [10] is highlighted for element frequency measurement. Elastic Sketch [7] is capable of computing the union across various sets. Heavykeeper [11] emphasizes the measurement of heavy-hitter. In addition, LossRadar [12], FlowRadar [13], and FermatSketch [14] focus on a limited set of measurement tasks.

The trend towards specialization in previous solutions stems from distinct research streams and conflicting optimization goals. Existing work in set operations can be divided into three main streams: the database domain, which addresses tasks like cardinality of inner join, set union, cardinality measurement, and heavy-hitter detection, and the network measurement domain, which focuses on element frequency measurement, heavy-changer detection, element frequency distribution, entropy measurement, and set difference (packet loss detection). Additionally, theoretical research stream targets element frequency measurement. These efforts have not produced unified algorithms for cross-domain problems. In contrast, our approach unifies set operations across multiple domains. Furthermore, each task has conflicting optimization goals, making it difficult to generalize a single sketch that performs well across all measurement types.

Under these circumstances, when various set measurement tasks need to be performed, it is necessary to run multiple algorithms simultaneously, leading to significant overhead. Therefore, designing an algorithm capable of handling multiple set measurements is crucial. However, such algorithm poses significant challenges.

First, **designing an unified data structure to conduct multiple set measurement tasks:** In practical application

Corresponding author: Tong Yang (yangtong@pku.edu.cn).
Yanshu Wang and Tong Yang are with School of Computer Science, Peking University, Beijing, China.

scenarios, multiple tasks are often conducted simultaneously. For example, in the task of network traffic monitoring, it is necessary to measure flow size (frequency) and flow cardinality, detect heavy hitters, analyze flow distribution, and take the union of traffic measurement results when aggregating information from different measurement points. Different set measurement tasks often require different algorithms, and different algorithms typically utilize different data structures. In order to accommodate various tasks while conserving storage and computational resources, it is necessary to employ different algorithms on the same data structure.

Second, **optimizing for element frequency distribution:** In practical applications, the frequency of elements within a dataset often follows a Pareto distribution, where a small proportion of elements accounts for the majority of occurrences. For example, in network traffic, as shown in Figure 1, a small number of large flows [1] dominate the bulk of the traffic, while many flows are minor; in text processing, a few words occur very frequently, while the majority of words appear infrequently. The mixture of frequent and infrequent elements is the key source of estimation errors in set measurement tasks, especially those involving interactions between frequent and infrequent elements. It is crucial to appropriately handle the conflicts between elements.

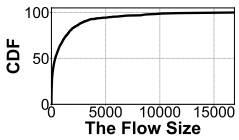


Fig. 1: The distribution of flow size.

To address these two challenges and bridge the gap, we propose *DaVinci Sketch*, a versatile sketch for efficient and comprehensive set measurement tasks. The novelty of *DaVinci Sketch* over existing solutions lies in its unified insertion process and query operations on a shared data structure, which dynamically places elements with varying frequencies into distinct data structures without requiring prior knowledge of their frequency. The guiding principle behind this placement is to use a dedicated structure for frequent elements, thereby reducing collisions among those elements that most significantly impact results. Furthermore, *DaVinci Sketch* extends existing sketch techniques to support querying the cardinality of inner join. Consequently, *DaVinci Sketch* can simultaneously measure up to nine different set measurement tasks using a single data structure.

The contributions of this paper can be summarized as follows:

- **Methodological Innovation:** We propose *DaVinci Sketch* to handle up to nine different set measurement tasks simultaneously.
- **Theoretical Contribution:** We prove the boundedness of the measurement results.
- **Experimental Comparison:** We conduct extensive experiments, demonstrating that *DaVinci Sketch* can achieve up to 59% memory savings and over 23x speed improvements in multi-task scenarios compared to traditional approaches.
- **Unification Across Domains:** Existing work in set measurements can be categorized into the database domain, the network measurement domain, and theoretical research. Our approach unifies the strengths across multiple domains.

[1]Large flows mean the flows with a high frequency of an element.

II. Background

A. Problem Definition

The set measurement tasks we focus on are divided into single-set and multi-set operations.

Let $\mathcal{F}$ and $\mathcal{G}$ be multisets with S and T elements, respectively. We represent a data element in a multiset by e_i, where $\mathcal{D}$ is the domain of all possible elements. Assume $|\mathcal{D}| = N$, such that $\mathcal{D} = \{e_{\beta_1}, \ldots, e_{\beta_i}, \ldots, e_{\beta_N}\}$. Within this context, $\mathcal{F} = [e_1, \ldots, e_i, \ldots, e_S]$ and $\mathcal{G} = [e'_1, \ldots, e'_j, \ldots, e'_T]$, where each element e_i or e'_j belongs to $\mathcal{D}$. Note that the elements in $\mathcal{D}$ are distinct, but those in $\mathcal{F}$ or $\mathcal{G}$ may not be.

For each multiset, we define a frequency vector: $\mathbf{f}$ for $\mathcal{F}$, as $(f_1, \ldots, f_i, \ldots, f_N)$, and $\mathbf{g}$ for $\mathcal{G}$, as $(g_1, \ldots, g_i, \ldots, g_N)$. Here, f_i and g_i denote the frequencies of the element e_{β_i} within $\mathcal{F}$ and $\mathcal{G}$, respectively.

TABLE I: Set Measurement Tasks.

Category	Measurement	Formula
Single-Set	Frequency	$\{(e_i, f_i) \mid e_i \in \mathcal{F}\}$
	Heavy-Hitter	$\{e_i \in \mathcal{F} \mid f_i > \theta\}$ or $\{e_i \mid f_i \in \text{Top } k\}$
	Distribution	Histogram of $\mathbf{f}$
	Entropy	$H(\mathcal{F}) = -\sum_{i=1}^{N} \frac{f_i}{S} \log \frac{f_i}{S}$
	Cardinality	$\lvert\{e_i \mid e_i \in \mathcal{F}\}\rvert$
Multi-Set	Heavy-Changer	$\{e_i \mid \lvert f_i - g_i\rvert > \delta\}$
	Difference	$\mathcal{F} \setminus \mathcal{G} = \{e_i \mid e_i \in \mathcal{F}, e_i \notin \mathcal{G}\}$
	Union	$\mathcal{F} \cup \mathcal{G} = \{e_i \mid e_i \in \mathcal{F} \text{ or } e_i \in \mathcal{G}\}$
	Inner Join	$J = \mathbf{f} \odot \mathbf{g} = \sum_{i=1}^{N} f_i \cdot g_i$

The set operations we support are shown in Table I. Single-set operations include element frequency measurement, which is the number of occurrences of each distinct element within the multiset; heavy-hitter detection, which identifies elements within a multiset whose frequencies exceed a predetermined threshold, θ, or those that are among the most frequent elements; element frequency distribution, which refers to the distribution of frequencies across different elements; entropy measurement, which provides insights into the randomness or predictability of the set; and cardinality measurement, which is the count of distinct elements within $\mathcal{F}$. Multi-set operations include heavy-changer detection, which detects streaming data sets whose frequencies undergo significant changes between two observations; difference and union operations; and the cardinality of the inner join between two sets. All listed queries stem from the element frequency problem. If new operations can be transformed into this framework, additional queries (not listed in Table I) may be supported in the future.

B. Sketch-Based Set Measurement Tasks

Sketches are a class of probabilistic data structures designed to approximate various statistical characteristics of large-scale

datasets efficiently. These algorithms are particularly valuable in big data contexts, where they facilitate the high-speed processing and analysis of voluminous data streams. However, to our knowledge, there is not yet a single sketch algorithm in the literature that covers the majority of measurement tasks. While various sketch data structures provide highly effective approximations for processing large-scale datasets, most of these algorithms are designed for specific statistical properties or types of queries.

There are several lines of related work focused on specific measurement tasks. For element frequency measurement, CM Sketch [10] is the classical algorithm. Count Sketch [15] and CU Sketch [16] improve the accuracy of CM Sketch, and Count Sketch is unbiased, i.e. on average, it produces values that are equal to the true parameter being estimated. Elastic Sketch [7] divided the elements into heavy-part and light-part. This adaptability leads to a higher accuracy in estimating statistical characteristics of skewed data distributions. To adapt to characteristics of skewed data distributions, more algorithms have been proposed, including variable counters size [17]–[20], enlarged count range [21]–[23], multi-level counters [24]–[32], multi-layer sketch [33], [34], frequency-aware updating [35]. A line of previous works focus on heavy-hitter detection [7], [36]–[41]. For the purpose of detecting heavy-hitters, two primary methodologies are utilized, each distinguished by its approach to managing temporal data: one employs exponentially decaying weights [42]–[44], while the other utilizes a sliding window mechanism [41], [45], [46]. These algorithms utilize a min-heap structure for the identification of large flows. Heavy-changer detection is usually derived from the heavy-hitters in the difference set. Element frequency distribution can be calculated by Expectation-Maximization algorithm in count sketch [47]. By leveraging the entropy of traffic distributions, a broad range of network applications can be facilitated, including anomaly detection, clustering to uncover intriguing patterns, and traffic classification [16]. IMP [132] is the inaugural method developed to estimate the entropy between each origin-destination pair. The AMS-estimator [48] presents an algorithm that approximates the empirical entropy of a stream of m values in a single pass. Defeat [49] applies the entropy of the empirical distribution of individual features to identify anomalous traffic patterns. A lines of work for cardinality estimation are also proposed, including MinCount [50], PCSA [51], LogLog [52], HLL [53], Sliding HLL [54], HLL-TailCut+ [55], Refined LL [56] and a sampling-based adaptive cardinality estimation [57]. To detect significant changes between two consecutive time intervals, various methods such as Fast Sketch [58], MV-sketch [59], LD-sketch [60], Modular Hashing [61], Deltoid [62], Reversible sketch [63], and Group testing [62] are used. These methods compute the difference sketch $S_d = |S_2 - S_1|$, where S_1 and S_2 are the sketches recorded for the two consecutive time intervals. Due to the linearity of sketches, the difference between flows can be estimated by querying the result. To calculate the difference and union of two sets, the algorithms encode the element IDs. The LossRadar [12], FlowRadar [13] and FermatSketch [14] ingeniously encode IDs into their data structures, thereby enabling operations such as set intersection and union. This allows for the computation results to be decoded. For cardinality of the inner join, the traditional algorithms include AGMS sketch [64], [65] and Fast-AGMS sketch [66]. They employ the count sketch to represent sets and use the inner product of each row to calculate the cardinality of the inner join. JoinSketch [9] separates frequency and infrequency and achieves a higher accuracy.

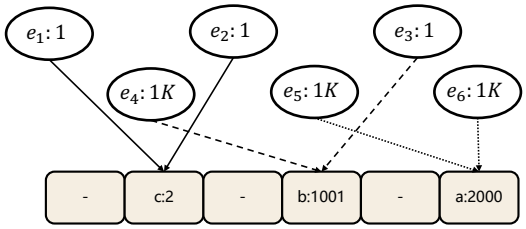


Fig. 2: Hash collisions.

Although previous algorithms can handle large and small flows separately [7], [9], they have not been applied to multiple tasks simultaneously. Implementing multiple tasks simultaneously requires significant resource consumption and space waste.

III. *DaVinci Sketch*

A. *Rationale of DaVinci Sketch*

We first analyze the causes of errors in set measurement tasks, and then introduce the rationale of *DaVinci Sketch* based on these causes.

The hash collisions can be classified into three types: (a) hash collisions between frequent elements, (b) hash collisions between frequent elements and infrequent elements, (c) hash collisions between infrequent elements. Different types of hash collisions impact the estimation errors of set measurement tasks to varying degrees. We will analyze how different types of hash collisions affect the result of set measurement tasks.

Consider a set $\mathcal{F}$ that consists of 6 distinct elements. The frequency vector of $\mathcal{F}$ is

$$\mathbf{f} = (f_1, f_2, \ldots, f_6) = (1, 1, 1, 1000, 1000, 1000),$$

where f_i represents the frequency of the i-th element e_i ($e_i \neq e_j \neq 0$). We mainly consider the error of element frequency estimation and the cardinality of the inner join. The hash collision type is shown in Figure 2. If we calculate $\mathbf{f} \odot \mathbf{f}$, the original correct inner joins are calculated as $1 \times 1 + 1 \times 1 + 1 \times 1 + 1,000 \times 1,000 + 1,000 \times 1,000 + 1,000 \times 1,000 = 3 + 3,000,000 = 3,000,003$. Type (a) collision errors significantly influence the results. As shown in Figure 2, type (a) collisions cause a 50% error in the frequency estimation of frequent elements (e_5, e_6). The impact on the cardinality of the inner joins is particularly significant; elements e_5 and e_6 contribute $2,000 \times 2,000 + 2,000 \times 2,000 = 8,000,000$, compared to the correct contribution of $1,000 \times 1,000 + 1,000 \times 1,000 = 2,000,000$. The error rate is $\frac{8,000,000-2,000,000}{3,000,003} \approx 199.9\%$. Similarly, type (b)

collision errors affect both element frequency estimation and the cardinality of the inner joins. Figure 2 illustrates that type (b) collisions can cause a 1000-fold error in infrequent element estimations and a 0.1% error in frequent element estimations. The impact on the cardinality of the inner join is also substantial; for instance, elements e_3 and e_4 would contribute $1,001\times1,001+1,001\times1,001=2,004,002$, compared to the correct contribution of $1\times1+1,000\times1,000=1,000,001$. The error rate is $\frac{2,004,002-1,000,001}{3,000,003}\approx 33.5\%$. Type (c) collision errors have a minor impact. As depicted in Figure 2, type (c) collisions change the frequency of elements e_1 and e_2 from 1 to 2. This minor change has little effect on the overall estimation error of frequent element estimations (compared to 1000-fold error of e_3). For the cardinality of the inner join, the influence is negligible. Elements e_1 and e_2 contribute $2\times2+2\times2=8$, compared to the correct contribution of $1\times1+1\times1=2$. The error rate is $\frac{8-2}{3,000,003}\approx 0.00019\%$. Given the small impact relative to the contributions from frequent elements, we conclude that type (a) and type (b) collisions significantly influence accuracy, whereas the impact of type (c) collisions is less critical. Therefore, to minimize errors, the algorithm should focus on reducing the occurrence of type (a) and type (b) collisions among frequent and infrequent elements. Therefore, properly handling the collisions between different elements can improve the accuracy of the algorithm. We propose *DaVinci Sketch* to handle such collisions.

The key idea of *DaVinci Sketch* is to distinguish between frequent and infrequent elements. As shown in Figure 3, *DaVinci Sketch* consists of three components: the frequent part, the element filter and the infrequent part. The frequent part is a hash table used to record frequent elements and evict infrequent elements to the element filter. The element filter is a TowerSketch [67] to filter out infrequent elements and insert larger infrequent elements [2] into the infrequent part. This filter improves the algorithm by focusing on elements that have moderate frequency. The infrequent part is a sketch data structure to encode key and frequency of different elements. To encode the keys efficiently, we employ Fermat's Little Theorem[3]. We also designed a cross-validation algorithm for *Fermat Sketch* to improve accuracy. Upon inserting an element, *DaVinci Sketch* prioritizes accumulation in the frequent part. Should a more frequent element arrive and the frequent part reaches its capacity, the least frequent element within it is then evicted to other two parts. This ensures that the frequent part primarily stores the most frequent elements, optimizing the efficiency of set measurement tasks.

This three-part design separates frequent and infrequent elements, ensuring that frequent and infrequent elements do not collide, thus mitigating type (b) collisions. Besides, frequent elements use a hash table to precisely store each element's frequency, alleviating type (a) collisions.

[2]Larger infrequent elements means the infrequent elements that are not filtered by TowerSketch.

[3]Fermat's Little Theorem states that if p is a prime number, then for any integer a such that: $a \not\equiv 0 \pmod p$, it holds that $a^{p-1}\equiv 1 \pmod p$.
Equivalently, this can be written as $a^{p-2}\times a\equiv 1 \pmod p$.

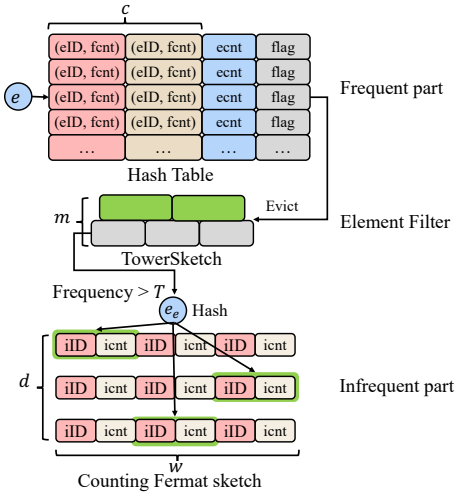


Fig. 3: Data structure of *DaVinci Sketch*.

B. Data Structure and Operations

1) Data Structure: As shown in Figure 3, *DaVinci Sketch* consists of frequent part, element filter and infrequent part.

Frequent part (FP): The frequent part is a hash table of k buckets and is associated with a hash function $H(.)$. The i-th bucket of the frequent part consists of evict counter FP[i].$ecnt$, the evict flag FP[i].$flag$ and c entries. Each entry stores an element ID FP[i][e].eID and frequency FP[i][e].$fcnt$.

Element filter (EF): The element filter is a TowerSketch consisting of m arrays. Each array consists of l_i counters, and is associated with a hash function $g_i(\cdot)$. The size of each counter in array $EF[i]$ is δ_i bits. The key difference between TowerSketch and the CM sketch is in array configuration: lower-level arrays have more, smaller counters, while higher-level arrays have fewer, larger ones, following the principle that set element frequencies are typically skewed, with frequent elements being more numerous and infrequent elements less so.

Infrequent part (IFP): The infrequent part is a counting Fermat sketch. The sketch consists of d arrays. Each array is composed of w buckets and is associated with a hash function $h_i(.)$ and a random function $\zeta_i(.)$. The bucket of the ith row and jth column is made up of a infrequent ID $IFP[i][j].iID$ and a infrequent counter $IFP[i][j].icnt$.

2) Operations: DaVinci Sketch supports three primary operations: inserting an element, performing operations between sets, and querying.

Insertion: When an element e is inserted, *DaVinci Sketch* uses Algorithm 1 for insertion. Additionally, we list the four cases in the algorithm separately (case1-case4).

Case 1: (lines 2–4 in algorithm 1) If bucket $FP[i]$ contains element e, the counter of element $fcnt$ will be increased by 1.

Case 2: (lines 5–8 in algorithm 1) If bucket $FP[i]$ does not contain element e but there exists at least one empty entry, we insert element e into an empty entry of bucket $FP[i]$.

Case 3: (lines 9–16 in algorithm 1) If bucket $FP[i]$ does not contain element e and it is full, we increase the evict counter

Algorithm 1: The insertion algorithm of frequent part.

Require: The element e.
1 $i \leftarrow H(e)$;
2 **if** $e \in FP[i]$ **then**
3 $FP[i][e].fcnt \leftarrow FP[i][e].fcnt + 1$;
4 **return**;
5 **else if** $FP[i]$ *is not full* **then**
6 $FP[i][e].fcnt \leftarrow 1$;
7 $FP[i][e].eid \leftarrow e$;
8 **return**;
9 **else if** $FP[i]$ *is full* **then**
10 $FP[i].ecnt \leftarrow FP[i].ecnt + 1$;
11 **if** $FP[i].ecnt > \lambda \times FP[i][e].fcnt$ **then**
12 insert $(FP[i][e].eid, FP[i][e].fcnt)$ to EF;
13 $FP[i][e].eid \leftarrow e$;
14 $FP[i][e].fcnt \leftarrow 1$;
15 $FP[i].flag \leftarrow$ True;
16 **return**;
17 **else**
18 insert $(e, 1)$ to EF;
19 **return**;

Algorithm 2: The insertion algorithm of infrequent part.

Input: The element ID e, the element frequency cnt.
1 **for** $i = 1$ **to** d **do**
2 $j \leftarrow h_i(e)$;
3 $IFP[i][j].iID \leftarrow (IFP[i][j].iID + cnt \times e) \mod p$;
4 $IFP[i][j].icnt \leftarrow IFP[i][j].icnt + \zeta_i(e) \times cnt$;

$FP[i].ecnt$. If $FP[i].ecnt$ is larger than $\lambda \times FP[i][e].fcnt$, element $FP[i][e]$ is classified as an infrequent element, and we evict $FP[i][e]$ and insert $(FP[i][e].eID, FP[i][e].fcnt)$ into the element filter.

Case 4: (lines 17–19 in algorithm 1) If bucket $FP[i]$ does not contain element e and it is full, we increment the eviction counter $FP[i].ecnt$. If $FP[i].ecnt$ is no larger than $\lambda \times FP[i][e].fcnt$, element e is believed to be an infrequent element, and we inserted $(e, 1)$ into the element filter.

To insert an element e into element filter, we first calculate the m hash functions to locate m counters: $EF_1[g_1(e)], EF_2[g_2(e)], \ldots, EF_m[g_l(e)]$. We call these counters the m mapped counters. Then, for each of the m mapped counters, we update it by the element frequency unless it is overflowed. To query the frequency of element e in the element filter, we simply report the minimum value among the m mapped counters. For an element e, the element filter processes it as follows. First, we insert it into the element filter and query the frequency of element e. Then, with the queried element frequency, we decide whether to insert element e into infrequent part according to a threshold T. If the queried result is larger the threshold T, we insert the e to the infrequent part. In other conditions, e is left in the element filter.

The insertion process of infrequent part is shown in algorithm 2. To insert the element into the infrequent part, we calculate the hash value of each array in infrequent part (lines 2–4 in algorithm 2). In each hashed bucket, we encode the element id and element frequency, respectively (lines 3–4 in algorithm 2). When we encounter non-numerical elements, there are two cases depending on the key length. For the case where the key is fixed-length and relatively short, we directly use the existing approach, as binary encoding in memory can be interpreted as numerical values, making non-numerical elements able to undergo numerical computation. Algorithm 2 supports non-numerical items through hash functions and modular arithmetic applied to their binary representation. For the case where the key is variable-length and relatively long (for example, exceeding the numerical representation range), we first hash the key into a fixed-length fingerprint. The result of the hash function can then be used for relevant numerical computations. A separate mapping between the fingerprint and the original key is maintained for query.

Performing operations between sets: Operations between sets including union and difference of two sets. Each set is represented by a *DaVinci Sketch* data structure. The result of operations also is a *DaVinci Sketch* data structure.

Algorithm 3: The union of the frequent part and infrequent part.

Input: The first set S_1, the second set S_2.
1 **for** $i = 1$ **to** k **do**
2 **foreach** $e \in FP_{S_1}[i][.].eID \cap FP_{S_2}[i][.].eID$ **do**
3 $TS.eID \leftarrow e$;
4 $TS.fcnt \leftarrow FP_{S_1}[i][e].fcnt + FP_{S_2}[i][e].fcnt$;
5 $topc \leftarrow$ getTopcEntry($TS \cup FP_{S_1 \setminus TS}[i] \cup FP_{S_2 \setminus TS}[i]$);
6 $FP[i]_{S_r}$.add($topc$);
7 $evictele \leftarrow FP_{S_1 \setminus topc}[i] \cup FP_{S_2 \setminus topc}[i]$;
8 **if** $evictele$ *is not empty* **then**
9 evict $evictele$ to IFP;
10 $f \leftarrow$ True;
11 $FP_{S_r}[i].flag \leftarrow FP_{S_1}[i].flag$ or $FP_{S_2}[i].flag$ or f;
12 **for** $i = 1$ **to** d **do**
13 **for** $j = 1$ **to** w **do**
14 $j \leftarrow h_i(e)$;
15 $IFP_{S_r}[i][j].iID \leftarrow (IFP_{S_1}[i][j].iID + IFP_{S_2}[i][j].iID) \mod p$;
16 $IFP_{S_r}[i][j].icnt \leftarrow (IFP_{S_1}[i][j].icnt + IFP_{S_2}[i][j].icnt) \mod p$;
17 **return** S_r;

Union: The *DaVinci Sketch* data structure is distinctively divided into the frequent part, the element filter and the infrequent part. The process for calculating the union result considers these three parts independently. For element filter, we can obtain the result by summing each row and column of two tower sketch correspondingly. It is worth noting that, in order to prevent overflow, only half of the value range of the element can be used when performing the union operation. Then, we calculate the results of the frequent part (lines 1–11 in algorithm 3) and the infrequent part (lines 12–17 in algorithm 3). It is noteworthy that there is a difference between merging measurements after using the union operation (denoted as union version) and directly using a single measurement structure to measure the entire set (denoted as original version) when analyzing the allocation of elements. For example, an element may be in the frequent part of the union version but not in the frequent part of the

original version. We conducted experiments on CAIDA data to measure the distribution of frequent elements identified by both the original version and union version (where we consider the top α elements to be frequent elements, and α is the capacity of the frequent part). The results showed that the F1 score for the original version was 0.73, while the F1 score for the union version was 0.77. We also analyzed the proportion of frequent elements that were not included in the frequent part. For the original version, this proportion was 0.26, and for the union version, it was 0.22. Since the union version utilizes twice the space before merging, the frequent elements are more likely to remain in the frequent part before the merge, and the frequent part before merging is more likely to stay in the frequent part after merging. Therefore, the union version captures the frequent elements more accurately.

Difference: When calculating the difference set $A - B$ between sets A and B, previous approaches [12]–[14] primarily addressed the case where A contains B (notated as $B \subset A$). We also have extended this approach to accommodate scenarios where A and B do not mutually contain each other. For given sets ($A = \{a, a, b, d\}$) and ($B = \{a, b, b, c\}$), the operation yields ($A-B = \{a, -b, d, -c\}$), where: The positive sign ($+$) indicates elements that are present in set (A) but not in set (B). The negative sign ($-$) indicates elements that are present in set (B) but not in set (A). In line with this methodology, we describe set difference algorithm. The element filter of the result can be calculated by subtracting each row and column of the two sketch data structures correspondingly. The algorithm for computing the difference between the frequent part and the infrequent part of two sets is similar to the union algorithm, except that addition is replaced with subtraction.

Querying: We support query operations for element frequency measurement, heavy-hitter detection, cardinality measurement, element frequency distribution, entropy measurement, heavy-changer detection and cardinality of inner join. Additionally, set union and set difference are operations involving two sets, and queries can be performed on the resulting sets after these operations. The element frequency measurement is described in Algorithm 4. The heavy-hitter detection algorithm simply adds a threshold check based on the frequency measurement. The cardinality measurement algorithm calculates the cardinality for different parts of the *DaVinci Sketch* and then combines the results from these parts. The cardinality in the frequency part is obtained directly, while the linear counting algorithm [68] is applied to compute the cardinality in other parts. Duplicates are then removed based on the flags in the frequency part. The frequency distribution is estimated by decoding the frequency results and counting how many elements correspond to each frequency, iteratively refined using the EM algorithm [47]. The entropy measurement is calculated by applying the frequency results to the entropy formula, and heavy-changer detection is performed by subtracting the *DaVinci Sketch* of two time windows and conducting heavy-hitter detection on the resulting sketch. The query of the cardinality of the inner join is described in the following text of Section III-B2. Since heavy hitters, heavy changers, as well as the measurement of element frequency distribution, entropy, and cardinality, are all based on querying element frequency, the cardinality of the inner join is calculated separately. In the following sections, we primarily focus on querying element frequency and the cardinality of the inner join.

Algorithm 4: The element frequency querying algorithm.

```
   Data: The element e.
1  i ← H(e);
2  if e ∈ FP[i] and FP[i].flag is False then
3  |  cnt ← FP[i][e].fcnt;
4  |  return cnt;
5  else if e ∈ FP[i] and FP[i].flag is True then
6  |  cnt ← FP[i][e].fcnt;
7  else if e ∉ FP[i] then
8  |  cnt = 0;
9  elemap ← Decode();
10 if elemap[e] is not empty then
11 |  return elemap[e] + cnt + T;
12 else
13 |  Initiate counter array eccounters;
14 |  for i ← 1 to m do
15 |  |  eccounters[i] ← EF[i][g_i(e)];
16 |  if minimum(eccounters) ≥ T then
17 |  |  Initiate counter array ifpcounters;
18 |  |  for i ← 1 to d do
19 |  |  |  ifpcounters[i] ← IFP[i][h_i(e)];
20 |  |  return cnt + minimum(ifpcounters) + T;
21 |  else
22 |  |  return minimum(eccounters);
```

Element frequency: Algorithm 4 outlines the procedure to query the frequency of an element e. The query process is consisted of two steps, i.e. querying the frequent part and the infrequent part. The algorithm begins with hashing the element to determine its location i within the frequent part of the data structure. If e is found in $FP[i]$ and the flag at this location is $False$, indicating no need for further querying, the frequency cnt of e is directly returned (lines 2–4 in algorithm 4). If e is present in $FP[i]$ but with a $True$ flag, or if e is not found in $FP[i]$, additional steps are taken to accurately compute its frequency (lines 5–8 in algorithm 4).

For elements not directly found or requiring further querying, the algorithm initiates a decoding process through the decode function to identify any additional contributions to the element's frequency from the infrequent part of the data structure. If the process decodes the frequency of e, the frequency is added to cnt (Lines 9–12 in algorithm 4). If decoding process does not reveal the frequency of e, the algorithm employs a minimum computation of selected counters from element filter and a median computation of infrequent part. If the query result from element filter is larger than the threshold, the algorithm returns the sum of the frequency count plus T $cnt + T$, otherwise, return the result from element filter (lines 13–22 in algorithm 4).

Algorithm 5 describes the procedure for decoding the frequencies of elements within the infrequent part. The decode

Algorithm 5: The decode algorithm.

1 **Function** `canDecode`(i, j)**:**
2 $e \leftarrow IFP[i][j].iID \times IFP[i][j].icnt^{(p-2)} \mod p$;
3 **return** $(j == h_i(e)$ **and** $Query_{EF}(e) \geq T)$ **or** $(j == h_i(p-e)$ **and** $Query_{EF}(p-e) \geq T)$;
4 **Function** `Remove`$(IFP[i][j], IFP[i'][j'])$**:**
5 $FP[i][j].icnt \leftarrow FP[i][j].icnt - IFP[i'][j'].icnt$;
6 $FP[i][j].iID \leftarrow FP[i][j].iID - IFP[i'][j'].iID \mod p$;
7 **Function** `Decode`(IFP)**:**
8 Initiate empty queue $queue$;
9 Initiate empty map $elemap$;
10 **for** $i = 1$ **to** d **do**
11 **for** $j = 1$ **to** w **do**
12 **if** $IFP[i][j]$ *is not empty* **then**
13 $queue$.enqueue($IFP[i][j]$);
14 **while** $queue$ *is not empty* **do**
15 $IFP[i][j] \leftarrow queue$.dequeue();
16 **if** *canQuery*(i, j) **then**
17 $e \leftarrow IFP[i][j].iID \times IFP[i][j].icnt^{(p-2)} \mod p$;
18 **if** $j == h_i(e)$ **then**
19 $elemap[e] = elemap[e] + IFP[i][j].icnt$;
20 $rmkey = e$;
21 **else**
22 $elemap[e] = elemap[e] - IFP[i][j].icnt$;
23 $rmkey = p - e$;
24 **for** $i' = 1$ **to** d **do**
25 Remove$(IFP[i'][h_{i'}(rmkey)], IFP[i][j])$;
26 **if** $IFP[i'][h_{i'}(rmkey)] \neq 0$ **then**
27 $queue$.enqueue($IFP[i'][h_{i'}(rmkey)]$);
28 **return** $elemap$;

algorithm is crucial for accurately determining element frequencies that are not directly stored in the frequent part.

The canDecode function (lines 1–3 in algorithm 5) calculates an element's decoded value, subsequently determining its validity based on predefined hash functions. According to Fermat's Little Theorem, e equals $IFP[i][j].iID \times IFP[i][j].icnt^{(p-2)} \mod p$. If the decoded id e can be re-hashed to the same position, indicating the element is correctly decoded. In addition, because the element in infrequent part is evicted from the element filter, which has the threshold T, the frequency in element filter is used for cross-validation. Such **double verification** is intended to ensure a higher decoding rate. Note that we validate both e and $p - e$ for the set difference operation. This is essential because any negative equivalent, such as $-e$, will be adjusted to $p - e$ by the modular operation, ensuring all values within the appropriate range defined by the modulus p. This careful consideration prevents errors in the identification process by accounting for the cyclic nature of modular arithmetic.

The remove function adjusts the infrequent part structure (lines 4–6 in algorithm 5). The adjustments are made to the count ($icnt$) and identifier (iID) of elements.

The decode algorithm is shown as lines 7–28 in algorithm 5. Initially, it initiates a queue with all non-empty elements in infrequent part. Subsequently, through an iterative process involving the canDecode and Remove functions, the algorithm updates an element map ($elemap$), accruing decoded elements alongside their corresponding counts. At last the element counter map $elemap$ is returned.

The cardinality of the inner join: To calculate the cardinality of the inner join, we decompose the cardinality of the inner join into multiple parts and calculate the cardinality of the inner join for each part separately. We define the frequent vector as $\mathbf{f}_F = (f_{F1}, f_{F2}, \ldots, f_{FN})$, $i = 1, \ldots, N$. This vector represents frequencies of elements in the frequent part. The infrequent vector is denoted as $\mathbf{f}_I = (f_{I1}, f_{I2}, \ldots, f_{IN})$, capturing frequencies of elements recorded in the infrequent part. The filter vector is denoted as $\mathbf{f}_E = (f_{E1}, f_{E2}, \ldots, f_{EN})$.

The composition of these vectors $\mathbf{f}_F$, $\mathbf{f}_I$ and $\mathbf{f}_E$ encapsulate the partial frequency vectors for elements recorded in their respective parts. The aggregate frequency vector $\mathbf{f}$ is a sum of the frequent, infrequent, and element filter, expressed as $\mathbf{f} = \mathbf{f}_F + \mathbf{f}_I + \mathbf{f}_E$. Assuming another dataset's frequency vectors are $\mathbf{g}_F$, $\mathbf{g}_I$, and $\mathbf{g}_E$, the cardinality of the inner join of $\mathbf{F}$, and $\mathbf{G}$, denoted by $\mathbf{J}$, simplifies the calculation of join operations as follows:

$$
\begin{aligned}
\mathbf{J} &= \mathbf{f} \odot \mathbf{g} \\
&= (\mathbf{f}_F + \mathbf{f}_I + \mathbf{f}_E) \odot (\mathbf{g}_F + \mathbf{g}_I + \mathbf{g}_E) \\
&= \mathbf{f}_F \odot \mathbf{g}_F + \mathbf{f}_F \odot \mathbf{g}_I + \mathbf{f}_F \odot \mathbf{g}_E \\
&\quad + \mathbf{f}_I \odot \mathbf{g}_F + \mathbf{f}_I \odot \mathbf{g}_I + \mathbf{f}_I \odot \mathbf{g}_E \\
&\quad + \mathbf{f}_E \odot \mathbf{g}_F + \mathbf{f}_E \odot \mathbf{g}_I + \mathbf{f}_E \odot \mathbf{g}_E \\
&= \mathbf{J}_{FF} + \mathbf{J}_{FI} + \mathbf{J}_{FE} + \mathbf{J}_{IF} + \mathbf{J}_{II} + \mathbf{J}_{IE} + \mathbf{J}_{EF} + \mathbf{J}_{EI} + \mathbf{J}_{EE}.
\end{aligned}
$$

This formula highlights the method to compute the contributions of the interactions between the frequent, infrequent, and element filter parts across two sets (FP_{S_1} and FP_{S_2}). The join result, denoted as $\mathbf{J}$, is dissected into nine principal components: $\mathbf{J}_{FF}$, $\mathbf{J}_{FI}$, $\mathbf{J}_{FE}$, $\mathbf{J}_{IF}$, $\mathbf{J}_{II}$, $\mathbf{J}_{IE}$, $\mathbf{J}_{EF}$, $\mathbf{J}_{EI}$, and $\mathbf{J}_{EE}$. These components elucidate the interactions between the frequent, infrequent, and element filter parts of each set. Specifically: $\mathbf{J}_{FF}$ is calculated by iterating through each element e located in the intersection of the frequent parts of both sets, multiplying their frequency counts $FP_{S_1}[H(e)][e].fcnt \times FP_{S_2}[H(e)][e].fcnt$, and accumulating this product to the join result. $\mathbf{J}_{FE}$, $\mathbf{J}_{EF}$, $\mathbf{J}_{FI}$ and $\mathbf{J}_{IF}$ capture the cardinality of the inner join stemming from the interactions between the frequent part of one set and the infrequent part or element filter of the other. For $\mathbf{J}_{FI}$, each element e in the frequent part of S_1 and S_1. Because we can obtain element key from the frequent part, the frequency of other part can be directly queried. The result can be obtained by multiplying the two values. For $\mathbf{J}_{IE}$ and $\mathbf{J}_{EI}$, we use the same hash function for the row of the infrequent part and the element filter, and the number of buckets are either equal or multiples of each other. When calculating the cardinality of the inner join, we fold the array with the larger number of buckets, and then perform multiplication and addition at corresponding positions. For $\mathbf{J}_{II}$, and $\mathbf{J}_{EE}$, we can obtain the result by performing the dot product at corresponding positions and then calculating the average across different rows.

3) Analysis of Time Complexity: Let P_{FP} denote the probability that element stays in the frequent part, P_{EF} the prob-

ability it stays in the element filter, and P_{IFP} the probability it stays in the infrequent part.

The frequent part is a multi-bucket hash table, where the access complexity is $c+2$ (accessing c buckets + one access to $ecnt$ + one access to $flag$). The element filter is an m-layer TowerSketch, with an access complexity of m. The infrequent part is a Fermat sketch, with the complexity of inserting and fast querying (using a count-sketch style query in Counting Fermat sketch) being d, and the full decoding complexity is wd^2 [14]. Thus, the time complexity for inserting and fast querying an element is given by $O(c+m+d)$ as follows: $P_{FP}\times(c+2)+P_{EF}\times(c+2+m)+P_{IFP}\times(c+2+m+d) = (P_{FP}+P_{EF}+P_{IFP})(c+2)+(P_{EF}+P_{IFP})m+P_{IFP}d$.

For better visualization of the actual complexity in practice, we conducted tests with $d=3$, $m=2$, and $c=7$, and as shown in Figure 8a, the average memory access (time complexity) of our algorithm was 6.68, which is significantly lower than that of the compared algorithm, which was 29.47. Additionally, if full decoding is required, the time complexity becomes $O(c+m+wd^2)$.

IV. Theoretical Analysis

Since all the tasks are based on the measurement of frequencies, we will derive the error bound related to frequencies, and other errors are similar. The frequency of element in *DaVinci Sketch* consists of three parts, frequent part, element filter and infrequent part, i.e. $f = f_{FP}+f_{EF}+f_{IFP}$, where f represents the total frequency of an element, f_{FP} represents the frequent part of the element, f_{EF} represents the element filter, and f_{IFP} represents the infrequent part of the element.

f_{FP} part is a hash table which accurately records the element frequency. The infrequent part accurately estimates the decoded element frequency. And for undecoded element frequency of infrequent part and all element frequency of the element filter, *DaVinci Sketch* gives rough estimates of element frequency. From this perspective, the frequency can also be expressed as a composition of the precise part and the rough estimates of element filter and infrequent part. Thus, the frequency estimation of each element can be refined as:

$$f = f_{FP} + f_{IFP}^{Precise} + f_{IFP}^{Rough} + f_{EF}. \tag{1}$$

To better complete the proof, we first consider a one-dimensional basic structure, namely an counter array A of length R, is associated with a hash function $\theta(.)$ and a random function $\phi(.)$. When an element e arrives, the corresponding counter is updated as $A[\theta(e)] \leftarrow A[\theta(e)] + \phi(e)\times 1$. When querying, *DaVinci Sketch* returns $A[\theta(e)]$.

Lemma 1. *The frequency estimation for element e is f_e, then f_e is unbiased, i.e. $E(\widehat{f_e}) = f_e$.*

Proof.

$$\begin{aligned}\widehat{f_e} &= \sum_j \left(\mathbb{I}_{\theta(j)=\theta(e)}\cdot f_j\cdot\phi(j)\right)\cdot\phi(e)\\ &= \left(f_e\cdot\phi(e)^2\right) + \sum\left(\mathbb{I}^{j\neq e}_{h(j)=h(e)}\cdot f_j\cdot\phi(j)\cdot\phi(e)\right).\end{aligned} \tag{2}$$

The indicator function, denoted as $\mathbb{I}$, is a function that evaluates a given condition. It returns a value of 1 when the condition is true and 0 when the condition is false. Notice that, $\phi(e)=0$ as the expected value of an even distribution is the sum of the values divided by the number of values, which in this case is 0. In addition, $E(\phi(e)^2)=1$ as $\phi(e)^2\to\{1\}$. Knowing these expectations, we wish to calculate $E(\widehat{f_e})$, the expectation of the estimator for f_e:

$$\begin{aligned}E(\widehat{f_e}) = &\left(f_e\cdot E\left(\phi(e)^2\right)\right) +\\ &\sum_j\left(E\left(\mathbb{I}^{j\neq e}_{h(j)=h(e)}\right)\cdot f_j\cdot E\left(\phi(j)\right)\cdot E\left(\phi(e)\right)\right).\end{aligned} \tag{3}$$

By knowing $\phi(e)=0$, we simplify the equation to:

$$E(\widehat{f_e}) = f_e\cdot E\left(\phi(e)^2\right). \tag{4}$$

Since $E\left(\phi(e)^2\right)=1$, our final result becomes:

$$E(\widehat{f_e}) = f_e. \tag{5}$$

□

Lemma 2. *The variance of the frequency estimation $A[\cdot]$ is $\frac{||A||_2^2}{R}$.*

Proof. Consider the variance of the estimated frequency $\widehat{f_e}$:

$$\mathrm{Var}(\widehat{f_e}) = E\left[\left(\widehat{f_e}-f_e\right)^2\right] \tag{6}$$

$$= E\left[\left(\sum_j \mathbb{I}^{h(j)=h(e)}_{j\neq e}\cdot f_j\cdot\phi(j)\cdot\phi(e)\right)^2\right] \tag{7}$$

$$= E\left[\sum_{j\neq e}\mathbb{I}_j^2\cdot f_j^2\cdot\phi(j)^2\cdot\phi(e)^2\right] \tag{8}$$

$$+ E\left[\sum_{\substack{j\neq e\\ k\neq e}}\mathbb{I}_j\mathbb{I}_k\cdot f_j\cdot f_k\cdot\phi(j)\phi(k)\cdot\phi(e)^2\right]. \tag{9}$$

By knowing that, $E\left(\mathbb{I}^{j\neq e}_{h(j)=h(e)}\right)=\frac{1}{R}$, $\phi(e)=0$ and $E\left(\phi(e)^2\right)=1$, the variance simplifies to:

$$\mathrm{Var}(\widehat{f_e}) = E\left[\sum_{j\neq e}\mathbb{I}_j^2\cdot f_j^2\right] = \frac{1}{R}\sum_{j\neq i} f_j^2 = \frac{||F||_2^2}{R}, \tag{10}$$

where $||F||_2^2$ represents the L^2-norm squared of the frequency vector f, excluding the frequency of element e itself. □

Lemma 3. *Given an error tolerance probability of $\frac{1}{k}$, the error bound for frequency estimation of the basic structure is given by:*

$$\Pr\left(\left|\widehat{f_e}-f_e\right| > \sqrt{\frac{k}{R}}||F||_2\right) < \frac{1}{k}. \tag{11}$$

Proof. We use Chebyshev's inequality to prove this lemma:

$$\Pr\left(|X-E(X)| > \sqrt{k}\sigma\right) < \frac{1}{k}. \tag{12}$$

And according to Lemma 2, where $\sigma=\sqrt{\mathrm{Var}(\widehat{f_e})}$, substituting into Chebyshev's inequality yields:

$$\Pr\left(\left|\widehat{f_e}-f_e\right| > \sqrt{\frac{k}{R}}||F||_2\right) < \frac{1}{k}. \tag{13}$$

□

A. Error Analysis of DaVinci Sketch

Theorem 1. *Given an error tolerance probability of $\frac{1}{k}$, the error bound for frequency estimation of the DaVinci Sketch is given by:*

$$f - error_1 \leq \hat{f} \leq f + error_1 + \frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1, \tag{14}$$

$$error_1 = \sqrt{\frac{k}{R_{IFP}}}||F_{IFP}||_2, \tag{15}$$

where R_{EF} is the length of selected array in element filter and R_{IFP} is the length of count Fermat sketch. F_{EF} and F_{IFP} are the elements frequency in the element filter and infrequent part.

Proof. To calculate the error bound of frequency estimated by *DaVinci Sketch*, we decompose the error bound of frequency estimation according to Equation 1. Then, We calculate the error bound of each component and sum up the results. Because the frequency estimations for $\widehat{f_{FP}}$ and $\widehat{f_{IFP}^{Precise}}$ are accurate, they are error-free.

$$Error[\hat{f}] = Error[\widehat{f_{IFP}^{Rough}}] + Error[\widehat{f_{EF}}]. \tag{16}$$

The frequency of infrequent part is queried by d basic structures and our algorithm returns the median of each queried results. According to Equation 13, the error bond of infrequent part is:

$$f_{IFP}^{Rough} - error_1 \leq \widehat{f_{IFP}^{Rough}} \leq f_{IFP}^{Rough} + error_1. \tag{17}$$

For element filter which uses count min update, we use the error bound in [67]. Let $\delta_0 = 0$. Note that $\delta_0 \leq \delta_1 \leq \cdots \leq \delta_m$. Given an arbitrary element e_j, without loss of generality, we assume its real frequency f_j satisfies $2^{\delta_{t-1}-1} \leq f_j \leq 2^{\delta_t-1}$, where $1 \leq t \leq m$. Let n_e be the number of elements and f_s be the sum of the frequency sizes of all elements, i.e., $f_s = \|F_{EF}\|_1 = \sum_{j=1}^{n_e} f_j$. Given an arbitrary small positive number ϵ, when $f_j + \epsilon \cdot f_s < 2^{\delta_t-1}$, the estimation error of element e_j is bounded by $\Pr\{\hat{f}_j \leq f_j + \epsilon \cdot f_s\} > 1 - \prod_{i=t}^{d} \frac{1}{\varepsilon \cdot w_i}$. Let $k = \prod_{i=t}^{d} \varepsilon \cdot w_i$ and we can get $\varepsilon = \frac{k}{\prod_{i=t}^{d} w_i}$.

$$0 \leq \widehat{f_{EF}^{Rough}} \leq f_{EF}^{Rough} + \frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1. \tag{18}$$

For *DaVinci Sketch*:

$$f - error_1 \leq \hat{f} \leq f + error_1 + \frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1. \tag{19}$$

□

Theorem 2. *Given an error tolerance probability of $\frac{1}{k}$, the biasedness of frequency estimated is bounded by $\frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1$.*

Proof. According to theorem 1:

$$f - error_1 \leq \hat{f} \leq f + error_1 + \frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1. \tag{20}$$

The biasedness of the estimated frequency is bounded by $\frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1$. The element filter employs small counters and large number of buckets, i.e. w_i is large. And element filter keeps infrequent element, i.e. $\|F_{EF}\|_1$ is small. Hence, the unbiased bound $\frac{k}{\prod_{i=t}^{d} w_i}\|F_{EF}\|_1$ is very small. □

V. Experimental Results

In this section, we first introduce the experimental setup, then use micro-benchmarks to test the accuracy of *DaVinci Sketch* across different set operations. Finally, in the Overall Performance section, we comprehensively test the performance and throughput of *DaVinci Sketch* when handling multiple overall performance tasks simultaneously.

Dataset: We conduct the evaluation using three real-world data traces:

(1) **CAIDA** [69] is an anonymized IP trace stream collected in 2019 from the Center for Applied Internet Data Analysis (CAIDA). This dataset provides valuable insights into Internet traffic, enabling researchers to study network performance, security, and usage patterns.

(2) **MAWI** [70] serves as a comprehensive database designed to aid researchers in evaluating traffic anomaly detection methods. This resource provides a collection of labels identifying traffic anomalies within the MAWI archive.

(3) **TPC-DS** [71] is widely recognized as the industry standard benchmark for evaluating the performance of decision support solutions, including big data systems. It captures various essential elements of a decision support system, such as query processing and data maintenance tasks.

Table II summarizes the characteristics of these datasets, including the number of packets, the number of flows, and cardinality.

TABLE II: Dataset statistics

Datasets	# of packets	# of flows	cardinality
CAIDA	2,472,727	109,642	109,642
MAWI	2,000,000	200,471	200,471
TPC-DS	4,903,874	1,834	1,834

Platform and implementation: Our evaluations are conducted on a single server equipped with an Intel(R) Core(TM) i9-10980XE CPU, operating at a base frequency of 3.00 GHz, capable of reaching up to 4.80 GHz under maximum load. This CPU comprises 18 cores and 36 threads, supported by a substantial 24.8MB L3 cache and 18MB of L2 cache, ensuring high performance and responsiveness. All algorithms are implemented in C++ and compiled using g++ version 7.5.0 on Ubuntu, optimized with the -O3 compiler flag for maximum efficiency. The hash functions utilized are Bob Hash, known for its speed and effectiveness in data handling. The full implementation of *DaVinci Sketch* has been open-sourced for community use and further development [72].

Metrics: The evaluation of the algorithm involves various metrics, each serving a unique purpose in assessing performance.

Average Relative Error (ARE) is defined as $\text{ARE} = \frac{1}{|\Omega|}\sum_{f_i \in \Omega}\frac{|v_i - \hat{v}_i|}{v_i}$, where Ω represents the set of all elements, v_i the true size of element e_i, and $\hat{v}_i$ its estimated frequency. In contrast, Average Absolute Error (AAE) can be defined as $\text{AAE} = \frac{1}{|\Omega|}\sum_{f_i \in \Omega}|v_i - \hat{v}_i|$.

F_1 Score calculates as F_1 Score $= \frac{2 \cdot PR \cdot RR}{PR + RR}$, with PR (Precision Rate) indicating the ratio of correctly reported instances to all reported instances, and RR (Recall Rate) being the ratio of correctly reported instances to all correct instances.

Relative Error (RE) is determined by RE $= \frac{|Tru-Est|}{Tru}$, contrasting the true and estimated statistics, Tru and Est, respectively.

Weighted Mean Relative Error (WMRE) is articulated as WMRE $= \frac{\sum_{i=1}^{z}|n_i-\hat{n}_i|}{\sum_{i=1}^{z}\left(\frac{n_i+\hat{n}_i}{2}\right)}$, with z marking the maximum element frequency and n_i, $\hat{n}_i$ the true and estimated numbers of frequency of element i.

Throughput (Mpps) denotes the processing speed, measured in million packets per second.

Setup: We compare our algorithm with fifteen algorithms in ten tasks: Hashpipe, ElasticSketch, CocoSketch, FCM-sketch, CM, CU, CountHeap [73], UnivMon, MRAC, FlowRadar, LossRadar, Fermat, JoinSketch, F-AGMS and SkimmedSketch. For heavy-hitter detection and heavy-changer detection, we set their thresholds Δ_h and Δ_c to about 0.02% and 0.01% of the total packets respectively. The configurations of these competitors are recommended in the literature.

A. Micro-benchmarks

In this section, we primarily conduct experiments to compare the performance of *DaVinci Sketch* in various set measurement tasks against other algorithms. The experimental results for the CAIDA, MAWI, and TPC-DS datasets are shown in Figure 4, Figure 5, and Figure 6, respectively.

Element frequency estimation: For CAIDA dataset, as shown in Figure 6a, *DaVinci Sketch* demonstrates a higher accuracy than other algorithms. Utilizing only 200KB of memory, the Average Relative Error (ARE) of *DaVinci Sketch* is significantly smaller, being 23.1 times, 15.2 times, 7.4 times, and 9.9 times smaller than that of CM, CU, Elastic, and FCM, respectively. The result of MAWI dataset is similar, and the TPC-DS dataset shows instability of results due to the small number of flows. We also conducted experiments on AAE, but due to space limitations, we have only included the AAE for element frequency estimation, as shown in Figure 7c. The results show that the AAE performance of *DaVinci Sketch* is also better than existing algorithms in most cases.

Heavy-hitter Detection: For CAIDA dataset, as shown in Figure 6b, experimental results indicate that *DaVinci Sketch* achieves comparable accuracy with HashPipe and elastic sketch. Additional, *DaVinci Sketch* has a higher accuracy than other algorithms. With only 200KB of memory, the F1 score of *DaVinci Sketch* is 99.7%, while that of Coco and UniMon is lower than 99%. The result of MAWI dataset is similar, and the TPC-DS dataset also shows instability in the results due to the small number of flows.

Heavy-changer Detection: For CAIDA dataset, as shown in Figure 6c, the *DaVinci Sketch* achieves a higher accuracy than other algorithms in detecting heavy changes. With just 200KB of memory, *DaVinci Sketch* achieves a 100% F1 score, surpassing the performance of other algorithms which remain below 97.0%. The results of MAWI and TPC-DS datasets are similar.

Cardinality Estimation: For cardinality estimation of CAIDA dataset, as shown in Figure 6d, *DaVinci Sketch* significantly outperforms other algorithms. With 200KB of memory, the Relative Error (RE) of *DaVinci Sketch* is much lower, being 0.00021, 109.9 times, 262.3 times, and 361.3 times smaller than UnivMon, Elastic, and FCM, respectively. The results of MAWI and TPC-DS datasets are similar.

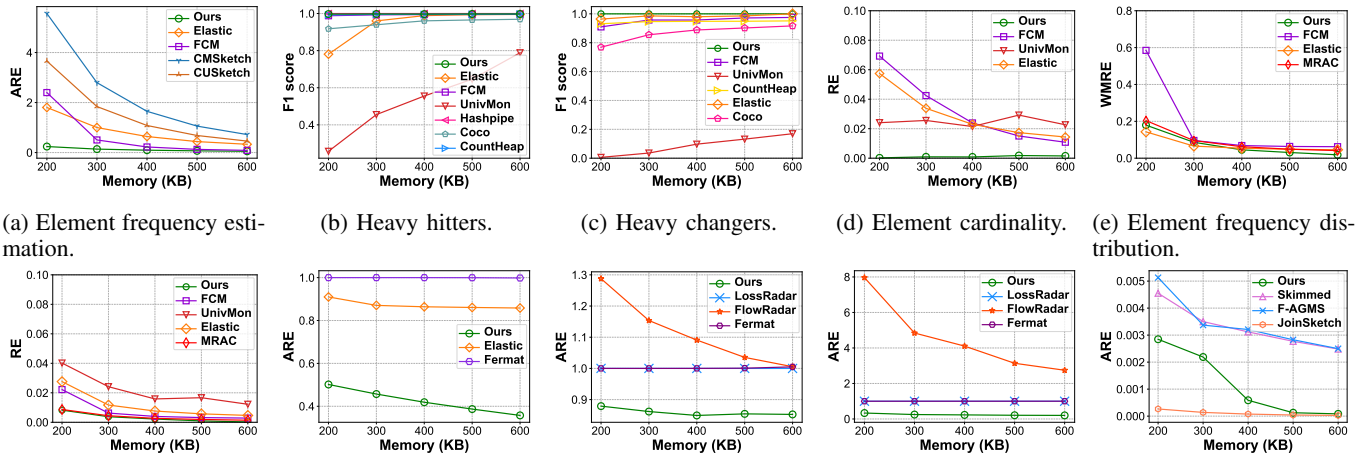
Element frequency distribution For CAIDA dataset, as shown in Figure 6e, *DaVinci Sketch* achieves comparable accuracy with Elastic sketch and MRAC. Besides, *DaVinci Sketch* outperforms FCM. When using 600KB of memory, the Weighted Mean Relative Error (WMRE) of *DaVinci Sketch* is notably lower, being 0.018, 2.4 times, 3.4 times and 2.2 times smaller than Elastic, FCM and MRAC, respectively. The results of TPC-DS dataset are similar. And for MAWI datasets, *DaVinci Sketch* is comparable to the optimal algorithm.

Entropy Estimation: For CAIDA dataset, as shown in Figure 6f, *DaVinci Sketch* shows a higher accuracy than other algorithm in entropy estimation. With 600KB of memory, the ARE of *DaVinci Sketch* is an impressive 0.000229417, 53.2, 20.3, 12.4, 6.1 times smaller than that of UnivMon, Elastic, FCM and MRAC. The results of TPC-DS dataset are similar. And in the MAWI datasets, *DaVinci Sketch* is comparable to the optimal algorithm.

Union: To experimentally evaluate the union operation of two sets, we first compute the union and then calculate the frequency to assess the union operation. For CAIDA dataset, as shown in Figure 6g, *DaVinci Sketch* demonstrates a higher accuracy than other algorithms. Utilizing only 200KB of memory, the Average Relative Error (ARE) of *DaVinci Sketch* is significantly smaller, being 1.8 times and 1.9 times smaller than that of Elastic and Fermat, respectively. The results of MAWI and TPC-DS datasets are similar.

Difference: To evaluate the difference operation, we first compute the difference of two sets and then evaluate the accuracy of frequency measurement. We test two scenarios: subtracting half of the set from the whole set (inclusion difference, Figure 6j) and subtracting the last two-thirds from the first two-thirds (overlap difference, Figure 6h). The *DaVinci Sketch* demonstrates superior accuracy compared to other algorithms in handling inclusion differences. With just 200KB of memory, the Average Relative Error (ARE) of *DaVinci Sketch* is significantly reduced, being 23.8 times smaller than FlowRadar, 2.9 times smaller than LossRadar, 2.9 times smaller than Fermat. The results of MAWI and TPC-DS datasets are similar. Similarly, for overlap differences, the *DaVinci Sketch* also exhibits a higher accuracy than comparative algorithms. Utilizing the same memory allocation of 200KB, the ARE of *DaVinci Sketch* is substantially lower, amounting to 1.4 times less than FlowRadar, 1.1 times less than LossRadar, and 1.1 times less than Fermat. And for the results of MAWI and TPC-DS datasets, *DaVinci Sketch* also has the best performance.

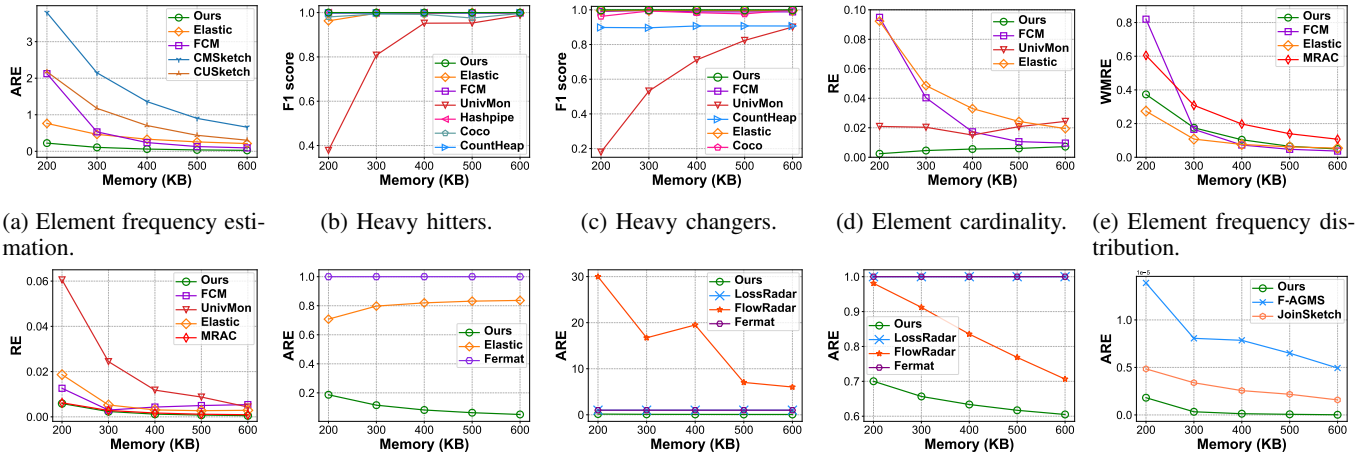
The cardinality of the inner join: For CAIDA dataset, as shown in Figure 6i, *DaVinci Sketch* shows comparable accuracy with JoinSketch and outperforms Skimmed and F-AGMS in calculate the cardinality of the inner join. With 600KB of memory, the ARE of *DaVinci Sketch* is 29.7, 29.8 times smaller than that of Skimmed, Elastic and F-AGMS.



(a) Element frequency estimation. (b) Heavy hitters. (c) Heavy changers. (d) Element cardinality. (e) Element frequency distribution.

(f) Entropy. (g) Union of two sets. (h) Difference of two sets (overlap difference). (i) Difference of two sets (inclusion difference). (j) The cardinality of the inner join.

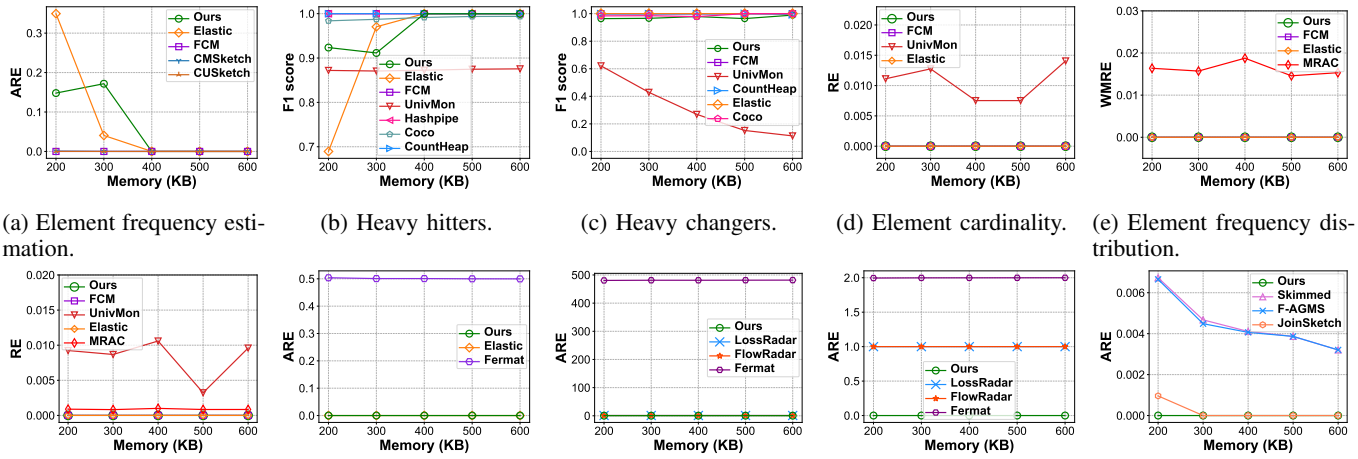
Fig. 4: Experimental results of CAIDA dataset.



(a) Element frequency estimation. (b) Heavy hitters. (c) Heavy changers. (d) Element cardinality. (e) Element frequency distribution.

(f) Entropy. (g) Union of two sets. (h) Difference of two sets (overlap difference). (i) Difference of two sets (inclusion difference). (j) The cardinality of the inner join.

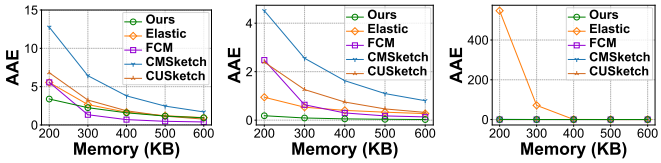
Fig. 5: Experimental results of MAWI dataset.



(a) Element frequency estimation. (b) Heavy hitters. (c) Heavy changers. (d) Element cardinality. (e) Element frequency distribution.

(f) Entropy. (g) Union of two sets. (h) Difference of two sets (overlap difference). (i) Difference of two sets (inclusion difference). (j) The cardinality of the inner join.

Fig. 6: Experimental results of TPC-DS dataset.



(a) CAIDA dataset. (b) MAWI dataset. (c) TPC-DS dataset.

Fig. 7: AAE of different datasets.

And for the results of MAWI and TPC-DS datasets, *DaVinci Sketch* has the best performance.

B. Overall Performance

In the evaluation of overall performance, we employ the *DaVinci Sketch* to execute a multitude of measurement tasks. We compare its results with those of a combination of state-of-the-art algorithms for each measurement task, which achieve comparable or lower [4] accuracy for the same tasks. We refer to the algorithm that combines multiple algorithms to achieve the same functions and accuracy as the Composite Set Operations Algorithm (CSOA). The accuracy of different tasks is shown in Table III. We tried different comparison algorithms and finally selected three algorithms based on a comprehensive consideration of the number of algorithms (choosing as few comparison algorithms as possible) and the accuracy of the algorithms (choosing comparison algorithms with as high accuracy as possible). The CSOA consists of three algorithms: **FCM** for element frequency, heavy-hitter detection, heavy-changer detection, cardinality measurement, element frequency distribution, and entropy measurement. **Fermat Sketch** for set difference and union operations. **JoinSketch** for the cardinality of the inner join.

We primarily analyze the average memory access, throughput and the memory savings of *DaVinci Sketch* while achieving the same accuracy for different set operations. We use the CAIDA dataset in the experiment of overall performance.

Average Memory Access: The experimental results of Average Memory Access (AMA) [5] are shown in Figure 8a. As the memory increases, the number of memory accesses for both *DaVinci Sketch* and CSOA decreases because more elements are stored in the frequent part and CSOA also features a structure similar to the frequent part in some of its algorithms. On average, the number of memory accesses for *DaVinci Sketch* is 22.60% of that for the CSOA algorithm.

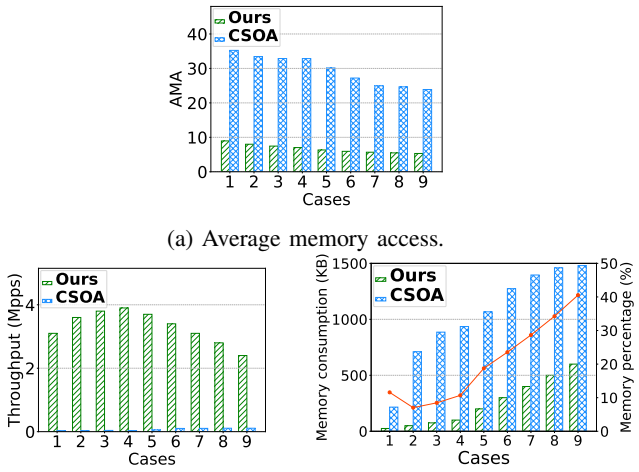
TABLE III: Accuracy under different cases.

	Frequency	HH	HC	Card	Distribution	Entropy	Union	Difference	Inner join
Case 1	2.7	0.93	0.84	0.0043	0.58	0.054	0.85	1.4	0.31
Case 2	1.2	0.98	0.95	0.0036	0.33	0.031	0.73	1.1	0.27
Case 3	0.67	0.99	0.97	0.0022	0.26	0.022	0.66	0.96	0.56
Case 4	0.45	0.99	0.98	0.0034	0.23	0.017	0.62	0.88	0.33
Case 5	0.16	1.00	1.00	0.0059	0.12	0.0068	0.50	0.77	0.044
Case 6	0.081	0.99	1.00	0.0082	0.063	0.0033	0.42	0.71	0.035
Case 7	0.046	1.00	1.00	0.012	0.034	0.0017	0.36	0.68	0.018
Case 8	0.027	1.00	1.00	0.015	0.017	0.00080	0.30	0.66	0.0083
Case 9	0.017	1.00	1.00	0.017	0.010	0.00040	0.26	0.64	0.0054

[4]Some algorithms cannot achieve our level of accuracy.

[5]The AMA is defined as the total number of memory accesses divided by the total number of insertions.

Throughput Analysis: As shown in Figure 8b, we can see that *DaVinci Sketch* demonstrates higher throughput than CSOA. In case 4, the throughput of *DaVinci Sketch* is significantly higher, being 3.9 (Mpps), 111.9 times higher than that of CSOA. In case 9, which is our worst case, the throughput of *DaVinci Sketch* is 2.4, being 22.1 times higher than that of CSOA.



(a) Average memory access.

(b) Throughput. (c) Memory consumption.

Fig. 8: Experimental results of overall performance.

Memory Savings: We evaluate the memory consumption and memory percentage [6] of each algorithm across different cases. *DaVinci Sketch* consistently shows superior memory savings. As illustrated in Figure 8c, *DaVinci Sketch* outperforms CSOA in terms of reduced memory consumption. In case 2, the memory savings of *DaVinci Sketch* are particularly significant, reducing 660 KB of memory, which is 7.03% of CSOA' memory. In case 9, which is our worst case, the memory used by *DaVinci Sketch* is 40.54% of CSOA' memory.

VI. Conclusion

In this paper, we present *DaVinci Sketch*, which can integrate multiple set measurement tasks within a unified framework. In addition, *DaVinci Sketch* can simultaneously perform up to nine different measurement tasks with a single data structure and a unified operation. We have fully implemented a *DaVinci Sketch* prototype. Experimental results verify that 1) *DaVinci Sketch* can achieve high accuracy in 9 set measurement tasks; 2) DaVinci Sketch can save more memory (by more than 59%) in multi-task scenarios and achieve high throughput (more than 23 times faster than other methods).

Acknowledgment

This work was supported in part by the National Key R&D Program of China (No. 2024YFB2906603), the National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009), research grant No. SH-2024JK29, and High Performance Computing Platform of Peking University.

[6]The memory percentage is calculated as the ratio of *DaVinci Sketch*'s memory consumption to that of CSOA.

REFERENCES

[1] Y. Park, B. Mozafari, J. Sorenson, and J. Wang, "Verdictdb: Universalizing approximate query processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1461–1476.

[2] I. Nunes, M. Heddes, P. Vergés, D. Abraham, A. Veidenbaum, A. Nicolau, and T. Givargis, "Dothash: Estimating set similarity metrics for link prediction and document deduplication," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 1758–1769.

[3] X. Wan and X. Han, "Efficient top-k frequent itemset mining on massive data," *Data Science and Engineering*, pp. 1–27, 2024.

[4] J. S. Culpepper and A. Moffat, "Compact set representation for information retrieval," in *International Symposium on String Processing and Information Retrieval*. Springer, 2007, pp. 137–148.

[5] B. Chen, Z. Lv, X. Yu, and Y. Liu, "Sliding window top-k monitoring over distributed data streams," *Data Science and Engineering*, vol. 2, pp. 289–300, 2017.

[6] L. Yuan, C.-N. Chuah, and P. Mohapatra, "Progme: towards programmable network measurement," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007, pp. 97–108.

[7] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[8] Y. Wang, D. Li, Y. Lu, J. Wu, H. Shao, and Y. Wang, "Elixir: A high-performance and low-cost approach to managing hardware/software hybrid flow tables considering flow burstiness," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 535–550.

[9] F. Wang, Q. Chen, Y. Li, T. Yang, Y. Tu, L. Yu, and B. Cui, "Joinsketch: A sketch algorithm for accurate and unbiased inner-product estimation," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–26, 2023.

[10] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[11] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[12] Y. Li, R. Miao, C. Kim, and M. Yu, "Lossradar: Fast detection of lost packets in data center networks," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016, pp. 481–495.

[13] L. Yuliang, M. Rui, K. Changhoon, and Y. Minlan, "Flowradar: A better netflow for data centers," in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, 2016, pp. 311–324.

[14] K. Yang, Y. Wu, R. Miao, T. Yang, Z. Liu, Z. Xu, R. Qiu, Y. Zhao, H. Lv, Z. Ji *et al.*, "Chamelemon: Shifting measurement attention as network state changes," *arXiv preprint arXiv:2301.00615*, 2023.

[15] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[16] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 1, pp. 145–156, 2006.

[17] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 241–252.

[18] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic count filters," *Acm Sigmod Record*, vol. 35, no. 1, pp. 26–32, 2006.

[19] G. Einziger and R. Friedman, "Counting with tinytable: Every bit counts!" in *Proceedings of the 17th International Conference on Distributed Computing and Networking*, 2016, pp. 1–10.

[20] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proceedings of the 2017 ACM international conference on Management of Data*, 2017, pp. 775–787.

[21] R. Stanojevic, "Small active counters," in *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 2007, pp. 2153–2161.

[22] J. Qi, W. Li, T. Yang, D. Li, and H. Li, "Cuckoo counter: A novel framework for accurate per-flow frequency estimation in network measurement," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 2019, pp. 1–7.

[23] T. Yang, J. Xu, X. Liu, P. Liu, L. Wang, J. Bi, and X. Li, "A generic technique for sketches to adapt to different counting ranges," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2017–2025.

[24] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 121–132, 2008.

[25] N. Hua, B. Lin, J. Xu, and H. Zhao, "Brick: A novel exact active statistics counter architecture," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 89–98.

[26] D. Nyang and D. Shin, "Recyclable counter with confinement for real-time per-flow measurement," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 3191–3203, 2016.

[27] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[28] M. Chen, S. Chen, and Z. Cai, "Counter tree: A scalable counter architecture for per-flow traffic measurement," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1249–1262, 2016.

[29] Y. Zhou, P. Liu, H. Jin, T. Yang, S. Dang, and X. Li, "One memory access sketch: a more accurate and faster sketch for per-flow measurement," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–6.

[30] J. Gong, T. Yang, Y. Zhou, D. Yang, S. Chen, B. Cui, and X. Li, "Abc: a practicable sketch framework for non-uniform multisets," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2380–2389.

[31] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 741–756.

[32] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li, "Diamond sketch: Accurate per-flow measurement for real ip streams," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 1–2.

[33] Y. Zhou, H. Jin, P. Liu, H. Zhang, T. Yang, and X. Li, "Accurate per-flow measurement with bloom sketch," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 1–2.

[34] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1449–1463.

[35] J. Bruck, J. Gao, and A. Jiang, "Weighted bloom filter," in *2006 IEEE International Symposium on Information Theory*. IEEE, 2006, pp. 2304–2308.

[36] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[37] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 113–126.

[38] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 2584–2593.

[39] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 2019, pp. 334–350.

[40] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. 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, 2020, pp. 1574–1584.

[41] X. Gou, L. He, Y. Zhang, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, "Sliding sketches: A framework using time zones for data stream processing in sliding windows," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1015–1025.

[42] A. Shrivastava, A. C. Konig, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1417–1432.

[43] S. Matusevych, A. Smola, and A. Ahmed, "Hokusai-sketching streams in real time," *arXiv preprint arXiv:1210.4891*, 2012.

[44] G. Cormode, F. Korn, and S. Tirthapura, "Exponentially decayed aggregates on data streams," in *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 1379–1381.

[45] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[46] F. Xing, F. Zhang, X. Tian, W. Li, H. Chen, and H. Jin, "Identifying the most recent heavy hitters in large-scale streams using block-wise counting," in *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2017, pp. 239–244.

[47] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, pp. 177–188, 2004.

[48] A. Chakrabarti, G. Cormode, and A. McGregor, "A near-optimal algorithm for computing the entropy of a stream," in *SODA*, vol. 7. Citeseer, 2007, pp. 328–335.

[49] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina, "Detection and identification of network anomalies using sketch subspaces," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006, pp. 147–152.

[50] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Randomization and Approximation Techniques in Computer Science: 6th International Workshop, RANDOM 2002 Cambridge, MA, USA, September 13–15, 2002 Proceedings 5*. Springer, 2002, pp. 1–10.

[51] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[52] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 2003, pp. 605–617.

[53] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, 2013, pp. 683–692.

[54] Y. Chabchoub and G. Heébrail, "Sliding hyperloglog: Estimating cardinality in a data stream over a sliding window," in *2010 IEEE International Conference on Data Mining Workshops*. IEEE, 2010, pp. 1297–1303.

[55] Q. Xiao, Y. Zhou, and S. Chen, "Better with fewer bits: Improving the performance of cardinality estimation of large data streams," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[56] L. Wang, T. Yang, H. Wang, J. Jiang, Z. Cai, B. Cui, and X. Li, "Fine-grained probability counting for cardinality estimation of data streams," *World Wide Web*, vol. 22, pp. 2065–2081, 2019.

[57] R. Cohen and Y. Nezri, "Cardinality estimation in a virtualized network device using online machine learning," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 2098–2110, 2019.

[58] Y. Liu, W. Chen, and Y. Guan, "A fast sketch for aggregate queries over high-speed network traffic," in *2012 Proceedings IEEE INFOCOM*. IEEE, 2012, pp. 2741–2745.

[59] L. Tang, Q. Huang, and P. P. Lee, "Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2026–2034.

[60] Q. Huang and P. P. Lee, "A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams," *Computer Networks*, vol. 91, pp. 298–315, 2015.

[61] M. Kallitsis, S. A. Stoev, S. Bhattacharya, and G. Michailidis, "Amon: An open source architecture for online monitoring, statistical analysis, and forensics of multi-gigabit streams," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1834–1848, 2016.

[62] G. Cormode and S. Muthukrishnan, "What's hot and what's not: tracking most frequent items dynamically," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 249–278, 2005.

[63] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004, pp. 207–212.

[64] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 20–29.

[65] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy, "Tracking join and self-join sizes in limited storage," in *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 1999, pp. 10–20.

[66] G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 13–24.

[67] K. Yang, S. Long, Q. Shi, Y. Li, Z. Liu, Y. Wu, T. Yang, and Z. Jia, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[68] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, pp. 208–229, 1990.

[69] CAIDA, "The caida ucsd anonymized internet traces 2018," https://www.caida.org/catalog/datasets/passive_dataset/, 2018.

[70] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking," in *ACM CoNEXT '10*, Philadelphia, PA, December 2010.

[71] TPC, "tpc-ds," in *https://www.tpc.org/tpcds/*, 2024.

[72] "Davinci sketch," https://github.com/DaVinciSketch/DaVinci-Sketch, 2024.

[73] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.