A Fast and Compact Method for Unveiling Significant Patterns in High Speed Networks

Tian Bu*, Jin Cao*, Aiyou Chen*, Patrick P. C. Lee†
*Bell Laboratories, Alcatel-Lucent, NJ, USA
†Department of Computer Science, Columbia University, New York, NY, USA
{tbu, cao, aychen}@research.bell-labs.com, pclee@cs.columbia.edu

***Abstract*— Identification of significant patterns in network traffic, such as IPs or flows that contribute large volume (heavy hitters) or introduce large changes (heavy changers), has many applications in accounting and network anomaly detection. As network speed and the number of flows grow rapidly, tracking per-IP or per-flow statistics becomes infeasible due to both the computational overhead and memory requirements. In this paper, we propose a novel *sequential hashing scheme* that requires only $O(H \log N)$ both in memory and computational overhead that are close to being optimal, where N is the the number of all possible keys (e.g., flows, IPs) and H is the maximum number of heavy keys. Moreover, the generalized sequential hashing scheme makes it possible to trade off among memory, update cost, and detection cost in a large range that can be utilized by different computer architectures for optimizing the overall performance. In addition, we also propose statistically efficient algorithms for estimating the values of heavy hitters and heavy changers. Using both theoretical analysis and experimental studies of Internet traces, we demonstrate that our approach can achieve the same accuracy as the existing methods do but using much less memory and computational overhead.**

I. Introduction

Monitoring and detecting significant behaviors in a network, such as the presence of persistent large flows or a sudden increase in network traffic due to the emergence of new flows, are essential for network provisioning, management and security because significant behaviors often imply events of interests.

In this paper, we focus on the detection of two important significant behaviors known as *heavy hitters* and *heavy changers*. A heavy hitter is a key whose traffic exceeds a pre-defined threshold, whereas a heavy changer is a key whose change in traffic volume between two monitoring intervals exceeds a pre-defined threshold[1]. Here a key represents a source IP address/port, a destination IP address/port, or their combinations such as the five-tuple flow. For instance, a flow that accounts for more than 10% of total traffic, which is a heavy hitter by flows, may suggest the violation of a service agreement. On the other hand, a sudden increase of traffic volume flowing to a destination, which is a heavy changer by destination, may indicate either a hot spot, the beginning of a DoS attack, or traffic rerouting due to link failures elsewhere. The goal of the *heavy key detection problem* is to identify all *heavy keys* (i.e., either heavy hitters or heavy changers) and estimate their associated values with a low error rate while minimizing both memory usage and computational overhead.

However, as the Internet continues to grow in size and complexity, the ever-increasing network bandwidth poses great challenges on monitoring heavy keys in real time due to computational and storage constraints. To identify any network flow that causes significant volume change, the system should scale up to at least 2^{104} keys[2]. Some fundamental requirements for monitoring and detecting significant patterns in real time for high bandwidth links are discussed below.

- *Fast per-packet update*. The per-packet update speed has to be able to catch up with the link bandwidth even in the worst case when all packets are of the smallest possible size. Otherwise the real time constraint is violated.
- *Fast discovery of significant patterns*. The detection delay of significant patterns should be short such that important events like network attacks and link failures can be responded in time before any serious damage is made.
- *High accuracy*. Both false positive and false negative rates should be minimized. It is well understood that having a false negative may miss an important event and thus delay the necessary reaction. Having a false positive, on the other hand, may trigger unnecessary responses that waste resources.

Data monitoring algorithms based on efficient data structures have been studied for heavy hitter detection (e.g., [7]) or traffic-volume query (e.g., [1]). Estan and Varghese [5] use parallel hash tables to identify large flows using a memory that is only a small constant larger than the number of large flows. However, such schemes only address heavy hitter detection, but not heavy changer detection. Both [6] and [12] address heavy changer detection. In particular, [12] proposes a modularized hashing scheme to narrow down a candidate set of heavy keys. Our proposed approach improves the scheme in [12] not only on accuracy but also on computational overhead and memory usage.

In summary, the main contributions of this paper are:

- We derive a lower bound of memory usage when applying parallel hash tables for heavy key detection for a given

[1]There are more sophisticated definitions of "change" that account for traffic forecast models. However, the technique we develop in the paper would also apply to such definitions with linear forecast models. Using the simple definition of change allows us to explain the technique more clearly.

[2]This number is calculated based on the number of possible five-tuple flows: source IP address (32 bits), source port (16 bits), destination IP address (32 bits), destination port (16 bits), and protocol (8 bits). The number may be significantly smaller for realistic traffic since not all possible combination of these fields are possible

error rate.

- We propose a *sequential hashing scheme* that uses multi-level hash arrays for fast and accurate detection of heavy keys while minimizing computational overhead and memory usage. Moreover, we demonstrate our scheme can trade off between computational overhead and memory usage so as to maximize the overall system performance when being implemented on different hardware architectures.
- We design efficient yet accurate methods for estimating the values of heavy keys. We also demonstrate that our estimation methods can further reduce errors introduced in the detection stage. With the help of the estimation step, our detection scheme can be more memory efficient by allowing a high error rate in the detection stage and then eliminating the errors in the estimation stage.
- Through extensive simulation using real Internet traces collected from a high speed link, we show that our scheme yields more accurate results, yet is more memory and computationally efficient, than existing work.

The balance of the paper is organized as follows. We derive a lower bound of memory requirement when using parallel hash tables for heavy hitter/changer detection in Section II. Section III focuses on the multi-level hash techniques for heavy hitter/changer detection. In Section IV, we present efficient algorithms for estimating the values of heavy hitters/changers. Section V shows the evaluation results using Internet traces. We conclude the paper in Section VI.

II. Lower bound of Memory Requirement of a Hash Array

We model the set of network traffic within a measurement interval as a stream of data that arrive sequentially, where each item (x, v_x) consists of a key $x \in \{0, 1, \ldots, N-1\}$ and an associated value v_x. The identification of heavy keys (i.e., either heavy hitters or heavy changers) is straightforward if all values of v_x are known. However, tracking the exact values of v_x for all x may not be feasible for large N. To overcome this, as proposed in [5], [12], we introduce here the use of a single hash array for approximating the heavy keys where a hash array consists of M hash tables each with K buckets. The hash functions for each table are chosen independently from a class of 2-universal hash functions, and so the K buckets of each table form a random partition of N keys. We define $y_{m,j}$ as the sum of v_x for all x in the jth bucket in the lth table. Table I summarizes the important notation used in this paper.

In this section, we derive the lower bound of memory (in terms of the total number of buckets in a hash array) required for identifying the heavy keys in network traffic using a single hash array. We first describe our analysis for the case of heavy hitter detection. We will then show how the results may also apply to heavy changer detection.

A. Memory lower bound for heavy hitter detection

Recall a heavy hitter is a key x whose traffic value v_x exceeds a pre-specified threshold t. Suppose there are H heavy hitters. Call a bucket *heavy* iff its y value crosses the threshold t. For any heavy hitter, it is easy to see that the bucket that it falls into in each of the M tables is a heavy bucket. Therefore, a superset of heavy hitter keys, say $\mathcal{C}$, can be formed by using the intersection of M subsets, each of which consists of keys in the *heavy* buckets corresponding to one table.

TABLE I
Notation

x, v_x	key and the value associated with key x in the stream
N, N_i	size of key set
M, M_i	number of hash tables in one hash array
U	memory size (total number of buckets)
H	true number of heavy hitters/changes
K	size of a hash table
γ	H/K
ϵ, α	expected number of false positives divided by H (Eq. 2)
D	number of hash arrays (also the number of words in a key)
$\mathcal{C}$, $\mathcal{C}_i$	size of the candidate set of heavy hitters
$y_{m,j}, y_{i,m,j}$	sum of v_x for all x mapped to bucket j of table l

(Notation with subscript i denotes the corresponding quantities for the ith hash array in the sequential hashing scheme presented in Section III.)

In order to derive the lower bound, we assume that the traffic distribution is very skewed such that the sum of any set of non-heavy hitter key values is less than the threshold, i.e., the contributions of non-heavy hitters are negligible. expected size of order H, Assume that $H \ll N$. Let Z be the number of heavy hitters contained in an arbitrary bucket, and let $\gamma = H/K$, i.e, $K = \gamma^{-1}H$. The following two lemmas describes the distribution of Z and the expected size $E|\mathcal{C}|$ of set $\mathcal{C}$ in the lower bound case.

Lemma 1: $Z \approx Binomial(\frac{1}{K}, H)$. When H is large (say greater than 100), $Z \approx Poisson(\gamma)$.

The proof is straightforward and is omitted. When $\gamma = \log 2$ (see Theorem 1 below), Lemma 1 indicates that about 50% of the buckets do not contain any heavy hitters and that among heavy buckets about 70% of them contain exactly one heavy hitter.

Lemma 2: $E|\mathcal{C}| \approx H + (N-H)(1-(1-\frac{1}{K})^H)^M$. When H is large, then

$$E|\mathcal{C}| \approx H + (N-H)(1-e^{-\gamma})^M. \quad (1)$$

Proof: Let p_e be the probability that a non-heavy hitter falls into the set $\mathcal{C}$. Notice the probability that a non-heavy hitter falls into the *heavy* buckets of the l-th table is $p_l \approx 1-(1-\frac{1}{K})^H$, since each heavy hitter can be treated independently as an approximation due to $H \ll N$. The result follows readily from $p_e = \prod_{l=1}^{M} p_l$ and $E|\mathcal{C}| = H + (N-H)p_e$. ■

For the set $\mathcal{C}$, let ϵ be the *expected normalized false positives* defined as the expected number of false positives divided by H[3], i.e.,

$$E|\mathcal{C}| = H + \epsilon H. \quad (2)$$

Then by (1), for a given value ϵ and a large H, the required number of tables of the hash array is

$$M = -\frac{\log(N\epsilon^{-1}H^{-1})}{\log(1-e^{-\gamma})}. \quad (3)$$

Therefore, the required memory, say $U \equiv MK$, is logarithmic in N and linear in H. The following theorem states the

[3]The expected false positive error of the set $\mathcal{C}$, defined by the number of false positives divided by the size of $\mathcal{C}$, is $\epsilon/(1+\epsilon)$.

minimal memory requirement for achieving a specified false positive error.

Theorem 1: Given an expected normalized false positives, ϵ, the memory size U is minimized when $K = H/\log 2$ and $M = \log_2(N\epsilon^{-1}H^{-1})$ for a large H (say larger than 100).

The proof is based on minimizing the memory size directly, but the details are omitted due to space limitation. In fact, this memory optimization problem is essentially the same memory optimization problem in the design of Bloom filter [2]. A nice survey of Bloom filter and its network related applications is given in [3].

There is a trade-off between the memory requirement and the hash computations for achieving a fixed false positive error. Figure 1 shows the trade-off between M and U for the case where $N = 2^{32}$, $H = 1000$ in the lower bound case. The circles represent the optimal pair of (M, U) such that U is minimized. To achieve the same expected normalized false positive error ($\epsilon = 10^{-6}$ or $\epsilon = 10^{-3}$), we can in fact use just half of the optimal number of hashing tables with the price of increasing the memory size by about only 20%. This may be desirable when hash operations are considered expensive.

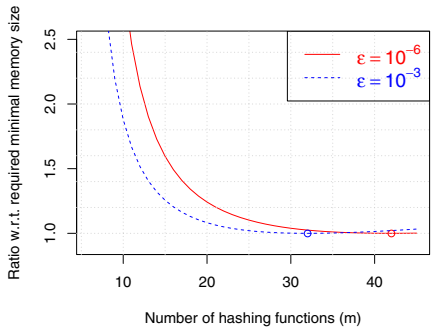


Fig. 1. Trade-off between the number of hashing table and memory size

B. Memory for heavy changer detection

For the (m, j)th bucket, let $y^{(1)}_{m,j}, y^{(2)}_{m,j}$ be the bucket values in interval 1 and 2 respectively, and let $y_{m,j} = y^{(2)}_{m,j} - y^{(1)}_{m,j}$ be the change in the bucket value. For the heavy changer case, a bucket is considered *heavy* iff $|y_{m,j}|$ crosses a pre-specified threshold t. When the values of non-heavy changers are negligible, unlike the heavy hitter case presented in the above, it is now possible that some positive changers and negative changers collide in the same bucket such that the bucket is *not heavy* (i.e., $|y_{m,j}|$ is less than t). Therefore, the outcome of the threshold test does not fully reflect the values of heavy keys, and there will be a false negative error in addition to the false positive error when using the intersections of heavy buckets to identify the heavy changers. To control the false negative error, a notion of *misses* has been introduced in [6] and [12] to refer to those non-heavy buckets, so that a key is included in the candidate set if it falls into at least $M - r$ heavy buckets, where r is the number of allowed misses. We refine this criterion by using an additional constraint: for a miss (i.e., a non-heavy bucket) to be considered legitimate, the bucket value in either $y^{(1)}_{m,j}$ of $y^{(2)}_{m,j}$ has to cross the threshold t. We found the inclusion of this refined criterion very useful in reducing the false positives. With the allowed r misses, the false positive rate will increase, and hence the memory requirement will increase. It is also clear that when the values of non-heavy hitters or changers become significant, both false negative and false positive rates will increase using the same hash array, and so does the memory requirement for a given false positive rate.

III. Sequential Hashing Scheme for Identifying Heavy Keys

To identify the heavy keys in a total of N keys using a single hash array, one has to enumerate the entire key space to see if each key falls into some heavy bucket in each of the tables in the hash array. Such an approach, however, is computationally expensive or even infeasible if the key space is very large.

In this section, we propose a general framework of using a multi-level hashing scheme for recovering H heavy elements in N keys when enumerating the entire key space becomes computationally prohibitive. The multi-level hashing scheme allows us to divide the original problem into much smaller sub-problems where the exhaustive search can be applied. We then focus on a special version of the general multi-level hashing scheme called *sequential hashing*, which has a few desirable properties. Lastly, we present a mathematical analysis of its complexity in terms of memory and computation, and discuss the design parameter optimization for memory and computation cost for a targeted false positive rate.

A. Multi-level hashing

To illustrate the general idea of multi-level hashing, for a key x with $n = \log_2 N$ bits, we first focus on identifying a sub-key of x with b bits that belongs to a heavy key. We assume b is sufficiently small (say 4, 8) such that enumeration of this sub-key space for the identification of the heavy sub-keys is now *trivial* using a hash array as described in Section II. Next, we combine the heavy sub-keys that have just been found with some remaining bits (say 2,4 bits) of the key to form a larger sub-key with more bits, say b' bits. Enumeration of this larger sub-key space (with b' bits) is now significantly reduced because the smaller sub-keys (with b bits) for heavy keys are already known. Therefore, we can again use a new hash array to identify the larger sub-keys of the heavy keys. Repeating the process, we can eventually discover the key values of the heavy keys in the original key space.

For easy enumeration, a similar idea is proposed in [12] that divides a large key into smaller words that are enumerated first. However, they choose to directly combine all enumerated words of the heavy keys in one step without more intermediate steps. This makes it not only more computationally expensive, but also less flexible on trading off between computation and space as will be described later.

B. Sequential Hashing Scheme

We now propose a sequential hashing scheme for identifying heavy keys which is a special version of the multi-level hashing scheme discussed above. Our sequential hashing scheme consists of two major steps: (1) *update* step, which includes the

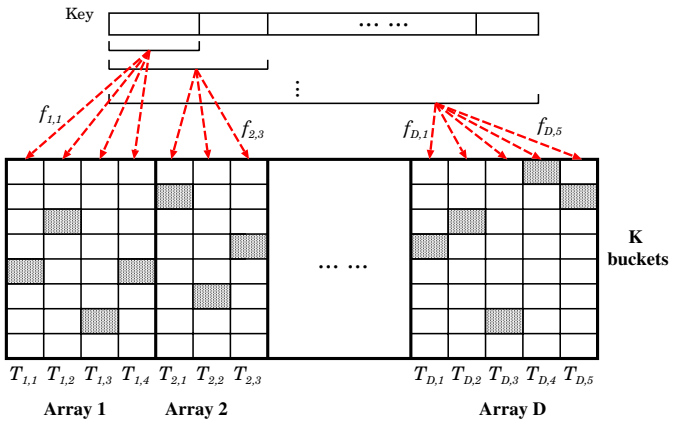


Fig. 2. Relationship between a key and the hash arrays in the sequential hashing scheme.

value of a key into the associated buckets of the hash arrays, and (2) *detection* step, which determines the set of heavy keys.

Figure 2 depicts the relationship between a key and the hash arrays in the sequential hashing scheme. We partition a key x into D words $w_1 w_2 \cdots, w_D$ such that each word w_i has b_i bits, where $1 \leq i \leq D$. We now consider the sub-key $w_1 \cdots w_i$, formed by the first i words of key x. Let $N_i = 2^{\sum_{r=1}^{i} b_r}$, and let $\mathcal{N}_i$ be the corresponding sub-key space $\{0, 1, \cdots, N_i - 1\}$, which contains all possible values of sub-key $w_1 \cdots w_i$. In each sub-key space $\mathcal{N}_i$, let $\mathcal{H}_i$ denote the set of sub-keys of those heavy keys in the original key space. Note that $\mathcal{H}_i$ is at most of size H. In addition, we construct a set of D hash arrays, in which the ith hash array corresponds to sub-key $w_1 \cdots w_i$ and contains M_i hash tables $\mathcal{T}_{i,1}, \cdots, \mathcal{T}_{i,M_i}$.

Algorithm 1 Update step

Input: a key x with value v
1: Partition key x into D words as $w_1 w_2 \cdots w_D$, where word w_i has b_i bits for $1 \leq i \leq D$
2: **for** i = 1 to D **do**
3: **for** j = 1 to M_i **do**
4: Increment the counter of bucket $f_{i,j}(w_1 \cdots w_i)$ in hash table $\mathcal{T}_{i,j}$ with value v

To begin with, Algorithm 1 outlines the update step. For each incoming key $x = w_1 \cdots w_D$ with value v, we associate the sub-key $w_1 \cdots w_i$ with hash function $f_{i,j}$ to bucket $f_{i,j}(w_1 \cdots w_i) \in \{1, \cdots, K\}$ in hash table $\mathcal{T}_{i,j}$, where $1 \leq i \leq D$, $1 \leq j \leq M_i$, and $1 \leq k \leq K$. We then increment the counter in the bucket with value v.

Algorithm 2 summarizes the detection step for the case of heavy hitter detection. The main idea is to decompose the original problem of finding H heavy keys into a sequence of D nested sub-problems, each of which determines a candidate set $\mathcal{C}_i$ from subspace $\mathcal{N}_i$ as an approximation of $\mathcal{H}_i$. We first identify $\mathcal{C}_1$ by searching for all values in $\mathcal{N}_1$ that have all their associated buckets in $\mathcal{T}_{1,1}, \cdots \mathcal{T}_{1,M_1}$ considered to be heavy, i.e., the counter of a bucket exceeds a pre-specified threshold. To determine $\mathcal{C}_i$, where $2 \leq i \leq D$, we first concatenate each sub-key $x' \in \mathcal{C}_{i-1}$ with an arbitrary word $w_i \in \{0, \cdots, 2^{b_i} - 1\}$ to form x''. We then include x'' into $\mathcal{C}_i$

Algorithm 2 Detection step

Inputs: hash tables $\{\mathcal{T}_{i,j}\}_{1 \leq i \leq D, 1 \leq j \leq M_i}$ with heavy buckets
Output: a set of heavy keys
1: Set $\mathcal{C}_0 = \{0\}$ and $\mathcal{C}_i = \phi$ for $1 \leq i \leq D$
2: **for** i = 1 to D **do**
3: **for all** $x' \in \mathcal{C}_{i-1}$ **do**
4: **for** w_i = 0 to $2^{b_i} - 1$ **do**
5: $x'' = x' \times 2^{b_i} + w_i$
6: Set $flag$ = TRUE
7: **for** j = 1 to M_i **do**
8: **if** bucket $f_{i,j}(x'')$ in $\mathcal{T}_{i,j}$ NOT heavy **then**
9: Set $flag$ = FALSE
10: Exit the for-loop of lines 7-10
11: **if** $flag$ == TRUE **then**
12: Add x'' to $\mathcal{C}_i$
13: return $\mathcal{C}_D$

if all its associated buckets in $\mathcal{T}_{i,1} \cdots \mathcal{T}_{i,M_i}$ are heavy (i.e., the variable *flag* remains TRUE). We continue this process and finally return the candidate set $\mathcal{C}_D$.

Note that Algorithm 2 is illustrated for heavy hitter detection. For heavy changer detection, we include r_i allowed misses for the ith hash array and modify Line 8 as follows, i.e., we set *flag* to FALSE if bucket $f_{i,j}(x'')$ is a non-legitimate miss, or the number of legitimate misses over the i hash array exceeds r_i (see Section II-B for details).

In the following, we present a mathematical complexity analysis of our sequential hashing scheme in terms of memory and computation, and discuss the design choice to achieve the most savings in both memory and computation for a targeted false positive rate. We first analyze the situation when the non-heavy keys have negligible contribution to the counter values, and then discuss how our result can be extended to the situation of significant non-heavy keys. We show that with the right design choice, our scheme can reduce the computation in the detection step from $\emptyset(N)$ (by enumerating all N keys) to $O(H \log_2 N)$ with very little increase in total memory. We also conduct complexity comparison between our scheme and the competing schemes, and show that our scheme is superior in terms of both memory and computation.

C. Mathematical complexity analysis when non-heavy keys are negligible

Assume the heavy keys are distributed randomly in the key space, then it can be shown that the expected size of $\mathcal{H}_i$ (i.e., the distinct first i words of H heavy keys) is

$$E|\mathcal{H}_i| \approx N_i \left[1 - \left(1 - \frac{1}{N_i}\right)^H\right] \approx H, \tag{4}$$

where the approximation holds when $N_i \gg H$[4]. When the non-heavy keys have negligible contribution to the counter values, the optimal value of K which minimizes the memory requirement is $K = \gamma^{-1} H$ with $\gamma = \log 2$, which is independent of the size of the key space. Therefore, we can choose the same number of buckets K for the hash tables in each hash array.

[4]In practice, this will be satisfied when $N_i \geq 64H$.

For the ith sub-problem, where $1 \le i \le D$, suppose that the expected number of false positives normalized by H is α_i for $1 \le i \le D-1$ and ϵ for $i = D$, i.e.,

$$E|\mathcal{C}_i| = H + \alpha_i H, \quad E|\mathcal{C}| = E|\mathcal{C}_D| = H + \epsilon H, \quad 1 \le i < D.$$

Therefore the expected number of keys to be enumerated for each sub-problem is 2^{b_1} for $i = 1$, and $(1+\alpha_{i-1})H2^{b_i}$ for $2 \le i \le D$. Since the complexity of each sub-problem is determined by the size of keys to be enumerated, it is now natural to let all the sub-problems have the same expected number of keys to be enumerated. This can be achieved by letting $\alpha_i = \alpha$, and dividing the whole key into D words such that

$$2^{b_1} = (1+\alpha)H2^b, \quad \text{and} \quad b_i = b, \quad 2 \le i \le D. \tag{5}$$

Under this setting, we now consider two main quantities for the complexity study when the non-heavy keys are negligible: update memory and recovery cost, and we list the results for other quantities in Table II. We are interested in how the complexity grows as a function of H and N.

1) Update Memory: By applying (3) to each sub-problem i (replacing N with $(1+\alpha)H$), $1 \le i \le D-1$, the required total number of hash tables with a size $K = \gamma^{-1}H$ is

$$\begin{aligned}\overline{M} &= \sum_{i=1}^{D-1} r\log_2((1+\alpha)H\alpha^{-1}H^{-1}) + r\log_2(N\epsilon^{-1}H^{-1}) \\ &= r\log_2\frac{N}{\epsilon H} + r(D-1)\log_2(1+\alpha^{-1}),\end{aligned} \tag{6}$$

where $r = -1/\log_2(1-e^{-\gamma})$. Notice that the first quantity in (6) is the total number of the tables required to recover the H heavy keys using a single random hash array by enumerating all the keys in the original space, for the same normalized false positive number ϵ. Therefore, the latter quantity in (6) is the additional number of tables required for the sequential hashing scheme, which decreases when α increases.

2) Detection Cost: We define the detection cost as the number of hash operations needed to recover all heavy keys. Since the number of keys to be enumerated is $(1+\alpha)H2^b$ for each sub-problem under our setting (5), and in the worst case, for each sub-key, we need to check all M_i tables to include or exclude it, the total hash computation required is

$$\text{Computation} \le (1+\alpha)2^b H\overline{M} = \gamma^{-1}(1+\alpha)2^b \times (\text{Memory}). \tag{7}$$

D. Design choices when non-heavy keys are negligible

Given a normalized false positive number ϵ, our sequential hashing scheme has two tuning parameters: α, the intermediate normalized false positives, and b the number of bits of each word except the first one. Notice that by (5), the number of total words D is a function of α, b since

$$\log_2(1+\alpha) + bD = \log_2(H^{-1}N). \tag{8}$$

Now we formulate the design problem as an optimization problem where we try to minimize both the memory increase and the computational cost, i.e., following (3) and (7), we want to

$$\text{minimize } (D-1)\log_2(1+\alpha^{-1}) \quad \text{and} \quad (1+\alpha)2^b,$$

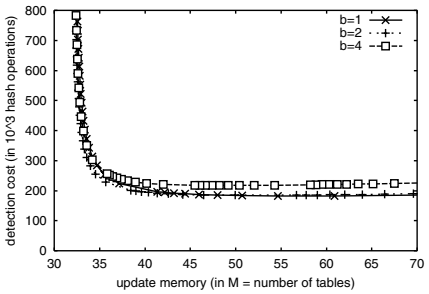


Fig. 3. Trade-off between update memory and detection cost with $N = 2^{32}$ and $H = 500$.

given the constraint (8) and $(1+\alpha)2^b \ge 64$ so that (4) will be satisfied. Notice that the computation is exponential in b, therefore we should let b small. For a fixed small b, if $\alpha = O(\log_2 N)$, then the memory increase is bounded by a constant and the computation is $O((\log_2 N)^2)$. If α is of $O(1)$, then the memory increase is $O(\log N)$ and the computation is $O(\log N)$ as well. For practical values of $\log_2 N$ (say 32 bits), we found that there is little difference in the memory increase when b is between 1 to 5 bits if we set $(1+\alpha)2^b \ge 64$ (the number of tables differ at most 2).

To understand the above results, Figure 3 illustrates how our sequential hashing scheme trades off between update memory and detection cost. Here, we evaluate the values of b for the case when non-heavy keys have negligible contribution to the counter values, and hence $\gamma = \log 2$. We assume that $N = 2^{32}$, $H = 500$ (and hence $K \approx 722$), $N_1 = 2^{16}$ (and hence $N_i \ge 64H$), $\epsilon = 0.2\%$. We then vary α to obtain the corresponding update memory (in terms of $\overline{M}$) and detection cost. As shown in the figure, when b = 1 or 2, a smaller detection cost is obtained as compared to b = 4, while the difference between b=1 and 2 is very small. For example, when b = 2 and α = 9, we have $M \approx 33$ (where M_1 = 4, $M_i = 2$ for $2 \le i \le D-1$, $M_D = 15$, and D = 9), while the detection cost is about 400K, which is twice the minimum detection cost achieved by larger update memory. Note that the number of tables in the lower-bound memory requirement is $\log_2 \frac{N}{\epsilon H}$ = 32, where the heavy-key detection is done by enumeration of the entire key space. Thus, with only one extra table, we can recover all heavy keys with manageable detection cost.

E. Extension in the presence of significant non-heavy keys

In Section II-A, we show that the required memory for detecting H heavy hitters in N keys is at least $O(H\log(N/\epsilon H))$ for a given normalized false positive number ϵ. In the presence of significant non-heavy keys, [12] introduced the so-called *ϵ-approximate* heavy keys and non-heavy keys (see [6]) and studied its false negatives and false positives respectively. By plugging in ϵ in Theorem 2 of [12], we can in fact show that the memory requirement can achieve $O(H\log(N/\epsilon H))$ for the normalized false positive and false negative number ϵ. In this case, the complexity results in (3) and (7) still hold but with different values of r in (6). Therefore the design choice studied in Section III-D also applies here.

F. Comparison with other schemes

There are two existing schemes that are mostly related to this work for heavy change detection, the *Deltoids* approach proposed in [4], and the *reversible-sketch* approach proposed in [12]. Table II compares complexities of these two schemes with that of sequential hashing. Suppose that we set $\epsilon H = \Theta(1)$, meaning that the number of false positives is controlled within a constant factor. It should be noted that as compared to the other two approaches, our sequential hashing scheme requires less memory in general to achieve a given fixed false positive rate, since, as shown above, we need only $\Theta(1)$ extra hash tables with respect to the lower-bound memory requirement. In addition, for the detection step, our scheme has the same $\log_2 N$ complexity as Deltoids in terms of both memory and computation, and is much cheaper than Reversible sketch, whose complexity is sub-linear of the original key space (on the order $O(N^{1/\log\log N}\log\log N)$ in memory and $O(N^{3/\log\log N}\log\log N)$ in computation). In addition, both Reversible sketch and Deltoids require a verification step to reduce the number of false positives, but our scheme is simpler since the verification step is incorporated into the last hash array which maps the keys from the original key space. Later in Section V, we shall demonstrate our advantages using experimental studies of real traces.

IV. Estimating Values of Heavy Keys Using Linear Regression

In this section, we present a maximum likelihood based method for estimating the heavy key values under a linear regression model. This estimation can be useful for two reasons. First, when the number of heavy keys are large, it is important to provide some guidance so that one can look at the most important ones first. Second, using the estimated values, we can reduce the false positive rate by eliminating those non-heavy elements included in the set. It is important to realize that in the sequential hashing detection algorithms that we presented earlier, we did not fully utilize the information in the counter values and all we did is a threshold test. In the experimental studies in Section VI, we will show that by using estimation we can reduce the false positive rate significantly at the expense of only a small increase in the false negative rate.

Given a candidate set $\mathcal{C}$ of the heavy keys, let V be a vector of length $|\mathcal{C}|$ representing their values, and let Y be a vector of length L representing the counter values (or change in counter values for heavy changer), for those buckets that contain at least one candidate key. Now we can write

$$Y = AV + \delta, \tag{9}$$

where A is a $L \times |\mathcal{C}|$ matrix whose columns represent how each candidate is mapped to the counter buckets that Y represents, and δ represents the contribution from the remaining non-heavy keys to Y.

A. Heavy Hitter Estimation

Based on the empirical studies of real traces, for heavy hitter estimation, we find that the distribution of δ is well approximated by a Weibull distribution with mean θ and shape parameter β, i.e., $(\delta/\theta)^\beta \sim \text{Exp}(1)$, where Exp(1) stands for the exponential distribution with mean 1. Figure 4 shows Weibull-QQplot of the observed δ distribution for the detection of at most 500 heavy hitters in a real trace studied later in Section V, using a hash array with $M = 33$ tables and $K = 722$ buckets per table. It is easy to see that the Weibull distribution gives an excellent approximation as a straight line indicates an exact Weibull distribution.

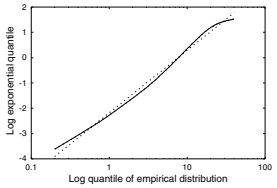


Fig. 4. The error distribution of bucket values in a hash array for heavy hitter detection with $K = 722$, $M = 33$, and 500 heavy hitters (see Experiment 1 in Section V). The dotted line indicates the true Weibull distribution.

When the shape parameter is 1, a Weibull reduces to an exponential distribution. In this case, the maximum likelihood estimate $\hat{V}_{MLE}$ is equivalent to solving the following linear programming problem with respect to V

$$\text{maximize} \sum_{l=1}^{L} A_l V \quad \text{subject to} \quad (y_l - A_l V) \geq 0, \tag{10}$$

where y_l is the l-th element of Y and A_l is the l-th row of A.

A computationally cheaper estimator of V, the *countmin* estimator, has been proposed in [4]. The *countmin* estimator for the value of a candidate heavy hitter key is essentially the minimum of all bucket values of y that contain the candidate key. It is straightforward to show that if all the heavy buckets contains exactly one heavy hitter, the maximum likelihood estimator $\hat{V}_{MLE}$ reduces to the *countmin* estimator $\hat{V}_{min}$. However, from Lemma 1, this is not true and only around 70% of the heavy buckets contain exactly one heavy hitter when $\gamma = \ln 2$ and the candidate size is close to H. It can be shown that both $\hat{V}_{min}$ and $\hat{V}_{MLE}$ have some small positive bias, which is approximately

$$\text{bias} \;\approx\; E\left[\min_{1\leq m\leq M} \tilde{Y}_m\right],$$

where $\tilde{Y}_m$ is a non-heavy bucket in table $1 \leq m \leq M$. Because non-heavy buckets are abundant (50% when the candidate size is close to H with $\gamma = \ln 2$ by using Lemma 1), the bias can be approximated accurately using a non-parametric method by obtaining many samples of M non-heavy buckets and then taking the empirical mean of the minimum of each sample.

For a general Weibull error model, although it is straightforward to express maximum likelihood estimate also as a solution to a constrained optimization problem, we have not found an efficient solver for the constrained optimization problem at the time of the writing. Based on the experimental results that we will present in Section VI, we have found that the solution from the linear programming problem in (10) with bias correction gives very accurate estimates for

TABLE II
COMPLEXITY COMPARISON BETWEEN REVERSIBLE SKETCH, DELTOIDS, AND SEQUENTIAL HASHING (THE FIRST TWO ROWS ARE DERIVED FROM [12])

.

	Update step			Detection step	
	memory	memory accesses	operations	memory	operations
Reversible Sketch	$\Theta(\frac{(\log N)^{\Theta(1)}}{\log\log N})$	$\Theta(\frac{(\log N)}{\log\log N})$	$\Theta(\log N)$	$\Theta(N^{\frac{1}{\log\log N}} \cdot \log\log N)$	$O(HN^{\frac{3}{\log\log N}} \cdot \log\log N)$
Deltoids	$\Theta(H \log N)$	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(H \log N)$	$O(H \log N)$
Sequential Hashing	$\Theta(H \log \frac{N}{\epsilon H})$	$\Theta(\log \frac{N}{\epsilon H})$	$\Theta(\log \frac{N}{\epsilon H})$	$\Theta(H \log \frac{N}{\epsilon H} + \alpha H)$	$\Theta(H \log \frac{N}{\epsilon H} + (D-1)\alpha H 2^b)$

our trace study. In addition to the *countmin* estimator, a least-square estimator of the heavy hitters was proposed in [9] for the linear regression problem in (9), which can be viewed as the maximum likelihood estimate when the error distribution follows a normal distribution. In Section VI, we shall compare with least square method and show that the estimates obtained from solving the linear programming problem are superior.

B. Heavy Changer Estimation

Based on the empirical studies of real traces, we find the distribution of δ in the heavy changer case is well approximated by a double exponential distribution. In such case, the maximum likelihood estimate $\hat{V}_{MLE}$ for the linear regression problem in (9) can be obtained from solving the following L_1-regression problem:

$$\hat{V}_{MLE} \quad \text{minimizes} \quad \sum_{l=1}^{L} |y_l - a - A_l V|, \tag{11}$$

which can be done using standard packages. Interestingly, when all the heavy buckets contain exactly one heavy hitter, then $\hat{V}_{MLE}$ corresponds to the median estimator, similar to that proposed in [8]. The median estimator for the value of a candidate key is the median of all bucket values of y that contain the candidate key.

V. EXPERIMENTAL STUDIES

When non-heavy key values are no longer negligible as is often the case in real traces, some of the non-heavy buckets will be considered to be heavy, leading to more false positives. Therefore, we need additional memory to counter this noise effect. In this section, we use trace-driven simulation to study various choices of parameters so as to tolerate the presence of noise.

Using Internet traces captured from various sources, we evaluate our sequential hashing scheme in identifying heavy keys (i.e., heavy hitters or heavy changers) whose values (i.e., data volumes or change of data volumes) exceed a pre-specified threshold. Any key whose value is below the threshold is considered to be a non-heavy key and treated as noise. Our scheme is compared with Deltoids [6] using its publicized software. In addition, we analyze the improvements when our scheme is coupled with linear regression presented in Section IV.

In summary, using a memory-efficient data structure, we show that our sequential hashing scheme provides much more accurate heavy hitter and heavy changer detection than does Deltoids in the presence of noise. With linear regression, the accuracy of our scheme is further improved. Moreover, we show that our scheme allows fast detection and supports large key space.

We cannot compare our scheme directly with Reversible sketch [12] as its source code is not publicized. However, in our evaluation, our scheme uses much fewer counters for evaluation as compared to the evaluation in [12], while still providing accurate estimates.

Traces: The results presented here are based on an one-hour uni-directional trace from NLANR [11]. The trace contains about 50 GB of Internet traffic collected from 10:00pm to 11:00pm on June 1, 2004 at an OC-192 link connecting between Indianapolis and Kansas city in the United States. The huge volume of collected traffic allows us to demonstrate the effectiveness of our sequential hashing scheme in a high-speed network. We repeat our evaluation using the NLANR traces collected from the same source but at other times as well as using private traces collected at an OC-48 link of an ISP, and similar results are observed.

We divide the one-hour NLANR trace into six 10-minute intervals. For heavy hitter detection, we identify the source IPs whose data volume exceeds a threshold in each interval. We then average the results across all intervals. On the other hand, for heavy-changer detection, we identify the source IPs whose absolute change of data volume is above a threshold in each pair of adjacent intervals. We then average the results across all pairs of adjacent intervals.

Experiment Setup: Unless otherwise stated, our discussion focuses on the 32-bit key space based on source IP addresses. However, we also experiment the 64-bit key space defined by source-destination IP addresses.

In practice, given a user-specified threshold, one can approximate the worst-case number H of heavy keys over a data stream. For example, the number of heavy hitters that exceeds 1% of the traffic is at most 100 (see [10] for a more detailed account of these descriptions). In our experiments, we assume H = 500 and vary K and M such that the size of K and M are sufficient for identifying at most H heavy keys.

For evaluation purpose, we select different thresholds, each of which corresponds to a true number of heavy keys. We therefore maintain a baseline structure that keeps track of the per-key data volume for such threshold selection. The baseline structure is also used for assessing the accuracy of finding heavy keys.

Metrics: We are mainly interested in two accuracy metrics: (1) *false positive rate*, defined as ratio of the number of non-heavy keys to the number of keys returned by the sequential hashing scheme, and (2) *false negative rate*, defined as the number of true heavy keys that are not returned by the sequential hashing scheme.

Experiment 1 (Analysis of finding heavy hitters (without linear regression)): To counter the noise effect, we need additional memory by increasing K and/or M for successful heavy key detection. Thus, we study the impact with different choices of K and M. We begin our analysis by first excluding linear regression described in Section IV.

As shown in Section IV, when H = 500, then we choose K = 722 (with 9 hash arrays where $K = \frac{H}{\ln 2}$) and M = 33 (where $M_1 = 4$, $M_2 = \cdots = M_8 = 2$, and $M_9 = 15$) for our sequential hashing scheme. Here, we increase the memory by 50% and 100% by using different values of K and M shown in Table III (where each counter is assumed to be of size 4 bytes).

TABLE III
CONFIGURATIONS OF K AND M.

K	M	$(M_1, M_{2\leq i\leq 8}, M_9)$	# of counters (Memory Size)
722	33	(4, 2, 15)	23826 (93 KB)
722	50	(6, 3, 23)	36100 (141 KB)
722	66	(8, 4, 30)	47652 (186 KB)
1083	33	(4, 2, 15)	35739 (140 KB)
1444	33	(4, 2, 15)	47652 (186 KB)

Figure 5 shows the false positive rate of finding heavy hitters using the sequential hashing scheme (note that since every bucket that contains heavy hitters must be a heavy bucket, there is no false negative). With the original noise-free configuration K = 722 and M = 33, the false positive rate can be as high as 32%. However, by increasing the number of counters, we can reduce the false positive rate significantly to less than 6% by increasing K by 50% (for M = 33 and K = 1083) and further to less than 3% by doubling K (for M = 33 and K = 1444).

In the presence of noise, we note that increasing K is more advantageous than increasing M. Intuitively, as K increases, the values of non-heavy keys are distributed across more buckets. Thus, a bucket that contains only non-heavy keys is less likely to become a heavy bucket, leading to a reduced false positive rate. Also, increasing M is less desirable in practice because it increases the number of hash operations needed to record keys into the hash arrays.

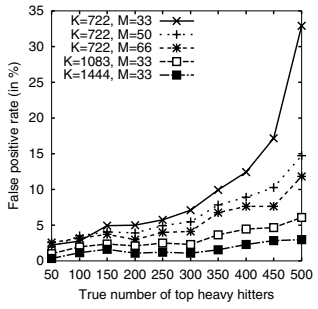


Fig. 5. False positive rate of finding heavy hitters using sequential hashing scheme in Experiment 1.

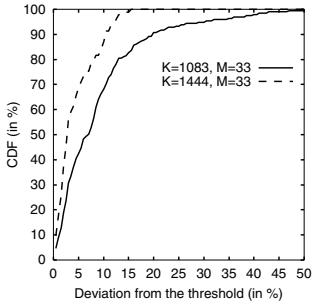
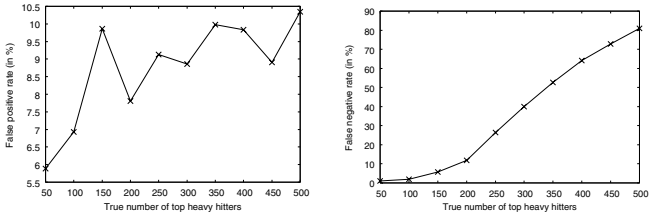


Fig. 6. Deviations of false positives from the threshold for finding the top 500 heavy hitters in Experiment 1.

To further examine the values of the false positives, Figure 6 depicts the percentage of deviations of these values with respect to the threshold for the case of finding the top heavy hitters using the configurations with M = 33 and K = 1083 and 1444. In fact, most of the false positives do not actually deviate much from the threshold. For instance, the proportion of false positives that have values within 15% of the threshold is more than 80% when M = 33 and K = 1083, and achieves 100% when M = 33 and K = 1444. It shows that the heavy hitter candidates returned from the sequential hashing scheme can effectively approximate the set of true heavy hitters.

We now compare our scheme with Deltoids using its publicized software. Here, we set the number of hash tables and the number of buckets in each hash table to be 4 and 361, respectively. Since each of its buckets is associated with $\log_2 N + 1 = 33$ counters, its total number of counters is no less than all of our configurations.

Figure 7 shows the accuracy of using Deltoids to identify heavy hitters. While Deltoids has less than 10% of false positive rate, its false negative rate is significantly high (up to 80%) as more heavy hitters need to be identified, meaning that many true heavy hitters evade detection. This shows that with the same or even less amount of memory, our sequential hashing scheme provides a much more accurate heavy hitter detection than does Deltoids.



(a) False positive rate (b) False negative rate

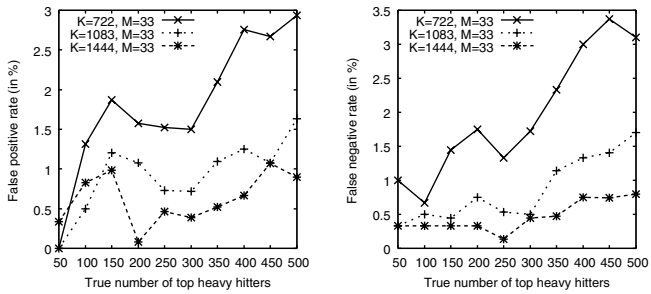
Fig. 7. Accuracy of Deltoids in finding heavy hitters in Experiment 1.

Experiment 2 (Analysis of finding heavy hitters (with linear regression)): Here, we analyze how the heavy hitter detection benefits from linear regression presented in Section IV. As Experiment 1 shows that increasing K outperforms increasing M, we focus on the configurations with M = 33, and K = 722, 1083, and 1444.

Figure 8 shows the accuracy of finding heavy hitters when we couple our sequential hashing scheme with linear regression. Referring to Figure 5 in Experiment 1, for K = 722 and M = 33, the false positive rate for identifying 500 heavy hitters is almost 32% without linear regression. However, linear regression reduces this false positive rate to less than 3%, while introducing a false negative rate 3.1% (i.e., the total error rate is about 6%). Also, for K = 1083 and M = 33, the false positive rate for identifying 500 heavy hitters is also about 6% when no linear regression is used. This shows that linear regression can reduce the amount of memory required to achieve the same total error rate.

In terms of the accuracy of estimation, we show that linear regression provides a better estimate of values of heavy hitters than does the least-square method proposed in [9]. Here, we consider the following two error measures:

$$Err_1 = \frac{1}{|\mathcal{C}|}\sum_{x\in\mathcal{C}} |v_x^{est} - v_x|,$$
$$Err_2 = \sqrt{\frac{1}{|\mathcal{C}|}\sum_{x\in\mathcal{C}} (v_x^{est} - v_x)^2},$$



(a) False positive rate (b) False negative rate

Fig. 8. Accuracy of finding heavy hitters with linear regression in Experiment 2.

where v_x and v_x^{est} are respectively the true value and the corresponding estimate of key x, and $\mathcal{C}$ is the final candidate set returned from the sequential hashing scheme.

Table IV shows the error measures (in unit MB) of both linear and least-square regressions in estimating the data volumes of the top 500 heavy hitters. It shows that linear regression always outperforms least-square regression in both error measures. We have also tried other types of error measures and linear regression still provides better results.

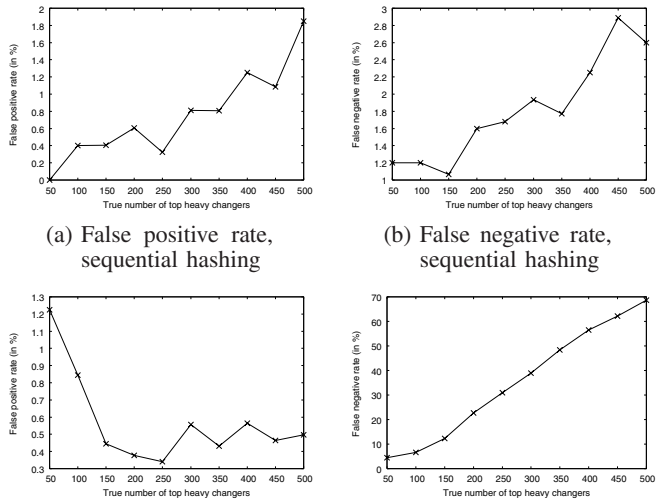
TABLE IV
ACCURACIES OF LINEAR AND LEAST-SQUARE REGRESSIONS.

Configuration	Linear		Least-square	
	Err_1	Err_2	Err_1	Err_2
K=722, M=33	0.7788	0.9929	2.1593	2.8382
K=1083, M=33	0.3890	0.5357	0.7567	0.9754
K=1444, M=33	0.2145	0.3030	0.3813	0.4971

Experiment 3 (Accuracy of finding heavy changers): We now compare both our sequential hashing scheme (with linear regression) and Deltoids in heavy changer detection. Since the positive and negative changes can cancel each other, some of the buckets that contain heavy changers will not be identified as heavy buckets. Therefore, we need even more memory to mitigate this impact. Here, for our sequential hashing scheme, we consider the configuration with K = 1444 and M = 33, while for Deltoids, we use the same configuration as in Experiment 1. Thus, both approaches are allocated with the same number of counters.

Figure 9 depicts the accuracy of finding heavy changers both schemes. While Deltoids only has at most 1.2% false positive rate, its false negative rate can be as high as 70%. On the other hand, with the same number of counters, our sequential hashing scheme bounds the false positive and negative rates within 3%.

Experiment 4 (Analysis of 64-bit key space): We further evaluate our sequential hashing scheme using the 64-bit source-destination IP pair as the key space. Here, we set M = 66. For the case of finding top 500 heavy hitters, the false positive and negative rates are 1.6% and 0.6%, respectively. However, for the case of finding top 500 heavy changers, we set K = 1800 to further counter the cancellation of positive and negative changes. The false positive and negative ratios are 0.6% and 2.9%, respectively.



(a) False positive rate, sequential hashing (b) False negative rate, sequential hashing

(c) False positive rate, Deltoids (d) False negative rate, Deltoids

Fig. 9. Experiment 3: Accuracy of finding heavy changers.

VI. CONCLUSION

In this paper, we consider how to identify the keys (e.g., IPs or flows) that have large data volume or large volume change in a high speed network. Given the infeasibility of tracking all keys, we first derive the lower-bound memory requirement for recovering heavy keys with respect to a fixed false positive rate. Next we propose a sequential hashing scheme that can achieve accurate and fast identification of heavy keys. between the memory usage and In addition, we propose a linear-regression-based method to accurately estimate the values of heavy keys and to further improve the accuracy of heavy-key identification. Finally, we show via extensive trace-driven simulation that our scheme is more robust in identifying heavy keys as compared to the Deltoids approach.

REFERENCES

[1] Abhishek Kumar and Jun Xu and Jia Wang and Oliver Spatschek and Li Li. Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement. In *Proc. of IEEE INFOCOM*, Mar. 2004.
[2] B. Bloom. Space/time trade-offs in hashing coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: a survey. *Internet Mathematics*, 1(4):485:509, 2003.
[4] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, 2003.
[5] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Trans. on Computer Systems*, 21(3):270–313, Aug 2003.
[6] G. Cormode and S. Muthukrishnan. What's New: Finding Significant Differences in Network Data Streams. In *Proc. of IEEE INFOCOM*, Mar. 2004.
[7] M. Kodialam, T. Lakshman, and S. Mohanty. Runs bAsed Traffic Estimator (RATE): A Simple, Memory Efficient Scheme for Per-Flow Rate Estimation. In *Proc. of IEEE INFOCOM*, Mar. 2004.
[8] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Internet Measurement Conference*, 2003.
[9] G. M. Lee, H. Liu, Y. Yoon, and Y. Zhang. Improving sketch reconstruction accuracy using linear least squares method. In *Internet Measurement Conference*, 2005.
[10] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. VLDB*, 2002.
[11] NLANR. Abilene-III Trace Data. http://pma.nlanr.net/Special/ipls3.html.
[12] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. Dinda, M. Kao, and G. Memik. Reverse hashing for high-speed network monitoring: algorithms, evaluation, and applications. In *IEEE INFOCOM*, Barcelona, Spain, April 2006.