BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time

Mark Braverman, Xiaoqi Chen, Shir Landau Feibish, Jennifer Rexford
{mbraverm,xiaoqic,sfeibish,jrex}@cs.princeton.edu
Princeton University

ABSTRACT

Network administrators constantly need to monitor network traffic for congestion and attacks. To do so, they must perform a large number of measurements on the traffic simultaneously, to detect different types of anomalies such as heavy hitters or super-spreaders. However, performing all of these measurements within the constrained memory architecture of modern network devices poses significant challenges. Specifically, despite significant growth in the memory *size* in these devices, the number of memory *accesses* per packet remains extremely limited. We propose BeauCoup, a system based on coupon collectors that supports many traffic-monitoring queries simultaneously while making at most *one* memory update for each packet. BeauCoup can be implemented in environments where memory access is very expensive, such as high-speed programmable switches.

1 INTRODUCTION

Network operators constantly monitor network traffic to detect attacks, performance problems, and faulty equipment. For example, they need to identify flows that are heavy hitters (e.g., a destination IP that receives a large volume of traffic) or super-spreaders [28] (e.g., a source IP that communicates with many different destination IP addresses). Network operators typically worry about multiple kinds of problems simultaneously. For example, they may check continuously for worms, port scans, DDoS attacks, SYN floods, and heavy flows. Each of these monitoring tasks (or *queries*) can measure the traffic based on different characteristics. To detect a worm we may need to look for source IPs that are sending traffic to many different destinations, whereas to detect a heavy hitter flow we may look for a flow (defined by the tuple *(source IP, destination IP)*) that generates a large volume of traffic. Indeed, efficient techniques for network traffic monitoring have been widely studied [19, 20, 26, 27, 31].

Emerging programmable network switches can analyze traffic directly in the data plane as packets stream by. These programmable switches offer significant flexibility in the tasks they can perform; yet, to maintain line rate, they impose significant constraints on memory and processing resources. As the packet is processed by the switch, a limited number of simple computations may be performed; additionally, the switch has a finite amount of SRAM for maintaining state across packets. Furthermore, each packet passing through the switch may access only a small handful of locations in this memory. In fact, the limitation on the number of on-chip memory *accesses* per packet is much more significant than the constraint on the total *size* of memory, particularly when trying to support many queries simultaneously.

While some past systems can support *one* query entirely in the data-plane hardware (e.g., for heavy-hitter detection [2, 27]), existing techniques for handling *multiple* queries rely heavily on software support:

- **Sampling:** The simplest approach is to sample packets in the data plane [6], and have a software controller compute multiple statistics on the samples. While useful for detecting high-volume flows, sampling is less effective for more sophisticated queries such as counting the number of distinct attributes (e.g., distinct destination IPs in detecting sources that are super-spreaders).
- **Key-value stores:** Systems such as Marple [22] and Sonata [16] use the data plane to support *portions* of multiple queries, including packet filtering and simple key-value stores. However, to maintain per-flow state, these systems offload storage and processing to software after exhausting the limited data-plane resources.
- **Sketches:** Compact data structures are an attractive alternative to storing per-flow state. Several recent works collect information about all potentially relevant flows in a data-plane sketch, and then have the controller software analyze the sketch to compute the statistics of interest [19, 20, 29], Some solutions can support *multiple* queries on the *same* packet dimension (or *key*), using a general purpose sketch. However, supporting queries with multiple *different* keys would require a separate data structure for each dimension.

While using compact data structures to answer queries *approximately* is clearly on the right track, we need new techniques that can handle numerous heterogeneous queries in the data plane, despite the limitation on memory accesses. We present BeauCoup, which supports a general query abstraction that counts the number of distinct items (i.e., with different *attributes*) seen across a set of related packets (with the same *key*), and flags the keys with distinct counts above

a *threshold*. For example, in the super-spreader query, the source IP is the key, the destination IP is the attribute, and the threshold checks whether a source IP has communicated with too many different destination IPs. However, to handle many queries with different keys and attributes accurately and efficiently, BeauCoup *cannot* simply maintain multiple sketches (on different keys) and alternate among the sketches arbitrarily for different packets.

Instead, we appeal to the *coupon collector problem* [13] to identify a suitable subset of packets (based on their attributes) to trigger updates for each query. A coupon collector is an algorithm that consists of a set of coupons, and a probability for selecting each coupon. For each item in the stream, one of the coupons is selected randomly (with replacement), and the coupon collector is satisfied when all of the coupons have been collected. In our setting, we select a coupon for each query by computing a random hash function over the packet's attribute. By adjusting the parameters of the coupon collector for each query, BeauCoup bounds the total number of memory accesses for each packet. In essence, each packet collects a coupon for a bounded number of queries, but in a carefully chosen way as to achieve high accuracy for all of the queries. In designing and evaluating BeauCoup, we make the following contributions:

Limited access memory model: We introduce a computational model for analyzing network traffic under a constant number of memory accesses per packet. Our model brings together streaming inputs (e.g., cash-register [21]) with constraints on how memory is divided and accessed (e.g., cell-probe [30]). This model, presented in § 2, is relevant to hardware devices that process a streaming input, and is of independent interest.

Query compilation that bounds memory access: We show how to map each query to an equivalent query composed solely of a distinct counter in § 3. We execute each of the given queries using a coupon collector, which is configured to approximately identify when the distinct count has reached the threshold. Our query compiler allocates the limited memory access among multiple queries. It finds the optimal coupon-collector parameters for each query, and mediates the contention between queries to perform at most one memory access per packet, as described in § 4.

Adapting BeauCoup to PISA architecture: We outline how to implement BeauCoup within the packet-processing pipeline of PISA (Protocol-Independent Switch Architecture) switches in § 5. We compile the queries to generate coupon parameters that fit the computational constraints of PISA. Then, we map the coupon collection scheme to use TCAM match rules, to collect coupons for all queries in parallel.

In the remainder of the paper we describe the results of our evaluation in § 6, summarize related work in § 7, and present concluding remarks in § 8.

2 MANY QUERIES WITH FEW UPDATES

Limited memory access has become a major constraint in hardware devices, due to the mismatch between the growth of network link speed and the relatively slower growth of memory access speed. The main goal of BeauCoup is to support multiple network monitoring queries on hardware which only allows limited memory access. We begin this section by framing the computational model of limited memory access. We then describe how to formulate queries to distinct counters, and finally we define the relevant performance metrics for algorithms created for this model.

2.1 Limited Memory Access Model

Our model for processing network packets closely resembles the *cash register* streaming model (addition only), while the memory access constraint resembles a variant of *cell probe* model (can access a limited number of memory locations).

An algorithm $\mathcal{A}$ running under the limited memory access streaming model will observe an infinite stream of network packets $i_1, \ldots, i_N$ sequentially. When processing packet i_t, the algorithm may opt to read or write $O(1)$ memory locations, as well as produce some output $\mathcal{A}(i_1, \ldots, i_t)$. Now, we formally define the limited memory access streaming model:

(1) **Word-based access:** A word is a minimal unit of memory to be read or written together. Depending on the platform, it typically contains 32 or 64 bits.
(2) **Constant number of memory accesses:** When processing packet i_t, the algorithm can only access Γ words of memory per packet, where Γ is a small constant. The algorithm can read a particular word $\mathcal{M}[j]$, perform arbitrary computation, and write a new value into the same memory location; this is considered one memory access.
(3) **Sub-linear memory size:** The system has memory array $\mathcal{M}$ with a limited size, typically sub-linear to the stream size, with the j-th word referred to as $\mathcal{M}[j]$.

Sub-constant Access Memory Model We further define a sub-model of the limited memory access model with $\gamma \leq 1$ memory access per packet. Note that the algorithm cannot make fractional word access, therefore can only access a few whole words per packet (fewer than Γ), and should access fewer than γ words per packet *on average*.

In this paper, we will be using the sub-constant access memory model. For completeness, we summarize the parameters of the model in the following definition.

Definition 2.1. A streaming algorithm $\mathcal{A}$ in the sub-constant access memory model with parameters (M, w, γ, Γ) is an algorithm that uses a memory of M words of w bits each, and moreover:

- No more than Γ memory access (read/write operations) are permitted per item; and

- On any stream of N items, no more than $\gamma \cdot N + o(N)$ accesses are performed in total.

Typically, we expect memory size $M = o(N)$, but not as constrained as in traditional streaming algorithms (e.g., memory size $M = N^{1/2}$ is often acceptable). Γ is a relatively small constant (such as 1 or 5), and γ is very small (e.g. 0.01).

The main advantage of sub-constant access algorithms is that we can expect to run many of them in parallel on a single Constant Access Memory unit. Generally speaking, if we have a family of k sub-constant-access algorithms $\{\mathcal{A}_i\}_{i=1}^{k}$ with parameters $(M_i, w, \gamma_i, \Gamma)$, we can expect to be able to run them jointly on the same stream with the resulting algorithm $\mathcal{A}$ having parameters $\left(\sum_{i=1}^{k} M_i, w, \sum_{i=1}^{k} \gamma_i, \Gamma\right)$. $\mathcal{A}$ can be executed on the Constant Access Memory Model, as long as $\sum_{i=1}^{k} \gamma_i < \Gamma$, assuming we manage to time-share each $\mathcal{A}_i$'s memory access.

2.2 Flexible Queries as Distinct Counters

A variety of network-monitoring tasks can be modelled as a distinct counter, counting the number of distinct attributes seen across a set of packets. For example, a *super-spreader* is defined as a host in the network thad sends packets to many (e.g., 1000+) distinct destinations. Finding a super-spreader can be characterized as finding a set of packets with the same source IP address, and many distinct destination IP addresses. In this section, we formally define our query model, and give some more examples of queries in this format.

A query q consists of a key projection key_q, attribute projection $attr_q$, and threshold T_q. When executing this query, each packet i from the input stream is projected to a key $key_q(i)$ and an attribute $attr_q(i)$. The query is defined as: for any particular key k, output an alert (q, k) when the set of input packets satisfying $key_q(i) = k$ has at least T_q different values of $attr_q(i)$. That is, output (q, k) when:

$$\left|\{attr_q(i) \mid key_q(i) = k\}\right| > T_q.$$

The super-spreader example we mentioned earlier can be expressed as a query with $key_q(i) = \{i.srcIP\}$, $attr_q(i) = \{i.dstIP\}$, and threshold $T_q = 1000$. Our goal is to build a system that simultaneously executes a set of queries $Q = \{q_1, q_2, \dots\}$, and output alerts (q_j, k_j) while processing the input packet stream.

In Table 1, we present more examples of how to formulate common network-monitoring tasks in our query model. In particular, we assume $i.timestamp$ is distinct across all packets, so the user may write a query to count packets by defining $attr_q(i) = \{i.timestamp\}$, i.e., counting the number of timestamps seen. We also note that filtering operations can be implicitly reduced into this query formulation, as shown in the SYN-flood example above—by projecting irrelevant packets to a fixed value, the distinct counting query effectively ignores them.

Name	Key	Attribute	Threshold
Super-spreader	$srcIP$	$dstIP$	1000
Port scan	$srcIP$	$dstPort$	100
Heavy hitter IP pair	$\{srcIP, dstIP\}$	$timestamp$	10000
Heavy hitter IP&port pair	$\{srcIP, srcPort, dstIP, dstPort\}$	$timestamp$	10000
SYN-flood	$\{dstIP, dstPort\}$	$\{srcIP, srcPort\}$ if TCP SYN, otherwise $\emptyset$	5000

Table 1: Examples of query parameters.

Many other network-monitoring tasks can be expressed in this formulation by using a combination of packet IP addresses, ports, timestamps, etc. as the query key and query attribute. Our goal in this work is to execute many such queries simultaneously using one algorithm, under a strict memory access constraint of $\Gamma = 1$ shared among all queries.

One natural way to share memory accesses is to allocate γ_q memory accesses to each query q, such that each query q, when run individually in a sandboxed environment, makes fewer than γ_q memory access per packet. When running together, all queries would make $\sum_{q \in Q} \gamma_q$ memory accesses per packet on average. We can thus satisfy the average memory access constraint easily by using small γ_q such that $\sum_{q \in Q} \gamma_q \leq \Gamma$. However, when processing a single packet, we need to arbitrate between multiple queries when they simultaneously want to access memory, so we do not violate the constraint of accessing at most $\Gamma = 1$ word per packet.

2.3 Performance Metrics

It may be impossible to make timely and accurate reports whenever a query exceeds its threshold, especially when operating under memory access constraint Γ. An approximated algorithm may alert too early, or too late. We define the relative error of an algorithm as follows:

- **True count:** Say the algorithm first outputs (q, k) at time t, after witnessing input stream $i_1, i_2, \dots, i_t$; at this time, the true number of distinct attributes seen by the algorithm is $\mathcal{T}(\mathcal{A}^q) = \left|\{attr_q(i) \mid key_q(i) = k, i \in i_1, i_2, \dots, i_t\}\right|$.
- **Absolute error:** However, the algorithm should have reported when there is exactly T_q distinct items. We define its absolute error as $|\mathcal{T}(\mathcal{A}^q) - T_q|$.
- **Relative error:** We normalize and use $\frac{|\mathcal{T}(\mathcal{A}^q) - T_q|}{T_q}$ as the relative error of output (q, k). This scaled error includes both the bias $E[\mathcal{T}(\mathcal{A}^q)] - T_q$ and the variance of $\mathcal{T}(\mathcal{A}^q)$.

3 QUERY AS A COUPON COLLECTOR

In this section, we examine how BeauCoup answers a single query.

A coupon collector consists of a set of coupons and a probability for selecting each coupon. In the basic algorithm, for each packet, one of the coupons is randomly selected (with replacement), and the algorithm completes when all of the coupons have been collected. Intuitively, the coupon collector can answer the following question: if there are k coupons, what is the probability that one would need over j packets to collect all of the coupons. By adjusting the likelihood (probability) of collecting each coupon, we minimize the expected number of coupons that need to be updated for each packet. Therefore, we can simultaneously try to collect coupons for many queries, while only performing sub-constant memory access on average.

We now show how coupon collectors may be used as distinct counters. Then, we explain how a query may be transformed to a coupon collector.

3.1 Coupon Collector as a Distinct Counter

It is impossible to answer distinct counting queries exactly under the sub-constant memory access model. Fortunately, approximate algorithms [1, 7, 8, 12–14] provide answers with relatively high accuracy, while only requiring memory that is logarithmic in the number of distinct keys in the stream.

We utilize a well-known algorithm called the coupon collector as the a distinct counter. In the basic coupon collector problem [13], the input is a set of coupons P such that each coupon c_i is issued with probability p_i. At each time, one of the coupons is selected at random (with replacement). The coupon collector problem for a given n is defined as how many coupons need to be drawn in expectation until a collection of n different coupons have been drawn.

Let's look at a simple example. There are five major airlines in the United States, and Jane Doe files one of them uniformly at random every time she travels. In expectation, she needs to fly 11.42 times to experience all five airlines. The analysis is as follow: her first flight is always a new airline, while the second flight has a $\frac{4}{5}$ chance to be on another new airline—it takes $\frac{5}{4}$ flights in expectation to experience a second new airline (collect the second coupon). Similarly, she needs to travel $\frac{5}{3}$ times to experience a third airline, and so on; in total it takes $\sum_{i=1}^{5} \frac{5}{i} = 11.42$ travels (trials) to try all airlines, i.e., collect all five coupons.

For the most simple case of collecting n out of m coupons, where all have the same probability $p = \frac{1}{m}$, we can analyze the coupon collector problem as follows:

- With j coupons already collected, the probability that the next coupon is a new, unseen coupon (out of the $m - j$ uncollected ones) is $\frac{m-j}{m}$.
- Thus, the number of new coupons to be drawn until receiving a new unseen coupon is a geometric random variable $Geo(\frac{m-j}{m})$ with expectation $\frac{m}{m-j}$.
- We need to collect n new coupons, hence the total coupons that need to be drawn is $\sum_{j=0}^{n-1} Geo(\frac{m-j}{m})$, and we need to draw in expectation $\sum_{j=0}^{n-1} \frac{m}{m-j}$ coupons.

To count distinct elements seen within an attribute space, instead of drawing coupons at random, we can map random subsets of attributes to each coupon. Assume we are given a sequence of elements from attribute set $\mathcal{S} = \{attr_q(i),\ \forall i\}$. By randomly choosing m disjoint, equal-sized subsets $S_1, S_2, \ldots, S_m \subset \mathcal{S}$ and mapping them to m coupons, we essentially draw a new coupon with probability $p = \frac{|S_j|}{\mathcal{S}}$ every time we see a new element from $\mathcal{S}$, while seeing repeated elements does not lead to new coupons being collected. Hence, if we again require collecting n out of m coupons, we can analyze the expected number of distinct elements to be seen as follows:

- Given $|\mathcal{S}|$ is large, with j coupons already collected, the probability that the next distinct item in $\mathcal{S}$ leads to a different coupon is $p(m-j) = \frac{(m-j)|S_j|}{|\mathcal{S}|}$.
- Hence, the number of distinct items needed before we collect another coupon is a geometric random variable $Geo(p(m-j))$ with expectation $\frac{1}{p(m-j)}$.
- The total number of distinct items needed, until we collect n coupons, can thus be written as $\sum_{j=0}^{n-1} \frac{1}{p(m-j)}$.

We can adjust n, m, and p to tune the number of distinct items that need to be seen before the coupon collector is satisfied. In particular, increasing p or m leads to fewer distinct items needed, while increasing n leads to more.

3.2 Transform Query to Coupon Collector

Recall that, as defined in Section 2, an algorithm running query q takes input packet stream $i_1, i_2, \ldots$ and projects each packet into a key $key_q(i)$ and an attribute $attr_q(i)$; this query should output an alert k when it has seen enough packets with $key_q(i) = k$ with more than T_q distinct $attr_q(i)$.

To implement query q under the memory access constraint, we can use a random hash function with a coupon collector. Assume we have a memory model with w-bit words, and we are allowed $\gamma \leq 1$ memory accesses per packet. We can pack all coupons as bits in one memory word, and read/write all of them at once. For example, if w=32 bits, we can pack and collect at most 32 coupons.

We illustrate this process in Figure 1. Say we use $m = 4$ coupons to implement our query, each activates with a certain probability $p = 1/8$, and requires all of them to be collected. Note that in this case $m \cdot p = 1/2 < 1$, which means in expectation half of the packets will have no coupon. Packet 1 maps to key C, and activates the first coupon. We locate

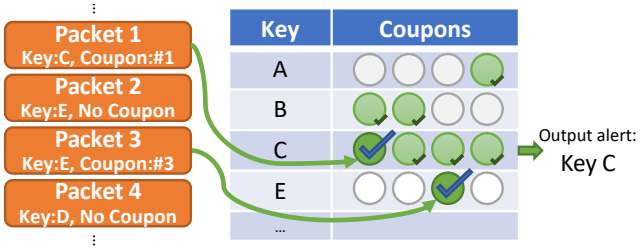

Figure 1: Queries are expressed as collecting coupons. When enough coupons are collected, the algorithm outputs an alert.

the memory location $\mathcal{M}[C]$ and mark its first bit to 1. At this point, all four bits in $\mathcal{M}[C]$ are set to 1, thus the coupon collection is finished and we output an alert C. Packet 2 does not activate any coupon. Packet 3 activates the third coupon, and has key E. Since it is the *first* coupon collected for key E, we assign a new memory location $\mathcal{M}[E]$ and mark its third bit to 1; we do not output anything after processing packet 3, as $\mathcal{M}[E]$ does not have all bits set to 1 yet. Since packets with key D have not collected any coupon yet, the memory does not contain a bitmap for key D.

In this example, checking whether a memory location has collected all the coupons is very straightforward, as we can simply compare the memory word with m binary bits $111\ldots1$. We call the special case *full coupon collection*, i.e., $n = m$. This is somewhat easier to implement on practical hardware (via a simple comparison). We call the $n < m$ case *partial coupon collection*, which requires counting 1 bits in a memory word on the hardware to check if there are n coupons collected. As we show later in the evaluation in Section 6, partial coupon collection significantly improves accuracy, at the cost of using more coupons (hence, using more hardware resources).

In summary, the algorithm $\mathcal{A}^q$ executes as follow:

- Given the distinct counting threshold T_q, we first find a combination of m, n, and p such that, it takes approximately T_q distinct attributes in expectation to collect n out of m coupons, where each attribute maps to each coupon with probability p.
- We use a random hash function $h(attr_q(i)) \rightarrow (0, 1]$ to map random subsets of attributes to a coupon. In particular, we say coupon j is activated by input packet i if $h(attr_q(i)) \in (j \cdot p, (j+1) \cdot p]$.
- For each input packet i, we compute $key_q(i)$ and $attr_q(i)$. If the hash value indicates coupon j is activated, i.e.,

$$\exists 0 \leq j < m, h(attr_q(i)) \in (j \cdot p, (j+1) \cdot p],$$

  we collect this coupon, by setting j-th bit of memory word $\mathcal{M}[key_q(i)]$ to 1. We also check if the number of 1 bits has exceeded n, and if so, output an alert.
- In practice, we have a memory array with bounded size using an integer index. We can use another hash function to map $key_q(i)$ to an index, as well as use a part of memory word length w to store the key itself in the table to detect hash collisions. This effectively makes the memory array a hash table, and hash collisions are rare as long as the array is sufficiently large.

Note that algorithm $\mathcal{A}^q$ accesses memory with probability $m \cdot p$, hence we require $m \cdot p \leq \gamma$ memory accesses per packet. Also, in the worst case the algorithm requires $\gamma \cdot N$ memory size, linear to the stream size (when all packets have different keys), although real-world traffic exhibits strong locality (different packets often share the same key) and the algorithm uses $o(N)$ memory empirically.

3.3 Adjusting Coupon Probabilities

To find the right coupon collector for a given query, we need to find n, m, and p which satisfy a few constraints: memory word size $n \leq m \leq w$, memory access constraint $m \cdot p \leq \gamma$, and activation threshold $\sum_{j=0}^{n-1} \frac{m}{p(m-j)} \approx T_q$. Out of all of the allowed combinations, we should choose the one providing the highest accuracy (minimized average relative error). One of our main challenges is to find these parameters according to the query threshold provided, such that we achieve better accuracy both analytically and empirically.

Let's look at a simple example. Suppose a query searches for all source IPs that send packets to at least 1024 destination IPs. In this case, we have $key_q(i) = \{i.srcIP\}$, $attr_q(i) = \{i.dstIP\}$, and, most importantly, $T_q = 1024$. Intuitively, if the condition is formulated using a single coupon ($n = m = 1$), BeauCoup sets the probability to collect this coupon to be $p = 1/1024$. The algorithm first selects a random hash function h. For each packet i traversing through the switch, we "collect" the single coupon if $h(i.dstIP) \in [0, 1/1024)$. For any particular key k, the stream of packets satisfying $i.srcIP = k$ and having distinct $i.dstIP$ will form a sequence of Bernoulli trials; the required number of trials is geometrically distributed and therefore, in expectation we see $\frac{1}{p} = 1024$ distinct $i.dstIP$ before the single coupon is collected. The total memory access probability in this case is exactly $m \cdot p = 1/1024$, and thus compatible with a memory access constraint $\gamma = 0.001$.

As a slightly more complex example, if the memory access requirement is loosened to $\gamma = 0.2$, we can use an alternative scheme: to collect $n = 13$ coupons out of $m = 20$ in total, and $p = 1/1024$. The algorithm still first selects a random hash function h, and for each incoming packet i, checks if $h(i.dstIP) \leq p \cdot m$. If that's the case, we find out the activated coupon $\lfloor \frac{h(i.dstIP)}{p} \rfloor$; for example, $h(i.dstIP) = 0.01$, so we activated the 10-th coupon. We then read the memory word stored at address $i.srcIP$, set the 10-th bit to 1, and check if there are already 13 bits set to 1 before writing it back.

For any particular $i.srcIP$, the required number of distinct $i.dstIP$ to observe until collecting the $(j+1)$-th coupon (with j coupons already collected) is geometrically distributed with $Geo(p(m-j))$, hence the total expected distinct items to observe is $E[\mathcal{T}(\mathcal{A}^q)] = \sum_{j=0}^{n-1} \frac{1}{p(m-j)} = 1029$ before $n = 13$ coupons are activated. This setup requires on average $p \cdot m = 0.0196$ memory access per packet. Although the expected activation time $E[\mathcal{T}(\mathcal{A}^q)]$ deviates from our goal T_q by 0.5%, this setup leads to a much lower average relative error thanks to using more coupons. (In fact, it is BeauCoup's optimal strategy under $\gamma = 0.2$ and $T_q = 1024$, with minimal average relative error.)

As a rule of thumb, we should use as many coupons as possible, as more coupons lead to lower relative error. Meanwhile, for a given T_q, using more coupons both leads to a smaller per-coupon probability p_q and larger total memory access $m_q \cdot p_q$. Thus, for a given query q with threshold T_q and memory access constraint γ_q, we need to find a small p_q and large n_q, m_q to maximize accuracy, subject to $p_q \cdot m_q \leq \gamma_q$ and $E[\mathcal{T}(\mathcal{A}^q)] \approx T_q$.

Note that our distinct counters operate at a regime where γ is extremely small (e.g., $\gamma = 0.01$), and each packet could have different key $key_q(i)$. This differentiates our approach from other algorithms, such as UnivMon [20] which supports only one query key, or HyperLogLog [12] which uses $\gamma = 1$ memory access per item.

4 MULTIPLE CONCURRENT QUERIES

Section 3 showed how to support a single query under a memory access constraint. In this section, we show how to execute multiple queries together under a total memory access constraint, as well as how to activate coupons and perform coupon collection for multiple queries.

Given a set of queries Q, with their query parameters (n_q, m_q, p_q) which already satisfies individual memory access constraints γ_q, our goal now is to design an algorithm $\mathcal{A}^Q$ that runs all of these queries simultaneously, under global memory access constraints. While the average memory access constraint $\overline{\gamma}$ words per packet is straightforward to satisfy (as long as $\sum_{q \in Q} \gamma_q \leq \overline{\gamma}$), it is non-trivial to satisfy the *hard* constraint imposed by our hardware target, namely accessing at most $\Gamma = 1$ word per packet.

In this section, we discuss how we combine all queries by embedding their coupons into the output space of random hash functions, and activate at most one coupon globally across all queries.

4.1 Group Queries with the Same Attribute

Our first step is to group different queries together based on the attribute they are counting. Different queries counting the same attribute use random hash functions over that attribute to activate their coupons. If we use the same hash function for these queries, we can allocate different output ranges to each query, such that at most one coupon will be activated among them. Therefore, one random hash function per attribute is sufficient for all queries.

As an example, say queries q_1 and q_2 both use $attr_{q_1}(i) = attr_{q_2}(i) = \{i.dstIP\}$. q_1 uses $m_{q_1} = 2$ coupons each with probability $p_{q_1} = 1/8$, while q_2 uses $m_{q_2} = 2$ coupons each with probability $p_{q_2} = 1/64$. We then define a single random hash function $h^{\{i.dstIP\}}$ over the attribute, and partition its range $(0, 1]$ for the two queries: $(0, 1/8]$ for coupon 1 of q_1, $(1/8, 2/8]$ for coupon 2 of q_1; $(2/8, 2/8+1/64]$ for coupon 1 of q_2, and $(2/8 + 1/64, 2/8 + 2/64]$ for coupon 2 of q_2. For other output values, no coupon is associated. More queries using the same attributes are stacked accordingly. Note that we would never run out of the $(0, 1]$ range because total memory access across all queries $(\sum_q m_q \cdot p_q)$ is smaller than $\overline{\gamma} \leq 1$.

4.2 Select at Most One Coupon to Update

After grouping queries by their attributes, we define one random hash function for each of the attributes, such as $h^{\{i.srcIP\}}$, $h^{\{i.dstPort\}}$, $h^{\{i.timestamp\}}$, etc. When a packet arrives, we compute all these random hash functions to find out if any hash function's output value is associated with a coupon. If there is exactly one coupon activated, we obtain its query q, coupon ID, as well as the query key associated with this packet $key_q(i)$. Subsequently, we can collect this new coupon into memory location $\mathcal{M}[(q, key_q(i)]$. We note that different queries may share $\mathcal{M}$ by writing to different memory indexes. However, if there is more than one coupons activated, we need to perform tie-breaking. BeauCoup only tie-breaks if there are exactly two coupons, by tossing a coin and allowing each coupon to succeed with 50% probability. We discuss technical details of how to implement the coin toss in Section 5.3. When there are more than two coupons, we discard all of them; this has limited effect on our system's accuracy, as we can prove in Appendix C that the probability of having ≥ 3 coupons activated is merely a few percent.

5 IMPLEMENTING BEAUCOUP ON PISA

High-speed programmable switches can run measurement algorithms at line rate, offering unprecedented visibility to network operators. However, switch data planes impose a constraint on memory accesses per packet, due to the fundamental mismatch between link speed and memory throughput In this section, we give an overview of the PISA programmable switch and its other constraints beside limited memory access. We discuss the needed modifications to BeauCoup, so that it may run within these constraints.

5.1 Programmable Switch Architecture

A PISA (Protocol Independent Switch Architecture) switch [3] implements a series of match-action tables as a pipeline to achieve programmable packet processing. At each pipeline stage, the switch can *match* on a packet's header data or metadata using bit patterns defined by the switch controller, and execute the corresponding *actions* associated with the matched rule. The actions can be arithmetic computations over the packet's header data or metadata, or it can be a read or write to on-board register memory. The register memory is arranged with a fixed word length—a maximum of 32 bits in current hardware. We can only perform $O(1)$ memory reads/writes per packet, and these accesses must be shared among various switch functionality. We also have a limited memory size, but as we discussed earlier, onboard SRAM is getting cheaper and is less of a bottleneck in modern systems, while the mismatch between link speed rate and memory throughput leads to a memory access constraint.

The lookup table entries support both exact matching via SRAM (Static Random Access Memory) and ternary matching via TCAM (Ternary Content Addressable Memory), and are originally motivated by matching network routing rules based on IP address prefix. Our hardware target also supports computing hash functions over header data as a possible action; this facilitates our use of random hash functions to implement probabilistic coupon collection. Although hardware targets impose constraints on the number of match rules and number of hash functions we can compute, these constraints are not bottlenecks in our system implementation.

For the purpose of this paper, we abstract all other hardware limitations away and assume the hardware target supports infinite memory with word size $w = 32$ bits, and allows $\Gamma = 1$ memory access per packet for monitoring purpose. An algorithm can use arbitrarily many exact and ternary match rules when implementing the monitoring queries.

Upon deciding to output an alert, the algorithm can trigger a special action to send a packet to the switch's controller port, or to a special output port that is linked to a monitoring server. The controller or server can then process the query and key, and act accordingly.

5.2 Finding Query Parameters

To run a set of queries Q on a PISA switch, we need to first find parameters for each individual query $q \in Q$, namely the total number of coupons m_q, number of coupons to collect n_q, and per-coupon probability p_q.

On practical systems such as PISA switches, random hash functions output binary bits. We therefore need to match on these bits to implement probability p_q. In this paper, we use the most straightforward form of probability $p_q = 2^{-b}$ by exactly matching b random bits, in order to simplify matching different coupons with a single hash function. For probabilities not in 2^{-b} form, [25] discussed how to approximate them using ternary match rules on random binary bits.

We implement a straightforward design to split the global memory access constraint $\overline{\gamma}$ among all queries: the *strictly fair* policy, limits the memory access of every individual query to $\gamma_q = \frac{\overline{\gamma}}{|Q|}$. However, we note that a query with large threshold T_q may not need all of its fair share, and a more optimized policy may reallocate a portion of the memory accesses to queries with smaller thresholds.

Now, given a query's average memory access constraint γ_q, the next step is to find parameters (n_q, m_q, p_q), that minimize average relative error while satisfying the memory access constraint $m_q \cdot p_q \leq \gamma_q$. We use the following heuristic algorithm to find the optimal parameters for query q:

(1) Iterate through progressively smaller power of two coupon probabilities ($p_q = 1/2, 1/4, \ldots$). For the current trial, say probability is fixed at $p_q = \frac{1}{2^b}$.
(2) Compute the constraint on the total number of coupons $\overline{m} = min(w, \lfloor \frac{\gamma_q}{p_q} \rfloor)$, based on memory access constraint and word length $w = 32$ bits.
(3) Try all combinations of $1 \leq n_q \leq m_q \leq \overline{m}$, and find the combination with expected output $E[\mathcal{T}(\mathcal{A}^q)] = \sum_{j=0}^{n-1} \frac{1}{p(m-j)}$ closest to the required threshold T_q. (We restrict $n_q = m_q$ if we must use full coupon collection.)
(4) Denote the closest combination (n_q, m_q, p_q) as currently optimal, if its relative bias $|E[\mathcal{T}(\mathcal{A}^q)] - T_q|/T_q$ is either better than the previous optimal combination, or is already smaller than a predetermined tolerance 1%. We choose 1% here because among all possible query parameters, there's approximately a 1% gap between consecutive thresholds. Hence, a narrower tolerance will prohibit all but one set of parameters to be used. Also, average relative error due to natural random variance in the process is much larger than 1%.

We terminate the process when p_q is so small that the algorithm waits for longer than T_q for the first coupon, i.e., when $w \cdot p_q < \frac{1}{T_q}$. The currently optimal combination (n_q, m_q, p_q) is then saved as q's query parameters. We repeat this process for all queries $q \in Q$.

5.3 Using TCAM to Activate Coupons

Ternary Content-Addressable Memory (TCAM) is capable of simultaneously matching a binary string (e.g., `10010`) with many ternary matching rules (e.g., `1***0`). Originally designed for forwarding network packets to their desired destinations, TCAM is also useful for us to probabilistically activate coupons and run our coupon collector scheme.

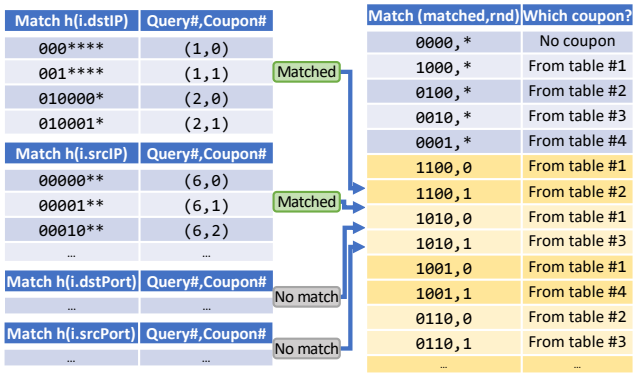

Figure 2: Using TCAM rules to activate coupons.

Recall that our system compiles a query q into a series of coupons, and activates them each independently and probabilistically when witnessing unique $attr_q(i)$ values. We can compute a random hash function $h(attr_q(i))$, and match on its bits to get the desired probability. When enough coupons for a single query and key are collected, we say this coupon collector finished collection and outputs an alert.

As described in Section 4.1, all queries using the same attribute are grouped together and use a single random hash function. We generate the output TCAM match rules to map the distinct output ranges of this hash function to individual coupons of each query, and since the match rules are disjoint, at most one coupon will be activated. Using the example in Section 4.1 again, two $p = 1/8$ coupons and two $p = 1/64$ coupons can be implemented as bit matching rule `000***`, `001***`, `010000`, and `010001`, respectively, over a 6-bit output of random hash function. This process is repeated for all attributes, to generate multiple TCAM match tables.

Still, there may be more than one coupon activated across all the hash functions we ran. We use another TCAM lookup table to select the activated coupon if there's exactly one, and perform tie-breaking when more than one is activated. In our implementation, we match on one extra random bit to break tie a fairly when there are exactly two coupons activated, and discard all coupons if there are more than two. The entire process, including the example from Section 4.1, is illustrated in Figure 2.

Each activated coupon carries metadata about query q, including its threshold T_q, key projection key_q, and how many coupons to collect in total. We can subsequently read the coupon bitmap stored at memory location $\mathcal{M}[(q, key_q(i))]$, add one coupon by setting a particular bit to 1, and check if we collected sufficient coupons by counting if there's already n_q bits set to 1.

To implement BeauCoup on PISA hardware, we allocate a sufficiently large hash-indexed register memory array as $\mathcal{M}$, which maps $(q, key_q(i))$ to an integer index using another random hash function. We can also record the flow keys into the register array, and detect hash collisions (which has very small probability when the array is sufficiently large). We ignore those coupons upon hash collision.

5.4 Life of a Packet

To summarize, the switch processes packet i as follows:

(1) We compute hash functions over all different query attributes $h(attr_q(i))$, for all $attr_q$.
(2) We use TCAM to match on each random hash function's output, and the TCAM rules may suggest some coupons are activated. With high probability, we will get ≤ 1 coupons; we use TCAM again to tie-break if there are two coupons. We ignore the unlikely case where three or more coupons are activated simultaneously.
(3) Given coupon j of query q is activated, we read memory at location $\mathcal{M}[(q, key_q(i))]$, which stores a bitmap of m_q coupons. We collect the current coupon j by setting j-th bit to 1 in the bitmap, then check if we have collected n_q coupons; if so, we output $(q, key_q(i))$.
(4) Note that future packets may again add coupon to this already saturated coupon bitmap, and repeat the same alert for all coupons collected. We can use other established mechanisms, such as a bloom filter, to de-duplicate the alert and not send $(q, key_q(i))$ repeatedly when more coupons get activated. Alternatively, the alert receiver can send a control message to the programmable switch to clear up the bitmap at memory location $\mathcal{M}[(q, key_q(i))]$.

6 EVALUATION

In this section, we use traffic traces collected from an Internet backbone switch to evaluate BeauCoup's accuracy and resource utilization.

We use the CAIDA Anonymized Internet Traces Dataset 2018 [4], collected over 5 minutes on March 15th 2018 13:00, which includes 135 Million packets in total. There are 3.0 Million unique IP pairs (with 0.8 Million unique source IPs and 0.7 Million unique destination IPs), and 8.6 Million unique flows (defined as the 5-tuple of IP source/destination, protocol, and source/destination ports).

We evaluate the algorithm based on average relative error—whether the alert was sent just in time, or too early or too late. It is either averaged over all alerts belonging to a single query, or all alerts across all queries.

While our algorithm can be implemented on PISA hardware, we use a Python-based experimental implementation to collect ground-truth and analyze statistics such as average relative error. We repeat each experiment 100 times with different random hash functions to observe the variance of our algorithm's accuracy. We also evaluated how much hardware resources our algorithm used, including TCAM match rules and actual memory access per packet. Note that algorithms

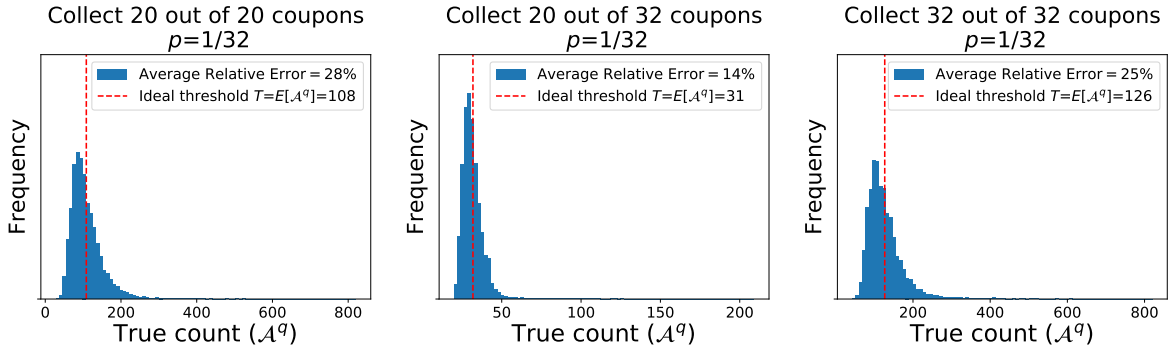

Figure 3: Using partial coupon collection (stop when $n = 20$ out of $m = 32$ coupons are collected) leads to narrower distribution of how many distinct items are collected, hence a lower average relative error.

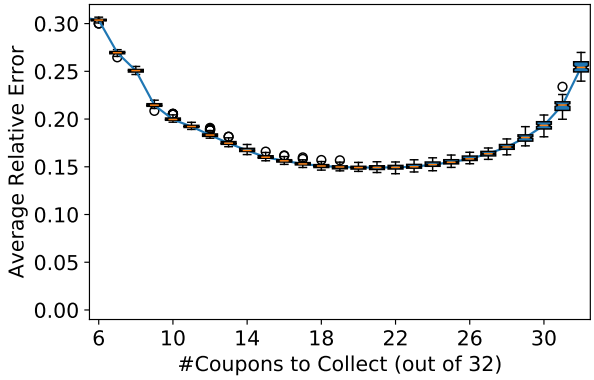

Figure 4: For partial coupon collection with fixed $m = 32$ total coupons, collecting $n = 20 \sim 22$ coupons leads to the lowest error.

run on PISA programmable switches can always run at full line rate (e.g., 100Gbps per port), as long as we do not run out of the switch's hardware resource constraint.

6.1 Comparing Partial/Full Collection

We start by evaluating the coupon collector for a single query—the super-spreader query that identifies source IP addresses that send traffic to more than T_q distinct destination IP addresses (i.e., $key_q(i) = i.srcIP$ and $attr_q(i) = i.dstIP$). The experiment analyzes BeauCoup's empirical behavior answering a single query, and also compares the full coupon collection approach with partial coupon collection.

In this experiment, we first fix each coupon for activation probability $p = 1/32$ for a new unique $i.dstIP$, or equivalently matching for a particular output value of a 5-bit random hash function. Say we define $m = 32$ such coupons (all values of the 5-bit hash), and require all of them be collected ($n = m = 32$). In expectation, we need to observe $\sum_{i=1}^{32} Geo(32/i) \approx 127$ distinct IP addresses, hence this configuration is useful for a threshold near 127. We can adjust the threshold by changing the number of coupons we collect; for example, if we define only $m = 20$ coupons each with $p = 1/32$ and wait for all of them be collected, we need to observe in expectation about 108 distinct addresses.

The expectation of the number of distinct items seen before the coupon collector finishes collecting, or $E[\mathcal{T}(\mathcal{A}^q)]$, can be easily adjusted by changing the number of coupons or changing per-coupon probability. However, our goal is to lower the variance of how many distinct addresses are seen before an alert is sent ($\mathcal{T}(\mathcal{A}^q)$), which directly impacts average relative error. In Figure 3 we plot the probability density function of $\mathcal{T}(\mathcal{A}^q)$, which is equivalent to the sum of PDF of a series of geometric variable. As we can see, the distribution is slightly left-skewed.

The benefit of using partial coupon collection is immediately demonstrated in Figure 3. When we only collect $n = 20$ out of $m = 32$ coupons defined before sending an alert, the distribution is much narrower, with average relative error 14%, compared with the 25% when using $n = m = 32$ coupons or 28% when using $n = m = 20$ coupons.

We further analyze the effect of only partially collecting coupons in Figure 4. Here we always use $m = 32$ coupons in total, and different algorithms wait until they have collected $n = 6$ to 32 coupons before sending an alert. In each case we have different expected number of distinct element to be seen, and we define this value as threshold T_q to calculate the average relative error under the ideal case. We can see that using $20 \sim 22$ out of 32 coupons gives us the lowest average relative error, i.e., the most narrow distribution.

Therefore, empirically, optimal query parameters shall define as many coupons per query as possible, and set n_q to be approximately $\frac{2}{3}m_q$. This produces the lowest average relative error.

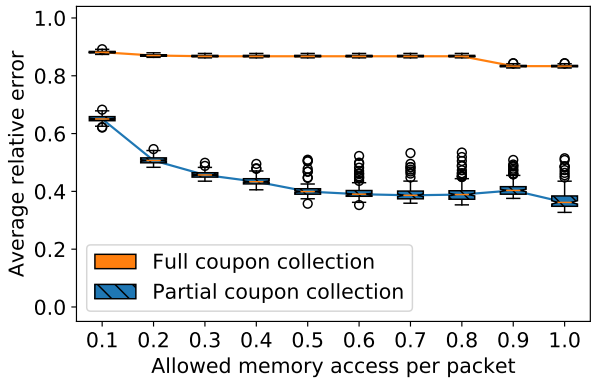

Figure 5: Partial coupon collection is significantly more accurate than full coupon collection, and its error decreases faster when given more memory access.

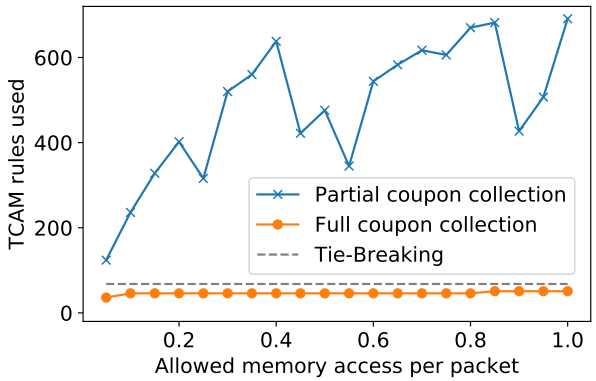

Figure 6: TCAM match rules used by full and partial coupon collection, as well as tie-breaking between multiple activated coupons.

6.2 Adjusting Memory Access Constraint

Now we run BeauCoup with multiple queries and observe its average relative error under varying per-packet memory access constraint.

We wrote $|Q| = 26$ distinct queries that resemble various monitoring demands a network administrator may have, choosing keys and attributes from timestamp, IP addresses, and ports, etc. The threshold ranges from 100 to 10000, and is selected based on the likely use cases of the particular query. In each experiment, we vary the average memory access constraint $\overline{\gamma}$ from 0.1 to 1 word per packet, and the BeauCoup compiler computes the optimal query parameters n_q, m_q, and p_q for each query q using memory access constraint $\gamma_q = \frac{\overline{\gamma}}{|Q|}$.

Once query parameters have been set, we convert all the queries into TCAM match rules over random hash functions, and run the algorithm (by executing these TCAM matches) over the input packets. The resulting algorithm computes 11 random hash functions per packet for different random attributes, and uses less than 1000 TCAM match rules for coupon activation and tie breaking, both are well under the hardware switch's capacity.

6.2.1 Overall accuracy. In Figure 5, we compare BeauCoup's accuracy between using only full coupon collection (requiring $n_q = m_q$) versus allowing partial coupon collection. The average relative error is much lower when we allow partial collection. This requires slightly more complicated arithmetic with no significant performance hit (e.g., Intel CPUs have special instruction supporting counting binary ones in memory words, and PISA switches can use a few TCAM table lookups to count segment-by-segment). Thus, we should use partial collection whenever the hardware target permits.

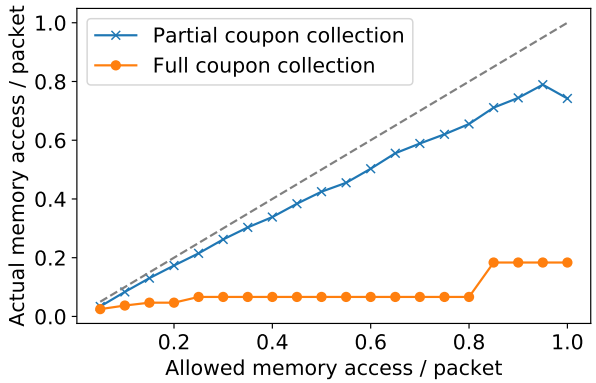

Figure 7: Actual memory access per packet, compared with the allowed average memory access per packet ($\overline{\gamma}$) specified as input.

Naturally, when given more memory access, both full-collection and partial-collection variants of BeauCoup improved their average relative error.

6.2.2 Resource utilization. We now analyze the hardware resource utilization of the two variants of the coupon collection algorithm. First, in Figure 6 we plot the number of TCAM matching rules used for activating coupons (match on random hash functions) and for tie breaking. Since there are 11 hash functions for our particular set of queries, we need $11 + \binom{11}{2}$ TCAM rules for tie-breaking (plus two special rules for zero or too many coupons). Full coupon collection uses much fewer coupons, hence much fewer TCAM match rules than partial coupon collection. This is due to a natural property of a coupon collector—the last coupon is the hardest to collect, hence smaller numbers of coupons are sufficient for a very large threshold. Nevertheless, partial coupon collection uses about 600 TCAM rules, or 72% of the theoretical maximum ($|Q| \cdot w = 26 \cdot 32 = 832$).

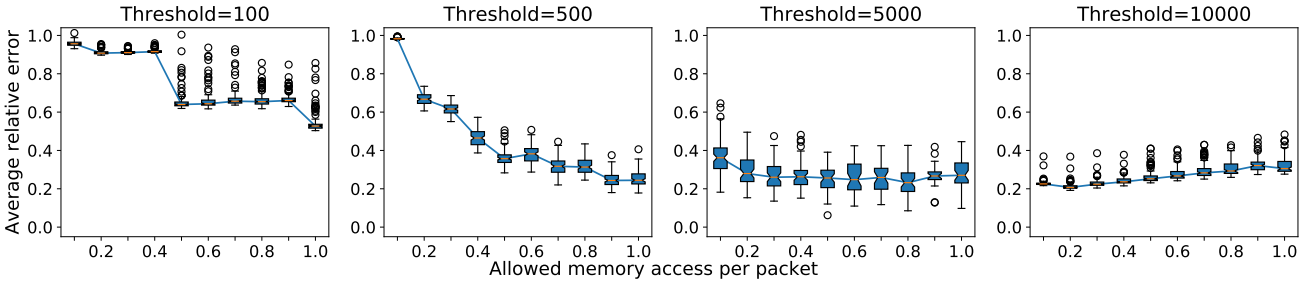

Figure 8: Query with the lowest threshold experiences the most significant accuracy improvement when allowing more memory access per packet.

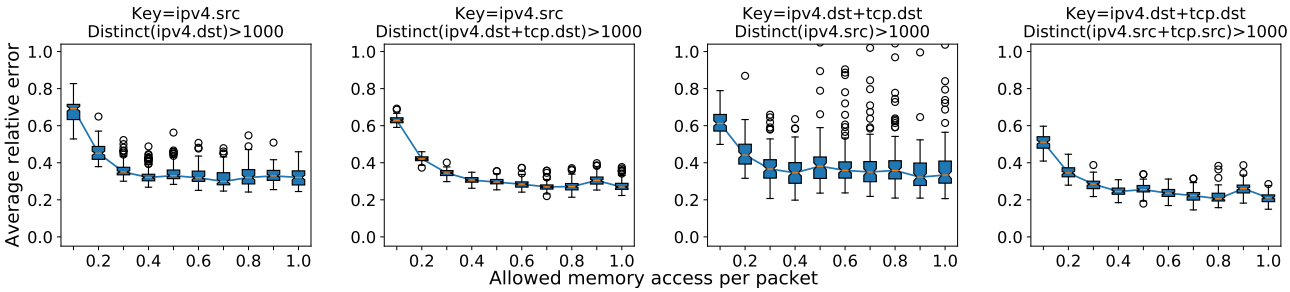

Figure 9: Queries with the same threshold exhibits similar accuracy improvement trend when given more allowed memory access, despite different key and attribute definitions.

Meanwhile, we also plot the actual memory access per packet when running our algorithm, compared with the allowed memory access $\overline{\gamma}$. As shown in Figure 7, if partial coupon collection is allowed, the algorithm uses almost all the memory access quota efficiently, hence the usage closely tracks the $y = x$ line. In contrast, when we must use full coupon collection, most queries do not fully use their allocated memory access. We conclude that partial coupon collection, when supported by hardware target, can more effectively use the allowed memory access.

We also analyzed the empirical memory size requirement for our algorithm when running on the CAIDA trace. We analyze the memory size needed M, i.e., the number of query keys with at least one coupon collected, with respect to the length N of the stream, and found that we indeed use sublinear memory size $M = o(N)$, implying that the real-world Internet traffic trace exhibits strong locality. In fact, we have approximately $M = \Theta(N^{0.75})$ when using partial coupon collection and $M = \Theta(N^{0.80})$ for full coupon collection.

6.2.3 Effect on individual queries. We also compare the effect of increasing memory access $\overline{\gamma}$ on individual query's average relative error. In Figure 8, we choose four different queries with various T_q from 100, 500, 5000, to 10000 and analyze their accuracy under partial coupon collection. Naturally, the query with the lowest threshold is the hardest to execute, as it requires high probability coupons and easily run out its memory access budget. Increasing $\overline{\gamma}$ allows it to increase accuracy significantly. For queries with larger T_q, the improvement is not as significant.

Notably, the query with $T_q = 10000$ already reached its optimal accuracy when $\overline{\gamma} = 0.2$, and its accuracy slightly deteriorates when we allow more memory accesses. This is due to having collisions with other queries when the system has more than one activated coupon and enters tie-breaking, which skews the activation probability of each coupon.

We also compare different queries with the same $T_q = 1000$ yet with different key_q and $attr_q$ definition. Here we use four queries as an example, the first one being super-spreader. As we can see from Figure 9, their average relative error has almost the same relation regarding memory access constraint $\overline{\gamma}$. The third plot in Figure 9 has a slightly higher variance, and is because this particular query produces fewer outputs in our experiment trace, hence there are more outliers for the average relative error statistics.

7 RELATED WORK

Streaming and memory model: In [21], Muthukrishnan surveyed several established streaming analysis models, and used an abstraction of maintaining one high-dimensional vector. Each incoming item will change one entry in the vector. The streaming models differ in the changes they can make to items in the vector: *cash register* is addition only, *turnstile* allows addition and subtraction, and *strict turnstile* allows addition and subtraction, yet requires the entries to be always non-negative. Subsequently, queries are made against

this high dimensional vector. Our paper uses the cash register model with unit addition, for an individual query and substreams of the input stream partitioned by the query key.

The cell probe model [18, 23, 30] is a limited memory access model often used to prove data structure lower bounds. In [30], Yao proved that $\lceil \log(n) \rceil$ probes (memory accesses) are necessary to check whether an item exists in a memory array of size n. Larsen et al. [18] discussed other similar lower bounds on how many memory accesses are necessary to solve a certain problem. Usually, in the cell probe model the algorithm is allowed to be adaptive, meaning that it can decide which memory address to look at next based on the content of memory it has already read earlier. We modified the cell probe model to allow at most Γ memory words to be accessed per packet, while introducing a new notion of sub-constant memory access, requiring the algorithm to make at most γ memory accesses per packet on average. This model is abstracted from our experience working with high-speed programmable switches, yet we can also identify similar situations in other computing architectures where low latency is required or a memory cache hierarchy exists.

In [24], Pontarelli et al. proposed a related model where a system has both faster on-chip memory and slower, larger off-chip memory, and can only perform a limited number of off-chip memory accesses per packet. In [17], Kim et al. implemented a practical off-chip memory for PISA switches.

Network monitoring query: Marple [22] and Sonata [16] proposed query languages to allow operators express complex and composite queries over network traffic, and subsequently run the queries (or part of them) in the switch data plane. Sonata supports running multiple queries together, however it does not use approximation when answering queries, therefore will run out of hardware resources quickly when there are too many queries.

HashPipe [27] and PRECISION [2] answer an approximate counting query on a single key type, and report the flow key with the largest count (the heavy hitter flows). FlowRadar [19] uses XOR and counters to answer an exact counting query across multiple switches in the network. Although its update can be done in data plane, it requires offline decoding to recover the counters for individual flow keys.

UnivMon [20] proposed using a universal sketch to answers many different queries over a single flow key. For input length m with n different items, it maintains $O(log(n))$ different count-sketches, and requires $O(log(n))$ memory access per packet in the worst case. It is non-adaptive in that the memory updates are the same regardless of the query, however the sketch analysis is relatively complex and needs to be done outside of the network data plane. It is also possible to implement many sketches, such as Count-Sketch [5], Count-Min Sketch [10], HyperLogLog [12], etc., or more recent works like ElasticSketch [29], in the switch data plane to answer various types of queries over a single flow key.

Our work is unique in supporting both multiple flow keys and multiple queries (distinct counts over different attributes). It also runs within the switch data plane without the need for offline analysis. The distinct counter abstraction allows network operators to express many queries to observe a variety of anomalies, and running many approximated queries provides unique value alongside with existing systems that run a handful of exact queries.

Approximate count-distinct: [9] surveyed prior works on approximately counting the number of distinct elements, which can be roughly categorized into two flavors: K-Minimum-Value and Distinct Sampling. K-Minimum-Value [1] computes a random hash function over all input elements, and uses the k smallest values observed to infer how many distinct elements has been observed. Distinct Sampling [15] samples new distinct element at a small probability, and infer the count by the number of items sampled. We can sample an item out of 2^n distinct items, if we wait for n consecutive leading zeros in the output bits of a random hash function.

HyperLogLog [11, 12] builds upon the idea of Distinct Sampling but instead partitions the incoming stream into k sub-streams and use k independent estimators, and output the harmonic mean of their estimates. Each estimator records the longest consecutive leading zeros seen from the output of a random hash function.

Our implementation of coupon collector for distinct counting query can be viewed as a modified version of the HyperLogLog algorithm with only 1-bit counters, and the counters are only changed to 1 with probability p, which somewhat resembles Distinct Sampling. We incurred approximation error in this process, in exchange for limited memory access: all counters are packed into one memory word. This also enabled us to use sub-constant memory access, while all variants of HyperLogLog uses at least one memory access per new item.

8 CONCLUSION

We present BeauCoup, a streaming algorithm to simultaneously execute many distinct counting queries under sub-constant memory access model. BeauCoup enables network operators to run many network monitoring queries simultaneously over high-speed network traffic, under the strict memory access constraints of the PISA programmable network switches. Evaluation showed that BeauCoup can efficiently use all memory access allocated to the algorithm, uses a moderate amount of other hardware resources, and can achieve 20% to 40% average relative error on our test queries over a real-world internet backbone traffic trace.

REFERENCES

[1] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. 2002. Counting distinct elements in a data stream. In *RANDOM*. ACM.

[2] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *IEEE International Conference on Network Protocols*.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Conference*. 99–110.

[4] CAIDA. 2018. The CAIDA UCSD Anonymized Internet Traces 2018 - March 15th. (2018).

[5] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theor. Comput. Sci.* 312, 1 (2004), 3–15.

[6] Benoit Claise. 2004. Cisco Systems NetFlow Services Export Version 9. *RFC 3954* (2004).

[7] Edith Cohen. 1997. Size-Estimation Framework with Applications to Transitive Closure and Reachability. *Journal of Computing and System Sciences* 55 (1997), 441–453.

[8] Edith Cohen. 2015. All-Distances Sketches, Revisited: HIP estimators for massive graphs analysis. *IEEE Transactions on Knowledge and Data Engineering* 27 (2015). Issue 9.

[9] Graham Cormode. 2011. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers* (2011).

[10] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[11] Marianne Durand and Philippe Flajolet. 2003. Loglog counting of large cardinalities. In *European Symposium on Algorithms*. Springer, 605–617.

[12] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AOFA)*.

[13] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. 1992. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. *Discrete Applied Mathematics* 39, 3 (1992), 207–229.

[14] P. Flajolet and G. N. Martin. 1985. "Probabilistic Counting Algorithms for Data Base Applications". *Journal of Computing and System Sciences* 31 (1985), 182–209.

[15] Phillip B Gibbons. 2001. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, Vol. 1. 541–550.

[16] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*. 357–371.

[17] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic External Memory for Switch Data Planes. In *ACM Workshop on Hot Topics in Networks*. 1–7.

[18] Kasper Green Larsen, Jelani Nelson, and Huy L Nguyên. 2015. Time lower bounds for nonadaptive turnstile streaming algorithms. In *ACM Symposium on Theory of Computing*. ACM, 803–812.

[19] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*.

[20] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM*. 101–114.

[21] S. Muthukrishnan. 2005. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science* 1, 2 (2005).

[22] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*. 85–98.

[23] Mihai Patrascu. 2008. *Lower bound techniques for data structures*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[24] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. 2018. EMOMA: Exact Match in One Memory Access. *IEEE Trans. Knowl. Data Eng.* 30, 11 (2018), 2120–2133.

[25] Ori Rottenstreich, Yossi Kanizo, Haim Kaplan, and Jennifer Rexford. 2018. Accurate traffic splitting on commodity switches. In *ACM SPAA*.

[26] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. 2008. cSamp: A System for Network-Wide Flow Monitoring. In *USENIX NSDI*. 233–246.

[27] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *ACM SIGCOMM Symposium on SDN Research*. 164–176.

[28] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. In *Network and Distributed System Security Symposium*.

[29] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM*. 561–575.

[30] Andrew Chi-Chih Yao. 1978. Should Tables Be Sorted? (Extended Abstract). In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. 22–27.

[31] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*. 29–42.

Appendices

A REMARKS FOR SUB-CONSTANT MEMORY ACCESS MODEL

We argue that when sub-constant memory access becomes the primary constraint, algorithms cannot utilize larger memory size well. We demonstrate this notion in an extreme case: set word length to $w = 1$ under access constraint $\Gamma = 1$, i.e., the algorithm is restricted to access only one bit of information.

We first assume the memory size is also only one bit. To answer a distinct query with threshold T, the optimal strategy is to collect one coupon with probability $1/T$. The number of distinct attributes seen until this coupon is collected (this bit set to one) is a geometric variable $X = Geo(1/T)$ with expectation T.

Now let's see if an algorithm with N bits of memory can do better. Its strategy should also be to collect one coupon (write to one bit) with probability p_j, for every bit $1 \le j \le N$. Since the algorithm cannot look at two bits, it must output an alert based on the bit it is currently accessing. Thus, its behavior for writing into each bit is independently a geometric random variable $Geo(p_j)$ (with respect to the number of

distinct attributes seen). The algorithm's behavior is the minimum of N geometric variables, which is also a geometric variable:

$$Y = Geo\left(1 - \prod_{j=1}^{N}(1-p_j)\right)$$

Thus, the two algorithm will behave similarly in terms of output if we require $E[X] = E[Y] = T$. Note that if we set all p_js be the same, and $N \to \infty$, we have

$$\begin{aligned} 1 - \prod_{j=1}^{N}(1-p_j) &= 1/T \\ \Rightarrow e^{-Np} &= 1 - 1/T \\ \Rightarrow p &= -\frac{\log(1-1/T)}{N} \\ &\approx \frac{1}{NT} \end{aligned}$$

Some readers may argue that an algorithm can still perform two state transfers even with only one bit of memory, namely from 0 to 1 and from 1 to accept. Under such setup, the optimal algorithm with 1-bit memory size behaves as the Negative Binomial Distribution random variable (sum of two Geometric variable):

$$Pr[X = x] = \frac{4(1+x)(1-2/T)^x}{T^2},$$

while the algorithm with N bit memory behaves as the minimum of N i.i.d Negative Binomial Distribution random variables. This complicates the computation, yet empirically does not give much benefit either.

We conclude that under 1-bit memory access per packet constraint, we did not gain any asymptotic benefit by having larger memory size.

Now we assume we have multiple queries to compute with different keys, and still have a one bit memory access constraint. Obviously, different queries must use disjoint bits, and each should opt for memory access for only $\gamma_q = O\left(\frac{1}{T_q}\right)$ fractions of the input. Since we require $\gamma \geq \sum_{q\in Q}\gamma_q$, we need $\sum_q 1/T_q \leq O(\gamma)$.

Therefore, to achieve non-trivial accuracy, a query with threshold T should roughly cost $1/T$ memory accesses per packet. Hence, under 1-bit memory access constraint, we need a large query threshold T_q such that $\sum_{q\in Q} 1/T_q = O(1)$.

B USING WORD MEMORY AS DISTINCT COUNTERS

Currently, we use the w-bit word memory as a bitmap for the coupon collector, however there may be better ways to use the 2^w states available. We now analyze how far away our scheme is from the optimal.

Let us fix $w = 32$ bits and arbitrarily choose our goal as to count exactly $n = 5$ distinct elements (and report on the sixth element). In our current scheme, we effectively use a random hash function to map all input elements into 32 bins, then wait until at least five bins are nonempty. During this process, hash collisions may occur, Where we may map two different items to the same bin, thus effectively under-counting. (It is impossible to over-count.) The probability for under-counting to occur is exactly

$$1 - \frac{P_6^{32}}{32^6} = 39.2\%.$$

In the meantime, an ideal scheme using the 32-bit memory can split the entire 2^{32} state space to a few stages, remembering which elements have been seen. In particular, it may use $X > 32$ bins, and represent which bin is nonempty using $\binom{X}{1}$ states when there's one item, which two bins are nonempty using $\binom{X}{2}$ states, etc. We now require

$$\sum_{i=1}^{n}\binom{X}{i} \leq 2^w$$

For $w = 32$ and $n = 5$, we have $X \leq 220$. Thus, the ideal algorithm has a lower under-counting probability of

$$1 - \frac{P_6^{220}}{220^6} = 6.6\%.$$

Thus, we can somewhat say the $n = 5$, $m = 32$ coupon collection scheme is only using the states in w=32 bit with about 60% efficiency. This is adequate for our purpose of network monitoring queries, when a 40% relative error is tolerated in exchange for limited memory access. However, future works may improve on how to store states in the w-bit word, for example by using it as a bloom filter, to achieve closer to optimal accuracy.

C PROBABILITY OF COUPON COLLISIONS

Although the expected number of coupons activated per packet is bounded by $\overline{\gamma} \leq \Gamma \leq 1$, it is possible to have multiple coupons activated simultaneously, triggering a tie-break. We can bound the probability of tie-breaking events as follows:

Recall that coupons defined over the same attribute are all grouped together and use different output ranges of one random hash function, so they will never collide. Thus, collision happens across multiple hash functions. Now we analyze the probability for having multiple hash functions where each reports one coupon as activated.

We consider the system uses $H > 3$ random hash functions, each with activation probability $x_1, x_2, \ldots, x_H$, and we have $\sum x_i \leq \overline{\gamma} \leq 1$. Each random hash function will activate

coupons independently, hence the total number of activated coupons is the sum of H Bernoulli random variables.

In our current system implementation, we only perform tie-breaking when $C = 2$ and ignore all coupons when $C \geq 3$. The probability for having more than $C \geq 3$ coupons activated is maximized when all hash functions share the same probability, i.e., $x_i = \frac{\overline{\gamma}}{H}$. In this case, the number of coupons activated follows a binomial distribution $B(n = H, p = \frac{\overline{\gamma}}{H})$. Hence, plug in $\overline{\gamma} = 1$ (maximum allowed) and $H = 11$ (our example query set), and we have

$$\Pr\left[B(n = H, p = \frac{\overline{\gamma}}{H}) \geq 3\right] = 7.11\%$$

This is smaller than or on par with the optimal average relative error achieved by coupon collectors for distinct counting (about $15\% \sim 30\%$), and therefore not fundamental to BeauCoup's error. We further note that this probability grows very slowly with H, and is only 8.0% when $H = 10^4$.

Still, it creates a downward bias for individual coupon's activation probability; we leave the correction for this bias in query planning for future work.