Finding Hierarchical Heavy Hitters in Data Streams

Graham Cormode[*]
Rutgers University
graham@dimacs.rutgers.edu

Flip Korn
AT&T Labs-Research
flip@research.att.com

S. Muthukrishnan[†]
AT&T Labs-Research
muthu@research.att.com

Divesh Srivastava
AT&T Labs-Research
divesh@research.att.com

Abstract

Aggregation along hierarchies is a critical summary technique in a large variety of online applications including decision support, and network management (e.g., IP clustering, denial-of-service attack monitoring). Despite the amount of recent study that has been dedicated to online aggregation on sets (e.g., quantiles, hot items), surprisingly little attention has been paid to summarizing hierarchical structure in stream data.

The problem we study in this paper is that of finding Hierarchical Heavy Hitters (HHH): given a hierarchy and a fraction ϕ, we want to find all HHH nodes that have a total number of descendants in the data stream larger than ϕ of the total number of elements in the data stream, after discounting the descendant nodes that are HHH nodes. The resulting summary gives a topological "cartogram" of the hierarchical data. We present deterministic and randomized algorithms for finding HHHs, which builds upon existing techniques by incorporating the hierarchy into the algorithms. Our experiments demonstrate several factors of improvement in accuracy over the straightforward approach, which is due to making algorithms hierarchy-aware.

1 Introduction

Aggregation along hierarchies is a critical summary technique in a large variety of online applications including decision support (e.g., OLAP), network management (e.g., IP clustering, denial-of-service attack monitoring), text (i.e., on prefixes of strings occurring in the text) and XML summarization (i.e., on prefixes of root-to-leaf paths in the XML data tree). In these applications, the data is inherently hierarchical and one needs to maintain aggregates at different levels of the hierarchy over time in a dynamic fashion. Below we describe two applications from network management that motivate this need.

Network-Aware Clustering: The goal of network-aware clustering is to identify "groups" (e.g., hosts under the same administrative domain) based on access patterns, in particular, those responsible for a significant portion of a Web site's requests (measured in terms of the number of IP flows). Knowledge of *busy* clusters can be very useful for packet forwarding in routers, QoS differentiation, proxy positioning, and server replication [18]. Algorithms for network-aware clustering are based on longest prefix match of IP addresses, and have been studied in a dynamic setting [2]. Such busy clusters can occur at multiple levels of the IP address hierarchy, after discounting descendant busy clusters. ∎

DoS Attack Monitoring: In a (distributed) denial-of-service SYN-flooding attack, attackers blast SYN packets (to initiate TCP sessions) at a victim without subsequently acknowledging the victim's SYN-ACK packets with ACK packets to complete the "threeway handshake", using up the resources of the victim. Such a DoS attack may be detected when there is a large

[*] Supported by NSF ITR 0220280 and NSF EIA 02-05116

[†] This author is also with Rutgers University. Supported by NSF CCR 0087022, NSF ITR 0220280 and NSF EIA 02-05116.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

disparity between the number of SYN and ACK packets received by a host, and can be used to monitor both the destination IP addresses (victims) under attack, and also the source IP addresses (attackers). This can be done by maintaining statistics, for IP address prefixes at different levels of aggregation, of the ratio of the number of ACK and SYN packets. Indeed, automated aggregate-based congestion control mechanisms (e.g., based on prefix subnets) have been proposed for regulating Internet traffic to protect against such DoS attacks [21]. ■

A *heavy hitter* (HH) is an element whose frequency in a data set is no smaller than a user-supplied threshold; the problem of finding HHs in data streams has been studied extensively [3, 6, 16, 22, 4]. These algorithms maintain summary structures that allow element frequencies to be estimated, within a prespecified error bound; the algorithms differ in whether they make deterministic or probabilistic guarantees on the error bound, and whether they operate over insert-only streams or streams where elements can be inserted and also deleted.

The problem we study in this paper is that of finding *hierarchical heavy hitters* (HHH) in data streams: given a hierarchy and a fraction ϕ, we want to find all nodes in the hierarchy that have a total number of descendant elements in the data stream no smaller than ϕ of the total number of elements in the data stream, after discounting descendant nodes that are HHH nodes themselves. This is a superset of the heavy hitters consisting of only data stream elements, but a subset of the heavy hitters over all prefixes of all elements in the data stream. It thus provides a topological "cartogram" of the hierarchical data in the stream.

A naive way of computing HHHs, using existing techniques for maintaining heavy hitters, would be to find heavy hitters over all prefixes of all elements in the data stream; the desired HHHs can be determined in a post-processing step. We argue that this approach can be considerably improved in practice (in terms of the space used, or the answer quality) by incorporating knowledge of the hierarchy into algorithms for computing heavy hitters.

In this paper, we develop such *direct* approaches for two versions of the problem. The first problem deals with insert-only streams; this is appropriate for the network-aware clustering application, where each IP flow contributes to an element in the data stream. The second problem deals with streams where elements can be inserted and also deleted; this is appropriate for the DoS attack monitoring application, where SYN packets can be treated as insertions of elements in the data stream, and ACK packets can be treated as deletions. For the former problem, we present algorithms that maintain sample-based summary structures, with deterministic error guarantees for finding HHHs. For the latter problem, we present a randomized algorithm for finding HHHs, with probabilistic guarantees, using sketch-based summary structures. Our experiments demonstrate several factors of improvement in space and accuracy over straightforward approaches, due to the hierarchy-aware nature of our algorithms.

2 Problem Definition

In this section, we formally define the hierarchical heavy hitters problem. We then discuss approaches to solving this problem, under two different models of data streams.

2.1 The Problem

We first review the definition of heavy hitters.

Definition 1 (Heavy Hitter) *Given a (multi)set S of size N and a threshold ϕ, a* Heavy Hitter *(HH) is an element whose frequency in S is no smaller than $\lfloor \phi N \rfloor$. Let f_e denote the frequency of each element e in S. Then $HH = \{e \mid f_e \geq \lfloor \phi N \rfloor\}$.* ■

The *heavy hitters problem* is that of finding all heavy hitters, and their associated frequencies, in a data set. (In any data set, there are no more than $1/\phi$ heavy hitters.) This problem is solved exactly over a stored data set, using the SQL query:

```
SELECT   S.elem, COUNT(*)
FROM S
GROUP BY S.elem
HAVING   COUNT(*) >= ⌊φN⌋
```

In the data stream model of computation, where each data element in the stream can be examined only once, it is not possible to keep exact counts for each data element without using a large amount of space. To use only small space, the paradigm of approximation is adopted, to output only items that occur with a proportion between $(\phi - \epsilon)$ and ϕ. The problem of finding HHs in data streams has been studied extensively (see [4] for a brief survey), based on the maintenance of summary structures that allow element frequencies to be estimated.

Definition 2 (Hierarchical Heavy Hitter) *Given a (multi)set S of elements from a hierarchical domain D of height h, let elements(T) be the union of elements that are descendants of a set of prefixes T of the domain hierarchy. Given a threshold ϕ, we define the set of* Hierarchical Heavy Hitters *of S inductively. HHH_0, the hierarchical heavy hitters at level zero, are simply the heavy hitters of S. Given a prefix p at level i in the hierarchy, define $F(p)$ as $\sum f(e) : e \in elements(\{p\}) \wedge e \notin elements(\cup_{\ell=0}^{i-1} HHH_\ell)$. HHH_i is the set of Hierarchical Heavy Hitters at level i, that is, the set $\{p \mid F(p) \geq \lfloor \phi N \rfloor\}$. The set of Hierarchical Heavy Hitters, HHH, is $\bigcup_{i=0}^{h} HHH_i$.* ■

Note that, since $\sum_p F(p) = N$, the number of hierarchical heavy hitters is no more than $1/\phi$.

Consider an example consisting of a multiset S of 32-bit IP addresses. Such an example might arise in the network-aware clustering application, where the IP addresses are the source IP addresses associated with individual Web requests. Let the counts of descendants, associated with (some of the) IP address prefixes in S, with $N = 100,000$ elements, be as follows: $135.207.50.250/24(2003)$, $135.207.50.250/25(1812), 135.207.50.250/26(1666)$, $135.207.50.250/27(1492), 135.207.50.250/28(1234)$, $135.207.50.250/29(1001), 135.207.50.250/30(767)$, $135.207.50.250/31(404)$ and $135.207.50.250/32(250)$, where $ipaddr/b(c)$ indicates that the IP address prefix obtained by taking the leading b bits of the IP address $ipaddr$ has a descendant leaf count of c. Using $\phi = 0.01$, only $135.207.50.250/29$ and $135.207.50.250/24$ are HHHs, the first because its descendant count exceeds the threshold $(100,000 * 0.01 = 1000)$, and the latter because its descendant count, after discounting the count associated with its descendant HHH 135.207.50.250/29 also exceeds the threshold.

Note that HHHs can include elements in the input, as well as their prefixes, and a prefix may be a heavy hitter without any of its descendant elements being a heavy hitter. In the above example, the (leaf) element 135.207.50.250/32 is not a HHH, but its prefix 135.207.50.250/29 is a HHH. Finding heavy hitters consisting of only elements would hence return too little information. Finding heavy hitters over all prefixes of all elements would return too much information, of little value. This would be a superset of the HHHs, containing not just the HHHs, but also *each* of its prefixes in the hierarchy. In the above example, this would return all 29 prefixes of 135.207.50.250/29, not all of which are of interest.

The *hierarchical heavy hitters problem* we study in this paper is that of finding all hierarchical heavy hitters, and their associated frequencies, in a data stream. The HHH problem cannot be solved exactly over data streams in general. Hence, we study the following (approximate) problem in this paper:

Definition 3 (HHH Problem) *Given a data stream S of elements from a hierarchical domain D, a threshold $\phi \in (0,1)$, and an error parameter $\epsilon \in (0,\phi)$, the* Hierarchical Heavy Hitter Problem *is that of identifying prefixes $p \in D$, and estimates f_p of their associated frequencies, on the first N consecutive elements S_N of S to satisfy the following conditions (i)* accuracy: $f_p^* - \epsilon N \leq f_p \leq f_p^*$, *where f_p^* is the true frequency of p in S_N. (ii)* coverage: *All prefixes q* not *identified as approximate HHHs have $\sum f_e^* : e \in elements(\{q\}) \wedge e \notin elements(P) < \lfloor \phi N \rfloor$, for any supplied $\phi \geq \epsilon$, where P is the subset of p's which are descendants of q.* ■

The above definition only pertains to correctness and does not say anything about the *goodness* of a solution to the HHH problem. For example, a set of heavy hitters would satisfy this definition, as would the full domain hierarchy, but these are not likely to be good solutions. Rather, a good solution is one that satisfies correctness in small space. (We use this metric to evaluate the algorithms in Section 3.) This is for two reasons. First, for semantics, we want to weed out superfluous information (e.g., the above example illustrates how heavy hitters provides too much information, of little value). Second, for efficiency, we want to minimize the amount of space and time required for processing over a data stream. The above notion of correctness closely corresponds to our definition of Hierarchical Heavy Hitters:

Proposition 1 *The size of the set of Hierarchical Heavy Hitters is the size of the smallest set that satisfies the correctness conditions of Definition 3.* ■

Note that any data structure that can satisfy the accuracy constraints for $\phi = \epsilon$ will satisfy it for all $\phi \geq \epsilon$.

2.2 Solution Approaches

A straightforward way of solving the HHH problem, using existing techniques for maintaining heavy hitters, would be to find heavy hitters and frequency estimates over all prefixes of all elements in the data stream; the desired HHHs can be determined in a post-processing step. Such an approach suffers from several problems. First, it could take a lot of space to maintain summary information *independently* for each level in the hierarchy. (Indeed, this is confirmed by our experiments in Section 3.) Second, the update costs could be more expensive, partly due to the extra space required.

We develop *direct, hierarchy-aware* solutions for the HHH problem, under two different data stream models.

- One model is the insert-only model of data streams, also known as *cash-register* model [26], where data stream elements cannot be deleted. For this data stream model, we propose deterministic algorithms that maintain sample-based summary structures, with worst-case error guarantees for finding HHHs. Some of them can be shown to have an *a priori* space bound.

- The second model, the *turnstile model* [26], permits both insertions and deletions of elements in the data stream. For this problem, we present a randomized algorithm for finding HHHs, with probabilistic error guarantees, using a sketch-based summary structure that occupies a bounded amount of space.

Our algorithms are general, and work on any hierarchy, whether it is provided explicitly or implicitly (e.g., the hierarchy is a full binary tree). We discuss these algorithms in more detail in the next few sections, and experimentally demonstrate several factors of improvement they achieve in space and accuracy over the straightforward approaches. For exposition, we consider the case where the data stream elements are leaves in the domain hierarchy.

3 Sample-based Approach

In this section, we present deterministic algorithms for finding HHHs in insert-only data streams. Here the user supplies error parameter ϵ in advance and can supply any threshold ϕ at runtime to output ϵ-approximate HHHs above this threshold.

An indirect way to find HHHs, based on existing techniques, would be to first find heavy hitters, and their associated frequency estimates, over all prefixes of all elements in the data stream. The `LossyCount` algorithm from [22] can be used as a "black box" independently at each level of the domain hierarchy. This requires $O(\frac{1}{\epsilon}\log \epsilon N)$ space at each level resulting in a total of $O(\frac{h}{\epsilon}\log \epsilon N)$ space. The desired HHHs can be extracted in post-processing as follows. The tuples are scanned in postorder across levels. Let (f_p, Δ_p) denote the auxiliary information associated with prefix p (see [22] for details). During the scan we maintain the sum of f_e's among the HHH children $\{e\}$ of each prefix p, denoted F_p. If $(f_p + \Delta_p - F_p \geq \lfloor \phi N \rfloor)$, then we output p as a HHH and reset the F_p summator.

As we shall see, this naive approach is inefficient. Hence, we propose *direct, hierarchy-aware* strategies. We describe these below, prove their correctness, and present runtime analysis. Our experiments verify that the direct strategies perform significantly better than the indirect one.

3.1 Framework and Notation

Our algorithms maintain a trie data structure T consisting of a set of tuples which correspond to samples from the input stream; initially, T is empty. Each tuple t_e consists of a prefix e that corresponds to elements in the data stream. If $t_a(e)$ is the parent of t_e, then $a(e)$ is an ancestor of e in the domain hierarchy, that is, $a(e)$ is a prefix of e. Associated with each value is a bounded amount of auxiliary information used for determining lower- and upper-bounds on the frequencies of elements whose prefix is e ($f_{min}(e)$ and $f_{max}(e)$, respectively). The input stream is conceptually divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$; we denote the current bucket number as $b_{current} = \lfloor \epsilon N \rfloor$. There are two alternating phases of the algorithms: insertion and compression. During compression, the space is reduced via merging auxiliary values and deleting. The procedures for insertion and compression vary from strategy to strategy and are described in more detail below. At any point, we can extract and output HHHs given user-supplied ϕ. Next, we describe the strategies using this framework and give algorithms for insertion, compression and output for each.

3.2 The Strategies

We now describe each of our four strategies in turn.

Strategy 1:

We maintain auxiliary information (g_p, Δ_p) associated with each item p, where the g_p's are *frequency differences* between p and its children $\{e\}$ (specifically, $g_p = f_{min}(p) - \sum_e f_{min}(e)$). This allows for fewer insertions because, unlike the naive approach where we insert all prefixes for each stream element, here we only need to insert prefixes until we encounter an existing node in T corresponding to the inserted prefix. This is an immediate benefit due to being "hierarchy-aware". We can derive $f_{min}(p)$ by summing up all g_e's in the subtree of t_p in T; $f_{max}(p)$ is obtained from $f_{min}(p) + \Delta_p$.

During compression, we scan through the tuples in postorder and delete nodes satisfying $(g_e + \Delta_e \leq b_{current})$ that have no descendants. Hence, T is a complete trie down to a "fringe". Figure 1 gives the algorithm. All $t_q \notin T$ must be below the fringe and, for these, $g_q \equiv f_{min}(q)$. Any pruned nodes t_q must have satisfied $(f_{max}(q) \leq b_{current})$ due to the algorithm, which gives the criteria for correctness:

$$f_q^* - \sum_p f_p^* \leq f_{max}(q) - \sum_p f_{min}(p) = g_q + \Delta_q \leq \lfloor \epsilon N \rfloor .$$

Since the values of g_p in the fringe nodes of T are the same as $f_{min}(p)$, the data structure for Strategy 1 uses exactly the same amount of space as the naive strategy. Therefore,

Proposition 2 *For a given ϵ, Strategy 1 finds HHHs using $O(\frac{h}{\epsilon}\log(\epsilon N))$ space.* ∎

The `Output` function for this strategy takes ϕ as a parameter and chooses a subset of the prefixes in T satisfying correctness. The algorithm is fairly straightforward and is described in Figure 1. The same output function is used for all the strategies proposed in this section.

Strategy 2:

We now consider another benefit of being hierarchy-aware. Let $\{d(e)\}$ denote the deleted descendants of a node t_e. We observe that one can improve the bounds on the Δ_e's by keeping track of the maximum $(g_{d(e)} + \Delta_{d(e)})$ over all $d(e)$'s; we denote this statistic as m_e. Thus, the auxiliary information associated with each element e is (g_e, Δ_e, m_e), where g_e and Δ_e are defined as before. Figure 2 presents the algorithm.

```
Insert (e,c):
   if $t_e$ exists then
      $g_e += c$;
   else
      Insert (p(e),0);
      create $t_e$;
      $g_e = c$;
      $\Delta_e = b_{current} - 1$;

Compress:
   for each $t_e \in T$ in postorder do
      if (($t_e$ has no descendants)
      && ($g_e + \Delta_e \leq b_{current}$)) then
         $g_{p(e)} += g_e$;
         delete $t_e$;

Output(e, $\phi$):
/* $G_e = \sum_x g_x$ of HHH descendants of $e$ */
   let $G_e = g_e$ for all $e$;
   for each $t_e$ in postorder do
      if ($G_e + \Delta_e \geq \lfloor \phi N \rfloor$) then
         print(e);
      else
         $G_{p(e)} += G_e$;
```

Figure 1: Algorithm for Strategy 1

We can show that $m_e < b_{current}$ as follows. The statistic m_e maintains the largest value of $(g_{d(e)} + \Delta_{d(e)})$ over all deleted $d(e)$'s. Thus, any new stream element that has e as a prefix could not possibly have occurred with frequency more than m_e. Suppose $d(e)$ was deleted just after block $b' < b_{current}$. Hence, $(g_{d(e)} + \Delta_{d(e)})$ must have been less than b' at the time of deletion and therefore $(g_{d(e)} + \Delta_{d(e)}) \leq b_{current}$. Since the only difference between this strategy and Strategy 1 is that Δ_e's are initialized to $m_{p(e)}$ rather than $(b_{current} - 1)$, Strategy 2 cannot contain more tuples than Strategy 1. This yields the following result:

Proposition 3 *For a given ϵ, Strategy 2 finds HHHs in $O(\frac{h}{\epsilon}\log(\epsilon N))$ space.* ∎

Strategy 3:

The motivation for this strategy is to allow intermediate nodes of T, as well as nodes without descendants, to be deleted. The auxiliary information associated with each element e is (g_e, Δ_e), where g_e and Δ_e are defined as before. When a new element e is inserted, its Δ_e is initialized using the auxiliary information of its closest ancestor in T as $g_{a(e)} + \Delta_{a(e)}$, requiring only one operation since none of e's prefixes are inserted. Figure 3 presents the algorithm.

We can show this algorithm is correct as follows. First we show that, for any $t_q \notin T$, $f^*_q - \sum f^*_p \leq b_{current}$, for p's that are children of q in T.

```
Insert (e,c):
/* $t_{a(e)}$ is the parent node of $t_e$ */
   if $t_e$ exists then
      $g_e += c$;
   else if $t_{a(e)}$ exists then
      Insert (p(e),0);
      create $t_e$;
      $g_e = c$;
      $\Delta_e = m_e = m_{a(e)}$;
   else
      Insert (p(e),0);
      create $t_e$;
      $g_e = c$;
      $\Delta_e = m_e = b_{current} - 1$;

Compress:
   for each $t_e \in T$ in postorder do
      if (($t_e$ has no descendants)
      && ($g_e + \Delta_e \leq b_{current}$)) then
         $g_{a(e)} += g_e$;
         $m_{a(e)} = \max(m_{a(e)}, g_e + \Delta_e)$;
         delete $t_e$;
```

Figure 2: Algorithm for Strategy 2

Proposition 4 *$\forall e$ $f_{min}(e) \leq f^*_e \leq f_{max}(e)$, at all timesteps.*

Proof (Sketch). By induction on $b_{current}$.
Basis: ($b_{current} = 1$) $\forall e$ $\Delta_e = 0$ and, hence, $f_{min}(e) = f_{max}(e) = f^*_e$.
Induction Step: During compression, there is no change of values $(f_{min}(e), f_{max}(e))$, for all $e \in T$. Consider a new node t_e that has been inserted with $g_e \equiv 1$ and $\Delta_e \equiv g_{a(e)} + \Delta_{a(e)}$; the value of Δ_e ensures that $f_{min}(e) \leq f^*_e \leq f_{max}(e)$. ∎

Proposition 5 *If $t_q \notin T$, then $f^*_q - \sum_p f^*_p \leq b_{current}$.*

Proof. When an element e is deleted, $g_e + \Delta_e \leq b_{current}$. This remains true for items after deletion (until inserted again). As we showed above, $\forall e$ $f^*_e \leq f_{max}(e)$, at all times. Recall that $g_q \equiv f_{min}(q) - \sum_p f_{min}(p)$, where $f_{min}(p) \leq f^*_p$ and $f_{min}(q) \leq f^*_q$. Hence,

$$\begin{aligned} f^*_q - \sum_p f^*_p &\leq f_{max}(q) - \sum_p f_{min}(p) \\ &= (f_{min}(q) + \Delta_q) - \sum_p f_{min}(p) \\ &= \left(f_{min}(q) - \sum_p f_{min}(p) \right) + \Delta_q \\ &= g_q + \Delta_q \leq b_{current}. \end{aligned}$$

∎

```
Insert (e,c):
/* $t_{a(e)}$ is the parent node of $t_e$ */
/* $p(e)$ is the immediate prefix of $e$ */
if $t_e$ exists then
    $g_e += c$;
else if $t_{a(e)}$ exists then
    create $t_e$;
    $g_e = c$;
    $\Delta_e = g_{a(e)} + \Delta_{a(e)}$;
else
    create $t_e$;
    $g_e = c$;
    $\Delta_e = b_{current} - 1$;

Compress:
for each $t_e \in T$ in postorder do
    if $(g_e + \Delta_e \leq b_{current})$ then
        if $(|e| > 1)$ then
            Insert($p(e)$, $g_e$);
        delete $t_e$;
```

Figure 3: Algorithm for Strategy 3

Strategy 4:

This hybrid strategy combines ideas from Strategies 2 and 3 by using the control structure of Strategy 3 as a basis and incorporating the auxiliary statistic m_e from Strategy 2 to obtain smaller Δ-values. Figure 4 presents the algorithm.

```
Insert (e,c):
/* $t_{a(e)}$ is the parent node of $t_e$ */
/* $p(e)$ is the immediate prefix of $e$ */
if $t_e$ exists then
    $g_e += c$;
else if $t_{a(e)}$ exists then
    create $t_e$;
    $g_e = c$;
    $\Delta_e = m_e = m_{a(e)}$;
else
    create $t_e$;
    $g_e = c$;
    $\Delta_e = m_e = b_{current} - 1$;

Compress:
for each $t_e \in T$ in postorder do
    if $(g_e + \Delta_e \leq b_{current})$ then
        if $(|e| > 1)$ then
            Insert($p(e)$, $g_e$);
            $m_{p(e)} = \max(m_{p(e)}, g_e + \Delta_e)$;
        delete $t_e$;
```

Figure 4: Algorithm for Strategy 4

3.3 Experiments

We ran experiments to compare the proposed strategies on a variety of real and synthetic data sets in terms of space usage as well as runtime costs. We examined the effect of the depth of the hierarchy by varying the granularity of the path strings (e.g., bit- or byte-level); the "bushiness" of the hierarchy, as determined by the address space at each node in the paths (e.g., octets for IP addresses); and the *a priori* worst-case error ϵ.

3.3.1 Setup

We used the following real data sets: (1) TCPSRC, 140K source IP addresses for TPC sessions, as captured using Gigascope [5]; and (2) FLOWDEST, 100K destination IP addresses for flows passing through an edge router, as captured using the Cisco NetFlow option [27].

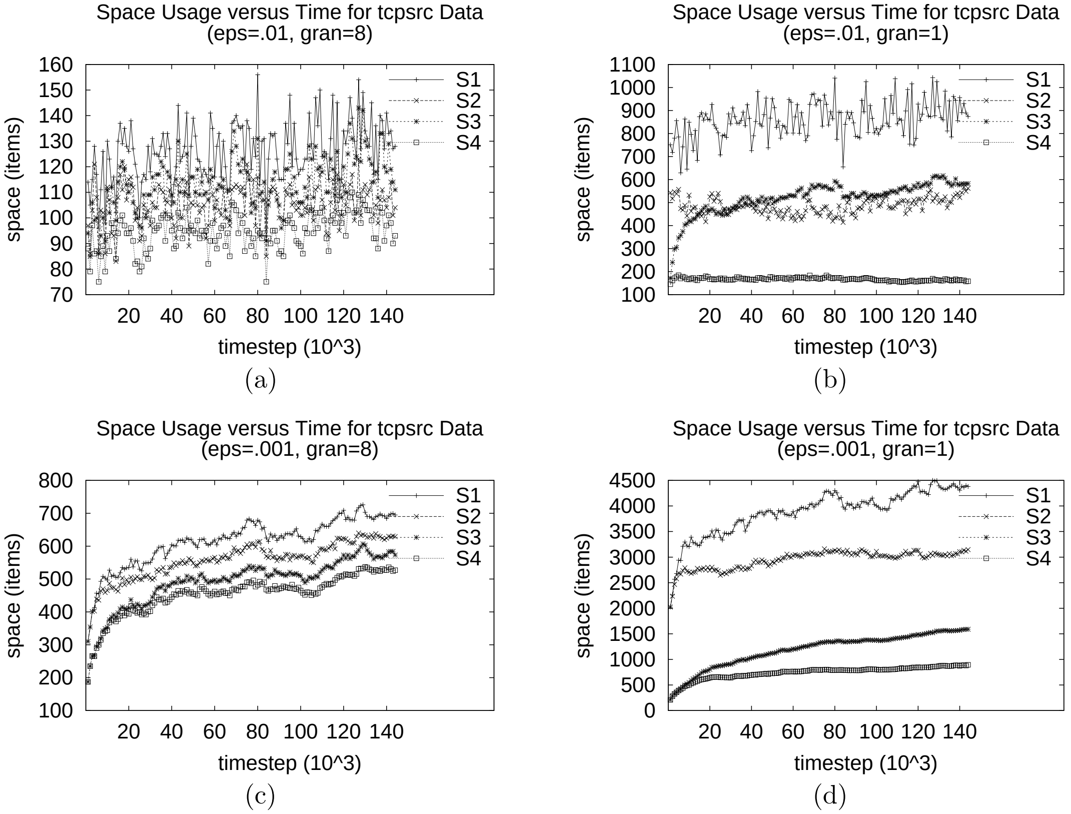
3.3.2 Comparison of Strategies

In the first set of experiments, we report the *a posteriori* space utilization in terms of the number of tuples at each timestep. Figure 5 gives a comparison of the four strategies on TCPSRC. The left column considered byte-level prefixes (granularity = 8) and the right column bit-level prefixes (granularity = 1). The top row was run with $\epsilon = .01$; the bottom row was run with $\epsilon = .001$. Recall that the naive strategy and Strategy 1 use the same amount of space; therefore, we do not report the numbers from the naive strategy in Figure 5. Figure 6 compares the strategies using the FLOWDEST data, with $\epsilon = .01$ and granularity = 8.

The observations are as follows:

- Strategy 2 always does better than Strategy 1;
- Strategy 3 does well when ϵ is small and does considerably better than Strategy 2 when the hierarchy is deep and thin. However, Strategy 3 did worse than Strategy 2 in Figure 5(b) and in Figure 6;
- Strategy 4 (Hybrid) did the best in all cases. It is the strategy of choice.

Figure 7 compares the speed of the strategies by measuring the total number of insertions, deletions and updates to the summary structure. The data set used was TCPSRC and the operations were totaled after 140K timesteps, with granularity = 8 and $\epsilon = 0.1$. It presents this breakdown as histogram bars where the height gives the sum of all operations. "Strategy 0" represents the naive (blackbox) approach, which requires far more updates than the other strategies because every prefix of every element is inserted. The differences between Strategies 1-4 are small. Strategy 2 performs more operations than Strategy 1 due to the extra pruning. Strategy 4 performs more operations than Strategy 3 due to the extra pruning.



(a) (b)

(c) (d)

Figure 5: Comparison of the strategies on TCPSRC. Left column is for shallow/bushy hierarchy; right column is for deep/thin hierarchy; ϵ decreases from top to bottom.

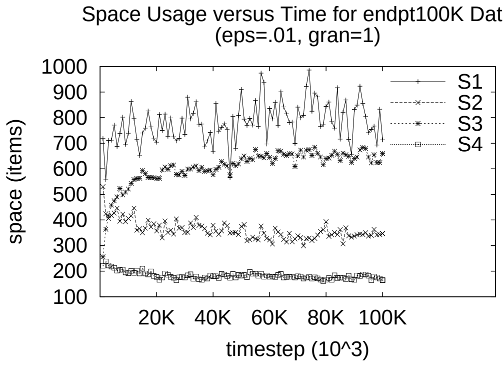


Figure 6: Comparison of the strategies on FLOWDEST data, with $\epsilon = .01$ and granularity $= 1$.

3.4 Extensions

So far we have assumed that data stream elements are leaves of the domain hierarchy. Our algorithms can be extended easily to allow prefixes as input elements in the data stream, by explicitly maintaining additional counts with each tuple in the summary structure, and using these counts suitably.

4 Sketch-based Approach

Informally, we use *sketch* here to refer to a data structure on a distribution $A[1 \cdots U]$ where $A[i]$ is the number of times i is seen in the data stream. It has the following properties: it uses small space, can be maintained efficiently as new items are seen in the data stream, and can be used to estimate parts of the distribution A to some precision with high probability. Now, the performance and choice of sketches depends on (a) whether items are only inserted, or they are both inserted and deleted, (b) whether one seeks rangesum $\sum_{k=i}^{k=j} A[k]$ or only point estimates in which case $i = j$, (c) the precision desired and required probability of success, and (d) whether the data stream is well-formed or not. A data stream is *well-formed* if $A[i] \geq 0$ at all times and *ill-formed* otherwise. In general one expects data streams to be well formed because one does not delete an item unless it was inserted earlier. However, sometimes, as an artifact of subtractions performed by algorithms that use sketches, the underlying data stream may be inferred to be ill-formed. Many different sketches are known in the literature that tradeoff space and update times for the features above.

Sketches are used to build high level algorithms. One such procedure that is of interest to us is that to

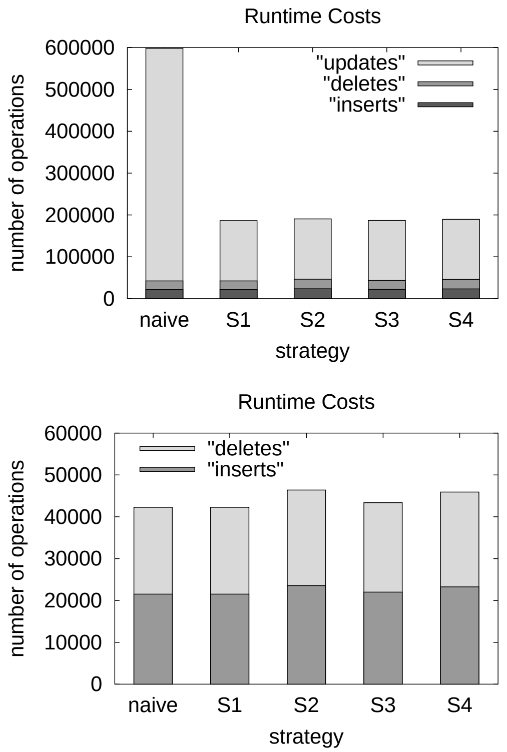


Figure 7: Comparing the runtime performance of the strategies on TCPSRC data.

determine the Heavy Hitters (HHs) on a data stream. Again there are many such algorithms [3, 13, 4, 7] and they differ in their performance and applicability based on the sketches they use.

Our algorithm here will use sketches too. In fact, one can think of a few different ways to estimate HHHs using sketches and HH algorithms, but the central challenge is to find one that is most efficient and is simple. The sketch method we describe allows the probabilistic solution to the hierarchical heavy hitter problem, in the model where the input consists of a sequence of insertions and deletions of items. Note that the deterministic algorithms described above do not solve this problem, and that they will produce incorrect output on these more general kinds of data streams.

To motivate the discussion, let us consider a rather natural algorithm for finding HHHs. It relies on a bottom-up traversal of the tree. The algorithm first considers the nodes at depth h denoted S_h, and determines all HHs, denoted H_h. For each node $i \in H_h$, we determine its parent $j \in S_{h-1}$ and subtract (the estimate for) f_i from f_j. Following that, we determine HHs in S_{h-1} using these updated frequencies, and recurse up the tree. This algorithm will work, but it needs a powerful sketch function. In particular, the sketch needs to work not only with insert and delete of items, but also with a potentially ill-formed data stream. This is because when we update frequency of a node by subtracting an estimate for its child, the result may well be negative if the child's frequency was an overestimate; this effect is likely to be more severe if a node has several children or descendants that are HHs. This in turn implies that the HH algorithm must also work for ill-formed sequences. Sketches with this property exist, e.g., in [11]; they use inner products of the data stream with random vectors generated by 4-wise independent random variables, and a fairly sophisticated "group testing" procedure atop. The total space used is $\Omega(h^2)$ with significantly large constants in the $\Omega()$. Furthermore, the accuracy of estimating HHHs decreases dramatically as the algorithm progresses up the tree. We refer to this approach as the naive sketch strategy.

```
Insert $(e, c)$:
  $n = n + 1$;
  For each level $l$ of the hierarchy
     For $i = 1$ to $3\log(1/\delta)$, $j = 1$ to $8/\epsilon^2$
        If $(f_{i,j}(e) = 1)$
           $sum[l][i][j] = sum[l][i][j] + 1$;
     $e = p(e)$;

Delete $(e, c)$:
  $n = n - 1$;
  For each level $l$ of the hierarchy
     For $i = 1$ to $3\log(1/\delta)$, $j = 1$ to $8/\epsilon^2$
        If $(f_{i,j}(e) = 1)$
           $sum[l][i][j] = sum[l][i][j] - 1$;
     $e = p(e)$;

Weight $(p, l)$:  returns a value
  for $i$ = 1 to $3\log 1/\delta$
     $t = 0$;
     for $j = 1$ to $8/\epsilon^2$
        $t = t + f_{i,j}(p) * (2 * sum[l][i][j] - n)$;
     $avg[i] = t * \epsilon^2$;
  return median($avg$);

Output($\phi$, $e$, $l$):  returns a value
  $w$ = Weight($e$,$l$);
  if $w < \lfloor \phi n \rfloor$
     return 0;
  else for each child $c$ of $e$
     $W = W$ + Output($\phi, c, l + 1$);
     if $(w - W \geq \lfloor \phi n \rfloor)$
        print($e$);
        return($w$);
     else
        return($W$);
```

Figure 8: Sketch-based Algorithm

In what follows, we will describe an algorithm which is very simple: it relies on sketches that use only pairwise independence and does not need any HH algorithm. As a result, it is quite efficient among such algorithms.

We will proceed as follows. First we will describe a procedure to find hierarchical heavy hitters using sketches as an oracle, in which we assume that we can recover the exact frequency of a range using sketches. We will then discuss how to build sketches which have tunable error parameter, and show how they perform in practice on real data sets.

4.1 Search Algorithm

We imagine that we have sketches that are able to return the frequency of a contiguous range of leaf elements, and describe how to use this primitive in order to find Hierarchical Heavy Hitters. Let the current number of items be n. On receiving a new item, we update the sketches to reflect this, and increment the total count n. To find the hierarchical heavy hitters, we perform a top down search of the hierarchy, beginning at the root node. The search proceeds recursively, and the recursive procedure run on a node returns the total weight of all hierarchical heavy hitters that are descendents of that node.

- Compute w the weight of the current node, as the range sum of all leaf nodes beneath it.
- If $w < \lfloor \phi n \rfloor$ then return 0.
- Else, recursively find W: the sum of the weights of HHHs within any child nodes of current node.
- If $w - W \geq \lfloor \phi n \rfloor$ then the current node is a HHH, output it, and return w; else, return W.

The above procedure works because of the observation that if there is a HHH in the hierarchy below a node, then the range sum of leaf values must be no less than the threshold of $\lfloor \phi n \rfloor$. We then include any node that meets the threshold, after the weight of any HHHs below has been removed. The number of queries made to sketches depends on the height of the hierarchy, h, the maximum branching factor of the hierarchy, d, and the frequency parameter ϕ as hd/ϕ, which governs the running time of this procedure.

4.2 Sketch Construction

The sketch needed for the algorithm above needs only to work with insert and delete of items, and be able to estimate the frequency of each node in the tree. This is a rather simple requirement, and we will use the Random Subset Sums introduced in [13]. These work in the following fashion: we create subsets of the universe so that for any set, the probability that any member of the universe is in that set is $\frac{1}{2}$. We keep a counter for each set, and when a new item arrives, we increment the counters of every set which includes that item. Departures of items can be incorporated by performing the inverse operation: decrement the counters of every set which includes the item. We can then very quickly answer point queries for the frequency of item i with a pass over the set of counters. We observe that, if i is included in the set and the counter for the set is c, then $2c - n$ is an unbiased estimator for the count of i; if i is not in the set, then $n - 2c$ is an unbiased estimator for count of i. By taking the average of $O(1/\epsilon^2)$ such estimates, then the resulting value is correct up to an additive quantity of $\pm\epsilon n$, with constant probability. Taking the median of $O(\log 1/\delta)$ independent repetitions amplifies this to probability $1 - \delta$. A more detailed analysis is given in [13]. If we keep such a sketch for each of the h levels of the hierarchy, then range sums can be computed as point queries. An important detail is how to store the subsets with which the sketches are created. Clearly, explicitly storing random subsets will be prohibitively costly in terms of memory usage. However, for the expectation and variance calculations, we only require that the sets are chosen with pairwise independence. It therefore suffices to use functions drawn from a family of pairwise independent hash functions f mapping from prefixes onto $\{-1, 1\}$, which defines the "random subsets": for set j we compute $f_j(i)$: if the result is 1, then i is included in set j, else it is excluded. Such functions can easily be computed, as the inner product of the binary representation of i with a randomly chosen seed [13]. Putting all this together gives us the following result.

Proposition 6 *Our algorithm uses Random Subset Sums as the sketch to find Hierarchical Heavy Hitters. The space required is $O(\frac{h}{\epsilon^2}\log(1/\delta))$. Searching for the hierarchical heavy hitters requires time $O(\frac{hd}{\phi\epsilon^2}\log(1/\delta))$. With probability at least $1 - \delta$, then the output conforms to the requirements of Definition 3. The time to process an update to the sketches is $O(\frac{h}{\epsilon^2}\log(1/\delta))$.* ■

The pseudo-code to implement this algorithm is presented in Figure 8. An important point here is that the space and running time of this method depends strongly on the space and time of the sketch procedures. Using different sketch constructions would impact on these costs. Future work will examine the benefits and disadvantages of different sketch constructions. We shall compare our methods against the naive, "bottom-up" strategy described above. This method had the disadvantage that it is necessary to keep a heavy hitters data structure for every level of the hierarchy, in addition to the sketches used to estimate the frequency of prefixes. So our sketch strategy uses less space than the naive strategy. Similarly, since the naive strategy has to update the heavy hitter data structure at each level for every insertion or deletion, the time cost is also greater. For the naive algorithm, we make use of the data structure proposed in [4] to find the Heavy Hitters at each level, and we use the same sketch structure to compute estimated counts.

Algorithm	Naive	Naive	Aware	Aware
Hierarchy	Shallow	Deep	Shallow	Deep
$\epsilon = 0.1$	4143	33144	15	120
$\epsilon = 0.01$	66033	582264	2673	21384
$\epsilon = 0.001$	1245033	9960264	389961	3119688

Table 1: Space Usage in machine words for both methods with varying parameters

Formally then,

Proposition 7 *The naive algorithm uses space* $O((\frac{h}{\epsilon^2} + \frac{h^2 \log d}{\phi}) \log(1/\delta))$ *and time per update* $O((\frac{h}{\epsilon^2} + h^2 \log d) \log(1/\delta))$. *The time to find HHHs is* $O((\frac{h}{\epsilon^2} + \frac{h^2 \log d}{\phi}) \log(1/\delta))$. ∎

4.3 Experiments on Sketches

We performed a series of experiments on sketch based methods to evaluate their accuracy, using the same data sets as before. We implemented the two versions of sketch based methods: the "hierarchy aware" sketch methods described in Section 4.1, and the non-hierarchy aware "naive sketch" method. We tried a number of experiments, based on a variety of settings of the sketch parameters, type of hierarchy, and compared how the two methods fared.

We first examined the space usage of the hierarchy aware method against that of the naive one on the TCPSRC data. The space usage for various settings of ϵ, and both shallow and deep hierarchies is shown in Table 1. We see that the naive algorithm uses significantly more memory than the hierarchy aware approach. However, we also note that as the parameter ϵ decreases, the amount of space increase with $O(1/\epsilon^2)$, so that for values of ϵ approaching 0.001, the space overhead becomes significant for both methods, which may be undesirable in some applications where a fine grained accuracy is required.

The results of our experiments on real data are shown in Figure 9. We plot the output size of the method every thousand items. The methods are implemented to guarantee correctness of the output; therefore, the less reliable the estimates, then the more items have to be output in order to make this guarantee. The better a method is, the fewer items will be output to make this guarantee. We make the following observations:

- The hierarchy aware method produces approximately the same amount of output items as the naive method, despite using significantly less memory and having faster update times.

- When we reduce ϵ to being half the value of ϕ, then the uncertainty in the estimated values decreases and so fewer items are output in order to guarantee correctness. When we reduced ϵ further, the output size got arbitrarily close to the 'optimum' output size found by computing the number of exact HHHs.

- We also examined the effect of decreasing the parameter δ (not graphed) to improve the quality output. While this did consistently bring the size of the output closer to the optimal size, it did not seem to make a very large difference, since even with $\delta = 0.01$, the results were very close to the optimal size.

We also note that the size of the output is smaller than the size of the output with deterministic methods for corresponding values of ϵ. However, the amount of space used to achieve this is many, many times larger than for the sampling based methods, so these sketches are not really competitive for the insertions only scenario; it is only when deletions are possible and sample based methods will not work that sketch based solutions are required.

5 Related Work

Finding "heavy hitters" in network measurement data is an important problem for network management and has been considered for TCP sessions [9, 7]. The need for maintaining multilevel heavy hitters on variable-length IP address prefixes was articulated in [10, 18].

There are several related queries in the area of Online Analytical Processing (OLAP). Fang et al. introduced the iceberg query for finding `GROUP BY` aggregates above a supplied threshold [8]; Beyer et al. generalized this to `CUBE BY` ("iceberg cubes") [1]. Lakshmanan et al. describe the generalized MDL approach for characterizing range query results succinctly using hierarchical regions [19]. Lakshmanan et al. proposed the quotient cube summary technique, which partitions the data cube into equivalence classes of cells (at different levels in the hierarchy) with identical aggregate values [20]. The semantics of the summaries resulting from these queries are different from hierarchical heavy hitters.

There has been much recent work on online algorithms for maintaining frequency distributions over a data stream including histograms [14, 15, 12] and hot items [22]. However, almost all of this work applies to sets. For hierarchical data, Kleinberg studied the problem of maintaining "hierarchical bursts" in texts [17].

Our deterministic algorithms use sample-based methods akin to [22, 14, 23, 25, 24] and our randomized algorithms use sketch-based methods akin to [15, 12, 11]. But we build on them with higher level algorithms to determine HHHs.

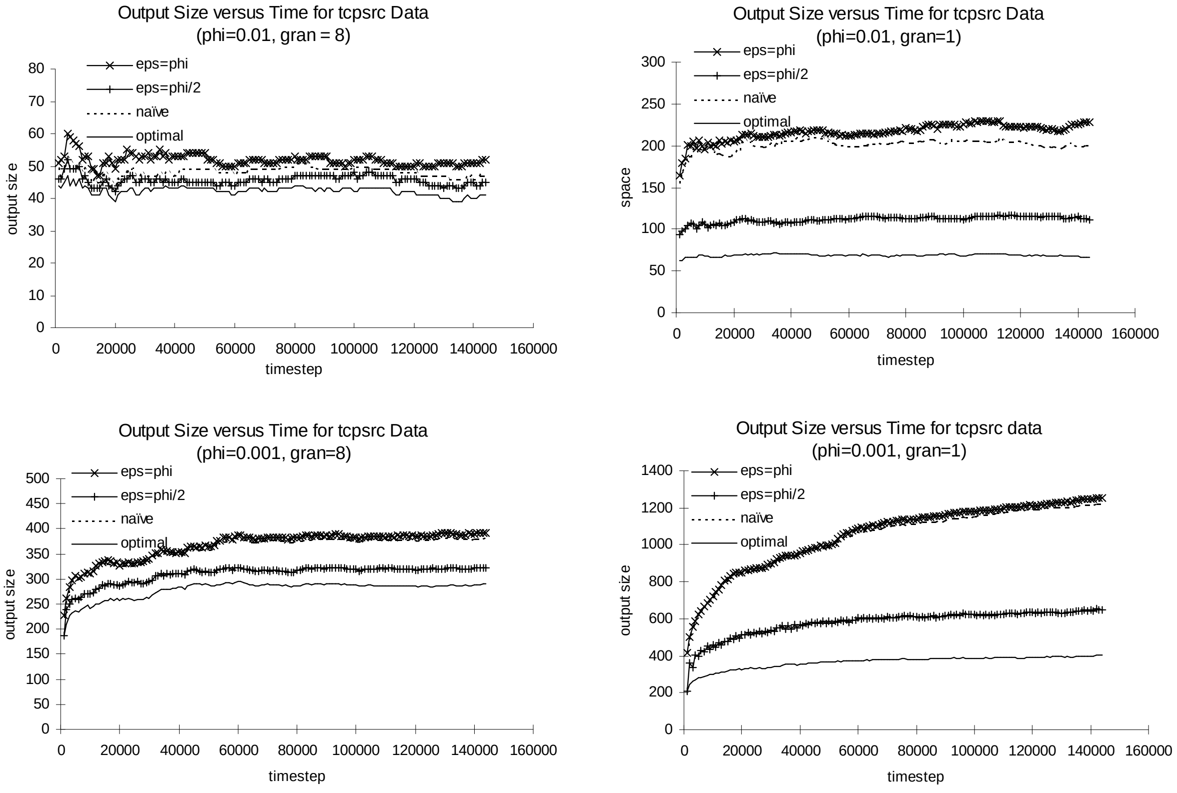


Figure 9: Results of experiments using sketch based methods

6 Conclusions

Aggregation along hierarchies is a critical summary technique in a large variety of applications including OLAP, IP network management, and text or XML summarization. We formalize the problem of finding heavy hitters in massive data streams that considers their hierarchical structure. Such hierarchical heavy hitters (HHHs) present a “cartogram” summary of the data stream distribution.

We present a comprehensive set of solutions to the problem of estimating HHHs on data streams. In particular, we present both deterministic, sample-based and randomized, sketch-based algorithms for efficiently finding HHHs using small space; these respectively work for data streams that allow only insertions, and those that allow both insertions as well as deletions of items. These solutions build upon known data stream summarization methods by explicitly incorporating hierarchy into the algorithms. Our experimental study shows that such hierarchy-aware algorithms significantly outperform the straightforward methods.

A cartogram view of hierarchical data streams is likely to find many applications. We leave it open to generalize our techniques to data streams that have multiple attributes with independent hierarchies.

References

[1] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg CUBE. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 1999.

[2] A. L. Buchsbaum, G. S. Fowler, B. Krishnamurthy, K.-P. Vo, and J. Wang. Fast prefix matching of bounded strings. In *Proceedings of SIAM ALENEX*, 2003.

[3] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Procedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

[4] G. Cormode and S. Muthukrishnan. What’s hot and what’s not: tracking most frequent items dynamically. In *Proceedings of ACM Principles of Database Systems*, 2003.

[5] C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high performance network monitoring with an SQL interface. In *SIGMOD Conference 2002*, page 623, 2002.

[6] E. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 348–360, 2002.

[7] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of ACM SIGCOMM*, pages 323–336, 2002.

[8] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases*, pages 299–310, 1998.

[9] A. Feldmann, A. G. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: methodology and experience. In *Proceedings of ACM SIGCOMM*, pages 257–270, 2000.

[10] T. M. Gil and M. Poletto. MULTOPS: a datastructure for bandwidth attack detection. In *10th USENIX Security Symposium*, pages 23–38, 2001.

[11] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. QuickSAND: quick summary and analysis of network data. Technical Report 2001-43, DIMACS, 2001.

[12] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *International Conference on Very Large Databases (VLDB)*, pages 79–88, 2001.

[13] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: dynamic maintenance of quantiles. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 454–465, 2002.

[14] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 58–66, 2001.

[15] S. Guha, N. Koudas, and K. Shim. Data streams and histograms. In *ACM Symposium on Theory of Computing (STOC)*, pages 471–475, 2001.

[16] R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in sets and bags. *ACM Transactions on Database Systems*, 28:51–55, 2003.

[17] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001.

[18] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proceedings of the ACM SIGCOMM Conference*, pages 97–110, 2000.

[19] L. V. S. Lakshmanan, R. T. Ng, C. X. Wang, X. Zhou, and T. Johnson. The generalized MDL approach for summarization. In *International Conference on Very Large Databases (VLDB)*, pages 766–777, 2002.

[20] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient cube: how to summarize the semantics of a data cube. In *International Conference on Very Large Databases (VLDB)*, pages 778–789, 2002.

[21] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *Computer Communications Review*, 32(3):62–73, 2002.

[22] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *International Conference on Very Large Databases (VLDB)*, pages 346–357, 2002.

[23] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1998.

[24] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 251–262, 1999.

[25] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, pages 315–323, 1980.

[26] S. Muthukrishnan. Data streams: algorithms and applications. In *ACM-SIAM Symposium on Discrete Algorithms*, http://athos.rutgers.edu/∼muthu/stream-1-1.ps, 2003.

[27] Cisco Netflow. http://www.cisco.com/warp/public/-732/netflow/.