TreeSensing: Linearly Compressing Sketches with Flexibility

Zirui Liu[†], Yixin Zhang[†], Yifan Zhu[†], Ruwen Zhang[†], Tong Yang[†],
Kun Xie[‡], Sha Wang[*], Tao Li[*], Bin Cui[†]
[†] School of Computer Science, Peking University, Beijing, China
[‡]College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
[*]School of Computer, National University of Defense Technology, Changsha, China
[†]{zirui.liu, zrw, bin.cui}@pku.edu.cn, {yxzh, yifan.zhu}@stu.pku.edu.cn, yangtongemail@gmail.com
[‡]xiekun@hnu.edu.cn [*]{ws0623zz, taoli_network}@163.com

ABSTRACT

A *Sketch* is an excellent probabilistic data structure, which records the approximate statistics of data streams. Linear additivity is an important property of sketches. This paper studies how to keep the linear property after sketch compression. Most existing compression methods do not keep the linear property. We propose `TreeSensing`, an accurate, efficient, and flexible framework to linearly compress sketches. In `TreeSensing`, we first separate a sketch into two parts according to counter values. For the sketch with small counters, we propose a technique called *TreeEncoding* to compress it into a hierarchical structure. For the sketch with large counters, we propose a technique called *SketchSensing* to compress it using compressive sensing. We theoretically analyze the accuracy of `TreeSensing`. We use `TreeSensing` to compress 7 sketches and conduct two end-to-end experiments: distributed measurement and distributed machine learning. Experimental results show that `TreeSensing` outperforms prior art on both accuracy and efficiency, which achieves up to 100× smaller error and 5.1× higher speed than state-of-the-art CluserReduce. All related codes are open-sourced.[1]

1 INTRODUCTION

A *Sketch* is an excellent probabilistic data structure [1–5], which records the approximate statistics of data streams by maintaining a summary. Thanks to their fast speed and small memory overhead, sketches are widely used in the fields of database [6–9], data mining [10–13], and networks [14, 15] to perform various tasks, such as frequency estimation [16, 17], finding top-k items [18, 19], distributed data analysis [20], and the acceleration of machine learning [21].

Linear additivity is an important property of sketches. Consider a distributed data analysis scene consisting of n measurement nodes. Each node N_i builds a local sketch S_i using its local data shard D_i, and periodically sends the sketch to a central analyzer. Suppose S is the sketch built from all data shards $D_1, \cdots, D_n$ together. The linear additive property guarantees that $S = \sum_{i=1}^{n} S_i$, where the addition operation between sketches refers to adding up the counters in the same position. For example, suppose $n = 2$, $S_1 = [0, 1, 1, 1, 1]$ and $S_2 = [0, 1, 2, 3, 4]$, then we have $S = [0, 2, 3, 4, 5]$. Under the linear property, the central analyzer can first aggregate the n received sketches into $S = \sum_{i=1}^{n} S_i$ and then use the aggregated sketch S to answer queries, rather than query each of the n sketches .

This paper studies how to keep this linear property after sketch compression. Sketch compression is an important mechanism for many practical scenarios [21, 22]. A compression algorithm could keep the linear additive property: Given a sketch compression algorithm $C(\cdot)$, let $C(S_i)$ denote the compression of sketch S_i. The compression algorithm keeps the linear property if it satisfies $C(\sum_{i=1}^{n} S_i) = \sum_{i=1}^{n} C(S_i)$. It is important for a sketch compression algorithm to keep the linear property. 1) Consider a distributed data analysis scene [23–26] with n nodes and one central analyzer. In each measurement period, each node N_i builds a local sketch S_i, compresses the sketch to $C(S_i)$, and sends $C(S_i)$ to the central analyzer. The analyzer wants to acquire the aggregated sketch $\sum_{i=1}^{n} S_i$ and use it to perform global analysis tasks. If the compression algorithm keeps the linear property, the analyzer just needs to sum up all received data $\sum_{i=1}^{n} C(S_i) = C(\sum_{i=1}^{n} S_i)$, and then perform one recovery operation to directly acquire the aggregated sketch $S = \sum_{i=1}^{n} S_i$. Otherwise, the analyzer must first perform n recovery operations to get $S_1 \cdots S_n$, and then aggregate the n recovered sketches to get $\sum_{i=1}^{n} S_i$, which is very inefficient. 2) Another scene where linear property matters is *Secure Aggregation*, which is a key topic in federated learning [27–29] and cloud computing [30]. Consider a federated learning scene [27] where n workers collaboratively train a model under the coordination of an aggregator. In each training period, each worker N_i computes gradient G_i using its local data. To protect the local gradient G_i[2] against the untrusted aggregator, each worker N_i masks its gradient into $G_i + \varepsilon_i$ and sends the masked gradient to the aggregator. Here, ε_i is a random noise coordinated by all workers to guarantee that $\sum_{i=1}^{n} \varepsilon_i = 0$. The aggregator aggregates all masked gradients to get $G = \sum_{i=1}^{n} (G_i + \varepsilon_i) = \sum_{i=1}^{n} G_i$, and broadcasts the aggregated gradient G to each worker. It is reported that the training time of such system is dominated by the expensive communication overhead [27]. Therefore, it is lucrative to reduce the communication overhead by encoding local gradient G_i into sketch S_i [21] and compressing S_i into $C(S_i)$ to send. In such case, only a linear algorithm can work. If the compression algorithm keeps the linear property, the aggregator can aggregate the received data to get $\sum_{i=1}^{n} (C(S_i) + \varepsilon_i) = \sum_{i=1}^{n} C(S_i) = C(\sum_{i=1}^{n} S_i)$, and broadcast $C(\sum_{i=1}^{n} S_i)$ to each worker. Each worker then recovers the aggregated sketch $S = \sum_{i=1}^{n} S_i$ to get the aggregated gradient. Otherwise, the aggregator cannot aggregate the received data.

Existing methods to compress sketches can be divided into two categories: *linear compression* methods and *non-linear compression* methods. 1) To our knowledge, Hokusai [31] is the only linear method. To compress the sketch by λ times, Hokusai first divides every λ counters into the same group, and then merges the counters in each group by summing them up. 2) Typical non-linear methods

[1] https://github.com/TowerSensing/TowerSensing

[2] Local gradient may leak private information about locally stored data [28].

include Elastic [32] and Cluster-Reduce [22]. The key difference between Elastic and Hokusai is that Elastic merges the counters by taking the maximum value in each group, which compromises the linear property and makes it cannot compress the sketches with negative counters (*e.g.*, the Count sketch [33]). Cluster-Reduce [22] further proposes to first rearrange similar counters into the same group, and then merges each group by taking the maximum value.

In this paper, we propose **`TreeSensing`**, an *accurate, efficient, and flexible framework to linearly compress sketches*. To compress a sketch S, we first separate S into two partial sketches: a sketch $\dot{S}$ with only small counters (called *small sketch*) and a sketch $\ddot{S}$ with only large counters (called *large sketch*). To get the small sketch $\dot{S}$, we set all counters with large values in the full sketch S to zero. Similarly, we get the large sketch $\ddot{S}$ by setting all small counters in S to zero. In other words, $S = \dot{S} + \ddot{S}$. For example, suppose $S = [1, 89, 2, 3, 1]$, then we have $\dot{S} = [1, 0, 2, 3, 1]$ and $\ddot{S} = [0, 89, 0, 0, 0]$. Next, we propose two key techniques, namely *TreeEncoding* and *SketchSensing*, to compress the small sketch and the large sketch, respectively. Both the two techniques keep the linear property. In addition, each of the two techniques can be used alone. For example, in applications that only care about the accuracy of frequent items (*e.g.*, SketchML [21]), we can just use *SketchSensing* to compress the large sketch and discard the small sketch.

The **first key technique** of `TreeSensing`, namely *TreeEncoding*, compresses the small sketch into a hierarchical structure called *HierariacalTree*. This data structure is designed based on our observation that in practice, most counters of a sketch have small values, and only a small fraction of counters have large values. Therefore, it is wasteful to record all of these values using equal-sized (32-bit) counters in a flat structure, like existing sketch compression methods do [22, 31, 32]. *HierarchicalTree* uses smaller-sized (*e.g.*, 4-bit) counters, and organizes them into a shape of tree. One overflowed counter automatically enlarges its size by expanding to the higher layer of the tree and setting an overflow indicator. We further propose a smart technique called ShiftBfEncoder (see § 3.2) to compress the overflow indicator of *HierarchicalTree*. The structure of *HierarchicalTree* is flexible. We can choose to send different layers of *HierarchicalTree* to achieve different level of accuracy. Under limited bandwidth, we can just send the high layer(s) of *HierarchicalTree* to approximately recover the sketch, and an approximately recovered sketch can also provide high accuracy (see § 3.2). Although we can use *TreeEncoding* to encode the entire sketch, we find the limitation of this hierarchical structure: it is unfriendly to frequent items. Specifically, as higher layer has fewer counters, there are more collisions between frequent items in higher layer, which significantly degrades the accuracy of top-k frequent items.

To address the above limitation, we propose the **second key technique** of `TreeSensing`, namely *SketchSensing*. *SketchSensing* employs the theory of *compressive sensing* [34] to compress the large sketch in a nearly lossless way. Compressive sensing is a technique widely used in signal processing field for acquiring and recovering a sparse signal. When original data is sparse, compressive sensing provides near-perfect recovery. Recall that we have the observation that only a small fraction of counters have large values. Thus, after setting small counters to zero, the resulted large sketch will be very sparse. This sparse sketch perfectly suits the requirements of compressive sensing. *SketchSensing* works in a flexible way: it compresses the large sketch into many small fragments (called sensing fragments). The recovery process does not need all sensing fragments. Under limited bandwidth, one can just send a partial collection of sensing fragments to approximately recover the sketch, and an approximately recovered sketch can also provide high accuracy and effective query rate (see § 3.3).

`TreeSensing` has three key advantages: 1) **Accurate recovery:** `TreeSensing` achieves up to 100× smaller error than state-of-the-art Cluster-Reduce [22] (see Figure 9). 2) **Efficient compression:** `TreeSensing` can compress a 13.2MB sketch by 10 times within 69 milliseconds, outperforming Cluster-Reduce by 5.1× (see Figure 12). `TreeSensing` achieves 316MHz throughput on FPGA platforms (see Table 6). 3) **Flexibility:** `TreeSensing` allows users to choose appropriate compression techniques and compression ratio to strike a balance between bandwidth and accuracy. `TreeSensing` can optionally recover the original sketch from a fraction of compressed data, and the partially recovered sketch also provides high accuracy. All related codes are open-sourced anonymously [35].

This paper makes the following key contributions.

- We introduce the linear additive property in sketch compression, and we believe this is an important property.
- We propose `TreeSensing`, a generic framework to linearly compress sketches, which is efficient, accurate, and flexible.
- We theoretically analyze the accuracy of `TreeSensing`.
- We use `TreeSensing` to compress 7 sketches (CM [1], CU [36], Count [33], MinMax [21], CMM [37], CML [38], and CSM [39]), and apply `TreeSensing` to three applications: distributed measurement, distributed ML, and join-aggregate estimation.
- We extensively evaluate the performance of `TreeSensing`, showing it outperforms prior art on both accuracy and efficiency.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce some typical sketches in § 2.1. We summarize existing works of sketch compression in § 2.2. We introduce the fundamental concepts of compressive sensing in § 2.3.

Table 1: Symbols frequently used in this paper.

Symbols	Meaning
d	Number of arrays in a sketch
w	Number of counters in each array of a sketch
$\mathcal{A}_i$	The i^{th} array of a sketch
τ	Separating threshold of `TreeSensing`
l	Number of layers in *HierarchicalTree*
L_i	The i^{th} layer of *HierarchicalTree*
n_i	Number of counters in L_i
δ_i	Each counter in L_i contains δ_i bits
κ_i	κ_i adjacent counters in L_i are with a counter in L_{i+1}
λ	Compressing ratio in *SketchSensing*
r	Rounding parameter in *SketchSensing*
f	Number of fragments in *SketchSensing*
m	Number of counters in each fragment in *SketchSensing*

2.1 Sketches for Data Stream Measurement

A *Sketch* is an excellent probabilistic data structure that can perform various tasks such as frequency estimation, finding frequent items, cardinality estimation, *etc.* Sketch uses a small digest to record approximate information of the data stream. Typical sketches include CM [1], CU [36], Count [33], CSM [39], CML [38], CMM [37], and more [40, 41]. A sketch usually consists of multiple arrays, each of which has many counters. Take the most well-known CM sketch [1]

as an example. A CM sketch consists of d arrays $\mathcal{A}_1, \cdots, \mathcal{A}_d$. Each array $\mathcal{A}_i$ has w counters and is associated with a hash function $h_i(\cdot)$ that maps each item into a counter in it. To insert an item e, it increments each of *the d hashed counters* $\mathcal{A}_1[h_1(e)], \cdots, \mathcal{A}_d[h_d(e)]$ by one. For query, it returns the minimum of the d hashed counters. Other sketches devise more smart techniques to further improve the accuracy and achieve unbiased estimation [33, 36–39, 42].

There is also a line of sketches that propose to use hierarchical structures to fit the highly skewed data distribution [43, 44], such as Counter-Braid [45], Pyramid [46], and Diamond [47]. However, to guarantee that any practical large value can be represented, their structure must have many layers. In addition, these sketches fix their hierarchical structures before insertion, and thus have slow speed because each insertion may access all layers in the worst case. We believe that a better solution is to first use the simple and flat data structure at insertion, and then proactively compress the sketch into a hierarchical structure when requiring to transmit.

2.2 Related Work on Sketch Compression

Until now, there are only a few works on sketch compression, including Hokusai [31], Elastic [32], and Cluster-Reduce [22]. To our knowledge, Hokusai [31] is the only work that keeps the linear property. To compress a sketch $\mathcal{S}$ with w counters by λ times, Hokusai first divide the counters into $\frac{w}{\lambda}$ groups, each of which has λ counters. Next, Hokusai merges the λ counters of each group into one by taking the sum of these counters. Based on Hokusai, Elastic [32] makes small modification by taking the maximum of the counters in each group, which improves the accuracy but compromises the linear property. In addition, this modification makes Elastic cannot compress the sketches with negative counters (*e.g.*, Count sketch [33]). Cluster-Reduce [22] proposes the idea of nearness clustering, which works by rearranging similar counters into the same group. It then uses a technique named unique reducing to get the representative value of each group. Compared with prior solutions, Cluster-Reduce achieves the best accuracy, but it also cannot keep the linear additive property.

Another kind of data compression methods, lossless compression methods, can also be applied to compress sketches. Typical methods include Huffman coding [48], Deflate [49], and more [50, 51]. However, the compression and recovery speed of lossless methods are usually slow because of complicated arithmetic computation. More importantly, all lossless methods do not keep the linear property.

2.3 Preliminary of Compressive Sensing

Compressive sensing [34] is a technique that can efficiently recover sparse signals. Classical compressive sensing incorporates two procedures: *sensing* and *recovery*. 1) Given an original signal $\vec{x}$ of length n, the *sensing* procedure compresses $\vec{x}$ into a compact vector $\vec{y}$ of length m by multiplying a $m \times n$ sensing matrix Φ, *i.e.*, $\vec{y} = \Phi \times \vec{x}$. Common sensing matrices include Gaussian Matrix (GM) [52] and Bernoulli Matrix (BM) [53]. 2) The *recovery* procedure reconstructs $\vec{x}$ using $\vec{y}$ and Φ by solving a linear system: $\sum_{j=1}^{n} \Phi_{i,j} \cdot x_j = y_i$. Under the prior knowledge that $\vec{x}$ is sparse, the problem of solving the above linear system can be transformed into an optimization problem: *minimize:* $\|x\|_1$, *subject to:* $\vec{y} = \Phi \times \vec{x}$. Theoretical analysis guarantees that the solution $\vec{x}'$ is close to $\vec{x}$ as long as $\vec{x}$ is sparse enough [54–56]. Besides L1 optimization [57], other recovery algorithms include Orthogonal Matching Pursuit (OMP) [58], Iteratively Reweighted Least Square (IRLS) [59], and more [60, 61]. In practice, compressive sensing has been widely adopted in the field of signal processing [62] and data transmission [63, 64]. We believe that applying compressive sensing to compress probabilistic data structures is a promising open area.

3 THE TREESENSING ALGORITHM

In this section, we first take the most widely-used CM sketch as an example to explain how `TreeSensing` works. We present the workflow of `TreeSensing` in § 3.1, and describe the two key techniques of `TreeSensing`, namely *TreeEncoding* and *SketchSensing* in § 3.2 and § 3.3. We explain how to compress other sketches in § 3.4.

3.1 TreeSensing Overview

Rationale: We aim at devising an accurate, efficient, and flexible framework to linearly compress sketches. 1) We observe that in practice, the counters of a sketch are highly skewed: most counters have small values and only a small fraction of counters have large values.[3] It is wasteful to record these skewed values using equal-sized counters (32-bit). Thus, we propose the *TreeEncoding* technique to compactly encode the sketch into a hierarchical structure with smaller-sized counters (*e.g.*, 4-bit). Our methodology is to dynamically assign appropriate number of bits for different original counters: larger counters are allocated more bits in multiple layers and smaller counters are allocated fewer bits in one layer. 2) Although we can use *TreeEncoding* to encode the entire sketch, we find it is not suitable for large counters: we must use many layers to record a large counter, which leads to slow compression speed and large memory overhead. In addition, we notice that there are many applications (*e.g.*, distributed ML in § 5.2) that only use large counters. These applications are sensitive to the error of large counters but indifferent to small counters. It is desired to devise another method that specifically compresses large counters in a nearly-lossless way. Recall that only a small fraction of counters have large values. If we neglect small counters and only consider the large counters, the sketch will become a sparse array. This sparsity well suits the requirement of compressive sensing [34]. Thus, we further propose *SketchSensing* technique, which uses compressive sensing to compress large counters in a nearly lossless way.

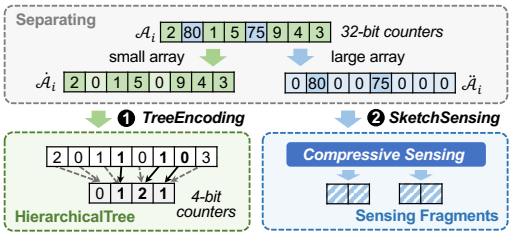


Figure 1: `TreeSensing` Overview.

As shown in Figure 1, in `TreeSensing`, we propose two techniques: *TreeEncoding* and *SketchSensing*. For each array $\mathcal{A}_i$ in a CM sketch, we first use a threshold τ (named separating threshold) to separate it into two partial arrays: an array $\dot{\mathcal{A}}_i$ with only small counters (called *small array*) and an array $\ddot{\mathcal{A}}_i$ with only large counters (called *large array*). Specifically, for each counter in $\mathcal{A}_i$,

[3] This phenomenon stems from the unbalanced data distribution in most real-world applications. For example, in network stream, most flows are small flows and a small fraction of large flows contribute to most traffic [43].

we set it to zero if it is larger than τ, and the resulted array is the small array $\dot{\mathcal{A}}_i$. Similarly, we can get the large array $\ddot{\mathcal{A}}_i$. Obviously, $\mathcal{A}_i = \dot{\mathcal{A}}_i + \ddot{\mathcal{A}}_i$. For the small array, `TreeSensing` uses the *TreeEncoding* technique to compactly encode it into a hierarchical data structure called *HierarchicalTree*. Each counter in *HierarchicalTree* has smaller size than the counters in original sketch. For the large array, which is sparse, `TreeSensing` uses the *SketchSensing* technique to compress it into many small fragments, named *sensing fragments*. Both *TreeEncoding* and *SketchSensing* keep the linear property. In addition, *TreeEncoding* and *SketchSensing* are orthogonal, each of which can be used alone. Note that both *TreeEncoding* and *SketchSensing* do not guarantee exact recovery, but under good choice of parameters, they can achieve nearly lossless recovery.

3.2 The TreeEncoding Algorithm

Overview: *TreeEncoding* uses a structure called *HierarchicalTree* to compactly record the small array. *HierarchicalTree* organizes small-sized counters (*e.g.*, 4-bit) into a tree shape. Once a counter overflows, it automatically enlarges its size by expanding into a counter in higher layer, and set a corresponding 1-bit overflow indicator to *true*. We propose a technique called ShiftBfEncoder to further compress the overflow indicators. Since *HierarchicalTree* only encodes counters with small values, it does not need to use many layers. In practice, two or three layers suffice for good performance. For recovery, we can either use the entire *HierarchicalTree* to recover the small array at best effort, or use the high layer(s) of *HierarchicalTree* to approximately recover the small array.

Data structure: As shown in Figure 2, the data structure of *HierarchicalTree* consists of l layers: $L_1, \cdots, L_l$. Each counter in L_i contains δ_i bits. Two adjacent layers L_i and L_{i+1} are associated in the following way: κ_i adjacent counters in L_i are associated with one counter in L_{i+1}. For example, $L_i[0], \cdots, L_i[\kappa_i - 1]$ are associated with $L_{i+1}[0]$. We call $L_{i+1}[0]$ the *parent counter* of $L_i[0], \cdots, L_i[\kappa_i - 1]$. Each layer L_i has n_i counters, where $n_{i+1} = \lceil n_i / \kappa_i \rceil$. In addition, each counter $L_i[j]$ (except the counters in the highest layer) uses a 1-bit indicator $I_i[j]$ to record whether it overflows.

Compression: To compress a small array $\dot{\mathcal{A}}_i$ of w counters, we build an empty *HierarchicalTree* that has $n_1 = w$ counters in the first layer. Then we sequentially insert each counter in $\dot{\mathcal{A}}_i$ into the *HierarchicalTree*. To insert the j^{th} counter $\dot{\mathcal{A}}_i[j]$, we increment $L_1[j]$ by $\dot{\mathcal{A}}_i[j]$. If $L_1[j]$ overflows, *i.e.*, $\dot{\mathcal{A}}_i[j] \geqslant 2^{\delta_1}$, we perform the following carry-in operation: we set the overflowed counter $L_1[j]$ to $\dot{\mathcal{A}}_i[j] \% 2^{\delta_1}$, set its 1-bit indicator $I_1[j]$ to *true*, and then increment its parent counter by $\lfloor \dot{\mathcal{A}}_i[j] / 2^{\delta_1} \rfloor$. In this way, the parent counter merges its child counters by summing up their overflowed values, which is a lossy process. If the parent counter does not overflow, the carry-in operation ends; otherwise, we repeat the carry-in operation on the parent counter. In our implementation, we use SIMD to accelerate the compression procedure (see § 6.3).

Best recovery: To recover a small array at best effort (using the entire *HierarchicalTree*), we first build an auxiliary *HierarchicalTree*. The auxiliary *HierarchicalTree* has the same shape as the compressed *HierarchicalTree*, but each of its counter contains *32 bits*. We first set the l^{th} layer of the auxiliary *HierarchicalTree* to L_l, *i.e.*, set $L'_l = L_l$. Next, we recover the counters of the auxiliary *HierarchicalTree* layer by layer from L'_{l-1} to L'_1. To recover the j^{th} counter in L'_i, *i.e.*, $L'_i[j]$, we first set $L'_i[j]$ to $L_i[j]$. Then we check the 1-bit indicator of $L_i[j]$ to see whether it overflows. If so, we add $L'_i[j]$ by $L'_{i+1}[\lfloor j/\kappa_i \rfloor] \times 2^{\delta_i}$, where $L'_{i+1}[\lfloor j/\kappa_i \rfloor]$ is the parent counter of $L'_i[j]$. The recovery process is repeated layer by layer. Finally, we get the recovered small array, which is L'_1.

Approximate recovery: The recovery procedure of *TreeEncoding* can be flexible. We can just use high layer(s) of *HierarchicalTree* to approximately recover the small array. To recover from a partial *HierarchicalTree*, we just perform the basic recovery process above, and treat the values of the counters in the missing layers as zero.

Discussion: *TreeEncoding* cannot guarantee exact recovery, but under good choice of parameters, it can achieve nearly lossless recovery. For each parent counter, we define it as a *pure counter* if it records the overflowed value of only one child counter. We can see that only when all parent counters are pure can *TreeEncoding* achieve exact recovery. This is because each non-pure counter merges the overflowed values of child counters by summing them up, which is a lossy process. For 2-layer *HierarchicalTrees* ($l = 2$), we can modify the merging method to take the maximum among all overflowed values, so as to reduce the error. We will empirically analyze the compression error of *TreeEncoding* in Table 3.

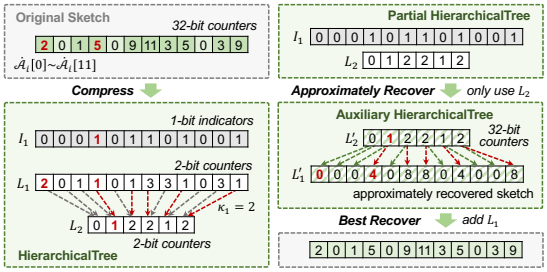


Figure 2: Example of the TreeEncoding Algorithm.

Example (Figure 2): We use an example to show how to use *TreeEncoding* to compress and recover a small array. We use a *HierarchicalTree* consisting of $l = 2$ layers, both of which use 2-bit counters ($\delta_1 = \delta_2 = 2$), and we set $\kappa_1 = 2$. **1) Compression:** To compress the small array $\dot{\mathcal{A}}_i$, we sequentially insert each of its counter into the empty *HierarchicalTree*. Specifically, to insert $\dot{\mathcal{A}}_i[0] = 2$, we increment $L_1[0]$ by 2. To insert $\dot{\mathcal{A}}_i[3] = 5$, we increment $L_1[3]$ by 5. Since $L_1[3]$ overflows, we perform the carry-in operation: we set $L_1[3] = 5\%4 = 1$, set the 1-bit indicator of $L_1[3]$ to *true*, and increment the parent counter $L_2[1]$ by $\lfloor 5/4 \rfloor = 1$. **2) Approximate recovery:** We use L_2 and the 1-bit indicator of L_1 to approximately recover the small array. First, we build an auxiliary *HierarchicalTree* with 32-bit counters, and set its second layer L'_2 to L_2. Then we sequentially recover each counter in L'_1. Specifically, to recover $L'_1[0]$, we first set it to $L_1[0]$. As L_1 is missing, we set $L'_1[0] = 0$. As $L_1[0]$ does not overflow, its recovered value is 0. To recover $L'_1[3]$, we first set $L'_1[3] = 0$. As $L_1[3]$ overflows, we add $L'_1[3]$ by $L'_2[1] \times 4$ to get $L'_1[3] = 4$, where $L'_2[1]$ is the parent counter. Finally, we get the approximately recovered array (L'_1). **3) Best recovery:** We can use the entire *HierarchicalTree* to recover the array at best effort. In this example, if we add L_1 to L'_1, the recovered array will be exactly the original array.

[Optional] ShiftBfEncoder optimization (Figure 3): We propose a smart technique, namely *ShiftBfEncoder* (Shifting Bloom filter Encoder), to compress the 1-bit indicators by >10× (see Figure 6(d)). We notice that in practice, only a few counters overflow, so

the 1-bit indicators are very sparse. As shown in Figure 3, ShiftBfEncoder is also a bit string. To encode a 1-bit indicator, we first divide the indicator into many consecutive groups, each of which contains x bits ($x = 4$ in our example). For each group, we hash it to γ positions in the ShiftBfEncoder, which is implemented by hashing the group ID γ times ($\gamma = 2$ in our example). We insert this group into ShiftBfEncoder by performing bitwise OR operation in the γ positions. To recover a 1-bit indicator, we query each group in ShiftBfEncoder: We recover this group by performing bitwise AND operation on its γ hashed values. As shown in Figure 3, to encode a group 0100, we locate its two hashed positions 0010 and 0000, and perform bitwise OR operation on them: 0100|0010 = 0110 and 0100|0000 = 0100. To recover this group, we perform bitwise AND operation on its two hashed positions to get 0110&0100 = 0100.

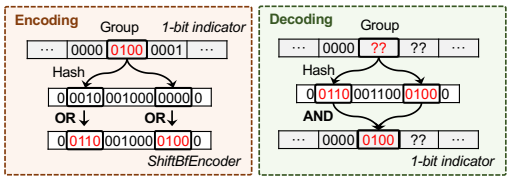


Figure 3: Example of the ShiftBfEncoder algorithm.

ShiftBfEncoder can be further optimized. Notice that ShiftBfEncoder has false positive error incurred by hash collisions: Consider a given group with true value 0000, if the values of its two hashed positions are 1010 and 1000, its recovered value will be 1010&1000 = 1000, where an error occurs at the first bit. We can reduce this error by adding more hashes, but this will reduce the sparsity of ShiftBfEncoder, degrading overall accuracy. We observe that an error that occurs at higher layer of *HierarchicalTree* has more serious impact than an error occurs at lower layer, because higher layer records higher significant bits of the original counters. Thus, we propose to use more hashes for the groups in higher layer, and use fewer hashes for the groups in lower layer. In our implementation, each group in the i^{th} layer uses i+1 hashed positions. In this way, the groups in higher layer will have fewer errors.

Linear property: *TreeEncoding* keeps the linear property. In our implementation, instead of building d *HierarchicalTree*s with $n_1 = w$ to compress each array $\dot{\mathcal{A}}_i$, we build one *HierarchicalTree* with $n_1 = d \times w$ to compress the d arrays $\dot{\mathcal{A}}_1, \cdots, \dot{\mathcal{A}}_d$. Let $C_T(S)$ be the *HierarchicalTree* of sketch S. To aggregate two *HierarchicalTree*s $C_T(S_1)$ and $C_T(S_2)$, we sum up every two counters in the same position, during which if a resulted counter overflows, we perform the carry-in operation to update the parent counter and set the indicator/ShiftBfEncoder. We merge the indicators/ShiftBfEncoders of $C_T(S_1)$ and $C_T(S_2)$ by performing bitwise OR operation. The resulted *HierarchicalTree* $C_T(S_1) + C_T(S_2)$ is exactly the *HierarchicalTree* built from $S_1 + S_2$, namely $C_T(S_1) + C_T(S_2) = C_T(S_1 + S_2)$.

3.3 The SketchSensing Algorithm

Compression: To compress a large array $\ddot{\mathcal{A}}_i$ of w counters, we first round all counters down to the multiple of one predefined parameter r (called rounding parameter). For example, if $r = 4$, a counter with the value of 69 will be rounded to 68. This process reduces the information of the original array. Next, we evenly split $\ddot{\mathcal{A}}_i$ into f fragments (called *original fragments*), each of which has $m = \frac{w}{f}$ counters. We generate the sensing matrix ϕ in compressive sensing from a random seed. Given a predefined compressing ratio λ, the size of ϕ should be $m \times m'$ where $m' = \frac{m}{\lambda}$. Finally, we compress the f original fragments into f smaller fragments (called *sensing fragments*) by multiplying each original fragment with ϕ. In this way, each original fragment is compressed by λ times, and thus the original array is compressed by λ times.

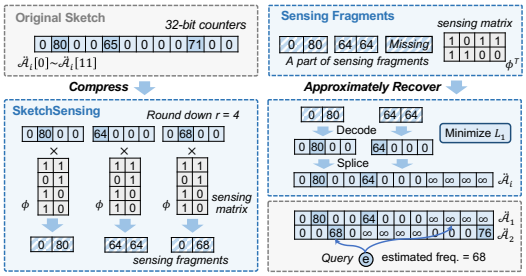


Figure 4: Example of the SketchSensing Algorithm.

Recovery: To recover a large array, we first decode each original fragment from the sensing fragment and sensing matrix. As described in § 2.3, this can be done by solving an L_1 optimization problem. Compressive sensing guarantees that the original fragment can be perfectly recovered if it is sparse enough. Finally, we assemble the decoded fragments into a complete array.

Approximate recovery: The recovery procedure of *SketchSensing* can be flexible. We can just use a partial set of sensing fragments to approximately recover a sketch. An approximately recovered sketch consisting of d incomplete arrays can also answer queries: For each query, it accesses d hashed counters, some of which might be missing. We consider the value of the missing counters as invalid, and report the minimum value among other valid counters. As long as one of the d hashed counters is valid, the recovered sketch can report a valid result. Our theoretical and experimental results show that a sketch recovered from 70% sensing fragments can report 95% valid results and provide accurate estimation (ARE < 0.1).

Example (Figure 4): We use an example to show how to use *SketchSensing* to compress and approximately recover a large array, and how to query an approximately recovered sketch. We set $r = 4$, $f = 3$, and $\lambda = 2$. **1) Compression:** To compress the large array $\ddot{\mathcal{A}}_i$, we first round all of its counters down to the multiple of the rounding parameters ($r = 4$), and then evenly split $\ddot{\mathcal{A}}_i$ into $f = 3$ original fragments, each of which has $\frac{n}{f} = 4$ counters. We generate a 4×2 sensing matrix ϕ, and multiply each fragment with ϕ to get the sensing fragments. **2) Approximate recovery:** We use the first two sensing fragments to approximately recover the large array. We decode each original fragment by solving L_1 optimization, and assemble the decoded fragments into one array. As the third sensing fragment is missing, we treat the counters in the third original fragment as invalid. **3) Query an approximately recovered sketch:** Suppose we have an approximately recovered sketch with two arrays. To query item e, we locate its two hashed counters, where the first counter is invalid, so we return the value of the second hashed counter 68 as the estimated frequency.

Linear property: *SketchSensing* keeps the linear property. Let $C_S(S)$ be the concatenation of all sensing fragments of sketch S. Since the compression operation of *SketchSensing* is just matrix multiplication, which satisfies the distributive law. Therefore, we naturally have $C_S(S_1) + C_S(S_2) = C_S(S_1 + S_2)$.

3.4 Compressing Other Sketches

In addition to CM, TreeSensing can also compress other sketches, such as CU [36], Count [45], CMM [37], CML [38], CSM [39], and *etc.* We briefly explain how to use TreeSensing to compress them.

Count [33]: Count sketch can provide unbiased estimation, which uses the same data structure as CM. Besides, it has d hash functions $s_1(\cdot), \cdots, s_d(\cdot)$, each of which maps an item to +1 or −1 uniformly. To insert item e, Count adds each hashed counter $\mathcal{A}_i[h_i(e)]$ by $s_i(e)$. To compress the Count sketch, we modify the separating strategy by separating counters according to their absolute values. In *TreeEncoding*, a counter of *HierarchicalTree* can overflow positively or negatively, in which cases we just perform the carry-in operation by adding or decreasing its parent counter. In *SketchSensing*, we modify the rounding down operation to round each counter towards zero. For example, if the rounding parameter $r = 4$, a counter with the value of −63 will be rounded to −60. The other procedures of TreeSensing remain unchanged.

CU [36], CMM [37], CML [38], and CSM [39]: These sketches devise many smart techniques to improve accuracy. They all use the same data structure as CM. For example, CU sketch introduces a conservative update (CU) strategy by only incrementing the smallest counter(s) in insertion. We can directly use TreeSensing to compress these sketches, where we use smaller τ and r because the counters in these sketches are generally smaller than in CM.

4 MATHEMATICAL ANALYSIS

4.1 Analyses of TreeEncoding

We first derive the correct rate of a CM sketch recovered using *TreeEncoding* in Theorem 4.1. Then we derive the error bound of a recovered CM sketch in Theorem 4.2.

Consider a data stream with N distinct items $e_1, \cdots, e_N$. Consider the j^{th} array of a CM sketch and its *HierarchicalTree* T_j, which is a sub-part of the entire *HierarchicalTree*. Let $\theta_{i,j}$ be the ratio of distinct items whose carry-in operations end at the i^{th} layer in T_j. We have $\sum_{i=1}^{l} \theta_{i,j} = 1$. Note that for two different arrays of a CM sketch: $\mathcal{A}_{j_1}$ and $\mathcal{A}_{j_2}$, θ_{i,j_1} might not be equal to θ_{i,j_2}. This is because for an item, where its carry-in operation stops will also depend on the counts of other distinct items hashed to the same or nearby counters. In other words, $\theta_{i,j}$ is decided by not only the data distribution, but also the hash function $h_j(\cdot)$ of the j^{th} array.

Theorem 4.1. *The correctness rate of the estimation for an arbitrary item satisfies* $C = 1 - \prod_{j=1}^{d}(1 - \mathcal{P}_j)$, *where* $\mathcal{P}_j = \sum_{i=1}^{l}\left(\theta_{i,j} \cdot \prod_{k=1}^{i} P_{k,j}\right)$ *and* $P_{i,j} = \left(1 - \frac{1}{n_i}\right)^{N\sum_{k=i}^{l}\theta_{k,j}-1}$.

Proof. *Let $P_{i,j}$ denote the probability that one arbitrary counter at the i^{th} layer of the HierarchicalTree T_j records the exact value, i.e., no hash collisions occur in this counter. Note that only the items whose carry-in operations end at the i^{th} layer or above the i^{th} layer can have hash collisions with this counter. The number of items whose carry-in operations reach the i^{th} layer is $N \cdot \sum_{k=i}^{l}\theta_{k,j}$. As each of these items is uniformly hashed into one of the n_i counters in the i^{th} layer, we have $P_{i,j} = \left(1 - \frac{1}{n_i}\right)^{N\sum_{k=i}^{l}\theta_{k,j}-1}$.*

Let $\mathcal{P}_j$ denote the expectation of the probability that an arbitrary counter in the j^{th} array of the recovered CM sketch reports the exact frequency of one item. For an item whose carry-in operation ends at the i^{th} layer, its reported frequency is exact iff no hash collisions occur in its counters in layer $L_1, \cdots, L_i$. Thus, we have $\mathcal{P}_j = \sum_{i=1}^{l}\left(\theta_{i,j} \cdot \prod_{k=1}^{i} P_{k,j}\right)$.

Let C denote the correctness rate of the estimation for one arbitrary item. Since this item has an error iff there are collisions in all of its d hashed counters, we have $C = 1 - \prod_{j=1}^{d}(1 - \mathcal{P}_j)$. □

Theorem 4.2. *Given an item e_k, its estimated frequency $\widehat{f_k}$ reported by a recovered HierarchicalTree has the following error bound:*

$$\Pr\left\{\left|\widehat{f_k} - f_k\right| \leqslant \epsilon V\right\} \geqslant 1 - \prod_{j=1}^{d}\left(\frac{1}{\epsilon}\sum_{i=1}^{l}\frac{\theta_{i,j}}{n_i}\right)$$

where ϵ is a small variable, f_k is the real frequency of e_k, and V is the size of the data stream, i.e., $V = \sum_{u=1}^{N} f_u$.

Proof. For *HierarchicalTree* T_j, each of its layers L_i can be considered to correspond with one *virtual hash function* $h_j^i(\cdot)(1 \leqslant h_j^i(\cdot) \leqslant n_i)$ that maps item into a counter in L_i, which is determined by the initial hash function $h_j(\cdot)$ and the carry-in operations.

Let $I_{i,j,k,u}$ be an indicator variable, which is 1 if $h_j^i(e_k) = h_j^i(e_u)$. Due to the pairwise independent property of hash functions, we have $\mathbb{E}\left(I_{i,j,k,u}\right) = \frac{1}{n_i}$.

Then we define a non-negative variable $X_{k,j}$ as follows:

$$X_{k,j} = \sum_{i=1}^{l}\left[\theta_{i,j} \cdot \sum_{u=1}^{N}\left(f_u \cdot I_{i,j,k,u}\right)\right]$$

Note that if there is a collision between e_k and e_u in some layer, then the overestimation error of $\widehat{f_k}$ caused by e_u because of this collision is at most f_u. Therefore, $X_{k,j}$ reflects an upper bound of the expectation of the error caused by collisions happening at all the layers when querying an arbitrary counter in the recovered sketch. Then we have $\widehat{f_k} \leqslant f_k + \min_{j=1}^{d}\left(X_{k,j}\right)$.

The expectation of $X_{k,j}$ can be calculated as follows:

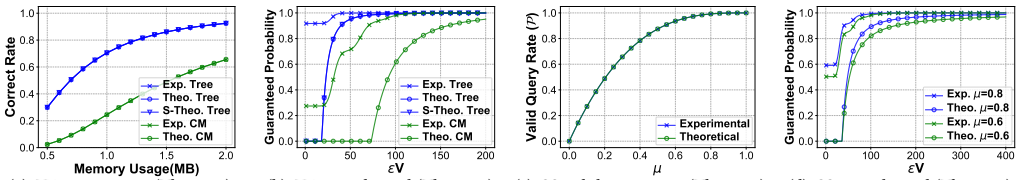
$$\begin{aligned}\mathbb{E}\left(X_{k,j}\right) &= \sum_{i=1}^{l}\left[\theta_{i,j} \cdot \sum_{u=1}^{N}\left(f_u \cdot \mathbb{E}\left(I_{i,j,k,u}\right)\right)\right] \\ &= \sum_{i=1}^{l}\left[\theta_{i,j} \cdot \sum_{u=1}^{N} f_u \cdot \frac{1}{n_i}\right] = \sum_{i=1}^{l}\left[\frac{\theta_{i,j}}{n_i} \cdot V\right] = V \cdot \sum_{i=1}^{l}\frac{\theta_{i,j}}{n_i}\end{aligned}$$

According to the Markov inequality, we have:

$$\begin{aligned}&\Pr\left\{\left|\widehat{f_k} - f_k\right| \geqslant \epsilon V\right\} \leqslant \Pr\left\{\left(\min_{j=1}^{d} X_{k,j}\right) \geqslant \epsilon V\right\} \\ &= \prod_{j=1}^{d}\Pr\left\{X_{k,j} \geqslant \epsilon V\right\} \leqslant \prod_{j=1}^{d}\left[\frac{1}{\epsilon V} \cdot \mathbb{E}\left(X_{k,j}\right)\right] = \prod_{j=1}^{d}\left(\frac{1}{\epsilon}\sum_{i=1}^{l}\frac{\theta_{i,j}}{n_i}\right)\end{aligned}$$

□

Discussion: 1) In Theorem 4.1 and Theorem 4.2, $\theta_{i,j}$ is jointly determined by data distribution and hash functions. We can simplify the form of $\theta_{i,j}$ by ignoring the effect of hash collisions. Specifically, given an arbitrary item e_u, suppose its frequency satisfies that



(a) TE correct rate (Thm. 4.1) (b) TE error bound (Thm. 4.2) (c) SS valid query rate (Thm. 4.3) (d) SS error bound (Thm. 4.4)

Figure 5: Comparison of experimental and theoretical results ("S-Theo." refers to the simplified theorems using $\hat{\theta}_i$).

$2^{\sum_{i=1}^{l_u-1}\delta_i} \leqslant f_u < 2^{\sum_{i=1}^{l_u}\delta_i}$. Then we assume all the d carry-in operations of e_u ends at the l_u^{th} layer. In this way, for $\forall i \in [1, l]$, we have $\theta_{i,1} = \cdots = \theta_{i,d} = \hat{\theta}_i$, where $\hat{\theta}_i$ is the ratio of distinct items whose carry-in operations end at the i^{th} layer. Finally, we can simplify the two theorems by replacing $\theta_{i,j}$ with $\hat{\theta}_i$. 2) To get the theoretical results of non-compressed CM sketch, we can set $l = 1$ (in such case $\theta_{1,j} = 1$ for $\forall j$) in Theorem 4.1 and Theorem 4.2. Afterwards, the two theorems will degenerate into the corresponding correct rate and error bound of the non-compressed CM sketch.

Experimental analysis (Figure 5(a)-5(b)): We conduct experiments to validate Theorem 4.1 and Theorem 4.2, and compare *HierarchicalTree* with non-compressed CM sketch. To evaluate the worst-case performance of *TreeEncoding*, we use Zipf dataset with the skewness of 0 (see details in § 6.1). We set $d = 3$ and $l = 3$. Figure 5(a) shows the experimental and theoretical correct rate of *HierarchicalTree* and CM sketch. We can see that the theoretical results are highly consistent with the experimental results for both *HierarchicalTree* and non-compressed CM, and *HierarchicalTree* always outperforms CM experimentally and theoretically. Figure 5(b) shows the guaranteed probability (error bound). We can see that when $\epsilon V > 30$, the experimental results of *HierarchicalTree* are close to the theoretical results, and *HierarchicalTree also always outperforms CM experimentally and theoretically.* In addition, from Figure 5(a)-5(b), we observe that the simplified theorems of *TreeEncoding* are almost the same as the original theorems, meaning that our assumption on $\hat{\theta}_i$ is reasonable.

4.2 Analyses of SketchSensing

We derive the valid query rate of the approximate recovery process of *SketchSensing* in Theorem 4.3, and derive the error bound of an approximately recovered CM sketch in Theorem 4.4.

Theorem 4.3. *After decoding sensing fragments with a ratio of μ, SketchSensing can report valid results for an arbitrary item e_k with a ratio of $\mathcal{P} = 1-(1-\mu)^d$, where d is the number of arrays in a sketch.*

Proof. Let $I_i, \cdots, I_d$ be d indicating variables where $I_i = 1$ if the i^{th} hashed counter of e_k $\mathcal{A}_i[h_i(e_k)]$ is valid. The approximately recovered sketch reports valid result for e_k if and only if at least one hashed counter is valid. Therefore, we have

$$\mathcal{P} = \Pr\left\{\bigvee_{i=1}^{d}(I_i = 1)\right\} = 1 - \prod_{i=1}^{d}\Pr\{I_i = 0\} = 1-(1-\mu)^d$$

□

Theorem 4.4. *For an approximately recovered CM sketch using sensing fragments with a ratio of μ, when it reports a valid result for item e_k, the estimated frequency $\widehat{f_k}$ has the following error bound:*

$$\Pr\left\{\left|\widehat{f_k} - f_k\right| \leqslant \epsilon V\right\} \geqslant 1 - \frac{(\frac{\mu}{w\epsilon}+1-\mu)^d - (1-\mu)^d}{1-(1-\mu)^d},$$

where f_k is the real frequency of e_k (suppose $f_k > \tau$), V is the size of the data stream, i.e., the number of items in the data stream, and w is the number of counters in each array of the CM sketch.

Proof. Let $\overline{\mathcal{A}_i}[h_i(e_k)]$ be the value of the i^{th} hashed counter of e_k before performing the rounding down operation. The expected number of items mapped to the i^{th} hashed counter is at most $f_k + \frac{V}{w}$. Thus, we have $\mathbb{E}(\mathcal{A}_i[h_i(e_k)]) \leqslant \mathbb{E}\left(\overline{\mathcal{A}_i}[h_i(e_k)]\right) \leqslant f_k + \frac{V}{w}$.

According to the Markov inequality, we have

$$\Pr\{|\mathcal{A}_i[h_i(e_k)] - f_k| \geqslant \epsilon V\} \leqslant \frac{\mathbb{E}(\mathcal{A}_i[h_i(e_k)]) - f_k}{\epsilon V} \leqslant \frac{1}{\epsilon w}$$

Next, according to the law of total probability and the Binomial theorem, we have

$$\Pr\left\{\left|\widehat{f_k} - f_k\right| \geqslant \epsilon V\right\} = \sum_{i=1}^{d}\Pr\{\psi_i\}\cdot\Pr\left\{\left|\widehat{f_k} - f_k\right| \geqslant \epsilon V \mid \psi_i\right\}$$

$$< \sum_{i=1}^{d}\frac{\binom{d}{i}\mu^i(1-\mu)^{d-i}}{1-(1-\mu)^d}\cdot\left(\frac{1}{\epsilon w}\right)^i = \frac{(\frac{\mu}{w\epsilon}+1-\mu)^d - (1-\mu)^d}{1-(1-\mu)^d}$$

where ψ_i denotes the event that there are i valid counters among the d hashed counters of e_k. □

Experimental analysis (Figure 5(c)-5(d)): We conduct experiments to validate Theorem 4.3 and Theorem 4.4. We use the Zipf dataset with the skewness of 0 and set $d = 3$ (see details in § 6.1). Figure 5(c) shows the experimental and theoretical valid query rate of *SketchSensing* under different fragment decoding ratio μ. We can see that the theoretical results are highly consistent with the experimental results. Figure 5(d) shows the guaranteed probability (error bound). We can see that higher decoding ratio μ goes with higher guaranteed probability, and when $\epsilon V > 55$, the experimental results of *SketchSensing* are close to the theoretical results.

5 APPLICATIONS

5.1 Distributed Measurement

Consider a distributed measurement system with n measurement nodes and one central analyzer. In each measurement period, each node N_i builds a local sketch S_i. At the end of each measurement period, each node N_i uses `TreeSensing` to compress its local sketch to $C(S_i)$, and then sends the compressed sketch to the analyzer. The analyzer aggregates the n compressed sketches into one $\sum_{i=1}^{n} C(S_i)$. As `TreeSensing` keeps the linear property, we have $\sum_{i=1}^{n} C(S_i) = C(\sum_{i=1}^{n} S_i)$. Afterwards, the analyzer performs once recovery operation to get $\sum_{i=1}^{n} S_i$, and uses this aggregated sketch to answer queries. In this scenario, we can flexibly choose to use the two components of `TreeSensing`. For applications that only care about the accuracy of frequent items, we can only use *SketchSensing*. We can also dynamically choose appropriate compression ratio of

SketchSensing and number of transmitted layers of *TreeEncoding* to strike a balance between bandwidth and accuracy.

Privacy-preserving distributed measurement: In the above scenario, an attacker inside the network may eavesdrop on the compressed sketches, making the measurement process not secure. This problem is significantly severe in mobile scenarios such as wireless sensor network (WSN) data aggregation [65, 66], smart meter data collection [67, 68], and mobile users data collection [27, 69] because of the open wireless medium. Next, we leverage the linear property to combine `TreeSensing` with homomorphic encryption to achieve efficient privacy-preserving distributed measurement. Homomorphic encryption is a kind of encryption method that allows operations on plaintext to be performed by operating on corresponding ciphertext: Let $E(x)$ denote the encryption of the message x using a public-key. $E(x)$ satisfies that $E(x+y) = E(x) \cdot E(y)$. Typical homomorphic cryptosystems include Benaloh [70], Paillier [71], and more [72]. In our design, each node first encrypts the compressed sketch $C(S_i)$ into $E(C(S_i))$ using a public-key, and then sends the encrypted message to the central analyzer. To guarantee efficiency, we only encrypt a small part of the compressed sketch, *e.g.*, several counters in each sensing fragment [4], or some counters in high layers of *HierarchicalTree*. Note that we do not encrypt the ShiftBfEncoder. The central analyzer multiplies the n received messages to get $\prod_{i=1}^{n} E(C(S_i))$, and then decrypts this ciphertext using the secret-key and recovers the aggregated sketch. From the properties of homomorphic encryption, we have $\prod_{i=1}^{n} E(C(S_i)) = E(\sum_{i=1}^{n} C(S_i))$. As `TreeSensing` keeps the linear property, we have $\sum_{i=1}^{n} C(S_i) = C(\sum_{i=1}^{n} S_i)$. And thus, we have $\prod_{i=1}^{n} E(C(S_i)) = E(C(\sum_{i=1}^{n} S_i))$. Therefore, the central analyzer just performs one decryption and one recovery operation.

5.2 Distributed Machine Learning

Typical distributed machine learning (DML) system consists of multiple workers and one parameter server. Each worker proposes gradient based on its own data shard, and then the parameter server aggregates gradients from all workers and broadcasts model update. To speedup DML, people try to compress the gradients using some low-precision methods, so as to reduce the communication time [21, 73–79]. SketchML [21] uses a Quantile sketch [80] to encode each gradient value into an integer within a range ($[0, max\_value]$), and uses a MinMax sketch to compactly record these integers. With `TreeSensing`, we can build a larger MinMax sketch on each worker, and then compress it into small size to send. In real-world DML systems, we have the following two observations: 1) Most counters of a MinMax sketch are either unused or have small values. This is because the gradient values usually conform to a nonuniform distribution, where most of the values are close to 0 [21]. 2) Small gradients are less important than large gradients for model convergence, so small counters in MinMax sketch are less important than large counters. Based on the two observations, we only use *SketchSensing* to compress the large sketch, and discard the small counters. Before compression, we not only set the counters with small values to *zero*, but also set the unused counters to *zero*. This is because the unused counters are not hashed by any key, and thus will never be accessed in query, so we can simply ignore them to improve sparsity.

[4] A sensing fragment cannot be decoded if some of its counters are missing.

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

Platform: We conduct experiments on a CPU platform (§ 6.2-§ 6.6) and an FPGA platform (§ 6.7). The CPU platform is an 18-core 4.2GHz CPU server (Intel i9-10980XE) with 128GB 3200MHz DDR4 memory and 24.75MB L3 cache. The FPGA platform (Virtex-7 VC709) is integrated with XC7VX690T-2FFG1761I [5] with 433200 Slice LUTs, 866400 Slice Register, 1470 Block RAM Tile (*i.e.*, 52.9Mb on-chip memory), 850 Bonded IOB, and 32 BUFGCTRL.

Implementation and settings: On CPU, we implement `TreeSensing` with C++ and build it with g++ 7.5.0. We use 32-bit Murmur Hash [82] and BOB Hash [83] with different seeds. For *TreeEncoding*, we list three recommended shapes of *HierarchicalTree* in Table 2. We also implement three existing sketch compression algorithms: Cluster-Reduce [22], Hokusai [31], and Elastic [32]. We apply `TreeSensing` and existing algorithms on six sketches: CM [1], CU [36], Count [33], CMM [37], CML [38], and CSM [39]. In addition, we use C++ to simulate a distributed machine learning system and deploy SketchML [21] in it, where we use `TreeSensing` and Cluster-Reduce [22] to compress the MinMax sketch [21].

Table 2: Parameters of three shapes of *HierarchicalTree*.

Shape	l	δ_1	δ_2	δ_3	κ_1	κ_2	SIMD	Ratio
#1	2	4	8	–	4	–	×	5.0
#2	3	4	4	4	2	2	×	5.0
#3	2	8	16	–	8	–	✓	3.0

Datasets: For experiments on data streams, we use two real-world datasets and one synthetic dataset. ***1) CAIDA [84]:*** IP trace datasets collected on backbone links by CAIDA 2018. We use two traces of different sizes: a small-scale 1-minute trace containing about 30M items, and a large-scale 1-hour trace containing about 1.5G items. ***2) Criteo [85]:*** An online advertising click data stream consisting of about 45M ad impressions. ***3) Zipf:*** We use Web Polygraph [86] to generate multiple datasets according to Zipf distribution [44] with different skewness. Each dataset has 32M items. For experiments on machine learning (§ 6.6), we use the ***Twin gas sensor arrays dataset*** download from UCI Machine Learning Repository [87], which contains 640 instances and 10^5 features.

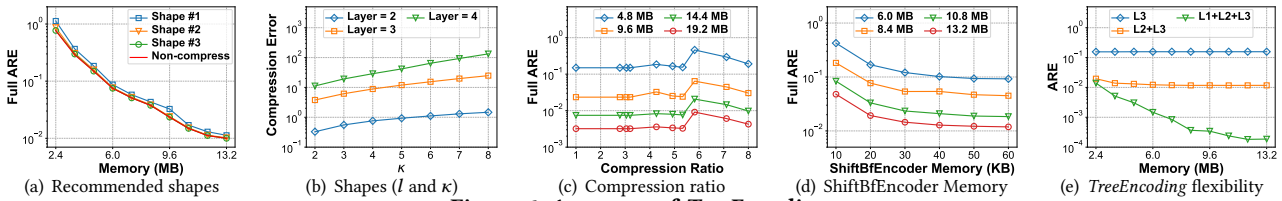
Metrics: 1) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{e_i \in \Psi} |f_i - \hat{f}_i|/f_i$, where f_i is the real frequency of item e_i, $\hat{f}_i$ is the estimated frequency, and Ψ is the query set. For *Top-k ARE*, Ψ consists of the top-k frequent items ($k = 500$ by default). For *Full ARE*, Ψ consists of all items. **2) Compression Error (CE):** $\frac{1}{d \times w}\sum_{i=1}^{d}\sum_{j=1}^{w}\left|\widehat{\mathcal{A}_i}[j] - \mathcal{A}_i[j]\right|$, where $\widehat{\mathcal{A}_i}$ is the i^{th} array of the recovered sketch. **3) Loss:** For classification (using logistic regression), we use cross entropy. For linear regression, we use L2-norm.

6.2 Impact of Algorithm Parameters

By default, we use CM sketch and the 1-minute CAIDA dataset, and set $\tau = 4096$ and $r = 4$. For experiments reporting top-k ARE, we only use *SketchSensing*. For experiments reporting full ARE, we use both *TreeEncoding* and *SketchSensing*, where we use the *HierarchicalTree* of Shape #1 (without SIMD) and set $\lambda = 10$.

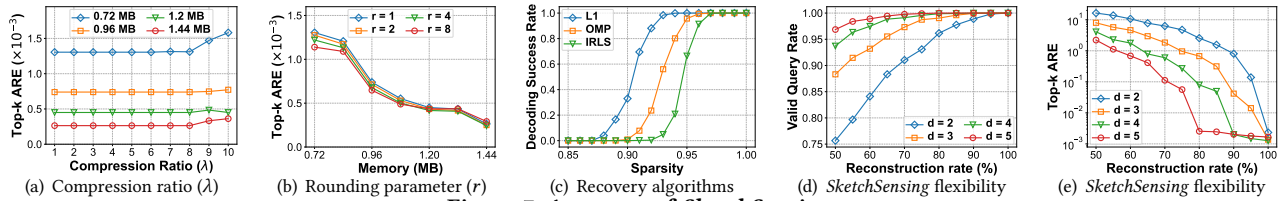
Performance of three recommended HierarchicalTrees (Figure 6(a)): We find that all the three shapes achieve similar accuracy as the non-compressed sketch. When compressing a 13.2KB sketch,

[5] "XC7VX690T-2FFG1761I" is the manufacturer part number of a Xilinx FPGA [81].



(a) Recommended shapes (b) Shapes (l and κ) (c) Compression ratio (d) ShiftBfEncoder Memory (e) *TreeEncoding* flexibility

Figure 6: Accuracy of *TreeEncoding*.



(a) Compression ratio (λ) (b) Rounding parameter (r) (c) Recovery algorithms (d) *SketchSensing* flexibility (e) *SketchSensing* flexibility

Figure 7: Accuracy of *SketchSensing*.

the ARE of the three shapes are 0.0113, 0.01012, and 0.0100, while that of the non-compressed sketch is 0.0100.

Impact of *HierarchicalTree* shape (Figure 6(b)): We find that a *HierarchicalTree* with smaller l and κ has smaller error. This is because larger l and κ go with more collisions at high layer. We fix the memory of *HierarchicalTree* to be about 1.4MB, and vary the shape of *HierarchicalTree* by changing l, κ, and δ (suppose all layers use the same κ). We can see when $l = 2$ and $\kappa = 4$, *TreeEncoding* has < 1 compression error. In practice, we recommend to use a 2-layer *HierarchicalTree* with $\kappa < 8$.

Impact of *TreeEncoding* compression ratio (Figure 6(c)): We find that the compression ratio of *TreeEncoding* can be roughly tuned by varying the shape of *HierarchicalTree*. We set $l = 2$, $\kappa_1 = 4$, and change δ_1 and δ_2 from 2 to 8 to achieve a compression ratio from 2.8 to 8.0. We can see that in general, smaller compression ratio goes with smaller ARE, and when compression ratio is <5.4, *TreeEncoding* achieves nearly lossless recovery. More finer-grained compression ratios can be achieved by further changing l and κ.

Impact of the memory of ShiftBfEncoder (Figure 6(d)): We find that the memory of ShiftBfEncoder is negligible. We can see that to compress a 13.2MB sketch, a 40KB ShiftBfEncoder suffices for high accuracy. The ARE of a non-compressed 13.2MB sketch is 0.01 (see Figure 6(a)), while the ARE of *TreeEncoding* using 40KB ShiftBfEncoder is 0.012. By contrast, the 1-bit indicators need about 600KB memory, which is 15× larger than ShiftBfEncoder.

Flexibility of TreeEncoding (Figure 6(e)): We find that an approximately recovered sketch from a partial *HierarchicalTree* can provide accurate estimation. We use the *HierarchicalTree* of *Shape #2*, and recover the sketch using the highest layer (L3), the highest two layers (L2+L3), and the entire *HierarchicalTree* (L1+L2+L3), respectively. Here, the query set consists of the items with real frequencies larger than 256. We can see that the approximately recovered sketches using L3 and L2+L3 achieve about 0.1 and 0.01 ARE, respectively.

Table 3: Compression error of *TreeEncoding*.

Shape	Memory	Merge	Pure Ratio	CE	ARE
#1 (2 layers)	9MB	sum	98.08%	0.85	0.028
	9MB	max		0.47	0.027
	12MB	sum	98.89%	0.48	0.012
	12MB	max		0.27	0.011
#2 (3 layers)	9MB	sum	99.77%	0.30	0.026
	12MB	sum	99.87%	0.17	0.011

Compression error of TreeEncoding (Table 3): We find that *TreeEncoding* has always <0.85 CE and <0.03 ARE, meaning that although *TreeEncoding* is not lossless, the recovered sketch has small error and can provide very high accuracy. We can see that >98% parent counters are pure counters, which is the reason why *TreeEncoding* achieves almost similar accuracy as lossless methods. We also observe that the maximum merging method has smaller error than sum merging. Thus, when using 2-layer *HierarchicalTree* ($l = 2$), we recommend to use the maximum merging method.

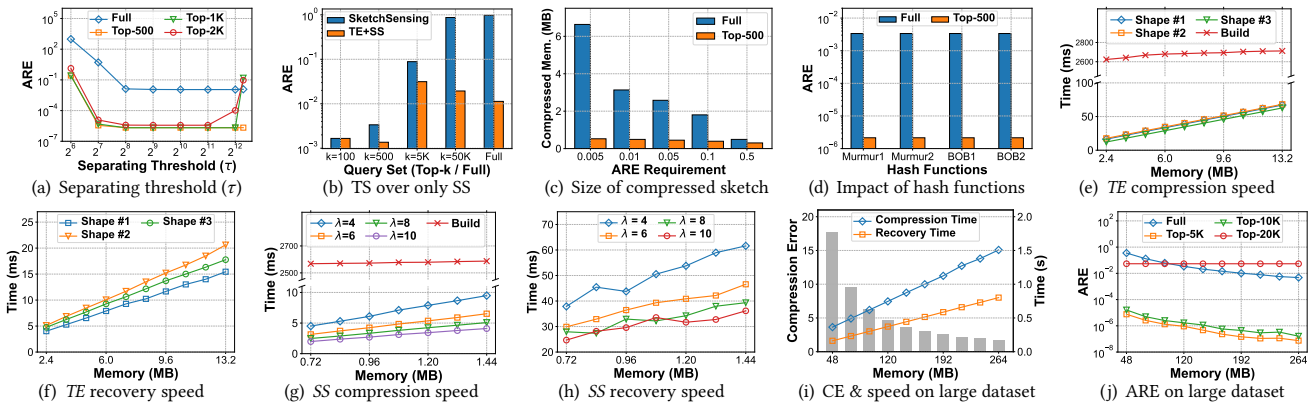
Impact of *SketchSensing* compression ratio (λ) (Figure 7(a)): We find that *SketchSensing* always achieves similar accuracy as the non-compressed sketch under different compression ratio. This is because under current separating threshold, the large sketch is sparse enough for compressive sensing to achieve lossless recovery.

Impact of rounding parameter (r) (Figure 7(b)): We find that the optimal value of r is 4. When using small memory (<1.2MB), larger r goes with smaller top-k ARE, because the rounding down operation offsets the overestimation error of CM. When using large memory (1.44MB), the ARE of $r = 8$ is larger than that of using small r. This is because larger r leads to more underestimation error. In practice, we recommend to set r to a small value, such as 4.

Table 4: Performance of *SketchSensing* ('SR' refers to the success rate of decoding each sensing fragment).

Matrix	Algo.	Top-k ARE ($\times 10^{-3}$)			CE ($\times 10^{-2}$)	SR
		k=500	k=1000	k=1500		
BM	L1	1.76	3.03	4.27	0.00	100%
	IRLS	1.69	2.93	4.12	3.10	100%
	OMP	1.74	3.29	4.50	1.23	99.2%
GM	L1	1.76	3.03	4.27	0.00	100%
	IRLS	1.96	3.24	4.67	2.98	97.8%
	OMP	2.28	3.69	3.69	1.22	95.8%

Impact of sensing matrix and recovery algorithm (Table 4 and Figure 7(c)): We find Bernoulli Matrix (BM) and L1 optimisation are the most suitable sensing matrix and recovery algorithm for *SketchSensing*. We evaluate *SketchSensing* under different sensing matrices (BM [53], GM [52]) and recovery algorithms (L1 [57], OMP [58], IRLS [59]). We can see that *SketchSensing* has high accuracy across all sensing matrices and recovery algorithms, where the combination of BM/GM and L1 achieves lossless recovery. From Figure 7(c), we also observe that L1 is better than OMP and IRLS. In practice, we recommend to use BM and L1 because they have



(a) Separating threshold (τ) (b) TS over only SS (c) Size of compressed sketch (d) Impact of hash functions (e) *TE* compression speed

(f) *TE* recovery speed (g) *SS* compression speed (h) *SS* recovery speed (i) CE & speed on large dataset (j) ARE on large dataset

Figure 8: Performance of `TreeSensing` ('TS', 'TE', and 'SS' means 'TreeSensing', 'TreeEncoding', and 'SketchSensing').

relatively higher accuracy, and BM only involves fast integer calculations (whereas GM involves float calculations).

Flexibility of SketchSensing (Figure 7(d)-7(e)): We find that an approximately recovered sketch from a part of sensing fragments can also provide accurate estimation. We can see that when $d = 5$, a sketch recovered from 70%/90% sensing fragments can report >95%/>99.9% valid results with <0.1/<0.005 top-k ARE.

Impact of separating threshold (τ) (Figure 8(a)): We find that the optimal value of the separating threshold is from 256 to 2048. When τ is small, the large sketch is not sparse enough for compressive sensing to achieve lossless recovery, and thus both top-k ARE and full ARE become large. When τ is large, top-500 ARE remains unchanged because compressive sensing always achieves lossless recovery. But as τ grows larger, top-1000 and top-2000 ARE become larger. This is because when τ is larger than the frequency of the k^{th} largest item, some items in the top-k query set will be compressed using *TreeEncoding*, which has larger error than *SketchSensing*. In practice, we recommend setting τ to the minimum value that can make the large sketch sparse enough for compressive sensing to achieve lossless recovery, *e.g.*, the 95^{th}-percentile of all counters.

TreeSensing over only using SketchSensing (Figure 8(b)): We find that TreeSensing suits for the tasks of full ARE, and SketchSensing suits for the tasks of top-k ARE. For tasks reporting top-100 ARE, `TreeSensing` and *SketchSensing* have the same accuracy. As k increases, the ARE gap between `TreeSensing` and *SketchSensing* becomes larger. For tasks reporting full ARE, `TreeSensing` has 100× lower ARE than *SketchSensing*. Thus, for applications that only care about the accuracy of frequent items, we recommend using just *SketchSensing*. Otherwise, we recommend using `TreeSensing`.

Minimum size of the compressed sketch (Figure 8(c)): We find that as the required ARE goes larger, the minimum size of the compressed sketch goes smaller. And in the tasks of reporting top-k ARE, the compressed sketch can be significantly smaller than that in the tasks of reporting full ARE. We use `TreeSensing` to compress a 15MB sketch, and report the minimum size of the compressed sketch at different ARE requirements. When the required full/top-k ARE is 0.5, we can compress the sketch by 30.5×/49.6×.

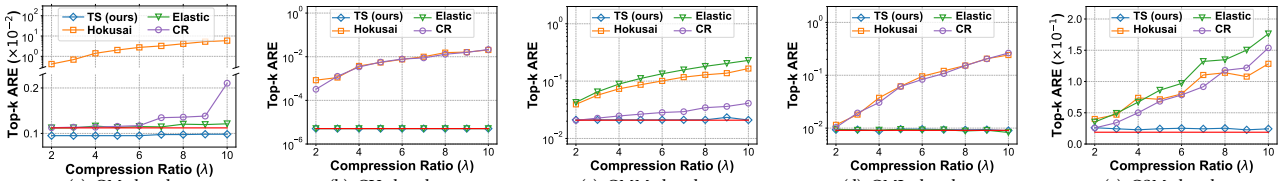
Impact of different hash functions (Figure 8(d)): We find that `TreeSensing` has almost the same accuracy across different hash functions, meaning that `TreeSensing` can be applied to the sketch using any hash function that has uniformly distributed outputs.

Efficiency of TreeEncoding (Figure 8(e)-8(f)): We find that the compression/recovery speed of *TreeEncoding* reaches up to 1.38Gbps/7.04Gbps. *TreeEncoding* only takes 78ms/20ms to compress/recover a 13.2MB sketch. By contrast, it takes 2.71s to build a 13.2MB sketch, which is 34.7× larger than the compression time.

Efficiency of SketchSensing (Figure 8(g)-8(h)): We find that larger compression ratio λ goes with faster compression/recovery speed of *SketchSensing*, which reaches up to 2.51Gbps and 0.32Gbps, respectively. This is because larger λ goes with smaller sensing matrix, and thus leads to smaller computation overhead. We can see that *SketchSensing* only takes 4.5 milliseconds to compress a 1.44MB sketch by 10 times, and takes 36 milliseconds to recover it. By contrast, it takes 2.59 seconds to build a 1.44MB sketch, which is 575.6× larger than the compression time.

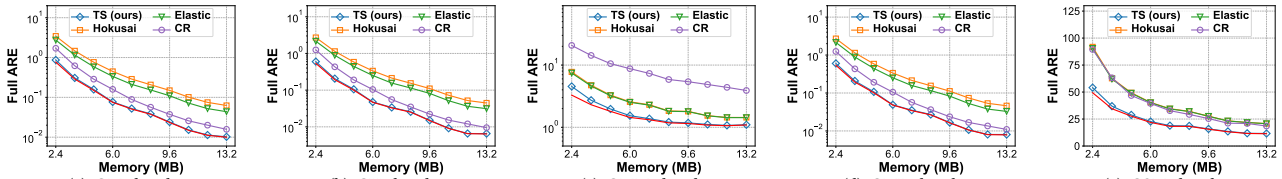
Performance of TreeSensing on large-scale dataset (Figure 8(i)-8(j)): We find that on large-scale datasets, `TreeSensing` also has fast speed and high accuracy. We use the large-scale CAIDA dataset with 1.5G items, where we build larger sketches and use `TreeSensing` to compress them. We can see that for a 264MB sketch, it only takes 1.5s and 0.76s for `TreeSensing` to compress and recover it, respectively. The Compression Error is smaller than 2, and the full/top-5K ARE is smaller than $10^{-2}/10^{-7}$, respectively.

Parameter setup methods: 1) For *TreeEncoding*, we recommend using a *HierarchicalTree* with $l = 2$ and $\kappa < 8$ (Figure 6(b)), and using a small-sized ShiftBfEncoder, such as 0.3% of the sketch size (Figure 6(d)). We can dynamically tune δ to balance between accuracy and compression ratio, and in general, *TreeEncoding* can achieve lossless recovery under <5.4 compression ratio (Figure 6(c)). For example, we recommend using Shape #1 in Table 2, which can compress the sketch by 5 times and achieve nearly lossless recovery (CE<0.85). If user wants to use SIMD acceleration, we recommend Shape #3 because it satisfies the alignment requirement of 256-bit AVX2 register. 2) For *SketchSensing*, we recommend using a small r, such as 4 (Figure 7(b)), and using Bernoulli Matrix [53] and L1 optimization [57] (Table 4 and Figure 7(c)). We can dynamically tune λ to balance between accuracy and compression ratio, and in general, *SketchSensing* can achieve lossless recovery when $\lambda < 8$ (Figure 7(a)). We recommend setting the fragment size m to 256 or 512. 3) Finally, we recommend setting the separating threshold τ to the minimum value that can make the large sketch satisfy the sparsity requirement of compressive sensing (Table 8(a)). In practice, we can set τ to the 95^{th}-percentile of the values of all counters.



(a) CM sketch (b) CU sketch (c) CMM sketch (d) CML sketch (e) CSM sketch

Figure 9: Comparison of top-k ARE (red line indicates the ARE of non-compressed sketch).



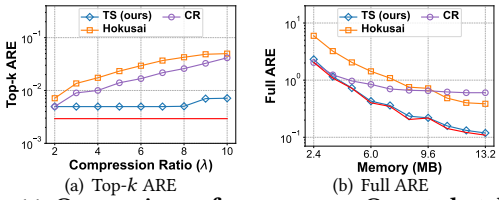
(a) CM sketch (b) CU sketch (c) CMM sketch (d) CML sketch (e) CSM sketch

Figure 10: Comparison of full ARE (red line indicates the ARE of non-compressed sketch).

6.3 Comparison with Prior Art

We compare TreeSensing (TS) with Hokusai [31], Elastic [32], and Cluster-Reduce (CR) [22] on five sketches with only positive counters (CM [1], CU [36], CMM [37], CML [37], and CSM [39]) and one sketch with both positive and negative counters (Count [33]). By default, we use the 1-minute CAIDA and set $\tau = 4096$. For tasks reporting top-k ARE, we only use *SketchSensing* to compress a 0.78MB sketch. For tasks reporting full ARE, we use both *TreeEncoding* (Shape #4, with SIMD acceleration) and *SketchSensing* (with $\lambda = 6$). We set the memory of the compressed sketch to be the same across all algorithms, which is $\frac{1}{2}$ of the original memory.

Top-k ARE on the sketches with only positive counters (Figure 9): We find that for all of the five sketches, the top-k ARE of TreeSensing is almost the same as the non-compressed sketch, meaning that *SketchSensing* achieves nearly lossless recovery. In particular, on CU sketch, the top-k ARE of TreeSensing is at least 100× lower than CR and Hokusai. We observe that the top-k ARE of TreeSensing is sometimes lower than non-compressed sketch. This is because the rounding technique fortunately offsets the overestimation error. We notice that on CU and CML, Elastic also achieves similar top-k ARE as the non-compressed sketch. This is because the maximum merge operation of Elastic tends to protect large counters. However, as will be described later, Elastic is unfriendly to small counters and thus have poor full ARE.

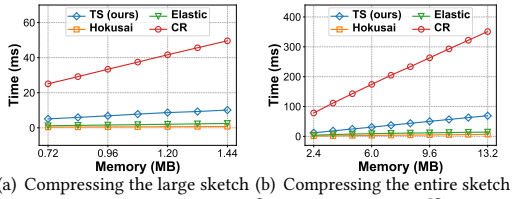


(a) Top-k ARE (b) Full ARE

Figure 11: Comparison of accuracy on Count sketch (red line indicates the ARE of non-compressed sketch).

Full ARE on the sketches with only positive counters (Figure 10): We find that for all of the five sketches, the full ARE of TreeSensing is almost the same as the non-compressed sketch, meaning that TreeSensing achieves nearly lossless recovery. In addition, we can see that TreeSensing always outperforms other algorithms on all the five sketches. Specifically, on CM sketch, TreeSensing achieves at least 6.2×, 4.3×, and 1.7× smaller full ARE than Hokusai, Elastic, and Cluster-Reduce. In particular, we notice that the full ARE of Elastic is significantly higher than TreeSensing and CR. This is because the merging operation of Elastic inevitably loses information of the infrequent items.

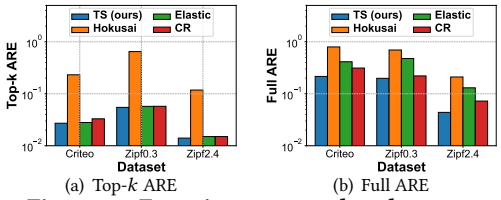
Accuracy on Count sketch (Figure 11): We find that compared to the non-compressed sketch, TreeSensing has slightly higher top-k ARE and almost the same full ARE. We can see that TreeSensing significantly outperforms other algorithms on both top-k ARE and full ARE. Specifically, TreeSensing achieves up to 8.2× smaller top-k ARE and up to 6.4× smaller full ARE than other algorithms. Note that Elastic cannot compress the sketch with negative counters.



(a) Compressing the large sketch (b) Compressing the entire sketch

Figure 12: Comparison of compression efficiency.

Compression efficiency (Figure 12): We find that TreeSensing is 3∼6 times faster than Cluster-Reduce, but is slower than Hokusai and Elastic. In Figure 12(a), we only use *SketchSensing*. We can see that to compress a 1.44MB sketch, it takes 10.0ms, 0.7ms, 2.4ms, and 50.0ms for TreeSensing, Hokusai, Elastic, and CR. In Figure 12(b), we use both *TreeEncoding* and *SketchSensing*. We can see that to compress a 13.2MB sketch, it takes 69.1ms, 6.7ms, 17.0ms, and 353.1ms for TreeSensing, Hokusai, Elastic, and CR.

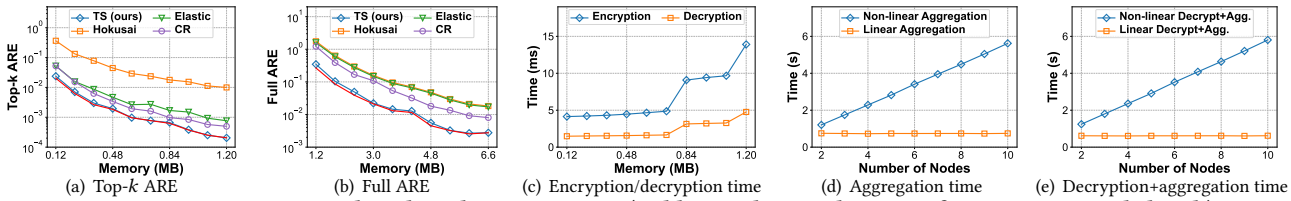


(a) Top-k ARE (b) Full ARE

Figure 13: Experiments on other datasets.

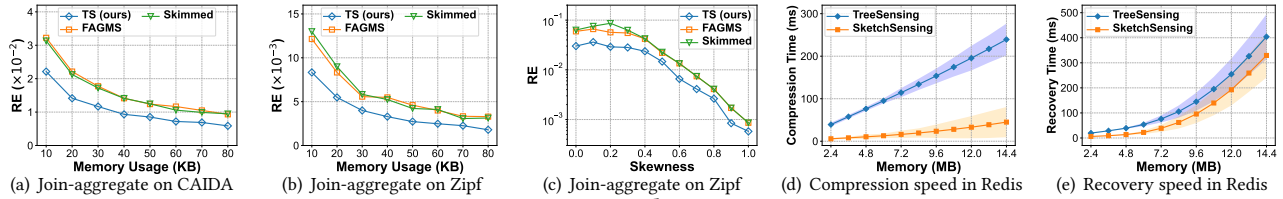
Evaluation on other datasets (Figure 13): We find that TreeSensing also outperforms other algorithms on other datasets. For example, on Zipf2.4 dataset, the top-k/full ARE of TreeSensing is 8.4×/4.8×, 1.1×/3.0×, and 1.1×/1.7× lower than Hokusai, Elastic, and Cluster-Reduce, respectively.

Summary (Table 5): We first summarize the trade-off of prior art. 1) Linearity: Elastic and CR do not satisfy the linear property, and thus they cannot achieve fast aggregation in distributed measurement.



(a) Top-k ARE (b) Full ARE (c) Encryption/decryption time (d) Aggregation time (e) Decryption+aggregation time

Figure 14: Experiments in distributed measurement (red line indicates the ARE of non-compressed sketch).



(a) Join-aggregate on CAIDA (b) Join-aggregate on Zipf (c) Join-aggregate on Zipf (d) Compression speed in Redis (e) Recovery speed in Redis

Figure 15: Experiments in data management scenes.

2) Generality: Elastic cannot compress the sketch with negative counters, so it cannot compress Count sketch. 3) Top-k accuracy: As a whole, Elastic has better top-k ARE than CR and Hokusai. On CM, CU, and CML, Elastic has almost the same top-k ARE as non-compressed sketch. But on CMM and CSM, its top-k ARE is the worst. 4) Full accuracy: As a whole, CR has better full ARE than Elastic and Hokusai. But on CMM, its full ARE is the worst. We notice that none of prior art achieve nearly the same full ARE as non-compressed sketch. 5) Speed: Hokusai and Elastic are very fast. But CR is very slow due to its large computation complexity.

By contrast, `TreeSensing` satisfies the linear property and is general to the sketch with negative counters. `TreeSensing` has almost the same top-k ARE and full ARE as non-compressed sketch, meaning that it can achieve nearly lossless recovery. In addition, `TreeSensing` achieves consistently high accuracy across all the six sketches, whereas none of prior art has stable ARE across all sketches. Finally, `TreeSensing` is significantly faster than CR but slightly slower than Hokusai and Elastic.

Table 5: Summary of empirical comparison with prior art.

Algorithm	Linearity	Generality	Top-k ACC.	Full ACC.	Speed
Hokusai	✓	✓	Low	Low	Very fast
Elastic	×	×	High	Low	Very fast
CR	×	✓	Medium	Medium	Slow
TS (ours)	✓	✓	High	High	Fast

6.4 Application in Distributed Measurement

We apply `TreeSensing` to a distributed measurement system and compare it with Hokusai [31], Elastic [32], and Cluster-Reduce (CR) [22] on CM [1]. By default, there are eight measurement nodes in our system. For tasks reporting top-k ARE, we only use *SketchSensing*. For tasks reporting full ARE, we use both *TreeEncoding* and *SketchSensing*. We set the memory of the compressed sketches to be the same across all algorithms, which is $\frac{1}{2}$ of the original memory. We use Paillier cryptosystem [71] to encrypt 5% counters in each sensing fragment, achieving privacy-preserving measurement.

Accuracy in distributed measurement (Figure 14(a)-14(b)): We find that `TreeSensing` always outperforms other algorithms on both top-k ARE and full ARE, and `TreeSensing` achieves almost the same accuracy as non-compressed sketch. From Figure 14(a), we can see that when using local sketches of 0.6MB, the top-k ARE of `TreeSensing` is 48.83×, 3.74×, and 2.44× lower than Hokusai, Elastic, and Cluster-Reduce, respectively. From Figure 14(b), we can see that when using local sketches of 6.6MB, the full ARE of `TreeSensing` is 6.58×, 6.18×, and 2.90× lower than than Hokusai, Elastic, and Cluster-Reduce, respectively.

Efficiency of privacy-preserving distributed measurement (Figure 14(c)): We find that the time cost of homomorphic encryption and decryption is very small. We can see that it only takes 13.9ms/4.7ms to encrypt/decrypt a 1.2MB sketch, respectively.

Comparison between linear and non-linear aggregation (Figure 14(d)-14(e)): We find that linear aggregation is significantly faster than non-linear aggregation. In Figure 14(d), we compare the aggregation time between linear and non-linear modes of `TreeSensing`. We can see that when there are 8 nodes, linear aggregation is 4.9× faster than non-linear aggregation. As the system scale grows, this gap will become larger. This is because in linear mode, we just perform one recovery operation, while in non-linear mode, we need to recover all local sketches. In Figure 14(e), we compare the decryption+aggregation time in privacy-preserving measurement between the two modes. The results are similar as in Figure 14(d). In linear mode, we just perform one decryption and one recovery operation, whereas in non-linear mode, we must decrypt and recover all local sketches.

6.5 Application in Data Management Scenes

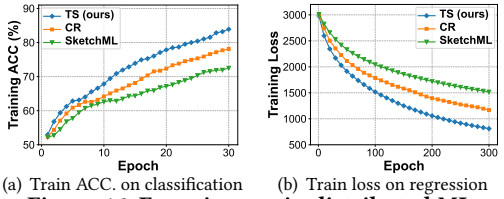
To better show the benefit of `TreeSensing` to data management community, we apply `TreeSensing` to the task of join-aggregate estimation [88], and integrate `TreeSensing` into Redis [89].

Join-aggregate estimation (Figure 15(a)-15(c)): Given two data streams F and G with N distinct items. Let f_i and g_i denote the frequencies of an item e_i in F and G. The result of the join-aggregate query is defined as $J = \sum_{i=1}^{N} f_i \cdot g_i$. Join-aggregate estimation is the base of many data management applications, such as query optimizer in DBMS [7, 90, 91], traffic analyzer in DSMS [92, 93], and more [94]. For example, consider the case of distributed multi-way join in DBMS, we want to perform the join operation on multiple tables distributed on different nodes. A good join-aggregate estimation algorithm can guide us to devise an optimal join plan, which minimizes the volume of intermediate relations and the communication time. Sketches are widely used for join-aggregate estimation. Typical sketches include AGMS [88], Fast-AGMS (FAGMS) [15], Skimmed sketch [93], and more [95–98]. Specifically, Skimmed sketch [93] proposes to separate frequent and infrequent items to reduce the estimation variance. This separation idea is similar

to TreeSensing. But skimmed sketch is designed specifically for join-aggregate estimation, whereas our aim is to design a general framework to accurately and flexibly compress all sketches.

We conduct experiments of join-aggregate estimation on CAIDA and Zipf datasets, where we use TreeSensing to compress a Count sketch [33], and compare its accuracy with non-compressed FAGMS and Skimmed sketch. We control the compressed memory of the Count sketch to be the same as the memory of non-compressed FAGMS and Skimmed sketch. We use the metric of relative error (RE), which is defined as $\sum |J - \hat{J}|/J$. From Figure 15(a)-15(c), we can see that TreeSensing always outperforms Skimmed sketch and FAGMS on the two datasets. The accuracy of Skimmed sketch is always inferior than TreeSensing and sometimes even inferior than FAGMS. This is because under small separating threshold, the separated frequent items by Skimmed sketch have large error, which degrades the accuracy. On the other hand, under large separating threshold, Skimmed sketch will degenerate into plain Count sketch. In conclusion, in the task of join-aggregate estimation (especially in distributed scenario), besides devising smart data structures, another promising direction is to use TreeSensing or other compression algorithms to efficiently compress sketches, so that we can build larger sketches to improve accuracy.

TreeSensing in Redis (Figure 15(d)-15(e)): Redis is a widely used in-memory data structure store. We integrate TreeSensing into Redis as a persistence tool to reduce the storage overhead of sketches. We first implement a CM sketch using Redis Module, which supports basic insertion and query commands. Then we implement TreeSensing by adding four commands to the CM sketch: *TreeSensing* compression/recovery, and *SketchSensing* compression/recovery. We evaluate the compression/recovery speed of *TreeSensing* and *SketchSensing* with Redis 5.0.7. All experiments are repeated 30 times and the average (±std) time is plotted. We can see that TreeSensing can smoothly work on top of Redis, where TreeSensing only takes less than 0.24/0.41 seconds to compress/recover a 14.4MB sketch.



(a) Train ACC. on classification (b) Train loss on regression

Figure 16: Experiments in distributed ML.

6.6 Application in Distributed ML

We apply TreeSensing to a distributed machine learning system and compare it with Cluster-Reduce (CR) [22] and SketchML [21] on two tasks: classification (using logistic regression) and regression (using linear regression). In our system, there are eight workers and one parameter server. We fix the available transmission bandwidth of the three algorithms to be about $\frac{1}{6}$ of the exact gradients. As described in § 5.2, for TreeSensing and Cluster-Reduce, we build larger MinMax sketches to encode gradients, and use *SketchSensing* to compress it into small memory before transmission.

Classification (Figure 16(a)): We find that the training accuracy of TreeSensing always outperforms CR and SketchML. TreeSensing improves the training accuracy by up to 6%/12% than CR/SketchML. After 30 epochs, the accuracy of TreeSensing reaches 83.8%, while that of CR and SketchML are 78.1% and 72.5%.

Regression (Figure 16(b)): We find that the training speed of TreeSensing is 1.7× faster than CR and 3× faster than SketchML. Here, training faster means achieving higher accuracy using the same time. When training loss reaches 1450, TreeSensing uses 100 epochs, CR uses 170 epochs, and SketchML uses 300 epochs.

Summary and analysis: The results show that TreeSensing can improve the training accuracy and speed in distributed ML. This is because under fixed transmission bandwidth, by compressing the sketches into small memory, we can build larger local sketches to encode gradients more accurately. Therefore, the parameter server can recover gradients more accurately and update parameters more precisely, making the training process faster and more accurate.

6.7 Evaluation on FPGA Platform

We implement TreeSensing on an FPGA network platform (Virtex-7 VC709). For *TreeEncoding*, we use Shape #3 without ShiftBfEncoder. The implementation of *TreeEncoding* consists of 4 modules: separating small sketch, setting 1-bit indicators, setting L_1, and setting L_2. For *SketchSensing*, the implementation consists of m'+1 modules (m' is the # column of the sensing matrix ϕ): separating the large sketch and rounding down, and multiplying a counter with a corresponding counter in the j^{th} column of the sensing matrix (multiplying_j). For example, in the multiplying_j module, the i^{th} counter is multiplied with $\phi[i][j]$. In this way, each counter participates in the calculation of each column, which fully utilizes hardware parallelism. In addition, as each element in Bernoulli Matrix is 0/1, we use conditional operations to further accelerate the computation speed. Both *TreeEncoding* and *SketchSensing* are fully pipelined, which process one 32-bit counter in each clock cycle.

Table 6 shows the clock frequency and all hardware resources used by *TreeSensing*. We can see that *TreeEncoding* and *SketchSensing* achieve 526MHz and 316MHz clock frequency, meaning that TreeSensing has 316MHz throughput. The LUT (Look-Up-Table), Register and Block Tile resource usage are all <0.5%, which is nearly negligible. In summary, on FPGA, TreeSensing has negligible resource overhead and fast compression speed, which is at least 11.4× higher than that on CPU platform. Therefore, it is lucrative to further speed up TreeSensing with FPGA.

Table 6: Performance on FPGA Platform.

Module	Resource Overhead					Frequency (MHz)
	LUTs	Register	Block Tile	IOB	BUFGCTRL	
TE	0.02%	0.01%	0.10%	4.24%	3.12%	526
SS	0.16%	0.04%	0.20%	4.24%	0.00%	316
TE+SS	0.17%	0.05%	0.31%	4.24%	3.12%	316

7 CONCLUSION

This paper proposes TreeSensing, an accurate, efficient, and flexible framework to linearly compress sketches. In TreeSensing, we first separate the original sketch into two partial sketches, and then compress the two partial sketches with two key techniques, namely *TreeEncoding* and *SketchSensing*. Theoretical analyses of TreeSensing are provided. We use TreeSensing to compress 7 sketches, and conduct two end-to-end experiments: distributed measurement and distributed ML. We also implement TreeSensing on FPGA and integrate it into Redis. Experimental results show that TreeSensing significantly outperforms existing solutions.

REFERENCES

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

[2] Graham Cormode and Minos Garofalakis. Sketching probabilistic data streams. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 281–292, 2007.

[3] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *SIGMOD*, 2015.

[4] Rui Zhu, Bin Wang, Xiaochun Yang, Baihua Zheng, and Guoren Wang. Sap: Improving continuous top-k queries over streaming data. *IEEE Transactions on Knowledge and Data Engineering*, 29(6):1310–1328, 2017.

[5] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. Stable learned bloom filters for data streams. *Proceedings of the VLDB Endowment*, 13(12):2355–2367, 2020.

[6] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. Online sketch-based query optimization. *arXiv preprint arXiv:2102.02440*, 2021.

[7] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung 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.

[8] Asoke Datta, Yesdaulet Izenov, Brian Tsan, and Florin Rusu. Simpli-squared: A very simple yet unexpectedly powerful join ordering algorithm without cardinality estimates. *arXiv preprint arXiv:2111.00163*, 2021.

[9] Benwei Shi, Zhuoyue Zhao, Yanqing Peng, Feifei Li, and Jeff 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.

[10] Yufei Tao, George Kollios, Jeffrey Considine, Feifei Li, and Dimitris Papadias. Spatio-temporal aggregation using sketches. In *Proceedings. 20th International Conference on Data Engineering*, pages 214–225. IEEE, 2004.

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

[12] Wei Xie, Feida Zhu, Jing Jiang, Ee-Peng Lim, and Ke Wang. Topicsketch: Real-time bursty topic detection from twitter. *TKDE*, 2016.

[13] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. A learned sketch for subgraph counting. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2142–2155, 2021.

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

[15] Graham Cormode and Minos 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] Graham Cormode, Samuel Maddock, and Carsten Maple. Frequency estimation under local differential privacy. *Proceedings of the VLDB Endowment*, 14(11):2046–2058, 2021.

[17] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1937–1940, 2019.

[18] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, 2010.

[19] Ankush Mandal, He Jiang, Anshumali Shrivastava, and Vivek Sarkar. Topkapi: parallel and fast sketches for finding top-k frequent elements. *Advances in Neural Information Processing Systems*, 31, 2018.

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

[21] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin 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.

[22] Yikai Zhao, Zheng Zhong, Yuanpeng Li, Yi Zhou, Yifan Zhu, Li Chen, Yi Wang, and Tong Yang. Cluster-reduce: Compressing sketches for distributed data streams. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2316–2326, 2021.

[23] Ahmed S Abdelhamid, Ahmed R Mahmood, Anas Daghistani, and Walid G Aref. Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2455–2469, 2020.

[24] Nicolas Kourtellis, Herodotos Herodotou, Maciej Grzenda, Piotr Wawrzyniak, and Albert Bifet. S2ce: a hybrid cloud and edge orchestrator for mining exascale distributed streams. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 103–113, 2021.

[25] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[26] Ahmed R Mahmood, Ahmed M Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S Abdelhamid, Mohamed S Hassan, Walid G Aref, and Saleh Basalamah. Tornado: A distributed spatio-textual stream processing system. *Proceedings of the VLDB Endowment*, 8(12):2020–2023, 2015.

[27] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[28] Kalikinkar Mandal and Guang Gong. Privfl: Practical privacy-preserving federated regressions on high-dimensional data over mobile networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 57–68, 2019.

[29] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.

[30] Kalikinkar Mandal, Guang Gong, and Chuyi Liu. Nike-based fast privacy-preserving highdimensional data aggregation for mobile devices. *IEEE T Depend Secure; Technical Report; University of Waterloo: Waterloo, ON, Canada*, pages 142–149, 2018.

[31] Sergiy Matusevych, Alex Smola, and Amr Ahmed. Hokusai-sketching streams in real time. *arXiv preprint arXiv:1210.4891*, 2012.

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

[33] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. 2002.

[34] Richard G Baraniuk. Compressive sensing [lecture notes]. *IEEE signal processing magazine*, 24(4):118–121, 2007.

[35] Treesensing related codes. https://github.com/TowerSensing/TowerSensing.

[36] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, 2002.

[37] Fan Deng and Davood Rafiei. New estimation algorithms for streaming data: Count-min can do more. *Webdocs. Cs. Ualberta. Ca*, 2007.

[38] Guillaume Pitel and Geoffroy Fouquier. Count-min-log sketch: Approximately counting with approximate counters. *arXiv preprint arXiv:1502.04885*, 2015.

[39] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking*, 20(5):1622–1634, 2012.

[40] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. Salsa: self-adjusting lean streaming analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 864–875. IEEE, 2021.

[41] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *SIGMOD*, 2016.

[42] Florin Rusu and Alin Dobra. Sketching sampled data streams. In *2009 IEEE 25th International Conference on Data Engineering*, pages 381–392. IEEE, 2009.

[43] Lada A Adamic and Bernardo A Huberman. Power-law distribution of the world wide web. *science*, 287(5461):2115–2115, 2000.

[44] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[45] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. *Proc. ACM SIGMETRICS*, 36(1):121–132, 2008.

[46] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[47] Tong Yang, Siang Gao, Zhouyi Sun, Yufei Wang, Yulong Shen, and Xiaoming Li. Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2650–2662, 2019.

[48] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[49] Peter Deutsch. Rfc1951: Deflate compressed data format specification version 1.3, 1996.

[50] Jorma Rissanen and Glen G Langdon. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.

[51] Fabian Mentzer, Luc Van Gool, and Michael Tschannen. Learning better lossless compression using lossy compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6638–6647, 2020.

[52] Stanislaw J Szarek. Condition numbers of random matrices. *Journal of Complexity*, 7(2):131–149, 1991.

[53] Emmanuel J Candes and Terence Tao. Near-optimal signal recovery from random projections: Universal encoding strategies? *IEEE transactions on information theory*, 52(12):5406–5425, 2006.

[54] Emmanuel J Candès and Michael B Wakin. An introduction to compressive sampling. *IEEE signal processing magazine*, 25(2):21–30, 2008.

[55] David L Donoho. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

[56] Graham Cormode and S Muthukrishnan. Combinatorial algorithms for compressed sensing. In *International colloquium on structural information and communication complexity*, pages 280–294. Springer, 2006.

[57] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[58] Yagyensh Chandra Pati, Ramin Rezaiifar, and Perinkulam Sambamurthy Krishnaprasad. Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In *Proceedings of 27th Asilomar conference on signals, systems and computers*, pages 40–44. IEEE, 1993.

[59] Peter J Green. Iteratively reweighted least squares for maximum likelihood estimation, and some robust and resistant alternatives. *Journal of the Royal Statistical Society: Series B (Methodological)*, 46(2):149–170, 1984.

[60] Deanna Needell and Joel A Tropp. Cosamp: Iterative signal recovery from incomplete and inaccurate samples. *Applied and computational harmonic analysis*, 26(3):301–321, 2009.

[61] Thomas Blumensath and Mike E Davies. Iterative thresholding for sparse approximations. *Journal of Fourier analysis and Applications*, 14(5):629–654, 2008.

[62] Irena Orović, Vladan Papić, Cornel Ioana, Xiumei Li, and Srdjan Stanković. Compressive sensing in signal processing: algorithms and transform domain formulations. *Mathematical Problems in Engineering*, 2016, 2016.

[63] Haipeng Peng, Ye Tian, Jürgen Kurths, Lixiang Li, Yixian Yang, and Daoshun Wang. Secure and energy-efficient data transmission system based on chaotic compressive sensing in body-to-body networks. *IEEE transactions on biomedical circuits and systems*, 11(3):558–573, 2017.

[64] Linghe Kong, Liang He, Xiao-Yang Liu, Yu Gu, Min-You Wu, and Xue Liu. Privacy-preserving compressive sensing for crowdsensing based trajectory recovery. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 31–40. IEEE, 2015.

[65] Bartosz Przydatek, Dawn Song, and Adrian Perrig. Sia: Secure information aggregation in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 255–265, 2003.

[66] Haowen Chan, Adrian Perrig, and Dawn Song. Secure hierarchical in-network aggregation in sensor networks. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 278–287, 2006.

[67] Gergely Ács and Claude Castelluccia. I have a dream!(differentially private smart metering). In *International Workshop on Information Hiding*, pages 118–132. Springer, 2011.

[68] Fengjun Li, Bo Luo, and Peng Liu. Secure information aggregation for smart grids using homomorphic encryption. In *2010 first IEEE international conference on smart grid communications*, pages 327–332. IEEE, 2010.

[69] Slawomir Goryczka and Li Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE transactions on dependable and secure computing*, 14(5):463–477, 2015.

[70] Josh Benaloh. Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography*, pages 120–128, 1994.

[71] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[72] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[73] Mu Li, Ziqi Liu, Alexander J Smola, and Yu-Xiang Wang. Difacto: Distributed factorization machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 377–386, 2016.

[74] Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment*, 9(12):960–971, 2016.

[75] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[76] Konstantin Mishchenko, Eduard Gorbunov, Martin Takáč, and Peter Richtárik. Distributed learning with compressed gradient differences. *arXiv preprint arXiv:1901.09269*, 2019.

[77] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Grace: A compressed communication framework for distributed machine learning. In *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*, pages 561–572. IEEE, 2021.

[78] Atal Sahu, Aritra Dutta, Ahmed M Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. Rethinking gradient sparsification as total error minimization. *Advances in Neural Information Processing Systems*, 34:8133–8146, 2021.

[79] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[80] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

[81] Fpga xc7vx690t-2ffg1761i specifications. https://dir.heisener.com/specification-pdf/en/XC7VX690T-2FFG1761I.pdf.

[82] Murmur hashing source codes. https://github.com/aappleby/smhasher.

[83] Hash website. http://burtleburtle.net/bob/hash/evahash.html.

[84] The caida anonymized internet traces. https://www.caida.org/catalog/datasets/passive_dataset.

[85] The criteo dataset. Available: https://ailab.criteo.com/ressources/.

[86] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

[87] Arthur Asuncion and David Newman. Uci machine learning repository, 2007.

[88] Noga Alon, Phillip B Gibbons, Yossi Matias, and Mario 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*, pages 10–20, 1999.

[89] The redis in-memory data store. https://redis.io.

[90] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[91] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, 2018.

[92] Konstantin Kutzkov, Mohamed Ahmed, and Sofia Nikitaki. Weighted similarity estimation in data streams. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 1051–1060, 2015.

[93] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Processing data-stream join aggregates using skimmed sketches. In *International Conference on Extending Database Technology*, pages 569–586. Springer, 2004.

[94] Yilei Wang and Ke Yi. Secure yannakakis: Join-aggregate queries over private data. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1969–1981, 2021.

[95] Sumit Ganguly, Deepanjan Kesh, and Chandan Saha. Practical algorithms for tracking database join sizes. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 297–309. Springer, 2005.

[96] Florin Rusu and Alin Dobra. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)*, 33(3):1–46, 2008.

[97] Florin Rusu and Alin Dobra. Statistical analysis of sketch estimators. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 187–198, 2007.

[98] Walter Cai, Magdalena Balazinska, and Dan 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.