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

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

ABSTRACT

Network administrators constantly monitor network traffic for congestion and attacks. They need to perform a large number of measurements on the traffic simultaneously, to detect different types of anomalies such as heavy hitters or super-spreaders. Existing techniques often focus on a single statistic (e.g., traffic volume) or traffic attribute (e.g., destination IP). However, performing numerous heterogeneous measurements within the constrained memory architecture of modern network devices poses significant challenges, due to the limited number of memory accesses allowed per packet. We propose BeauCoup, a system based on the *coupon collector problem*, that supports multiple distinct counting queries simultaneously while making only a small constant number of memory accesses per packet. We implement BeauCoup on PISA commodity programmable switches, satisfying the strict memory size and access constraints while using a moderate portion of other data-plane hardware resources. Evaluations show BeauCoup achieves the same accuracy as other sketch-based or sampling-based solutions using 4x fewer memory access.

CCS CONCEPTS

• **Networks** → **Data path algorithms**; **Network measurement**;

KEYWORDS

Streaming Algorithm, Sketching, Distinct Counting, Data Plane, Programmable Switch, Network Measurement

ACM Reference Format:
Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3387514.3405865

1 INTRODUCTION

Network operators constantly monitor network traffic to detect attacks, performance problems, and faulty equipment. To ensure that networks are functioning properly, network operators often need to monitor for *multiple* kinds of problems *simultaneously*, including worms, port scans, DDoS attacks, SYN floods, and heavy-hitter flows.

A variety of network-monitoring tasks can be modelled as counting the number of distinct attributes seen across a set of packets. As the simplest example, to detect a host that is spreading a worm we may look for a *super-spreader* [31], or a source IP that sends packets to many (e.g., 1000+) distinct destinations. However, there may be multiple hosts that are spreading worms, thus we need to identify *all* the source IPs sending traffic to many destinations. Furthermore, different tasks may define their keys differently: to identify victims of a DDoS attack, for example, we need to instead look for *destination* IPs that are receiving from many distinct *source* IPs. The diversity of monitoring tasks with different key definitions makes executing them simultaneously even more challenging.

Emerging programmable switches can analyze traffic directly in the data plane as packets stream by, making these devices well-suited for performing such telemetry tasks. However, the memory and processing resources of these switches are extremely limited. Traditionally, researchers have focused on the limited memory *space* in the data plane, designing compact data structures that can compute approximate answers for a single traffic-monitoring query [4, 17, 20, 23, 29, 31, 32], or multiple queries over the same key [22, 32].

Extending these solutions to support multiple queries over different keys would require instantiating multiple separate data structures. Having separate data structures would consume precious memory space in the data plane, but this is not the only problem. To maintain line rate, programmable switches only allow a small constant number of *memory accesses* per packet, making it infeasible to update multiple data structures for every packet.

Most existing techniques for handling *multiple* queries rely heavily on software running outside of the data plane, introducing communication overhead and latency. The simplest approach is to randomly sample packets in the data plane [3, 8], and have the software compute multiple statistics on the samples. While useful for detecting high-volume flows, random sampling significantly reduces the accuracy for queries that count the number of distinct attributes. To improve accuracy, several recent works collect information about all potentially relevant flows in the data plane, and have the software compute the statistics of interest [16, 20, 25]. However, these solutions introduce a tension between the volume of data exported from the data plane and the number and diversity of queries that can be answered with reasonable accuracy in real time.

Instead, we need new techniques that can handle numerous heterogeneous queries directly in the data plane, despite the limited memory space and memory access. We present **BeauCoup**, which supports a general query abstraction that counts the number of

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405865

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, when searching for worms, a packet's source IP is the key, its destination IP is the attribute, and the threshold decides how many distinct destination IPs are needed to flag a source IP as a worm sender. Our goal is to generate an alarm for those source IPs, approximately, within a reasonable error such as 20%-30% of the threshold. BeauCoup runs multiple queries simultaneously, under a strict per-packet memory access constraint. BeauCoup also allows users to define arbitrary packet-header field tuples as query keys and attributes, providing great expressiveness. The query set can be updated on the fly without the need to re-compile the data-plane program; re-compilation is required only when new header field tuples are defined.

The design of BeauCoup takes inspiration from the *coupon-collector problem* [14]. Using super-spreader detection as an example, suppose we want to know if a sender has sent packets to at least 130 different destination IP addresses. Instead of recording all destination IPs we see, we define 32 *coupons*, and map each destination IP to one of the 32 coupons uniformly at random. Now, for each packet from that sender, we extract the destination IP and *collect* its associated coupon. The coupon may be a duplicate (was already collected earlier), either because the same destination IP appears twice, or because two destination IPs map to the same coupon. We then wait until we have collected each of the 32 coupons at least once to flag the sender as a super-spreader.

The coupon-collector problem asks how many random draws (with replacement) are needed to collect all of the coupons, i.e., have every coupon drawn at least once. With 32 coupons, we need 129.9 draws in expectation. We therefore can use a 32-coupon collector to identify if a particular sender is sending to 130 (or more) distinct destination IPs. Answering a query with a different threshold (say, 1000 destination IPs) requires tuning the coupon collector's configuration, by changing the number of coupons (m), the probability (p) of drawing each coupon for a new destination IP, or the number of coupons that must be collected (n). Essentially, we are using a m-bit vector to estimate whether the number of distinct items seen has exceeded a threshold. A naive m-bit coupon collector is equivalent to either a HyperLogLog [13] register with m 1-bit hash functions, or a m-bit Bloom Filter [1] with only 1 hash function. We discuss the equivalence in more detail in Section 7.

The challenge in designing BeauCoup lies in applying the coupon-collection problem to *multiple queries*, each with *different keys and attributes* entirely in the data plane, under strict memory constraints. To limit memory size, BeauCoup must keep coupon state small, devote state to a key only when needed, and share memory across queries and keys. Furthermore, to limit the memory accesses when processing a packet, BeauCoup collects *at most one* coupon per packet. BeauCoup must ensure each query only draws a coupon with a small enough probability, and coordinate among different queries to avoid collecting many coupons concurrently. Thus, BeauCoup must tune the coupon-collector parameters (i.e., m, p, and n) carefully to *simultaneously* achieve accurate results for each query and ensure that the combination of queries does not violate the memory access constraint. Finally, we must implement each part of the BeauCoup algorithm using only the operations available in high-speed programmable switches.

Name	Key	Attribute	Threshold
Super-spreader	$srcIP$	$dstIP$	1000
DDoS victim	$dstIP$	$srcIP$	1000
Port scan	$\{srcIP, dstIP\}$	$dstPort$	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 count-distinct query definitions.

In designing and implementing BeauCoup, we make the following contributions:

- **Algorithm (§2)**: Data-plane algorithm for multiple count-distinct queries under memory size and access constraints.
- **Compiler (§3)**: Method for optimizing the accuracy of a set of queries subject to the memory constraints.
- **Prototype (§4)**: System that translates high-level queries into data-plane configuration that runs on a PISA hardware switch.

We evaluate our prototype in §5, discuss future work in §6, compare with related work in §7, and conclude in §8.

Ethics Statement: This work does not raise any ethical issues.

2 THE BEAUCOUP ALGORITHM

We now show the BeauCoup algorithm for network-monitoring queries. We first present a query model based on distinct counting, that supports a variety of network-monitoring tasks. Next, we discuss how to use coupon collectors to implement these queries. Finally, we discuss how to use coupon collectors to run multiple queries simultaneously, under a strict per-packet memory access constraint.

2.1 Query: Count-Distinct with Threshold

A wide variety of network-monitoring tasks can be characterized as a query q which (1) maps each packet i to a key $key_q(i)$, (2) counts the number of distinct attributes $attr_q(i)$ that appear for each key, and (3) applies a threshold T_q to the count to decide whether to report a key. That is, BeauCoup should output an alert (q, k) for query q and key k, when the packets in a time window W satisfy:

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

For the super-spreader example in the Introduction, the key is the packet's source IP, the attribute is the destination IP, and the threshold is 1000. For DDoS detection, we can instead use the destination IP as a packet's key, use the source IP as the attribute, and perhaps use a higher threshold like 10000.

In Table 1, we present more examples of common network-monitoring tasks under our query model. In particular, the special attribute $i.timestamp$ is unique 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 unique timestamps seen. Filtering operations can also be expressed in this query formulation, as shown in the SYN-flood example above—by mapping irrelevant packets to a fixed value, the distinct counting query effectively ignores them.

Notation	Definition
$key_q(\cdot)$	Key definition for query q
$attr_q(\cdot)$	Attribute definition for query q
T_q	Threshold for query q
W	Time window for answering queries
Γ	Maximum memory access per packet
c	Number of accesses for collecting one coupon
S	Memory size
m_q	Total number of coupons for query q
p_q	Probability of drawing a particular coupon
n_q	Number of different coupons to collect
γ_q	Average number of coupons activated per packet

Table 2: Notations used in the paper.

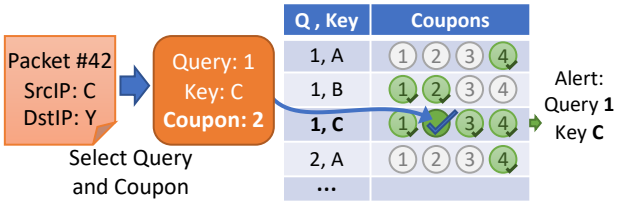

Figure 1: We collect coupons by updating bit vectors in an in-memory coupons table.

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 attribute.

Our goal is to build a system that simultaneously executes a set of queries $Q = \{q_1, q_2, \dots\}$ and outputs alerts (q_j, k_j), subject to the hardware constraints of a maximum memory size S and at most Γ *memory accesses* per packet. In the rest of this section, we discuss how BeauCoup achieves $\Gamma = O(1)$, i.e., answering multiple queries in the data plane using a small *constant* number of memory accesses per packet, independent of the number of queries.

2.2 Updating the Coupon-Collector Table

We maintain a table with bit vectors representing the coupon collectors, as shown in Figure 1. Upon collecting the first coupon for the query-key pair (q, k), BeauCoup creates a new table entry; when the bit vector indicates enough coupons have been collected, BeauCoup generates an alert for (q, k).

The example in Figure 1 uses 4-coupon collectors for all queries. When a packet arrives at the switch, BeauCoup first selects a query and a coupon. In this case, coupon #2 for query q_1 is selected, and we can extract the query key C from the packet, using the query's key definition. Now BeauCoup finds the coupon collector in the in-memory coupon table under row $(1, C)$, and collects the second coupon by marking the bit vector's second bit to 1. If there is no such row in the table, we allocate a new row and collect the single coupon. Since now all four coupons are collected at least once for row $(1, C)$, BeauCoup reports that key C satisfied query q_1. Other packets may collect coupons for other queries, or do not collect any coupon at all.

The coupon table shown in Figure 1 is designed to fit the hardware constraints of PISA programmable switches:

- **Compact rows:** Each row of the table stores one w-bit word as a bit vector, representing at most w coupons, where each bit represents whether a particular coupon has been collected at least once. (We also store two more words of auxiliary data per row, to record a timestamp and a checksum of the query key, which are used for detecting timeouts and hash collisions.)
- **Space efficiency:** We only maintain the bit vector for a query key when there's at least one coupon collected for that key. Therefore, although each query has many keys (e.g., 2^{32}), only a small fraction of *active keys* occupies memory. Different keys (such as keys A, B, and C for query q_1) and different queries (such as queries q_1 and q_2) effectively multiplex a shared memory space, and a new entry is created when a key collects its first coupon.
- **Limited access:** BeauCoup only needs to access the in-memory table when it needs to collect a coupon. When a packet does not produce any coupon for a query, we do not need to access memory. This effectively allows us to multiplex memory accesses across queries, by having different packets updating the table for different queries.

A coupon collector defines m coupons, a probability p for drawing each coupon in a random draw, and stops when there are at least n different coupons collected, i.e., each of these n coupons had been drawn at least once. Since BeauCoup uses a random (yet fixed) mapping from attributes to coupons, observing a new, unseen attribute is equivalent to randomly drawing a coupon. Seeing the same attribute more than once has no effect on the coupon collector, as it merely draws the same coupon again. With an appropriate combination of parameters (m, p, n), the coupon collector can be used to indicate if there are more than T_q distinct attributes seen, while automatically ignoring duplicate attributes.

2.3 Selecting a Query and a Coupon

We now discuss how we select one coupon for a given query, and how we coordinate between multiple queries.

Selecting one of m coupons. For every query q, with key definition key_q and attribute definition $attr_q$, BeauCoup applies a random hash function h on packet i's attribute $attr_q(i)$, where $h : \{attr_q(i), \forall i\} \to [0, 1)$, and checks if the output of the hash function falls into a range. For example, suppose query q uses four coupons ($m_q = 4$) and selects each coupon with probability $p_q = 1/8$. Then, BeauCoup would map all attributes satisfying $h(attr_q(i)) \in [0, 1/8)$ to coupon #1; similarly, coupons #2, #3, and #4 are associated with output ranges $[1/8, 2/8)$, $[2/8, 3/8)$, and $[3/8, 4/8)$, respectively. If the output of the hash function for packet i falls in $[0, 4/8)$, BeauCoup sets the bit for the associated coupon to 1 for that query-key pair, creating an entry in the table if needed.

If the output of the hash function falls in $[4/8, 1)$, BeauCoup does *not* need to access memory for this query on behalf of this packet, and it can use the memory access for other queries. We define γ_q as the *average* number of activated coupons allowed per packet for query q; with the random hash function h, the example query only activates $\gamma_q = m_q \cdot p_q = 1/2$ coupons per packet in expectation. A small $\gamma_q < 1$ has two main advantages. First, the coupon table does not need to maintain state for every active key. Instead, BeauCoup

only allocates memory for a query-key pair upon collecting the first coupon for that key. Second, a small γ_q allows multiple queries to run concurrently under a maximum memory access constraint $\Gamma = O(1)$. In particular, when a particular query q is not collecting a coupon, BeauCoup can devote the unused memory access "budget" to collect a coupon for another query, as we discuss next.

Each query q has its own limit γ_q on how many coupons to collect per packet. For simplicity, we assume a naive fair allocation that gives each query the same share of memory accesses. Given that collecting a coupon costs c memory accesses and a total memory access budget of Γ per packet, we limit each query to collect at most $\gamma_q = \Gamma/|c \cdot \mathcal{Q}|$ coupons per packet on average. Therefore, each query's coupon-collector configuration should satisfy $m_q \cdot p_q \leq \gamma_q$. However, a naive choice of hash functions could have a single packet need to collect a coupon for many different queries, even if the average rate of memory accesses is constant. To obey the strict per-packet memory access constraint Γ, BeauCoup coordinates the hash functions across the queries, first among all queries using the same attribute, and second across sets of queries using different attributes.

Grouping queries with the same attribute. Queries may have the same attribute definition (say, destination IP) but with different key definitions (say, source IP for query q_1, and source IP and source port tuple for query q_2). These queries can use the *same* hash function, applied to their common attribute, to draw their coupons. To guarantee that at most one query collects a coupon, BeauCoup divides the hash output across the queries. For example, suppose query q_1 uses $m_1 = 2$ coupons each with probability $p_1 = 1/4$, while query q_2 uses $m_2 = 2$ coupons each with probability $p_2 = 1/8$. We partition the range $[0, 1)$ of the hash output as follows: $[0, 1/4)$ for coupon #1 of q_1, $[1/4, 2/4)$ for coupon #2 of q_1, $[2/4, 2/4 + 1/8)$ for coupon #1 of q_2, and $[2/4 + 1/8, 2/4 + 2/8)$ for coupon #2 of q_2. Other output values are not associated with any coupon. We illustrate this example in Figure 2. We can stack additional queries using the same attribute accordingly. Note that we never run out of the $[0, 1)$ range, as long as the total memory accesses across all queries ($\sum_q m_q \cdot p_q$) is bounded by $\Gamma \leq c$, i.e., each packet collects at most one coupon.

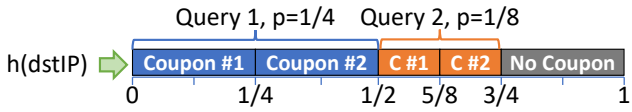

Figure 2: Different queries use disjoint ranges to map the random hash function's output to coupons.

Coordinating across queries with different attributes. To support queries with different attribute definitions, BeauCoup constructs one random hash function for each unique attribute (e.g., one hash function for destination IP, one for timestamp, and so on). When a packet arrives, BeauCoup computes all of these random hash functions to determine if any hash function's output value is associated with a coupon for some query. If only one hash function draws a coupon, BeauCoup collects the coupon for the associated query and key. However, if multiple coupons are drawn, we perform tie-breaking. Currently, BeauCoup only tie-breaks if exactly two hash functions draw coupons, by tossing a coin and allowing each coupon to succeed with 50% probability; we discuss how to implement the coin toss in Section 4.1. When more than two hash functions draw coupons, we do not collect *any* of them; this has little effect on accuracy, as we prove in Appendix B that the probability of drawing many coupons for one packet is very small.

With the coordination within and across hash functions, BeauCoup can now guarantee collecting *at most one* coupon per packet, without meaningfully impacting the accuracy of individual query's coupon collectors. Each individual query still collects coupons with the right probability, as if it is the only query running in the system. Given the strict memory access constraint, such coordination is what makes it possible to run many queries simultaneously while maintaining reasonable accuracy for all of them.

3 THE BEAUCOUP QUERY COMPILER

For each query q, BeauCoup computes three coupon-collector parameters: collect n_q out of m_q coupons, each with probability p_q. Taking the threshold T_q and the average per-packet coupon limit γ_q for all queries $q \in \mathcal{Q}$ as input, the BeauCoup compiler produces the configuration of $\{m_q, p_q, n_q\}$ that maximizes accuracy. A configuration satisfies the average per-packet coupon limit as long as $m_q \cdot p_q \leq \gamma_q$, which means a query produces at most γ_q coupons per packet in expectation. However, characterizing a coupon collector's accuracy for tracking the threshold T_q is less straightforward. We want the *number of random draws* needed until the coupon collector collects enough coupons to both be *unbiased* (close to T_q in expectation) and *stable* (has small variance). In this section, we first define and analyze an accuracy metric for coupon-collector configurations, then present our method for finding the best configuration for each query.

3.1 Coupon Collector's Accuracy

Given a specific query threshold T_q, a coupon-collector configuration is accurate if the number of random draws it needs has an expectation close to T_q and a small variance. Let us first analyze the expectation. We note that the traditional coupon-collector problem requires $n = m = 1/p$, so we present the following analysis for our generalized coupon-collector problem ($1 \leq n \leq m$, $0 \leq p \leq 1/m$):

Lemma 3.1. *A generalized coupon collector with m coupons in total, each coupon having probability p being drawn upon each random draw, and stops after collecting n different coupons, needs in expectation* $CC(m, p, n) \triangleq \sum_{j=0}^{n-1} \frac{1}{p(m-j)}$ *draws.*

Proof. With j coupons already collected, the probability that the next draw produces a new, unseen coupon (out of the $m - j$ remaining) is $p(m - j)$. Thus, the number of draws needed until receiving a new coupon is a geometric random variable $Geo(p(m-j))$ with expectation $\frac{1}{p(m-j)}$. We need to collect n new coupons, hence the total number of draws is $\sum_{j=0}^{n-1} Geo(p(m-j)) = \sum_{j=0}^{n-1} \frac{1}{p(m-j)}$ in expectation. □

However, the configuration with the closest expectation $CC(m, p, n)$ from T_q may have a large variance in the number of draws needed. Therefore, we define *Relative Error*, an accuracy metric for a distinct counting algorithm running query q with threshold T_q, that

simultaneously captures the bias and variance of a coupon-collector configuration.

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

By running the same algorithm many times with different random hash functions, we can have many observations of Relative Error for the same query, and we can subsequently define *Mean Relative Error* as the mean of all observations.

Next, we discuss how BeauCoup finds a coupon-collector configuration with small Mean Relative Error for every query.

3.2 Finding the Best Configuration

The BeauCoup compiler needs to identify one coupon-collector configuration for every query given the query's threshold T_q, and we focus on how we satisfy the strict per-packet memory access constraint. When implementing BeauCoup on PISA switches, our choice for m_q, p_q, and n_q is subject to hardware constraints. Namely, since a memory word is $w = 32$-bit we require $m_q \leq 32$, and to facilitate efficient mapping from random hash function to coupons we require p_q to be an integer power of two. Also, we must satisfy the average per-packet coupon limit γ_q: we require in expectation that we collect fewer than γ_q coupons per packet, i.e., $m_q \cdot p_q \leq \gamma_q$.

Thus, we use the following procedure to find the configuration given threshold T_q and per-packet coupon limit γ_q:

(1) For all feasible coupon probabilities $p_q = 2^{-j}$, we calculate the maximum number of coupons allowed, based on both the per-packet coupon limit and the word length: $\overline{m_q} = min(w, \gamma_q/p_q)$. We stop if $\overline{m_q} < 1$.
(2) For each p_q, we identify all feasible configurations $1 \leq n_q \leq m_q \leq \overline{m_q}$. We then calculate their expected number of draws $CC(m_q, p_q, n_q)$ for all feasible configurations, and accept a configuration as reasonable when it is within a 5% tolerance from T_q, i.e., $0.95T_q < CC(m_q, p_q, n_q) < 1.05T_q$. The 5% tolerance is selected because the minimum relative error for the optimal collectors is about 10%, and is relaxed when no reasonable configuration was found.
(3) Given all of the reasonable configurations, we choose the optimal configuration based on their minimum relative error, according to a lookup table prepared via simulations (shown later in the Evaluation section in Figure 5).

4 BEAUCOUP ON PISA HARDWARE

In this section, we describe how we implement BeauCoup on PISA programmable switches. PISA switches always process packets at line rate (at least 100Gbps per port), which requires the algorithms running on it to comply with several hardware-imposed resource constraints.

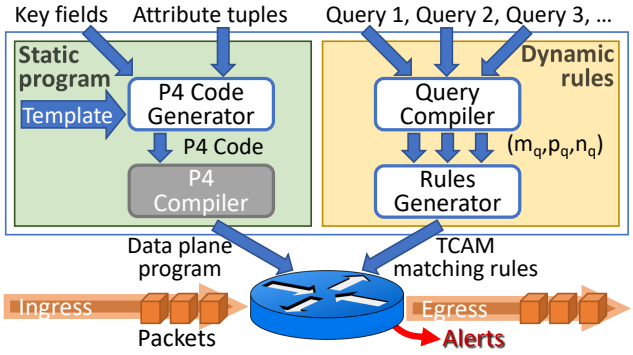

Figure 3: BeauCoup runs queries by installing a static data-plane program on the PISA switch, then generating and installing TCAM rules on the fly.

PISA switches have two kinds of memory. Ternary Content-Addressable Memory (TCAM) holds match-action rules installed by the control software, while Static Random Access Memory (SRAM) holds general-purpose register arrays that can be updated within the data plane. TCAM can simultaneously match a bit string with many match rules, and is typically used for forwarding packets by matching on the IP prefix. BeauCoup utilizes a small fraction of the available TCAM space to efficiently implement both the mapping from attributes to coupons and the tie-breaking process between queries. Meanwhile, BeauCoup collects coupons by updating SRAM entries. The SRAM memory space is limited (several megabytes), and more importantly we can only perform a small, constant number of memory accesses to SRAM per packet. In this paper, we primarily focus on the limited SRAM space and the limited number of SRAM accesses allowed.

BeauCoup's implementation has two components: the data-plane program executes the logic for collecting coupons, and the control algorithm transforms queries into coupon-collector configurations, as illustrated in Figure 3. Now we first introduce how we implement the data-plane program to run the coupon collectors on PISA hardware, then discuss how BeauCoup as a whole executes and updates queries.

4.1 Using TCAM for Drawing Coupons

BeauCoup needs to draw coupons based on the output of random hash functions. Since each hash function maps to a large number of coupons, we utilize the TCAM to efficiently check if the hash function's output value maps to any of the ranges defined by the coupons.

Each random hash function's output is encoded into 16 bits, and each coupon's corresponding range is translated to a bit prefix match for these random bits. For example, we translate the coupons of q_1 and q_2 shown in Figure 2 into matching rules in Table #1 in Figure 4. Coupon #1 of query q_1 matches on range $[0, 1/4)$, which is transformed to a bit prefix match `00*` (the first rule in Table #1). Coupon #2 of query q_2 matches on $[2/4 + 1/8, 2/4 + 2/8)$, which is transformed to prefix `101*` (the last rule in Table #1).

After we use TCAM tables to match on every hash function's output, we use a bit vector to represent if any one of the many hash

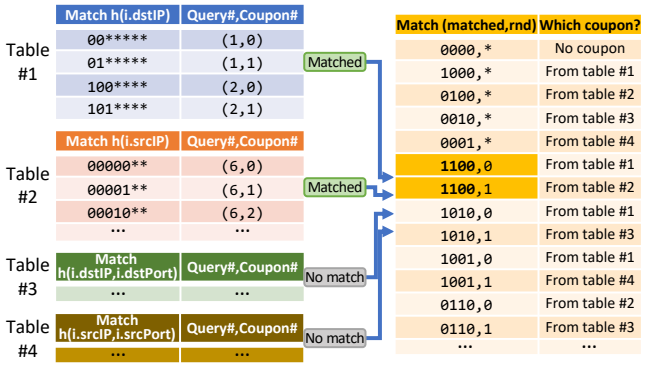

Figure 4: Using TCAM rules to draw coupons.

functions had matched with a coupon. As there could be zero or more coupons, we again use the TCAM to efficiently tie-break and select one coupon to collect when there may be multiple coupons available. The matching rules are trivial when there are zero or exactly one coupon matched. If there are exactly two coupons available, we flip a random coin (by using a random bit from the random number generator) to fairly tie-break and select one of the two for collection. We ignore all coupons if there are more than three; this has very minor effect on BeauCoup's accuracy, as we discuss in Appendix B.

We illustrate the coupon matching and the tie-breaking process in Figure 4. There are four random hash functions and four corresponding match tables (on the left) to draw coupons. After matching, Table #1 and #2 produced coupons while Table #3 and #4 did not. We use the bit vector `1100` to represent which tables produced coupons. A tie-breaking table (on the right) uses TCAM match rules to match on the bit vector `1100`, and there are two matching rules (highlighted in yellow). The table matches on the random bit to tie-break, and chooses either the coupon from Table #1 or the one from Table #2 as the final coupon for collection.

4.2 Recording Coupons in SRAM

After BeauCoup has selected a query q and chosen a coupon c for packet i (using TCAM matching), we need to collect c into the in-memory coupon table. We used the SRAM-based register arrays on PISA switches to record coupons and other states. Each array holds S memory words, indexed 0, 1,..., S-1, and each word has 32 bits. Given an index, we can read the existing value at this index, perform arithmetics, and write a new value; this counts as one memory access.

BeauCoup first extracts the query key $key_q(i)$ from the packet, then locates an index using the tuple $(q, key_q(i))$. We use an indexing random hash function H to map the tuple into an array index, denoted $idx = H(q, key_q(i))$.

BeauCoup defines three register arrays, each with S words. $\mathcal{TS}[\cdot]$ stores timestamps, and is used to enforce the query time window W for every coupon collector; we reclaim memory when a collector is timed out before collecting enough coupons. $\mathcal{QK}[\cdot]$ stores 32-bit checksums $checksum(key_q(i))$ and is used to detect hash collisions in the indexing hash function H, avoiding two keys adding coupons into the same collector bit vector. Finally, $CC[\cdot]$ stores all the coupon collector bit vectors.

The process for collecting the coupon c for query q and key $key_q(i)$ is as follows, accessing at most three words of memory, First, we calculate the array index $idx = H(q, key_q(i))$, and encode the coupon into a variable $onehot(c)$, a 32-bit binary string "`000...010...0`" with all bits `0` except one 1 at the location corresponding to the coupon c. Subsequently, we check whether we are creating a new coupon collector or adding this coupon to an existing collector, using query time window W and current timestamp $i.timestamp$:

- **Create new collector:** If $\mathcal{TS}[idx] < i.timestamp - W$, the current collector has expired. We allocate a new coupon collector by setting $\mathcal{TS}[idx] \leftarrow i.timestamp$ as well as $\mathcal{QK}[idx] \leftarrow checksum(key_q(i))$. We initialize the collector bit vector with one coupon: $CC[idx] \leftarrow onehot(c)$.
- **Update existing collector:** If $\mathcal{TS}[idx] \geq i.timestamp - W$ and $\mathcal{QK}[idx] = checksum(key_q(i))$, we accumulate into an existing coupon collector. We update its bit vector using bitwise-OR: $CC[idx] \leftarrow (CC[idx] \vee onehot(c))$. Now, if the number of one bits in $CC[idx]$ reaches n_q, we output an alert $(q, key_q(i))$.
- **Handle collision**: If $\mathcal{TS}[idx] \geq i.timestamp - W$ yet $\mathcal{QK}[idx] \neq checksum(key_q(i))$, we encountered a hash collision; the system ignores this coupon. This indicates there are too many active coupon collectors, hence the system is running out of memory. We discuss how to address memory size constraint and hash collisions in Section 6.

We note that coupon collectors for different queries uses the same block of memory space, statistically multiplexing their memory demand. Therefore, we may encounter high memory load when many different queries simultaneously collect coupons for many keys. We discuss BeauCoup's memory size requirement under real-world traffic settings in Section 5.2.3.

4.3 Query Compiler and Code Generation

Figure 3 presents the high-level architecture of the BeauCoup system. Given a set of queries $\mathcal{Q}$, we first run a query compiler (using the algorithm in Section 3.2) to compute a configuration $\{m_q, p_q, n_q\}$ for each query q, and produce the hash functions for attributes. The *query compiler* generates an intermediate representation with the mapping from each hash function's output values to all of the coupons. Subsequently, the *rules generator* uses these mappings to generate the TCAM matching rules and the corresponding action parameters, representing the query set $\mathcal{Q}$.

Meanwhile, BeauCoup generates the P4 code for the switch using a python-based *code generator*. The generator uses an algorithm template (approximately 750 lines), written under the Jinja [28] templating language, that implements BeauCoup's data-plane algorithm. Jinja enables auto-generating repeated P4 elements, such as defining multiple hash functions and variables, as demonstrated in Appendix A. Given the queries' key fields and attribute tuples as input, the code generator prepares the definition for hash functions, then expands the template into a P4 [9] program (approximately 1500 lines), which is subsequently compiled and installed into the PISA switch. When the TCAM matching rules are installed in the tables specified by the P4 program, the switch executes the query

set Q. We have open-sourced the complete template program, the code generator, as well as the query compiler on GitHub[1].

Although the packet parser (header field definitions), hash functions, and query key extraction rules are part of the P4 data-plane program, the TCAM matching rules can be updated on the fly. The user may frequently change the query set Q, by first running the query compiler and the rules generator, then installing the new matching rules, as long as all queries are using existing key fields and attribute tuples already defined in the data-plane program. This also avoids the potential network downtime caused by re-installing a new data-plane program, which would temporarily interrupt the switch's normal operation. The green shaded box on the left half of Figure 3 represents the heavy-weight update of the data-plane program, which is largely static, while the yellow shaded box on the right represents light-weight update of query matching rules, which can be installed swiftly without causing downtime. Still, using a new header field in a query's key or attribute definition requires re-generating P4 code and re-compiling the data-plane program.

5 EVALUATION

In this section, we demonstrate that BeauCoup can accurately and efficiently execute multiple queries. We first show that the query compiler produces good parameters for coupon collection. Then, we investigate BeauCoup's performance when answering queries over a real-world traffic trace, under limited memory access constraint, and show it achieves the same accuracy using 4x fewer memory accesses than alternatives. Finally, we show BeauCoup 's data-plane program only uses a modest fraction of the available hardware resources on a commodity switch.

5.1 Evaluating the Query Compiler

We now investigate the coupon-collector configurations generated by the query compiler under different thresholds T_q and average per-packet coupon limit γ_q. The compiler's running time is negligible (< 1ms) given its time complexity $O(w^2|Q|)$.

Recall that the query compiler outputs the configuration $\{m_q, p_q, n_q\}$ with the lowest Mean Relative Error given that its expected number of draws $CC(m_q, p_q, n_q)$ is close to the query threshold T_q. In Figure 5 we plot the minimum possible Mean Relative Error of various configurations, when the expected number of draws exactly matches the threshold ($T_q = CC(m_q, p_q, n_q)$). We note that adjusting p_q does not noticeably change the error, and only plotted the relationship between Mean Relative Error and (m_q, n_q) for all configurations in $2 \le n_q \le m_q \le 64$.

As we can see from Figure 5, in general, using more coupons leads to lower error. We can further observe that for any given m_q (total coupons), the configuration with minimal Mean Relative Error corresponds to a choice of n_q around $0.75m_q$. That is, the coupon-collector configuration should stop when around three-fourths of coupons are collected, as this leads to the least variance in the number of random draws required. We also verified that the $n_q \approx 0.75m_q$ heuristic still holds with thousands of coupons, although we defer a rigorous analysis to future work. However, when memory access is extremely constrained, the compiler often selects $n_q = m_q = 1$, as the configurations using more coupons consume many more memory accesses per packet.

[1]https://github.com/Princeton-Cabernet/BeauCoup

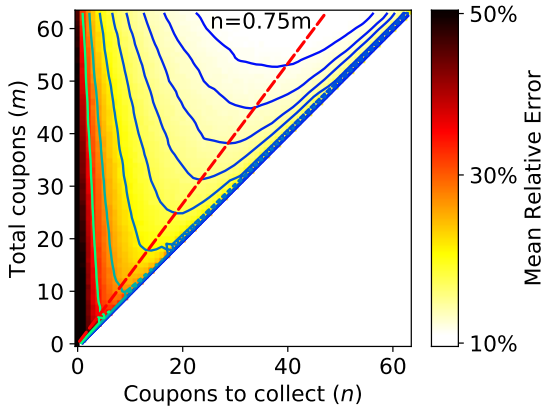

Figure 5: When using various coupon collector configurations, we find that collecting approximately $n = 0.75m$ out of m coupons produce the lowest error.

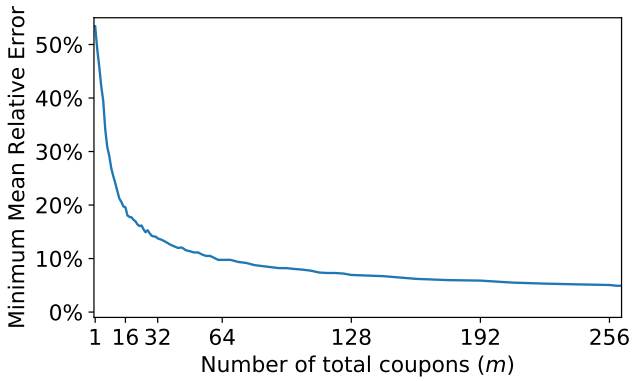

Figure 6: Using more coupons lead to lower Mean Relative Error. A coupon collector can achieve 13.7% minimum error when using $m = 32$ coupons.

We now look at the relationship between the minimum Mean Relative Error and the total number of coupons (m_q), as shown in Figure 6. In our current prototype implementation, we restrict the query compiler to use at most $m_q = 32$ coupons, as one memory read on the PISA hardware reads a 32-bit memory word. Using $m_q = 32$ coupons achieves 13.7% minimum error, which means BeauCoup may send a super-spreader alert upon seeing 860~1140 distinct IP addresses, given the threshold 1000. We note that BeauCoup can maintain more coupons in a collector by using multiple memory words, if a higher accuracy is desired. Using $m_q = 64$ coupons achieves 9.8% minimum error, while using 128, 256, or 1024 coupons achieves 6.9%, 5.0%, or 3.1% error respectively. These errors are comparable with the HyperLogLog distinct counting algorithm using the same memory space.

5.2 Query Accuracy

Now we evaluate the accuracy of BeauCoup queries over real-world network traffic, by first running a single query and comparing BeauCoup with related works, then run many queries simultaneously. Our experiments mostly focus on BeauCoup's accuracy under the limited memory *access* constraint by providing abundant memory for all algorithms. We also present some results regarding limited memory space.

5.2.1 One Query and One Key. We first demonstrate BeauCoup's coupon collectors are an efficient way to perform distinct count queries, by comparing them against other approximate distinct counting algorithms. Here we only focus on counting distinct attributes for one particular query and one particular key, as other distinct counting algorithms are designed for only one key and cannot support multiple keys.

In this experiment, we use different algorithms to count the number of distinct source-destination IP pairs in the traffic, and stop when the estimate exceeds $T = 1000$ distinct IP pairs. All algorithms are implemented in Python. We use the CAIDA Anonymized Internet Traces Dataset 2018 [5] (CAIDA trace), and repeat all runs 100 times with different random seeds.

HyperLogLog [13] is a widely-used approximate distinct counting algorithm, that counts distinct items by counting the maximum number of leading zeros seen from a random hash function. The algorithm splits its input and feeds them to multiple independent estimators, and outputs the harmonic mean across all estimators. We use a HyperLogLog instance with 64 estimators.

UnivMon [22] is the state-of-the-art multi-purpose measurement sketch that runs on PISA programmable switches, and can compute various functions over a set of attributes, including distinct counting. NitroSketch [21] performs sampling over sketch memory updates to reduce a sketching algorithm's memory access while preserving its accuracy. The authors of NitroSketch had proposed applying the NitroSketch technique to UnivMon to reduce UnivMon's average memory access per packet. We hereby refer to the new algorithm as NitroSketch-UnivMon. NitroSketch-UnivMon supports all the queries supported by UnivMon, including distinct counting. NitroSketch-UnivMon is the only sketch we are aware of that achieves fewer than one memory access per packet on average and supports distinct counting. We use 16 layers of 4x1024 CountSketch for UnivMon, and change NitroSketch's sampling parameters to let NitroSketch-UnivMon achieve different average memory access per packet.

We also include a packet sampling approach in the comparison. As analyzed by Spang and McKeown [30], it is possible to estimate the distinct number of flows (attributes) given a sampled subset of all packets, using a statistical estimator [6]. We sample each packet with a small probability p, and record each sampled packet's IP pair. Subsequently, we feed the sampled subset to the estimator.

We first note that the memory size used by BeauCoup is minimal: a coupon collector uses one word of memory, at most $w = 32$ bits. Including auxiliary data (timestamp and checksum), each key uses three words, or 96 bits. Meanwhile, one HyperLogLog instance with 64 estimators uses 320 bits of memory. As we discussed in Section 5.1, when using the same number of bits of memory space, coupon collectors can achieve comparable accuracy as HyperLogLog.

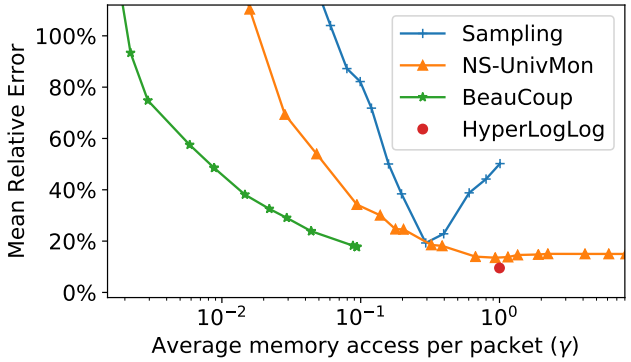

Figure 7: BeauCoup's coupon collector approach uses 4x fewer memory access than NitroSketch-UnivMon or sampling to achieve the same accuracy.

On the other hand, NitroSketch-UnivMon uses 256 kilobytes of memory space and is not directly comparable, as it is a multi-purpose sketch supporting more than distinct counting. It is possible to fit a handful of instances of NitroSketch-UnivMon into a switch's data-plane memory space, but it is unfeasible to run multiple queries with multiple keys, which requires thousands of instances. Packet sampling uses $O(p \cdot L)$ memory space, proportional to the sampling probability and stream length.

Since we need to simultaneously answer multiple queries under a total per-packet memory access constraint, each BeauCoup query can only make a very small number of memory accesses per packet. We now compare the accuracy of each distinct counting algorithm under the same average memory access constraint of $\gamma \leq 1$ words per packet:

- When using packet sampling, for each sampled packet, we need to access two words of memory to save its IP pair. Thus, we can satisfy the per-packet memory access constraint by setting the sampling probability to $p = \gamma/2$.
- For NitroSketch-UnivMon, we tune each layer's NitroSketch sampling probability individually to achieve $\gamma/16$ average memory access, thus making total memory access across all layers to fit within γ words per packet. Since not all layers use their access budgets fully, we record the actual number of total memory accesses in experiments.
- For BeauCoup coupon collectors, recall that collecting each coupon requires accessing $c = 3$ words (for coupon vector, timestamp, and checksum). We specify an average per-packet coupon limit $\gamma_q = \gamma/c$, and use the BeauCoup query compiler to find the coupon collector configuration that satisfies the constraint. Here we also record the actual number of memory accesses.
- Finally, although HyperLogLog is very accurate, it always accesses exactly one word of memory per packet, regardless of the number of estimators. We nevertheless included its accuracy for reference.

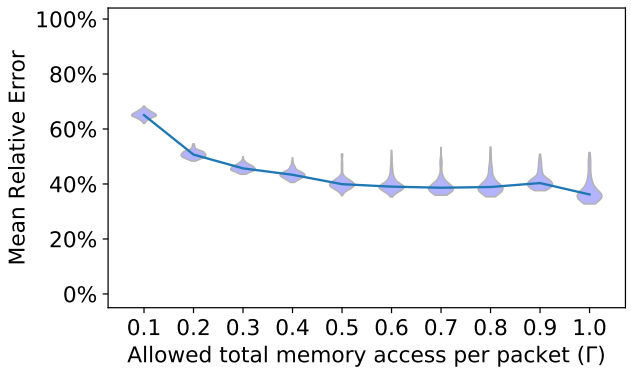

Figure 8: The average error of all queries gradually improve as we allow more memory access per packet, which is shared among all queries.

In Figure 7, we show that BeauCoup's coupon collector achieves the same accuracy (Mean Relative Error, plotted on y-axis) using at least 4x fewer memory accesses (γ, plotted on x-axis with log scale), compared with NitroSketch-UnivMon, packet sampling, or HyperLogLog.

We note that the statistical estimator used by the packet sampling approach [30] is designed for sparse samples, looking at IP pairs sampled exactly once or twice. Thus, it works better for sparse samples and performs poorly with a very high sampling rate above 0.5, creating non-monotonicity in the figure.

To achieve less than 25% Mean Relative Error for queries, BeauCoup needs 0.04 words of memory access per packet, which means we can run about 25 queries together per word of memory access per packet, while NitroSketch-UnivMon requires 0.2 words of memory access, and can only run about five queries for the same memory access limit. At higher error ranges (e.g., to achieve less than 50% Mean Relative Error), BeauCoup only needs 0.009 words of memory access, while NitroSketch-UnivMon requires 0.09, yielding a 10x saving. The improvements are similar for other attribute definitions and thresholds.

5.2.2 Multiple Queries and Keys. Next, we run BeauCoup with multiple queries and observe the average relative error under varying memory access constraints. We wrote $|Q| = 26$ queries that resemble monitoring demands a network administrator may have, with keys and attributes defined using combinations of source and destination IP addresses and TCP/UDP ports. The queries use various different combinations of packet header fields as their key and attribute definitions. Some queries also use the timestamp as the attribute definition—recall that we can count the number of packets by performing distinct counting over timestamps. The thresholds range from 100 to 10000, and are selected based on the likely use cases of the particular queries. In each experiment, we set Γ, the total memory access constraint for all queries, from 0.1 to 1 access per packet. We then run the query compiler to fairly allocate memory access and generate the coupon-collector configuration for each query.

After obtaining the coupon-collector configurations, we run BeauCoup in a python-based simulator, which is behaviorally equivalent to the data-plane P4 program, but allows us to freely tune all parameters and concurrently run many simulations with different random seeds. We once again use the CAIDA trace in the following experiments.

Average accuracy across queries. Figure 8 shows the overall accuracy of all queries, measured by Mean Relative Error, given different total memory access limits Γ. We can observe that when the memory access limit becomes lower, the error becomes higher, and the accuracy of different queries gradually converges. This is because when we have abundant memory accesses, the queries with higher thresholds do not need to use all of their fair share of memory accesses, and can achieve better accuracy than those actually constrained by memory access; when all queries are constrained, the fair allocation policy leads to similar accuracy for all queries.

Per-query accuracy. Now we scrutinize the accuracy of each query. We first compare the effect of increasing memory access limit Γ on each query's average relative error. In Figure 9, we choose four different queries with various T_q from 100, 500, 5000, to 10000 and analyze their accuracy. Naturally, the query with the lowest threshold is the hardest to execute, as it requires coupons with larger probability p_q and easily exhausts its memory access budget. Increasing Γ allows the query to increase accuracy significantly. For queries with larger T_q, the improvement is not as significant.

Notably, the query with $T_q = 10000$ reaches its optimal accuracy when $\Gamma = 0.2$, and its accuracy slightly deteriorates when we allow more memory accesses. This is due to having collisions with other queries when the system draws more than one coupon and enters tie-breaking more often, which slightly skews the probability of drawing each coupon.

We also compare different queries with the same $T_q = 1000$ yet with different key_q and $attr_q$ definitions. Here we use four queries as an example, the first one being super-spreader. As we can see from Figure 10, their average relative error has almost the same relationship regarding the total memory access constraint Γ. The third plot in Figure 10 has a slightly higher variance, and is because this particular query outputs fewer alarms in our experiment trace, hence has more outliers for the average relative error statistics.

5.2.3 Memory Size. So far, we have focused on limited memory access and assumed unlimited memory size and an infinite time window. However, practical systems have a limited amount of memory (S) and can run out of space for large window size W.

We first observe that the number of unique query keys present in the traffic usually follows power law. For a stream of L packets, we can observe L^{α_q} unique keys, with α_q being specific to the traffic and different key definitions. For the CAIDA trace, α_q ranges between 0.7 to 0.85. Therefore, given the average per-packet coupon limit γ_q, we can give an upper bound $(\gamma_q L)^{\alpha_q}$ for the number of coupon collectors needed for query q, and therefore the maximum total memory needed by all queries is upper-bounded by $\sum_{q \in Q} (\gamma_q L)^{\alpha_q}$.

Figure 11 shows the actual memory space requirement of BeauCoup with regards to different time window sizes W, when processing the same query set Q under the CAIDA trace, under a log-log scale. We can observe that the relationship between the memory

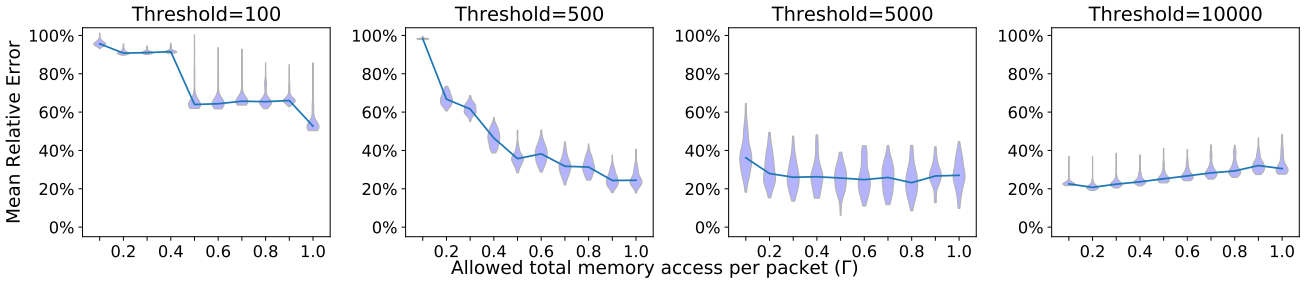

Figure 9: Query with the lowest threshold experiences the most significant accuracy improvement when allowing more memory access per packet.

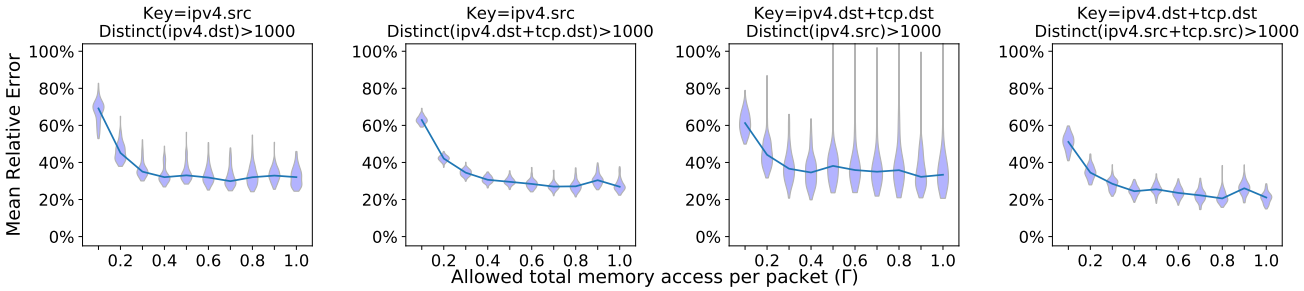

Figure 10: Queries with the same threshold exhibits similar accuracy improvement trend when given more allowed memory access, despite different key and attribute definitions.

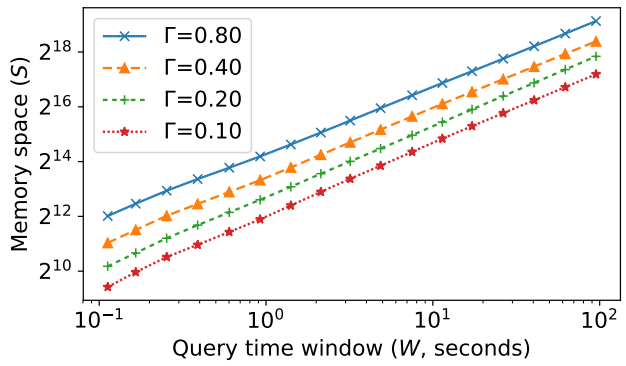

Figure 11: The query time window size W and the memory space S (number of coupon collector bit vectors) required by BeauCoup follows power law.

size and window size closely follow a power law with an exponent $\alpha = 0.80$. For example, for a time window of $W = 1$ second and memory access limit of $\Gamma = 0.1$ word per packet, BeauCoup needs to store 4096 coupon collectors (48 kilobytes), while doubling the time window to $W = 2$ seconds enlarges the memory size requirement by $2^{\alpha} = 1.74$ times, to 7150 collectors (84 kilobytes). A practical system on PISA switches can easily support 65,536 collectors, corresponding to a time window $W = 30$ seconds for the CAIDA trace. Still, BeauCoup is optimized for memory access constraint, and we defer the discussion on how to adapt BeauCoup with insufficient memory in Section 6.

Component	Match Coupons	Extract Key	Collect Coupons	Teardown	Overall
TCAM	39.6%	2.3%	0%	0%	**13.2%**
SRAM	9.1%	2.1%	26.3%	0%	**12.3%**
Instruction	25.0%	7.3%	5.4%	3.1%	**12.8%**
Hash Unit	50.0%	61.1%	29.1%	0%	**41.7%**

Table 3: BeauCoup's hardware resource utilization, categorized into four functional components.

5.3 Hardware Resource Utilization

To run on PISA switches and process packets at 100Gbps line rate, BeauCoup's data-plane program must satisfy other resource constraints beyond limited memory access. BeauCoup's auto-generated P4 data-plane program runs on an EdgeCore Wedge100-32BF programmable switch. It consumes about 40% of the programmable switch's hash calculation units and less than 15% of other resources. We note that BeauCoup is not bottlenecked by TCAM match table size. The current version of our data-plane program supports matching each attribute's hash function output to 4096 different coupons; since every query uses at most 32 coupons, the program supports at least $\frac{4096}{32} = 128$ queries for each attribute. 4096 is the default size for the TCAM match tables set by the compiler, and can be extended as needed. Resource utilization other than TCAM is independent of the number of simultaneous queries we run.

To produce a more detailed picture of BeauCoup's resource utilization, we slice the data-plane program into four sequential functional components, and in Table 3 we drill down the utilization for different types of resources by each component. We can see different functional components have distinctive resource utilization profiles.

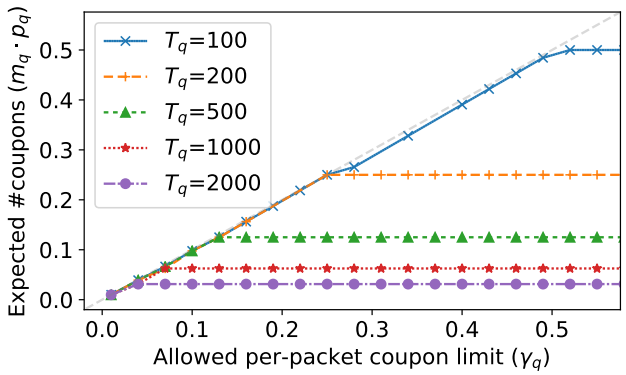

Figure 12: Queries with higher threshold T_q need fewer memory accesses per packet.

For example, matching coupons extensively uses hash units to calculate random hash functions and uses TCAM to draw coupons, while not using much SRAM; in contrast, collecting coupons requires no TCAM, but uses SRAM to store the bit vectors.

Although the BeauCoup data-plane program uses more hardware resources than running one instance of HyperLogLog or UnivMon for a single key definition, we note that the data-plane program already supports various different key and attribute definitions, allowing us to install new queries on the fly without re-compiling the data-plane program. Furthermore, BeauCoup does not exhaust any one switch resource, and its unique resource usage profile cohabitates well with other typical resource-heavy switch functions or algorithms. When two algorithms use the same resource heavily but at different pipeline stages, we can tessellate them without causing resource contention. For example, performing Equal-Cost Multi-Path (ECMP) routing requires computing hash functions late in the switching pipeline, where BeauCoup does not compute many hash functions when collecting coupons; running network measurement sketches like UnivMon [22] or PRECISION [4] requires using SRAM memory early in the pipeline, whereas BeauCoup does not consume a lot of SRAM early in the pipeline when it is matching coupons.

6 DISCUSSION

Fairness between queries. In this paper, we use a fair allocation policy to distribute the limited memory access among all queries.However, queries with larger thresholds require fewer memory accesses to achieve the same accuracy. Figure 12 evaluates the optimal configurations found by the query compiler under different per-packet coupon limit γ_q, for various query thresholds T_q. A query with a small threshold of $T_q = 100$ almost always uses all of its budget (with $m_q \cdot p_q$ very close to γ_q), while queries for larger thresholds do not need their full share. We can improve the allocation policy to redistribute these "leftover" budget to improve the accuracy of the queries with the lowest thresholds. We can repeat the process until the leftover is negligible or no query can be improved.

Multi-stage coupon table. Our current prototype uses a single hash-indexed array for storing coupons. Extending this structure to a multi-stage table would offer several benefits. First, hash collisions are inevitable even when the hash table is lightly filled; using multiple tables can provide a query-key pair more chances to insert successfully despite hash collisions. With more memory accesses, we can also allow simultaneously collecting at most 2 or 3 coupons per packet. Second, we can use multiple stages of tables to assign more coupons to each collector, for example by using two tables to implement $m = 64$ coupons per collector.

Memory space. In designing BeauCoup our main concern was supporting multiple queries with limited memory access. If memory size becomes constrained, BeauCoup has two possible ways to address the issue. First, we can voluntarily limit memory access (Γ) below the limit imposed by the hardware; a smaller Γ reduces space requirements, as demonstrated in Figure 11. Second, we can implement an eviction mechanism that finds the coupon collectors least likely to succeed; for example, we could look at the number of coupons not yet collected, and how much time has elapsed since the last coupon was collected by this collector.

Distributed Monitoring. Currently, BeauCoup processes traffic at a single switch. To extend BeauCoup to multiple vantage points, we could use multiple switches to run the same random hash functions and a centralized collector to collect all the coupons. Each switch only needs to send packet to the centralized collector when a new coupon is collected. We can minimize the traffic overhead by specifying a small per-packet coupon limit, and de-duplicating the coupons at the switches before sending. Similar to HyperLogLog registers, BeauCoup coupon collector vectors are trivially mergeable.

Security. Some network queries look for adversarial traffic, and an attacker is motivated to craft its attacking traffic to disrupt those queries. As BeauCoup uses random hash functions with random seeds, the attacker cannot predict which packets lead to coupon collection without knowing the seeds. However, with the seeds leaked, the attacker can precisely know which packets trigger a coupon, and thus can deliberately craft traffic to avoid being reported. We therefore should periodically replace the hash seeds and make sure they are not leaked.

Our current prototype uses the CRC-32 family of hash functions with different polynomials, natively available on the programmable switch hardware. CRC-32 is prone to linear correlation, and an attacker may recover the seed when it simultaneously controls the input packets and observes the output coupon activation (performing a *Known Plaintext Attack*). To defend against powerful attackers, a more secure BeauCoup implementation should use cryptographic hash functions. We leave this as future work.

7 RELATED WORK

Approximate distinct counting. Plenty of related work discusses how to approximately count distinct elements under limited *memory space*, culminating in the widely-used HyperLogLog [13] distinct counting algorithm. [10] surveyed these prior works, which can be roughly categorized into two flavors: K-Minimum-Value and Distinct Sampling. K-Minimum-Value [2] computes a random hash function over all input elements, and uses the k smallest values observed to infer how many distinct elements exist. Distinct

Sampling [15] samples new distinct elements at a small probability, and infers 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 [12, 13] builds upon the idea of Distinct Sampling but instead partitions the incoming stream into k sub-streams and uses k independent estimators, and outputs the harmonic mean of their estimates. Each estimator records the longest consecutive leading zeros seen from the output bits of a random hash function. We note that our implementation of a m-bit coupon collector is in fact equivalent to the HyperLogLog algorithm using $1/p$ sub-streams, with the $1/p$ estimators each output only one bit. However, we only store the output of first m estimators, truncating the other $1/p - m$ estimators to reduce memory access. Alternatively, a coupon collector can be viewed as a $1/p$-bit Bloom Filter with only one hash function, truncated to the first m bits to reduce memory access. Bloom Filters are originally designed for membership queries but can also be used for approximate distinct counting, as analyzed by Assaf et al. [1].

We also note that universal sketching (UnivMon [22]) can compute many different functions over the input frequency vector, as long as the function is monotonic and bounded by the l_2-norm. In particular, it can compute distinct counting (the l_0-norm). For input length L with A unique items (attributes), UnivMon maintains $log(A)$ different count sketches, and requires $\Gamma = O(log(A))$ memory access per packet in the worst case.

Memory model. In [24], Muthukrishnan surveyed several established streaming analysis models, and used an abstraction of maintaining one high-dimensional vector. Each incoming item changes 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 falls under the cash register model, for each individual query and sub-streams of the input stream partitioned by the query key.

The cell probe model [19, 26, 33] is a limited memory access model often used to prove data structure lower bounds. In [33], Yao proved that $\lceil \log(S) \rceil$ probes (memory accesses) are necessary to check whether an item exists in a memory array of size S. Larsen et al. [19] 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 adapt cell probe into stream processing to allow at most Γ memory words to be accessed per packet, while introducing a new notion of sub-constant memory access, requiring each query to access fewer than one memory word 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. For example, a modern CPU has a cache size of a few megabytes. The traditional streaming algorithm model strives to fit an entire data structure (sketch) within this cache size, while our model resembles limiting the number of accesses to external memory or disks, which are slower to access but considerably larger.

In [27], 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 [18], Kim et al. implemented a practical off-chip memory for PISA switches.

Reducing memory access. NitroSketch [21] is a novel technique that reduces memory access for sketching algorithms. The authors identified memory access as one of the most expensive operations when running network measurement tasks on CPUs, and proposed to sample on memory accesses to improve performance. Given a sampling probability p, all the +1 updates to the original sketch data structures are changed to $+1/p$ updates with probability p. A smaller p can further reduce memory accesses and accommodate faster packet processing. NitroSketch can be applied to many exising measurement sketches, including Count Sketch [7] and Count-Min Sketch [11], to improve performance without significantly impact accuracy. Compared with the naive approach of sampling packets, NitroSketch achieves better accuracy when given the same amount of memory access.

NitroSketch can be applied to UnivMon and produce a distinct counting algorithm with sub-constant memory access. UnivMon consists of multiple layers each hosting a Count Sketch. For every incoming packet, we first select which UnivMon layers to update using the original UnivMon mechanism, then each layer independently samples the counter updates into its Count Sketch using the NitroSketch mechanism, possibly using different sampling parameters according to the rate of each layer's incoming packets. The combined data structure NitroSketch-Univmon now uses sub-constant average memory access, and the accuracy loss is negligible when we reduce memory access by 50%-75% percent. However, the accuracy for distinct counting suffers greatly when we reduce memory access by 90%-99%, as we have shown in Section 5.2.

8 CONCLUSION

We present BeauCoup, a system for simultaneously running many distinct-counting based network monitoring queries, under limited memory access per packet. BeauCoup is implemented on PISA programmable switches and consume only moderate hardware resources, and evaluation showed it uses 4x fewer memory accesses to achieve the same error rate compared with other state-of-the-art measurement sketch.

ACKNOWLEDGMENTS

This research is supported in part by NSF Grant No. CNS-1704077, the NSF Alan T. Waterman Award Grant No. 1933331, a Packard Fellowship in Science and Engineering, the Simons Collaboration on Algorithms and Geometry and The Eric and Wendy Schmidt Fund for Strategic Innovation.

We sincerely thank the anonymous reviewers and our shepherd Dave Levin for their thoughtful comments and feedback. We also thank David Walker, Satadal Sengupta, and Mina Tahmasbi Arashloo for their help and feedback for this paper.

REFERENCES

[1] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. 2018. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM 2018*. IEEE, 2204–2212.

[2] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. 2002. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 1–10.

[3] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. 2020. Routing Oblivious Measurement Analytics. In *IFIP Networking*.

[4] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing Heavy-Hitter Detection Algorithms for Programmable Switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[5] CAIDA. 2018. The CAIDA UCSD Anonymized Internet Traces 2018 - March 15th. (2018). https://www.caida.org/data/passive/passive_dataset.xml

[6] Anne Chao. 1984. Nonparametric estimation of the number of classes in a population. *Scandinavian Journal of Statistics* (1984), 265–270.

[7] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15.

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

[9] The P4 Language Consortium. 2018. $P4_{16}$ Language Specifications. (2018). https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

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

[11] 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.

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

[13] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AOFA)*.

[14] 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.

[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] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. 2018. Network-Wide Heavy Hitter Detection with Commodity Switches. In *ACM SIGCOMM Symposium on SDN Research*. 8:1–8:7.

[18] 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.

[19] 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.

[20] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*. 311–324.

[21] Zaoxing Liu, Ran Ben Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. NitroSketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*. 334–350.

[22] 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.

[23] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. 2020. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *SIAM-ACM Symposium on Algorithmic Principles of Computer Systems*. 31–44.

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

[25] 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.

[26] Mihai Patrascu. 2008. *Lower Bound Techniques for Data Structures*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.

[27] Salvatore Pontarelli, Pedro Reviriego, and Michael Mitzenmacher. 2018. EMOMA: Exact Match in One Memory Access. *IEEE Transactions on Knowledge and Data Engineering* 30, 11 (2018), 2120–2133.

[28] Daniel Rubio. 2017. Jinja templates in Django. In *Beginning Django*. Springer, 117–161.

[29] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research*. 164–176.

[30] Bruce Spang and Nick McKeown. 2019. On estimating the number of flows. In *Stanford Workshop on Buffer Sizing*.

[31] 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*.

[32] 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.

[33] Andrew Chi-Chih Yao. 1978. Should Tables Be Sorted? (Extended Abstract). In *Foundations of Computer Science*. 22–27.

Appendices are supporting material that has not been peer-reviewed.

A TEMPLATING P4

We use the python-based Jinja templating library to automatically expand our code template into P4 code. Here, we show two excerpts from the template that highlights how templating helps us efficiently generate the P4 data plane program.

Example 1: generate code for every hash function.

```
struct ig_metadata_t {
  {% for h in hash_functions %}
    bit<16> h_{{h.id}};
    bit<1> h_{{h.id}}_matched;
    bit<8> h_{{h.id}}_query_id;
    bit<8> h_{{h.id}}_coupon_id;
    bit<8> h_{{h.id}}_query_n;
    bit<4> h_{{h.id}}_query_keydefn;
  {% endfor %}
  bit<32> coupon_onehot;
  bit<1> random_coin;
  //...
}
{% for h in hash_functions %}
  action calc_hash_{{h.id}}(){
    ig_md.h_{{h.id}}=hash_{{h.id}}.get({ {{h.fields}} });
  }
  action set_h_{{h.id}}_matched(bit<8> qid, bit<8> cid,
       bit<8> n, bit<4> kdf){
    ig_md.h_{{h.id}}_matched=1;
    ig_md.h_{{h.id}}_query_id=qid;
    ig_md.h_{{h.id}}_coupon_id=cid;
    ig_md.h_{{h.id}}_query_n=n;
    ig_md.h_{{h.id}}_query_keydefn=kdf;
  }
  action set_h_{{h.id}}_no_match(){
    ig_md.h_{{h.id}}_matched=0;
  }
{% endfor %}
```

Example 2: generate match-action logic.

```
action write_onehot(bit<32> o){
  ig_md.coupon_onehot = o;
}
table tb_set_onehot {
  key = {
    ig_md.h_selected_coupon_id: exact;
  }
  size = 32;
  actions = {
    write_onehot;
  }
  default_action = write_onehot(0);
  const entries = {
    {% for i in range(32) %}
      {{i}} : write_onehot(32w{{2**i}});
    {% endfor %}
  }
}
```

B COUPON COLLISION PROBABILITY

In this section, we show that ignoring all coupons when there are more than three coupons simultaneously matched by multiple hash functions only affects BeauCoup's accuracy by a few percent.

Although we restrict the expected number of coupons drawn per packet $\sum_{q \in Q} \gamma_q$ be bounded by 1, it is possible to have multiple coupons drawn 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 drawing one coupon.

We consider the system uses $H \geq 3$ random hash functions, each with activation probability $x_1, x_2, \ldots, x_H$, and we have $\sum x_i \leq 1$. Each random hash function will activate coupons independently, hence the total number of coupons drawn 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 $C \geq 3$. We can prove that the probability for having more than $C \geq 3$ coupons drawn is maximized when all hash functions share the same probability, i.e., $x_i = \frac{1}{H}$, due to the inequality of arithmetic and geometric means. In this case, the number of coupons drawn follows a binomial distribution $B(n = H, p = \frac{1}{H})$. Hence, plug in $H = 11$ (from the example query set we used in Section 5), we have

$$\Pr\left[B(n = H, p = \frac{1}{H}) \geq 3\right] = 7.11\%.$$

That is, the probability for a packet matches with more than 3 coupons is at most 7.11%.

This is smaller than or on par with the optimal average relative error achieved by coupon collectors for distinct counting (about $10\% \sim 20\%$), 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 small bias for individual coupon's activation probability; we leave the correction for this bias in the query compiler for future work.