Locality-Sensitive Sketching for Resilient Network Flow Monitoring

Yongquan Fu, Dongsheng Li, Siqi Shen, Yiming Zhang, Kai Chen*

ABSTRACT

Network monitoring is vital in modern clouds and data center networks for traffic engineering, network diagnosis, network intrusion detection, which 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. In addition, deploying the sketch involves fine-grained performance control and instrumentation.

This paper presents a locality-sensitive sketch (LSS) to be resilient to hash collisions. LSS proactively minimizes the estimation error due to hash collisions with an autoencoder based optimization model, and reduces the estimation variance by keeping similar network flows to the same bucket array. To illustrate the feasibility of the sketch, we develop a disaggregated monitoring application that supports non-intrusive sketching deployment and native network-wide analysis. 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%.

ACM Reference Format:
Yongquan Fu, Dongsheng Li, Siqi Shen, Yiming Zhang, Kai Chen. 2019. Locality-Sensitive Sketching for Resilient Network Flow Monitoring. In *Proceedings of* . ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 flow size estimation, flow distribution, heavy hitters. Recently, the self-running network proposal [17] 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. Traffic statistics tasks require advanced data structures and traffic statistical algorithms [9, 18, 37]. Many space- and time-efficient approaches [4, 10, 15, 21, 26–28, 30, 33–36, 45, 46, 51] 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.

Generally, a sketch builds a dimensional-reduction representation to approximately capture traffic counters. Its physical structure is a memory-efficient and constant-speed bucket array to accumulate incoming flow counters. Existing sketch structures [8, 11, 12] hash incoming packets to randomly chosen buckets and take the accumulated counter in these buckets as the estimator. Recently, OpenSketch [52], UnivMon [33], SketchVisor [21], ElasticSketch [50], and SketchLearn [22] further extend the generality of the sketch structure to support diverse monitoring tasks.

The sketch based monitoring approach faces two weaknesses. First, the estimation is sensitive to hash collisions, i.e., multiple keys are mapped to the same bucket, as this noisy bucket no longer returns exact results for any of inserted keys. Existing approaches typically aggregate multiple independent bucket arrays in order to relieve the degree of hash collisions. However, as we show in Figure 1, existing sketch structures such as count-sketch CS [8] and count-min sketch CM [12] are sensitive to hash collisions, where noisy estimators become the majority as the sketch becomes more compressive with respect to the number of inserted keys. Recently, ElasticSketch [50] and SketchLearn [22] track large flows with a hash table and separate large flows from the sketch structure. 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.

Second, the sketch based monitoring system needs fine-grained performance control and instrumentation. The need

*Yongquan Fu (yongquanf@nudt.edu.cn), Dongsheng Li (dsli@nudt.edu.cn), Siqi Shen (siqishen@gmail.com), Yiming Zhang (sdiris@gmail.com) are with Science and Technology Laboratory of Parallel and Distributed Processing, College of Computer Science, National University of Defense Technology, Kai Chen (kaichen@cse.ust.hk) is with SING Lab, Hong Kong University of Science and Technology

of coping with line-rate packets increases the resource contentions of the sketch structure with colocated deployed systems [21]. Further, modifying the sketching based monitoring applications introduces complicated debugging and performance issues.

To address the first weakness of existing sketches, we present a new class of sketch called locality-sensitive sketch (LSS for short) that is resilient to hash collisions. Our key observation is that: *if a noisy bucket contains similar values, then the average should approximate the original value well*. To that end, LSS approximately minimizes the estimation error based on the equivalence relationship between a sketch and a linear autoencoder model; furthermore, LSS reduces the variance of the estimation error by clustering similar key-value pairs based on a lightweight K-means clustering process [23].

We present two optimization techniques to make LSS practical for streamed monitoring requirements. First, the clustering process should be online to adapt the streamed flows. We exploit the temporally self-similar nature of the network traffic [29], by training an offline cluster model with traffic traces and mapping online flow records with trained cluster centers. Second, the insertion process should deal with incremental flow counters, since the flow size grows as packets are delivered. We cache the flow size in a Cuckoo filter [16], and remap the flow to the nearest cluster center.

We address the second weakness by presenting a disaggregated monitoring application in Section 4 that implements the LSS sketch in a non-intrusive approach and allows for native network-wide analytics. The framework decouples the line-rate packet streams from the sketching process for scalability purpose. An ingestion component at server or middlebox splits line-rate packet streams to flowlets [2, 24, 54] and aggregates real-time flowlet counters to reduce the monitoring traffic, and publishes them to a publish/subscribe framework. The flowlet-counter stream are subscribed by the sketch maintenance component that dynamically keeps the LSS sketch in a sliding window model. Streamed LSS sketches are subscribed by the query component to perform the network-wide analysis.

In Section 5, testbed shows that the framework adapts to line rates and provides accurate query results. Real-world trace-driven simulation confirms that LSS dramatically reduces the estimation error under the same memory footprint. 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%.

2 MOTIVATIONS

Each flow is typically represented as a key-value pair, where where the key is defined by a combination of packet fields, e..g, the 5-tuple representation, and the value summarizes the flow's statistics, e.g., packet numbers or byte counts. Existing sketch based monitoring applications work at the packet streams. For each incoming packet, a sketch based monitor inspects the packet header to extract the key and calculate the packet's value, then insert this record to the sketch data structure, which incrementally accumulates the value of the given key with one or multiple hash based bucket arrays. 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. We illustrate the insertion and query processes of existing sketches in details in Appendix A.2.

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 stream of sketch structures, where each sketch keeps a fixed number of key-value pairs, and a *query* component that transforms monitoring tasks to query primitives on the sketch.

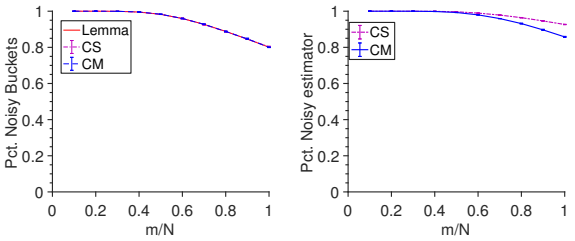
2.1 Resilience

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.

We quantify the expected number of noisy buckets that have hash collisions. We take Count-sketch (CS for short) [8] and count-min sketch (CM) [12] as examples, as both are probably two of the most popular sketch structures. We bound the expected number of buckets that suffer from hash collisions in Lemma 1.

LEMMA 1. *Assume that each key is mapped to a bucket in each bank uniformly at random. Let m denote the number of buckets, N the number of unique keys. For a sketch with c banks of bucket arrays, where each bucket array is of size $\frac{m}{c}$, the expected percent of noisy buckets is* $1 - e^{-cN/m} - \frac{cN}{m} \cdot e^{-c(N-1)/m}$.

The proof is due to the ball-bin model [3] that characterizes the expectation of the number of keys per bucket. The details are put in the Appendix A.1. We illustrate the hash collisions in Figure 1(a), the theoretical results match with empirical hash collisions. We can see that, the probability of hash collisions increases fast with decreasing ratios between the number of buckets and the number of unique keys. Figure



(a) Noisy buckets vs. $\frac{m}{N}$. (b) Noisy estimator vs. $\frac{m}{N}$.

Figure 1: The percents of noisy buckets as well as noisy estimators for count-sketch CS [8] and count-min sketch CM [12] as a function of the ratio of the number m of buckets to the number N of flows. The number c of banks is set to three based on recommended parameters. We plot the theoretical expected values and the empirical values based on a CAIDA trace (the statistics is introduced in section 5.3).

1(b) also confirms that hash collisions significantly degrade the effectiveness of CM and CS.

In addition, researchers have bounded the prediction accuracy based on the accumulation sum of the inserted items (See the Appendix A.2 for a detailed introduction). Unfortunately, the performance bounds are proportional to the accumulated sum of all items. Since network flows are typically highly skewed (refer to Figures 6 and 11), the sum of all items is still orders of magnitude larger than a single item.

2.2 Application

Having presented the hash-collision problem, we next discuss the implementional challenges faced by the sketch based monitoring applications for modern network management tasks. (i) **Scalability challenge**. Although a sketch only produces flow-level estimation results, existing monitoring applications feed packet-granularity streams to the sketch. As network is getting faster from 10 Gbps to 40 Gbps and beyond, more packets must be inspected for the same amount of time, which implies that the sketch's space and time complexity must be tightly controlled. Given n packets being in the same flow, the sketch still needs $O(n \times k)$ hash-function evaluations, where k denotes the number of bucket arrays in the sketch. (ii) **Fine-instrument challenge**. For heterogeneous and multi-tenant cloud data center networks, network monitoring desires to be non-intrusive and modular. It would maximize the generality under diverse environments. Existing sketching based solutions integrates these components into the deployed platform. Developers to perform fine-grained instrumentation to the operating system and programs, introducing complicated debugging and performance issues. Moreover, it is difficult to modify the sketching algorithm after deployment, due to the tight coupling with the program. However, the sketch component and the query component may undergo frequent updates, as the monitoring application has to meet diverse network management needs. Also, the network ingestion component requires developers to perform fine-grained instrumentation.

2.3 High-level Overview of Our Work

To minimize the effects of hash collisions while simultaneously keeping the simplicity of the hashing based data structures, this paper turns from passively tolerating noisy buckets to proactively recovering the noisy buckets. Suppose that all items are of the same value, we can see that the average of a bucket's accumulation returns the correct result for each of inserted items. Thus, in order to recover the noisy bucket, we need to map similar key-value pairs to the same bucket array, and recover the noisy bucket by averaging its accumulation counter. Combining these insights, we present a new class of sketch called locality-sensitive sketching or LSS for short. LSS has two distinct merits: (i) **Resilient to hash collisions**. LSS averages the bucket's counter to produce the estimator that is equivalent to optimize an autoencoder framework. (ii) **Locality-sensitive to reduce the variance of the estimator**. LSS learns the cluster structure of network flows based on transferred learning from offline traces, and clusters similar network flows to the same bucket array in order to minimize the estimation variance.

Example: Figure 2 illustrates the difference between the count-min sketch CM [12], count-sketch CS [8] and LSS. CM and CS insert each item to a random bucket in each bucket array. While LSS clusters similar items to the same bucket array and maps each item to only one bucket. From the query result in the rightmost column, we can see that LSS significantly reduces the estimation error compared to CM and CS. This is because LSS groups similar items together to reduce the estimation variance, and averages the bucket's counter to repair the prediction error. While CM and CS passively tolerate hash collisions.

Monitoring Application: To illustrate the feasibility of LSS and overcome the deployment hurdles, we present a disaggregated monitoring framework in Section 4 that implements LSS in a non-intrusive and modular monitoring application. The framework disaggregates the sketch components from the ingestion components, so as to allow for smooth modifications of sketch structures and coping with the underlying physical environments.

We reduce the monitoring traffic with a network ingestion component that splits real-time packet streams to flowlets [2, 24]. We temporally accumulate flowlet counters with a high-performance hash table based on Trumpet [39]. The

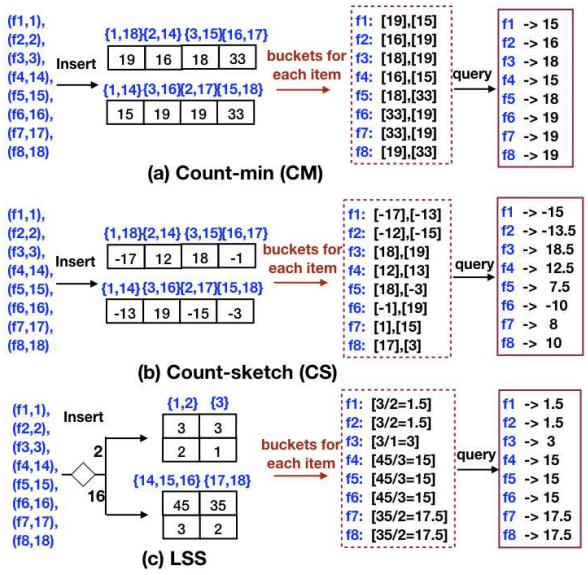


Figure 2: Sketch estimation with count-min sketch CM, count-sketch CS and our work LSS. Each sketch consists of two bucket arrays. The leftmost column represents the sketch after inserting a sequence of items (items that are mapped to each bucket are listed on the top of the bucket.); the middle column represents the value of each mapped bucket for each item (For LSS, we list the average of the mapped bucket for each item.); the rightmost column contrasts the original value with the estimated value for each kind of sketch.

hash table aggregates online packets by flow identifiers and accumulates flow counters, and reduces packet-processing delay by cache prefetching and batch processing. Suppose that the average packet size is 1,000 bytes, a flowlet has 100 flows and each flow has 100 packets on average, and a key-value pair size is 8 bytes. The traffic of a flowlet will be $100 \times 100 \times 1{,}000 = 10^7$ bytes. While the size of 100 flow counters will be $100 \times 8 = 800$ bytes. The monitoring traffic results in over 10^4 times reduction in volume.

3 LOCALITY-SENSITIVE SKETCH

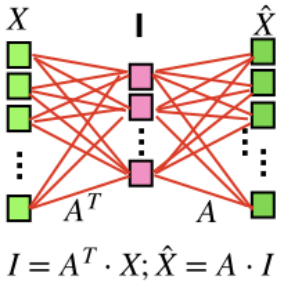
We present a new class of sketch called LSS that provides accurate approximations in a compact space. Table 1 lists key notations.

3.1 Framework

3.1.1 Autoencoder based Recovery. Having shown that hash collisions are inherent in any hashing based sketches, we next present an autoencoder guided approach to proactively minimizes the estimation error of noisy buckets.

Table 1: Key notations.

Notation	Meaning
N	Number of unique keys
X	Key-value streams
$\hat{X}$	Estimated key-value streams
A	Indicator matrix
$\{C_i\}$	Cluster centers
I	Bucket array
k	Number of cluster centers
m	Number of buckets



$$I = A^T \cdot X; \hat{X} = A \cdot I$$

Figure 3: Illustration of the autoencoder and the sketching process for Theorem 2.

We model the hash process of a sketch structure that randomly maps incoming items to a bucket array uniformly at random. Assume that a sketch consists of one bucket array for ease of analysis. 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 I 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 $l \neq j, l \in [1, m]$.

We next show in Theorem 2 that, the sketch is mathematically equivalent to a linear autoencoder[1] : *The insertion process corresponds to the encoding phase of the autoencoder; the query process corresponds to the decoding phase of the autoencoder.*

THEOREM 2. *A sketch with one bucket array is equivalent to a linear autoencoder: the insertion process corresponds to an encoding phase* $I = \left(A^T X\right)$, *while the query phase corresponds to a decoding phase* $\hat{X} = A \cdot I$.

[1] An autoencoder [5] is a neural network that takes a vector x at the input, and reconstructs the input vector at the output layer. An autoencoder consists of an encoder that maps the input to a hidden layer $f = \sigma_e(W_e x + b_e)$, and a decoder that reconstructs the input as $\hat{x} = \sigma_d(W_d f + b_d)$, where b_e, b_d serve as bias variables, W_e and W_d are weight matrices that map the input to the hidden layer, and the hidden layer to the output, respectively. Generally, an autoencoder enforces parameter sharing $W_e = W_d^T$ (called weight tying) to avoid overfitting.

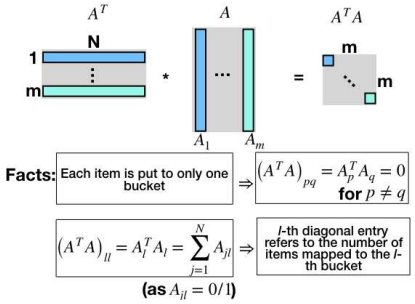


Figure 4: The product A^TA yields a diagonal matrix where non-diagonal entries are all zeros, and diagonal entries refer to the numbers of key-value pairs mapped to the corresponding buckets.

The proof is due to the algebraic transformation of the insertion and the query process of the sketch. The details are put in the Appendix B.1. Figure 3 illustrates the mathematical equivalence between the sketching process and the autoencoder. The immediate result is that we can formulate an optimization framework for the sketch. Assume that the mapping matrix is a variable, we can formulate an optimization problem:

$$\min_A \left\|\hat{X} - X\right\| = \left\|\left(AA^T\right)X - X\right\| \tag{1}$$

To derive optimized solution for Eq (1), assuming that the mapping matrix is a random variable, the Principle Component Analysis (PCA) [5] finds a dimensional-reduction hyperplane with the smallest reconstruction error for a one hidden-layer autoencoder. Unfortunately, it will require to keep the whole stream X, which is infeasible for network monitoring context. Moreover, PCA calculates a dense matrix A, while the sketch enforces the matrix A to be ultra-sparse.

Although we cannot derive the optimized mapping matrix A based on the autoencoder, *we can asymptotically minimize the reconstruction error by relaxing the mapping matrices for the insertion and query processes*. Specifically, let A denote the mapping matrix for the insertion matrix, let B denote the mapping matrix for the query phase, we find an optimized matrix B with respect to A in Eq (1):

$$\min_B \left\|X - BA^TX\right\| = \min_B \left\|\left(I - BA^T\right)X\right\| \tag{2}$$

We can derive a closed-form solution of B for Eq (2) as:

$$B = A\left(A^TA\right)^{-1} \tag{3}$$

Although the mapping matrix A is still random, the product $C = A^TA$ is a diagonal matrix where the i-th diagonal entry counts the number of items mapped to the i-th bucket, as illustrated in Figure 4.

Suppose that we keep the diagonal matrix C with $O(m)$ space, we obtain an approximately optimization representation:

$$\hat{X} = A(C)^{-1}A^TX \tag{4}$$

Eq (4) can be formulated as an encoder-decoder phase: *(a) encoder: $I = A^TX$; (b) decoder: $\hat{X} = A(C)^{-1}I$*. This formulation inspires a new class of sketch structure that proactively minimizes the estimation error, by averaging each bucket in the bucket array I with the numbers of keys mapped to this bucket.

3.1.2 Variance Minimization. Having presented the autoencoder inspired sketch-recovery process, we next present a lightweight approach to minimize the variance of the prediction error because of the skewed distributions of items. Our key insight is that, *to minimize the variance, we have to group keys with similar values to the same bucket array*. The grouping requirement belongs to an unsupervised learning problem that clusters items to minimizes the intra-cluster variance, and maximizes the inter-cluster dissimilarity. We choose the well-studied K-means clustering method [23] to automatically find cluster centers, which represents clusters with a list of cluster centers. Then, we map flow records to the nearest cluster centers. More details of the K-means model can be found in the Appendix B.2.

Specifically, we can bound the estimation variance of the autoencoder based estimator with clustered inputs. 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. Suppose that we have grouped items by the similarity of values, we assume that the difference $\left|X_i^j - \mu\right|$ is bounded by a positive constant M for any variable X_i^j. We next bound the range between the average estimator Y_j and the ground-truth value X_i^j for the j-th bucket as follows:

THEOREM 3. *Y_j is an unbiased estimator for any variable X_i^j. $Pr\left(\left|Y_j - \mu\right| \geq a\right) \leq \frac{M^2}{a^2 n_j{}^2}$ for a positive constant a. Moreover, $Pr\left(\left|Y_j - X_i^j\right| \geq a\right) \leq \frac{M^2}{(a-M)^2 n_j{}^2}$.*

The proof is due to the concentration bounds of the Chebyshev's inequality, whereas the details are put in the Appendix B.3. We can see that after clustering the input to similar groups, the average estimator not only produces an unbiased result, but also keeps the estimator bounded with probability proportional to the squared cluster's interval M.

3.1.3 Sketch Membership. A sketch structure does not keep the key-value membership itself. However, querying a

non-existing key is meaningless, thus in practice, a sketch is usually combined with a membership-representation data structure, e.g., Bloom filter [47], d-left hash table [7] or a Cuckoo Filter [16]. The cuckoo hash table that inserts items based on cuckoo hashing, is shown to be more efficient than the Bloom filter at low false positives [13, 16, 53]. Thus we keep the membership with a cuckoo hash table[2] that supports efficient insertion and deletion of items. 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) [13, 16, 53]. 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.

3.2 LSS Structure and Operations

For ease of presentation, we present the basic idea of the LSS with simplifications. We assume that the input is represented as a list of unique key-value pairs, thus no duplication exists for any pair of keys. In the next subsection, we propose practice approaches to deal with duplicated items.

An LSS is organized as a number k of bucket arrays. 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 represent the clusters with k cluster centers. LSS maps each item to only one bucket array that corresponds to the nearest cluster center for this item.

As shown in Figure 2 (c), 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.

An LSS provides *group*, *insert*, and *query* operators to support the dimension-reduction representation of key-value streams.

3.2.1 Group. LSS groups similar items together, by calculating a list of cluster centers as clustering reference points for items. As discussed in the above subsection, we choose the well-known K-means clustering method to find cluster centers due to its simplicity and competitive performance, although more complex clustering methods may achieve slightly better performance.

Grouping flows to clusters should cope with online streams. Training the cluster model for packet streams is infeasible, in contrast, we need to perform one-pass processing for online network flows: we initialize cluster centers a priori, and map streamed network flows with initialized cluster centers. Fortunately, it is well known that the flow-size distributions are self-similar [6, 29], thus the cluster structure is transferrable. Our experiments in section 5 also confirm this observation. Therefore, we find clustering patterns on packet traces in an offline manner, then group online flows with obtained clustering patterns in the offline phase.

Offline Training: We obtain flow traces and train the K-means clustering mode in an offline manner. 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.

Online Mapping: To speed up the mapping process, we sort the cluster centers a priori, which takes $O(k \log k)$ time, where k denotes the number of cluster centers. Then, for each online key-value pair, we directly map this pair to the nearest cluster center with a binary-search process on sorted cluster centers in time $O(\log k)$. Finally, based on the index of the cluster center, we map this key-value pair to the corresponding bucket array in the LSS sketch.

3.2.2 Insert. LSS maps a key-value pair to the nearest cluster center, and accumulates the value to the corresponding bucket array. Algorithm 1 shows the steps of inserting a new key-value pair into LSS. First, we locate the bucket array corresponding to the nearest cluster center to the incoming key-value pair. Second, we choose a random bucket by hashing the key with one hash function. Third, we accumulate the key-value pair to the bucket: (i) ValSum = ValSum + value; (ii) KeyCount = KeyCount + 1(if and only if key has not been hashed into this bucket array). Finally, we store the cluster index of the incoming key for network-wide key-value queries.

Complexity: We represent the cluster-index field with eight bits that indexes $2^8 = 256$ clusters in total. Each key-value pair is mapped to only one bucket in a LSS sketch, which involves only one hash-function evaluation. Accessing a (2,4)-filter involves two hash-function evaluation.

3.2.3 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

[2]Briefly, the Cuckoo Filter inserts items based on cuckoo hashing, which uses multiple hash functions to map each item to candidate buckets. For an incoming item, if one of candidate buckets has empty slots, then we calculate the digest of this item and put the digest to one empty slot; otherwise, we pick one nonempty slot and displace its digest to its alternative candidate bucket, then we put the new item to this slot. The displaced digest may further "kick out" other digests until no displacements of existing digests, or reaching a maximum number of displacements.

Algorithm 1: Insert a non-duplicated Key-value pair to LSS. I denotes the bucket array. $h(\cdot)$ denotes the hash function. C_i denotes the i-th cluster center. h_{finger} denotes the hash function for the fingerprint calculation

```
Insert(κ, v)
i_κ = argmin_i ||v − C_i||;
bucket = I_{i_κ}[h(κ)];
bucket.ValSum += v, bucket.KeyCount +=1;
Store (h_finger(κ), i_κ) to the Cuckoo hash table;
```

Algorithm 2: Key-value query for a key κ.

```
Query(κ)
Query the Cuckoo hash table to get the cluster index i_κ for κ;
bucket = I_{i_κ}[h(κ)];
return bucket.ValSum / bucket.KeyCount ;
```

the approximated result. Algorithm 2 summarizes the steps for the query process.

Time Complexity: Querying a (2,4)-filter needs two hash-function calculations. Obtaining the LSS bucket's counter needs one hash-function calculation. Thus the time complexity of the query process is the same as that of the insertion process.

3.3 Handling Duplicated Online Streams

LSS requires to always keep a flow within the nearest cluster. 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 propose a duplication adaptive maintenance method to dynamically maintain flow records towards the nearest cluster. Algorithm 3 summarizes the duplication-adaptive maintenance process. If the flow has not been inserted to LSS, then we put it into the bucket array corresponding to the closest cluster center; otherwise, the flow has been mapped to LSS, we locate the mapped cluster of this flow, select the corresponding bucket array, and then increment the flow record at the mapped bucket. Finally, 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, and insert it to the bucket array corresponding to the nearest cluster center.

Complexity: 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

Algorithm 3: Duplication-adaptive LSS maintenance.

```
InsertDuplicate(κ, v, CH)
Query CH with κ to get its cluster index i_κ;
if i_κ is NULL then
    i_κ = argmin_i ||v − C_i||;
    I_{i_κ}[h(κ)].ValSum += v, I_{i_κ}[h(κ)].KeyCount +=1;
    Store (h_finger(κ), (i_κ, v)) to CH;
else
    Accumulate v to the current value v_κ of κ in CH;
    I_{i_κ}[h(κ)].ValSum += v;
    Retrieve the value v*_κ for the key κ in CH ;
    Find the nearest cluster center i* for v*_κ;
    if i* ≠ i_κ then
        I_{i_κ}[h(κ)].ValSum -= v*_κ, I_{i_κ}[h(κ)].KeyCount -=1;
        I_{i*}[h(κ)].ValSum += v*_κ, I_{i*}[h(κ)].KeyCount +=1;
        Store (h_finger(κ), (i*, v*_κ)) to CH;
```

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. During the insertion process, we temporally keep a Cuckoo hash table CH; while after the sketch terminates the insertion process, we squeeze the Cuckoo hash table to keep only the fingerprint and the cluster index.

3.4 LSS Parameters

We next 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 $\frac{H_i d_i \mu_i}{\sum_j H_j d_j \mu_j} \cdot m$

Figure 5: Disaggregated monitoring architecture VS. monolithic architecture.

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.

4 DISAGGREGATED MONITORING APPLICATION

Having presented the LSS sketch, to illustrate the feasibility of the locality-sensitive sketching, we next propose a monitoring application that implements LSS in a modular and non-intrusive framework.

As the network flows are infinite in essence, recent network flows are usually more important. In a sliding window model, each packet is sequentially and independently processed in a one-pass manner. For a sequence based window, it processes past N items; while for a time based window, it processes items in a past time period. The framework supports both sliding window models, although the time based window may ingest too many flows during bursty periods, while the sequence based window is more robust in this case.

As shown in Figure 5, the proposed monitoring architecture splits the monitoring application into non-coherent ingestion, sketching, query runtime functions that can be horizontally scaled in the data center. A monitoring function atomically defines an intermediate stage in the monitoring process. Each ingestion function colocates with the server or middlebox to aggregate packet streams to flowlet streams. Second, each sketching function maintains the LSS sketch based on ordered flowlet streams. Third, each query function performs monitoring queries on LSS sketches. Finally, we keep system configuration up to date via a global coordinator. A publish/subscribe (Pub/Sub for short) framework[3] delivers ordered streaming messages across monitoring functions. We choose the Pulsar messaging system originally created at Yahoo [14] as the Pub/Sub underlay.

(i) **Ingestion Stage**: The ingestion stage provides a device-independent key-value intermediate presentation model for network monitoring. It splits packets at servers or middleboxes at line rates to flowlets, and publishes key-value formatted flowlet-record messages in a batch mode to the Pub/Sub framework. When a packet arrives, we look up the hash table with a key calculated based on the hash of its 5-tuple information: If the key is in the hash table, then we accumulate the per-flow counter with this packet's information; If the key is not in the hash table and there exists empty entries in the table, then we put the key and the corresponding per-flow counter to the hash table; Otherwise, we publish all accumulated flow counters in batch, 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. We handle duplicated flow records based on subsection 3.3.

(iii) **Query Stage**: 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.

[3]The Pub/Sub topic framework provides seamless messaging supports for monitoring functions, which represents message flows among disaggregated components. One or multiple producer entities publish messages towards the same topic, then the Pub/Sub messaging framework delivers ordered messages to consumers subscribed to the same topic.

Moreover, the framework supports a network-native query interface. Each sketching component publishes to the same sketching topic, then a centralized query component subscribes to this sketching topic and performs queries on received sketches. Moreover, some network management tasks may need to query historical sketches during a time interval. To that end, the query component stores the received sketch and its arrival timestamp in a persistent storage, and then lists sketches within a given time interval.

5 EVALUATION

5.1 Experimental Setup

We ran experiments on a multi-tenant private cluster to evaluate the locality-sensitive sketching and disaggregated monitoring. The cluster is shared by tens of different clients. 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 set up the Apache Pulsar 2.2.0 Pub/Sub as a standalone service on a dedicated server. We configure Apache Pulsar with the default setup. 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. Each LSS sketch is published to the Pub/Sub framework after it accumulates 10,000 flows.

Default LSS Parameters: We set the sliding window to consist of 10,000 flows by default. We dimension 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 evaluation in Subsection 5.3.2.

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.

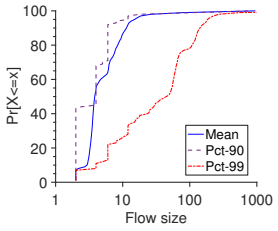


Figure 6: Traffic distribution of the testbed.

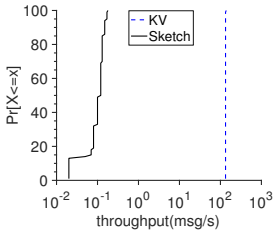


Figure 7: Publishing throughput for the ingestion and sketching components.

5.2 Testbed Results

We summarize the ground-truth distribution of flows captured in the private cluster. Figure 6 plots the cumulative distribution functions (CDF) of network flows in each interval. We see that the mean and the 90-th percentile of the network flows are less than 10 for over 90% of all traces. However, the 99-th percentiles of traces span over three orders of magnitudes, thus the network flow distribution is highly skewed, which is similar to the CAIDA traffic trace in the next subsection.

5.2.1 Disaggregated Performance. We test the publishing throughput for the ingestion and the sketching components. Each flowlet message consists of 1,000 temporary flow counters in the hash table, while each sketching message consists of one LSS sketch. Figure 7 plots the CDFs of the message throughputs of the ingestion components and those of the sketching components. We see that the throughput of the ingestion component is nearly three orders of magnitudes larger than that of the sketching component, since the ingestion component depends on the line rates, while the sketching component depends on the readiness of the sliding window.

We next compare the relative performance of the ingestion component and the sketching component. Figure 8 shows the

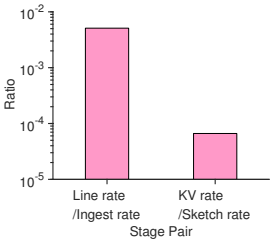
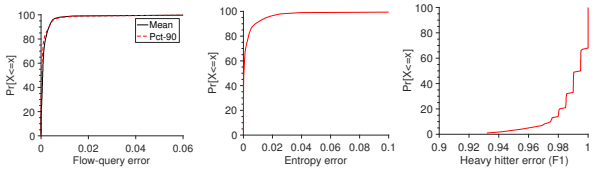


Figure 8: Packet rate vs. the ingestion rate, and key-value arrival rate vs. sketch insertion rate.



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

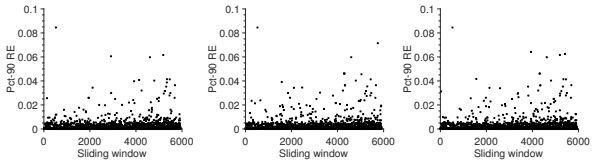
Figure 9: Performance of representative monitoring tasks on the testbed.

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-cache misses.

5.2.2 Sketching Performance. (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 9(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 9(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 9(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.



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

Figure 10: 90-th percentile of flow-query relative errors on three servers running the query component.

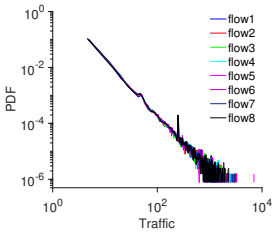


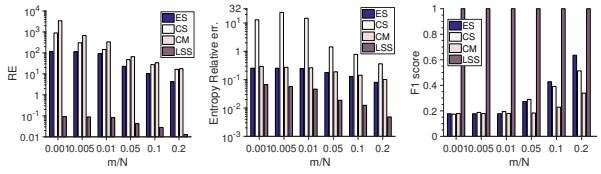
Figure 11: The probability density distributions (PDFs) of flow sizes in the CAIDA data set. We separate records to eight contiguous epochs denoted from flow1 to flow8, and plot the PDFs for each epoch. We see that the PDFs of different parts match well with each other.

(iv) **Estimation stability**: We next test the estimation stability on the testbed. Figure 10 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.

5.3 Trace-driven Simulation

Our testbed is limited by the server scale. Therefore, we perform a real-world trace-driven experiment study. We replay network traces collected on February 18, 2016 at the Equinix-Chicago monitor by CAIDA [50], and feed to the Apache Pulsar Pub/Sub software framework. We follow the default parameters of the testbed study. We calculate K-means cluster centers by randomly sampling 10,000 flows from the trace. Figure 11 shows that different traces share nearly identical power-law flow-size distributions, thus the flow size distribution is not only skewed, but also temporally self-similar across epochs. This is because the self-similarity is a latent property in the network traffic [6, 29].

5.3.1 Comparison. (i) **Vary Memory**: We compare LSS with count-min (CM) [12], count-sketch (CS) [8], and Elastic Sketch (ES) [50] that are most related with our work. CM and CS are commonly used to find heavy hitters and perform flow queries [21, 33, 38]. We set the same memory footprint for



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

Figure 12: Accuracy of LSS and CM, CS, ES in terms of the ratios of the number of LSS buckets to the number of flows.

all compared sketch structures. We follow the recommended parameter configuration for CM [12], CS [8] and ES [50].

Figure 12 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 locality sensitivity and autoencoder based error minimization.

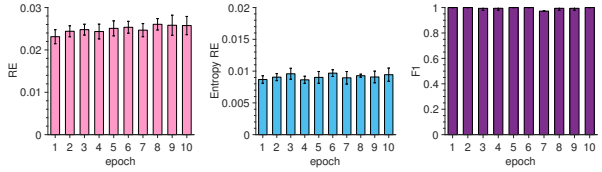
For the flow-entropy task, LSS' relative error is 4.3 to 13 times smaller than that of ElasticSketch, 4.8 to 14 times smaller than that of CM, and 70 to 200 times smaller than that of CS. ElasticSketch's accuracy is similar to that of CM in most cases, while CS has a much larger relative error than other 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 an autoencoder 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.

(ii) **Varying Flows**: Having shown that filtering large flows from the sketch is less effective than an autoencoder based recovery of the locality-sensitive bucket arrays, we next compare CS, CM and LSS that do not filter flows with hash tables. Figure 13 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.

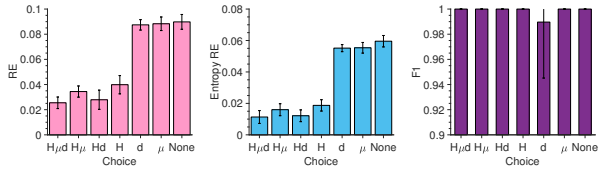
(a) Flow query (b) Entropy (c) Heavy hitters

Figure 13: Performance of CS, CM and LSS as we vary the fractions of inserted flows.



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

Figure 14: LSS performance of different epochs by reusing the offline cluster model of the first epoch.



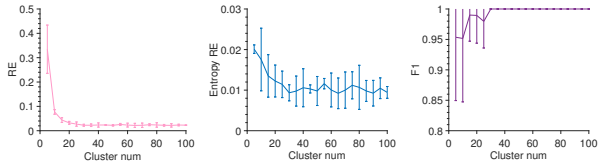
(a) Flow query (b) Entropy (c) Heavy hitters

Figure 15: LSS performance vs. bucket-array policies.

5.3.2 Sensitivity. Having shown that LSS remains fairly accurate across different memory footprints, we next evaluate the sensitivity of LSS. We fix all but one parameters to the default configuration for the Testbed evaluation, and study the performance variation as we change a specific parameter.

(i) **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 14 shows that three monitoring tasks remain fairly accurate across epochs. Since the cluster model captures the global structure of the flow distribution.

(ii) **Varying Bucket-array Policy**: We next test the effectiveness of the heuristics to configure the size of bucket arrays. Figure 15 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.



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

Figure 16: LSS performance as a function of the numbers of clusters.

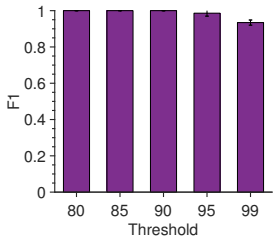


Figure 17: F1 scores as a function of heavy-hitter thresholds.

(iii) **Number of Clusters**: Next, we evaluate LSS' accuracy with respect to the number of clusters. Figure 16 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 17 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.

6 RELATED WORK

Modern network measurement systems typically rely on programmable frameworks to perform diverse monitoring tasks. First, the end based approach such as Trumpet [39], deTector [42], Confluo [25] relies on edge servers to perform end-to-end packet-stream monitoring. To increase in-network visibility, an in-network based approach such as OpenSketch [52], Planck [43], Marple [40], Everflow [54], FlowRadar [31], UnivMon [33], Sonata [19] combines the software-defined framework and the programmability of switches to track fine-grained traffic statistics. A hybrid approach such as PathDump [48], SwitchPointer [49] and [44] combines the resource-intensive end servers and the in-network visibility of switches. We present a disaggregated monitoring framework that can be incorporated with end hosts and programmable switch based systems, in order to maximize the network visibility and support non-intrusive monitoring.

Existing sketches typically choose to tolerate hash collisions with space redundancy. For instance, state-of-the-art sketch structures [8, 11, 12, 50] choose the least affected bucket from multiple independent bucket arrays. Recently, ElasticSketch [50] 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 [8, 11, 12]. Unfortunately, as heavy hitters only represent a small fraction of items, the count-min sketch is still sensitive to hash collisions. Our work proactively mitigates the downsides of the hash collisions with locality-aware clustering and bucket averaging techniques.

The sketch structure has been augmented in several dimensions. UnivMon [33] uses an array of count sketch to meet generic monitoring tasks. SketchVisor [21] augments the sketch with a fast-path ingestion path to tolerate bursty traffic. SketchLearn [22] 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 [50] based on inferred flow distributions. Although our work is orthogonal to these studies, the LSS sketch structure can be combined to these frameworks to improve the sketching efficiency.

Several studies propose to keep network flow statistics in a hash table [1, 39] at end hosts, whereas the storage requirement is on the order of the number of network flows. As network flows arrive continuously, the hash table based monitoring application incurs expensive memory costs that reduce the available resources for colocated tenants. Since for multi-tenant data centers, the monitoring application has to control its resource usage to maximize the available resource to meet tenants' needs. Moreover, the hash table needs to dynamically adjust the data structure when hash collisions occur, i.e., multiple keys are mapped to the same bucket, with linear hashing [32], Cuckoo hashing [41], or hopscotch hashing [20]. The hash table is agnostic of the self-similarity structure of flow counters, while the sketch can exploit this property to compress the flow counters to a constant-size array.

7 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 an autoencoder model to minimize the estimation error. We showed that LSS is equivalent to a linear autoencoder that minimizes the recovery error. To illustrate the feasibility of LSS sketch, we present a disaggregated monitoring application that decomposes monitoring functions to disaggregated components, which allows for

non-intrusive sketch deployment and native network-wide analytics. Extensive evaluation shows that LSS achieves close to optimal performance with a tiny memory footprint, which generalizes to diverse monitoring contexts.

REFERENCES

[1] O. Alipourfard, M. Moshref, Y. Zhou, T. Yang, and M. Yu. A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research, SOSR 2018, Los Angeles, CA, USA, March 28-29, 2018*, pages 18:1–18:14, 2018.

[2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 503–514, 2014.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1):180–200, 1999.

[4] R. Ben-Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 127–140, 2017.

[5] Y. Bengio, A. C. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, 2013.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 267–280, 2010.

[7] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *Proceedings of the ACM SIGCOMM 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Pisa, Italy, September 11-15, 2006*, pages 315–326, 2006.

[8] 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*, pages 693–703, 2002.

[9] B. Choi, J. Park, and Z. Zhang. Adaptive packet sampling for accurate and scalable flow measurement. In *Proceedings of the Global Telecommunications Conference, 2004. GLOBECOM '04, Dallas, Texas, USA, 29 November - 3 December 2004*, pages 1448–1452, 2004.

[10] G. Cormode. Data sketching. *Commun. ACM*, 60(9):48–55, 2017.

[11] G. Cormode and M. Hadjieleftheriou. Finding the frequent items in streams of data. *Commun. ACM*, 52(10):97–105, 2009.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004, Proceedings*, pages 29–38, 2004.

[13] 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*, pages 139–151, 2016.

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

[15] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[16] 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*, pages 75–88, 2014.

[17] N. Feamster and J. Rexford. Why (and how) networks should run themselves. *CoRR*, abs/1710.11583, 2017. URL http://arxiv.org/abs/1710.11583.

[18] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 139–152, 2015.

[19] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 357–371, 2018.

[20] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 350–364, 2008.

[21] 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*, pages 113–126, 2017.

[22] 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*, pages 576–590, 2018.

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

[24] S. Kandula, D. Katabi, S. Sinha, and A. W. Berger. Dynamic load balancing without packet reordering. *Computer Communication Review*, 37(2):51–62, 2007.

[25] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 2019. URL https://www.usenix.org/conference/nsdi19/presentation/khandelwal.

[26] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM Internet Measurement Conference, IMC 2003, Miami Beach, FL, USA, October 27-29, 2003*, pages 234–247, 2003.

[27] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pages 177–188, 2004.

[28] A. Lall, V. Sekar, M. Ogihara, J. J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance 2006, Saint Malo, France, June 26-30, 2006*, pages 145–156, 2006.

[29] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, 1994.

[30] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina. Detection and identification of network anomalies using sketch subspaces. In *Proceedings of the 6th ACM SIGCOMM Internet Measurement Conference, IMC 2006, Rio de Janeriro, Brazil, October 25-27, 2006*, pages 147–152, 2006.

[31] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 311–324, 2016.

[32] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings.*, pages 212–223, 1980.

[33] 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*, pages 101–114, 2016.

[34] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008, Annapolis, MD, USA, June 2-6, 2008*, pages 121–132, 2008.

[35] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 346–357, 2002.

[36] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, pages 398–412, 2005.

[37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 419–430, 2014.

[38] 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*, pages 14:1–14:13, 2015.

[39] 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*, pages 129–143, 2016.

[40] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 85–98, 2017.

[41] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[42] Y. Peng, J. Yang, C. Wu, C. Guo, C. Hu, and Z. Li. detector: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 55–68, 2017.

[43] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. B. Carter, and R. Fonseca. Planck: millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 407–418, 2014.

[44] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 123–137, 2015.

[45] R. T. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference, IMC 2004, Taormina, Sicily, Italy, October 25-27, 2004*, pages 207–212, 2004.

[46] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, pages 164–176, 2017.

[47] H. Song, S. Dharmapurikar, J. S. Turner, and J. W. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, pages 181–192, 2005.

[48] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with pathdump. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 233–248, 2016.

[49] P. Tammana, R. Agarwal, and M. Lee. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 453–466, 2018.

[50] 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*, pages 561–575, 2018.

[51] M. Yoon, T. Li, S. Chen, and J. Peir. Fit a compact spread estimator in small high-speed memory. *IEEE/ACM Trans. Netw.*, 19(5):1253–1264, 2011.

[52] 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*, pages 29–42, 2013.

[53] 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*, pages 97–108, 2013.

[54] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 479–491, 2015.

A SUPPLEMENTARY OF SECTION II

A.1 Proof of Lemma 1

Lemma: Let m denote the number of buckets, N the number of unique keys. For a sketch with c banks of bucket arrays, where each bucket array is of size $\frac{m}{c}$, the expected percent of noisy buckets is $1 - e^{-cN/m} - \frac{cN}{m} \cdot e^{-c(N-1)/m}$.

PROOF. For a sketch with one bucket array that consists of m buckets, the expected number of keys per bucket amounts

to $\frac{N}{m}$. The expected number of empty buckets is: $\sum_i \left(1-\frac{1}{m}\right)^N = m\left(1-\frac{1}{m}\right)^N \approx me^{-N/m}$. Similarly, the expected number of buckets with one key amounts to: $\sum_i \binom{N}{1}\left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{N-1} \approx Ne^{-(N-1)/m}$. As a result, the expected percent of buckets that contain at least two keys is $\left(m - me^{-N/m} - Ne^{-(N-1)/m}\right)\cdot\frac{1}{m}$. The expected percent of noisy buckets is $1 - e^{-N/m} - \frac{N}{m}\cdot e^{-(N-1)/m}$.

For a sketch with c banks of bucket arrays, where each bucket array is of size $\frac{m}{c}$. We can see that each bucket array still receives N keys. Thus following the same derivation, we have that the corresponding expected percent of noisy buckets is $1 - e^{-cN/m} - \frac{cN}{m}\cdot e^{-c(N-1)/m}$. □

A.2 CM and CS Performance Bounds

A count-min sketch 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.

The insertion process for Count-Sketch differs a bit, as it chooses k random sign functions r_i ($i \in \{1, 2, \cdots, k\}$) to weigh the value by a random sign from $\{+1, -1\}$, and update each selected bucket by the weighted value of the given key, i.e., value $\cdot r_i$(key). For Count-min, it directly increases the counter of the selected bucket by the value of the incoming key.

To query a given key, we use 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). For Count-Sketch, it calculates the median of weighted values stored in each bucket, i.e., median value $\left\{bucket(h_j(key))\cdot r_j(key)\right\}$. While for the Count-min, it approximates the value of a given key by the minimum of mapped buckets. Count-min and Count-sketch needs k hash-function computations and k memory operations when inserting or querying a key-value pair.

For count-min sketch (CM) [12], the probability of the minimum of the inserted buckets is greater than the ground-truth value by $\frac{2}{m}\|v(x)\|_1$ is:

$$Pr\left[min\left(I\left[i\right]\left[h_i\left(x_i\right)\right]\right) - v\left(x_i\right) \geq \frac{2}{m}\|v\left(x\right)\|_1\right] \leq \frac{1}{2^k}$$

, and the probability of the median of the inserted buckets is

$$Pr\left[\left(\hat{x}_i - x_i\right)^2 > \frac{t}{k}\cdot\frac{\|x\|_2^2}{m}\right] < 2e^{-\Omega(t)}$$

for CS [8]

B SUPPLEMENTS OF SECTION III

B.1 Proof of Theorem 2

Theorem 2: A sketch with one bucket array is equivalent to a linear autoencoder: the insertion process corresponds to an encoding phase $I = \left(A^T X\right)$, while the query phase corresponds to a decoding phase $\hat{X} = A \cdot I$.

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

For the query process 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 $I(j)$ as the approximated value for X_i. Similarly, based on the i-th row vector of A, denoted as $A(i, :)$, we can equivalently represent the approximated value as $\hat{X}_i = A(i, :) \cdot I$. Therefore, the approximated values for all inserted keys can be calculated as a decoding phase: $\hat{X} = A \cdot I$. □

B.2 K-means Model

Specifically, the K-means clustering method minimizes the variance of each cluster by finding a set of k points (called centroids) such that the potential function is minimized

$$F\left(S\right) = \sum_{x\in S}\min_{c\in C}\|x - c\|^2 \tag{5}$$

, and finally outputs a list of cluster centers that minimizes the variance of within-cluster values. We choose the Lloyd's algorithm to optimize Eq. (5), which initializes centroids arbitrarily, partitions points by the nearest centroid, and updates the centroids of each cluster until convergence. The training process of the K-means clustering method has to repetitively update the cluster centers until convergence.

B.3 Proof of Theorem 3

Theorem 3: Y_j is an unbiased estimator for any variable X_i^j. $Pr\left(\left|Y_j - \mu\right| \geq a\right) \leq \frac{M^2}{a^2 n_j{}^2}$ for a positive constant a. Moreover, $Pr\left(\left|Y_j - X_i^j\right| \geq a\right) \leq \frac{M^2}{(a-M)^2 n_j{}^2}$, for positive constants a.

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:

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

By Chebyshev's inequality, we bound the range of Y_j as : $Pr\left(\left|Y_j - \mu\right| \geq a\right) \leq \frac{Var\left[Y_j\right]}{a^2} \leq \frac{M^2}{a^2 {n_j}^2}$

Second, the following inequality holds:

$$Pr\left(\left|Y_j - X_i^j\right| \geq a\right) = Pr\left(\left|Y_j - \mu + \mu - X_i^j\right| \geq a\right)$$
$$\leq Pr\left(\left(\left|Y_j - \mu\right| + \left|X_i^j - \mu\right|\right) \geq a\right)$$
$$= Pr\left(\left(\left|Y_j - \mu\right|\right) \geq a - \left|X_i^j - \mu\right|\right)$$
$$\leq Pr\left(\left|Y_j - \mu\right| \geq a - M\right)$$
$$\leq \frac{M^2}{(a-M)^2 {n_j}^2}$$

The second inequality holds due to the triangle inequality condition ($\left|Y_j - \mu + \mu - X_i^j\right| \leq \left|Y_j - \mu\right| + \left|X_i^j - \mu\right|$). □