Heavy Hitters in Streams and Sliding Windows

Ran Ben-Basat Gil Einziger Roy Friedman Yaron Kassner
Department of Computer Science, Technion, Haifa 32000, Israel
{sran,gilga,roy,kassnery}@cs.technion.ac.il

***Abstract*—Identifying heavy hitter flows is a fundamental problem in various network domains. The well established method of using sketches to approximate flow statistics suffers from space inefficiencies. In addition, flow arrival rates are dynamic, thus keeping track of the most recent heavy hitters poses a challenge. Sliding window approximations address this problem, reducing space at the cost of increasing point query time.**

This paper presents two novel algorithms for identifying heavy hitters in streams and sliding windows. Both algorithms use statically allocated memory and support constant time point queries. For sliding windows, this is an asymptotic improvement over previous work. We also demonstrate reduced memory requirements of up to 85% in streams and 66% in sliding windows over synthetic and real Internet packet traces.

I. Introduction

Per flow monitoring capabilities are desired for network algorithms in various domains such as load balancing [9], routing, fairness [17], intrusion detection [22], [15], caching [10] and policy enforcement [23]. However, with ever increasing line speeds, it is difficult to implement such capabilities in networking devices. SRAM can keep up with line speed, but its limited capacity cannot accommodate all the flows. In contrast, DRAM can accommodate all flows, but cannot be read at line speed. Thus, algorithms that perform decisions on a per-packet basis may not be able to work with DRAM.

Previous approaches suggested replacing counters with small estimators, trading accuracy for reduced space. Such space reduction enables fitting more counters into SRAM [11], [24]. Alas, the potential benefit for further improvement with estimators is limited, as their suggested counter size are already 8-12 bits. Consequently, further improvements in line rates might render tracking of all flows infeasible.

This suggests that significant future enhancements should be explored through other methods. For example, traditional counting *sketches* such as *Count Min Sketch (CMS)* [6] and multi stage filters [13] do not require maintaining a counter per flow and can be deployed using compressed counters; indeed, sketches were successfully implemented in hardware. However, we argue that better alternatives exist.

Counter based heavy hitters algorithms were developed over the years in domains such as big data analysis and databases. These algorithms are based on hash-tables and were shown to be more space efficient than sketches, both empirically and asymptotically. Out of these algorithms, *Space-Saving* [21] is considered state-of-the-art [5]. Unfortunately, it is difficult to implement Space-Saving in network devices, as it uses dynamic memory allocation and pointers.

In addition, as more and more packets and flows pass through the router, every counting mechanism is eventually going to run out of capacity. Furthermore, as time passes, different elements become heavy hitters, and there is a need to keep track of the most recent heavy hitters. Therefore, a sliding window approach is attractive. However, in that case, previous works do not support constant time point queries. Such queries are very important for networking devices motivating the search for new solutions.

Our Contributions: We start by introducing *Compact Space-Saving (CSS)*, a complete redesign of the Space-Saving algorithm, which is considered the best for both the $top-k$ and heavy hitters problems. Unlike Space-Saving, CSS does not require pointers and uses statically allocated memory. We demonstrate a reduction of up to 85% compared to the original implementation of Space-Saving, which works in $O(1)$ with high probability (w.h.p.), and up to 75% compared to the heap implementation of Space-Saving, which works in logarithmic time complexity. This means that by using CSS, one can allocate 4-7 times as many counters as with the alternatives with the same amount of memory.

Next, we introduce *Window Compact Space-Saving (WCSS)*, an extension of CSS that monitors flows over a sliding window. Both CSS and WCSS operate at $O(1)$ complexity and limit themselves to statically allocated memory without using pointers. Moreover, to the best of our knowledge, WCSS is the first algorithm that supports point queries in constant time complexity under the sliding window model. This capability is significant, as it allows switches, routers and middle-boxes to quickly obtain a frequency estimation of an arbitrary flow in real time, e.g., on each packet arrival.

Finally, we show a method that further reduces the space requirement of our algorithms by storing fingerprints of identifiers instead of actual identifiers. In this case, the approximation guarantee is identical to that of sketches, yet we demonstrate space reduction of up to 7 times compared to them.

Paper roadmap: We start by surveying related works in Section II. Formal definitions and problem statements are given in Section III. Our improved Space-Saving implementation, including its analysis and optimizations, is presented in IV. Our solution for the heavy hitters problem over a sliding window, which relies on our Space-Saving implementation as a black-box, appears in Section V. A performance study of the memory consumption of our Space-Saving implementation vs. previous works is reported in Section VI. We conclude with a discussion in Section VII.

II. Related Work

A. Sketch Based Techniques

For many problems, *Counting Sketches* such as *Multi Stage Filters* [13] and *Count Min Sketch* [6] are a popular solution. These algorithms offer the following guarantee: for a stream of size N, their frequency estimation is correct up to an additive $N\epsilon$-error, only with probability of at least $1-\delta$.

These methods are typically optimized for hardware implementation as they use statically allocated memory and do not store explicit flow identifiers. Therefore, these methods can mainly answer point queries. That is, given a flow identifier, the sketching method generates an estimation for that flow. Despite these limitations, many networking applications found this approach useful [4], [6], [8], [13], [14], [20], [21].

B. Counter Based Techniques

Counter based techniques are typically not tailored for hardware and therefore do not limit themselves to point queries or statically allocated memory. These methods usually use hash tables to store the explicit keys of frequent elements and can also answer more complex queries such as "what are the frequent elements?" on demand.

These methods are typically analyzed only with respect to the number of items they store, disregarding other implementation overheads. Famous examples include *Space-Saving* [21], *Lossy counting (LC)* [20] and *Frequent* [7], [18].

C. Sliding Window Algorithms

Related work on heavy hitters over sliding windows is summarized in Table 4. First to study this problem were Arasu and Manku [2]. Their algorithm supports queries for finding the frequent elements and their estimated frequency. They provided an algorithm that uses $O\left(\epsilon \log^2 \frac{1}{\epsilon}\right)$ counters and allows $O\left(\epsilon \log \frac{1}{\epsilon}\right)$ time for queries and updates. The result was improved by Hung and Ting [19], who reduced space requirements and update time to $O\left(\frac{1}{\epsilon}\right)$. Finally, Hung et al. [16] gave an improved algorithm with constant *update time*. Since the supported queries require outputting all $O(\frac{1}{\epsilon})$ high-frequency elements, their runtime is essentially optimal. However, their data structure does not support constant time *point-queries*, i.e., estimation of a specific element frequency. Additionally, the above algorithms use dynamically allocated memory.

III. Preliminaries

Let S be a *stream* of *items* over a universe U of *elements*. That is, we refer to each instance of an element from U appearing in S as an item. Denote by $[r]$ the set $\{0,1,...,r-1\}$, by N the number of items in S, and by ω the number of bits required to represent an item in S, i.e. $\omega \triangleq \lceil \log |U| \rceil$. We define the *frequency* of an element x in S as the number of items corresponding to x in S, and denote this quantity by f_x. We further denote by f_x^W the frequency of x in the *last* W elements of S. See Table I for a summary of these notations. An approximate counting algorithm supports two operations:

- ADD(x) – append x to S.
- QUERY(x) – return an estimation $\widehat{f_x}$ for f_x.

Symbol	Meaning
S	stream
N	number of elements in the stream
U	the universe of elements
$[r]$	the set $\{0,1,...,r-1\}$
ω	number of bits required to represent an item in S
f_x	the frequency of an element x in S
f_x^W	the frequency of x in the last W elements of S
$H(x,y,z)$	memory of a hash table with x items, key size y bits and value size z
nchains(x,y,z)	number of chains in a hash table with x items, key size y bits and value size z

Table I
LIST OF SYMBOLS

We also consider an extension of this problem for the *sliding window* model, where QUERY(x) is required to estimate f_x^W rather than f_x.

Problem definitions: We consider the following variants of the approximate counting and heavy hitters problems:

- ϵ-**COUNTING**: $Query(x)$ returns an estimation $\widehat{f_x}$ that satisfies $f_x \leq \widehat{f_x} \leq f_x + N\epsilon$.
- (ϵ,δ)-**COUNTING**: $Query(x)$ returns an estimation f_x that satisfies $f_x \leq \widehat{f_x} \leq f_x + N\epsilon$, with probability of at least $1-\delta$.
- (W,ϵ)-**WINDOW COUNTING**: QUERY(x) returns an estimation $\widehat{f_x^W}$ such that $f_x^W \leq \widehat{f_x^W} \leq f_x^W + W\epsilon$.
- (W,ϵ,δ)-**WINDOW COUNTING**: $Query(x)$ returns an estimation $\widehat{f_x^W}$ such that $f_x^W \leq \widehat{f_x^W} \leq f_x^W + W\epsilon$, with probability of at least $1-\delta$.
- (W,θ,ϵ)- **HEAVY HITTERS**: $Query(x)$ returns a set of elements $H \subset U$, such that for every element x, $f_x^W \geq \theta \cdot W \implies x \in S$, and $f_x^W < (\theta-\epsilon)\cdot W \implies x \notin S$.

W is assumed to be known in advance and does not change throughout the stream.

Our algorithms use TinyTable [12], which offers a space efficient and pointer-less chain based hash table. We denote by $H(x,y,z)$ the memory requirement for storing a hash table with at most x items whose key size is y bits and their value size is z bits. We denote by nchains(x,y,z) the number of chains in a table with parameters as defined above.

IV. Frequency Estimation over Streams

In this section, we consider the problem of identifying heavy hitters in a stream. We start by describing the Space-Saving algorithm in Section IV-A. Section IV-B describes *Compact Space-Saving* (CSS), a complete redesign of Space-Saving, which reduces the memory consumption by up to 85%. While CSS solves the ϵ-COUNTING problem, Section IV-G shows how to further optimize space consumption for the weaker (ϵ,δ)-COUNTING problem.

A. Space-Saving

For self-containment, we start by describing the Space-Saving algorithm [21] that solves the ϵ-COUNTING problem and can be also used to solve the heavy hitters and the top-k problems. Schematically, Space-Saving allocates a fixed number of counters; whenever an item arrives, Space-Saving increments its counter. If the arriving item does not have

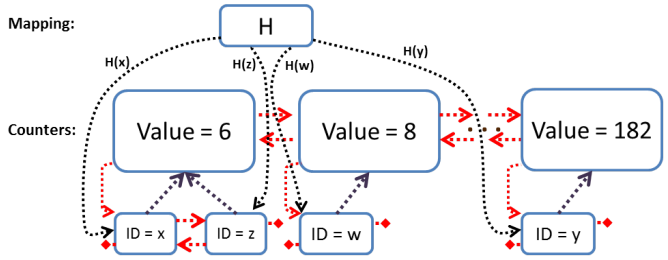


Figure 1. A Stream-Summary data structure example in the Space-Saving algorithm [21]. The structure shows that x and z both arrived 6 times, w has arrived 8 times while the highest frequency item y has arrived 182 times. The data structure operates in $O(1)$ time, but requires many pointers (the dotted arrows).

Algorithm 1 Space-Saving

Initialization: $C = \emptyset$

1: **function** ADD(Item x)
2: **if** $x \in C$ **then**
3: Increment c_x
4: **else**
5: **if** $|C| < M$ **then**
6: $c_x = 1$
7: $C = C \cup \{x\}$
8: **else**
9: $m = \text{argmin}_{y \in C} c_y$
10: $C = C \setminus \{m\} \cup \{x\}$
11: $c_x \leftarrow c_m$
12: Increment c_x
13: **function** QUERY(Item x)
14: **if** $x \in C$ **then return** c_x
15: **else return** $\min_{y \in C} c_y$

an allocated counter, this item inherits the minimal value counter while the previously associated item is "forgotten". Algorithm 1 provides a high level pseudo code of Space-Saving without implementation details. The code utilizes M counters and denotes the current counter-set by C.

Space-Saving operates in $O(1)$ hash-table operations per update and requires the optimal number of $O(\frac{1}{\varepsilon})$ counters. It satisfies the following:

Theorem 4.1: ([21]) Using $\frac{1}{\epsilon}$ counters, Space-Saving answers $Query(x)$ in $O(1)$ time such that $f_x \leq \widehat{f_x} \leq f_x + N\epsilon$.

However, a bit level analysis of Space-Saving reveals much room for improvement, as it employs a large number of pointers to keep operations O(1). Figure 1 illustrates *Stream-Summary*, one of the key data structures in Space-Saving. Stream-Summary keeps a group of counters for each counter value as a doubly linked list. It also stores the groups in a doubly linked list in a sorted order. Additionally, it has pointers from each counter to its group. The excessive use of pointers, as shown by the figure, requires large amounts of space we save by reverting to a pointer-less data structure. For example, consider three elements, X, Y and Z which appear with different frequencies. Counting the hash-chain pointers, omitted from the figure for clarity, Stream-Summary contains no less than 27 pointers ! Thus, the fast operation time of Space-Saving comes with significant overheads. We refer to this implementation of Space-Saving as **SSL**. Previous works addressed this inefficiency and suggested *Space-Saving with a Heap (SSH)*, a heap based implementation that requires less space, but operates in logarithmic time complexity.

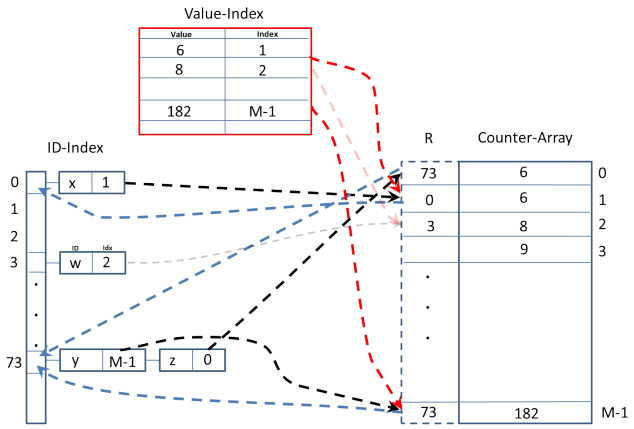


Figure 2. Data structures used by our CSS algorithm, where the algorithm state is the same as in Figure 1. The ID-Index table shows that counter 1 is allocated for x, item y was allocated the largest counter $(M-1)$, and z is tracked by counter 0. Value-Index shows that the highest index counter with value 6 is 1, and $\mathbf{R}$ allows us to find chain 73, in which we find the minimal item by checking it points to counter 0. We emphasize that arrows used in the figure are not memory pointers, but array indices.

B. Compact Space-Saving (CSS)

We now describe *Compact Space-Saving (CSS)*, a complete redesign of Space-Saving, which requires considerably less space than previous alternatives, allowing item additions and point queries in $O(1)$ time (w.h.p.). Moreover, CSS operates on statically allocated memory, does not require pointers, and is therefore better suited for hardware implementation. CSS uses the following data structures, as shown in Figure 2:

- **Counter-Array:** An array of $M \triangleq \epsilon^{-1}$ counters , sorted by their values.
- **ID-Index:** A mapping between an item ID and its allocated counter in Value-Index (described below). The mapping is implemented as a compact, pointer-less, chain-based hash table borrowed from [12].
- **Value-Index:** A mapping between a value of a counter appearing in Counter-Array and the index of the largest counter in Counter-Array with the same value.
- **R:** A reverse mapping from a counter in Counter-Array to the chain in ID-Index containing the counters' item. It is implemented as an array of chain-IDs of length M.

Next, we describe how CSS utilizes these data structures to perform the operations presented in Algorithm 1.

1) **Increment c_x:** In order to keep Counter-Array sorted, we swap the incremented counter with the highest index counter in this array that has the same value. To that end, we start by finding the counter i currently allocated for x using ID-Index. Next, we find the highest index counter j that has the same value as counter i using Value-Index. We proceed by swapping counters (i, j) in Counter-Array, in $\mathbf{R}$, and in ID-Index. Finally, we increment the counter and update Value-Index.
2) **Finding the value of c_x:** Using ID-Index, we find the index of the counter allocated for x, and return its value.
3) **Re-allocating the minimal counter for x (lines 9-11)** Since Counter-Array is sorted, the minimal counter is simply stored at index zero. Using $\mathbf{R}$, we find the current

element associated with it and remove it from ID-Index. Finally, we add x to ID-Index and update $\mathbf{R}$ accordingly.

Our design aims to be frugal in the number of bits required for each entry. First, for finding the node in ID-Index that points at counter i we iterate over chain $R[i]$ until we find the right node. We avoid keeping the actual item-ID in R as the chain-ID and Counter-Array index suffice for finding the node in ID-Index. To further reduce memory overheads, we use array indices rather than pointers to memory addresses. Given the size of actual pointers in modern architectures, this saves a considerable amount of memory. For example, each entry in ID-Index only requires $\lceil \log M \rceil$ bits to store the counter index.

C. Flush Operation

When stream algorithms operate over a long period of time, their accuracy degrades to the point where they become useless. A possible solution is switching to the sliding window model, as in the (W, ϵ)-WINDOW COUNTING problem, for which we present an efficient algorithm in Section V.

Alternatively, one can periodically *flush* the data structure and start with a fresh counting process. CSS supports such operations in $O(1)$ time. This plays a crucial part in Section V, where we present WCSS for the sliding window model.

Flushing CSS in $O(1)$ time requires overcoming several challenges; first, we need to reset the values of all the counters. This is accomplished by keeping an additional variable ℓ, which counts how many counters are currently used by CSS. We refer to such counters as *live*. When an instance of CSS is initialized, ℓ is set to zero. It is increased whenever a new counter is used. The flush operation simply sets ℓ to 0, indicating that there are no live counters. When a query for an item arrives, if its counters location is smaller than ℓ, the counter is alive and we return its value; otherwise, we return the value of the minimal counter, as in Line 15.

The second challenge is keeping the ID-Index table from growing every time a counter is allocated for a new item. This is done by using the reverse mapping $\mathbf{R}$; whenever a new counter is allocated, we check if it has a reverse mapping. If such a mapping exists, it points to an item that previously used this counter (i.e., before the CSS instance was flushed). In this case, we remove the old item from ID-Index, thereby ensuring it contains no more than k records. Value-Index is cleared in a similar fashion.

D. SAILing Streams

We now present a near-optimal data structure for implementing Counter-Array, named *Sorted Array of Increasing Length (SAIL)*. While a simple M-sized array of $\log N$ bits counters suffices for implementing Counter-Array, we use the fact that the sum of counters is bounded by N to save space. Our data structure optimizes the memory requirements, while keeping all the required operations running in $O(1)$ time.

The intuition behind SAIL is to pre-allocate each counter with the maximal number of bits it may require, considering the array is sorted. For example, while the largest counter can potentially reach a value of N, the value of the smallest

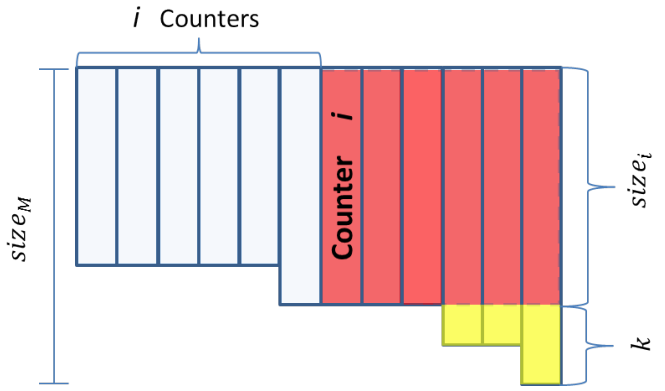


Figure 3. The offset of counter i is computed by subtracting the areas of the red rectangle and the excess bits below it from the total size of the SAIL.

counter, cannot exceed $\lfloor \frac{N}{M} \rfloor$. Similarly, the value of the 8'th largest counter is at most $\lfloor \frac{N}{8} \rfloor$, etc. SAIL allocates a different number of bits for different counters according to our observation, as illustrated in Figure 3. While this sacrifices bit alignment, we show that we can still perform increments and queries over the individual counters in $O(1)$ time.

1) Counter Sizes: SAIL allocates $size_i \triangleq \lceil \log(N+1) \rceil - \lfloor \log(M-i) \rfloor$ bits for the i^{th} counter. To show that this is sufficient, we first show a couple of technical lemmas.

Lemma 4.2: $\forall N \in \mathbb{N}$ and $x \geq 1 : \lfloor \frac{N}{x} + 1 \rfloor \leq \lceil \frac{N+1}{x} \rceil$.

Proof: To see this, consider the case where $\frac{N}{x} \in \mathbb{N}$, and thus

$$\left\lfloor \frac{N}{x} + 1 \right\rfloor = \frac{N}{x} + 1 = \frac{N}{x} + \left\lceil \frac{1}{x} \right\rceil = \left\lceil \frac{N+1}{x} \right\rceil.$$

Otherwise, when $\frac{N}{x} \notin \mathbb{N}$, we get $\lfloor \frac{N}{x} + 1 \rfloor = \lceil \frac{N}{x} \rceil \leq \lceil \frac{N+1}{x} \rceil$. ■

Lemma 4.3: $\forall r \in \mathbb{R}$ such that $r > 1 : \log \lceil r \rceil \leq \lceil \log r \rceil$.

Proof: Let $k \in \mathbb{N}$ such that $2^{k-1} < r \leq 2^k$. Now $k-1 < \log r \leq k$, and therefore $\lceil \log r \rceil = k$ and $\log \lceil r \rceil \leq k = \lceil \log r \rceil$. ■

We now establish the following:

Lemma 4.4: Let $c_0 \leq c_1 \leq \ldots \leq c_{M-1} \in [N+1]$ such that $\sum_{i \in [N]} c_i \leq N$, then $\forall i \in [M] : c_i \leq \left\lfloor \frac{N}{M-i} \right\rfloor$ and can be represented using $size_i$ bits.

Proof: Notice that since $c_i \leq \left\lfloor \frac{N}{M-i} \right\rfloor$, the number of bits required for representing counter i is:

$$\left\lceil \log \left\lfloor \frac{N}{M-i} + 1 \right\rfloor \right\rceil \leq \left\lceil \log \left\lceil \frac{N+1}{M-i} \right\rceil \right\rceil \leq \left\lceil \left\lceil \log \frac{N+1}{M-i} \right\rceil \right\rceil$$
$$= \lceil \log(N+1) - \log(M-i) \rceil \leq \lceil \log(N+1) \rceil - \lfloor \log(M-i) \rfloor = size_i,$$

where the first two inequalities follow from Lemma 4.2 and Lemma 4.3 above. ■

2) Offset Calculation: To efficiently access counter i, the $(i+1)$'th smallest counter, we first compute its bit-offset inside the SAIL, as demonstrated in Figure 3:

1) Compute $size_i$.
2) Denote by k the difference in bits between the size of counter i and the size of the largest counter, i.e.,

$$k \triangleq size_{M-1} - size_i.$$

3) Compute the actual offset which is given by:

$$\begin{aligned}\text{OFFSET}_i &= \sum_{j=0}^{i-1} size_j = \mathcal{S} - \sum_{j=i}^{M-1} size_j \\ &= \mathcal{S} - size_i \cdot (M-i) - \sum_{j=i}^{M-1}(size_j - size_i) \\ &= \mathcal{S} - size_i \cdot (M-i) - \sum_{j=i}^{M-1}\sum_{\ell=1}^{k} \ell \cdot \mathbb{1}_{size_j - size_i = \ell} \\ &= \mathcal{S} - size_i \cdot (M-i) - \sum_{\ell=1}^{k} 2^{k-\ell} \cdot \ell \\ &= \mathcal{S} - size_i \cdot (M-i) - (2^{k+1} - k - 2) \qquad (1)\end{aligned}$$

Finally, the counter is stored in bits $[\text{OFFSET}_i, \text{OFFSET}_i + size_i - 1]$.

3) Space Analysis: We now prove that SAIL can achieve an optimal data representation, up to a small additive factor. We first calculate the total amount of information stored. This gives us a lower bound on the number of bits required.

Theorem 4.5: The minimal number of bits required to store M counters whose total sum is at most N is at least $\mathcal{B} \triangleq M \log\left(\frac{N+1}{M}\right)$.

Proof: First notice that there are $\binom{N+M}{M}$ possible configurations of M counters whose sum is at most N. The number of required bits is then:

$$\left\lceil \log \binom{N+M}{M} \right\rceil \geq \log\left(\frac{N+M}{M}\right)^M \geq M \log\left(\frac{N+1}{M}\right)$$

bits. ∎

We now show that SAIL matches the information theoretic lower bound up to a small additive value.

Theorem 4.6: The memory requirement of the SAIL structure is $\mathcal{S} \triangleq \sum_{i\in[M]} size_i \leq M \cdot (size_0 + 2) \leq M(\mathcal{B}+3)$ bits.

Proof: We can derive the size of SAIL $\mathcal{S}$ from equation (1). Let $0 \leq \mu < 1$ be s.t. $(\log M - \mu) \in \mathbb{N}$.

$$\begin{aligned}\mathcal{S} &= (\lceil \log(N+1) \rceil - \lfloor \log(M) \rfloor) \cdot M \\ &\quad + (2^{\lfloor \log M \rfloor + 1} - \lfloor \log M \rfloor - 2) \\ &\leq (\lceil \log(N+1) \rceil - \lfloor \log(M) \rfloor) \cdot M + 2^{\lfloor \log M \rfloor + 1} \\ &\leq M \cdot (\lceil \log(N+1) \rceil - \log M + 2^{1-\mu} + \mu) \\ &\leq M \cdot (\lceil \log(N+1) \rceil - \log M + 2)\,.\end{aligned}$$

We get that $\mathcal{S} \leq M \cdot (size_0 + 2)$, i.e., the average counter length is larger than the smallest counter by at most two bits. ∎

Our method takes at most three bits per counter on top of the information theoretic lower bound, or only two bits in case $N+1$ is a power of two!

E. Runtime Analysis

Theorem 4.7: The running time of point queries and update operations in CSS is O(1) w.h.p., while returning the entire list of heavy hitters takes $O(\frac{1}{\varepsilon})$ time.

The proof is immediate from the description of these operations in Section IV-B.

F. Memory Analysis

The following lemma allows us to obtain an improved bound over the number of records in Value-Index in the case where $\epsilon^{-1} \geq \sqrt{2N}$.

Lemma 4.8: Let $x_0, x_1, \ldots x_{k-1}$ be integers such that $\sum_{i=0}^{k-1} x_i = N$. Then the number of distinct values in $x_0, x_1, \ldots x_{k-1}$ is less than $1 + \sqrt{2N}$.

Proof: Let $V = v_0 \leq v_1 \leq v_{r-1}$ be the set of distinct values taken by $x_0, x_1, \ldots x_{k-1}$. Then $N = \sum_{i=0}^{k-1} x_i \geq \sum_{i=0}^{r-1} v_i \geq \sum_{i=0}^{r-1} i = \frac{r(r-1)}{2} > \frac{(r-1)^2}{2}$ immediately implying that $r < 1 + \sqrt{2N}$. ∎

Next, we calculate the memory requirements for each of the data structures in our implementation.

- **Counter-Array:** As we analyzed above, the size of the SAIL data structure is $\mathcal{S} \leq M \cdot (size_0 + 2)$.
- **ID-Index:** This is a hash table, which contains at all times at most ϵ^{-1} records (one for each counter). The keys are represented by ω bits and the values with $\log \epsilon^{-1}$ bits. Therefore, ID-Index requires $H(\epsilon^{-1}, \omega, \log \epsilon^{-1})$ bits.
- **Value-Index:** This structure contains at most ϵ^{-1} records, one for each counter. On the other hand, according to Lemma 4.8, it contains no more than $1 + \sqrt{2N}$ records, one for each possible value. Each key is represented by $\log(N+1)$ bits and each value by $\log \epsilon^{-1}$ bits. Therefore the number of bits required by Value-Index is $H(\min\{\epsilon^{-1}, 1+\sqrt{2N}\}, \log(N+1), \log \epsilon^{-1})$.
- **R:** This is an array of ϵ^{-1} records, each of size $\log \text{nchains}(\epsilon^{-1}, \omega, \log \epsilon^{-1})$ bits. Alternatively, it can be implemented by storing the IDs of the items associated with each counter.

The total memory required (using a succinct hash table [3]) is: (a) in case $\epsilon^{-1} < 1 + \sqrt{2N}$,

$$\begin{aligned}&\left(\epsilon^{-1} \log N\epsilon + \epsilon^{-1}\omega + \epsilon^{-1} \log N + \epsilon^{-1}\omega\right)(1 + o(1)) \\ &\qquad = \epsilon^{-1}(\log N + \log N\epsilon + 2\omega)\,(1 + o(1))\end{aligned}$$

or otherwise, (b)

$$\begin{aligned}&\left(\epsilon^{-1} \log N\epsilon + 2\epsilon^{-1}\omega + \sqrt{2N} \log\left(\sqrt{N}\epsilon^{-1}\right)\right)(1 + o(1)) \\ &\qquad \leq \epsilon^{-1}(\log N + \log N\epsilon + 2\omega)\,(1 + o(1))\,.\end{aligned}$$

Remark In order to completely benefit from the space savings of the SAIL structure, not all counters can be word aligned in memory. However, it is still possible to reach each such counter using a one or two register shift operations plus one or two bit-mask operations, which are very fast. Hence, in many cases, the benefit of the smaller memory footprint can make up for the additional operations overheads. For example, if the lower overheads indicates that our algorithm fits in SRAM instead of DRAM it is worth it. In software implementations, the data structure fits into fewer cache lines and enjoys better memory locality. In other cases, SAIL can be replaced with a simple array that slightly increases the space overheads.

G. (ϵ, δ)-Counting

In the weaker (ϵ, δ)-COUNTING problem, the approximation grantee is probabilistic. This definition is identical to that of Count Min Sketch [6] and similar constructions. Here, we suggest a method to further optimize the space consumption of CSS when we only need to solve (ϵ, δ)-COUNTING.

	Memory Requirement	Update Time	(W,θ,ϵ)-Heavy Hitters Query Time	(W,ϵ)-Window Counting Query Time	Static Memory
Arasu and Manku [2]	$O\left(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\right)$	N/A	✘
Hung and Ting [19]	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon}\right)$	N/A	✘
Hung et al. [16]	$O\left(\frac{1}{\epsilon}\right)$	$O(1)$	$O\left(\frac{1}{\epsilon}\right)$	N/A	✘
This Paper	$O\left(\frac{1}{\epsilon}\right)$	$O(1)$	$O\left(\frac{1}{\epsilon}\right)$	$O(1)$	✔

Table II
Comparison of Sliding Window Algorithms.

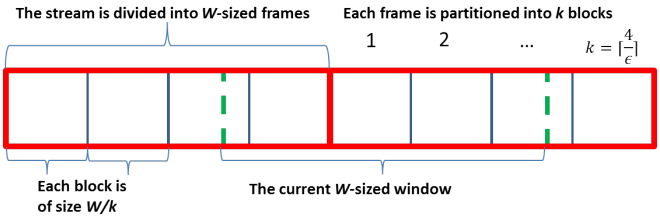


Figure 4. The stream is divided into intervals of size W called *frames*. Each frame is partitioned into k equal-sized *blocks*. The window of interest is also of size W, and overlaps with at most 2 frames and $k+1$ blocks.

Instead of using item-IDs of size ω-bits as keys for ID-Index, we can use hashed IDs (also called fingerprints) of size ω'-bits (where $\omega' < \omega$). Alas, this may result in collisions between different hashed IDs, which is the source of the probabilistic guarantee. We calculate how large ω' should be so that with probability $1-\delta$, no collisions will occur.

The probability for two different items to have the same fingerprint is: $2^{-\omega'}$. The probability that the fingerprint of a queried item is identical to that of *any* other item in the stream is then bounded by $N \cdot 2^{-\omega'}$. Finally, if we pick ω' to be $\log\frac{N}{\delta}$, we get that the probability for a wrong answer is at most $N \cdot 2^{-\log\frac{N}{\delta}} = \delta$. Notice, however, that when using this optimization, we no longer store explicit IDs. Consequently, we can only answer point queries about a specific key.

This optimization reduces the size of ID-Index from $H(\epsilon^{-1}, \omega, \log\epsilon^{-1})$ bits to $H(\epsilon^{-1}, \log\frac{N}{\delta}, \log\epsilon^{-1})$ bits. In addition, the size of each item in $\mathbf{R}$ reduces to $\log \text{nchains}(\epsilon^{-1}, \log\frac{N}{\delta}, \log\epsilon^{-1})$. This is a significant reduction in space when the size of item-IDs are large, e.g. when the item-IDs consists of 5-tuples.

V. Frequency Estimation over Sliding Windows

In this section, we present *Window Compact Space-Saving (WCSS)*, a novel algorithm for (W,ϵ)-Window Counting. WCSS divides the stream into *frames* of size W, where each frame is then further partitioned into $k \triangleq \lceil\frac{4}{\epsilon}\rceil$ equal-sized *blocks*. For simplicity, we assume the block size, $\frac{W}{k}$, is an integer. Figure 4 illustrates the algorithm's setting.

Intuitively, WCSS keeps count of how many times each item arrived during the last frame. Whenever an item arrival causes a counter to reach a multiple of the block size $\left(\frac{W}{k}\right)$, we say it *overflowed*. When an item overflows, we remember in which block that happened. To approximate the frequency of an item in the window, we can count the number of times it overflowed and multiply the result by $\frac{W}{k}$.

y	A CSS instance using k counters.
b	A queue of $k+1$ queues. An efficient implementation is described in Section V-B.
B	The histogram of b, implemented using a succinct hash table.
M	The index within the current frame.

Table III
Variables used by the (W,ϵ)-Window Counting algorithm.

Specifically, WCSS does not keep a unique counter for each item because if it did, the memory consumption would be at least W. Instead, frequencies are kept only for suspected heavy hitters within the last frame. This is achieved with an instance of the CSS data structure, denoted y. y contains k counters and provide a $\frac{W}{k}$-error frequency estimation when queried.

To keep record of the overflowing items, the algorithm uses a queue of queues named b that contains $k+1$ queues, one for each block that overlaps with the current window. Each queue in b contains the IDs of items that overflowed in its corresponding block. Whenever an item overflows, we append its ID to the current block's queue.

When a block ends, we remove from b the oldest queue, as it is associated with a block which no longer overlaps with the window. Instead, we add an empty queue for the next block. This keeps the number of queues in b fixed at all times. Furthermore, this process deletes overflows if they happened during a block that has already exited the window.

To answer a point query for an item, we multiply the number of overflows that item had by the block size, and add the *remaining* appearance-count. The latter is the remainder of the value reported by y when divided by the block size. The result may be smaller than the true frequency so we increase it by $2\frac{W}{k}$ to make the error one sided.

Constant time point queries are enabled through the B table, which counts the number of overflows for each item. That is, $B[x]$ is the number of times x appears in b. As multiple items may overflow in a single block, updating B whenever a block ends is infeasible in $O(1)$ time. We solve this by *deamortizating* the update process. Specifically, whenever an item arrives, we remove a *single* overflowed item from the oldest block in b (if such exists).

Table III provides a list of the variables used by WCSS, while Algorithm 2 provides the pseudo-code.

A. Analysis

For proving the correctness of WCSS, we first introduce some notations. x denotes the queried element; we assume that the current frame started when an item with time-stamp W was added and that the most recent item arrived at time-stamp $W+M$. Therefore, the algorithm is required to approximate the frequency of x inside the window $\langle M+1, M+2, \ldots, W+M\rangle$. We denote by y_t the value returned by y if queried about item x *after* the t-th item was added. We use the indicators x_t for specifying whether x has arrived at time t, and u_t for indicating whether x has overflowed at time t. Using these notations, we aim to approximate $f_x^W = \sum_{t=M+1}^{W+M} x_t$. For simplicity, mark $B[x]=0$ for x not in B.

Algorithm 2 (W,ϵ)-WINDOW COUNTING Algorithm

```
1: Initialization: y = CSS(k), M = 0, B = Empty hash,
                   b = Queue of k + 1 empty queues.
2: function ADD(Item x)
3:     M = M + 1  mod W
4:     if M = 0 then                                   ▷ new frame
5:         y.FLUSH()
6:     if M  mod W/k = 0 then                          ▷ new block
7:         b.POP()
8:         b.APPEND(new empty queue)
9:     if b.tail is not empty then                     ▷ remove oldest item
10:        oldID = b.tail.POP()
11:        B[oldID] = B[oldID] − 1
12:        if B[oldID] = 0 then
13:            B.REMOVE(oldID)
14:    y.ADD(x)                                        ▷ add item
15:    if y.QUERY(x)= 0 mod W/k then                   ▷ overflow
16:        b.head.PUSH(x)
17:        if B.CONTAINS(x) then
18:            B[x] = B[x] + 1
19:        else B[x] = 1                               ▷ Adding x to B

20: function QUERY(Item x)
21:    if B.CONTAINS(x) then
22:        return W/k · (B[x] + 2) + (y.QUERY(x)  mod W/k)
23:    else                                            ▷ no overflows
24:        return 2W/k +y.QUERY(x)
```

We first prove a helper lemma.

Lemma 5.1: Every $\frac{W}{k}$ subsequent arrivals of x must cause exactly one overflow.

Proof: After the last of the $\frac{W}{k}$ arrivals of x, it is associated with a counter in y. y is an over-estimator and therefore that counter is larger or equal to $\frac{W}{k}$. A counter with value $\frac{W}{k}$ or more can never be associated with other items because it is too large to become the minimum amongst k counters of sum less than W. This implies that x's counter must have overflowed at or before the last arrival of x.

If the overflow occurred before the first among the $\frac{W}{k}$ subsequent arrivals, any following arrival of x increases the counter by exactly one. Therefore, clearly the $\frac{W}{k}$ arrivals cause the counter to cross an integer product of $\frac{W}{k}$ exactly once.

If the first overflow occurred during the $\frac{W}{k}$ subsequent arrivals, $\frac{W}{k}$ more arrivals of x are needed for the second overflow to happen. In this case, x will then not overflow again during our $\frac{W}{k}$ arrivals. ∎

Corollary 5.2: Fewer than $\frac{W}{k}$ subsequent arrivals of x may cause at most one overflow.

Proof: If these arrivals cause more than one overflow, then a larger number of arrivals can also cause more than one overflow, contradicting Lemma 5.1. ∎

Theorem 5.3: Algorithm 2 solves (W,ϵ)-WINDOW COUNTING.

Proof: The structure of the proof is as follows. We first analyse the number of overflows before we flush y. Next, we consider the number of overflows after the flush. Then, we address the error caused by deleting items from the oldest block. We add the inherent error from y. Finally, we put it all together to get the approximation guarantee.

We start by analyzing the number of times x has overflowed before the flush. We split the arrivals of x to chunks of size $\frac{W}{k}$ and a remainder of less than $\frac{W}{k}$ arrivals. The number of chunks is $\left\lfloor \frac{\sum_{t=M+1}^{W-1} x_t}{\frac{W}{k}} \right\rfloor$, or $\left\lceil \frac{\sum_{t=M+1}^{W-1} x_t}{\frac{W}{k}} \right\rceil$ if we count the remainder as a chunk. According to Lemma 5.1 there is one overflow in each chunk and according to Corollary 5.2 there is at most one overflow during the remainder. Therefore, the number of overflows in the window before the flush is $\left\lfloor \frac{\sum_{t=M+1}^{W-1} x_t}{\frac{W}{k}} \right\rfloor \leq \sum_{t=M+1}^{W-1} u_t \leq \left\lceil \frac{\sum_{t=M+1}^{W-1} x_t}{\frac{W}{k}} \right\rceil$ (2).

Next, we consider what happens in the current frame, i.e., after the flush. The counter associated with x, if such exists, has also been associated with x with each overflow it has caused. This is because after the first time a counter overflows, it becomes too large to be replaced (see proof of Lemma 5.1). Since the counter is incremented by one every time, it had to go through all integer products of $\frac{W}{k}$, causing an overflow each time. Thus,

$$y_{W+M} = \frac{W}{k} \cdot \sum_{t=W}^{W+M} u_t + \left(y_{W+M} \mod \frac{W}{k} \right). \tag{3}$$

If no counter is associated with x, it has not overflowed and $y_{W+M} < \frac{W}{k}$. Thus, Equation 3 still holds.

Next, we connect between the actual number of overflows that occurred within the window, $\sum_{t=M+1}^{W+M} u_t$, and the number of overflows recorded in $B[x]$. The two quantities may not be the same, as the deamortization (Line 10) could remove overflowing items from b before they leave the window. The value of $B[x]$ is $\sum_{t=M_2+1}^{W+M} u_t$, for some $M \leq M_2 \leq M + \frac{W}{k}$. M_2 cannot be smaller than M because the first M overflows were removed from the oldest block in b if such existed. Consequently, between times M and M_2, according to Corollary 5.2, there is no more than one overflow. This means that the number of recorded overflows at time $W+M$, $B[x]$, might be lower by 1 than the actual count, i.e.,

$$\sum_{t=M+1}^{W+M} u_t - 1 \leq B[x] \leq \sum_{t=M+1}^{W+M} u_t. \tag{4}$$

Since y is a CSS instance with k counters, it solves the $\frac{1}{k}$-COUNTING problem, and therefore

$$\sum_{t=W}^{W+M} x_t \leq y_{W+M} \leq \sum_{t=W}^{W+M} x_t + \frac{W}{k}. \tag{5}$$

The output of the algorithm when queried for x is

$$\begin{aligned} \widehat{f_x^W} &= \frac{W}{k} \cdot (B[x]+2) + \left(y_{W+M} \mod \frac{W}{k} \right) \\ &=_{(3)} \frac{W}{k} \cdot (B[x]+2) + y_{W+M} - \frac{W}{k} \cdot \sum_{t=W}^{W+M} u_t. \end{aligned}$$

At last, we bound the error of the algorithm from above:

$$\begin{aligned} \widehat{f_x^W} &= \frac{W}{k} \cdot (B[x]+2) + y_{W+M} - \frac{W}{k} \cdot \sum_{t=W}^{W+M} u_t \\ &\leq_{(5)} \frac{W}{k} \cdot (B[x]+2) + \sum_{t=W}^{W+M} x_t + \frac{W}{k} - \frac{W}{k} \cdot \sum_{t=W}^{W+M} u_t \\ &\leq_{(4)} \frac{W}{k} \cdot \sum_{t=M+1}^{W+M} u_t + \sum_{t=W}^{W+M} x_t + 3\frac{W}{k} - \frac{W}{k} \cdot \sum_{t=W}^{W+M} u_t \\ &= \frac{W}{k} \cdot \sum_{t=M+1}^{W-1} u_t + \sum_{t=W}^{W+M} x_t + 3\frac{W}{k} \\ &\leq_{(2)} \frac{W}{k} \cdot \left\lceil \frac{\sum_{t=M+1}^{W-1} x_t}{\frac{W}{k}} \right\rceil + \sum_{t=W}^{W+M} x_t + 3\frac{W}{k} \\ &\leq \sum_{t=M+1}^{W+M} x_t + 4\frac{W}{k} \leq f_x^W + W\epsilon. \end{aligned}$$

And from below:

$$\begin{aligned}
\widehat{f_x^W} &= \frac{W}{k}\cdot(B[x]+2) + y_{W+M} - \frac{W}{k}\cdot\sum_{t=W}^{W+M} u_t \\
&\geq_{(5)} \frac{W}{k}\cdot(B[x]+2) + \sum_{t=W}^{W+M} x_t - \frac{W}{k}\cdot\sum_{t=W}^{W+M} u_t \\
&\geq_{(4)} \frac{W}{k}\cdot\sum_{t=M+1}^{W+M} u_t + \sum_{t=W}^{W+M} x_t + \frac{W}{k} - \frac{W}{k}\cdot\sum_{t=W}^{W+M} u_t \\
&= \frac{W}{k}\cdot\sum_{t=M+1}^{W-1} u_t + \sum_{t=W}^{W+M} x_t + \frac{W}{k} \\
&\geq_{(2)} \frac{W}{k}\cdot\left\lfloor \frac{\sum_{t=M+1}^{W-1} x_t}{\frac{W}{k}} \right\rfloor + \sum_{t=W}^{W+M} x_t + \frac{W}{k} \\
&\geq \sum_{t=M+1}^{W+M} x_t = f_x^W.
\end{aligned}$$

We have established that $f_x^W \leq \widehat{f_x^W} \leq f_x^W + W\epsilon$, and thus the algorithm solves (W,ϵ)-WINDOW COUNTING.

■

An immediate corollary of Theorem 5.3 is that our algorithm could be used for finding heavy hitters:

Theorem 5.4: By returning the set of elements for which $\widehat{f_x^W} \geq \theta\cdot W$, Algorithm 2 solves (W,θ,ϵ)- HEAVY HITTERS .

B. Efficient Data Structure for b

We describe a succinct implementation of the queue-of-queues data structure b, as illustrated in Figure 5. As every overflow requires a counter that reaches a multiple of $\frac{W}{k}$, at most k items can overflow in each frame, and the window overlaps with at most two frames. Therefore, b can contain at most $2k$ elements, and consequently, we keep the IDs of overflown items in an array O of size $2k$. Additionally, we use a $3k$-sized circular bit-array z , which specifies in which block each element has overflowed.

As the number of overflown items could vary between zero and $2k$, some of the cells in O and z may not be *active*. In order to allow $O(1)$ time operations, we keep the indices of the oldest and the newest elements in O, as well as the indices of the first and last active bits in z.

Since b has $k+1$ queues at any time, we have exactly k active zero-bits in z, each indicating the end of a queue. Every overflow is marked with a set bit in z, and its position specifies in which block the overflow took place. Thus, the number of active ones equals the number of overflows recorded in b.

The "First" bit in z specifies whether the earliest overflow occurred within the oldest block that overlaps with the window. Specifically, a set bit indicates that the first overflow is recorded in the oldest queue; otherwise, no element has overflowed in that block, and the length of the zero sequence determines when did the first overflow took place.

Whenever we append an item to the queue of the newest block (Line 16), we increment (modulo $2k$) the "New" index of O and insert the ID of the overflowing item. We also increment (mod $3k$) the "Last" index in z, inserting a set bit. In order to check whether the tail-queue is empty (Line 9), we simply examine the "First" index bit in z. For removing the oldest item from the oldest queue (Line 10), the "Old" and the "First" indices are incremented. Finally, for removing a queue (Line 7), we simply increment the "First" index.

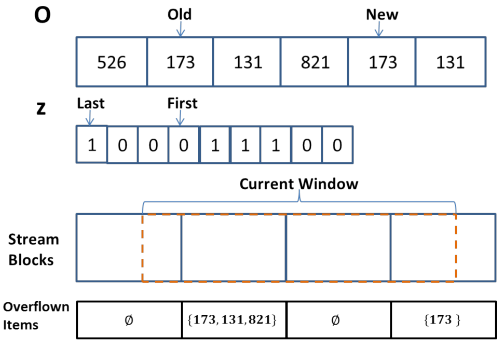


Figure 5. Example of the b data structure with $k=3$. Since the "First" bit in z is not set, no item has overflowed in the oldest block of the current window. Items 173, 131 and 821 have overflowed in the following block, while 173 overflowed in the most recent block as well. The histogram B contains only $\{173, 131, 821\}$, where $B[173]=2$, and $B[131]=B[821]=1$. Item 526 has overflowed before the current window and it is no longer a part of b.

Since our structure keeps $2k$ element IDs, a bit-array and a few indices, the space requirements for b are $k(2\omega+3)+O(\log k)$, i.e., only about 1.5 bits per item more than required for representing the IDs of the overflowing items!

C. Runtime Analysis

As can be observed in the description of WCSS and pseudo-code in Algorithm 2, point queries and updates involve a constant number of operations. Furthermore, as we maintain $\frac{1}{\varepsilon}$ counters in each CSS instance, we get the following theorem:

Theorem 5.5: The running time of point queries and update operations in WCSS is O(1) w.h.p., while returning the entire list of heavy hitters takes $O(\frac{1}{\varepsilon})$ time.

D. Memory Analysis

- **y:** A CSS instance initialized with an ϵ parameter of k^{-1}. Its space requirements can be found in subsection IV-F.
- **b:** A queue which requires $k(2\omega+3)+O(\log k)$ bits as discussed in subsection V-B.
- **B:** This is a hash table, which contains at all times at most $2k$ records (one for each overflow in the windows). The keys are represented by ω bits and the values with $\log(2k)$ bits. Therefore, B requires $H(2k,\omega,\log(2k))$ bits.

In total, the memory requirements when using a succinct hash table is

$$\begin{aligned}
&k(2\log W + 2\omega - \log k + 2\omega + 2\omega)(1+o(1)) = \\
&\qquad k(2\log W + 6\omega - \log k)(1+o(1))
\end{aligned}$$

E. (W,ϵ,δ)-WINDOW COUNTING

For window counting, the size of the item-ID is still a large portion of the space requirements. In order to use less memory, we can allow our algorithm to answer a query correctly with probability $1-\delta$, same as with (ϵ,δ)-COUNTING.

Similarly to Section IV-G, we replace item-IDs with hashed IDs of size $\omega' \triangleq \log\frac{W}{\delta}$, which guarantees that the probability of collision within the window is at most δ.

In addition to the reduction in the size of CSS, which reduces the size of ID-Index to $H(k,\log\frac{W}{\delta},\log k)$ and the

size of each item in $\mathbf{R}$ to $\log \text{nchains}(k, \log \frac{W}{\delta}, \log k)$, we also reduce the size of b to $k \log \frac{W}{\delta} + O(\log k)$ and the size of B to $H(2k, \log \frac{W}{\delta}, \log(2k))$. The overall reduction in space can be very significant when item-IDs are large.

VI. Evaluation

We compare the space requirements and accuracy of our algorithms with previous works on both simulated data and a real Internet packet trace.

The first dataset we use to evaluate our algorithm is a simulated trace of 10^7 packets, generated with a Zipf distribution of skew 1, over a domain of 2^{20} possible flows. We refer to this dataset as *Zipf*. The second dataset we use is the CAIDA Anonymized Internet Trace 2015 [1], or in short *CAIDA*. It is a trace from the 'equinix-chicago' high-speed monitor. We use 5-tuples from February 19th, 2015, from 13:00 o'clock. We count more than $18 \cdot 10^6$ packets. Before counting every 5-tuple, we apply a hash function to reduce the size of the 5-tuple to 62 bits, since some of the implementations required every item to be no more than 64 bit long.

We start by comparing CSS with previous implementations of Space-Saving. We used C code from [5] for the previous implementations of Space-Saving and compared it with our own Java implementation of CSS. The first benchmark we compare with, SSL, uses linked lists as illustrated in Figure 1. An alternative implementation to Space-Saving, SSH, is heap based. The latter gives a reduction in memory, but update operations take $O(\log k)$ time. We conducted several runs on the Zipf dataset with varying ϵ's. The memory for each algorithm was computed by counting the number of bits in the implementation for the parameters given during runtime. Figure 6(a) depicts the results. As shown, the memory of all algorithms is decreasing with ϵ, as expected. SSH requires less memory than SSL because it uses less pointers than SSL and saves the items in a heap. CSS requires considerably less space than both competing algorithms. It requires up to 75% less memory than SSH and 86% less than SSL. That is, we achieve a memory improvement of more than 7 times compared to the previous $O(1)$-time implementation of Space-Saving.

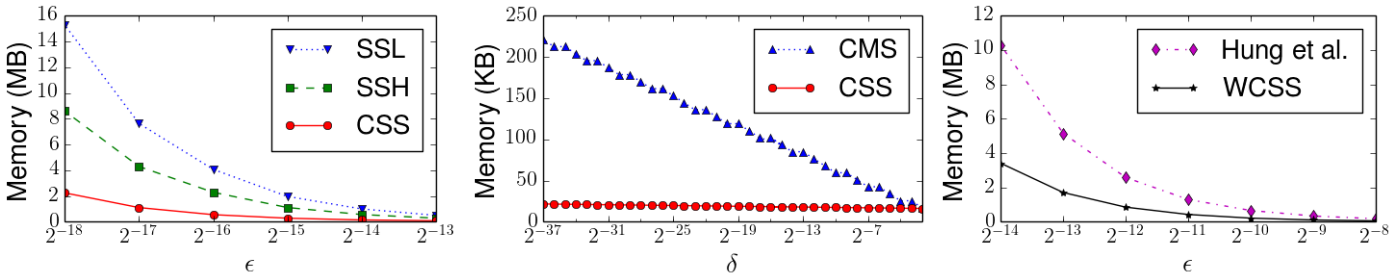
We continue by comparing the (δ, ϵ)-version of CSS to Count-Min-Sketch (CMS) [6]. We use the parameters of the CAIDA trace. We set $\epsilon = 2^{-10}$ and go over several values of δ. CMS requires $\lceil e/\epsilon \rceil \left\lceil \ln \frac{1}{\delta} \right\rceil \lceil \log_2 N \rceil$ bits of memory. For computing the space requirements of our algorithm, we count the number of bits used by all data structures in our Java implementation. Figure 6(b) compares between the two algorithms. As δ grows smaller, the space requirement of CMS grows substantially. In contrast, our algorithm's space grows much slower, since only one counter is kept for every suspected heavy hitter. Eventually, CMS achieves the same memory as CSS, but only, when δ becomes very large.

Next we compared WCSS with the previous state of the art for finding heavy hitters in windows, an algorithm by Hung et al. [16]. Their implementation uses a structure that is almost identical to the one presented in Figure 1. They use a doubly linked list L, where each node contains a value associated with one or more counters. Every node points to a doubly linked list of counters, each keeping the item ID, a pointer to its origin node in L and a queue of "overflow" timestamps (each representing $\frac{W}{m}$ appearances). Similarly to our analysis of b, the total number of overflows (and thus, sum of queue sizes) is bounded by m. For computing the memory required by the algorithm, we consider $m = \lceil \frac{4}{\epsilon} \rceil$ counters, each containing six 64-bits pointers – two for the counters list, two for the chain-based hash, one for the node in L and one for the timestamp queue. Additionally, each counter keeps the item ID, whose length is determined by the scenario – 4 bytes for the Zipf dataset and 8 bytes for CAIDA. Each node in L requires 3 pointers, two for L itself and one for the counter list. Additionly, each node requires a 2-bytes "difference" field, representing by how much the node's value is larger than its predecessor. For the mapping, we assume chain-based hashing is used with $2m$ lists, similarly to the publicly available implementation of [5], which uses a similar structure. Finally, the number of nodes in L is bounded by Lemma 4.8 to be $\min\{\sqrt{2W}, m\}$. We used the same parameters as the CAIDA trace and chose a window size of 2^{16}. Figure 6(c) compares the memory of the two algorithms, exhibiting that in addition to the reduction in query time to $O(1)$, our algorithm also obtains more than a threefold memory reduction.

Last, we evaluated the error of our window-counting algorithm compared to the Hung et al. algorithm. We set $\epsilon = 2^{-10}$ and the window size to 2^{16}, yielding an expected deviation of between 0 and 64 from the real value. We simulated both algorithms on the two datasets in two scenarios. In the first, a networking device needs to make a decision after the arrival of every packet, so we measure the error of the packets that just arrived. In the second scenario, the networking device occasionally answers a query for the heavy hitters. The histograms of the deviation from the true value are given in Figure 7. In all of these scenarios, WCSS is shown to be more accurate than Hung et al (we consider a deviation of 0 to be most accurate and a deviation of 64 as least accurate). The difference is greater when examining the deviations for the heavy-hitters. We conclude that WCSS offers improvement in the algorithmic level as well as in its compact implementation.

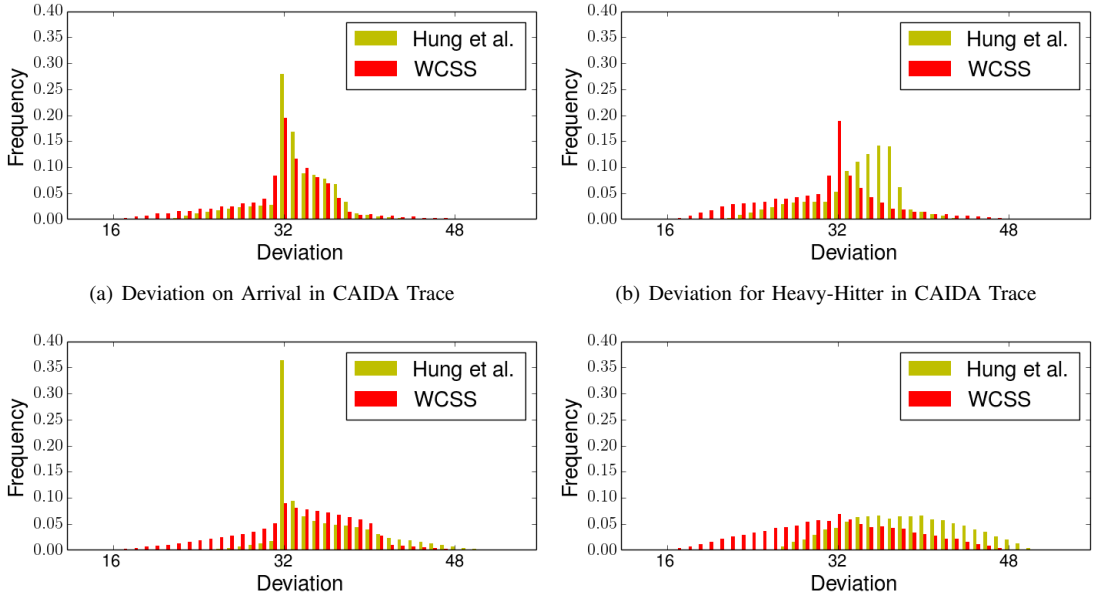
VII. Conclusions

In this work, we have suggested a new flavor of algorithms for the heavy hitters problem, namely, counter based algorithms with statically allocated memory. We combine the theoretical and empirical superiority of the counter-based approach with a relatively simple design that can potentially be ported to hardware. We have presented two algorithms: CSS for streams and WCSS for sliding windows. CSS is a redesign of the Space-Saving algorithm, which is considered best of breed for the stream case [5]. By using statically allocated memory and avoiding pointers, CSS reduces the space consumption of SSH by up to 75%, and that of the original Space-Saving implementation by up to 86%. This reduction is achieved when running over both synthetic data and a real TCP packet trace. Fast point queries are crucial



(a) ϵ-COUNTING Algorithms (b) (ϵ, δ)-COUNTING Algorithms, $(\varepsilon = 2^{-10})$ (c) (W, ϵ)-WINDOW COUNTING, $(W = 2^{16})$

Figure 6. Required space for certain accuracies for CSS and WCSS, compared to previous work (lower is better!).



(a) Deviation on Arrival in CAIDA Trace (b) Deviation for Heavy-Hitter in CAIDA Trace

(c) Deviation on Arrival for Zipf Distribution (d) Deviation for Heavy-Hitters for Zipf Distribution

Figure 7. Approximation error histograms for two datasets in two query models. Both algorithms are configured for the same approximation guarantee ($\varepsilon = 2^{-10}$, $W = 2^{16}$), Note that WCSS requires significantly less space for that guarantee. As can be seen, WCSS is empirically more accurate, especially on the Heavy-Hitter query model.

for many networking algorithms, as they make a decision for each packet. Both CSS and WCSS satisfy point queries in $O(1)$ time complexity (w.h.p.). For the sliding window case, supporting $O(1)$ point queries (w.h.p.) is an improvement over previous work. In addition, we reduced the memory of CSS and WCSS even further for scenarios where only point queries are required and a probabilistic estimation is allowed. In these scenarios, we compared CSS to CMS and demonstrated a considerable reduction in required space.

Acknowledgements: We thank Isaac Keslassy for helpful insights and comments. This work was partially funded by MOST grant #3-10886 and the Technion-HPI research school.

REFERENCES

[1] The caida ucsd anonymized internet traces 2015 - february 19th, http://www.caida.org/data/passive/passive2015dataset.xml

[2] Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of DB Systems, June 14-16, 2004. pp. 286–296

[3] Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS). pp. 787–796 (2010)

[4] Cohen, S., Matias, Y.: Spectral bloom filters. In: Proc. of the ACM International Conference on Management of Data (SIGMOD) (2003)

[5] Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. Proc. VLDB Endow. 1(2), 1530–1541 (Aug 2008), code: www.research.att.com/ marioh/frequent-items.html

[6] Cormode, G., Muthukrishnan, S.: An improved data stream summary: The count-min sketch and its applications. J. Algorithms 55 (2004)

[7] Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency estimation of internet packet streams with limited space. In: Proc. of the 10th Annual European Symposium on Algorithms. ESA, Springer-Verlag (2002)

[8] Dimitropoulos, X., Hurley, P., Kind, A.: Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. SIGCOMM Comput. Commun. Rev. 38(1) (Jan 2008)

[9] Dittmann, G., Herkersdorf, A.: Network processor load balancing for high-speed links. In: Proc. of the 2002 Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems. vol. 735

[10] Einziger, G., Friedman, R.: TinyLFU: A highly efficient cache admission policy. In: 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2014). pp. 146–153

[11] Einziger, G., Fellman, B., Kassner, Y.: Independent counter estimation buckets. The -34th Annual IEEE International Conference on Computer Communications (INFOCOM 2015), Hong Kong, PR China (2015)
[12] Einziger, G., Friedman, R.: Counting with TinyTable: Every Bit Counts! International Conference on Distributed Computing and Networking (ICDCN 2016)
[13] Estan, C., Varghese, G.: New directions in traffic measurement and accounting. SIGCOMM Comput. Commun. Rev. 32(4) (Aug 2002)
[14] Ficara, D., Pietro, A.D., Giordano, S., Procissi, G., Vitucci, F.: Enhancing counting bloom filters through huffman-coded multilayer structures. IEEE/ACM Trans. Netw. 18(6), 1977–1987 (2010)
[15] Garcia-Teodoro, P., Daz-Verdejo, J.E., Maci-Fernndez, G., Vzquez, E.: Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers and Security pp. 18–28 (2009)
[16] Hung, R.Y.S., Lee, L., Ting, H.: Finding frequent items over sliding windows with constant update time. Inf. Proc. Let.10' 110(7), 257–260
[17] Kabbani, A., Alizadeh, M., Yasuda, M., Pan, R., Prabhakar, B.: Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In: Proc. of the 18th IEEE Symposium on High Performance Interconnects. HOTI (2010)
[18] Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. 28(1) (Mar 2003)
[19] Lee, L., Ting, H.F.: A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In: Proc. of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006. pp. 290–297 (2006)
[20] Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proc. of the Int. Conf. on V.L. Data Bases. VLDB (2002)
[21] Metwally, A., Agrawal, D., Abbadi, A.E.: Efficient computation of frequent and top-k elements in data streams. In: IN ICDT (2005)
[22] Mukherjee, B., Heberlein, L., Levitt, K.: Network intrusion detection. Network, IEEE 8(3) (May 1994)
[23] Sommers, J., Barford, P., Duffield, N., Ron, A.: Accurate and efficient sla compliance monitoring. In: Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications. SIGCOMM, ACM (2007)
[24] Tsidon, E., Hanniel, I., Keslassy, I.: Estimators also need shared values to grow together. In: INFOCOM 2012. pp. 1889–1897