

Couper: Memory-Efficient Cardinality Estimation under Unbalanced Distribution

Xun Song¹, Jiaqi Zheng¹, Hao Qian¹, Shiju Zhao¹, Hongxuan Zhang¹, Xuntao Pan¹, and Guihai Chen¹

State Key Laboratory for Novel Software Technology, Nanjing University

{xunsong, hqian, shijuzhao, x_zhang, xuntaopan}@mail.nju.edu.cn {jzheng, gchen}@nju.edu.cn

Abstract—Estimating per-flow cardinality from high-speed data streams has many applications such as anomaly detection and resource allocation. Yet despite tracking single flow cardinality with approximation algorithms offered, there remain algorithmical challenges for monitoring multi-flows especially under unbalanced cardinality distribution: existing methods adopt a uniform sketch layout and incur a large memory footprint to achieve high accuracy. Furthermore, they are hard to implement in the compact hardware used for line-rate processing.

In this paper, we propose Couper, a memory-efficient measurement framework that can estimate cardinality for multi-flows under unbalanced cardinality distribution. We propose a two-layer structure based on a classic coupon collector’s principle, where numerous mice flows are confined to the first layer and only the potential elephant flows are allowed to enter the second layer. Our two-layer structure can better fit the unbalanced cardinality distribution in practice and achieve much higher memory efficiency. We implement Couper in both software and hardware. Extensive evaluation under real-world and synthetic data traces show more than 20× improvements in terms of memory-efficiency compared to state-of-the-art.

Index Terms—cardinality estimation, sketch, data streams

I. INTRODUCTION

Sketch algorithms measure fine-grained statistics from high-speed data streams by a single-pass processing. A data stream can be viewed as a union of multiple disjoint flows, of which the statistics of interest can be generally classified into two categories. One category is based on flow size estimation [7], [32], [46], [45], where each item in the stream only contains a flow ID f , and all items within a flow contributes equally to the flow size. In the other category, an additional element ID e is carried by each item to distinguish those belonging to the same flow. Each distinct element in a flow is counted only once regardless of the number of occurrences, referred to as *cardinality* estimation [2], [12], [40], [9], [37].

Many practical applications [28], [22], [15], [16] can be reduced to the cardinality estimation model, where the definition of f and e depend on the specific application scenarios. In production data centers, to achieve load balancing, the operator needs to timely obtain the number of parallel TCP connections maintained by each server. Here we can treat all packets sent to the same server as a flow and measure the number of distinct source IP addresses for each flow as its cardinality. Also, by tracking the number of distinct users accessing each file, an in-network K-V store system [17] can acquire the popularity of each content and adjust the cache priority accordingly

to improve performance. In addition, Google’s data analysis systems such as Sawzall [28], Dremel [22] and PowerDrill [15] periodically estimate the number of distinct users that search the same key, where we can model all search requests containing the same key word as a flow and user identifiers (e.g., their IP addresses) as the elements in each flow. Under the security consideration [11], [23], [27], [29], [36], the flows with extremely large cardinality, referred to as *super spreaders*, should be detected and reported timely, since a super spreader may signal a DDoS attack or malicious port scanning [31].

Compared to flow size estimation, the central challenge of cardinality estimation is the need for duplicate removal. Computing exact cardinality by recording all items is impractical due to the unaffordable storage burden. Therefore, a plethora of probabilistic data structures have been proposed to estimate the cardinality approximately, e.g., BJKST [2], HyperLogLog [12], [16], MRB [9], SMB [37], etc., which we refer to as cardinality estimators.

Nowadays, measuring the cardinality for single flow has been a well-studied problem. Using any of the aforementioned algorithms and only 1KB memory, one can accurately estimate the cardinality of a flow that contains millions of distinct items. Nevertheless, the challenges of tracking cardinality for multi-flows still remain. Existing methods follow the idea of CM-Sketch [7]: they uniformly divide the memory space into a number of cells shared by all flows, and use cardinality estimators as plug-ins. This uniform sketch layout suffers low memory efficiency in practice. Specifically, the real data stream often obeys highly-skewed distribution, i.e., the majority of the flows are *mice flows* (i.e., with small cardinality), while the minority of flows are *elephant flows* (i.e., with large cardinality). Under a uniform layout, all cells share the same size and must be capable to accommodate the largest flow. As a result, lots of memory are wasted by those cells that only record mice flows.

Existing methods do not scale well since their memory cost increases dramatically as the data stream contains more flows: The current big data stream may see hundreds of thousands of flows in a 5-minute epoch, resulting in considerable cost for maintaining their data structures in DRAM and persistent storage. In addition, to achieve line-rate processing, these measurement functions are increasingly implemented on the hardware devices including programmable switches [3], [42], Smart NICs [1] and FPGA [38], [42]. Their fast but small on-

chip memory (e.g., SRAM) enforces restrictions on the size of data structures. Therefore, we need to design the cardinality measurement framework with high memory-efficiency.

Prior art propose a series of techniques to improve the memory utilization, but they can only alleviate the problem to some degree, as none of them are directly optimized for better conquering the unbalanced data distribution to the best of our knowledge. For example, VHS [41] and CSE [44] propose to share memory among flows in a finer granularity, i.e., the minimal memory sharing unit is a bit or a register, rather than a complete estimator. This may reduce memory wastage but also increase the hash collision probability, degrading the accuracy a lot once the averaged per-flow memory size is relatively low.

In this paper, we propose Couper to perform per-flow cardinality estimation and super spreader detection in real time. Essentially, to tackle unbalanced distribution, Couper allocates a little memory for mice flows, while reserves more memory for elephant flows. Couper implements this adaptive allocation by designing a *two-layer* sketch layout. Mice flows are confined to the first layer by a classic coupon collector's principle, which only consumes a dozen of bits for each flow and thus saves memory significantly. And we further optimize the accuracy by (1) mitigating *overlapping bias* — a systematic error introduced by the hierarchical design, and (2) mitigating the hash collision error with a fine-grained cells-combining strategy.

Our contributions:

- We propose a memory-efficient framework Couper to measure cardinality for multi-flows.
- We extend Couper to detect super spreaders and generalize it to a meta-framework, which can be used to optimize existing super spreader detection algorithms.
- We conduct mathematical analyses for Couper.
- We evaluate its performance in comparison with prior art using real-world and synthetic data traces. The results show the improved memory efficiency (e.g., $16\times \sim 20\times$ lower memory cost to achieve the same accuracy).
- We implement Couper in both the software and hardware (the emerging programmable switch) and the source code is available on Github [8].

II. PRELIMINARIES


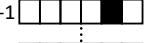

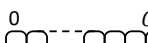
A. Problem Formulation

Let $S = \{p_1, p_2, p_3 \dots p_N\}$ be a data stream consisting of N items, where each item contains a pair of identifiers (flow ID f , element ID e). Those items with the same flow ID form a sub-stream, called *flow*. So S can be treated as a union of multiple disjoint flows, i.e., $S = \{f_1, f_2 \dots f_n\}$. Our goal is to estimate the cardinality (i.e., the number of distinct elements) for each flow.

B. Single Flow Cardinality Estimators

There are many existing solutions toward single flow cardinality estimation. We briefly review three representative estimators — LC, MRB and HLL++ — since they are typical

TABLE I: Single Flow Cardinality Estimator.

	LC	MRB	HLL(++)
Data Structure	0  $b-1$ Bitmap $B[b]$ ($\blacksquare = 1$ $\square = 0$)	$R-1$  1  0 $C-1$ 2-dim Bitmap $M[R][C]$	0  $C-1$ Counter array $H[C]$
Recording Procedure	Record (B, e): $i = \phi_1(e) \% b$ $B[i] = 1$	Record (M, e): $c = \phi_1(e) \% C$ $r = \min\{\#lz(\phi_2(e)), R-1\}$ $M[r][c] = 1$	Record (H, e): $c = \phi_1(e) \% C$ $v = \#lz(\phi_2(e)) + 1$ $H[c] = \max\{v, H[c]\}$
Query Procedure	QueryLC (B): $z=0$ for i in $[0, b)$: if $B[i]=0$: $z+=1$ return $b \times \ln \frac{b}{z}$	QueryMRB (M): $\text{base} = \min\{r \mid lf(M[r]) < \theta\}$ $\text{res} = 0$ for r in $[\text{base}, R)$: $\text{res} += \text{QueryLC}(M[r])$ return $2^{\text{base}} \times \text{res}$ θ is a pre-defined threshold.	QueryHLL (H): $\text{res} = \alpha \times C^2 \left(\sum_{x=0}^{C-1} 2^{-H[x]} \right)^{-1}$ if $\text{res} \leq 2.5 \times C$: $V \leftarrow \text{fraction of } 0$; $\text{res} = C \times \ln \frac{1}{V}$; return res

Note: $\phi_1(\cdot)$: output a positive integer. $\phi_2(\cdot)$: output a bit string.
 $\#lz(\cdot)$: the number of leading zeros. $lf(\cdot)$: the fraction of non-zero bits.

components of multi-flow measurement frameworks. Their recording and query procedures are given in Table I.

Linear Counting (LC) [40] estimator maintains a bitmap B of size b (initialized to 0 at the beginning). Each incoming item randomly selects an index i by a uniform hash function $\phi(\cdot)$, and set $B[i]$ to 1. Upon receiving a query, LC counts the number of “0” entries in B , denoted by z , then outputs the maximum likelihood estimation to the real cardinality s as:

$$\hat{s} = b \ln \frac{b}{z}$$

Multi-Resolution-Bitmap (MRB)[9] employs R C -bit-wide bitmaps and arrange them into a hierarchical structure. Each incoming item can only apply update to one level of the hierarchy, where each subsequent level has exponentially lower probability to receive the update. Specifically, a random hash function (ϕ_2 in Table I) is applied to the incoming item e and outputs a bit-string, where each bit has a probability of $\frac{1}{2}$ to be 0 or 1, then we count the number of leading zeros in it (denoted by r) and record e in layer r , so that the i -th layer will have a sampling probability of $\frac{1}{2^{i+1}}$ ($0 \leq i < R$). In query stage, MRB treats each layer as a distinct bitmap and applies LC to each of them. As the accuracy of bitmap (LC) decreases as its load factor (the fraction of “1” bits) increases, MRB only uses bitmaps with load factor lower than a threshold θ ($=0.93$).

HyperLogLog++ (HLL) [16] estimator H consists of C registers. For each incoming item e , HLL first performs $c = \phi_1(e) \% C$ to select a register $H[c]$, where $\phi_1(\cdot)$ is a uniform hash function. Then HLL generates a bit sequence $\omega := \langle x_{31}, x_{30}, \dots, x_0 \rangle$ by performing another hashing $\phi_2(\cdot)$ on e .¹ Finally, HLL updates $H[c]$ to $\max\{H[c], \#lz(\omega) + 1\}$, where $\#lz(\omega)$ denotes the number of leading zeros of ω . Upon

¹In fact, HLL only performs one hashing, and the result is split to two disjoint sub-strings. We logically treat them as the results from two independent hash functions $\phi_1(\cdot), \phi_2(\cdot)$.

receiving a query, HLL first estimates the cardinality as:

$$res = \alpha \cdot C^2 \left(\sum_{j=0}^{C-1} 2^{-H[j]} \right)^{-1}$$

where α is a correction factor. If res is found to be less than $\frac{5}{2}C$, HLL will treat the register array as a bitmap and use LC to estimate the final result.

C. The Case for Monitoring Multi-Flows (Related Work)

A straightforward solution to monitor multi-flows is to assign a separate cardinality estimator (e.g. HLL) to each flow through a hash table, which can be too costly when the number of flows is huge. What's more, resolving hash collisions leads to an unbounded packet processing delay and makes it a poor choice for online processing. Current solutions to this issue share memory among flows to reduce the memory cost. They can be categorized into sketch-based solutions and sample-based solutions.

Sketch-based methods can be regarded as variants of CM-Sketch. They arrange the memory space into an array of cardinality estimators. During processing, each flow is associated with k cells (using random hash functions) to record its elements. Note that any cell could be shared by multiple flows due to hash collisions, so the estimation given by a cell is the combined cardinality of all flows mapping to it, which has a positively biased error. Prior work adopt different strategies to alleviate the error. gSkt [47] makes k estimations independently and returns the minimum; VHS [41] and CSE [44] share memory among flows in a finer granularity by a virtual estimator design; rerSkt [38] has analogical idea with Count Sketch [5], it splits the background traffic between a primary estimator and a complement estimator. By subtracting the complement from the primary estimator, it statistically removes the error. This strategy is based on the assumption that the background traffic can be split evenly for each flow — though working well for large flows, it poses large variance to small flows.

Sample-based method [34] selects distinct elements using *Non-Duplicate Sampling* (NDS). NDS maintains a large bitmap to handle duplicates for all flows, each packet that maps to an empty bit in the bitmap has a probability p to be sampled, and the bit will be set to “1” whether or not the packet is sampled. Each sampled packet will be sent to a hash table which tracks sampled packets for each flow. However, the hash table can be huge and makes it impossible to fully implement NDS in fast but compact on-chip memory. To perform online processing, [34] proposes to offload the hash table to off-chip memory. Then the overall sample rate is limited by the throughput ratio between on-chip and off-chip, which could be rather low. And it needs to occupy considerable bandwidth between on/off chip, which should be reserved for more important applications. As it's costly and complicated to implement sample-based methods in practice, we focus on sketch-based methods in this work.

III. COUPER

A. Motivation and Overview

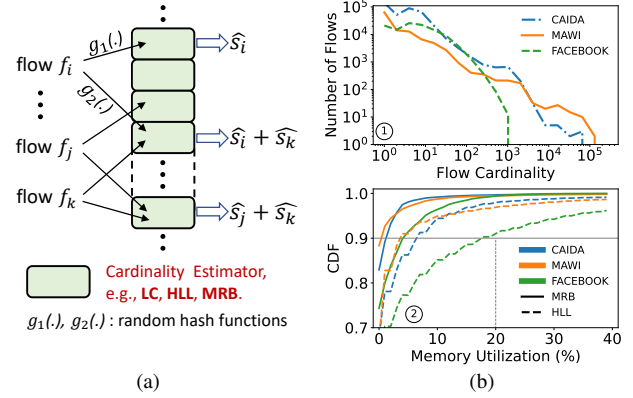


Fig. 1: (a) The data structure adopted by prior art (b) Highly skewed cardinality distribution in real data sets (①), where the current data structure suffers low memory efficiency (②).

Current frameworks and their drawbacks. Existing methods toward multi-flow cardinality measurement borrow the idea of memory-sharing from Count-Min Sketch. They arrange a block of memory into an array of cardinality estimators, as shown in Figure 1(a). Each flow randomly selects k (two in our figure) estimators to record its elements ($\langle f, e \rangle$ pairs). Due to hash collisions, an estimator could output the combined cardinality of multiple flows. On the one hand, plenty of estimators are required to avoid severe hash collisions among flows; On the other hand, most of the memory remains unused in estimators that only record mice flows, which take the majority in practical data distribution. As a result, a considerable amount of estimators will suffer low utilization.

To concretize our concern, we demonstrate the distribution of three real-world data traces, MAWI, CAIDA, FACEBOOK, in Figure 1(b). Note that in all the 3 data traces, the flows with cardinality of greater than 15 account for less than 20%, while the largest cardinality could achieve 10^5 . Furthermore, we experimentally investigate the memory utilization when processing those data sets using the sketch in Figure 1(a), where we try both MRB and HLL as the cardinality estimator and guarantee a hash collision rate of at most 10%. Figure 1(b) shows that under all (*estimator, data set*) combinations, more than 90% estimators have a utilization (i.e., the fraction of modified bits or counters) lower than 20%. It's predictable that such under-utilization can be more severe if we use LC [40], MinCount [2], or other less memory-efficient estimators.

Coupon Collector's Principle and How we use it. We first review the coupon collector's problem. A player hopes to collect coupons from a brand of gift boxes. There are b different coupons in total and each box randomly contains one with a positive integer between 1 and b , which is unseen until the box is opened. To collect t distinct coupons, the expected number of boxes needed is $\frac{b}{b} + \frac{b}{b-1} + \dots + \frac{b}{b+1-t} = \sum_{i=1}^t \frac{b}{b+1-i}$. The coupon collector's principle can help us to separate the numerous mice flows from the data stream and handle them in a memory-efficient manner.

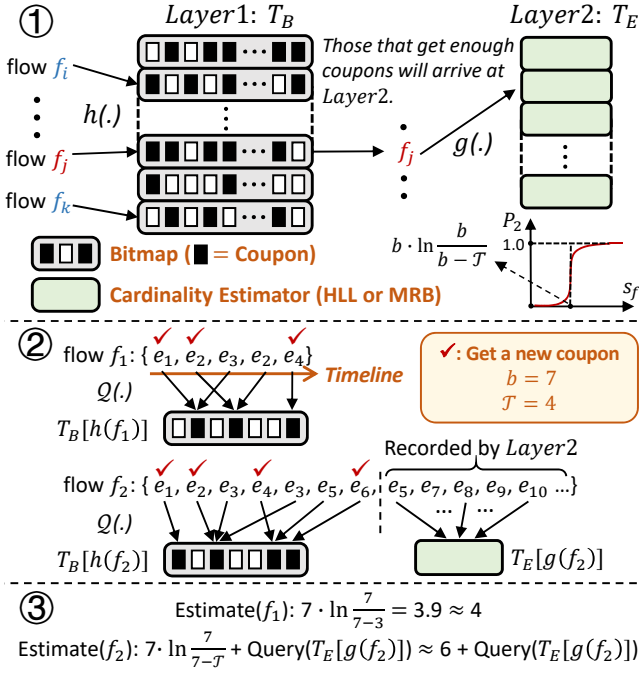


Fig. 2: The basic architecture of Couper. P_2 (y-axis) in ① represents the probability that a flow f arrives at Layer2 as a function of the flow cardinality s_f (x-axis). ③ demonstrates an example of estimating cardinality with Couper.

Specifically, when a data item $\langle f, e \rangle$ occurs, we assign a coupon numbered $Q(f \oplus e)$ to flow f , where $Q(\cdot) \in [1, b]$ is a random hash function and \oplus means a splicing operation. Since the results of $Q(\cdot)$ are assumed to be distributed uniformly, the b coupons will be collected with equal probability. Hence, the more distinct elements a flow f contains, the more distinct coupons it would obtain. If we specify a threshold T for winning the collection, flows with cardinality greater and smaller than $\sum_{i=1}^T \frac{b}{b+1-i}$ can be roughly separated.

In practice, the memberships of the b coupons can be represented as a bitmap of length b , where the i -th bit with value “1” (“0”) indicates that the coupon numbered i has (not) been collected. Therefore, we can assign a bitmap for each flow to track the coupons it collected. Upon the number of “1” bits exceed the threshold T for a flow f , we conclude that f is not a mice flow and record its (remaining) elements in a sketch (e.g., Figure 1(a)). The key advantage is that even a small bitmap with a dozen of bits is enough to confine a mice flow, we can thus reduce the memory wastage caused by numerous mice flows. What’s more, those mice flows are prevented from entering the sketch, of which the hash collision rate is mitigated significantly.

B. Basic Architecture

As shown in Figure 2, Couper adopts a 2-layer architecture. The first layer T_B is composed of L_1 bitmaps, each consuming b bits, and the second layer T_E is composed of L_2 cardinality estimators, each consuming E bits. We denote the fraction of memory taken by Layer1 as μ . There are two optional estimators for T_E — HLL++ and MRB — since they are

Algorithm 1 Insertion

```

1: Input: An item  $\langle f, e \rangle$ 
2:  $bm := T_B[h(f)]$ ;
3:  $coupon = 0$ ;
4: for  $i$  in  $[0, b]$  do
5:   if  $bm[i] == 1$  then // coupon
6:      $coupon += 1$ ;
7: if  $coupon < T$  then
8:    $bm[Q(f \oplus e)] = 1$ ; //assign a coupon Q
9: else
10:   $\text{Record}(T_E[g(f)], f \oplus e)$ ; //See Table I.
```

memory efficient to record large flows (For convenience, we abbreviate HLL++ to HLL afterwards). And E depends on the required standard error for each estimator. All bits and counters are initialized to 0 at the beginning. For convenience, key notations are summarized in Table II.

1) *Insertion (Algorithm 1)*: Upon receiving an item $p = \langle f, e \rangle$, we first turn to T_B and locate a bitmap for flow f by a uniform hash function $h(\cdot)$. The selected bitmap, which is $T_B[h(f)]$, keeps track of the coupons collected by flow f . The number of “1” bits in $T_B[h(f)]$ — denoted by C_f — determines the following actions:

Case 1: $C_f < T$. This means that the flow f has not collected enough coupons yet. We thus assign a coupon numbered $Q(f \oplus e)$ to the flow f by setting the corresponding bit in $T_B[h(f)]$ to “1”.

Case 2: $C_f \geq T$. This means that the flow f has collected enough coupons, or unfortunately conflicts with other flows that also map to $T_B[h(f)]$ and they together accomplish the collection. At this moment, we turn to Layer2 and keep $T_B[h(f)]$ unchanged. At Layer2, we record $f \oplus e$ with estimator $T_E[g(f)]$, where $g(\cdot) \in [0, L_2]$ is a random hash function.

We showcase an example in Figure 2 (②). A mice flow f_1 with 4 distinct elements is confined in Layer1 as it only gets 3 distinct coupons, which is less than $T = 4$. And another flow f_2 collects 4 out of the 7 coupons at the moment when the item $\langle f_2, e_6 \rangle$ occurs, then the remaining elements of f_2 arrive at Layer2.

2) *Estimate (Algorithm 2)*: When estimating the cardinality of some flow f , we first turn to Layer1 and check the number of coupons f has collected (i.e., C_f) by traversing the bitmap $T_B[h(f)]$. Similar to the *Insertion* stage, the estimate scheme

TABLE II: Notations

b/E	# bits per bitmap in T_B /# bits per estimator in T_E
μ	the fraction of memory taken by Layer1
T	the threshold for winning the collection
k_1/k_2	# hashed cells per flow in Layer1/Layer2 (§III-D).
L_1/L_2	# bitmaps in T_B /# estimators in T_E
C	# columns per MRB and # counters per HLL
R	# rows per MRB
$\phi_1(\cdot), \phi_2(\cdot)$	hash functions used by HLL and MRB (Table I)
$h_i(\cdot)/g_i(\cdot)$	uniform hash functions for T_B/T_E
$B^*/H^*/M^*$	the resulting bitmap/ HLL/ MRB of FGM
s_f/s_f	estimated/true cardinality of flow f

Algorithm 2 Estimate s_f

```

1: Input: A flow ID  $f$ 
2:  $bm := T_B[h(f)];$ 
3:  $coupon = 0;$ 
4: for  $i$  in  $[0, b)$  do
5:   if  $bm[i] == 1$  then // coupon
6:      $coupon += 1;$ 
7: if  $coupon < \mathcal{T}$  then
8:   return  $b \ln \frac{b}{b-coupon};$  // MLE
9: else
10:  return  $\text{Query}(T_E[g(f)]) + b \ln \frac{b}{b-\mathcal{T}};$  //see Table I.

```

also depends on C_f :

Case 1: $C_f < \mathcal{T}$. This means the flow f didn't arrive at Layer2 and all of its elements are devoted to coupon collection. We can thus estimate the cardinality of the flow f with the number of coupons it collected. According to Linear Counting Algorithm [40], the Maximum Likelihood Estimation (MLE) of the cardinality of f should be $b \ln \frac{b}{b-C_f}$.

Case 2: $C_f \geq \mathcal{T}$. This means the flow f arrived at Layer2 sometime in the past and some of its elements are recorded by $T_E[g(f)]$. We can use $T_B[h(f)]$ and $T_E[g(f)]$ to estimate the cardinality of the elements recorded by Layer1 and Layer2, respectively. Finally, to estimate s_f , a naive solution is to sum up the two estimations. This may cause an *overlapping error* — though negligible in most cases — we will discuss how to mitigate it in the next Section.

C. Mitigate Overlapping Error

In this section, we discuss how the naive estimate scheme could cause an *overlapping error*, then elaborate how we refine the vanilla design to mitigate the error.

The design of a 2-layer structure can improve memory utilization, but may cause some elements to be counted twice, which contradicts the definition of cardinality. An example in Figure 3 concretizes this issue: flow f records the first six elements (denoted as S_1) in $T_B[h(f)]$ and collects 4 coupons, after which the remaining elements (S_2) are recorded in $T_E[g(f)]$. Algorithm 2 actually gives an estimate of $|S_1| + |S_2|$, while the real cardinality is $|S_1 \cup S_2|$, where $|S|$ represents the cardinality of S . Hence, our estimate has a bias $\Omega = |S_1| + |S_2| - |S_1 \cup S_2| = |S_1 \cap S_2|$.

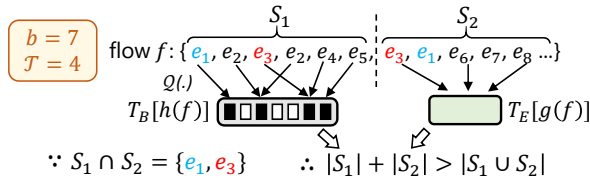


Fig. 3: The overlapping issue

One way to mitigate the error is to estimate Ω and subtract it from the initial estimation. So we first depict a simpler case where $|S_1 \cap S_2|$ can be estimated easily: As shown in Figure 4, S_1 and S_2 are recorded by two bitmaps bm_1, bm_2 (they

Algorithm 3 Estimate Ω

```

1: Input:  $T_B[h(f)], T_E[g(f)]$ 
2:  $bm_1 := T_B[h(f)];$ 
3:  $bm_2 \leftarrow$  an empty bitmap of length  $b$ 
4: if using Multi-Resolution-Bitmap then
5:    $M := T_E[g(f)];$ 
6:    $B_{or} \leftarrow$  an empty bitmap of length  $C$ 
7:   for  $r$  in  $[0, R)$  do
8:      $B_{or} = B_{or} \mid M[r];$  // “|” means bit-wise-or
9:   for  $c$  in  $[0, C)$  do
10:    if  $B_{or}[c] == 1$  then
11:       $bm_2[c \% b] = 1;$ 
12: if using HyperLogLog then
13:    $H := T_E[g(f)];$ 
14:   for  $c$  in  $[0, C)$  do
15:    if  $H[c] > 0$  then
16:       $bm_2[c \% b] = 1;$ 
17:  $B_{\cap} = bm_1 \& bm_2;$  // “&” means bit-wise-and
18:  $t_{mp} = 0;$ 
19: for  $i$  in  $[0, b)$  do
20:   if  $B_{\cap}[i] == 1$  then
21:      $t_{mp} += 1;$ 
return  $\Omega = \min\{b \ln \frac{b}{b-t_{mp}}, b \ln \frac{b}{b-\mathcal{T}}\};$ 

```

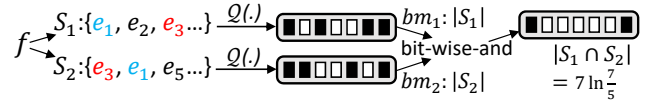


Fig. 4: A simpler case for resolving Ω

have two identical elements e_1, e_3). Since bm_1, bm_2 have the same length 7 and share the same hash function $Q(\cdot)$, we can *bit-wise-and* the two bitmaps and the resulting bitmap can be regarded as a creation of their intersection $S_1 \cap S_2$, which can be used to estimate $|S_1 \cap S_2|$.

In our *two-layer* design, bm_1 is exactly $T_B[h(f)]$, but bm_2 is unavailable. As a naive solution, we can additionally attach a bitmap to each estimator in T_E — when an item $\langle f, e \rangle$ is recorded by $T_E[g(f)]$, the affiliated bitmap will also record $\langle f, e \rangle$ with hash function $Q(\cdot)$. This method is easy to implement but results in additional memory usage. We instead propose to construct bm_2 with the data structure of $T_E[g(f)]$ only when performing a query.

However, if the cells in two layers (i.e., bitmap $T_B[h(f)]$ and estimator $T_E[g(f)]$) perform recording independently, there is no way to recover Ω . So the first step is to correlate $Q(\cdot)$ with the hash function used in the estimators of Layer2. In the vanilla design, $Q(\cdot)$ can be any random hash function, but now we re-define $Q(\cdot)$ as follows:

$$\tilde{Q}(e) := (\phi_1(e) \% C) \% b$$

Recall that MRB (resp. HLL) uses $\phi_1(e) \% C$ to determine which column (resp. counter) to apply recording (see Table I). Then the approach to estimate Ω is defined in Algorithm 3, and we show an example in Figure 5, where Layer2 uses MRB. The rationale is that, for MRB (resp. HLL), when an element e

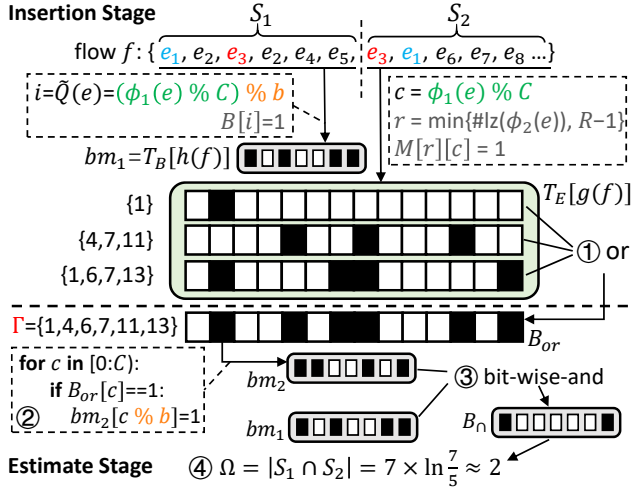


Fig. 5: An example of estimating Ω

is recorded in the k -th ($k = \phi_1(e) \% C$) column (resp. counter), we can deduce that if e is recorded by a bitmap using hash function $\tilde{Q}(\cdot)$, the $(k \% b)$ -th bit will be set to “1”. Therefore, we can find all non-empty columns in MRB (resp. non-zero counters in HLL), denoted as Γ , then for each $\gamma \in \Gamma$, we set $bm_2[\gamma \% b]$ to “1”.

Finally, we can replace #Line 10 of Algorithm 2 with:

```

{ Estimate  $\Omega$ ;
  return Query( $T_E[g(f)]$ ) +  $b \ln \frac{b}{b-\tau} - \Omega$ ;

```

Discussion. A) *Uniformity of $\tilde{Q}(\cdot)$.* To keep the randomness of the results given by $\tilde{Q}(\cdot)$, one should guarantee $C \gg b$ or $C \% b = 0$, which can be achieved easily for both HLL and MRB. B) *Accuracy.* One may observe that bm_2 tends to be FULL (i.e., all bits are set to “1”) as the flow cardinality increases, which means $\Omega \equiv b \ln \frac{b}{b-\tau}$ for those large flows. This may cause an underestimation bounded by $b \ln \frac{b}{b-\tau}$, which is negligible for large flows and seldom degrades their ratio error.

D. Mitigate Hash Collision Error

In the vanilla design described in §III-B, each flow maps to a single cell in each layer. This may cause A) a mice flow be misreported to Layer2 once an elephant flow maps to the same bitmap with it and, B) a flow in Layer2 suffers severe overestimation when conflicting with a particularly large flow. We mitigate this issue by allocating multiple cells (bitmaps in Layer1 and estimators in Layer2) for each flow, and merging them in a fine-grained manner.

Recording with multiple cells. Each flow will map to k_1 bitmaps with hash functions $h_1(\cdot), h_2(\cdot), \dots, h_{k_1}(\cdot)$, all of which will record the coupons it has collected. When checking the number of coupons collected by a flow f , we merge all the associated bitmaps (i.e., $T_B[h_1(f)], T_B[h_2(f)] \dots T_B[h_{k_1}(f)]$), then count “1” bits in the resulting bitmap. Similarly, a flow arriving at Layer2 will map to k_2 estimators using hash functions $g_1(\cdot), g_2(\cdot), \dots, g_{k_2}(\cdot)$, and records its elements in each of them. When querying for its cardinality, we generate a

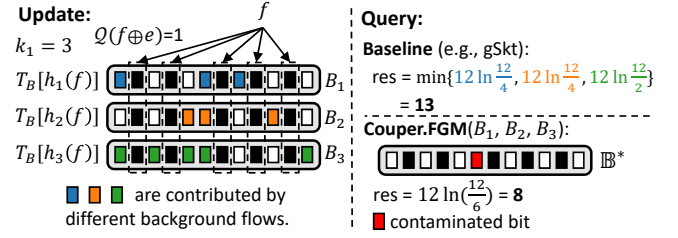


Fig. 6: The benefits of Fine-Grained-Merging

logical estimator by merging the k_2 estimators (i.e., $T_E[g_1(f)], T_E[g_2(f)] \dots T_E[g_{k_2}(f)]$).

Merging cells. Actually, gSkt [47] also records each flow with multiple independent cells. However, gSkt directly picks the minimum estimation from the k associated cells as the final result, and this coarse-grained cell combination fails to fully utilize the data structure of estimators to mitigate the hash collision error. On one hand, since the same type of cells share the same hash function (e.g., $Q(\cdot)$ for bitmaps, $\phi_1(\cdot), \phi_2(\cdot)$ for MRB), an update for the flow f will be synchronously recorded by all the associated cells; On the other hand, the noises recorded by different cells are random, indicating that the contaminated parts vary among cells (e.g., the i -th bit in $T_B[h_1(f)]$ may be set to “1” due to hash collision, while that in $T_B[h_2(f)]$ may still be “0”). Therefore, we can compare all the associated cells, pinpoint where they are different and correct the error by eliminating the differences. Our Fine-Grained-Merging (FGM for short) scheme is shown in Algorithm 4.

Figure 6 shows an example of why FGM outperforms the baseline strategy (gSkt). In query stage, gSkt picks the minimum estimation from B_1, B_2, B_3 and obtains $12 \ln \frac{12}{4} = 13$, while Couper applies FGM to B_1, B_2, B_3 and the resulting bitmap only contains one contaminated bit. Compared to gSkt, Couper reduces the overestimation by 5 (13-8).

Since the output of FGM is a single cell, FGM can be applied to our previous design directly. Specifically, we can make the following substitutions in Algorithm2 and Algorithm3:

```

T_B[h(f)] ← FGM(T_B[h1(f)] ... T_B[hk1(f)])
T_E[g(f)] ← FGM(T_E[g1(f)] ... T_E[gk2(f)])

```

Algorithm 4 Fine-Grained-Merging (FGM)

```

1: function MergeBitmap( $B_1, B_2 \dots B_{k_1}$ ) //Layer1
2:   Bitmap  $\mathbb{B}^*$ (empty);
3:   for  $i$  in  $[0, b)$  do
4:      $\mathbb{B}^*[i] = B_1[i] \& B_2[i] \dots \& B_{k_1}[i]$ ;
5:   return  $\mathbb{B}^*$ ;
6: function MergeHLL( $H_1, H_2 \dots H_{k_2}$ ) //Layer2
7:   HyperLogLog  $\mathbb{H}^*$ (empty);
8:   for  $i$  in  $[0, C)$  do
9:      $\mathbb{H}^*[i] = \min\{H_1[i], H_2[i] \dots H_{k_2}[i]\}$ ;
10:  return  $\mathbb{H}^*$ ;
11: function MergeMRB( $M_1, M_2 \dots M_{k_2}$ ) //Layer2
12:  MultiResolutionBitmap  $\mathbb{M}^*$ (empty);
13:  for  $r$  in  $[0, R)$  do
14:    for  $c$  in  $[0, C)$  do
15:       $\mathbb{M}^*[r][c] = M_1[r][c] \& M_2[r][c] \dots \& M_{k_2}[r][c]$ ;
16:  return  $\mathbb{M}^*$ ;

```

E. Application Study: Super Spreader Detection

1) *Detect Super Spreaders with Couper*: Couper can detect super spreaders in parallel with cardinality measurement. Formally, super spreaders are those flows with cardinality larger than a pre-specified threshold θ_s .

Recording super spreaders with a compact memory space requires to frequently compare cardinality among flows. We should perform the comparison efficiently to keep a high throughput. First, our two-layer structure naturally confines mice flows to Layer1, so we only consider flows that arrive at Layer2 as super spreader candidates. Second, to avoid computing cardinality each time a comparison is needed, we maintain a *digest* Λ for each estimator. Specifically, when HLL is used, the digest is the sum of all registers, i.e., $\Lambda = \sum_{i=0}^{C-1} H[i]$, and when MRB is used, the digest is the maximum level observed, i.e., $\Lambda = \max\{r | M[r] \text{ is not empty}\}$. Since Λ is positively related to the real cardinality, we can use Λ as an alternative when a comparison is needed.

The key advantage of this compromise is that to calculate a digest is much easier and can be amortized over every updates to the estimator. Take HLL as an example, each time an HLL records a new element and updates one of its registers, its Λ is also updated (e.g., if a register in HLL changes from 2 to 5, we increase Λ by $5 - 2 = 3$). The digest only consumes a dozen of bits for each estimator, but enables Couper to detect super spreaders without sacrificing the overall throughput.

We use a bucket table T_{SC} of length Z to record super spreader candidates, where each bucket contains a flow ID along with a digest. When a flow f updates at least one digest associated with its estimators, it will be reported to T_{SC} along with its minimum digest. We adopt *power-of-two-choices* [26], [6] with two hash functions $y_1(\cdot), y_2(\cdot) \in [0, Z)$ when trying to insert a (f, Λ) pair into T_{SC} , so as to mitigate hash collisions. Algorithm 5 demonstrates the insertion procedure of T_{SC} .

At the end of an epoch, for each flow recorded by T_{SC} , we estimate its cardinality (Algorithm 2). For those with cardinality larger than a pre-specified threshold θ_s , we report them as super spreaders.

2) *Improve prior work with Couper*: Existing super spreader detection methods like SS, VBF and CDS need to record all the observed $\langle f, e \rangle$ pairs in their data structures, which are vulnerable when contaminated by numerous mice

flows. We can alleviate this issue by filtering out mice flows with the first layer of Couper (CP: L_1).

The role of CP: L_1 is two-fold: A) In the update stage, all the data items will first arrive at CP: L_1 and take the same actions as we described in §III-B. An item $\langle f, e \rangle$ can be recorded by the detection algorithm only when f has collected enough coupons. By doing so, most of the mice flows are confined to CP: L_1 . B) In the query stage, the detection algorithm recovers super spreaders (denoted as S_R) from its data structure. Unfortunately, for some algorithms (e.g., VBF, CDS), S_R may contain some flows that have never shown up in the data stream, so we turn to CP: L_1 again to filter out those “fake flows”.

IV. MATHEMATICAL ANALYSIS

We conduct formal analysis for Couper under $k_1 = 2$ and $k_2 = 2$, which is the default configuration in our experiments.

A. Misreport Rate

We study the probability that a mice flow is misreported to Layer2. Formally, we define flows with cardinality no larger than $\sum_{i=1}^{\mathcal{T}-1} \frac{b}{b+1-i}$ as *mice flows* since they should only collect $\mathcal{T} - 1$ coupons by design. For convenience, we refer to all the other flows as *elephant flows* in this section.

Lemma 1. *Let $F(X, q)$ be the probability that a mice flow with X distinct elements gets q ($q \leq b, q \leq X$) coupons. Then*

$$F(X, q) = \binom{b}{q} \frac{q^X}{b^X} \sum_{i=0}^q (-1)^i \binom{q}{i} \left(1 - \frac{i}{q}\right)^X$$

The proof is omitted due to space constraints.

Let \mathbb{B}, \mathbb{B}' be the two (k_1) associated bitmaps of flow f . We will consider the misreport rate under the following 3 cases respectively. \mathbb{C}_1 : neither \mathbb{B} nor \mathbb{B}' conflicts with elephant flows. \mathbb{C}_2 : only one of \mathbb{B} and \mathbb{B}' conflicts with elephant flows. \mathbb{C}_3 : both \mathbb{B} and \mathbb{B}' conflict with elephant flows.

Given n_e the number of elephant flows, the probability that a bitmap isn't hashed by any elephant flows is $(1 - \frac{1}{L_1})^{k_1 n_e} = e^{-\frac{2n_e}{L_1}}$. Then we have $\mathbb{P}(\mathbb{C}_1) = (e^{-\frac{2n_e}{L_1}})^2 = e^{-\frac{4n_e}{L_1}}$, $\mathbb{P}(\mathbb{C}_2) = 2(1 - e^{-\frac{2n_e}{L_1}})e^{-\frac{2n_e}{L_1}}$ and $\mathbb{P}(\mathbb{C}_3) = (1 - e^{-\frac{2n_e}{L_1}})^2$.

Lemma 2. *A mice flow which collects q coupons (by itself) will be misreported with a probability lower than*

$$P(q) = \sum_{i=1}^3 \mathbb{P}(mr|\mathbb{C}_i, q) \mathbb{P}(\mathbb{C}_i)$$

where

$$\begin{aligned} \mathbb{P}(mr|\mathbb{C}_1, q) &= \sum_{\beta=\mathcal{T}-q}^{b-q} \binom{b-q}{\beta} (\xi^2)^\beta (1 - \xi^2)^{b-q-\beta} \\ \mathbb{P}(mr|\mathbb{C}_2, q) &= \sum_{\beta=\mathcal{T}-q}^{b-q} \binom{b-q}{\beta} (\xi)^\beta (1 - \xi)^{b-q-\beta} \\ P(mr|\mathbb{C}_3, q) &= 1 \end{aligned}$$

Algorithm 5 T_{SC} : Insertion

```

1: Inputs: A flow ID  $f$ 
2:  $\Lambda = \min\{T_E[g_1(f)].\Lambda, T_E[g_2(f)].\Lambda \cdots T_E[g_{k_2}(f)].\Lambda\}$ ;
3:  $SC_1 := T_{SC}[y_1(f)]$   $SC_2 := T_{SC}[y_2(f)]$ ;
4: if  $SC_1.f == None$  or  $SC_1.f == f$  then
5:    $(SC_1.f, SC_1.\Lambda) = (f, \Lambda)$ ; return ;
6: if  $SC_2.f == None$  or  $SC_2.f == f$  then
7:    $(SC_2.f, SC_2.\Lambda) = (f, \Lambda)$ ; return ;
8:  $i = \arg \min_{i=1,2} \{SC_i.\Lambda\}$ 
9: if  $SC_i.\Lambda \leq \Lambda$  then
10:    $(SC_i.f, SC_i.\Lambda) = (f, \Lambda)$  //expel the minor one

```

given N_1 the combined cardinality of all mice flows, and $\xi = 1 - (1 - \frac{1}{L_1 b})^{2N_1}$.

Proof. 1) \mathbb{C}_1 Let \mathbb{Q} ($|\mathbb{Q}| = q$) denote the coupons collected by f . The bits in \mathbb{B} (\mathbb{B}') that are set to “1” could be contributed by (1) \mathbb{Q} , and (2) noises from other mice flows. We can treat noises from mice flows independently and thus each bit in \mathbb{B} (\mathbb{B}') has a probability of $\xi = 1 - (1 - \frac{1}{L_1 b})^{2N_1}$ to receive noises, where N_1 means the combined cardinality of all mice flows. After FGM, the j -th bit ($j \notin \mathbb{Q}$) in the resulting bitmap \mathbb{B}^* is contaminated iff the j -th bit of both \mathbb{B} and \mathbb{B}' received noises, of which the probability is ξ^2 . So the number of contaminated bits in \mathbb{B}^* , denoted as β , should follow $Bino(b - q, \xi^2)$. When $q < \mathcal{T}$, a misreport under \mathbb{C}_1 happens when $\beta + q \geq \mathcal{T}$, of which the probability is

$$\mathbb{P}(mr|\mathbb{C}_1, q) = \sum_{\beta=\mathcal{T}-q}^{b-q} \binom{b-q}{\beta} (\xi^2)^\beta (1 - \xi^2)^{b-q-\beta}$$

2) \mathbb{C}_2 . Without loss of generality, we assume that \mathbb{B}' conflicts with elephant flows. We pessimistically assume that $\mathbb{B}^* = \mathbb{B}$, which means each bit in \mathbb{B}^* has a probability of ξ to receive noises. So the misreport rate under $q < \mathcal{T}$ and \mathbb{C}_2 is similar to $\mathbb{P}(mr|\mathbb{C}_1, q)$:

$$\mathbb{P}(mr|\mathbb{C}_2, q) = \sum_{\beta=\mathcal{T}-q}^{b-q} \binom{b-q}{\beta} (\xi)^\beta (1 - \xi)^{b-q-\beta}$$

3) \mathbb{C}_3 . As the probability of this happening is low, we pessimistically assume that f will be misreported under \mathbb{C}_3 :

$$P(mr|\mathbb{C}_3, q) = 1$$

□

By law of total probability, we have the following theorem.

Theorem 1. A mice flow with X distinct elements will be misreported with a probability bounded by

$$P_{mr}(X) = \sum_{q=1}^{\mathcal{T}-1} P(q)F(X, q) + \sum_{q=\mathcal{T}}^b F(X, q)$$

B. Estimation Error

For flows that are confined to Layer1, the error is bounded by $b \ln \frac{b}{b-\mathcal{T}+1}$, which is small. So we only analyze elephant flows arriving at Layer2. Let E_f denote the elements of flow f that are recorded by Layer2, our estimate can be written as $\hat{s}_f = \hat{e}_f + b \ln \frac{b}{b-\mathcal{T}} - \Omega$, where $\hat{e}_f = \text{Query}(\text{FGM}(T_E[g_1(f)], T_E[g_2(f)]))$ is an estimate to $e_f = |E_f|$. We first analyze the distribution of \hat{e}_f .

Lemma 3. \hat{e}_f approximately follows a normal distribution and its expectation and variance are as follow.

$$\begin{aligned} \mathbb{E}(\hat{e}_f) &= \mu_2 = e_f + \mu_\epsilon \\ \text{Var}(\hat{e}_f) &= (\sigma_2)^2 = (e_f)^2 \sigma_E^2 + \mu_\epsilon^2 \sigma_E^2 + \sigma_\epsilon^2 \end{aligned}$$

where

$$\begin{aligned} \sigma_E^2 &= \begin{cases} \frac{1.05^2}{C}, & \text{if using HLL.} \\ \frac{0.6367}{C}, & \text{if using MRB.} \end{cases} \\ \mu_\epsilon &= \begin{cases} C(\frac{N_2}{CL_2} - \sqrt{\frac{N_2}{\pi CL_2}}), & \text{if using HLL.} \\ \sum_{i=0}^{R-1} \frac{N_2}{L_2 2^i} (1 - \lambda_i), & \text{if using MRB.} \end{cases} \\ \sigma_\epsilon^2 &= \begin{cases} \frac{N_2(\pi-1)}{L_2 \pi}, & \text{if using HLL.} \\ \sum_{i=0}^{R-1} (\frac{N_2}{L_2 2^i})^2 C(1 - \lambda_i) \lambda_i + \frac{N_2}{L_2 2^i} C^2 (1 - \lambda_i)^2, & \text{if using MRB.} \end{cases} \\ \lambda_i &= e^{-\frac{N_2 + e_f L_2}{2^i C L_2}} \end{aligned}$$

$N_2 =$ the number of distinct elements recorded by Layer2.

Proof. We will discuss the cases for HyperLogLog and MRB, respectively.

1) *HyperLogLog.* Let \mathbb{H}, \mathbb{H}' be the two associated HLLs of f . Consider an arbitrary counter $\mathbb{T} = \mathbb{H}[t]$, and its counterpart $\mathbb{T}' = \mathbb{H}'[t]$, we denote the number of distinct elements \mathbb{T}, \mathbb{T}' have received as Z, Z' , which consists of A) the elements belonging to f , which is e_f , and B) the noises contributed by the background traffic, denoted as Y, Y' . Note that the background traffic includes all the distinct elements recorded by Layer2 (denoted as N_2) excluding those belonging to f . For simplicity, we treat the background elements as independent and obtain $Y, Y' \sim Bino(k_2 N_2, \frac{1}{CL_2}) \approx N(\frac{2N_2}{CL_2}, \frac{2N_2}{CL_2}(1 - \frac{1}{L_2})) \triangleq N(\mu, \sigma^2)$.

After FGM, the minor one of \mathbb{T} and \mathbb{T}' will contribute to the resulting HLL \mathbb{H}^* . Since \mathbb{T}, \mathbb{T}' are positively related to Z, Z' , the noises recorded by \mathbb{H}^* will be $Y^* = \min\{Y, Y'\}$ (i.e., we assume the minor counter value corresponds to the minor noises). The minimum of two i.i.d. gaussian variables can also be modeled as a gaussian variable [25]. Then we obtain $Y^* \sim N(\mu - \frac{\sigma}{\sqrt{\pi}}, \frac{\pi-1}{\pi}\sigma^2)$ [25]. Let Y_i^* denote the noises recorded by counter $\mathbb{H}^*[i]$, then $Y_0^*, Y_1^*, \dots, Y_{C-1}^*$ are independent, and the total noises recorded by \mathbb{H}^* will be

$$\epsilon = \sum_{i=0}^{C-1} Y_i^* \sim N\left(C\left(\mu - \frac{\sigma}{\sqrt{\pi}}\right), C\sigma^2 \frac{\pi-1}{\pi}\right) \triangleq N(\mu_\epsilon, \sigma_\epsilon^2)$$

The total elements recorded by \mathbb{H}^* is $e_f + \epsilon$. Since HLL is asymptotically unbiased and its standard error is $\frac{1.05}{\sqrt{C}}$, we have $\hat{e}_f = (e_f + \epsilon)\kappa = e_f \kappa + \epsilon \kappa$, where $\kappa \sim N(1, \frac{1.05^2}{C})$. As $\frac{\mathbb{E}(\epsilon)}{\text{Std}(\epsilon)} = O(\frac{\sqrt{C}\mu}{\sigma}) = O(\sqrt{C}) \gg 1$ and $\frac{\mathbb{E}(\kappa)}{\text{Std}(\kappa)} = O(\sqrt{C}) \gg 1$, $\epsilon \kappa$ can be treated as normally distributed [33] with expectation $\mathbb{E}(\epsilon)\mathbb{E}(\kappa) = \mu_\epsilon$ and variance $\mu_\epsilon^2 \cdot \text{Var}(\kappa) + 1^2 \cdot \sigma_\epsilon^2 = \mu_\epsilon^2 \frac{1.05^2}{C} + \sigma_\epsilon^2$ [33]. Finally, we obtain \hat{e}_f is normally distributed and its expectation and variance are as follow.

$$\begin{aligned} \mu_H &= e_f + \mu_\epsilon \\ \sigma_H^2 &= \text{Var}(e_f \kappa) + \text{Var}(\epsilon \kappa) = (e_f)^2 \frac{1.05^2}{C} + \mu_\epsilon^2 \frac{1.05^2}{C} + \sigma_\epsilon^2 \end{aligned}$$

2) *Multi-Resolution-Bitmap.* Let \mathbb{M} and \mathbb{M}' be the two associated MRB of f . The i -th layer of \mathbb{M} records some

elements from f itself, denoted as X_i , and noises from background traffic, denoted as Y_i . Recall that each element in f has a probability of $\frac{1}{2^{i+1}}$ to be recorded in the i -th layer of \mathbb{M} ($0 \leq i < R$), so $X_i \sim \text{Bino}(s_f, \frac{1}{2^i}) \approx N(\mu_X[i], \sigma_X[i]^2)$, where $\mu_X[i] = \frac{s_f}{2^i}$, and $\sigma_X[i]^2 = \frac{s_f}{2^i}(1 - \frac{1}{2^i})$. By treating background elements independently, we obtain $Y_i, Y'_i \sim \text{Bino}(k_2 N_2, \frac{1}{L_2 2^i}) \approx N(\frac{2N_2}{L_2 2^i}, \frac{2N_2}{L_2 2^i}) \triangleq N(\mu_Y[i], \sigma_Y[i]^2)$ (note that Y_i and Y'_i are i.i.d. variables).

Without loss of generality, we assume that \mathbb{M} is the one that receive less noises. After FGM, the noises recorded by the resulting MRB \mathbb{M}^* must be a subset of those recorded by \mathbb{M} . Consider a noise element that maps to $\mathbb{M}[i][j]$, the noise will stay in \mathbb{M}^* if and only if $\mathbb{M}'[i][j]$ is also set to "1". Let O'_i denote the number of "1" bits in $\mathbb{M}'[i]$. As each bit of $\mathbb{M}'[i]$ has a probability of $1 - (1 - \frac{1}{C})^{X_i + Y'_i} \approx 1 - e^{-\frac{X_i + Y'_i}{C}}$ to be set to "1", we obtain $O'_i \sim \text{Bino}(C, 1 - e^{-\frac{X_i + Y'_i}{C}})$. Since $\frac{\mathbb{E}(X_i)}{\text{Std}(X_i)} = O(\sqrt{\frac{N_2}{L_2}}) \gg 1$ and $\frac{\mathbb{E}(Y'_i)}{\text{Std}(Y'_i)} = O(\sqrt{s_f}) \gg 1$, $X_i + Y'_i$ can be replaced with $\mathbb{E}(X_i + Y'_i)$. So $O'_i \sim \text{Bino}(C, 1 - e^{-\frac{\mathbb{E}(X_i + Y'_i)}{C}}) \approx N(\mu_O[i], \sigma_O[i]^2)$, where $\mu_O[i] = C(1 - e^{-\frac{\mathbb{E}(X_i + Y'_i)}{C}})$ and $\sigma_O[i]^2 = C(1 - e^{-\frac{\mathbb{E}(X_i + Y'_i)}{C}})e^{-\frac{\mathbb{E}(X_i + Y'_i)}{C}}$. As any noise element has a probability of $\frac{O'_i}{C}$ to stay in \mathbb{M}^* , the noises recorded in $\mathbb{M}^*[i]$ is approximately

$$\epsilon_i = Y_i \frac{O'_i}{C} \sim N\left(\frac{\mu_Y[i]\mu_O[i]}{C}, \frac{\mu_Y[i]^2\sigma_O[i]^2 + \sigma_Y[i]^2\mu_O[i]^2}{C^2}\right) \\ \triangleq N(\mu_\epsilon[i], \sigma_\epsilon[i]^2)$$

So the noises recorded by \mathbb{M}^* is

$$\epsilon = \sum_{i=0}^{R-1} \epsilon_i \sim N\left(\sum_{i=0}^{R-1} \mu_\epsilon[i], \sum_{i=0}^{R-1} \sigma_\epsilon[i]^2\right) \triangleq N(\mu_\epsilon, \sigma_\epsilon^2)$$

MRB is asymptotically unbiased and its variance σ_m^2 is $\frac{0.6367}{C}$. By definition, $\hat{e}_f = (e_f + \epsilon)\kappa = e_f\kappa + \epsilon\kappa$, where $\kappa \sim N(1, \sigma_m^2)$. As $\frac{\mathbb{E}(\epsilon)}{\text{Std}(\epsilon)} = (\frac{\sigma_O[i]^2}{\mu_O[i]^2} + \frac{\sigma_Y[i]^2}{\mu_Y[i]^2})^{-\frac{1}{2}} \gg 1$ and $\frac{\mathbb{E}(\kappa)}{\text{Std}(\kappa)} = \frac{1}{\sigma_m} \gg 1$, we can treat $\epsilon\kappa$ normally distributed [33] and obtain $\epsilon\kappa \sim N(\mu_\epsilon \cdot 1, \sigma_\epsilon^2 \cdot 1^2 + \sigma_m^2 \mu_\epsilon^2) = N(\mu_\epsilon, \sigma_\epsilon^2 + \sigma_m^2 \mu_\epsilon^2)$. Then, we obtain \hat{e}_f is normally distributed and its expectation and variance are as follow.

$$\mu_M = e_f + \mu_\epsilon \\ \sigma_M^2 = \text{Var}(e_f\kappa) + \text{Var}(\epsilon\kappa) = (e_f)^2 \frac{0.6367}{C} + \mu_\epsilon^2 \frac{0.6367}{C} + \sigma_\epsilon^2$$

□

Theorem 2. The estimation of the cardinality of an elephant flow f , denoted as \hat{s}_f , has the following confidence interval:

$$(\mu_2 - Z_{\frac{1+\alpha}{2}}\sigma_2, \mu_2 + b \ln \frac{b}{b-\mathcal{T}} + Z_{\frac{1+\alpha}{2}}\sigma_2)$$

where α is the confidence level and $Z_{\frac{1+\alpha}{2}}$ is the $\frac{1+\alpha}{2}$ percentile for the standard Gaussian distribution (e.g., $Z_{\frac{1+\alpha}{2}} = 1, 2, 3$ for $\alpha = 0.68, 0.95, 0.99$).

Proof. Recall that $\hat{s}_f = b \ln \frac{b}{b-\mathcal{T}} + \hat{e}_f - \Omega$, since $\Omega \in [0, b \ln \frac{b}{b-\mathcal{T}}]$, we have

$$\hat{e}_f \leq \hat{s}_f \leq b \ln \frac{b}{b-\mathcal{T}} + \hat{e}_f$$

Since the confidence interval of \hat{e}_f is $(\mu_2 - Z_{\frac{1+\alpha}{2}}\sigma_2, \mu_2 + Z_{\frac{1+\alpha}{2}}\sigma_2)$, the Theorem follows. □

To verify the accuracy of our analysis, we compare the theoretical and experimental results on MAWI traces. Couper uses the default parameters ($b = 12, \mu = 0.6, \mathcal{T} = 9, k_1, k_2 = 2$) and the memory size is 1.5MB. Figure 7(a) shows that the theoretical misreport rate P_{mr} fits the experimental one closely. Figure 7(b) and (c) show the estimation accuracy for elephant flows when using HLL and MRB, respectively. The orange and blue line characterize the upper and lower bound of the confidence interval given $\alpha = 95\%$, and each flow is represented by a green point in the figure.

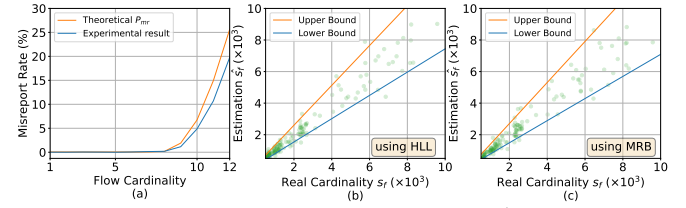


Fig. 7: Misreport Rate for $s_f \in [1, 12]$ ($\sum_{i=1}^{\mathcal{T}-1} \frac{b}{b+1-i} = 12$) and Estimation Accuracy for Elephant Flows.

V. EVALUATION

In this section, we evaluate the performance of the software implementation of Couper compared to existing frameworks.

A. Evaluation Setup

Parameter Settings:

- k_1, k_2 : 2, 3 and 4 are common choices of the number of hashed cells in sketch algorithms. Here we use $k_1, k_2 = 2$.

- E : the standard error of HLL or MRB is a monotonically decreasing function of E : $\sigma(E)$. We expect it to be less than a threshold $\theta_\sigma (= 0.15)$, so we set $E = \min\{x | \sigma(x) < \theta_\sigma\}$.

- \mathcal{T} : Since the bitmap will result in large estimation error when too many bits in it are set to ones, we set $\mathcal{T} = \frac{3}{4}b$ to avoid saturated bitmaps.

- b and μ : As the two most important parameters in Couper, they characterize how many cells are contained in each layer and how many flows will arrive at Layer2 (for a given cardinality distribution), and essentially determine the hash collision rate of the two layers (denoted by R_1, R_2). Intuitively, we should make both R_1 and R_2 as low as possible, so we ought to find the parameters that minimize $R_m = \max\{R_1, R_2\}$.

R_m should depend on the current cardinality distribution \mathcal{D} , the memory budget M , and the parameters (b, μ) . Formally, \mathcal{D} is defined as a 1-dim array, where $\mathcal{D}[x]$ denotes the number of flows with cardinality x . We say a hash collision happens to a flow in Layer- i only when all its k_i cells are shared with other flows, so the collision rate is calculated as $R_i = 1 - (1 - e^{-\frac{k_i n_i}{L_i}})^{k_i}$, where L_i can be obtained from (b, μ, M) :

$L_1 = M\mu/b, L_2 = M(1 - \mu)/E$; And n_i , the number of flows recorded by Layer- i , can be (approximately) calculated as follow: $n_1 = \sum_x \mathcal{D}[x], n_2 \approx \sum_{x \leq b \ln \frac{b}{b-\tau+1}} P_{mr}(x) \mathcal{D}[x] + \sum_{x > b \ln \frac{b}{b-\tau+1}} \mathcal{D}[x]$, where P_{mr} , the misreport rate, can be estimated using Theorem 1 given (b, μ, \mathcal{D}, M) . Hence, R_m can be analytically expressed as a function of (b, μ, \mathcal{D}, M) (denoted by $R_m(b, \mu, \mathcal{D}, M)$). We can search the parameter space $(\mathcal{B}, \mathcal{U})$ and find out an optimal parameter combination that best fits the current \mathcal{D}, M :

$$b^*, \mu^* |_{\mathcal{D}, M} = \arg \min_{b \in \mathcal{B}, \mu \in \mathcal{U}} R_m(b, \mu, \mathcal{D}, M) \quad (1)$$

Couper is agnostic to the ground-truth distribution of each data set. In our experiments, we generate \mathcal{D} by assuming a synthetic distribution and accordingly choose b, μ a priori. Here we use the Zipf- (α) distribution [30], which is widely used to mimic the distribution of unbalanced data stream [32], [43], [18], [46], where we can specify the degree of skewness through α . We apply $\mathcal{D} = \text{Zipf-}\alpha$, $M = (\sum_x \mathcal{D}[x]) \times 10\text{Bytes}$ to Eq.1 (i.e., we allocate 10Bytes to each flow in average) and Table III shows how the resulting b^*, μ^* vary with the skewness α . Both b^* and μ^* change monotonically as α ranges from 1.0 to 1.5. Since the skewness of real-world data stream is typically no less than Zipf-1.0 and seldom surpasses Zipf-1.5 [20], we can choose (b, μ) from the range $([20, 8], [0.5, 0.65])$. For example, if we have prior knowledge (e.g., from historical trace data) indicating that the skewness of current data stream is similar to Zipf-1.1, we should set $b = 16, \mu = 0.5$.

For a fair comparison, we directly assume a moderate degree of skewness (1.3) for all data sets and use $b = 12, \mu = 0.6$ in the following experiments, without fine-tuning parameters using their ground-truth distributions. And the sensitivity analysis for b and μ will be discussed soon in §V-C.

TABLE III: Recommended choices of b, μ under varying α

skewness α	1.0	1.1	1.2	1.3	1.4	1.5
b^*	20	16	13	12	10	8
μ^*	0.5	0.5	0.55	0.6	0.6	0.65

Datasets: We conduct the evaluation using three real-world data traces and one synthetic data trace: **A) CAIDA19:** It is the anonymized IP trace stream collected in 2019 from CAIDA [4]. We use the destination-IP as flow ID and the source-IP as element ID. **B) MAWI:** This is a traffic data repository maintained by the MAWI Working Group of the WIDE Project [21]. Flow ID and element ID are defined as source-IP and destination-IP carried in each packet respectively. **C) FACEBOOK:** We use the FACEBOOK page data set collected in 2017 [10]. Each row in the file is an edge with two nodes. Nodes represent the pages and edges are mutual likes among them. We use the first node of each edge as flow ID, and the second node as element ID. **D) Zipf:** We generate data sets where the cardinality of flows follow a Zipf-1.0 distribution.

We divide each data set into several subsets, conduct evaluations on one of them in each time window and obtain averaged results over all epochs. We summarize the statistics of different data subsets in Table IV.

TABLE IV: Statistics of data subsets.

Data set	# flows	$\sum_{i=1}^n s_f$	99.99th cardinality	99.9th cardinality
CAIDA	370K	3.9M	4.4K	1.4K
MAWI	100K	3.3M	49.0K	2.8K
FACEBOOK	110K	1.4M	0.7K	0.3K
Zipf-1.0	100K	2M	15K	1.6K

Metrics: See Table V and VI.

Implementation: We have implemented Couper and all the other frameworks in C++. All experiments are conducted on a server with dual 8C16T CPU (Intel Xeon Silver 4215R CPU @ 3.20GHz) and a total of 192G DDR4 RAM, running Ubuntu 18.04.1. We use 128-bit MurmurHash3 [24] as the default hash function. Before running each experiment on a data set, we load the whole data set into memory to exclude any disk I/O overhead.

B. Per-Flow Cardinality Estimation

The baselines involved in the comparisons include gSkt [47], rerSkt [38] and VHS [41]. We use two types of estimators (HLL and LC) for gSkt and rerSkt. The parameters of all baselines (e.g. size of bitmaps for LC, number of registers in HLL) remain unchanged from their original settings.

1) *Memory Efficiency & Accuracy:* First, we set an expected Average Ratio Error and compare AMC among all the frameworks (Figure 8 (a)-(b)). Specifically, we set an initial memory footprint for each framework, which is increased step by step until achieving the expected accuracy. The results on all the four data sets show that, to achieve an Average Ratio Error of 1.5 and 1.25, the AMC of Couper is at least $20\times$ and $16\times$ lower than the best baseline, respectively.

Second, we evaluate the Average Ratio Error over varied memory allocation (Figure 8 (c)-(f)). The results show that, with a memory footprint of 2MB, only Couper can guarantee the Average Ratio Error to be less than 1.5 on all the four data sets (We do not plot rerSkt-LC and gSkt as their error are too high to make sense). In addition, we summarize the 80-th and 99-th percentile error in Table VII. It is worth mentioning that the 99-th percentile error of Couper-MRB is $5.4\times \sim 12.2\times$, $48.5\times \sim 63.6\times$ and $4.9\times \sim 21.3\times$ lower than rerSkt-HLL, gSkt-HLL and VHS, respectively.

2) *Throughput:* Figure 9 compares the update and query throughput among all the frameworks. They all achieve high

TABLE V: Metrics for Per-Flow Cardinality Estimation

Ratio Error (RE)	$\max\{\frac{\hat{s}}{s}, \frac{s}{\hat{s}}\}$, where \hat{s} and s are estimated and true flow cardinality. Its minimum value is 1.
Average Memory Cost (AMC)	The averaged memory cost of each flow to achieve a specified accuracy.
Update Throughput	# packets can be processed per second.
Query Throughput	# queries can be performed per second.

TABLE VI: Metrics for Super Spreader Detection

Precision	the ratio of true super spreaders detected to all super spreaders reported by the sketch.
Recall	the ratio of true super spreaders reported by the sketch to all true super spreaders.
F1-Score	the harmonic average of precision and recall.

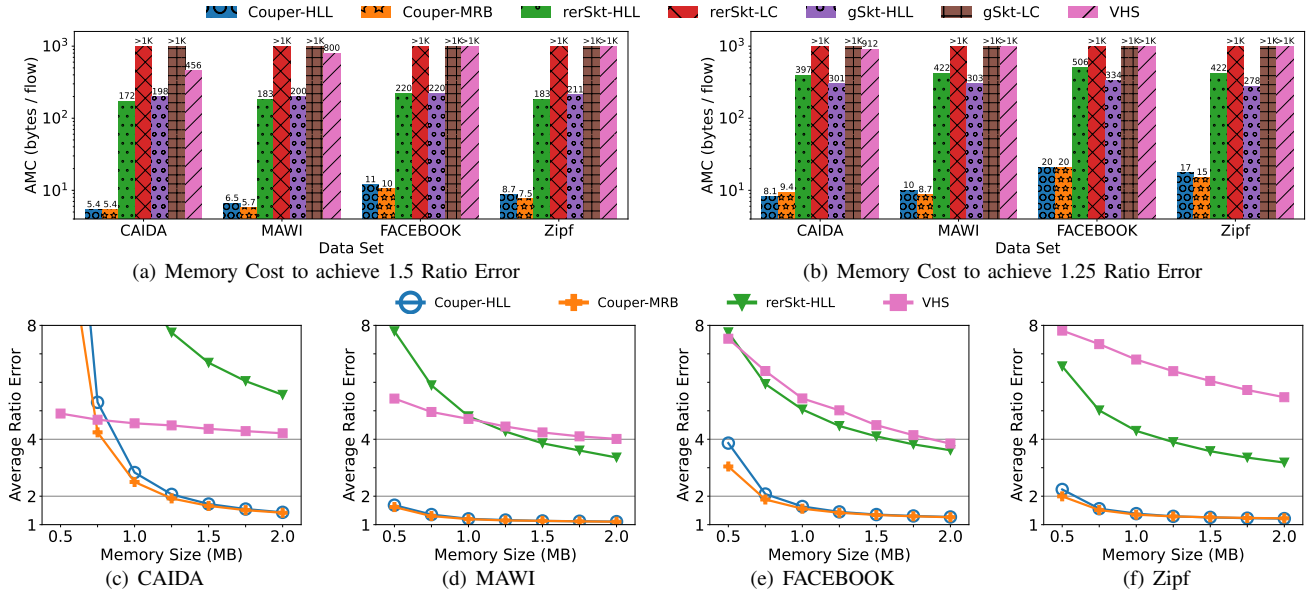


Fig. 8: Accuracy & Memory Efficiency Comparison for Per-Flow Cardinality Estimation

Data set	CAIDA		MAWI		FACEBOOK		Zipf	
Percentile	80-th	99-th	80-th	99-th	80-th	99-th	80-th	99-th
Couper-HLL	2	10	1.1	2.2	1.4	5.6	1.3	3.4
Couper-MRB	2	8	1.1	2.3	1.5	4.8	1.4	3.4
rerSkt-HLL	8.8	51	5	28	5.9	26	5	21
rerSkt-LC	21	122	17	124	11	56	10	57
gSkt-HLL	283	509	62	133	102	266	55	165
gSkt-LC	>1K	>1K	>1K	>1K	>1K	>1K	710	>1K
VHS	6	39	4	49	7	22	8	39

TABLE VII: Key Percentiles of Ratio Error Distribution

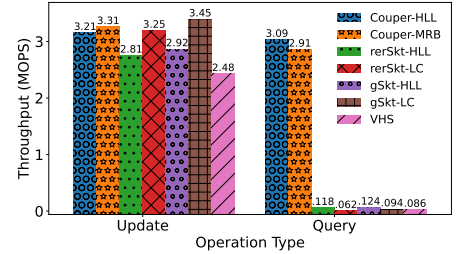


Fig. 9: Throughput

update throughput, but the query throughput of all the baselines can only achieve 0.1% ~ 2% of their update throughput, which is expected since they need hundreds or thousands of memory accesses and addition operations for each estimation. In contrast, Couper has much higher query throughput. The reason is that most mice flows are only recorded by Layer1, and the estimation to their cardinality has a time complexity of $O(b)$. As b is relatively small and mice flows take the majority, Couper achieves high amortized query throughput.

C. Ablation and Parameter Analyses

Ablation Studies. Here we tease apart the usefulness of the two optimization techniques, i.e., *removing overlapping error* (ROE, §III-C) and *fine-grained-merging* (FGM, §III-D).

- **ROE:** Figure 10 (a) compares the detailed error distribution when ROE is enabled and disabled. The flows are placed in bins based on their true cardinality. We plot the average ratio error for each bin, along with an error bar representing the 0.1- and 0.9- percentile error. We observe that the flows with cardinality within interval $[12, 26]$ see a remarkable accuracy improvement through ROE, e.g., the average ratio error is reduced by 20% ~ 30% and the 0.9-percentile error is reduced by 30% ~ 45%. As the flow cardinality exceeds 30, this improvement begins to diminish. This is expected since the overlapping error Ω ranges from 0 to $b \ln \frac{b}{b-\mathcal{T}}$, so ROE has

only marginal effect on those with cardinality far greater than $b \ln \frac{b}{b-\mathcal{T}}$.

- **FGM:** Now we study how much our FGM can improve over a traditional cells-combining strategy, which we call the *baseline*. The baseline and FGM have the following differences: A) When checking whether a flow has collected enough coupons, the baseline counts the number of “1” bits in each of the selected bitmaps, and returns true if all of them are no less than \mathcal{T} ; B) Upon receiving a query, the baseline performs estimation on all the hashed cells in each layer, and sum up the minimum result in each layer as the final estimation. We compare the number of misreported flows and average ratio error between the baseline and FGM, as shown in Figure 10(b). FGM has lower misreport rate and reduces the average error by more than 40%. Finally, Figure 10(c) compares the accuracy improvement brought about by each technique.

Sensitivity Analysis of parameter b and μ . Recall that (b, μ) can be chosen from the range $([20, 8], [0.5, 0.65])$. By default, we used $(b = 12; \mu = 0.6)$ in our previous experiments. Now we study the performance (in terms of AMC) of different parameter combinations, each of which corresponds to a Zipf prior distribution with skewness α chosen from the interval $[1.0, 1.5]$, and the mapping from α to (b, μ) is given in TableIII. The results in Figure 10 (d) indicate that A) different data sets prefer different parameters, which means their distributions

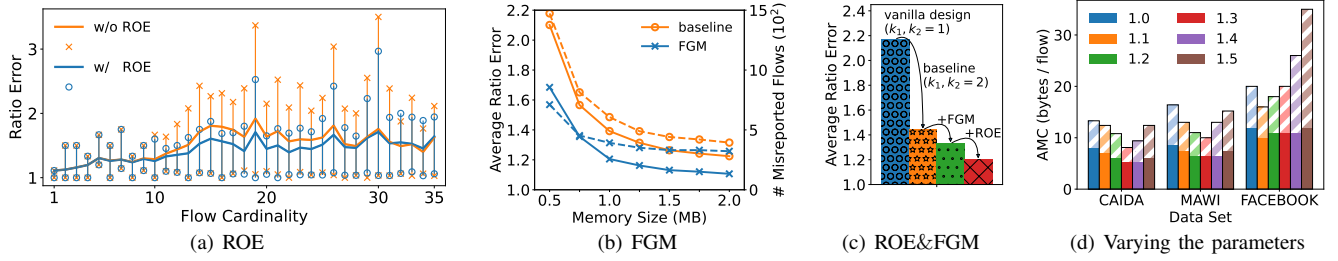


Fig. 10: Ablation and Parameter Analyses: Couper uses HLL in Layer2; (a),(b),(c) are conducted on MAWI data sets; (a),(c) use 1MB memory; The dotted and solid lines in (b) represent # misreported flows and average ratio error, respectively; The hatched and solid bars in (d) denote the AMC to achieve an average ratio error of 1.25 and 1.5, respectively.

TABLE VIII: Precision, Recall and F1-Score for Super Spreader Detection. Couper uses HLL in Layer2.

Methods	MAWI ($\tau = 0.5\%$)						MAWI ($\tau = 0.05\%$)						CAIDA ($\tau = 0.2\%$)						CAIDA ($\tau = 0.04\%$)					
	0.5MB			1MB			0.5MB			1MB			0.5MB			1MB			0.5MB			1MB		
	PRE	REC	F1	PRE	REC	F1	PRE	REC	F1	PRE	REC	F1	PRE	REC	F1	PRE	REC	F1	PRE	REC	F1	PRE	REC	F1
VBF	0	0	0	0	0	0	0	0	0	0.01	1	0.01	0	0	0	0	0	0	0	0	0	0.01	1	0.01
CDS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SS	0.94	0.97	0.95	0.94	0.97	0.95	0.73	0.97	0.83	0.9	0.99	0.95	0.77	1	0.87	0.94	1	0.97	0.1	0.82	0.17	0.3	0.97	0.45
Couper	1	0.97	0.98	1	0.97	0.98	0.82	0.97	0.89	0.95	0.96	0.96	1	0.95	0.97	1	0.95	0.98	0.46	0.94	0.62	0.79	0.94	0.86

have different skewness, e.g., FACEBOOK may have lower skewness than MAWI and CAIDA; B) A sub-optimal parameter combination may harm the performance of Couper, but it still significantly outperforms state-of-the-art.

D. Super Spreader Detection

Baselines include *Vector Bloom Filter* (VBF) [19], *Spread Sketch* (SS) [35] and *Connection Degree Sketch* (CDS) [39]. The authors of [35] did not specify the parameter σ in SS and we set it to 0.1 (after trying 0.05, 0.1 and 0.2, we find 0.1 to be the best). We use two *reversible* sketches and one *validation* sketch in CDS. We use $Z = 2048$ for Couper (Note that T_{SC} and T_E altogether accounts for the $1 - \mu$ memory allocated to Layer2).

1) *Accuracy*: Table VIII evaluates the *Precision*, *Recall* and *F1-Score* of all frameworks on CAIDA and MAWI traces under 0.5MB and 1MB memory footprint. The threshold of super spreader is defined as $\theta_s = \tau \sum_f s_f$, where $\tau < 1$ and $\sum_f s_f$ is the combined cardinality of all flows. We have the following observations. A) Under all conditions, VBF and CDS have extremely low precision or/and recall, their F1 Scores are both around 0. B) The state-of-the-art method SS performs well on MAWI, but has a low precision on CAIDA ($\tau = 0.04\%$), especially when the memory is tight. C) Couper always achieves the highest precision and F1-Score, while keeping a high recall.

The high accuracy of Couper benefits from its *two-layer* design. As most of the mice flows can be filtered out by Layer1, only a small portion of flows are considered as super spreader candidates and reported to T_{SC} , which makes it easy for Couper to handle a large number of flows. In contrast, other baselines have to record the elements from all the flows. Under tight memory constraints, multiple mice flows could be recorded in the same “bucket”, making it difficult to distinguish between buckets occupied by real super spreaders

and those accommodating many mice flows, finally incurring severe precision degradations.

2) *Improve prior work with Couper*: We use the first layer of Couper ($CP:L_1$) to optimize existing super spreader detection algorithms, as described in §III-E2. For a fair comparison, Couper should not incur additional memory cost, so we reserve 200KB memory for $CP:L_1$ (denoted as $M_{CP:L_1}$) when evaluating each algorithm. The results in Table IX show that all the frameworks gain improvements in precision (Δ_{pre}) and F1-Score (Δ_{F1}).

TABLE IX: Accuracy Improvement brought about by Couper

Data Set	SS (1MB)		CDF (30MB)		VBF (30MB)	
	Δ_{pre}	Δ_{F1}	Δ_{pre}	Δ_{F1}	Δ_{pre}	Δ_{F1}
CAIDA	0.28	0.26	0.58	0.43	0.04	0.06
MAWI	0.07	0.01	0.58	0.82	0.04	0.25

VI. HARDWARE IMPLEMENTATION

The Couper algorithm presented in previous sections is designed to be implemented as software running on general-purpose CPUs. In spite of the flexibility of supporting any kind of operations, CPU, along with the NICs in end hosts, can become a performance bottleneck. High-end commodity NICs can only achieve a throughput of ~ 200 Gbps, let alone the lower single core TCP processing capacity which struggles to achieve 100Gbps even with a highly optimized protocol stack. Thanks to the Protocol Independent Switch Architecture (PISA), P4-enabled programmable switches with multiple 100G/400G ports can offer over 10Tbps of throughput, which is orders of magnitude faster than high-end NICs. However, the PISA also comes with extra constraints. In this section, we describe how to adapt the software version of Couper to a hardware-friendly version which runs on the Tofino 1 silicon.

Figure 11 shows the logical PISA in Tofino, as well as the workflow of hardware-friendly Couper. Tofino consists of

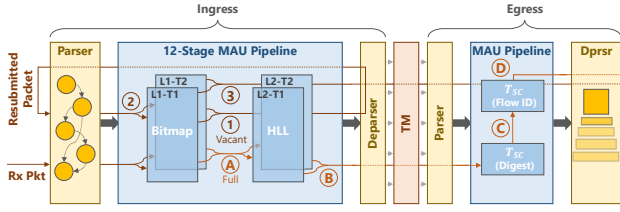


Fig. 11: Workflow of Hardware Implementation

two identical pipelines — Ingress and Egress — connected by a Traffic Management (TM) module. Each pipeline has a parser, 12 stages of Match-Action-Units(MAU) and a deparser. Stages are organized in a feed-forward fashion, which means packets can only travel from one stage to the next stage. Such an architecture brings the following critical constraints: **C1: All memory resources are stage-local.** When processing a packet, one cannot access registers allocated to previous stages, nor future stages. **C2: All operations within a stage are executed in parallel.** For example, given two operations $op1$ and $op2$, if $op2$ depends on the result of $op1$, they must be placed in different stages. **C3: No circular dependency is allowed.** This can be concluded by putting **C1** and **C2** together. **C4: Limited ALU and control flow support.** Compared with C/C++, only a subset of arithmetic and logical operations are allowed in Tofino. Unsupported operations include floating-point operations and loops. To fit Couper into rigid hardware constraints, we make the following modifications or workarounds:

(1) Change algorithm configuration: The bitmap size is changed from 12 to 16 since P4 only supports byte-aligned memory accesses. Also, Tofino cannot compute MurmurHash, so we replace all hash functions with the native supported CRC hash. Finally, to avoid circular dependency, we remove the power of two for T_{SC} .

(2) Replace unsupported operations with table lookup. Operations such as *a*) checking whether a bitmap is full, and *b*) counting leading zeros when updating HLLs do not have straightforward implementations in P4 due to **C4**. So we replace such operations with equivalent table lookup. For example, a bitmap cell is considered full if \mathcal{T} (12) out of b (16) bits are set to 1, which means we have $\binom{16}{12} = 1820$ distinct full-bitmap patterns. By hard-coding all patterns into TCAM, we are able to check bitmap fullness in one clock cycle.

(3) Redesign packet data-path to break circular dependency and enable multiple memory access. Full workflow is shown in Figure 11, each incoming packet goes to L1 and checks whether the flow it belongs to has got enough coupons. If not, it follows the data-path ①: set resubmit flag, inject the packet to the ingress parser again at the ingress deparser; ②: perform update on Layer1; ③: forward packet as usual. Otherwise, the packet will follow another data-path ④: perform update on layer2; ⑤: if Layer2 has its value changed, send a digest to the egress pipeline and store it in the digest part of T_{SC} ; ⑥: update the flowID part of T_{SC} ; ⑦: forward packet as usual.

Finally, the hardware version of Couper is implemented in P4_16. We present the resource utilization of the hardware-

TABLE X: Hardware Resource Utilization

SRAM	TCAM	Instructions	Gateway	Hash Dist Unit
12.6%	0.35%	4.43%	8.33%	22.22%

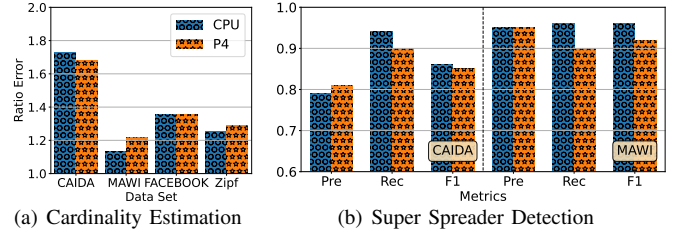


Fig. 12: Accuracy comparison between P4 and CPU implementations, the memory size is 1.5MB and Layer2 uses HLL.

friendly Couper in Table X and its accuracy compared with the software implementation in Figure 12. The evaluations are conducted on a programmable switch equipped with a Tofino 1 silicon (BFN-T10-032D). We use two servers to generate and receive packets, both of which are equipped with a 100Gbps Mellanox CX-6 NIC. To summarize, A) The algorithm consumes 22% of the programmable switch’s Hash Dist Unit since hash operations are heavily used to determine the indexes in register arrays. Utilization of other hardware components remains low and we expect Couper to easily coexist with other data-plane algorithms. B) For per-flow cardinality estimation, the P4 version achieves similar ratio error compared to the CPU version. C) For super spreader detection, the P4 version has slightly worse recall / F1-score, we owe the difference to the lack of the power of two choices.

VII. CONCLUSION AND FUTURE WORK

We present Couper, a memory efficient framework for per-flow cardinality estimation. In this work, we propose a two-layer structure to conquer the unbalanced cardinality distribution, along with two optimization techniques (i.e., ROE and FGM) to mitigate the overlapping error and hash collision error. And we further extend it to detect super spreaders with trivial additional overhead. Extensive experiments show that Couper works well on both software and hardware platforms. Next, we discuss potential limitations of Couper and opportunities for further improvement.

Tuning Parameters Adaptively. The current Couper chooses parameters (b, μ) by assuming a prior skewness for the underlying distribution (§V-A). The prior may not always fit the real distribution well and result in sub-optimal performance. We can enhance this point by tuning parameters adaptively. Specifically, at the end of each epoch, we can evaluate the performance of the current parameters (e.g., the memory utilization, hash collision rate, etc.), and decide on the parameters to be used in the next epoch. Typically, *Bayesian Optimization* [14], [13] can provide an efficient decision-making schema and enable Couper to quickly adapt to the real distribution.

Handle Imbalance at Varied Levels. The current two-layer structure treats all flows that arrive at Layer2 equally. Since the imbalance also exists in elephant flows, some estimators of Layer2 may still suffer low utilization. To tackle this issue, we

can make Layer2 hierarchical, where the estimator in higher level has larger capacity, while consuming more memory. For example, when using MRB as the cardinality estimator, we can fix the number of columns but vary the number of rows for MRB in different levels. A flow enters the next level when the associated MRB in the current level achieves a high load factor. Finally, we can estimate the cardinality for a flow by merging the associated MRB in all levels it arrives at.

REFERENCES

- [1] A. Agarwal, Z. Liu, and S. Seshan. {HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 719–741, 2022.
- [2] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.
- [3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.
- [4] CAIDA. <https://www.caida.org/catalog/datasets/overview/>, 2019.
- [5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.
- [6] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Citeseer, 2004.
- [7] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
- [8] Couper. <https://github.com/CouperProj/Couper.git>, 2022.
- [9] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166, 2003.
- [10] FACEBOOK. <https://snap.stanford.edu/data/facebook-large-page-page-network.html>, 2017.
- [11] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX security symposium (USENIX Security 15)*, pages 817–832, 2015.
- [12] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.
- [13] P. I. Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.
- [14] R. Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.
- [15] A. Hall, O. Bachmann, R. Büssow, S. Găncăanu, and M. Nunkesser. Processing a trillion cells per mouse click. *arXiv preprint arXiv:1208.0225*, 2012.
- [16] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013.
- [17] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NcCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.
- [18] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.
- [19] W. Liu, W. Qu, J. Gong, and K. Li. Detection of superpoints using a vector bloom filter. *IEEE Transactions on Information Forensics and Security*, 11(3):514–527, 2015.
- [20] N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 68(4):415–430, 2009.
- [21] MAWI. <https://mawi.wide.ad.jp/mawi/>, 2019.
- [22] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.
- [23] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006.
- [24] murmurhash. <https://sites.google.com/site/murmurhash/>, 2010.
- [25] S. Nadarajah and S. Kotz. Exact distribution of the max/min of two gaussian random variables. *IEEE Transactions on very large scale integration (VLSI) systems*, 16(2):210–212, 2008.
- [26] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
- [27] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internets. *ACM SIGCOMM computer communication review*, 31(4):15–26, 2001.
- [28] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.
- [29] D. Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *LISA*, pages 305–317, 2000.
- [30] D. M. Powers. Applications and explanations of zipf’s law. In *New methods in language processing and computational natural language learning*, 1998.
- [31] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.
- [32] P. Roy, A. Khan, and G. Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463, 2016.
- [33] A. Seijas-Macias and A. Oliveira. An approach to distribution of the product of two normal variables. *Discussiones Mathematicae Probability and Statistics*, 32(1-2):87–99, 2012.
- [34] Y.-E. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao. Online spread estimation with non-duplicate sampling. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 2440–2448. IEEE, 2020.
- [35] L. Tang, Q. Huang, and P. P. Lee. Spreadsketch: Toward invertible and network-wide detection of superspreaders. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1608–1617. IEEE, 2020.
- [36] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. Technical report, Carnegie-Mellon Univ Pittsburgh Pa School Of Computer Science, 2004.
- [37] H. Wang, C. Ma, S. Chen, and Y. Wang. Online cardinality estimation by self-morphing bitmaps. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1–13. IEEE, 2022.
- [38] H. Wang, C. Ma, O. O. Odegbile, S. Chen, and J.-K. Peir. Randomized error removal for online spread estimation in data streaming. *Proceedings of the VLDB Endowment*, 14(6):1040–1052, 2021.
- [39] P. Wang, X. Guan, T. Qin, and Q. Huang. A data streaming method for monitoring host connection degrees of high-speed links. *IEEE Transactions on Information Forensics and Security*, 6(3):1086–1098, 2011.
- [40] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.
- [41] Q. Xiao, S. Chen, M. Chen, and Y. Ling. Hyper-compact virtual estimators for big network data based on register sharing. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 417–428, 2015.
- [42] 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*, pages 561–575, 2018.
- [43] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li. Heavykeeper: An accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.
- [44] M. Yoon, T. Li, S. Chen, and J.-K. Peir. Fit a spread estimator in small memory. In *IEEE INFOCOM 2009*, pages 504–512. IEEE, 2009.
- [45] B. Zhao, X. Li, B. Tian, Z. Mei, and W. Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.
- [46] 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*, pages 741–756, 2018.
- [47] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O. Odegbile. Generalized sketch families for network traffic measurement. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–34, 2019.