Clustering-preserving Network Flow Sketching

Yongquan Fu [1], Dongsheng Li [1], Siqi Shen [1], Yiming Zhang [1], Kai Chen [2]
[1] Science and Technology Laboratory of Parallel and Distributed Processing,
College of Computer Science, National University of Defense Technology
[2] SING Lab, Hong Kong University of Science and Technology

***Abstract*—Network monitoring is vital in modern clouds and data center networks that need diverse traffic statistics ranging from flow size distributions to heavy hitters. To cope with increasing network rates and massive traffic volumes, sketch based approximate measurement has been extensively studied to trade the accuracy for memory and computation cost, which unfortunately, is sensitive to hash collisions.**

This paper presents a clustering-preserving sketch method to be resilient to hash collisions. We provide an equivalence analysis of the sketch in terms of the K-means clustering. Based on the analysis result, we cluster similar network flows to the same bucket array to reduce the estimation variance and use the average to obtain unbiased estimation. Testbed shows that the framework adapts to line rates and provides accurate query results. Real-world trace-driven simulations show that LSS remains stable performance under wide ranges of parameters and dramatically outperforms state-of-the-art sketching structures, with over 10^3 to 10^5 times reduction in relative errors for per-flow queries as the ratio of the number of buckets to the number of network flows reduces from 10% to 0.1%.

***Index Terms*—sketch, random projection, hash collision, clustering**

I. Introduction

Network measurement is of paramount importance for traffic engineering, network diagnosis, network forensics, intrusion detection and prevention in clouds and data centers, which need a variety of traffic measurement, such as delay, flow size estimation, flow distribution, heavy hitters [1], [2], [3]. Recently, the self-running network proposal [4], [5] highlights an automatic management loop for large-scale networks with timely and accurate data-driven network statistics as the driving force for machine learning techniques.

Network-flow monitoring is challenging due to ever increasing line rates, massive traffic volumes, and large numbers of active flows [6], [7], [8], [9]. Traffic statistics tasks require advanced data structures and traffic statistical algorithms. Many space- and time-efficient approaches have been studied, e.g., traffic sampling, traffic counting, traffic sketching. Compared to other approaches, the sketch has received extensive attentions due to their competitive trade off between space resource consumption and query efficiency. Further, multiple sketch structures can be composed for joint traffic analytics.

Existing sketch structures [10], [11], [12], [13] hash incoming packets to randomly chosen buckets and take the accumulated counter in these buckets as the estimator. Recently, OpenSketch [14], UnivMon [15], SketchVisor [16], ElasticSketch [17], and SketchLearn [18] further extend the generality of the sketch structure to support diverse monitoring tasks.

The sketch based monitoring approach has a degree of approximation error due to hash collisions of incoming items, as multiple keys may be mapped to the same bucket. Hash collisions are inevitable due to the randomness of the hash functions. Thus existing methods typically keep multiple independent copies of the sketch structure and find the least affected ones as the estimator. However, this approach wastes the space significantly. Recently, several approaches [17], [18] propose to separate large items from the rest into a hash table to reduce the estimation error. Unfortunately, the hash table needs to allocate dedicated space for new items, thus it is less efficient than the sketch with a constant-size bucket structure. Thus, finding a space-efficient approach that is resilient to hash collisions is an open question.

We present a new class of sketch called locality-sensitive sketch (LSS for short) that is resilient to hash collisions. LSS approximately minimizes the estimation error based on a theoretical equivalence relationship between the sketching error and the approximation error of the K-means clustering in Sec. IV. This equivalence provides two important insights for the sketching methodology: clustering similar items together reduces the approximation error, and averaging the bucket counter obtains an unbiased estimator. We exploit these two theoretical insights to the design of a locality-sensitive sketch structure.

We adapt to online and dynamic network flows with background clustering and lightweight temporal caching techniques. First, we maintain the clustering model in a background and periodical process, which obtains close-to-date samples and trains a clustering model that enables mapping online flow records with up-to-date cluster centers. Second, the insertion process should deal with incremental flow counters, since the flow size grows as packets are delivered. We adapt to monodically increasing flow counters with a temporal cache based on a lightweight Cuckoo hash table [19], [20], [21].

We perform extensive evaluation in Section VI. Testbed shows that the framework adapts to line rates and provides accurate query results. Trace-driven study reveals that LSS remains stable performance under wide ranges of parameters and dramatically outperforms state-of-the-art sketching structures, with over 10^3 to 10^5 times reduction in relative errors

This work was sponsored in part by National Key R&D Program of China under Grant No. 2018YFB0204300, and the National Natural Science Foundation of China (NSFC) under Grant No. 61972409, 61602500, 61402509, 61772541, 61872376.

for per-flow queries as the ratio of the number of buckets to the number of network flows reduces from 10% to 0.1%.

We summarize our contributions as follows:

- We provide a random-projection framework to quantify the expectation and the variance of the estimation error of the sketch structure.
- We establish the equivalence between the K-means clustering and the optimization of the sketch accuracy, from which we present a normalized sketch structure based on the clustering equivalence.
- We present the design and implementation of a locality-sensitive sketch for scalable and resilient network flow monitoring.
- We conduct extensive performance evaluation with testbed and trace-driven simulation, and confirm that the proposed sketch based monitoring application dramatically reduces the estimation error under the same memory footprint.

The rest of the paper is organized as follows. Sec. II provides background of the sketch based network flow monitoring process and summarizes related studies that are most related to our work. Sec. III presents a random-projection approach to accurately capture the expectation and the variance of the sketch's estimation error. Sec. IV establishes the connection between the optimized sketch and the K-means clustering and presents a normalized sketching methodology inspired by this connection. Sec. V next presents the design and implementation of the locality-sensitive sketch based monitoring solution. Sec. VI conducts extensive performance evaluation with testbed and trace datasets. We finally conclude in Sec. VII.

II. Background and Related Work

We present the background and related work in this section. Key notations include: N: Number of unique keys; X: Key-value streams; $\hat{X}$: Estimated key-value streams; A: Indicator matrix; $\{C_i\}$: Cluster centers; a: Bucket array; k: Number of cluster centers; m: Number of buckets.

A. Background

A sketch based monitoring application typically comprises an *ingestion* component that intercepts incoming packets from the physical network interface and generates key-value input for the sketch, a *sketching* component that feeds the key-value input to a sketch structure that approximates these key-value pairs with one or multiple hash based bucket arrays. For instance, existing sketch based monitoring applications directly ingest packet streams. Each flow is typically represented as a key-value pair, where the key is defined by a combination of packet-header fields and the value summarizes the flow's statistics, e.g., packet numbers or byte counts. For each incoming packet, a sketch based monitor inspects the packet header to extract the key and calculate the packet's value, then inserts this record to the sketch data structure. Finally, to estimate the accumulated value of a key, the monitor queries the sketch with the input key, which returns an approximate value over the shared bucket arrays for all inserted keys.

Concretely, count-min sketch (CM) [11], one of the most popular sketch methods, maintains k banks of arrays of size m, where k and m are chosen based on the accuracy requirement. To insert a key-value pair to the sketch, we chooses k uniformly-random hash functions h_j, $j \in \{1, 2, \cdots, k\}$ to map each key to a randomly chosen bucket from each bank. To query a given key, CM uses the same set of hash functions to select k buckets from each bank (for the j-th bank, the h_j(key)-th bucket is selected). CM approximates the value of a given key by the minimum of mapped buckets. Given a vector of items denoted as X, CM [11] shows that, the probability of the minimum of the inserted buckets is greater than the ground-truth value by $\frac{2}{m}\|X\|_1$ is at most $\frac{1}{2^k}$, where $\|X\|_1 = \sum_{j \in X} |j|$.

B. Related Studies

State-of-the-art sketch structures [10], [11], [12], [17] choose the least affected bucket from multiple copies of independent bucket arrays as the estimator. Recently, ElasticSketch [17] keeps heavy hitters separately with a hash table, and puts the rest of items to a count-min sketch. Thus it is less sensitive to heavy hitters compared to prior sketch structures [10], [11], [12]. However, as heavy hitters only represent a small fraction of items, the count-min sketch is still sensitive to hash collisions. SketchLearn [18] uses a multi-level array to keep the traffic statistics of specific flow-record bits, and separates large flows from the rest of flows like ElasticSketch [17] based on inferred flow distributions. Different from these studies, our work proactively applies a cluster-preserving approach to reduce the estimation error due to hash collisions. Further, although our work is orthogonal to these studies, the LSS sketch structure can be combined to these frameworks to improve the sketching efficiency.

III. Random Projection based Sketch Analysis

The sketch structure should remain fairly accurate under a wide range of parameter configurations. Unfortunately, a sketch is sensitive to hash collisions where multiple keys are mapped to the same bucket.

Assume that a sketch consists of one bucket array for ease of analysis. Solving multiple copies of bucket arrays is left as future work, which involves complicated order statistics over randomized data stream samples. Suppose that a sketch structure randomly maps incoming items to a bucket array uniformly at random. Let $X : N \times 1$ denote the vector of the streaming key-value sequence from the network ingestion component. Let $A : N \times m$ denote the indicator matrix of mapping the vector X to a bucket array a of size $m \times 1$. Let $A(i, j) = 1$ iff the i-th item X_i is mapped to the j-th bucket I_j, and $A(i, l) = 0$ for $l \neq j, l \in [1, m]$.

For example, we can represent the mapping matrix in Count-min (CM) as follows: Each key is mapped to only one bucket in a bucket array uniformly at random, which accumulates the key's value to the current counter. Thus the projection matrix

can be formulated as: $\sum_{j=1}^{m} A(i,j) = 1$, where $A(i,j) \in \{0,1\}$ for any entry (i,j).

Theorem 1 establishes the equivalence between the sketch with the random projection as follows:

Theorem 1. *A sketch with one bucket array is equivalent to a random projection: the insertion process corresponds to $a = (A^T X)$, while the query phase corresponds to $\hat{X} = A \cdot a$. The overall sketch is represented as $\hat{X} = AA^T X$.*

Proof. For each incoming key-value pair $(\kappa(i), X_i)$, the sketch selects only one bucket indexed by a variable j by hashing the key $\kappa(i)$ with a hash function, and appends the value scalar X_i to this bucket by incrementing the bucket's counter by X_i. Equivalently, we set the i-th row vector of A, denoted as $A(i,:)$, to a 0-1 vector, where only the j-th entry is one, i.e., $A(i,j) = 1$, and set other entries in this row vector to zeros. Consequently, we can equivalently transform this insertion choice as $a = a + A(i,:)^T \cdot X_i$. The insertion process for all key-value pairs can be represented as an algebraic equation: $a = (A^T X)$.

To estimate the value of a key $\kappa(i)$, the sketch selects the same bucket indexed by j by hashing $\kappa(i)$ with the same hash function as the insertion process, and then returns the bucket's counter $a(j)$ as the approximated value for X_i. Similarly, based on A's i-th row vector $A(i,:)$, we equivalently represent the approximated value as $\hat{X}_i = A(i,:) \cdot a$. Therefore, the approximated values for all inserted keys can be calculated as a decoding phase: $\hat{X} = A \cdot a = AA^T X$. □

From the random-projection results, the sketch is related to the compressed sensing problem, which seeks to recover the original signal X with a small number of linear measurements that collectively calculate the product $AA^T X$, where A denotes a sparse sign matrix. The sketch's goal is to approximate the original input with a small error. However, the sketch faces a more challenging context than the compressed sensing, since the linear matrix A is not preserved in the sketch, as each item is independently processed in the data stream. Thus we cannot directly apply the compressed-sensing results to recover the sketch.

We next show that we can quantify the expectation and the variance of the estimation based on the random-projection equivalence as follows:

Theorem 2. *Suppose that X are independent and identically distributed (iid) with expectation μ and variance σ^2. The expectation of the loss of each item $j \in [1,N]$ is $\frac{(N-1)\mu}{m}$, and the variance is $\frac{N-1}{m}\left(\sigma^2 + \left(1 - \frac{1}{m}\right)\mu^2\right)$*

Proof. The approximation loss of a sketch can be expressed as $\hat{X} - X = AA^T X - X = \left(AA^T - I\right) X$, where I denotes the identity matrix.

Recall that the product AA^T is an N-by-N symmetric matrix, where the diagonal entries are all set to ones since each item is mapped to only one bucket, and each non-diagonal entry (i,j) is one when i and j is mapped to the same bucket, and zero otherwise: $AA^T(i,j) = \begin{cases} 1, A(i,:) = A(j,:) \\ 0, else \end{cases}$.

Assuming that each item is mapped to a bucket uniformly at random, each entry (i,j) of the matrix AA^T has a bernoulli distribution, so that the probability $Pr\left[AA^T(i,j) = 1\right] = 1/m$ and $Pr\left[AA^T(i,j) = 0\right] = 1 - 1/m$ holds. Thus we write AA^T as $AA^T \sim Bernoulli(1/m)$.

Let $\Phi' = AA^T - I$ denote a normalized projection matrix, whose diagonal items are all set to zeros. Each non-diagonal entry in Φ' is the same as that in AA^T, both of which follow the Bernoulli distribution. We can derive the expectation $E\left[\Phi'(i,j)\right]$ as $\frac{1}{m}$, and the variance $Var\left[\Phi'(i,j)\right]$ as $\frac{1}{m} \cdot \left(1 - \frac{1}{m}\right)$.

First, the expectation of the approximation loss can be written as:

$$\begin{aligned} & E\left[\hat{X}(j) - X(j)\right] = E\left[\sum_{i=1}^{N-1} \Phi'(i,j) X(j)\right] \\ &= \sum_{i=1}^{N-1} E\left[\Phi'(i,j) X(j)\right] \\ &= \sum_{i=1}^{N-1} E\left[\Phi'(i,j)\right] E\left[X(j)\right] \\ &= \frac{(N-1)\mu}{m} \end{aligned}$$

The third line is due to the independence of the random-projection matrix and items.

Second, the variance of the approximation loss $Var\left[\hat{X}(j) - X(j)\right]$ can be derived as:

$$\begin{aligned} & Var\left[\sum_{i=1}^{N-1} \Phi'(i,j) X(j)\right] = \sum_{i=1}^{N-1} Var\left[\Phi'(i,j) X(j)\right] \\ &= \sum_{i=1}^{N-1} \begin{pmatrix} Var\left[\Phi'(i,j)\right] Var\left[X(j)\right] + \\ Var\left[\Phi'(i,j)\right] E\left[X(j)\right]^2 + \\ Var\left[X(j)\right] E\left[\Phi'(i,j)\right]^2 \end{pmatrix} \\ &= (N-1)\left(\frac{1}{m}\left(1 - \frac{1}{m}\right)\sigma^2 + \frac{1}{m}\left(1 - \frac{1}{m}\right)\mu^2 + \sigma^2\left(\frac{1}{m}\right)^2\right) \\ &= \frac{N-1}{m}\left(\sigma^2 + \left(1 - \frac{1}{m}\right)\mu^2\right) \end{aligned}$$

The second line is due to the independence between Φ' and X. □

We see that the approximation error of the sketch such as count-min [11] is proportional to the sum of the variance and the squared expectation of the network flow distribution.

A. Extensions

Further, several sketch structures such as count-sketch use signed mapping matrices. Each key is also randomly mapped to only one bucket, but a bucket accumulates a weighted value of the key by $+1$ or -1 randomly to the current counter. To account for the signed weights in the same framework, we extend the projection matrix to allow for negative items $\sum_{j=1}^{m} A(i,j) = +1$, or -1, where $A(i,j) \in \{-1,0,1\}$ for any entry (i,j). Thus, the sketch approximates the input with $AA^T X$ as before. The projection matrix AA^T can be written as:

$$AA^T(i,j) = \begin{cases} 1, A(i,:) = A(j,:) \\ -1, A(i,:) = -A(j,:) \\ 0, else \end{cases}$$

Lemma 1. *Suppose that X are iid with expectation μ and variance σ^2. The expectation of the loss of each item $j \in [1, N]$ is zero, and the variance is $\frac{N-1}{m} \cdot \left(\sigma^2 + \mu^2\right)$.*

Proof. The probability $Pr\left[AA^T(i,j) = 1\right]$ and $Pr\left[AA^T(i,j) = -1\right]$ are the same and both amount to $= 0.5/m$, and $Pr\left[AA^T(i,j) = 0\right] = 1 - 1/m$. From the probability distribution, we write the expectation $E\left[\Phi'(i,j)\right]$ as 0, and the variance $Var\left[\Phi'(i,j)\right]$ as $\frac{1}{m}$.

The approximation loss of this sketch can be expressed as $\hat{X} - X = AA^T X - X = \left(AA^T - I\right) X$, where I denotes the identity matrix. Let $\Phi' = AA^T - I$ denote the normalized projection matrix by setting all diagonal items to zeros. Next, the expectation of the approximation loss can be written as:

$E\left[\sum_{i=1}^{N-1} \Phi'(i,j) X(j)\right] = \sum_{i=1}^{N-1} E\left[\Phi'(i,j) X(j)\right]$
$= \sum_{i=1}^{N-1} E\left[\Phi'(i,j)\right] E\left[X(j)\right] = 0$

The third line is due to the independence of the random-projection matrix and items.

Second, the variance can be derived as:

$Var\left[\sum_{i=1}^{N-1} \Phi'(i,j) X(j)\right] = \sum_{i=1}^{N-1} Var\left[\Phi'(i,j) X(j)\right]$
$= \sum_{i=1}^{N-1} \begin{pmatrix} Var\left[\Phi'(i,j)\right] Var\left[X(j)\right] \\ +Var\left[\Phi'(i,j)\right] E\left[X(j)\right]^2 \\ +Var\left[X(j)\right] E\left[\Phi'(i,j)\right]^2 \end{pmatrix}$
$= (N-1) \cdot \left(\frac{1}{m} \cdot \sigma^2 + \frac{1}{m} \cdot \mu^2\right)$
$= \frac{N-1}{m}\left(\sigma^2 + \mu^2\right)$

The second line is due to the independence between Φ' and X. □

The proof follows from the similar procedure in Theorem 2. We see that although the expectation is reduced to zero, the variance is even greater.

IV. K-means Clustering Equivalence Analysis

Having quantified the accuracy and the variance of the sketch based monitoring process, we next present a cluster-preserving approach to improve the estimation accuracy and measurement concentration.

A. K-means Clustering

The K-means clustering problem that seeks to partition items to a set of groups with minimal variance has a close connection with data approximation [22]. It minimizes the variance of each cluster by finding a set of m points (called centroids) C such that the potential function is minimized

$$F(S) = \sum_{x \in S} \min_{c \in C} \|x - c\|^2, \tag{1}$$

where each centroid is equal to the average of the sum of items assigned to this cluster. Let $A \in \{0,1\}^{N \times m}$ denote the K-means clustering indicator matrix, with $A(i,j) = 1$ if i is mapped to the j-th cluster, and $A(i,j) = 0$ otherwise. We see that $AA^T X$'s i-th row represents the sum of items of the cluster assigned to i. Further, $A^T A$'s j-th diagonal entry represents the number of items in the j-th cluster. Thus, $A\left(A^T A\right)^{-1} A^T X$'s i-th row represents the centroid of i's assigned cluster. Consequently, we can represent the loss function of the K-means clustering with the mapping matrix A as follows:

$$\sum_{x \in S} \min_{c \in C} \|x - c\|^2 = \min_A \left\| X - A\left(A^T A\right)^{-1} A^T X \right\|_F^2 \tag{2}$$

Thus Eq (2) is structurally similar to a sketch that optimizes the similar objective $\min_A \left\| X - \hat{X} \right\| = \left\| X - \left(AA^T\right) X \right\|$. Further, the product $A^T A$ in Eq (2) has a unique structure: it is a diagonal matrix where non-diagonal entries are all zeros since each item is mapped to only one bucket, and each diagonal entry refers to the number of key-value pairs mapped to this corresponding bucket. Therefore, the matrix $\left(A^T A\right)^{-1}$ can be simply calculated by the inverse of each non-zero diagonal entry.

B. Novel Sketch

Based on these structural relationships, the K-means clustering solution leads to an important sketch design methodology: *clustering similar network flows to the same bucket helps reduce the approximation error.*

Let $\hat{X} = A\left(A^T A\right)^{-1} A^T X$ be a normalized sketch approximation:

- Insertion phase: Keep $a = A^T X$ by accumulating the value of each key to one of m buckets uniformly at random, and the diagonal matrix $A^T A$ by counting the number of unique keys mapped to each bucket with m counters;
- Query phase: Return $\hat{X} = A\left(A^T A\right)^{-1} a$, each of which is estimated by the division of the sum of values by the counter number of the bucket where this item is mapped to.

We next prove in Theorem 3 that, *averaging the bucket counter leads to an unbiased estimator*, and by clustering similar items together and mapping them to the same bucket array, the average estimator is bounded with probability proportional to the squared cluster's interval M.

Theorem 3. *Suppose that X are iid with expectation μ and variance σ^2. Assume that the difference $\left|X_i^j - \mu\right|$ is bounded by a positive constant M for any variable X_i^j.*

For a bucket j, let the items mapped to this bucket be represented as a set of independent and identically distributed variables: $\left\{X_i^j\right\}$. Let μ denote the expectation of the variable $\mu = E\left[X_i^j\right]$. Let n_j denote the number of items inserted to the j-th bucket, let $Y_j = \frac{\sum_i X_i^j}{n_j}$ denote the average based estimator.

Then Y_j is an unbiased estimator for any variable X_i^j. $Pr\left(|Y_j - \mu| \geq a\right) \leq \frac{M^2}{a^2 {n_j}^2}$ for a positive constant a. Moreover, $Pr\left(\left|Y_j - X_i^j\right| \geq a\right) \leq \frac{M^2}{(a-M)^2 {n_j}^2}$.

Proof. The expectation of Y_j is exactly the expectation of the variables. $E\left[Y_j\right] = \frac{1}{n_j} E\left[\sum_i X_i^j\right] = \frac{1}{n_j} \sum_i E\left[X_i^j\right] = \mu$

Therefore, Y_j is an unbiased estimator for $\left\{X_i^j\right\}$. Next, we bound the deviation degree of Y_j from its expectation as follows:

$$Var\left[Y_j\right] = E\left[\left(Y_j - \mu\right)^2\right] = E\left[\left(\frac{X^j}{n_j} - \mu\right)^2\right] =$$
$$E\left[\frac{1}{{n_j}^2}\left(\sum_i \left(X_i^j - \mu\right)\right)^2\right] \leq \frac{1}{{n_j}^2} E\left[M^2\right] = \frac{M^2}{{n_j}^2}$$

By Chebyshev's inequality, we bound the range of Y_j as :

$$Pr\left(\left|Y_j - \mu\right| \geq a\right) \leq \frac{Var\left[Y_j\right]}{a^2} \leq \frac{M^2}{a^2 {n_j}^2} \tag{3}$$

Second, the following inequality holds:

$$Pr\left(\left|Y_j - X_i^j\right| \geq a\right) = Pr\left(\left|Y_j - \mu + \mu - X_i^j\right| \geq a\right)$$
$$\leq Pr\left(\left(\left|Y_j - \mu\right| + \left|X_i^j - \mu\right|\right) \geq a\right)$$
$$= Pr\left(\left(\left|Y_j - \mu\right|\right) \geq a - \left|X_i^j - \mu\right|\right)$$
$$\leq Pr\left(\left|Y_j - \mu\right| \geq a - M\right)$$

which is less than or equal to $\frac{M^2}{(a-M)^2 {n_j}^2}$ by Eq (3). The second line is due to the triangle inequality condition ($\left|Y_j - \mu + \mu - X_i^j\right| \leq \left|Y_j - \mu\right| + \left|X_i^j - \mu\right|$). □

V. Sketch Design and Implementation

After presenting the cluster-preserving sketching methodology, we next present a new class of sketch called Locality-sensitive Sketch (LSS) that realizes the K-means clustering based normalized sketch structure.

A. Sketch Design

An LSS contains a number k of bucket arrays and a clustering model that consists of k cluster centers in a background and periodical process with up-to-date samples. A bucket array consists of a number of buckets, where each bucket has two fields: (i) A ValSum field that records the sum of values; (ii) A KeyCount filed that records the number of unique keys inserted to this bucket. Each bucket array corresponds to a cluster of similar items.

We map each item to the bucket array corresponding to the index of the nearest cluster center. For each incoming key-value item, we select the nearest cluster center with respect to the value, choose the corresponding bucket array, and insert the key-value item to a bucket indexed by the hash of the key. The bucket's ValSum counter is incremented by the incoming value, and the KeyCount increments by one iff the key is a new one.

Figure 1 plots an illustration of LSS data structure that is composed of two bucket arrays, each of which contains two buckets. Given an incoming key-value pair $(f5, 60)$, we compare $f5$'s value with two cluster centers and select the second bucket array denoted as array-2, since $f5$'s value is closer to 80. Then, we map $f5$ to the second bucket, and increment the bucket from $(83, 1)$ to $(143, 2)$. The average of this bucket is $\frac{143}{2} = 71.5$, which approximates the value of $f5$ with a small relative error.

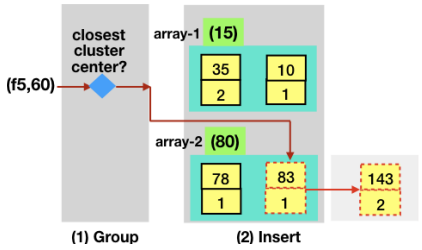


Fig. 1. Insert a key-value pair $(f5, 60)$ to an LSS instance. We assume an offline K-means cluster model with two cluster centers 15 and 80. One bucket array keeps items that are nearest to 15, while the other one keeps items that are nearest to 80.

B. Online Operations

1) Insert: LSS maps a key-value pair to the nearest cluster center, and accumulates the value to the corresponding bucket array. As the flow size is unknown before it completes, the ingestion process may publish multiple records for the same flow. Thus we have to efficiently identify the nearest cluster center for a dynamic flow and adjust the cluster mapping for changing flows.

We need to store the cluster index and the temporal value of the incoming key, in order to account for monotonically increasing flow counters. Thus we combine LSS with a membership-representation data structure, since a sketch structure does not keep the key-value membership itself, and querying a non-existing key is meaningless. As the Cuckoo table is shown to be more efficient than the Bloom filter at low false positives [19], [20], [21], we temporally keep a Cuckoo hash table of the form (fingerprint, (cluster-index, flow-counter)): the first field encodes the key, while the second and the third fields record the index to the nearest cluster and the current flow counter.

Step (i): we query the Cuckoo table with the item key to test whether it is a new key: If the flow has not been inserted to LSS, then we query the cluster model to find the nearest cluster center, and put it into the bucket array corresponding to the closest cluster center for this flow. We choose a random bucket by hashing the key with one hash function, and accumulate the incoming key-value pair to this bucket: (1) ValSum = ValSum + value; (2) KeyCount = KeyCount + 1 (if and only if key has not been hashed into this bucket array). Further, we also update the Cuckoo table to record the incoming item.

Step (ii): If the item key is duplicated, i.e., the network flow has sent some packets, then we obtain the recorded cluster index from the Cuckoo table, and select the corresponding bucket array with the same hash function, and adjust the mapped bucket with the item value: ValSum = ValSum + value.

Step (iii): Next, we check whether or not to move the flow to a new cluster: If the flow record is still nearest to the current cluster center, no movement should be made; otherwise, we need to move the flow record to the bucket array corresponding to the nearest cluster center: we delete the flow record from the current bucket array based on the record kept in the Cuckoo hash table (1) ValSum = ValSum - value, and (2) KeyCount

= KeyCount -1; then we insert this flow to the bucket array corresponding to the nearest cluster center similar to step (i).

2) Query: To query the value of a key on the LSS, we need to locate the bucket array. To that end, we query the Cuckoo hash table with the input key to get the cluster index of this key. Finally, we return the weighted value $\frac{ValSum}{KeyCount}$ as the approximated result.

Further, LSS supports diverse query tasks similar to existing sketch structures. We list the most representative ones:

(a) **Per-flow frequency and entropy query**. They track the traffic volume of each distinct flow, or count the flow bytes. LSS directly returns the size of a given flow. To query the size distribution of each inserted flow, we iteratively obtain approximation results with identifiers of inserted flows, then we build a list of approximated flow sizes as the flow size distribution. Similarly, we derive the entropy metric as the frequency distribution of approximated flow sizes.

(b) **Heavy hitters**. It finds top-K flows ingesting the most traffic volumes. For a given heavy-hitter detection threshold, we obtain approximated values of inserted flows from the LSS sketch, and select those exceeding the threshold as heavy hitters. Based on heavy hitters, we can also find flows spanning multiple windows that fluctuate beyond a predefined threshold, i.e., the heavy changes.

(c) **Flow cardinality**. LSS counts the exact number of distinct flows, since LSS maps each flow to a unique bucket. Therefore, we directly calculate the sum of KeyCount fields for each non-empty buckets, and return the accumulation result as the number of distinct flows.

C. Background Operation

Grouping flows to clusters should cope with online streams. To group similar network flows together, we need a clustering model. Further, we need to perform one-pass processing for online network flows. To that end, we reuse the samples kept in the Cuckoo hash table as the up-to-date samples, and provision the cluster model in a background process with these samples.

The clustering problem belongs to an unsupervised learning problem that clusters items to minimizes the intra-cluster variance. We obtain flow traces and train the K-means clustering mode in a periodical manner. We choose the well-studied K-means clustering method that represents clusters with a list of cluster centers. The K-means clustering model groups similar items together, by calculating a list of cluster centers as clustering reference points for items. We tune the number of clusters in order to obtain a fine-grained grouping model for the flow size distribution, which bounds the variance within each group in order to control the error variance of the average estimator.

D. Parameter Heuristics

We present parameter guidelines in order to trade off the estimation accuracy and the memory footprint.

Bucket-Array Size: We configure the size of a bucket array i based on the combination of three factors: (i) *Cluster entropy* H: For a cluster covering a short interval, a small bucket array is enough to achieve a low estimation error. This short cluster contains a low degree of uncertainty. The uncertainty of the cluster entries can be quantified with the **entropy**, $H_i = -\sum_{j \in S_i} f_j log f_j \in [0,1]$, where S_i denotes the set of unique items for the i-th cluster, f_j denotes the frequency of item j in this cluster. (ii) *Cluster center* μ: For a cluster with a large cluster center, it is likely to be the heavy tails of the flow's distribution, which needs more buckets to control the hash collisions. We quantify the cluster center with the ratio of each cluster center against the sum of all cluster centers, i.e., $\mu_i = \frac{\mu_i}{\sum_j \mu_j} \in [0,1]$. (iii) *Cluster density* d: For two clusters with approximately the same cluster uncertainty, a larger cluster need more buckets to reduce the estimation error. We quantify the cluster density with the ratio of the cluster entries to the total number of items, i.e., $d_i = \frac{d_i}{\sum_j d_j} \in [0,1]$. Let m denote the total number of buckets for LSS, H_i the entropy of the i-th cluster, g_i the i-th cluster center, and d_i the percent of items for the i-th cluster, we allocate $\frac{H_i d_i \mu_i}{\sum_j H_j d_j \mu_j} \cdot m$ buckets for the i-th bucket array. We derive these parameters through the offline K-means training process.

Number of Clusters: Finding the optimal number of K-means clusters is known to be NP-hard [23]. Thus we empirically determine the number of clusters based on sensitivity analysis that locates diminishing returns of the prediction accuracy.

Cuckoo table: For the sketch membership requirement, we set the number of hash functions to two and the number of slots per bucket to four in order to fit each bucket to a cache line (denoted as a (2,4) filter) [19]. For a f-bit digest, the upper bound of the false positive rate of an item is approximately $\frac{4*2}{2^f}$. We choose a 16-bit fingerprint with a false positive rate at 0.012%, which practically provides nearly-exact query. For a new key-value pair, we need two hash-function evaluations to visit the (2,4)-filter, and one hash-function evaluation to access the LSS sketch. To save the hashing complexity, we reuse the hash function across LSS bucket arrays, thus we only need three hash-function evaluations to insert an existing key-value pair.

E. Network Flow Monitoring

To illustrate the feasibility of the LSS sketch, we develop a monitoring application that implements the sketch in a modular framework based on a publish/subscribe (Pub/Sub for short) framework. A monitoring function atomically defines an intermediate stage in the monitoring process. The ingestion function colocates with the server or middlebox to aggregate packet streams to flowlet streams [24]. The sketching function maintains the LSS sketch based on flowlet streams. Finally, a query function performs monitoring queries on LSS sketches. (i) **Ingestion Stage**: The ingestion stage provides a device-independent key-value intermediate presentation model for network monitoring. It temporally aggregate packets at servers or middleboxes at line rates to flowlets [24], publishes key-value formatted flowlet-record messages in a batch mode to the Pub/Sub framework, and reset the hash table to accommodate for new entries.

(ii) **Sketching Stage**: The sketching component subscribes to one or multiple topics published by the ingestion components, then dynamically keeps an independent LSS sketch for each sliding window. For the sequence based sliding window, each LSS sketch keeps at most N flow records and is emitted to the sketch topic afterwards; while for the time based window, each LSS sketch is emitted after the interval ends. Upon receiving a flow record from a subscribed topic, the component selects the corresponding LSS sketch, groups this record towards the nearest cluster center, and inserts this record to the corresponding bucket array in the LSS sketch.

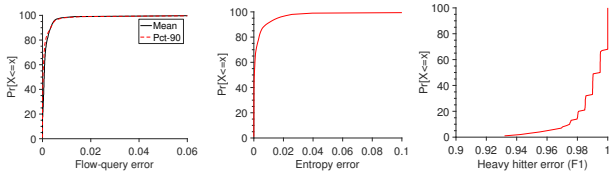
VI. Evaluation

A. Experimental Setup

We ran experiments on a multi-tenant private cluster to evaluate the locality-sensitive sketching and monitoring application. We set up the experiments on ten servers in two racks connected by a 10 Gbps switch, each server is configured as 8-core Intel(R) Xeon(R) CPU E5-1620, 47 GB memory, and Intel 10-Gigabit X540-AT2 network card. We choose the Pulsar messaging system originally created at Yahoo [25] as the Pub/Sub underlay. We set up the Apache Pulsar 2.2.0 Pub/Sub as a standalone service on a dedicated server. We split nine servers to two groups: (i) Six servers run the network ingestion component to produce flowlet records for port-mirrored traffic from the top-of-the-rack switch based on the Intel DPDK 16.04 interface, and publishes to the Pub/Sub framework; (ii) Three servers run the sketching component to maintain the LSS sketch for each of six ingestion servers.

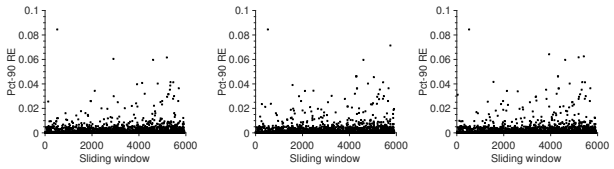
Default LSS Parameters: We set the sliding window to consist of 10,000 flows by default. We set the total number of buckets with respect to the number of flows in a sliding window. For a sliding window that consists of N flows, we set the default number m of LSS buckets to $0.1 \times N = 1,000$. For each LSS bucket, we set the storage size to four bytes (two bytes for each field). We set the default number of clusters to 30. Each cluster center is represented with four bytes. Thus an LSS with 1,000 buckets and 30 cluster centers takes 4.12KB. The offline traces take 10,000 flow samples, each sample is represented as four bytes, which take 40KB in total. We set the default heavy-hitter threshold to the 90-th percentile of the offline traces. We choose LSS' default parameters based on the diminishing returns via extensive sensitivity experiments in Subsection VI-C2.

Metrics: We choose three representative monitoring tasks to evaluate the sketch's performance, namely the flow-size query, the flow-entropy query, and the heavy-hitter query. We quantify the performance of the first two tasks with the relative error metric: defined as $|x_r - x_e| / (x_r)$, where x_r and x_e denoted the ground-truth metric and the estimated metric, respectively, and the last task based on the F1 score defined as the harmonic mean of the precision and the recall values, where the closer the F1 score towards one, the better the heavy-hitter estimator.



(a) Flow-size query (b) Flow entropy (c) Heavy hitters

Fig. 2. Performance of representative monitoring tasks on the testbed.



(a) Server 1 (b) Server 2 (c) Server 3

Fig. 3. 90-th percentile of flow-query relative errors on three servers running the query component.

B. Testbed Results

(i) **Flow-size query**: Next, we evaluate the relative error of estimated flow sizes. For each flow in each interval, we compare the estimated flow size against the ground-truth flow size. Figure 2(a) plots the CDFs of the mean relative errors. We see that the relative errors of over 90% of all estimations are smaller than 0.01. Since LSS accurately captures skewed flows with clustered bucket arrays.

(ii) **Flow-entropy query**: We next evaluate the accuracy of the entropy of the flow distribution for each interval. Figure 2(b) plots the CDFs of the relative errors of estimated flow entropies. We see that over 90% of estimations are smaller than 0.06, because of accurate estimations of flow sizes.

(iii) **Heavy hitter query**: Having shown that the flow entropy is accurately estimated, we next test the accuracy of estimated heavy hitters by calculating the F1 scores. Figure 2(c) plots the CDFs of F1 scores. We see that over 90% of tests are greater than 0.95. As LSS captures fine-grained flow distributions with clustered bucket arrays.

(iv) **Estimation stability**: We next test the estimation stability on the testbed. Figure 3 shows the 90-th percentiles of the flow-query relative errors of three query components. We see that most of the 90-th percentiles are zeros, while non-zero entries are smaller than 0.01 in most cases. Thus the estimation remains stably accurate across sliding windows.

(v) **Rate**: We next compare the relative performance of the ingestion component and the sketching component. Figure 4 shows the relative rate between the packet's arrival rate and the ingestion rate, as well as that between the flow-record arrival rate and LSS' insertion rate. We see that the arrival rate is orders of magnitude smaller than the corresponding consumption rate for both ingestion component and the sketching component. Since each component is tuned with respect to the input's arrival rate. We also constrain the size of the ingestion hash table and the LSS sketch in order to avoid CPU's L3-

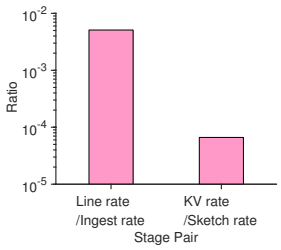


Fig. 4. Packet rate vs. the ingestion rate, and key-value arrival rate vs. sketch insertion rate.

cache misses.

C. Trace-driven Simulation

Our testbed is limited by the server scale. Therefore, we perform a real-world trace-driven experiment study with a public network dataset collected on February 18, 2016 at the Equinix-Chicago monitor by CAIDA [17] with 1799.7 million packets that last for one hour. We replay network traces and feed to the Apache Pulsar Pub/Sub software framework. We follow the default parameters of the testbed study.

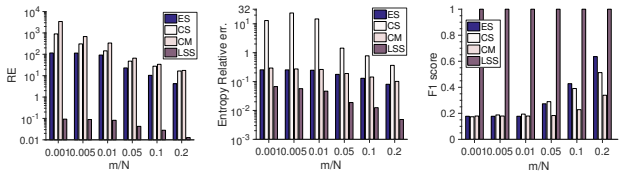
1) Comparison: (i) **Vary Memory**: We compare LSS with count-min (CM) [11], count-sketch (CS) [10], and Elastic Sketch (ES) [17] that are most related with our work. CM and CS are commonly used to find heavy hitters and perform flow queries [26], [15], [16]. We set the same memory footprint for all compared sketch structures. We follow the recommended parameter configuration for CM [11], CS [10] and ES [17].

Figure 5 plots the performance of the flow-size, flow-entropy, and heavy-hitter query tasks, as we vary the ratio between the number of buckets in LSS and the number of unique flows. We see that LSS significantly outperforms other sketch structures in all cases.

For the flow-size query tasks, LSS' relative error is over 10^3 to 10^5 times less than those of CS, CM and ElasticSketch, as the ratio between the number of buckets and the number of key-value pairs decrements from 10% to 0.1%. This is because LSS adapts to skewed flows with a cluster-preserving mapping process.

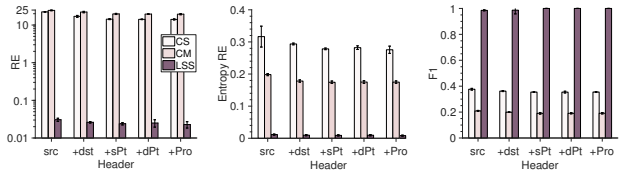
For the flow-entropy task, LSS' relative error is 4.3 to 13 times smaller than that of ElasticSketch, 4.8 to 14 times smaller than that of CM, and 70 to 200 times smaller than that of CS. ElasticSketch's accuracy is similar to that of CM in most cases, while CS has a much larger relative error than other methods. We can see that the flow-entropy task is less sensitive to flow-size errors, since the entropy depends on the frequency of each estimated value.

For the heavy-hitter task, LSS is close to optimal compared to the other methods, since LSS accurately estimates the size of each flow with a K-means clustering based recovery mechanism. ElasticSketch's accuracy is similar to CM and CS when the ratio $\frac{m}{N}$ is not greater than 0.1, and has a better F1 score than CS and CM afterwards, since ElasticSketch needs to keep large flows with the hash table and stores other flows to the count-min sketch.



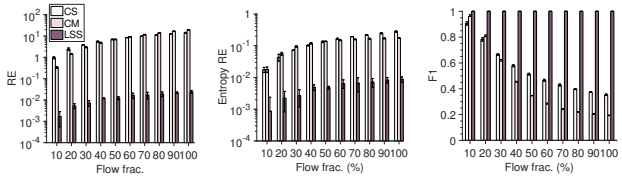
(a) Flow-size query (b) Flow entropy (c) Heavy hitters

Fig. 5. Accuracy of LSS and CM, CS, ES in terms of the ratios of the number of LSS buckets to the number of flows.



(a) Flow query (b) Entropy (c) Heavy hitters

Fig. 6. Accuracy of CS, CM and LSS as we expand the fields based on the 5-tuple model.



(a) Flow query (b) Entropy (c) Heavy hitters

Fig. 7. Performance of CS, CM and LSS as we vary the fractions of inserted flows.

(ii) **Varying Flow Fields**: Having shown that filtering large flows from the sketch is less effective than a K-means clustering based recovery of the locality-sensitive bucket arrays, we next compare CS, CM and LSS that do not filter flows with hash tables. We compare the stability of LSS with CS and CM, all of which do not filter heavy hitters like ElasticSketch. As shown in Figure 6, all sketch structures marginally improve the accuracy as we increment the key's input from one field to all five fields. LSS remains to be the most accurate sketch, since LSS captures fine-grained flow distributions with data-driven clustered buckets.

(iii) **Varying Flows**: Figure 7 shows that LSS remains fairly accurate across configurations, as we progressively add more flows in an epoch to the sketch. While CS and CM are severely affected due to hash collisions. Since LSS clusters similar flows to the same bucket array, and performs the error minimization for each bucket array.

(iv) **Generalization**: We next test whether LSS generalizes to different datasets. We use a network trace [27] collected on May 20, 2019 at the transit link of WIDE to the upstream ISP, with 14.0 million packets lasting for 899.99 seconds. From Figure 8, we see that LSS is 225 to 93425 times better than CM, CS, ES in the MAWI dataset. LSS consistently outperforms other methods in a new dataset.

2) Sensitivity: Having shown that LSS remains fairly accurate across different memory footprints, we next evaluate

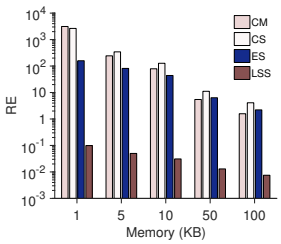
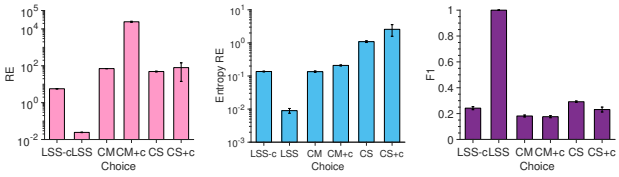
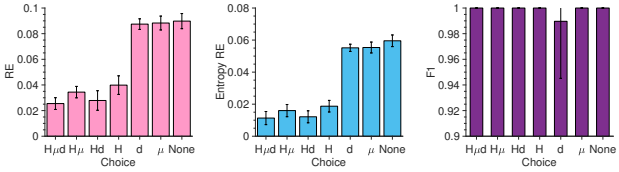


Fig. 8. Flow query on the MAWI dataset [27].



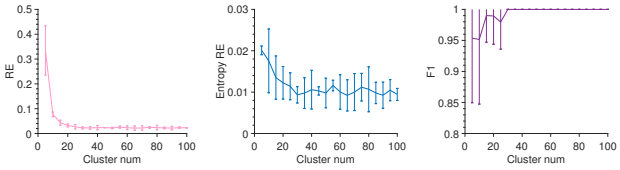
(a) Flow query (b) Entropy (c) Heavy hitters

Fig. 9. LSS performance vs. with or without clustering.



(a) Flow query (b) Entropy (c) Heavy hitters

Fig. 10. LSS performance vs. bucket-array policies.



(a) Flow query (b) Entropy (c) Heavy hitters

Fig. 11. LSS performance as a function of the numbers of clusters.

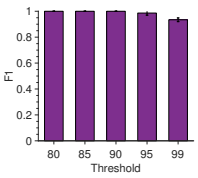
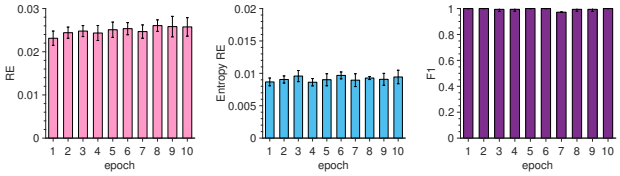


Fig. 12. F1 scores as a function of heavy-hitter thresholds.



(a) Flow query (b) Entropy (c) Heavy hitters

Fig. 13. LSS performance of different epochs by reusing the offline cluster model of the first epoch.

the sensitivity of LSS. We fix all but one parameters to the default configuration for the Testbed evaluation, and study the performance variation on the CAIDA dataset as we change a specific parameter. Performance conclusions generalize to other data sets.

(i) **With Or Without Clustering**: We first test the effectiveness of the clustering process for LSS. We skip the clustering process and map each flow record to all bucket arrays (denoted as LSS-c). We apply the clustering process to CM and CS (denoted as CM+c, CS+c, respectively). Figure 9 shows that LSS outperforms LSS-c by several times, thus the clustering is vital for LSS' performance. The clustering is useless for CM and CS, as both CM+c and CS+c degrades the prediction accuracy.

(ii) **Varying Bucket-array Policy**: We next test the effectiveness of the heuristics to configure the size of bucket arrays. Figure 10 shows that the combination of the cluster uncertainty (H), the cluster center (μ) and the cluster density (d) achieves high accuracy for three query tasks. We see that the cluster uncertainty is the most important metric, as removing the cluster uncertainty significantly degrades prediction accuracy.

(iii) **Number of Clusters**: Next, we evaluate LSS' accuracy with respect to the number of clusters. Figure 11 plots the variation of the estimation accuracy. We see that the estimation accuracy improves steadily with increasing numbers of clusters from two to ten. The diminishing returns occur when the number of cluster reaches 30.

(iv) **Varying Thresholds**: We also tested LSS' sensitivity to different heavy-hitter thresholds. Figure 12 shows the heavy-hitter performance degrades gracefully as we change the threshold percentiles from 80 to 99, since heavy hitters are more sensitive to estimation errors as we approach to tighter tails.

(v) **Offline Cluster-model Stability**: We tested LSS' sensitivity to offline clustering models by reusing the cluster centers that are trained with respect to the first epoch. Figure 13 shows that three monitoring tasks remain fairly accurate across epochs. Since the cluster model captures the global structure of the flow distribution.

VII. Conclusion

We have proposed a new class of sketch that is resilient to hash collisions, which groups similar items together to the same bucket array in order to mitigate the error variance, and optimizes the estimation based on the equivalence to the K-means clustering problem. To illustrate the feasibility of LSS sketch, we present a modular monitoring application that decomposes monitoring functions to disaggregated components. Extensive evaluation shows that LSS achieves close to a nearly optimal space-accuracy trade-off.

References

[1] Y. Fu, P. Barlet-Ros, and D. Li, "Every timestamp counts: Accurate tracking of network latencies using reconcilable difference aggregator," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 90–103, 2018.

[2] Y. Fu, D. Li, P. Barlet-Ros, C. Huang, Z. Huang, S. Shen, and H. Su, "A skewness-aware matrix factorization approach for mesh-structured cloud services," *IEEE/ACM Trans. Netw.*, vol. 27, no. 4, pp. 1598–1611, 2019.

[3] S. Shen, S. Hu, A. Iosup, and D. H. J. Epema, "Area of simulation: Mechanism and architecture for multi-avatar virtual environments," *TOMCCAP*, vol. 12, no. 1, pp. 8:1–8:24, 2015.

[4] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," *CoRR*, vol. abs/1710.11583, 2017. [Online]. Available: http://arxiv.org/abs/1710.11583

[5] T. Yang, L. Wang, Y. Shen, M. Shahzad, Q. Huang, X. Jiang, K. Tan, and X. Li, "Empowering sketches with machine learning for network measurements," in *Proceedings of the 2018 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2018, Budapest, Hungary, August 24, 2018*, 2018, pp. 15–20.

[6] Y. Fu, Y. Wang, and E. Biersack, "Hybridnn: An accurate and scalable network location service based on the inframetric model," *Future Generation Comp. Syst.*, vol. 29, no. 6, pp. 1485–1504, 2013.

[7] ——, "A general scalable and accurate decentralized level monitoring method for large-scale dynamic service provision in hybrid clouds," *Future Generation Comp. Syst.*, vol. 29, no. 5, pp. 1235–1253, 2013.

[8] Y. Fu and X. Xu, "Self-stabilized distributed network distance prediction," *IEEE/ACM Trans. Netw.*, vol. 25, no. 1, pp. 451–464, 2017.

[9] Y. Fu and E. Biersack, "MCR: structure-aware overlay-based latency-optimal greedy relay search," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 3016–3029, 2017.

[10] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, 2002, pp. 693–703.

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

[12] G. Cormode and M. Hadjieleftheriou, "Finding the frequent items in streams of data," *Commun. ACM*, vol. 52, no. 10, pp. 97–105, 2009.

[13] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, 2018, pp. 741–756.

[14] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, 2013, pp. 29–42.

[15] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 101–114.

[16] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 2017, pp. 113–126.

[17] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 561–575.

[18] Q. Huang, P. P. C. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, 2018, pp. 576–590.

[19] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, 2014, pp. 75–88.

[20] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, 2013, pp. 97–108.

[21] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy bloom filters for multi-set membership testing," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan-Les-Pins, France, June 14-18, 2016*, 2016, pp. 139–151.

[22] M. B. Cohen, S. Elder, C. Musco, C. Musco, and M. Persu, "Dimensionality reduction for k-means clustering and low rank approximation," in *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, 2015, pp. 163–172.

[23] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, pp. 264–323, 1999.

[24] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, 2016, pp. 129–143.

[25] A. P. develop group, "Apache pulsar framework," http://pulsar.apache.org, 2018.

[26] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "SCREAM: sketch resource allocation for software-defined measurement," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT 2015, Heidelberg, Germany, December 1-4, 2015*, 2015, pp. 14:1–14:13.

[27] W. M. WorkingGroup, "Packet traces from wide backbone," http://mawi.wide.ad.jp/mawi/, 2019.