CodingSketch: A Hierarchical Sketch with Efficient Encoding and Recursive Decoding

Qizhi Chen†, Yisen Hong†, Yihan Wu†, Tong Yang†, Bin Cui†

†*National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China*

***Abstract*—Sketch is a probabilistic data structure widely used in various fields due to its high accuracy under small memory. Designing hierarchical data structures for real-world datasets with high skewness is one of the main optimization directions of Sketch. However, there is still a big accuracy gap between the existing sketches and the optimum. To fill the gap, we propose a new sketch called Coding Sketch. For the first time, we used both hierarchical structure and nearly-lossless encoding-and-decoding to compress frequent items, which significantly improves the accuracy of frequent items. Besides, we propose flagless pruning to remove the additional flag bits in traditional hierarchical structure. Thus Coding Sketch can optimize the frequency estimation of both frequent and infrequent items. Our evaluation shows that our algorithm is 10 times more accurate than the state-of-the-art under the same memory cost. All related codes are open-sourced.[1]**

***Index Terms*—data stream, sketches, decoding**

I. INTRODUCTION

In many scenarios involving the processing of large-scale data, such as databases, data mining, and network management [11], [14], [18], [20], [27], [28], [40], [48], [58], the handling of data streams is a common requirement. *Sketch*, as an accurate, fast, and space-efficient probabilistic data structure, finds extensive utility in various data stream tasks, including frequency estimation [9], [16], [17], [23], finding top-k frequent flows [33], [35], [41], [45], [51], [59], join size estimation [5], [6], [15], [27], [38], and more [13], [19], [29]–[31], [47], [56].

Classic sketches like Count-Min [CM] [17], Conservative Update [CU] [23], and Count [9] efficiently manage data streams within a compact memory space by utilizing shared counters, albeit at the cost of determinism. These techniques encounter special properties when dealing with real-world data that often exhibits **highly skewed distributions** , *most flows are infrequent while only a few flows are frequent* [7], [12], [55], [56]. Classic sketches employ counters of uniform size, which can lead to shortcomings. When the counter size is too small, it can overflow due to the presence of a few frequent flows. Conversely, allocating large-size counters for most infrequent flows results in a wasteful use of memory resources. Space efficiency is a paramount attribute of sketches, and any form of memory wastage is unacceptable. Consequently, optimizing sketches for highly skewed data distributions has emerged as a hot research direction in recent years. Based on the structures, existing efforts in this area can be categorized into two main classes: *flat structure* sketches [7], [53] and *hierarchical structure* sketches [32], [49], [50], [55].

First, the *flat structure* sketches adapt the individual counter structure to accommodate the frequency distribution. For instance, Self-Adaptive Counters (SAC) [53] actively compress counters when their values are large to use smaller counters for recording the frequency of frequent flows. In the case of Self-Adjusting Lean Streaming Analytics (SALSA) [7], overflowing counters are merged with their neighbors. While these methods are friendly to infrequent flows due to their numerous small counters, they pose challenges for frequent flows. They employ probabilistic increment (SAC) or merging methods (SALSA) to approximate the frequency of frequent flows.

Second, *hierarchical structure* sketches segregate frequent and infrequent flows. Initially, all flows are assigned small memory-sized counters, and during the insertion process, counters for frequent flows overflow into a secondary structure. Although existing hierarchical structure sketches have made significant progress, they still have notable shortcomings. For instance, both Pyramid Sketch [55] and Stingy Sketch [32] employ binary trees as data structures. While tree structures perform well in highly skewed distributions, they inevitably lead to hash collisions in the upper layers, resulting in unacceptable exponential errors for frequent streams.

In summary, existing methods all lead to additional errors for frequent flows, including counter merging, probabilistic increment, and upper-layer sharing leading to conflicts. While this may have a minor impact on calculating average errors, in many tasks, such as heavy hitter and heavy change, inaccuracies in the frequency estimation of frequent flows can result in highly undesirable outcomes. Therefore, we hope to devise a sketch that can provide more accurate estimates for frequent flows while saving memory resources.

In this paper, we propose a new sketching framework called **Coding Sketch**, which achieves an order of magnitude improvement in accuracy compared to the state-of-the-art method while maintaining comparable insertion speed. During the item insertion process, we employ an efficient *encoding* algorithm to construct a hierarchical data structure. Before addressing queries, we employ a recursive *decoding*

[1] https://github.com/CodingSketch/Coding-Sketch

algorithm to transform the hierarchical structure into a flat one. Leveraging these innovations, our algorithm is capable of **completely resolving** high-level hash collisions, leading to a significant improvement in accuracy. Next, we delve into the two key techniques that underlie Coding Sketch.

The **first key technique** involves *bottom-up encoding* and *top-down decoding*, which, when combined, effectively address the hash collisions occurring in the upper layers—collisions that have hitherto constrained the accuracy of existing hierarchical structure sketches. In tree structures [32], [55], such collisions are inevitable because, upon overflow, each counter merely points to its parent, with the parent node aggregating the values of multiple children. Consequently, during queries, child nodes cannot differentiate errors from their siblings in the parent node. Coding Sketch, however, employs multiple locations to encode overflow values. As long as one of these locations remains free from collisions, the decoding algorithm can accurately retrieve the overflow value of the counter. We rigorously prove in Section § III that, given a counter count surpassing a certain lower bound, the decoding algorithm can recover all overflows *with high probability*. To elaborate, when inserting items, we initiate the process from the lowest layer of the data structure. Whenever a counter overflows, it accumulates in multiple positions in the upper layer, based on its location. These upper-layer counters may store multiple overflowed sums, which can be naturally encoded recursively into higher layers. This encoding strategy effectively exploits the redundancy inherent in high-level structures, enabling these sums to be re-separated into overflow values during decoding. In contrast to encoding, the decoding algorithm scans each layer recursively from top to bottom. By constantly looking for the pure pointer we define, we can finally get the true overflow value. We believe that the cost of recursive scanning is acceptable, especially in practice, where query and analysis tasks are often performed post-insertion of all items. In such typical scenarios, data stream processing requires the necessity of real-time insertion, but real-time query support is not required, and ample resources are available for executing the queries [4], [8], [22], [24], [27], [33], [34], [36], [42], [43], [45], [49], [51], [52], [54], [57]. We will discuss the reasonableness of this setup in related work § II.

The **second key technique** is the **Flagless Pruning** (§ III-C). Following the integration of flagless pruning, Coding Sketch no longer necessitates flag bits, resulting in substantial space savings. Flag bits constitute a fundamental component of existing hierarchical structure sketches. They are indispensable for the counters in the lower layers to discern if they have experienced overflow. However, we realize that setting flag bits is an extravagant approach. In Coding Sketch, each counter comprises a mere 2 to 4 bits. In such instances, flag bits account for up to $\frac{1}{3}$ of the total space consumption. Indeed, this wastage is a contributing factor preventing some existing sketches from achieving finer layering. Flagless pruning encompasses both explicit and implicit scenarios. In explicit pruning, when a counter in the higher layer is found to be 0, it guarantees that none of the lower-layer counters pointing to it overflow. A counter that doesn't overflow may yield a false positive, implying that none of the higher-level counters it points to are 0. Implicit flagless pruning, when combined with the decoding algorithm, effectively addresses this issue. Flagless pruning can save the space overhead of Coding Sketch as a whole, making the frequency estimation of all flows more accurate.

We hereby further propose two optimized versions of Coding Sketch. The first is the **unbiased version** (§ III-D2). After changing the counter to signed and modifying the encoding and decoding, we can extend Coding Sketch to the Count sketch to get an unbiased frequency estimation algorithm. For the second version, § III-D1, we introduce **automatic memory adjustment** of each layer. In the absence of prior knowledge of the data distribution, § III-D1 can optimize parameters to acquire desirable performance automatically. All related codes are open-sourced [2].

Key Contributions:

- We propose Coding Sketch, a new sketch framework with encoding and decoding that solving the problem of high-level hash collisions that hampers the existing hierarchical structure sketches to be friendly to frequent flows.
- We design a series of optimization techniques to ensure the high performance of Coding Sketch. Most importantly, by excluding the reliance on flag bits, Coding Sketch can achieve finer layering, ensuring high performances in both frequent flows and infrequent flows.
- We theoretically analyze the success rate of Coding Sketch decoding, and give a strict lower bound on memory.
- We conduct simulation experiments on typical tasks of frequency estimation and heavy-hitter estimation. The results show that Coding Sketch outperforms the state-of-the-art algorithm in accuracy and memory usage.

II. Background and Related Work

In this section, we first commence by providing a foundational understanding of frequency estimation in § II-A. Following that, we introduce some existing sketch algorithms in § II-B. Subsequently, we delve into some algorithms employed in other domains that utilize encoding and decoding strategies akin to those in Coding Sketch in § II-C. Finally, we illustrate the acceptability of decoding by presenting various applications of sketches in § II-D.

A. Preliminaries

Data Stream: A data stream $\mathcal{S}$ is a sequence $\{e_1, e_2, e_3, ..., e_N\}$ $(e_i \in E)$ of N items arriving in order, where E is the id set. These N items can have duplicate items. Algorithms running on data streams must be one-pass.

Frequency Estimation: Given a data stream $\mathcal{S} = \{e_1, e_2, e_3, ..., e_N\}$. We call the subset of items with the same id in $\mathcal{S}$ a flow. To ask the frequency of a flow e is to ask how often e occurs in $\mathcal{S}$, $f(e) = \sum_i \mathcal{I}\{e = e_i\}$. An algorithm $\hat{f}$ that supports frequency estimation means that $\hat{f}(e)(\forall e \in E)$

is an estimate of $f(e)$. Note that e is not known until the entire $\mathcal{S}$ is processed. So algorithm $\hat{f}$ should support the estimation of all flows in E during the process.

B. Related Sketches

Flat Structure Sketches: Self-adaptive Counters (SAC) [53] and Self-Adjusting Lean Streaming Analytics (SALSA) [7] are representatives of flat structure sketches. SAC employs a floating-point-like encoding technique and updates with a certain probability, but its probabilistic update method leads to larger errors for frequent flows. SALSA is an enhancement of another flat structure sketch known as the ABC sketch [25]. In SALSA, each counter starts with a small number of bits and merges with adjacent counters upon overflow, resulting in larger errors as overflows accumulate. SALSA also requires flag bits into the process.

Hierarchical Structure Sketches: Pyramid [55] and Stingy [32] are representatives of hierarchical structure sketches. Both utilize tree-like structures for counters, which can introduce errors due to counter sharing at higher layers. Moreover, Pyramid is significantly affected by flag bits, with $3/4$ of its counter states becoming sentinels. Stingy employs unique optimizations with counter automata but does not fundamentally eliminate flag bits. However, as hierarchical structures, they maintain higher accuracy compared to classical sketches, even when considering these errors.

Sketches Filtering Frequent Flows: Represented by Elastic is a series of sketches that separate frequent and infrequent items [51], [52], [54]. They divide the data structure into heavy part and light part to store frequent and infrequent items respectively. This is similar to the idea of our algorithm. The essence of their algorithms lies in identifying frequent items to support various tasks, such as Top-k detection. However, identifying these frequent items requires recording their IDs, which means the memory overhead of these algorithms is related to the length of the IDs. We compared the representative Elastic Sketch alongside our algorithm in the experimental section, where we specifically analyzed the differences in the properties of these two types of algorithms.

C. Related Work of Encoding and Decoding

PR-Sketch [39]: The PR-Sketch decodes keys of streaming data by establishing linear equations. But there are still errors in the least squares solution it finds, and the rationale is different from our decoding algorithm. It encodes the key instead of the frequency, so it cannot build a hierarchical structure to save space. And the goal of the PR-Sketch is high cover proportion rather than frequency estimation. It cannot be compared with our algorithm.

Bloomier Filter [10] and Invertible Bloom Lookup Tables [26]: Their algorithm's final proof method can be transformed into the process of finding the 2-core of a random hypergraph (e.g., see [21], [37]). In comparison to our algorithm, only the mathematical transformation is similar, while the actual tasks performed are entirely different. The Bloomier Filter constructs a static table rather than being oriented towards data stream scenarios. The Invertible Bloom Lookup Tables algorithm stores hashes and equivalent additional information to support restoration. In contrast, our algorithm is an optimization of the Sketch algorithm, requiring no extra information; in fact, it can even eliminate the need for the originally required flag bits.

Table I: Symbols used in this paper.

Notation	Meaning
e	An item in the data stream
L_i	The ith layer of Coding Sketch
$h_i(e)$	The ith hash function of item e, there are a total of d hash functions that map the item to d different positions in L_0
w_i	The number of counters for the array in L_i
B_i	The number of bits per counter in L_i
mx_i	$(mx_i = 2^{B_i} - 1)$ The maximum number that can be recorded by the counter in L_i
f_i	The flag bit array in L_i
$\mathcal{A}_i$	The counter array in L_i
$\widetilde{\mathcal{A}}_i$	The counter array in L_i after decoding, which contains $\mathcal{A}_i$ and the overflowed part
$H_{ij}(k)$	The jth hash function mapping a position k in L_i into a position in L_{i+1}. There are 3 hash functions for each layer i
$\mathcal{F}_i$	An array of sets of pointer, where each set comprises all pointers pointing to this location, $\mathcal{F}_i[k] = \{k', \exists_j H_{i-1j}(k') = k \mid \forall_{k' \in L_{i-1}}\}$

D. Related Work of Offline Query

SketchINT [49]: SketchINT represents a series of applications in data stream monitoring and anomaly detection using sketch algorithms [4], [22], [34], [42], [43], [49]. They emphasize anomaly patterns in data rather than real-time querying. Taking SketchINT as an example, data is collected periodically, followed by batch anomaly detection. Data collection occurs at resource-constrained edge computing nodes, necessitating efficient resource management. Data analysis, on the other hand, takes place on resource-rich central servers, aligning well with the nature of Coding Sketch.

Compass [27]: Compass represents a series of applications in batch data processing using sketch algorithms [8], [24], [27], [57]. In large-scale data processing, there is often a need to first collect a substantial volume of data and then perform batch analysis. In such scenarios, efficient data collection and storage are often more critical than real-time querying.

III. The Coding Sketch Algorithm

In this section, we present the bottom-up encoding and top-down decoding of Coding Sketch in §III-A and §III-B, respectively. Building upon this foundation, we introduce the final version of Coding Sketch in §III-C, which eliminates the need for flag bits, conserves a significant amount of memory, and supports deletion operations. Additionally, we have devised several other versions of Coding Sketch to extend its functionality, making full use of its remarkable extensibility. These alternative versions will be individually introduced in §III-D. Symbols used in this paper are listed in Table I.

A. The Fast and Efficient Encoding

Overview: When the frequency distribution exhibits high skewness, the counter of the classic sketch record the frequency of both frequent items and infrequent items. To accommodate both cases, the counter size must be sufficient

to store the frequency of the frequent flows. However, this approach results in significant memory wastage for counters that exclusively store infrequent flows. To mitigate this inefficiency, we can employ counters with fewer bits. Nevertheless, this choice may lead to overflow for frequent flows. To address this issue, we store the overflowed portions of counter values in a new hierarchical structure. Within this new layer, we can recursively apply the aforementioned approach to conserve memory until no overflows persist. This constitutes the fundamental principle underpinning memory-efficient hierarchical structure sketch algorithms. Due to the recursive structure, our attention is consistently directed toward only two adjacent layers. Subsequently, we will refer to the lower of these two layers as the "**current layer**" and the higher one as the "**upper layer**". Coding Sketch maintains multiple pointers between each layer, which seems simple, but this alone does not achieve significant optimization. Only through carefully designing the form of the pointers and the proportion of the layers, coupled with the introduction of special **decoding** techniques, does Coding Sketch achieve greater accuracy than all existing technologies.

1) Data Structure:

As shown in Figure 1, the data structure of Coding Sketch comprises multiple hierarchical layers, with the ith layer featuring w_i counters, each equipped with B_i bits. Except for the topmost layer, each layer includes flag bits, matching the length of w_i, to record instances of counter overflow. The number of layers in Coding Sketch and the bit size of counters per layer are intricately interwoven and contingent upon the underlying frequency distribution, as we will expound upon in § V. We recommend setting the number of layers to 5, which is usually fine enough for 32-bit counters. Within each layer, every counter is endowed with 3 pointers pointing to 3 distinct positions in the upper layer. These 3 pointers are solely contingent upon hash functions for the subscript of position within current layer, independent of the id of the specific item.

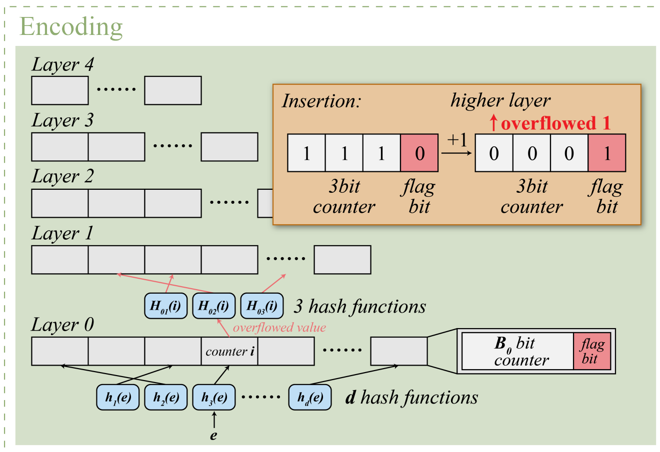


Figure 1: The Hierarchical Data Structure and Encoding Algorithm.

2) Operations:

Encoding: The insertion process in Coding Sketch corresponds to its encoding process, which is an online algorithm. Upon the arrival of an item e, we encode it into the data structure. Initially, we employ d hash functions to map it to d positions in the layer 0 counters using $h_i(e)$. For each subsequent layer, we perform recursive encoding following the steps outlined in Algorithm 1. Whenever the insertion of any counter leads to an overflow, we conduct new insertions at its 3 pointer positions in the higher layer while setting the flag bit of the current position to 1. It is worth noting that the 0th layer utilizes d hash functions, consistent with the parameters of the classic sketch. However, the data structures of the upper layers are uniformly set to 3, a constant recommended through mathematical analysis in § IV-A.

Algorithm 1: Encoding of Coding Sketch in the ith layer.

```
Input: Position p
Function Encode (p, i):
if A_i[p] < mx_i then
    A_i[p] ← A_i[p] + 1
    return
else
    A_i[p] ← 0
    f_i[p] ← 1
    for j in [0..3) do
        h ← H_ij(p)
        Encode(h, i + 1)
    return
```

Example: As shown in Figure 1, the item e is initially mapped to d positions within the layer 0. In the upper layers, overflow values are stored in 3 distinct locations using 3 hash functions. Here, we illustrate the version that incorporates flag bits, where each node comprises a 1-bit flag and a B_i-bit counter. During the insertion process for each node, as illustrated in the figure, we only clear the counter and encode 1 into the upper layer when an overflow occurs. Importantly, we set the flag bit to 1 upon the first instance of overflow.

Online Query: The Coding Sketch excels in post-decoding estimations, yet it still facilitates straightforward online queries. When seeking the frequency of an item e, we initially employ $h_i(e)$ to locate the d positions within the layer 0. Following a query approach akin to that of the Count-Min sketch, our objective is to retrieve the minimum value among these counters. However, it is important to note that some positions may have experienced overflow (indicated by a flag bit of 1), necessitating the continuation of the query into the upper layers. As shown in the Algorithm 2, querying the upper layers mirrors the process in the layer 0, returning the minimum value from the 3 positions and proceeding recursively in the event of overflow.

B. The Recursive and Error-free Decoding

Overview: Our objective is to conserve memory by employing fewer bits in the low-level counters while recording overflows through alternative methods. Nevertheless, if we merely employ the aforementioned query algorithm without further optimizations, the high-level counters may still exhibit errors

Algorithm 2: Query of Coding Sketch in the ith layer.

Input: Pos p
Output: Query result Q_e
Function Query (p, i):
if $f_i[p] = 1$ **then**
  $Q_e \leftarrow +\infty$
  for j in $[0..3)$ **do**
    $h \leftarrow H_{ij}(p)$
    $Q_e \leftarrow \min\{Q_e,$ Query$(h, i+1)\}$
else
  $Q_e \leftarrow 0$
$Q_e \leftarrow Q_e \times mx_i + \mathcal{A}_i[p]$
return Q_e

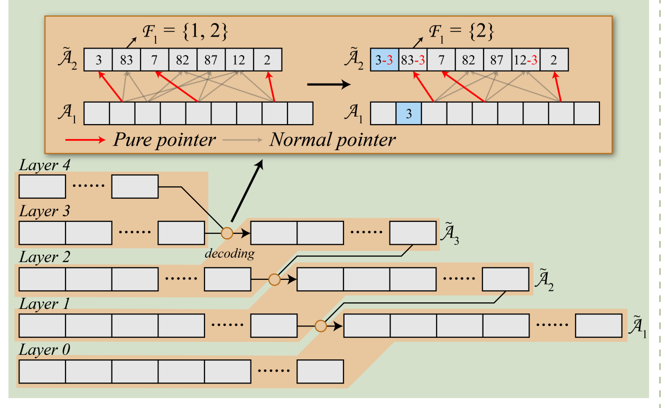


Figure 2: Example of the Decoding Algorithm.

attributable to hashing. We view this method primarily as a means of providing online guideline estimates. To attain more accurate estimations, we introduce a technique for high-probability decoding of the exact low-level overflow values from the upper layers.

Top-down: To obtain overflow values free from errors, the previously employed bottom-up querying method is no longer applicable. As shown in Figure 2, we must engage in a top-down decoding process, layer by layer. Upon completing the decoding of each layer, we will obtain a frequency estimation that encompasses both the original values and overflow values. We designate the counters at layer i upon the completion of decoding as $\widetilde{\mathcal{A}}_i$. The key distinction lies in the fact that $\mathcal{A}_i$ consists solely of original L_i counters, each comprising only B_i bits, with overflow values stored in higher layers Conversely, at this juncture, we opt for the use of 32-bit counters for the counters within $\widetilde{\mathcal{A}}_i$, without additional emphasis on memory conservation. This choice is informed by the fact that decoding is a one-time operation conducted after the insertion of all items, during which we perceive an abundance of resources to facilitate the decoding process.

Pure Pointer: To provide a clearer explanation of the decoding algorithm's process, we first introduce an auxiliary definition: **pure pointer**. Consider an extreme scenario where only one counter overflows to the upper layer. In such cases, we can directly retrieve the exact overflow value. The obstacle to obtaining the precise value arises when multiple pointers, positioned at different locations, point to the same upper layer location. If one of the 3 pointers in an overflowed location directs to a upper layer position without any other pointers from current layer pointing to it, we refer to this pointer as a pure pointer. As illustrated in Figure 2, a pure pointer indicates that the position it targets contains the exact value of the overflow at that location.

Decoding: In our pursuit of obtaining all precise overflow values, we aim to identify a pure pointer for each counter. The online query method referenced in Algorithm 2 lacks the capability to ascertain the purity of a pointer. However, with our current top-down decoding approach, we can locate all pointers to each counter in the upper layer. More precisely, we maintain a set $\mathcal{F}_i[k]$ for each counter within the upper layer $\mathcal{A}i+1$, where $\mathcal{F}_i[k] = \{k', \exists j H_{i-1j}(k') = k | \forall_{k' \in \mathcal{A}_{i-1}}\}$. An illustrative example of $\mathcal{F}_1$ is depicted in Figure 2. In current layer, the $1st$ and $2nd$ positions direct to the first position of the upper layer. Consequently, $F_1[1] = \{1, 2\}$. When a particular k meets the criterion $|\mathcal{F}_i[k]| = 1$, an element within $\mathcal{F}_i[k]$ signifies a pure pointer. We assign the counter of current layer corresponding to this pure pointer as $\hat{p}$, aware that it encompasses 3 pointers. Upon obtaining $\mathcal{A}_i[\hat{p}]$, we can eliminate $\hat{p}$ from the $\mathcal{F}_i$ associated with the other 2 pointers. This deletion operation may yield a new $\mathcal{F}_i[k']$ with a size of 1, thus unveiling a fresh pure pointer. By iterating this procedure, we have devised Decoding Algorithm 3. It is worth noting that this method can decode all overflow values when the higher layer's length surpasses a certain threshold with high probability, as elaborated in § IV-A. In practical implementation, we do not need to maintain the entire set $\mathcal{F}_i$; instead, it suffices to record the XOR sum of all elements in $\mathcal{F}_i$. This technique enables us to decode the entire data structure within a time complexity of $O(N)$, where N is the number of counters.

C. Flagless Prune

Overview: We have additionally devised an algorithm named **flagless pruning**, which achieves error-free decoding without the need for flag bits. In order to obtain $\mathcal{F}_i$ during decoding, each layer's counter in the current data structure needs to maintain an additional 1-bit flag. When each counter utilizes 3 bits, an additional 1 bit is required as a flag bit. This implies that 25% of the memory is allocated for flag bits. Therefore, eliminating the flag bits allows Coding Sketch to save a substantial amount of memory, representing a significant advancement. Moreover, the refined final version of Coding Sketch, without the flag bits, can support deletion operations, which were previously unsupported.

Prune: Now, without flag bits, we initially assume that all counters in the current layer have overflowed. If decoding succeeds, we can still obtain error-free values. However, based on our specific mathematical analysis in § IV-B, this approach requires even more memory overhead compared to using flag bits. Therefore, we introduce the concept of flagless

Algorithm 3: Decoding in the ith layer.

Input: The decoding result of the upper layer $\widetilde{\mathcal{A}}_{i+1}$
Output: The decoding result of this layer $\widetilde{\mathcal{A}}_i$
Function Decode ():
for k in $[0..w_i)$ **do**
  for j in $[0..3)$ **do**
    $h \leftarrow H_{ij}(k)$
    $\mathcal{F}_h \leftarrow \mathcal{F}_h \cup k$
while *find a pure pointer* k **do**
  $\widetilde{\mathcal{A}}_i[k] \leftarrow +\infty$
  for j in $[0..3)$ **do**
    $h \leftarrow H_{ij}(k)$
    $\mathcal{F}_h \leftarrow \mathcal{F}_h \setminus k$
    $\widetilde{\mathcal{A}}_i[k] \leftarrow \min\{\widetilde{\mathcal{A}}_i[k], \widetilde{\mathcal{A}}_{i+1}[h]\}$
  for j in $[0..3)$ **do**
    $h \leftarrow H_{ij}(k)$
    $\widetilde{\mathcal{A}}_{i+1}[h] \leftarrow \widetilde{\mathcal{A}}_{i+1}[h] - \widetilde{\mathcal{A}}_i[k]\}$
for k in $[0..w_i)$ **do**
  if $\widetilde{\mathcal{A}}_i[k]$ *never been decoded* **then**
    for j in $[0..3)$ **do**
      $h \leftarrow H_{ij}(k)$
      $\widetilde{\mathcal{A}}_i[k] \leftarrow \min\{\widetilde{\mathcal{A}}_i[k], \widetilde{\mathcal{A}}_{i+1}[h]\}$
for k in $[0..w_i)$ **do**
  $\widetilde{\mathcal{A}}_i[k] \leftarrow \widetilde{\mathcal{A}}_i[k] \times mx_i + \mathcal{A}_i[k]$
return $\widetilde{\mathcal{A}}_i$

pruning. Flagless pruning can be further categorized into two types: explicit pruning and implicit pruning. These two pruning techniques, when combined, allow us to identify all non-overflowed counters in the current layer. This combined approach achieves the same effect as using flag bits, as we did previously. For counters that are equal to 0 in the upper layer (Due to the top-down decoding approach, the upper layer at this time are already decoded with true values equal to 0.), the pointers in $\mathcal{F}_i$ also be considered pure pointers because these pointers explicitly indicate that the counters did not overflow. We refer to these as **explicit pruning**. By doing this, we eliminate their influence on the other $\mathcal{F}$ sets. However, explicit pruning cannot identify all positions with flag bits actually set to 0, which means that even after this process, many counters that did not actually overflow are still decoded as overflowed counters. We can handle this by treating them as if they had overflowed by 0 and attempting to decode this value forcibly. This requires revisiting the process of searching for pure pointers as in the previous decoding algorithm, which is **implicit pruning**. Implicit pruning does not introduce new operations; instead, after computing the new memory bound, the existing decoding operations can additionally handle false positive overflows that were not previously considered. We refer to counters that have not actually overflowed but are not identified by explicit pruning as false-positive counters. Currently, the size of the upper layer is not only related to the number of overflow counters but also to the presence of false-positive counters. We will discuss this in the mathematical analysis section (see § IV). This memory bound [46] is exceptionally well-managed, and in fact, in our extensive experiments, decoding never failed once the memory bound was satisfied. However, we still provide a solution for potential decoding failures, as shown in the pseudocode. We take the minimum of three hash positions from the remaining $\widetilde{\mathcal{A}}i+1$, which will be an overestimated value. It's worth noting that at this point, most of the values in $\widetilde{\mathcal{A}}i+1$ have already been successfully decoded and subtracted, so the remaining values will not have a significant error.

Example: Figure 3 illustrates an example of decoding using flagless pruning. Initially, we assume that all counters in current layer have overflowed, as indicated by the presence of 3 pointers for each counter. The red pointers represent pure pointers as defined earlier, pointing to positions with $|\mathcal{F}_i| = 1$. The blue pointers indicate pointers to counters with a value of 0, which are explicitly pruned and decoded as 0. Next, we need to remove the pointers to the decoded $\widetilde{\mathcal{A}}_i[k]$ and their overflow effects. As shown in the lower part of Figure 3, new pure pointers can be found among the remaining pointers.

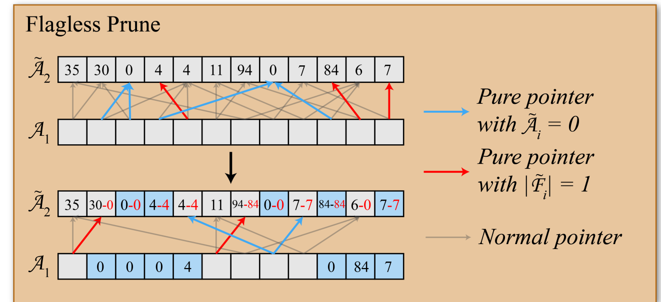


Figure 3: Example of the Flagless Pruning.

Delete: The ability to perform item deletions is a significant advantage of the Count-Min sketch. Existing hierarchical structure sketches [7], [32], [50], [55], due to their reliance on flag bits, do not support deletion operations. The only algorithm proposed to support deletion operations, the Diamond Sketch, uses an additional Sketch to record all deletions. The fundamental issue is that when deleting an item, it is unclear whether the flag bits should be correctly reset. Coding Sketch, through flagless pruning, is the first hierarchical structure sketch to overcome this problem.

D. More Further Optimizations

1) Adjust automatically:

In typical scenarios, leveraging prior knowledge of data distributions allows us to tailor the memory configuration of Coding Sketch accordingly. However, for situations devoid of any prior knowledge, we also offer specialized optimized versions of Coding Sketch capable of automatically adjusting memory allocations to accommodate varying data distributions.

Expansion Mechanism: In § IV-A, we will delineate how the lengths of upper layers are determined once the count

of overflowed counters is known. Counters that experience overflow during the insertion process continue to accumulate. Exceeding predetermined limits can potentially lead to the failure of the final decoding operation, resulting in unacceptably significant errors. Fortunately, the current number of overflowed counters can be ascertained through the flag bits. Adjustments can be made when approaching the limit. Specifically, when the overflowed counters in the layer i are on the verge of surpassing the allowed limit (still within the current threshold), we execute a localized decoding operation to obtain $\widetilde{\mathcal{A}i}$. Subsequently, we create a new layer $i+1$, doubling its length compared to the original. Guided by the decoded $\widetilde{\mathcal{A}i}$, we re-insert the overflowed counters into the new layer i and proceed to reconstruct the upper layers incrementally. This method inevitably introduces some time delay, but the overall average complexity of the algorithm is still $O(N)$.

2) Unbiased Version:

This variant of Coding Sketch can furnish unbiased estimations for the frequency of each flow. The property of unbiased estimation is required in certain specific tasks [5], [15], [33], [44] such as join size estimation and second norm estimation. The core concept aligns with techniques akin to Count Sketch. Upon the arrival of an item, it is mapped to d counters within layer 0 through d hash functions. What sets it apart from the previous versions is the utilization of another hash function to determine whether to insert $+1$ or -1. The insertion and carry processes for each layer remain analogous to the prior versions.

Ones-complement: All counters now employ signed numbers, albeit utilizing ones-complement rather than two's complement. This choice arises from the fact that counter overflows are now classified into two scenarios: positive overflow and negative overflow. Accordingly, we insert $+1$ or -1 upwards, respectively. If two's complement were employed, the maximum value for a negative number would exceed that of a positive number by 1. In the event of a position experiencing a negative overflow subsequent to a positive overflow, they would cancel each other out at upper layers. However, in reality, an additional 1 is introduced, resulting in an error. Cleverly, the additional 0 state created by using the ones-complement code is not wasted. We can distinguish whether a counter has never been accessed or has been accessed but its value has been offset to 0 by using $+0$ and -0. This way, we can still employ the explicit pruning of the flagless version of Coding Sketch without needing to record flag bits.

IV. Mathematical Analysis

In this section, we begin by establishing the correlation between the required length in the flag-bit version and the lower-level overflow counter, as detailed in § IV-A. Subsequently, we derive a memory design approach for the flagless version in § IV-B based on this relationship.

A. Analysis of the length of the upper layer

Considering the counter $\mathcal{A}_i$ in the layer i, we denote the number of overflowed counters among these w_i counters as K and the number of counters without overflows as $L = w_i - K$. Upon revisiting our algorithm, we observe that there will be $3 \times K$ pointers inserted into the w_{i+1} counters in the layer $(i+1)$.

Theorem 1. *The decoding process in the version with the flag bit when $w_{i+1} \geqslant 1.13 \times K$ can get all the $\mathcal{A}_i$ with a high probability.*

Our decoding algorithm can be reformulated as the process of finding the 2-core of a random hypergraph (definition see [21], [37]). A similar analytical approach can then be applied. Specifically, within this framework, we treat the counters in this layer and the higher layer as vertices in the hypergraph, while our pointers represent hyperedges. Our decoding process corresponds to the standard "peeling process," where we continuously identify vertices with degree 1 and remove them. However, in our work, the process of converting this hash structure into a hypergraph makes certain assumptions that are more lenient. There remains room for further optimization, as demonstrated in a recent article presented at SODA 2021 [46]. This paper introduces a novel design of specialized hash functions that enable a distinct transformation of the problem, resulting in tighter bounds. Specifically, with 3 hash functions, the new approach achieves successful decoding with high probability as long as $w_{i+1} \geqslant 1.13 \times K$. For situations involving a larger number of hash functions, corresponding thresholds exist. However, taking into account the trade-off between time complexity and performance, we recommend employing 3 hash functions in Coding Sketch. Our designed decoding algorithm ingeniously corresponds to the mathematical modeling in these articles, allowing us to apply their derivations.

B. Analysis of the decoding without flag bit

Theorem 2. *For the flagless version, we maintain the definitions of K and L as the same as those in Theorem 1. To ensure the error-free completion of the decoding algorithm with high probability, the following conditions must be met:*

$$w_{i+1} \geqslant 1.13(K + \sqrt{4.6L}) + \sqrt[4]{30.51K^3 \cdot L}$$

We will present the proof in three steps: First, we analyze the false positive probability similar to Bloom Filters. In the second step, we introduce a random variable $\hat{K}$, which represents the number of counters that need to be decoded when accounting for false positives. Finally, we provide an appropriate bound using the Chernoff inequality.

Step 1: We insert $3 \times K$ pointers into the upper layer. Here, we treat these $3 \times K$ insertions as completely independent events. For each counter in the upper layer, it is nonzero only when none of the $3 \times K$ pointers are directed towards it. Therefore, based on the Bernoulli Inequality, for all positions k in the upper layer, we have: $\mathbb{P}(\mathcal{A}[k] = 0) = (1 - \frac{1}{w_{i+1}})^{3K}$, $\mathbb{P}(\mathcal{A}[k] \neq 0) = 1 - (1 - \frac{1}{w_{i+1}})^{3K} \leqslant 1 - (1 - \frac{3K}{w_{i+1}}) = \frac{3K}{w_{i+1}}$

Furthermore, for each non-overflow position in current layer, there are 3 pointers. Only when all of these 3 pointers point to non-zero values, do we incorrectly classify it as an

overflow counter. Since the position itself did not overflow, we can consider these 3 pointers as new queries, independent of the original $3 \times K$ insertions. Consequently, we have: $\mathbb{P}(\text{false positives}) = \mathbb{P}(\neq 0)^3 \leqslant (\frac{3K}{w_{i+1}})^3$

In the subsequent derivations, we directly consider the probability of false positives as an upper bound, i.e., $(\frac{3K}{w_{i+1}})^3$. We then introduce the indicator variable $fp_k = \begin{cases} 1 & w \cdot p \cdot (\frac{3K}{w_{i+1}})^3 \\ 0 & w \cdot p \cdot 1 - (\frac{3K}{w_{i+1}})^3 \end{cases}$ to determine whether a false positive occurs at a non-overflow position. Then, we get L entirely independent random variables following the Bernoulli distribution.

Step 2: In the decoding process of the flagless version, explicit pruning considers all pointers directed to 0 as pure pointers. Therefore, we only need to account for overflowed and false-positive counters. We define the quantity of these counters as $\hat{K}$, which accounts for the implicit pruning scenario as well. To ensure successful decoding, similar to Theorem 1, we require $w_{i+1} \geqslant 1.13 \times \hat{K}$.

But now $\hat{K}$ is no longer a fixed value but a random variable. Specifically, $\hat{K} = K + \sum_{k=1}^{L} fp_k$. Let $\mu = \mathbb{E}[\hat{K}] = K + L \cdot (\frac{3K}{w_{i+1}})^3$. The second half is a binomial distribution. So according to Chernoff inequality, when we set $\lambda = \sqrt{4.6L} \geqslant \sqrt{\frac{\ln(10000)}{2} L}$ we have $\mathbb{P}(\sum_{i=k}^{L} fp_k \geqslant \mu + \lambda) \leq e^{-\frac{2\lambda^2}{L}} \leq \frac{1}{10000}$

This implies that we have $\hat{K} \leqslant K + \sqrt{4.6L} + L \cdot (\frac{3K}{w_{i+1}})^3$ with a high probability. An error rate of $\frac{1}{10000}$ is considered acceptable, and in practical experiments, this bound is sufficient. Moreover, the remaining part of the formula offers additional room for scaling.

Step 3: The last step is to solve the inequality $w_{i+1} \geqslant 1.13 \times (K + \sqrt{4.6L} + L \cdot (\frac{3K}{w_{i+1}})^3)$. According to the above derivation, we know that when this inequality is established, there is a high probability of $\hat{K} \leqslant K + \sqrt{4.6L} + L \cdot (\frac{3K}{w_{i+1}})^3$, so that there is a high probability of $w_{i+1} \geqslant 1.13 \times \hat{K}$, and then error-free decoding is possible. We assume $w_{i+1} = 1.13(K + \sqrt{4.6L}) + c_1 \cdot K\sqrt[3]{L} \geqslant c_1 \cdot K\sqrt[3]{L}$

$$(\frac{3K}{w_{i+1}})^3 \leqslant (\frac{3K}{c_1 \cdot \sqrt[3]{L} \cdot K})^3 = (\frac{3}{c_1 \sqrt[3]{L}})^3 = \frac{27}{c_1^3 \cdot L}$$

Substituting into the above inequality then we can get: $w_{i+1} = 1.13(K + \sqrt{4.6L}) + c_1 \cdot K\sqrt[3]{L} \geqslant 1.13(K + \sqrt{4.6L} + L \cdot (\frac{3K}{w_{i+1}})^3)$

$$c_1 \cdot K\sqrt[3]{L} \geqslant 1.13L \cdot \frac{27}{c_1^3 \cdot L} = \frac{1.13 \cdot 27}{c_1^3}$$

$$c_1 \geqslant \sqrt[4]{\frac{30.51}{\sqrt[3]{L} \cdot K}}$$

To sum up, the inequality always holds when $c_1 \geqslant \sqrt[4]{\frac{30.51}{\sqrt[3]{L} \cdot K}}$. So we can get Theorem 2 by substituting $c_1 = \sqrt[4]{\frac{30.51}{\sqrt[3]{L} \cdot K}}$ into $w_{i+1} = 1.13(K + \sqrt{4.6L}) + c_1 \cdot K\sqrt[3]{L}$.

V. Experimental Results

In this section, we evaluate the performance of Coding Sketch across various tasks. We begin by detailing our experimental setup in § V-A. Next, in § V-B, we delve into the hyperparameters suitable for Coding Sketch in the task of frequency estimation. Lastly, we conduct comparisons between Coding Sketch and state-of-the-art algorithms in terms of accuracy and speed in § V-C and § V-C3. Lastly, in Section V-D, we experiment with an unbiased version of Coding Sketch and assess its performance in the task of join size estimation.

A. Experimental Setup

Implementation and Settings: We have implemented Coding Sketch and all comparative algorithms in C++. Compilation was carried out using g++ 7.5.0 (Ubuntu 7.5.0-6ubuntu2) with the -O3 optimization option enabled. To ensure efficient hashing, all algorithms make use of the widely adopted MurmurHash [1] hash function. For the Stingy Sketch and Pyramid Sketch algorithms, we have utilized the open-source code provided by their original authors. For other algorithms, they have been independently reproduced by the authors of this paper. Additionally, the original Pyramid and Stingy papers introduce auxiliary techniques like Hash Split and Prophet Queue, which have broad applicability across various sketch algorithms. To maintain fairness in performance comparisons, we have deliberately refrained from incorporating these techniques. Our experiments were conducted on a CPU server equipped with an 18-core 4.2GHz Intel i9-10980XE processor. This server is accompanied by 128GB of 3200MHz DDR4 memory and boasts a 24.75MB L3 cache.

Datasets: We employed two authentic datasets to simulate real-world scenarios. Additionally, a synthetic dataset with adjustable skewness was employed to assess algorithm performance across diverse data distributions.

- **CAIDA:** CAIDA [3] contains 10 real IP tracking datasets collected from high-speed monitors on backbone links. Each item has 13 bytes IP address and 8 bytes time stamp. Each dataset has a total of about 1.3 million items and 26 million packets. In our experiments we only used a 1 million slice of it.
- **Campus:** Campus is a dataset of 10 real IP tracking data collected from gateways on our campus. Each dataset has about 180,000 items and 2.4 million packages. In our experiments, we only used a 1 million slice of it.
- **Kosarak:**The Kosarak dataset contains anonymized click-stream data from a Hungarian online news portal. Experiments indicate that this dataset also exhibits high skewness. We extracted a segment of data with a length of 1 million for our experiments.
- **Synthetic:** We generated datasets of length 1 million representing the Zipf distribution. Zipf's law is an experimental law that observes this distribution in the frequency of web page visits. When the parameter α of the Zipf distribution is larger, the skewness of the dataset is higher. We tested different α from 0 to 3.0 in our experiments to simulate different skewness. In particular, we use the Zipf($\alpha = 0.8$) dataset to

simulate a dataset with high skewness. This parameter can lead to a situation similar to real world datasets. In some cases, we use Zipf($\alpha = 0.3$) to simulate lower skewness datasets in order to demonstrate the generality of our algorithm.

Metrics: We measure the following metrics in our experiments:

• **Average Absolute Error (AAE):** $\frac{1}{|\Psi|}\sum_{e\in\Psi}|\hat{f}(e) - f(e)|$, $f(\cdot)$ and $\hat{f}(\cdot)$ are real and estimated frequency respectively. Ψ is the query set, usually the complete set E.

• **Average Relative Error (ARE):** $\frac{1}{|\Psi|}\sum_{e\in\Psi}|\hat{f}(e) - f(e)|/f(e)$, Ψ, $f(\cdot)$ and $\hat{f}(\cdot)$ are the same as those defined above.

• **Throughput**: We use million operations per second to measure the speed of various algorithms.

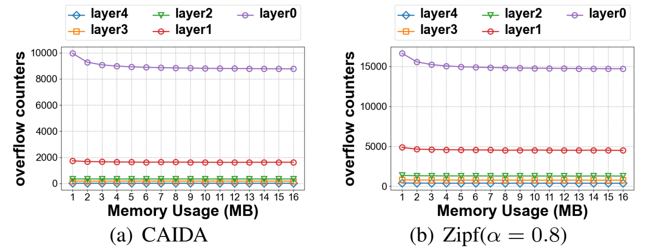
B. Impact of Algorithm Parameters

In this section, we present an overview of various parameters utilized by Coding Sketch and conduct experiments to elucidate the influence of select hyperparameters. The ability to configure these hyperparameters is a contribution of Coding Sketch. Pyramid and Stingy use a binary tree structure, which is equivalent to $w_i = w_{i-1}/2$. In our algorithm, w_i is set through mathematical derivation. Both Pyramid and Stingy default to all $b_i = 4$, but we believe that a smaller b_i can achieve better accuracy, which requires a trade-off with memory and time. We have conducted extensive experiments to analyze these parameters, which are overlooked in existing works, and have provided recommended settings.

• **The Number of Hash Functions** d: d denotes the number of positions to which an item is inserted upon arrival, corresponding to the creation of multiple counter arrays in classic sketches. The impact of this parameter has been extensively analyzed in existing literature. For our experiments, we default to $d = 3$.

• **The Total Number of Layers of Coding Sketch**: A greater number of layers enables finer hierarchical partitioning. However, an excessive number of layers can result in numerous overflows, leading to slow encoding—a scenario we aim to avoid. Although, as demonstrated in § IV-A, we have proven that upper layers require only linear memory to achieve successful decoding with high probability, the factor of 1.13 still implies that more layers demand increased memory. Considering a balanced assessment of accuracy and speed, we have opted for a 5-layer hierarchical data structure in our experimental setup.

• **Analysis of the Length of Each Layer**: When the total memory of the entire data structure and the number of bits per counter are fixed, dividing these resources allows us to determine the total number of counters. However, allocating these counters across different layers presents a challenge and requires the setting of a hyperparameter. In state-of-the-art techniques like Pyramid and Stingy, this parameter is typically determined by a fixed ratio, where the length of each higher layer is a constant multiple of the layer below it. In a binary tree structure, this constant is typically set to $\frac{1}{2}$. However, as the frequency distribution varies, so does the probability and quantity of overflows. This fixed allocation approach struggles to accommodate varying data skewness. High skewness leads to frequent conflicts in upper layers, while low skewness results in wasted memory in those layers. In Coding Sketch, the lower bound for the length of a upper layer can be determined based on the number of overflows in the current layer. Ideally, if we knew the overflow quantities in advance, we could optimize the allocation of layer lengths. Unfortunately, this level of knowledge is often unattainable, leading to a few scenarios: When the frequency distribution is unknown, we can employ an automatic adjustment version as described in § III-D1. Alternatively, we can set this parameter assuming a sufficiently high skewness, which essentially aligns with the Pyramid and Stingy approaches. For datasets with known distribution characteristics, we can estimate the corresponding overflow counter quantities to adjust layer lengths accordingly. The flexibility of Coding Sketch's adaptable data structure allows us to experimentally analyze the factors influencing the actual number of overflow counters, as we will demonstrate shortly.



(a) CAIDA (b) Zipf($\alpha = 0.8$)

Figure 4: Number of Overflow Counters vs. Memory.

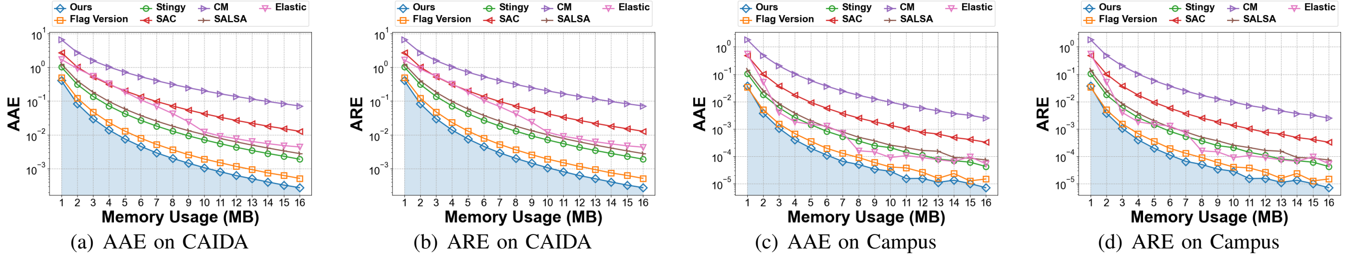
Number of Overflow Counters vs. Memory (Figure 4): In this experiment, we configured $B_i(i = 0, ..., 4) = (4, 3, 3, 2, 2)$ and fixed the memory size at 1MB. While this memory size is considered insufficient for Coding Sketch, it is instrumental for a more in-depth analysis. We simulated scenarios involving data with varying degrees of skewness using different Zipf datasets, where α ranged from 0.1 to 0.9. As α increases, we observe a corresponding increase in the number of overflow counters at each layer. This trend emerges due to the higher skewness of the dataset, resulting in an augmented quantity of frequent flows. Consequently, we witness more overflows, particularly in upper layers. The number of overflow counters in L_0 exhibits the most rapid growth. This phenomenon arises from the heightened sensitivity of lower-level counters to skewness fluctuations. Conversely, when the dataset exhibits lower skewness, we observe an absence of overflows beyond L_1. This absence arises because the frequencies of the most common flows in the dataset fail to trigger overflows.

C. Experiments on Frequency Estimation

In this subsection, we show the experimental results of Coding Sketch on the frequency estimation problem.

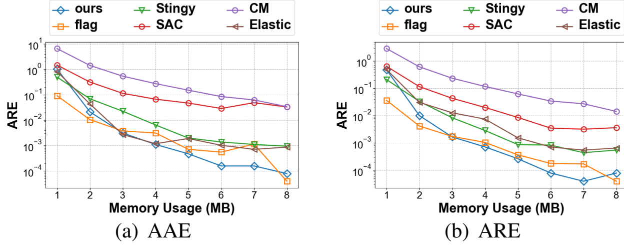
1) Accuracy vs. memory:

Experiment setting: We have conducted experiments using the CAIDA dataset, Campus dataset, Kosarak dataset, Zipf0.3 dataset, and Zipf0.8 dataset, evaluating both the Coding Sketch



(a) AAE on CAIDA (b) ARE on CAIDA (c) AAE on Campus (d) ARE on Campus

Figure 5: Comparison of accuracy on two real datasets.



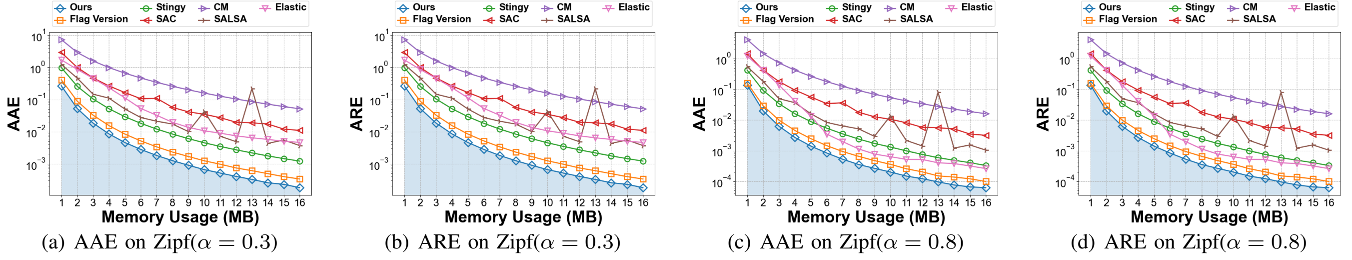
(a) AAE (b) ARE

Figure 6: Comparison of accuracy on Kosarak

and its flagless variant. We compare the accuracy performance(AAE and ARE) with other sketches including Stingy, Pyramid,Count-Min, SAC, Elastic.

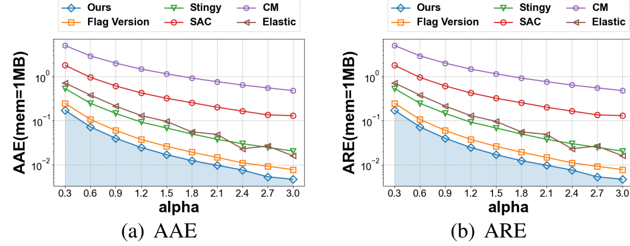
Real datasets (Figure 5): We conducted experiments evaluating the AAE (Average Absolute Error) and ARE (Average Relative Error) on the CAIDA, Campus and Kosarak real datasets. To facilitate a more detailed observation of differences, logarithmic plots were employed. The accuracy of all algorithms generally improves with increased available memory. CM (Count-Min) serves as a benchmark classic sketch, and all optimized sketches consistently outperform CM. As an example of flat structure sketches, SAC demonstrates an approximate 10-fold improvement over CM. SALSA lies in the intermediate zone between two kind of structure sketches, whereas Stingy represents the latest advancement in hierarchical structure sketches. Unlike other Sketches that do not record IDs, the core idea of Elastic is to identify frequent elements and separately record their IDs and more accurate occurrence counts. When there is enough memory to identify all frequent elements, Elastic can perform very well, such as with 7MB in the CAIDA dataset and 8MB in the Campus dataset. Below this memory threshold, the performance is average, but upon reaching this memory level, Elastic suddenly improves. However, increasing memory beyond a certain point does not further enhance Elastic's performance, as seen with more than 12MB in the CAIDA dataset and more than 10MB in the Campus dataset. This is because the frequent elements are already accurately identified, but filtering frequent elements always introduces errors. This type of error depends on the filtering algorithm, and increasing memory does not contribute significantly to reducing it. In contrast, other Sketches that do not record IDs do not have this type of error. It is noteworthy that Coding Sketch consistently exhibits superior accuracy compared to all existing algorithms. For comparative purposes, we have retained the experimental results of the version of Coding Sketch with flag bits. As expected, the experiments clearly indicate that the version of Coding Sketch with flagless pruning consistently outperforms its counterpart with flag. From Figure 5, it becomes evident that when utilizing 16MB of memory, Coding Sketch achieves a substantial reduction in AAE compared to the Flag version, Stingy, and Count-Min by factors of 2.25, 24.5, and 589, respectively. In terms of ARE, Coding Sketch achieves reductions of 1.5, 14.25, and 396 when compared to the Flag version, Stingy, and Count-Min, respectively. It is important to highlight that Coding Sketch demonstrates superior optimization in AAE compared to ARE because the ARE expression implies that the errors of frequent flows are calculated to a lesser extent. Coding Sketch effectively mitigates high-level collisions within hierarchical data structures, resulting in more accurate estimations of frequent flows. Consequently, Coding Sketch excels in AAE, which underscores the rationale for testing both AAE and ARE. In the Campus dataset experiments, when memory exceeds 12MB, Coding Sketch exhibits some fluctuations. This behavior arises from the fact that errors have already reached very low levels. It is worth noting that similar fluctuations are observed in CM errors when plotting them at 8 times the memory, illustrating that such fluctuations are not exclusive to Coding Sketch but are inherent to the data and memory constraints.

Synthetic datasets (Figure 7): We have selected $\alpha = 0.3$ and $\alpha = 0.8$ from the Zipf distribution datasets to serve as representatives of synthetic datasets. In the synthetic datasets, the overall performance trends of each algorithm align consistently with those observed in real datasets. Notably, even after conducting 50 repetitions of the experiment, both SALSA and SAC continue to exhibit fluctuations in their performance. Of particular interest is an anomalous peak observed in SALSA's performance at the 13MB memory mark. Upon further investigation, we have identified that during this interval, SALSA experiences multiple collisions among frequent flows. As a result, a counter undergoes repeated mergers, leading to a substantial increase in error. Additionally, the insertion speed of SALSA during this period significantly diminishes, as each arrival of these frequent flows necessitates accessing this particular counter. It's noteworthy that the same situation occurs when $\alpha = 0.3$ and $\alpha = 0.8$, as the IDs of frequent flows in the Zipf generation remain constant. In the case of $\alpha = 0.3$, the dataset exhibits lower skewness, resulting in a performance decline across all algorithms. Elastic essentially does not perform better than Stingy Sketch, except on the



(a) AAE on Zipf($\alpha = 0.3$) (b) ARE on Zipf($\alpha = 0.3$) (c) AAE on Zipf($\alpha = 0.8$) (d) ARE on Zipf($\alpha = 0.8$)

Figure 7: Comparison of accuracy on two Synthetic datasets.



(a) AAE (b) ARE

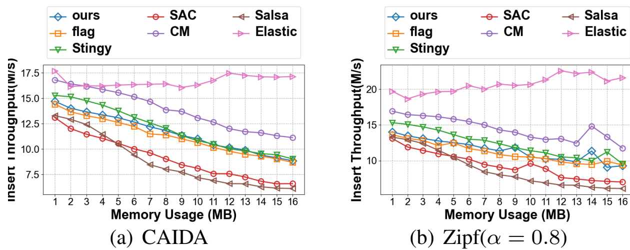
Figure 8: Comparison of accuracy on Skewness.

Zip0.8 dataset, which has higher skewness, where it surpasses Stingy. This is because it separately records the frequencies of frequent flows more accurately. This characteristic is also displayed in the subsequent frequent flows query tasks, and we consider this to be reasonable. Coding Sketch consistently ranks as the top-performing algorithm in this scenario.

Summary and analysis: We have observed that Coding Sketch outperforms other algorithms across both real and synthetic datasets. As evident from Figure 5 and Figure 7, when employing Coding Sketch with a memory capacity exceeding 4MB, its AAE and ARE exhibit significant reductions, approaching approximately 2 times, 10 times, and 100 times lower values than those of the Flag version, Stingy, and Count-Min, respectively.

2) Accuracy vs. Skewness:

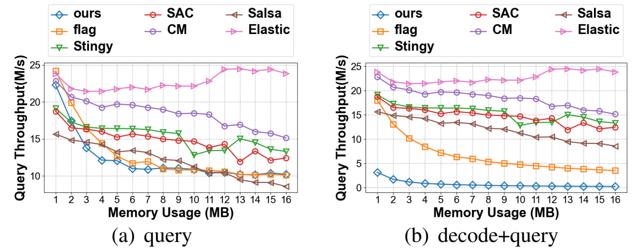
Experiment setting: We conducted experiments utilizing the Zipf datasets (α=0,...,3.0) with Coding Sketch and its flagless variant. For both versions of the Coding Sketch, we configured $B_i(i = 0, ..., 4) = (4, 3, 3, 2, 2)$. We compare their accuracy performance (AAE and ARE) with other sketches, including Stingy, Pyramid, Count-Min, Elastic and SAC. SALSA's performance decreases to an unacceptable level when the skewness of the dataset is high, therefore, we did not include it in this experiment.



(a) CAIDA (b) Zipf($\alpha = 0.8$)

Figure 9: Insert Throughput.

Analysis (Figure 8): The experimental results reveal that the accuracy of all algorithms increases as α escalates. Remarkably, even a venerable sketch like CM, which does not account for skewness, demonstrates this trend. This phenomenon is attributed to the fact that when the dataset size remains constant, greater skewness in the Zipf distribution leads to fewer distinct flows. Nonetheless, it is noteworthy that the optimized sketches exhibit an even more rapid enhancement in performance. Among these, Coding Sketch consistently attains the best accuracy.



(a) query (b) decode+query

Figure 10: Query Throughput.

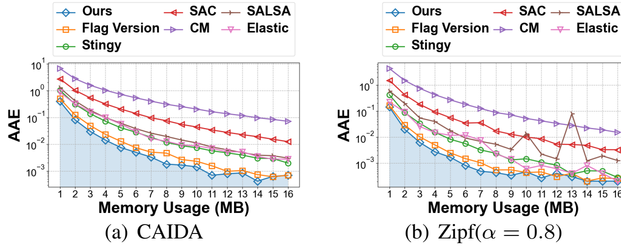
3) Experiments on Throughput:

Experiment setting: We assessed the throughput of Coding Sketch and conducted a comparative analysis with other algorithms using the CAIDA and Zipf($\alpha = 0.8$) datasets. In the experiment, we inserted 1 million items and measured the time required, repeating this process with a change in the hash function's seed and averaging the results over 10 iterations. While some techniques, such as hash separation and the Prophet Queue, have been proposed for general application in sketches like Stingy and Pyramid, we deliberately refrained from employing any of these methods in our experiments to maintain fairness. Consequently, the experimental results at this stage reflect the baseline operational speeds of these sketches.

Insert Throughput Analysis (Figure 9): As mentioned in its paper, Elastic achieves a high throughput through certain accelerations. CM, with its simple operations, is the second fastest. Stingy exhibited a structural advantage with tree nodes sharing similar physical locations, resulting in fewer memory accesses and securing the third-fastest position. Coding Sketch performed as the fourth fastest. The presence or absence of flag bits did not introduce bottlenecks during insertion, and the speed difference between the two versions remained minimal. Although Coding Sketch features a more intricate encoding compared to other algorithms, its insertion algorithm maintained efficiency without significant slowdown, consistently outpacing SALSA and SAC. We consider this performance to be highly commendable.

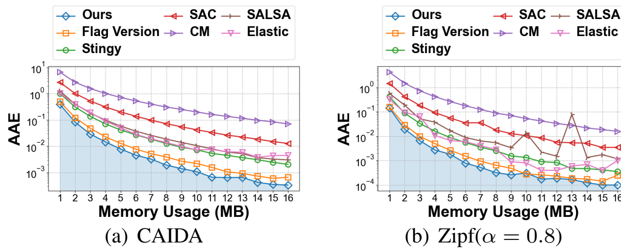
Query Throughput Analysis (Figure 10): Our algorithm's query efficiency is not high, as we previously assumed. Coding Sketch is suitable for scenarios where high accuracy is

required, and queries can be performed offline without the need for speed. Figure 10(a) displays the query throughput of all algorithms. Here, we did not include the time for decoding operations but assumed queries after decoding is completed. It can be seen that aside from the optimized Elastic and the simplest CM, the query throughput is comparable. Figure 10(b) includes the time for decoding operations, meaning the calculation for other algorithms includes all query times, while our algorithm includes one decoding time + all query times. It can be seen that at this point, our algorithm's throughput is lower by a factor. The version without flag bits performs even worse due to the longer decoding time required.



(a) CAIDA (b) Zipf($\alpha = 0.8$)

Figure 11: Flows with a frequency greater than 16



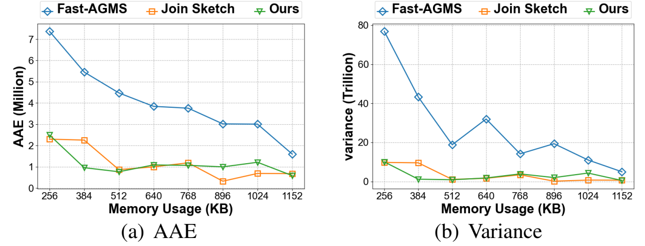
(a) CAIDA (b) Zipf($\alpha = 0.8$)

Figure 12: Top10000 flows

4) Queries on Frequent flows:

Experiment setting: In this experiment, we transitioned from querying estimates for all flows to focusing exclusively on a subset of frequent flows. The objective of this experiment is to provide a more intuitive demonstration of how our algorithm is particularly adept at handling frequent flows after effectively addressing high-level conflicts. To highlight this more, we just query AAE instead of ARE. We conducted queries under two specific conditions. The first involved querying the $10,000$ flows with the highest actual frequencies, a common parameter in Top-k query tasks. The second condition encompassed querying all flows with a true frequency exceeding 16, a more stringent criterion than the first. In our configuration, where $B_0 = 4$, this query included all flows necessitating insertion into L_1. The aim of this experiment was to evaluate our algorithm's capacity to estimate frequent flows accurately.

Analysis (Figure 11, 12): From Figure 11 and Figure 12, it is evident that the relative ordering of all algorithms remains unchanged. However, due to the smaller query set, all algorithms exhibit some fluctuations. The peaks observed in SALSA at 10MB and 13MB in the Zipf dataset are attributed to conflicts among frequent flows, and as such, they persist in this experiment. Comparatively, the relative positions of Stingy and SALSA have not undergone significant alterations. Nonetheless, their degree of optimization in this particular test is not as pronounced as it is in a full-stream query scenario. In this experiment, the difference in performance of Coding Sketch with or without flag bits removal is minimal. This is because, as long as there exist decoding algorithms, it can be ensured that the probability of high-level errors is exceptionally low. The impact of flagless pruning is to conserve more memory across all flows.



(a) AAE (b) Variance

Figure 13: Join Size Estimation.

D. Experiments on Join Size Estimation

Experiment setting: We tested the task of join size estimation on the CAIDA dataset, comparing the Fast-AGMS, Join Sketch, and Coding Sketch with memory ranging from 256KB to 1MB. Since both methods yield unbiased frequency estimates, the law of large numbers assures us that, after a sufficient number of repetitions, they consistently obtain the exact values. Therefore, in addition to AAE, we evaluated the variance of their estimates, which serves as an indicator of how quickly they converge to the precise value. This enables a comparative analysis of the strengths and weaknesses of the algorithms. The ground truth of the join size is extraordinarily large, reaching the magnitude of 2×10^9. Consequently, while the AAE values depicted in the graphs may appear substantial, they represent relatively minor errors.

Analysis (Figure 13): Clearly, Coding Sketch and Join Sketch always outperform Fast-AGMS. Furthermore, Coding Sketch and Join Sketch are comparable. Their significant difference, as mentioned in the related work section, is that Join Sketch requires recording the IDs of frequent items. In this experiment, the length of the IDs is 8 bits. If the IDs in actual datasets are shorter, then Join Sketch would perform better; otherwise, our algorithm would have the advantage.

VI. Conclusion

In this paper, we propose a sketch for a new type of hierarchical structure called Coding Sketch. First, Coding Sketch proposes an encoding and decoding algorithm to solve hash collisions in the upper layer that have been limiting the accuracy of existing hierarchical structure sketches. Second, the decoding algorithm in Coding Sketch can add flagless pruning and then the hierarchical structure no longer needs flag bits. This technique saves a lot of space for more counters, so Coding Sketch can obtain more accurate frequency estimates for both frequent and infrequent items. Experiments prove that Coding Sketch has an order of magnitude better accuracy than the state-of-the-art algorithm. All of our code is open sourced on Github [2].

ACKNOWLEDGEMENT

We thank all anonymous reviewers for their help in improving this paper. This work is supported by National Key R&D Program of China (No. 2022YFB2901504), and National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009).

REFERENCES

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

[2] Related source code. https://github.com/CodingSketch/Coding-Sketch.

[3] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[4] M. Ahmed, A. Naser Mahmood, and J. Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016.

[5] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. *Journal of Computer and System Sciences*, 64(3):719–747, 2002.

[6] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

[7] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik. SALSA: self-adjusting lean streaming analytics. In *ICDE*, pages 864–875. IEEE, 2021.

[8] W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*, pages 18–35, 2019.

[9] M. Charikar, K. C. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[10] D. Charles and K. Chellapilla. Bloomier filters: A second look. In *Algorithms-ESA 2008: 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings 16*, pages 259–270. Springer, 2008.

[11] M. Chiosa, T. B. Preußer, and G. Alonso. Skt: A one-pass multi-sketch data analytics accelerator. *Proceedings of the VLDB Endowment*, 14(11):2369–2382, 2021.

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

[13] G. Cormode. Data summarization and distributed computation. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 167–168, 2018.

[14] 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*, pages 13–24, 2005.

[15] 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*, pages 13–24, 2005.

[16] G. Cormode, S. Maddock, and C. Maple. Frequency estimation under local differential privacy. *Proceedings of the VLDB Endowment*, 14(11):2046–2058, 2021.

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

[18] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. *IEEE/ACM Transactions on Networking*, 2005.

[19] Z. Dai, A. Desai, R. Heckel, and A. Shrivastava. Active sampling count sketch (ascs) for online sparse estimation of a trillion scale covariance matrix. In *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[20] A. Datta, Y. Izenov, B. Tsan, and F. Rusu. Simpli-squared: A very simple yet unexpectedly powerful join ordering algorithm without cardinality estimates. *arXiv preprint arXiv:2111.00163*, 2021.

[21] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. Tight thresholds for cuckoo hashing via xorsat. In *International Colloquium on Automata, Languages, and Programming*, pages 213–225. Springer, 2010.

[22] Y. Du, H. Huang, Y.-E. Sun, S. Chen, and G. Gao. Self-adaptive sampling for network traffic measurement. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[23] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *ACM SIGCOMM CCR*, 2002.

[24] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data. Technical report, Technical Report, Dec. 2001. citeseer. nj. nec. com/gilbert01quicksand. html, 2001.

[25] 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 *IEEE BigData*, pages 2380–2389. IEEE Computer Society, 2017.

[26] M. T. Goodrich and M. Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799. IEEE, 2011.

[27] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. Compass: Online sketch-based query optimization for in-memory databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 804–816, 2021.

[28] Y. Izenov, A. Datta, F. Rusu, and J. H. Shin. Online sketch-based query optimization. *arXiv preprint arXiv:2102.02440*, 2021.

[29] P. Jia, P. Wang, J. Zhao, S. Zhang, Y. Qi, M. Hu, C. Deng, and X. Guan. Bidirectionally densifying LSH sketches with empty bins. In *SIGMOD Conference*, pages 830–842. ACM, 2021.

[30] J. Jiang, F. Fu, T. Yang, and B. Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 1269–1284, 2018.

[31] A. Kipf, D. Vorona, J. Müller, T. Kipf, B. Radke, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1937–1940, 2019.

[32] H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui. Stingy sketch: a sketch framework for accurate and fast frequency estimation. *Proceedings of the VLDB Endowment*, 15(7):1426–1438, 2022.

[33] 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 *KDD*, pages 1574–1584. ACM, 2020.

[34] C. Lou, Y.-E. Sun, H. Huang, Y. Du, S. Chen, G. Gao, and H. Xu. An efficient adaptive denoising sketch for per-flow traffic measurement. In *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pages 161–168, 2022.

[35] A. Mandal, H. Jiang, A. Shrivastava, and V. Sarkar. Topkapi: parallel and fast sketches for finding top-k frequent elements. *Advances in Neural Information Processing Systems*, 31, 2018.

[36] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, pages 398–412. Springer, 2005.

[37] M. Molloy. The pure literal rule threshold and cores in random hypergraphs. 2004.

[38] A. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire. Correlation sketches for approximate join-correlation queries. In *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[39] S. Sheng, Q. Huang, S. Wang, and Y. Bao. Pr-sketch: monitoring per-key aggregation of streaming data with nearly full accuracy. *Proceedings of the VLDB Endowment*, 14(10):1783–1796, 2021.

[40] B. Shi, Z. Zhao, Y. Peng, F. Li, and J. M. Phillips. At-the-time and back-in-time persistent sketches. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1623–1636, 2021.

[41] B. Shi, Z. Zhao, Y. Peng, F. Li, and J. M. Phillips. At-the-time and back-in-time persistent sketches. In *SIGMOD/PODS '21: International Conference on Management of Data*, 2021.

[42] Y. Shi and M. Wen. Srouting: Towards a better flow size estimation performance through routing and sketch configuration. In *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[43] X. Teng, Y.-R. Lin, and X. Wen. Anomaly detection in dynamic networks using multi-view time-series hypersphere learning. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 827–836, 2017.

[44] D. Ting. Count-min: Optimal estimation and tight error bounds using empirical error distributions. In *SIGKDD*, 2018.

[45] D. Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[46] S. Walzer. Peeling close to the orientability threshold–spatial coupling in hashing-based data structures. In *Proceedings of the 2021 ACM-SIAM*

Symposium on Discrete Algorithms (SODA), pages 2194–2211. SIAM, 2021.

[47] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. S. Lui, and X. Guan. A memory-efficient sketch method for estimating high similarities in streaming sets. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD, 2019*, pages 25–33. ACM, 2019.

[48] W. Xie, F. Zhu, J. Jiang, E.-P. Lim, and K. Wang. Topicsketch: Real-time bursty topic detection from twitter. *TKDE*, 2016.

[49] K. Yang, Y. Li, Z. Liu, T. Yang, Y. Zhou, J. He, T. Zhao, Z. Jia, Y. Yang, et al. Sketchint: Empowering int with towersketch for per-flow per-switch measurement. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2021.

[50] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li. Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2650–2662, 2019.

[51] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li. Heavyguardian: Separate and guard hot items in data streams. In *KDD*, pages 2584–2593. ACM, 2018.

[52] 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 *SIGCOMM*, pages 561–575. ACM, 2018.

[53] 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 *INFOCOM*, pages 2017–2025, 2019.

[54] 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 Trans. Netw.*, pages 1845–1858, 2019.

[55] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li. Pyramid sketch: a sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 10(11):1442–1453, 2017.

[56] Y. Yang, Y. Zhang, W. Zhang, and Z. Huang. GB-KMV: an augmented KMV sketch for approximate containment similarity search. In *ICDE*, pages 458–469. IEEE, 2019.

[57] Y. Zhai, H. Xu, H. Wang, Z. Meng, and H. Huang. Joint routing and sketch configuration in software-defined networking. *IEEE/ACM Transactions on Networking*, 28(5):2092–2105, 2020.

[58] K. Zhao, J. X. Yu, H. Zhang, Q. Li, and Y. Rong. A learned sketch for subgraph counting. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2142–2155, 2021.

[59] R. Zhu, B. Wang, X. Yang, B. Zheng, and G. Wang. Sap: Improving continuous top-k queries over streaming data. *IEEE Transactions on Knowledge and Data Engineering*, 29(6):1310–1328, 2017.