Finding Hierarchical Heavy Hitters in Streaming Data

GRAHAM CORMODE
AT&T Labs–Research
and
FLIP KORN
AT&T Labs–Research
and
S. MUTHUKRISHNAN
Rutgers University
and
DIVESH SRIVASTAVA
AT&T Labs–Research

Data items that arrive online as streams typically have attributes which take values from one or more hierarchies (time and geographic location; source and destination IP addresses; etc.). Providing an aggregate view of such data is important for summarization, visualization, and analysis. We develop an aggregate view based on certain organized sets of large-valued regions ("heavy hitters") corresponding to hierarchically discounted frequency counts. We formally define the notion of *Hierarchical Heavy Hitters* (HHHs). We first consider computing (approximate) HHHs over a data stream drawn from a single hierarchical attribute. We formalize the problem and give deterministic algorithms to find them in a single pass over the input.

In order to analyze a wider range of realistic data streams (e.g., from IP traffic monitoring applications), we generalize this problem to multiple dimensions. Here, the semantics of HHHs are more complex, since a "child" node can have multiple "parent" nodes. We present online algorithms that find approximate HHHs in one pass, with provable accuracy guarantees. The product of hierarchical dimensions form a mathematical lattice structure. Our algorithms exploit this structure, and so are able to to track approximate HHHs using only a small, fixed number of statistics per stored item, regardless of the number of dimensions.

We show experimentally, using real data, that our proposed algorithms yield outputs which are very similar (virtually identical, in many cases) to offline computations of the exact solutions whereas straightforward heavy hitters based approaches give significantly inferior answer quality. Furthermore, the proposed algorithms result in an order of magnitude savings in data structure size while performing competitively.

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Data Mining

General Terms: Algorithms, Experimentation, Performance, Theory

Additional Key Words and Phrases: data mining, approximation algorithms, network data analysis

Author's addresses: {graham,flip,divesh}@research.att.com; muthu@cs.rutgers.edu.
Work carried out while first author was at the Center for Discrete Mathematics and Computer Science (DIMACS); Bell Laboratories; and AT&T Labs–Research. The work of the first and third authors was partially supported by NSF ITR 0220280 and NSF EIA 02-05116.

1. INTRODUCTION

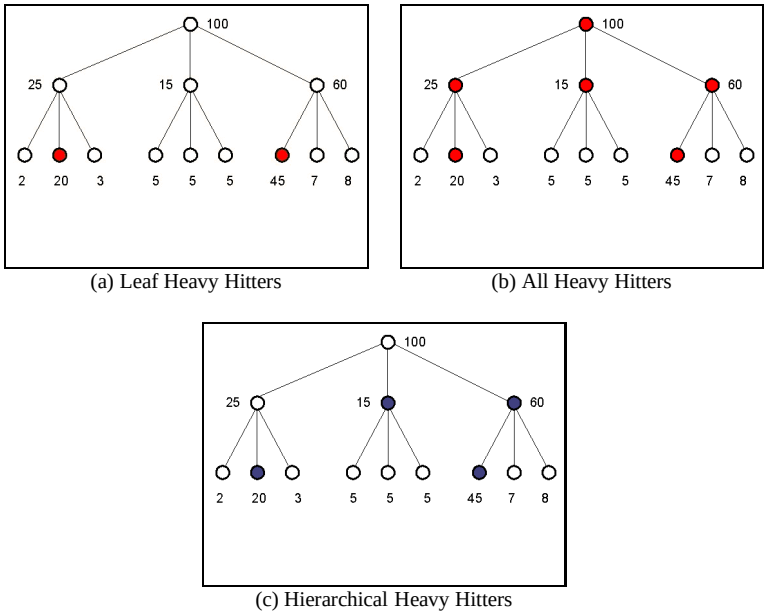
Emerging applications in which data is *streamed* typically have hierarchical attributes. The quintessential example of data streams is IP traffic data such as packets in an IP network, each of which defines a tuple (Source address, Source Port, Destination Address, Destination Port, Packet Size). IP addresses are naturally arranged into hierarchies: individual addresses are arranged into subnets, which are within networks, which are within the IP address space. For example, the address 66.241.243.111 can be represented as 66.241.243.111 at full detail, 66.241.243.* when generalized to 24 bits, 66.241.* when generalized to 16 bits, and so on. Ports can be grouped into hierarchies, either by nature of service ("traditional" Unix services, known P2P file sharing port, and so on), or in some coarser way: in [Estan et al. 2003] the authors propose a hierarchy where the points in the hierarchy are "all" ports, "low" ports (less than 1024), "high" ports (1024 or greater), and individual ports. So port 80 is an individual port which is in low ports, which is in all ports.

Data warehouses also frequently consist of data items whose attributes take values from hierarchies. For example, data warehouses accumulate data over time, so each item (e.g., sales) has a time attribute of when it was recorded. We can view hierarchical attributes such as time at various levels of detail: given transactions with a time dimension, we can view totals by hour, by day, by week and so on. There are attributes such as geographic location, organizational unit and others that are also naturally hierarchical. For example, given sales at different locations, we can view totals by store, city, state, country and so on.

Our focus is on aggregating and summarizing such data. A standard approach is to capture the value distribution at the finest detail in some succinct way. For example, one may use the most frequent items ("heavy hitters"), or histograms to represent the data distribution as a series of piece-wise constant functions. We call these *flat* methods since they focus on one (typically, the finest) level of detail. Flat methods are not suitable for describing the hierarchical distribution of values. For example, an item at a certain level of detail (e.g., first 24 bits of a source IP address) made up by aggregating many small frequency items may be a heavy hitter item even though its individual constituents (the full 32-bit addresses) are not. In contrast, one needs a *hierarchy-aware* notion of heavy hitters. Simply determining the heavy hitters at *each level* of detail will not be the most effective: if any node is a heavy hitter, then all its ancestors are heavy hitters too. For example, if a 32-bit IP address were a heavy hitter, then all its prefixes would be, too.

1.1 One Dimensional Hierarchical Heavy Hitters

We begin by introducing the concept of Hierarchical Heavy Hitters (HHHs) over data drawn from a single hierarchical attribute, before we consider the more general problem on data with multiple hierarchical attributes. Figure 1 shows an example distribution of $N = 100$ items over a simple hierarchy in one dimension, with the counts for each internal node representing the total number of items at leaves of the corresponding subtree. The traditional heavy hitters definition is, given a threshold ϕ, to find all items with frequency at least ϕN. Figure 1 (a) shows that setting $\phi = 0.1$ yields two items with frequency above 10. However, this does not adequately cover the full distribution, and so we seek a definition which also tells us about heavy hitters at points in the hierarchy other than the leaves. A natural approach is to apply the heavy hitters definition at each level of generalization: at the leaves, but also for each internal node. The effect of this definition is shown in Figure 1 (b). But this fails to convey the complexity of the distribution: is a node marked as signifi-



(a) Leaf Heavy Hitters

(b) All Heavy Hitters

(c) Hierarchical Heavy Hitters

Fig. 1. Illustration of HHH concept ($N = 100, \phi = 0.1$)

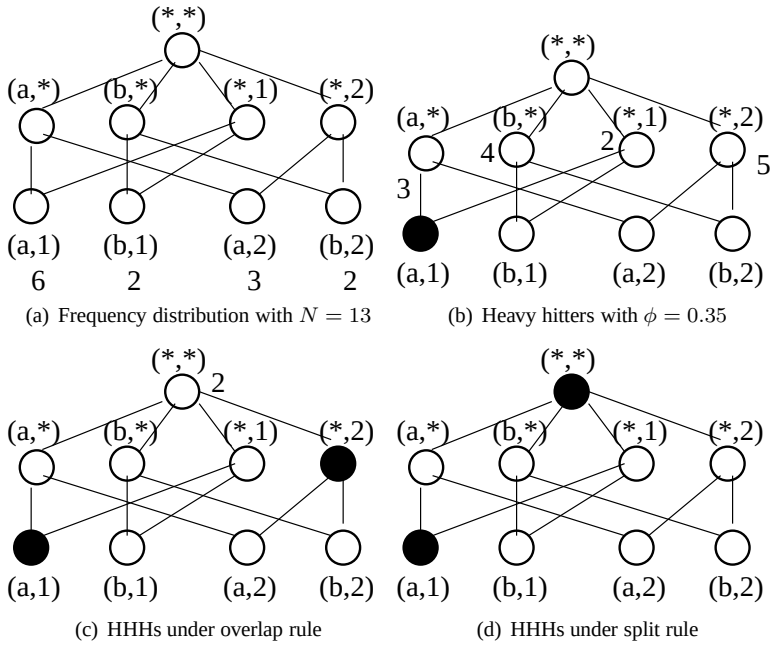
cant merely because it contains a child which is significant, or because the aggregation of its children makes it significant?

This leads us to our definition of HHHs given a fraction ϕ: find nodes in the hierarchy such that their HHH count exceeds ϕN, where the HHH count is the sum of all descendant nodes which have no HHH ancestors. This is best seen through an example, as shown in Figure 1 (c). Observe that the node with total count 25 is not an HHH, since its HHH count is only 5 (less than the threshold of 10): the child node with count 20 is an HHH, and so does not contribute. But the node with total count 60 is an HHH, since its HHH count is 15. Thus we see that the set of HHHs forms 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. The formal definition of this problem is given in Section 2.3.

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 and then discard extraneous nodes in a post-processing step. We argue that this approach can be considerably improved in practice (in terms of the space used and the answer quality) by incorporating knowledge of the hierarchy into algorithms for computing heavy hitters. We present algorithms that maintain sample-based summary structures, and provide deterministic error guarantees for finding HHHs in data streams.

1.2 Multi-dimensional Hierarchical Heavy Hitters

In practice, data warehousing applications and IP traffic data streams have several hierarchical dimensions. In the IP traffic data, for example, Source and Destination IP addresses and port numbers together with the time attribute yield 5 dimensions, although typically the Source and Destination IP addresses are the two most popular hierarchical attributes. So, in practice, one needs summarization methods that work for multiple hierarchical di-



(a) Frequency distribution with $N = 13$ (b) Heavy hitters with $\phi = 0.35$

(c) HHHs under overlap rule (d) HHHs under split rule

Fig. 2. Illustration of HHH in two dimensions

mensions. This calls for generalizing HHHs to multiple dimensions. As is typical in many database problems, generalizing from one dimension to two or more dimensions presents many challenges.

Multidimensional HHHs are a powerful construct for summarizing hierarchical data. To be effective in practice, the HHHs have to be *truly* multidimensional. Heuristics like materializing HHHs along one of the dimensions will not be suitable in applications. For example, as described by Estan et al. [2003], aggregating traffic by IP address might identify a set of popular domains and aggregating traffic by port might identify popular application types, but to identify popular combinations of domains and the kinds of applications they run requires aggregating by the two fields *simultaneously*.

A major challenge is conceptual: there are sophisticated ways for the product of hierarchies on two (or more) dimensions to interact and how precisely to define the HHHs in this context is not obvious. In the previous example, note that traffic generated by a particular application running on a particular server will be counted towards both the total traffic generated by that port as well as the total traffic generated by that server. Hence, there is implicit overlap. Alternatively, one may wish to count the traffic along one but not both of these generalizations (e.g., traffic on low ports is generalized to total port traffic whereas traffic on high ports is generalized to total server traffic). In this case, the traffic is split among its ancestors such that the resulting aggregates are on disjoint sets. This so-called "split case" was studied by Cormode et al. [2004]; here we focus on the "overlap" case.

As with summarization of data with a single hierarchical attribute, flat methods are inadequate because they do not capture heavy hitters at higher details, say traffic from a 24-bit subnet to another 24 bit subnet. One could try to run these flat methods at every possible

combination in the hierarchies, but this rapidly becomes too expensive. For example, determining a heavy hitter at every combination of detail of each hierarchy would be ineffective: any heavy hitter of 32-bit Source and Destination IP addresses means that all 32×32 of the i bit Source IP prefix and j bit Destination IP prefix for each i and j are heavy hitters. As in one dimensional HHHs, we need to discount the "descendant" heavy hitters while determining HHHs at any given level of detail. However, unlike the one dimensional case, it is not even clear how to discard nodes that do not qualify as HHHs in a post-processing step.

We show a simple example in Figure 2. Consider a two-dimensional domain, where the first attribute can take on values a and b, and the second 1 and 2. Figure 2(a) shows a distribution where $(a, 1)$ has count 6, $(a, 2)$ has count 3, $(b, 1)$ has count 2, and $(b, 2)$ has count 2. Moving up the hierarchy on one or other of the dimensions yields internal nodes: $(a, *)$ covers both $(a, 1)$ and $(a, 2)$ and has count 9; $(*, 2)$ covers both $(a, 2)$ and $(b, 2)$, and has count 5. Setting $\phi = 0.35$ means that a count of 5 or higher suffices, thus there is only one Heavy Hitter over the leaves of the domain. In the one-dimensional case, we can think of the count of a non-HHH node being propagated up to its ancestors. If we allow a node to count the contributions of all its non-HHH descendants, then we get the overlap case (since one input item may contribute to multiple ancestors becoming HHHs). Figure 2(c) shows the result on our example: the node $(*, 2)$ becomes an HHH, since it covers a count of 5. $(*, 1)$ is not an HHH, because the count of its non-HHH descendants is only 2. Note that the root node is not a HHH since, after subtracting off the contributions from $(a, 1)$ and $(*, 2)$, its remaining count is only 2. The contrasting split case is more procedural: we 'split' the count of non-HHH node evenly between its ancestors, so there is no double-counting. Thus the split count of $(*, 2)$ in Figure 2(d) is only 2.5, and the count of $(*, 1)$ is 1. Under this definition, the only non-leaf HHH is $(*, *)$. Since this case turns out to be somewhat more straightforward [Cormode et al. 2004], we focus exclusively on the overlap definition from now on.

1.3 Contributions

We address the challenge of defining and computing Hierarchical Heavy Hitters (HHHs), and our contributions are as follows:

(1) We introduce HHHs over one and multiple dimensions and give formal definitions of them. For online scenarios, we define an approximate notion of HHHs as well as accuracy and coverage guarantees required for correctness.

(2) We present online algorithms that find approximate HHHs in one pass, with accuracy guarantees, and provide proofs of their correctness. The algorithms use a small amount of space and can be updated to keep pace with high-speed data streams. The algorithms keep upper- and lower-bounds on the counts of items. Here, the items exist at various nodes in the hierarchy, and we must keep additional information to avoid over- and under-counting in the presence of parent(s) and descendants.
In multiple dimensions, the lattice property of the product of hierarchical dimensions is crucially exploited in our online algorithms to track approximate HHHs using only a small, fixed number of statistics per candidate node, regardless of the number of dimensions. We present two general online strategies for calculating HHHs over one and multiple hierarchical dimensions: one that maintains the full hierarchy down to a fringe ("Full Ancestry"), and one that allows intermediate node deletions ("Partial

Ancestry"). We present a complete analysis of the space and time requirements of our algorithms.
In comparison with our prior work [Cormode et al. 2003; 2004], here we provide additional algorithms, give full proofs of important properties of these algorithms, and carefully analyze their space and time requirements.

(3) We do extensive experiments with data from real IP applications and show that our proposed online algorithms yield outputs that are very similar (virtually identical, in many cases) to their offline counterparts. Our experiments demonstrate that (a) the proposed "hierarchy-aware" online algorithms yield high quality outputs with respect to the exact answer (almost identical) and significantly better than Heavy Hitters based approaches that do not account for descendant Heavy Hitters, based on a variety of precision-recall measures; (b) they have competitive performance and save an order of magnitude with respect to both space usage and output size, compared to finding Heavy Hitters on all prefixes; (c) our proposed Partial Ancestry strategy is better when space usage is of importance whereas our proposed Full Ancestry strategy is better when update time and output size is more crucial; and (d) the performance of the proposed algorithms in a data stream system is implementation-sensitive, and must be lightweight (e.g., based on hashing rather than a pointer-based data structure) and non-blocking to keep up with fast streaming rates, which we describe herein how to do.
Our prior work [Cormode et al. 2003; 2004] did not evaluate the accuracy of proposed online algorithm outputs with respect to the exact answers using precision-recall analysis, and did not evaluate the performance of these algorithms in a real data stream management system.

1.4 Outline

Section 2 formally defines hierarchical heavy hitters, for 1-d as well as 2-d, and their approximate online variants. Section 3 provides streaming algorithms to solve the approximation problems defined in Section 2. Section 4 experimentally evalutes these algorithms. Section 5 describes how the algorithms can be extended for distributed processing and handling deletions.

2. PROBLEM DEFINITIONS AND BOUNDS

2.1 Notation

Formally, we model the data as N d-dimensional tuples. Each attribute in the tuple is drawn from a hierarchy, and the attribute dimensions are numbered 1 to d. Let the (maximum) height of the hierarchy, or depth, of the ith dimension be h_i. For concreteness, we give examples consisting of pairs of 32-bit IP addresses, with the hierarchy induced by considering each octet (i.e., 8 bits) to define a level of the hierarchy. For our illustrative examples then, $d = 2$ and $h_1 = h_2 = 4$; our methods and algorithms apply to any arbitrary hierarchy. The *generalization* of an element on some attribute means that the element is rolled-up one level in the hierarchy of that attribute: the generalization of the IP address pair (1.2.3.4, 5.6.7.8) on the second attribute is (1.2.3.4, 5.6.7.*). We denote by $par(e, i)$ the parent of element e formed by generalizing on the ith dimension: $par((1.2.3.4, 5.6.7.*), 2) = (1.2.3.4, 5.6.*)$. In one-dimension, we may abbreviate this to $par(e)$. An element is *fully general* on some attribute if it cannot be generalized further, and this is denoted "*": the pair (*, 5.6.7.*)

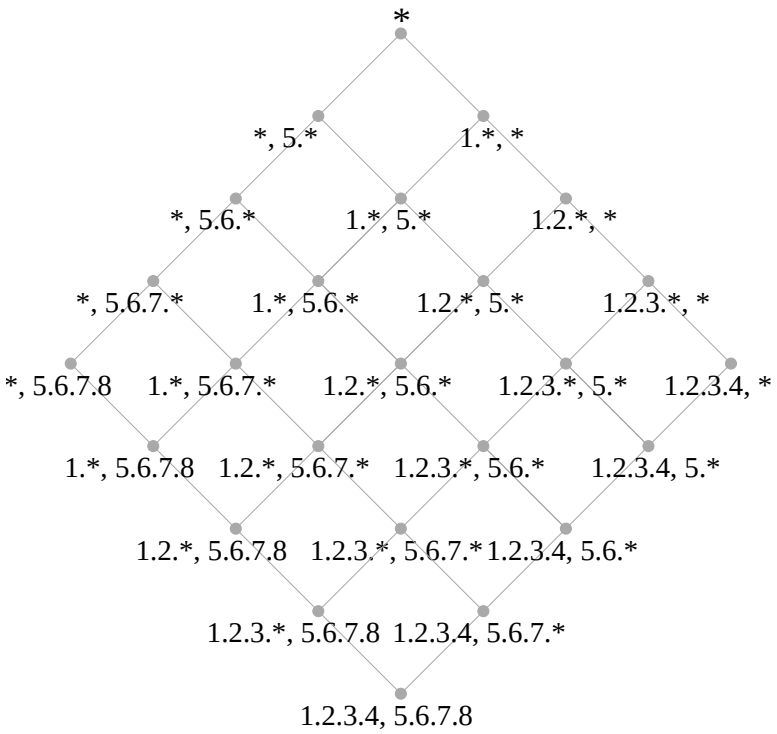


Fig. 3. The lattice induced by the element (1.2.3.4, 5.6.7.8)

is fully general on the first attribute but not the second. Conversely, an element is *fully specified* on some attribute if it is not the generalization of any element on that attribute. We denote the generalization relation by $\prec$: if p is generalizable to q, then we write this as $p \prec q$, with $p \preceq q$ defined as $(p \prec q) \vee (p = q)$. The generalization relation over a defined set of hierarchies generates a *lattice* structure that is the product of the 1-d hierarchies. Elements form the lattice nodes, and edges in the lattice link elements and their parents. The node in the lattice corresponding to the generalization of elements on all attributes we denote as "*", or ALL, and has count N. We will overload this notation to define the sublattice of a *set* of elements P as $(e \preceq P) \iff (\exists p \in P.e \preceq p)$. The total number of nodes in the lattice, H is computed as $H = \prod_{i=1}^{d}(h_i + 1)$.

An example is shown in Figure 3, where we show how the leaf element $(1.2.3.4, 5.6.7.8)$ appears at each point in the lattice. Modeling the structure of products of generalizations of items as a lattice is standard on work on computing data cubes [Agarwal et al. 1996] and iceberg cubes [Ng et al. 2001]. It is worth noting that structures induced by elements can partially overlap with each other. For example, $(1.2.*, 5.6.7.*)$, and all its generalizations, are also common to the structure induced by the element $(1.2.2.1, 5.6.7.7)$.

In order to facilitate referring to specific points in the lattice, we may label each element in the lattice with a vector of length d whose ith entry is a non-negative integer that is at most h_i, indicating the level of generalization of the element. The pair (1.2.3.4, 5.6.7.8) is at generalization level [4,4] in the lattice of IP address pairs, whereas (*, 5.6.7.*) is at

[0,3]. The *parents* of an element at $[a_1, a_2, \ldots, a_d]$ are the elements where one attribute has been generalized in one dimension; hence, the parents of elements at [4,4] are at [3,4] and [4,3]; items at [0,3] have only one parent, namely at [0,2], since the first attribute is fully general. Two elements x and y are *comparable* under the $\preceq$ relation if the label of y is less than or equal to that of x on every attribute: items at [3,4] are comparable to ones at [3,2], but [3,4] and [4,3] have no comparable elements. We define $Level(i)$, the ith level in the lattice as the set of labels where the sum of all values in the vector is i: hence $Level(8) = \{[4,4]\}$, whereas $Level(5) = \{[1,4],[2,3],[3,2],[4,1]\}$ and $Level(0) = \{[0,0]\}$. We may overload terminology and refer to an element being a member of the set $Level(l)$, meaning that the item has a label which is a member of that set. No pair of elements with distinct labels in $Level(i)$ are comparable: formally, they form an anti-chain in the lattice.[1] Equivalently, if x and y are at the same level, then $x \not\preceq y$ and $y \not\preceq x$. The levels in the lattice range from 0 to $L = \sum_i h_i$, and hence the total number of levels in the lattice is $L + 1$. We define the function `GeneralizeTo` which takes an item and a label, and returns the item generalized to that particular label. For example, `GeneralizeTo`((1.2.3.4, 5.6.7.8),[0,3]) returns (*, 5.6.7.*).

2.2 Heavy Hitters

We first review the definition of heavy hitters, before formally defining hierarchical heavy hitters later in this section.

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 ϕN. Let f_e denote the frequency of each element e in S. Then $HH = \{e \mid f_e \geq \phi N\}$.* □

The *heavy hitters problem* is that of finding all heavy hitters, and their associated frequencies, in a data set. In any data set, there can be no more than $1/\phi$ heavy hitters, by the definition of 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 [Cormode and Muthukrishnan 2003] for a brief survey), based on the maintenance of summary structures that allow element frequencies to be estimated.

2.3 Hierarchical Heavy Hitters over One Dimension

The preceding description of a lattice also applies when the data is drawn from a single hierarchical attribute, but the structure is simplified significantly. In particular, the lattice is simply a tree, because each (non-fully general) item has exactly one parent. We can define the Hierarchical Heavy Hitters over such a domain in an inductive fashion.

[1] An *anti-chain* is a set of elements from the lattice such that no two elements in the set are comparable.

DEFINITION 2 HIERARCHICAL HEAVY HITTER. *Given a (multi)set S of elements from a hierarchical domain D of depth h, and a threshold ϕ, we define the set of* Hierarchical Heavy Hitters *of S inductively.*

—*$\mathcal{HHH}_h$, the hierarchical heavy hitters at level h (the leaf level of the hierarchy), are simply the heavy hitters of S.*

—*Given a prefix p from $Level(l), 0 \leq l < h$ in the hierarchy, define $F_p = \sum f(e) : (e \in S) \wedge (e \preceq p) \wedge (e \not\preceq \mathcal{HHH}_{l+1})$. The set $\mathcal{HHH}_l$ is defined as the set*

$$\mathcal{HHH}_{l+1} \cup \{p : (p \in Level(l)) \wedge (F_p \geq \phi N)\}$$

—*The set of Hierarchical Heavy Hitters, $\mathcal{HHH}$, is the set $\mathcal{HHH}_0$.* □

Note that, because we can attribute each item from the input to at most one of the Hierarchical Heavy Hitters, and each HHH requires at least ϕN items from the input, then there can be at most $1/\phi$ HHHs in this setting.

The *hierarchical heavy hitters problem* we study 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 without using space linear in the input size. Hence, we will study the following (approximate) problem:

DEFINITION 3 HHH PROBLEM. *Given a data stream S of N 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 to output a set of prefixes $P \subseteq D$, and approximate bounds on the frequency of each $p \in P$, f_{min} and f_{max}: such that the following conditions are satisfied:*

(*1*) accuracy: *$f_{min}(p) \leq f^*(p) \leq f_{max}(p)$, where $f^*(p)$ is the true frequency of p in S, i.e., $f^*(p) = \sum_{e \preceq p} f(e)$; and $f_{max}(p) - f_{min}(p) \leq \epsilon N$.*

(*2*) coverage: *For all prefixes $q \notin P$, $\phi N > \sum f(e) : (e \preceq q) \wedge (e \not\preceq P)$.* □

LEMMA 1. *In one dimension, the size of the smallest set of Hierarchical Heavy Hitters that satisfies the Coverage constraint is equal to the size of the exact HHHs, $|\mathcal{HHH}|$.*

PROOF. First, observe that $\mathcal{HHH}$ satisfies the coverage constraint, by following the definition of F_p. Now let X be a set satisfying coverage that is smaller than $\mathcal{HHH}$, the set of HHHs computed by the exact algorithm. If X and $\mathcal{HHH}$ differ, then let p be a prefix that is in the symmetric difference of the two sets, and occurs at the deepest level, l, of those items (if there are many such items, one can be chosen arbitrarily). There are two cases to consider. (1) $p \in \mathcal{HHH} \backslash X$. This cannot be the case, since it means that X violates the coverage condition. $\mathcal{HHH}$ and X agree on all levels greater than l and the exact algorithm was "forced" to pick p, since $f(p) \geq \phi N$, and so X must include p as well or else it will violate coverage. (2) $p \in X \backslash \mathcal{HHH}$. Then, because X and $\mathcal{HHH}$ agree on all items at levels greater than l, we can remove p from X and replace it with $par(p)$ without violating the coverage condition (because $\mathcal{HHH}$ does not violate coverage). This does not increase the size of X, and may in fact reduce its size if $par(p)$ is already in X. Applying these two arguments repeatedly, we show that by repeatedly "pushing up" (that is, applying case (2)) items in X, we will end up with a set that is identical to $\mathcal{HHH}$, since eventually there will be no items p that are in the symmetric difference of the two sets, and they will be identical. As every step did not increase the size of the set X, we must conclude that $|\mathcal{HHH}| \leq |X|$, contradicting the initial assumption that X was smaller. □

This means that we will evaluate the quality of our solutions, which will guarantee to meet the accuracy and coverage constraints, by the size of their output. We will use the size of the set $\mathcal{HHH}$ (computed offline) to compare against.

2.4 Hierarchical Heavy Hitters in Multiple Dimensions

The general problem of finding *Multi-Dimensional Hierarchical Heavy Hitters* (HHHs) is to find all items in the lattice whose count exceeds a given fraction, ϕ, of the total count of all items, after discounting the appropriate descendants that are themselves HHHs. This still needs further refinement, since in this setting it is not immediately clear how to compute the count of items at various nodes in the lattice. In the previous section, with just a single hierarchy, the semantics of what to do with the count of a single element when it was rolled up was clear: simply add the count of the rolled up element to that of its (unique) parent. In this more general multi-dimensional case, each item has multiple parents — up to d of them. So this problem will vary significantly depending on how the count of an element is allocated to its parents. There are two fundamental variations to consider, which differ in how we allocate the count of a lattice node that is not a hierarchical heavy hitter when it is rolled up into its parents. Informally, the "overlap rule" allocates the full count of an item to each of its parents and, therefore, counted multiple times, in nodes that overlap. The overlap rule appears implicit in prior work on network data analysis [Estan et al. 2003], to show patterns of traffic over a multidimensional hierarchy of source and destination ports and addresses in what the authors call "compressed traffic clusters". Meanwhile, the "split rule" means that the count of an item is divided between its parents in some way. The split rule is considered by Cormode et al. [2004], and we do not discuss it further here, since it is less involved, and appears to have fewer applications.

For simplicity and brevity, we will describe the case where all the input data consists of elements which are fully specified on every attribute, i.e., leaf elements in the lattice. Our methods naturally and obviously extend to the case where the input can arrive as a mix of partially and fully specified items, although we do not discuss this case in detail.

By analogy with the semantics for computing iceberg cubes, the overlap case says that the count for an item should be given to each of its parents when the item is rolled up [Beyer and Ramakrishnan 1999]. The HHHs in the overlap case are those elements whose count is at least ϕN where N is the total count of all items, and $0 < \phi \leq 1$. When an item is identified as an HHH, its count is not passed up to either of its parents. This is one meaningful extension of the 1-d case, where the count of an item being rolled up is allocated to its *only* parent, unless the item is an HHH.

This seems intuitive, but there are many subtleties of this approach that will need to be handled in any algorithm to compute the HHHs under this rule. Suppose we kept only lists of elements at each level of generalization in the hierarchy, and updated these as we roll up items. Then the item $e = (1.2.3.4, 5.6.7.8)$ with a count of one (we will write f_e to denote the count of e, so here $f_e = 1$), would be rolled up to (1.2.3.*, 5.6.7.8) and (1.2.3.4, 5.6.7.*), each with a count of one. Rolling up each of these to the common grandparent of (1.2.3.4, 5.6.7.8) would give (1.2.3.*, 5.6.7.*) with a count of two. This is a problem, since this results from a single descendent with a count of one; we should like each item to contribute at most once to the count. So additional information is needed to avoid over-counting errors like this, and similar problems, which can grow worse as the number of attributes increases. To formally define the problem, we introduce the notion of the overlap count of an item, and will then show how to compute this exactly.

DEFINITION 4. **Hierarchical Heavy Hitters with Overlap Rule** *Let the input S consist of a set of elements e and their respective counts $f(e)$. Let $L = \sum_i h_i$. The Hierarchical Heavy Hitters are defined inductively based on a threshold $0 < \phi < 1$.*

—$\mathcal{HHH}_L$ contains all heavy hitters at level L: $e \in S$ such that $f_e \geq \phi N$.

—The overlap sublattice count of an element p at $Level(l)$ in the lattice where $l < L$ is given by $f_l(p) = \sum f(e) : (e \in S) \wedge (e \preceq p) \wedge (e \not\preceq \mathcal{HHH}_{l+1})$. The set $\mathcal{HHH}_l$ is defined as the set

$$\mathcal{HHH}_{l+1} \cup \{p : (p \in Level(l)) \wedge (f_l(p) \geq \phi N)\}$$

—The Hierarchical Heavy Hitters with the overlap rule for the set S is the set $\mathcal{HHH} = \mathcal{HHH}_0$. □

LEMMA 2. *Consider the lattice induced by an element (as in Figure 3, and let A denote the length of the longest anti-chain in this lattice. (i) In one dimension, $A = 1$; in two dimensions, $A = 1 + \min(h_1, h_2)$. In higher dimensions, we have $A \leq (\prod_{i=1}^{d}(1 + h_i))/\max_i(1 + h_i)$. (ii) The size of the set $\mathcal{HHH}$ under the overlap rule is at most A/ϕ.*

PROOF. (i) In a one-dimensional hierarchy and the induced lattice, clearly for any two elements, one must be the ancestor of (or equal to) the other, hence the anti-chain has size at most $A = 1$. For two dimensions, we have a product of hierarchies. From an element (x, y), we can find all its generalizations at $Level(\min(h_1, h_2)$, which contains $1 + \min(h_1, h_2)$ items, none of which are comparable. For example, in Figure 3, $Level(4)$ contains $(*, 5.6.7.8), (1.*, 5.6.7.*), (1.2.*, 5.6.*), (1.2.3.*, 5.*), (1.2.3.4, *)$. To see that this is the maximum possible, suppose w.l.o.g. that $h_1 < h_2$ and that we had more than $1 + h_1$ items: then at least two of them must be have the same value on the first attribute, and are therefore comparable. The same logic shows the upper bound on A for higher dimensions: two items are comparable if they share values in all but one of the dimensions, and so the tightest bound comes from letting this last dimension be the one with greatest depth.

(ii) The total number of HHHs is bounded in terms of the depth of the hierarchies. Each item in the input can be counted towards multiple members of $\mathcal{HHH}$, but these HHHs must be incomparable, else the item could not be counted towards all of them. Then these HHHs must form an anti-chain in the lattice, and so we bound this count by the size of the largest anti-chain. Hence, the sum of counts of HHHs can be at most AN. Since each HHH has count at least ϕN, we conclude the number of HHHs under the overlap rule can be at most A/ϕ. □

This gives evidence of the "informativeness" of the set of HHHs, and their conciseness. By contrast, if we propagated the counts of each item to every ancestor and found the Heavy Hitters at every level, then there could be as many as H/ϕ HHHs, where $H = \prod_{i=1}^{d}(h_i + 1)$. Even in low dimensions, H can be many times larger than A.

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, as formalized in the following definition.

DEFINITION 5. **Online HHH Problem: Overlap Case** *The Multi-Dimensional Hierarchical Heavy Hitters problem with the overlap rule on input S with threshold ϕ is to*

output a set of items P from the lattice, and their approximate counts f_p, such that they satisfy two properties:

(*1*) accuracy: $f_{min}(p) \leq f^*(p) \leq f_{max}(p)$, *where $f^*(p)$ is the true sublattice count of p in S, i.e., $f^*(p) = \sum_{e \preceq p} f(e)$; and $f_{max}(p) - f_{min}(p) \leq \epsilon N$.*

(*2*) coverage: *For all prefixes $q \notin P$, $\sum f(e) : (e \preceq q) \wedge (e \not\preceq P) < \phi N$.* □

This definition is identical to the definition of HHHs in one-dimension, extended to a multidimensional setting. Note that for accuracy, we ask for an accurate sublattice count for each output item, rather than the count discounted by removing the HHHs. This is a useful quantity that we can estimate with high accuracy. By appropriate rescaling of ϵ, one could find the discounted count accurately, however this comes at a high price for the required space, multiplying by a factor proportional to the largest possible number of HHH descendants. It was shown by Hershberger et al. [2005] that such a factor is essentially unavoidable, hence our focus on only providing accurate sublattice counts.

The "goodness" of an approximate solution is measured by how close it is in size to that of the exact solution. In the 1-d setting we proved in Lemma 1 the exact solution is the smallest satisfying correctness and, hence, a smaller approximate answer size is preferred. In the multi-dimensional problem, one can contrive examples where the approximate output can be smaller than the exact one.

EXAMPLE 1. *Suppose (1.2.*,5.6.*) has count 3*
(1.3.,5.6.*) has count 3*
(1.4.,5.6.*) has count 9*
(1.4.,5.7.*) has count 3*
(1.4.,5.8.*) has count 3*
and set the threshold ϕN to be 10, and error ϵN to be 2.

Suppose an approximate algorithm includes (1.4.,5.6.*) in the output. Under the overlap semantics, the counts for (1.4.*,5.*) and (1.*,5.6.*) are 6, so the approximate algorithm does not have to include these in the output. However, the exact definition would not output (1.4.*,5.6.*) and so would lead to counts of 15 for (1.4.*,5.*) and (1.*,5.6.*). Thus both of these items are HHHs under the exact definition. By repeating this structure several times, replacing $\{2, 3, 4, 6, 7, 8\}$ with distinct values, the exact algorithm can be forced to output many more items than an approximate algorithm.*

In the worst case, the output may be A times bigger than the smallest possible, where A is the size of the longest anti-chain in the lattice, as defined before. Nevertheless, such contrived examples seem rare in practice, and on real data we have observed that the output size of the exact algorithm always lower bounds the size of the approximate output. Exact algorithms to compute HHHs in multiple dimensions were given by Cormode et al. [2004]; we do not repeat them here, since they follow almost directly from the definition.

3. ONLINE ALGORITHMS

We develop *hierarchy-aware* solutions for the one- and multi-dimensional HHH problems, where new data stream elements only arrive and there are no deletions of previously seen items. For this data stream model, we propose deterministic algorithms that maintain sample-based summary structures, with deterministic worst-case error guarantees for finding HHHs. Here the user supplies error parameter ϵ in advance and can supply any threshold ϕ at query time to output ϵ-approximate HHHs above this threshold.

```
Insert(element e, count c):
/* par(e) is the parent of e */
01 forall (p : e ≺ p) do
02    if t_p exists in T then
02       f_p += c;
03    else
04       create t_p;
05       f_p = c;
06       Δ_p = b_current − 1;

Compress():
01 for each t_e ∈ T do
02    if (f_e + Δ_e ≤ b_current)) then
03       delete t_e;

Output(threshold φ):
01 for each t_e in postorder do
02    if (f_e + Δ_e > φN) then
03       print(e, f_e, f_e + Δ_e);
```

Fig. 4. Algorithm for Naive Strategy in arbitrary number of dimensions

3.1 Naive Algorithm

We first discuss a naive algorithm based on existing work that we will use as a baseline to compare our various results. At a high level, this algorithm keeps information for every label in the lattice, that is, it keeps H independent data structures. Each one of these returns the (approximate) Heavy Hitters for that point in the lattice. This will be a superset of the Hierarchical Heavy Hitters, and it will satisfy the accuracy and coverage requirements for any of our definitions of HHHs (one dimensional, or multi-dimensional overlap); however it will be very costly in terms of space usage. It also becomes very slow to process updates as the dimensionality and depths of the hierarchies increase. We evaluate the output on the metrics of the space used by the data structures, and the size of the output (i.e., number of items output). We expect this naive algorithm to do badly by these measures. Hence, we propose algorithms which keep one data structure to summarize the whole lattice, and show that they are empirically better in terms of space and output size.

In detail, the naive method works as follows: for every update e, we compute all generalizations of this item and insert each one separately into a different data structure for computing approximate counts of items. We ensure that there is one data structure for each different label. The `LossyCounting` algorithm due to Manku and Motwani [2002] can be used as a "black box" independently, one copy to summarize all items with the same label in the lattice structure. `LossyCounting` keeps track of a set of items seen in the stream with lower and upper bounds on their counts. When an item is observed in the stream which is recorded in the data structure, its bounds are updated accordingly; else, it is inserted with a lower bound of 1 and an upper bound of ϵN. Periodically, a "compress" operation is performed on the data structure, which removes all items whose upper bound is less than ϵN. It can be shown that this algorithm guarantees accuracy of ϵN for all item counts and requires $O(\frac{1}{\epsilon} \log \epsilon N)$ space.

Since we use H independent instances of this algorithm, and place each update into each of these H instances, the naive algorithm has an $O(\frac{H}{\epsilon} \log \epsilon N)$ overall space bound. Note that we could replace this algorithm with any approximate counting algorithm which finds all items occurring more than a specified fraction ϕ of the time with accuracy ϵ, such as the Misra-Gries [1982] algorithm or that of Metwally et al. [2005]. We use Lossy Counting

here since it has good practical performance on the realistic data sets that we use, and because it is the basis of the more advanced algorithms that we develop here, meaning we can directly compare the space savings of our approach.

The desired HHHs can be extracted in post-processing as follows. The tuples are scanned in postorder across levels. At each level, we output all Heavy Hitters that exceed the ϕN threshold. It is a simple observation that this approach will satisfy the necessary accuracy and coverage constraints in one dimension, and in higher dimensions; however, since it makes no adjustment to reduce the count based on descendant HHHs, then the size of the output will likely be much larger than the smallest possible. This naive algorithm can be thought of as running "heavy hitters for every label". The algorithm is given in Figure 4.

The time required to process each update is $O(H)$ plus the periodic pruning of the data structure every $1/\epsilon$ updates, which requires a linear scan of the data structure. The amortized cost is therefore worst case $O(H \log \epsilon N)$. Since the space used by Lossy Counting is observed to be closer to $O(\frac{1}{\epsilon})$ [Manku and Motwani 2002], the amortized costs may be dominated more by the insertion cost, which is $O(H)$ per insertion.

3.2 One Dimensional Case

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. Associated with each value is a bounded amount of auxiliary information used for determining the lower- and upper-bounds on the frequencies for elements whose prefix is e ($f_{min}(e)$ and $f_{max}(e)$, respectively). The input stream is conceptually divided into buckets of $w = \lceil \frac{1}{\epsilon} \rceil$ consecutive insertions; we denote the current bucket number as $b_{current} = \lceil \frac{N}{w} \rceil$. There are two alternating phases of the algorithms: insertion and compression. For every update e received, the `Insert` routine is called with parameters e and count 1. After every w updates (i.e., on the bucket boundaries), the `Compress` routine is called to prune away unnecessary information from the data structure, and keep it to a bounded size. During compression, the space is reduced via merging auxiliary values into the parent node and then deleting these nodes. We will show worst case space bounds that do not depend on the sequence of updates processed. 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 by calling the `Output` routine. This framework is closely based on the `LossyCounting` algorithm [Manku and Motwani 2002], which keeps similar information and uses similar routines to find HHs. It forms the basis of our naive algorithm, as described above. Next, we describe two strategies using this framework and give the algorithms for `Insert`, `Compress` and `Output` for each.

3.2.1 *Full Ancestry Algorithm.* Our first algorithm is a "hierarchy aware" version of the naive algorithm. It extends the naive algorithm by tracking information across levels of the hierarchy, rather than treating each level independently. The data structure tracks information about a set of nodes that vary over time, but which always form a subtree of the full hierarchy. When a new node is inserted, information stored by its ancestors is used to give more accurate information about the possible frequency count of the node. This has the twin benefits of yielding more accurate answers and keeping fewer nodes in the data structure (since we can more quickly determine if a node cannot be frequent and so does not need to be stored). Thus we are able to prove that the algorithm maintains the required

```
Insert(element e, count c):
/* par(e) is the parent of e */
01 if t_e exists in T then
02    g_e += c;
03 else
04    create t_e;
05    g_e = c;
06    if (e ≠' *')
07       Insert(par(e), 0);
08       Δ_e = m_e = m_par(e);
09    else
10       Δ_e = m_e = b_current − 1;

Compress():
01 for each t_e ∈ T in postorder do
02    if ((t_e has no descendants)
      ∧(g_e + Δ_e ≤ b_current)) then
03       g_par(e) += g_e;
04       m_par(e) = max(m_par(e), g_e + Δ_e);
05       delete t_e;

Output(threshold φ):
01 let F_e = f_e = 0 for all e;
02 for each t_e in postorder do
03    if (g_e + Δ_e + F_e > φN) then
04       print(e, f_e + g_e, f_e + g_e + Δ_e);
05    else
06       F_par(e) += F_e + g_e;
07    f_par(e) += f_e + g_e;
```

Fig. 5. Algorithm for Full Ancestry Strategy in one dimension

accuracy guarantees in space no worse than that used by the naive algorithm.

More formally, consider the set of nodes whose (unadjusted for HHH descendants) count exceeds the fraction ϵN for the current value of N. This induces a proper subtree of the hierarchical domain. The leaves of this subtree consist of nodes whose count exceeds this threshold, but none of their children do. This set of leaves we refer to as "the fringe", and they form an anti-chain under the $\prec$ relation. The goal of our first strategy is to (approximately) maintain the fringe as items arrive. In order to guarantee approximation, we may keep information about some nodes which are not in the fringe, but we will prune our data structure to remove as many nodes as possible that are not in the fringe. We enforce the property that if we store information about any node in our algorithm, then all of its ancestors are also stored. Hence, we denote this approach as the "full ancestry" method.

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 descendants $\{e\}$. That is, g_p bounds the number of nodes with prefix p that are not counted in descendant nodes of p. 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". Δ_p represents an upper bound on our uncertainty in the count, which is set when we insert the node t_e. Naively, we could set $\Delta_p = b_{current}$, by analogy with `Lossy Counting` [Manku and Motwani 2002], but we keep extra information in ancestor nodes to give a tighter bound. Let $\{d(e)\}$ denote the deleted children of a node t_e. We observe that one can improve the bounds on the Δ_e's by keeping track of $m_e = \max_{d \in d(e)}(g_d + \Delta_d)$. This is easy to maintain: following the deletion of a child, update m_e of its parent if

necessary. Thus, the auxiliary information associated with each element e that is stored in T is (g_e, Δ_e, m_e), where g_e and Δ_e are defined above. We extend the definition of m to nodes that are not materialized in the data structure by setting $m_q = m_{par(q)}$ for nodes q not in T. By applying this definition recursively, a value of m_q can always be found.

3.2.1.1 **Computation of f_{min} and f_{max}..** For any prefix p, we compute

$$f_{min}(p) = \sum_{e \preceq p} g_e$$

$$f_{max}(p) = f_{min}(p) + \Delta_p$$

if p is stored in T, and if not, we set $f_{max}(p) = f_{min}(p) + m_p = m_p$.

Insertion operation. To process a new update of e, with an update weight of c, we test to see whether e is present in T. If so, then we just have to increment g_e by c. Else, if not, we recursively call `Insert` with $(par(e), 0)$ (this ensures that the parent of the node is inserted in the data structure), and create a node to represent e. We use the m_e from the parent node to set $\Delta_e = m_{par(e)}$. Each insertion operation requires us to examine up to H nodes in T in the worst case; however, in practice we expect this to be smaller since the process only needs to find the closest ancestor of e that is present in T.

Compress operation. During compression, we scan through the tuples in postorder and find nodes satisfying $(g_e + \Delta_e \leq \lfloor \epsilon N \rfloor)$. These correspond to nodes whose contribution is sufficiently small that they can be removed without loss of accuracy. For each such node, if it has no descendants, then it is deleted from the data structure (and $m_{par(e)}$ is updated). Consequently, T is a complete trie down to a "fringe". All q not stored in T must be below the fringe. Any pruned nodes t_q must have satisfied $(f_{max}(q) \leq \lfloor \epsilon N \rfloor)$ due to the algorithm. If there are $|T|$ tuples in T, then the cost to perform a `Compress` operation is $O(|T|)$. Below we show that $|T| = O(\frac{H}{\epsilon} \log \epsilon N)$.

Output operation. The `Output` function for this strategy takes ϕ as a parameter and chooses a subset of the prefixes in T satisfying correctness. That is, we compute an overestimate of the adjusted sublattice count for each node by proceeding level by level from the leaves (see Definition 4). We initialize F_e, our estimate of the sublattice count of non-HHH nodes, to zero for all nodes. We proceed up the hierarchy and update F_e as we go. If a node is not an HHH, then we update $F_{par(e)}$ of its parent by adding on F_e. However, if the node e is an HHH, then we do not propagate the F_e count upward. We test whether e is an HHH by comparing $F_e + g_e + \Delta_e$ to ϕN: this compares an upper bound on the count of e to the threshold for being an HHH.

Figure 5 gives the algorithm. Below, we show that this correctly maintains the necessary constraints.

THEOREM 1. *The routines in Figure 5 guarantee the accuracy and coverage properties from Definition 3.*

PROOF. Accuracy requires that the estimated count of a node is within an ϵN additive factor of the true count of the node. Our output routine computes f_p as $\sum_{e \prec p} g_e$ and outputs $f_p + g_p$ as the approximate sublattice count for e (in line 04 of the Output routine). This is exactly equal to our earlier definition of $f_{min}(p)$. Observe that $f_{min}(p)$ counts only insertions to nodes in the subtree defined by p, and is therefore no more than $f^*(p)$. We argue that $f_{max}(p)$ is an upper bound on $f^*(p)$ by induction over the sequence of insertion

and compression operations. Clearly $f_{max}(p) = 0$ at the start of the stream is a valid upper bound. When we compress, we delete nodes that satisfy $f_{max}(p) \leq \epsilon N$ (line 02 of Compress), and we update the m value of p's parent to be max of $m_{par(p)}, f_{max}(p)$ (line 04). This ensures that $m_{par(p)} \geq f_{max}(p)$ for deleted p values at all times (this is true even if $par(p)$ gets deleted: we derive a value of m_p for nodes that are not materialized in the data structure because we defined $m_p = m_{par(p)}$). When we (re)insert a node we set Δ based on $m_{par(p)}$ (line 08 of Insert). Since $m_{par(p)} \geq f_{max}(p)$ when p was deleted, and no further insert operations have occurred to nodes in p's subtree (else p would have been reinserted earlier), while $m_{par(p)}$ can only have increased, then $f_{max}(p)$ continues to be an upper bound on f_p^*, the true count of p.

For the bounds on the uncertainty in our estimate of f, we show that for any node p, $f_{max}(p) - f_{min}(p) \leq \epsilon N$ as follows. If p is present in the data structure, then it has a value of Δ_p representing an upper bound on $f_{max}(p) - f_{min}(p)$ that was instantiated when p was inserted. Δ_p is instantiated based on m_p, which is the maximum over a subset of deleted nodes of their f_{max}. The value of m_p is never more than $b_{current}$, since this is the requirement for a node to be deleted. Hence, because Δ_p in a tuple in T is never changed, we conclude $f_{max}(p) - f_{min}(p) \leq b_{current} \leq \epsilon N$. Similarly, for a node p not present in the lattice, it has $f_{max}(p) - f_{min}(p) = m_p$, where again m_p is bounded by $b_{current}$ by the condition for compressing.

For coverage, we need to show that the output function is conservative, that is, based on the information available in the summary, it outputs any node when it is possible that it is above the threshold. We decide whether to output based on our computation of F_p (line 03 in Output): this is computed similarly to f_p, but does not include any contribution from nodes that are included in the output set of nodes, P. We see that for any prefix q,

$$\begin{aligned} F_q + g_q + \Delta_q &= f_{min}(q) + \Delta_q - \sum_{(e \preceq q) \wedge (e \preceq P)} g_e \\ &\geq (\sum f(e) : e \preceq q - \sum f(e) : (e \preceq q) \wedge (e \preceq P)) \\ &= \sum f(e) : (e \preceq q) \wedge (e \not\preceq P) \end{aligned}$$

That is, our computed value is always an overestimate of the condition from Definition 3, and so the algorithm guarantees coverage. □

THEOREM 2. *For a given ϵ, the Full Ancestry strategy finds HHHs in $O(\frac{H}{\epsilon} \log(\epsilon N))$ space.*

PROOF. Our proof proceeds in several steps. First, we show that the space used by our strategy is no more than that used by the same algorithm running on a modified version of the input stream. Then we argue that the space used to find HHHs on this modified stream is no more than the space used by the Lossy Counting algorithm of Manku and Motwani over this same stream [Manku and Motwani 2002]. We can then apply the space bounds of that algorithm.

Consider the space used by our algorithm after N updates have been seen. Then some nodes are materialized in our summary. Let the set of nodes in our summary that have no descendants that are also materialized define the fringe nodes (at time N): $FR = \{p \in T | \forall q \in T.q \preceq p \Rightarrow q = p\}$. Observe that every element from the input S is either a fringe node itself, or it has exactly one fringe node as an ancestor. Given a leaf e and

a set of fringe nodes FR, we rewrite the original stream of updates as a new stream, by replacing every node e in the update stream by the $p \in FR$ such that $e \preceq p$. We argue that if we run our algorithm on this modified stream, then we will generate a virtually identical data structure at time N as when we run our algorithm on the original stream. In order to do this, we will show that two invariants are preserved by the algorithm. We denote by S_{full} the algorithm running on the original stream, and S'_{full} the algorithm running on the modified stream.

Property 1. For any node e stored by S'_{full} with g, m and Δ, the node representing e in S_{full} has the same values of g, m and Δ.

Property 2. For any node e stored by our algorithms, all ancestors p of e satisfy $f_{max}(e) \leq f_{max}(p)$.

LEMMA 3. *If both these properties are satisfied, then after processing the same input, every node stored by S_{full} is also stored by S'_{full}, and further, that for every node e stored by S'_{full}, either e is stored by S_{full} or, if e is below the fringe, then p is stored by S_{full}, where $e \prec p$ and $p \in FR$.*

PROOF. This we prove by contradiction: suppose first that e is stored by S_{full} but not S'_{full}. For e to have been deleted in S'_{full}, it must be the case that at some point $f_{max}(e)$ was less than $b_{current}$. But at the same time e should have been deleted in S_{full}, since by Property 1, it has the same value of $f_{max}(e) = g + \Delta$. Note that all its descendants would also have been deleted, since by Property 2, all their f_{max} values were no bigger than that of e. Hence, we argue that this case cannot happen. Now, suppose that e is stored by S'_{full} but not by S_{full}. Then a similar argument based on Property 1 shows that e must be deleted by both algorithms at the same point. One difference to note is that S_{full} may store some e that is "below" the fringe of nodes FR. In this case, we argue that if S_{full} stores e then S'_{full} must store the fringe node that contains e, i.e., the $p \in FR$ such that $e \preceq p$. □

This means that the sets of nodes stored by both versions of the algorithm are not completely identical, since several nodes may be stored by S_{full} corresponding to only one in S'_{full}. However, at time N, then since there are no nodes stored by S_{full} below the fringe (since the state at this time defines the fringe), and so the set of prefixes stored in S_{full} and S'_{full} are identical after seeing the whole input.

LEMMA 4. *Our algorithm always maintains Properties 1 and 2.*

PROOF. We now show that Properties 1 and 2 always hold, by induction over the sequence of operations (`Insert` and `Compress`). The base case is to observe that initially the data structures are empty, and so trivially both properties hold. For an insert case, there are two cases to consider, depending on whether e is currently stored by S'_{full} or not.

Case 1. *Insertion of e which is already stored by S'_{full}.* By Lemma 3, then e is also stored by S_{full}, and by the inductive hypothesis, e has the same value of f_{max} in both versions. We update the g value for node e and do not alter Δ or m, so $f_{max}(e)$ increases by the same amount in both versions, preserving Property 1. Similarly, for all descendants of e, their f_{max} all increase by the same amount, so Property 2 is preserved.

Case 2. *Insertion of e which is not stored by S'_{full}.* By the above argument, then e is not stored by S_{full} either, and consequently all descendants of e are not stored by either version. We insert e with $g = c$ and $\Delta = m_{par(e)}$ in both versions of the algorithm, which

ensures Property 1. The f_{max} of all ancestors increases by c, while $f_{max}(e)$ takes the value of $c+m_{par(e)}$. We observe for any node p, then $m_p < g_p+\Delta_p$ by the way the m values are created: m_p represents the maximum $g+\Delta$ of a deleted descendant of p. If $m_p \geq g_p+\Delta_p$ when m_p is set, then p would also be deleted at the same time, so this is not possible. Then m_p is not modified (until another deletion occurs), while $g_p + \Delta_p$ cannot increase, and so we maintain the condition $m_p < g_p + \Delta_p$. So, when we insert a new node and initialize $\Delta_e = m_{par(e)}$, then we have $\Delta_e + g_e = \Delta_e + 1 \leq g_{par(e)} + \Delta_{par(e)} \leq f_{max}(par(e))$, which thus ensures that Property 2 is met.

Case 3. *Deletion of e which is stored by S'_{full}.* We know that e is stored by both S_{full} and S'_{full}, and has the same value of f_{max} in both. However, in order to delete e from S'_{full}, we must be sure that it has no descendants. If e is one of the fringe nodes, then descendants of e may be present in S'_{full} correspond to e in S_{full}. However, by Property 2, since these have f_{max} no greater than their ancestor, then if e can be deleted in S_{full}, by Property 1 and 2, these descendants can all be deleted, and then e itself can be deleted. We update the values of $m_{par(e)}$ identically in both cases, ensuring the preservation of Property 1. □

To complete the proof, we argue that the set of prefixes from FR stored by S'_{full} is a subset of the set of prefixes that would be stored by the Lossy Counting algorithm of Manku and Motwani [2002] run on the same stream.

LEMMA 5. *Given the same input, our full ancestry algorithm will never store more (leaf) elements than* `LossyCounting`.

PROOF. We argue that if an item is stored by our algorithm, then it is also retained by Lossy Counting. We consider the sequence of insertions of elements. Suppose our algorithm encounters some element that it is not currently storing. We use $\Delta_e = m_{par(e)}$ to insert the item with, noting that trivially $m_e \leq b_{current} - 1$ (this property is preserved by every operation that affects m_e in the algorithm given in Figure 5). Then there are two cases to consider:
(i) the item e is already being stored by Lossy Counting algorithm. Then, in Lossy Counting, the item has $g'_e + \Delta'_e \geq b_{current}$ (else it would have been compressed at the previous bucket boundary), while we insert the item with $c = 1$ so $g_e + \Delta_e = m_{par(e)} + 1 \leq b_{current} - 1 + 1 \leq g'_e + \Delta'_e$.
(ii) the item e is not already being stored by Lossy Counting. Then, we insert the items e with $\Delta_e = m_{par(e)} \leq b_{current} - 1 = \Delta'_e$, and $g_e = g'_e = 1$. In this case also, $g_e + \Delta_e \leq g'_e + \Delta'_e$.

From this point on, we do not change Δ'_e or Δ_e, and we update g'_e and g_e by the same amount for every insertion. Hence, the inequality $g_e + \Delta_e \leq g'_e + \Delta'_e$ remains. We therefore conclude that for as long as the item e is stored by our algorithm it is also stored by Lossy Counting: since the occurrence of the prefix in the Lossy Counting algorithm has a higher value of $g + \Delta$, it will never get deleted before the copy in the full ancestry algorithm (both algorithms use the same condition for deletion, testing whether $g + \Delta$ is less than $b_{current}$). Hence, the space required to store the fringe is at most that used by Lossy Counting to represent the input. □

The space used by Lossy Counting is at most $O(\frac{1}{\epsilon} \log \epsilon N)$ [Manku and Motwani 2002]. Lastly, we observe that for each fringe node, there are at most $H - 1$ non-fringe nodes also stored by our algorithm in T (these are the set of all ancestors of the element). So, we

```
Insert (element e, count c):
/* anc(e) is the closest ancestor of e */
/* par(e) is the immediate prefix of e */
01 if t_e exists in T then
02    g_e += c;
03 else
04    create t_e;
05    g_e = c;
06    if t_anc(e) exists in T then
07       Δ_e = m_e = m_anc(e);
08    else
09       Δ_e = m_e = b_current − 1;

Compress():
01 for each t_e ∈ T do
02    if (g_e + Δ_e ≤ b_current) then
03       if (e ≠' *') then
04          Insert(par(e), g_e);
05          m_par(e) = max(m_par(e), g_e + Δ_e);
06          delete t_e;

Output(threhold φ):
/* F_e = Σ_x f_x of non-HHH descendants of e */
01 let F_e = f_e = 0 for all e;
02 Enqueue every fringe node;
03 while queue not empty do
04    Dequeue e;
05    if e not in T then
06       Δ_e = m_a(e);
07    if (g_e + Δ_e + F_e > φN) then
08       print(e, f_e + g_e, f_e + g_e + Δ_e);
09    else
10       F_par(e) += F_e + g_e;
11    f_par(e) += f_e + g_e;
12    Enqueue par(e) if not already in queue;
```

Fig. 6. Algorithm for Partial Ancestry Strategy in one dimension

conclude that the space used by our algorithm is bounded by $O(\frac{H}{\epsilon}\log\epsilon N)$.

LEMMA 6. *Each update in the full ancestry algorithm in one dimension takes amortized time* $O(H\log\epsilon N)$.

PROOF. Each insertion takes time at most $O(H)$, in the case that none of the ancestors of the inserted item are materialized. The amortized cost of compress dominates the overall cost of updates. Each compress requires a linear pass over the data structure, to remove and push up counts of deleted nodes. Since we have just shown that the data structure is bounded by $O(\frac{H}{\epsilon}\log\epsilon N)$, and if we run compress after every $O(\frac{1}{\epsilon})$ insertions, then the amortized cost is $O(H\log\epsilon N)$. Note that this amortized cost can be made into a worst case cost by some careful use of buffers and incremental computation: essentially, instead of doing a full compress after some number of insertions, one does a small amount of compression work (processing $O(H\log\epsilon N)$ items) after every insertion. We omit the details from this presentation, since they are mostly straightforward from this description. □

3.2.2 *Partial Ancestry Algorithm.* We observe that the previous strategy can be wasteful of space, since it retains all ancestors of the fringe nodes, even when these have low (even zero) count. In this strategy, we allow such low count nodes to be removed from the data structure, thus potentially using less space. Thus, it is no longer the case that every

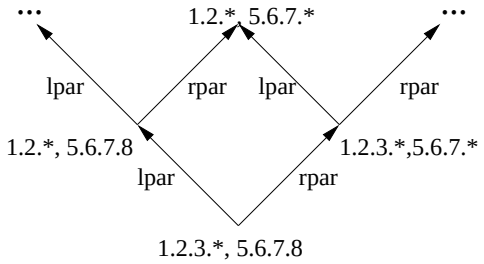


Fig. 7. The diamond property: Each item has at most one "common grandparent" in the lattice.

prefix that is stored has all its ancestors stored as well; hence, we only keep a partial ancestry. The bound m_e is obtained from the closest existing ancestor of the newly inserted element. Figure 6 presents the algorithms for the `Insert`, `Compress` and `Output` operations,

The auxiliary information associated with each element e is (g_e, Δ_e, m_e), which are defined as before. When a new element e is inserted, its Δ_e and m_e are initialized using the auxiliary information of its closest ancestor $anc(e)$ in T using $m_{anc(e)}$. Once the closest ancestor in T has been found, no further operations are required, in contrast to the full ancestry case where intermediate ancestors must also be inserted into the data structure. We compute f_{min} and f_{max} in the same way as for the complete ancestry case. We can show this algorithm, illustrated in Figure 6, is correct as follows.

THEOREM 3. *The algorithm in Figure 6 maintains the accuracy and coverage properties from Definition 3.*

PROOF. The proof of correctness is very similar to that for Theorem 1. For accuracy, observe that as before $f_{min}(p)$ is indeed a lower bound on $f^*(p)$ since it counts a subset of the updates that affected p. By the definition of the condition for deleting in `Compress`, we maintain the uncertainty in counts is bounded by ϵN. The only difference is that we get the bound m_p for inserting a new prefix from some ancestor of p rather than its parent. However, once again we can show that m_p is always less than $b_{current}$, because the m values are set based on $g + \Delta$ values from a deleted node, and for these nodes $g + \Delta \leq b_{current}$. This gives the accuracy condition. For coverage, we again use the bounds on the counts of items to act conservatively: because we compute F_p based on the upper bound of the count for p, less the lower bound on the count of the HHHs already output, then we never underestimate the count for p, and consequently never fail to output p when we need to. □

It seems that the partial ancestry algorithm should always use less space than the full ancestry version, but this is not an immediate consequence. When a new item is inserted, all its ancestors are forced to be present in the full ancestry algorithm, so this appears to use more space; however, it also means that they are inserted with the same value of Δ. In the partial ancestry case, when a prefix is deleted from the data structure, its count is passed on to its parent, which may be inserted if it is not present with a larger value of Δ than in the full ancestry case, and consequently this entry has a higher value of $g + \Delta$ than in the full ancestry case, making it harder to delete. So we do not try to argue that the partial ancestry algorithm will always use less space than the full ancestry, although we observe

in our experiments that this is the case on the data that we test on.

3.3 Multi-dimensional Algorithms

In this section, we consider the multidimensional (overlap) case, where the count of an item being rolled up is given to *each* of its parents. As discussed in Section 2.4, there are many subtleties of this approach that would need to be addressed by an online algorithm. A straightforward rolling up of an element's count to each of its parent elements, iteratively up the levels of the lattice, would result in *overcounting* errors, which are only worsened as the number of hierarchical attributes increases. To give a correct algorithm, we instead update the counts of not only the immediate parents of the deleted element, but also those of "grandparents", "great grandparents" (i.e., parents of parents, their parents) etc., but in a bounded fashion that depends on the dimensionality of the data. Our approach essentially applies the inclusion-exclusion principle, meaning that we add the count of the deleted node to parents, but subtract it from grandparents, add it to great-grandparents, until we reach the unique ancestor of the deleted item, corresponding to the generalization on each of the d attributes. We refer to this as the "diamond" property, since illustrated on a Hasse diagram, it resembles a diamond. This is shown in Figure 7 for 2-d; here the count for node (1.2.*, 5.6.7.*) can be obtained using inclusion-exclusion by adding the count of nodes (1.2.*, 5.6.7.8) and (1.2.3.*, 5.6.7.*), and subtracting the count of (1.2.3.*, 5.6.7.8). More generally, on a d-dimensional lattice, the diamond structure is an embedded d-dimensional hypercube.

More specifically, our algorithms for the overlap case maintain a summary structure T consisting of a set of tuples that correspond to samples from the input stream. Each tuple $t_e \in T$ consists of an element e from the lattice, and a bounded amount of auxiliary information. The algorithms we present for insertion into T, compression of T, and output are non-trivial extensions of the full and partial ancestry algorithms for the 1-d case, to carefully account for the problem of overcounting. With each element e, in the d-dimensional case, we maintain the auxiliary information (g_e, Δ_e, m_e), where:

—g_e is a lower-bound on the total count that is straightforwardly rolled up (directly or indirectly) into e,

—Δ_e is the difference between an upper-bound on the total count that is straightforwardly rolled up into e and the lower-bound f_e,

—$m_e = \max(|g_{d(e)}| + \Delta_{d(e)})$, over all descendants $d(e)$ of e that have been rolled up into e.

3.3.0.1 **Computation of f_{min} and f_{max}..** For any prefix p, we compute

$$f(p) = \sum_{e \preceq p} g_e$$

and from this we set

$$f_{min}(p) = f(p) - \Delta_p \text{ and } f_{max}(p) = f(p) + \Delta_p$$

if p is stored in T, and if not, we set

$$f_{min}(p) = f(p) - m_p \text{ and } f_{max}(p) = f(p) + m_p$$

where, as usual, we compute m_p by finding the minimum m value over all closest ancestors of p.

```
Insert(element e, count c):
01 if t_e exists in T then
02   g_e += c;
03 else
04   create t_e with (g_e = c, m_e = b_current − 1);
05   for p in ancestors of e in T
06     Δ_e = m_e = min(m_e, m_p);

Compress()
01 for l = L downto 0 do
02   for each node t_e at level l do
03     if (|g_e| + Δ_e ≤ b_current) then
04       for j = 1 to 2^d − 1 do
05         p = e; parcount = 0;
06         for i = 1 to d do
07           if (bit(i, j) = 1) then
08             p = par(p, i);
09             parcount += 1;
10         if (p in domain) then
11           factor = 2 * bit(1, parcount) − 1;
12           insert(p, g_e * factor)
13           if (parcount = 1) then
14             m_p = max(m_p, |g_e| + Δ_e);
15       delete(t_e);

Output(threshold φ):
01 F_e = f_e = 0 for all e;
02 for l = L downto 0 do
03 forall label ∈ Level(l) do
04   forall e ∈ D, level(e) ≥ l do
05     p = GeneralizeTo(e, label);
06     f_p += g_e;
07     if (∄h ∈ P : (e ≺ h) ∧ (h ≺ p))
08       F_p += g_e;
09   forall h ∈ P, level(h) ≤ l do
10     p = GeneralizeTo(h, label);
11     if (∄q ∈ P : (h ≺ q) ∧ (q ≺ p))
12       F_p += Δ_h;
13   forall h, h' ∈ P, level(h) ≥ l, level(h') ≥ l do
14     p = GeneralizeTo(glb(h, h'), label);
15     if (∄q ∈ P : ((h ≺ q) ∨ h'(≺ q)) ∧ (q ≺ p))
16       F_p += Δ_h;
17 forall p ∈ Level(l) with f_p > 0 do
18   if (F_p + Δ_p ≥ φN)
19     P = P ∪ {p};
20     print(p, f_p, f_p + Δ_p);
```

Fig. 8. Multidimensional Algorithm with Partial Ancestry

In Figure 8, we present the online algorithm for the d-dimensional case. Here we show the algorithm with Partial Ancestry. The Full Ancestry case is almost identical; the difference is that we insert each parent of e with count 0 when a new element e is inserted, and in the compress phase, we only delete items that have no descendants. As in the algorithms for the one dimensional case, the input stream is conceptually divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$, and the current bucket number is denoted as $b_{current} = \lfloor \epsilon N \rfloor$. The insertion phase is very similar to that of previous cases.

During compression, the algorithm scans through the tuples in the summary structure, and deletes elements whose upper bound on the total count is no larger than the current bucket number. When we find an item that can be deleted, we need to allocate its count to ancestors. In the one dimensional case, this meant simply allocating the count to its immediate parent. To generalize this to multiple dimensions requires us to apply the inclusion-

exclusion technique mentioned above: we add the count to all parents, subtract it from (common) grandparents, and so on. Concretely, suppose e is to be deleted. We consider the common ancestor a, defined by generalizing e on each of its non-general dimensions: $a = par(par(\ldots(e,1),2),\ldots d)$. For this discussion, assume e was non-general on all dimensions (dimensions that are general will, in effect, be ignored). Taking a and e together, we induce a sublattice of the lattice structure, consisting of all prefixes p such that $e \preceq p \preceq a$.

This sublattice is also a lattice, and contains 2^d prefixes, forming the structure of a d-dimensional hypercube. Each prefix in the lattice can be associated with a bit string of d bits, where the ith bit is 0 if e has not been generalized on dimension i, and 1 if it has. Thus, e is associated with 0^d, a with 1^d, and 10^{d-1} is $par(e,1)$. If e is at level l in the lattice, then a is at level $l-d$. The weight function, wt, applied to a bitstring, counts the number of 1s in the string. Therefore, the level of a prefix in the sublattice with binary label b is $l - wt(b)$.

Depending on the distance of a prefix in the sublattice, we either add or subtract the count of the deleted prefix e: we subtract g_e from the counts of prefixes with $wt(b) = 1$ (i.e., the parents), add g_e to the counts of prefixes with $wt(b) = 2$ (the common grandparents), and continue to alternately subtract (odd weight labels) and add (even weight labels) to all prefixes in the sublattice defined by a and e. This is performed in lines 4 to 12 of the `Compress` algorithm in Figure 8. The counter j cycles through all the binary labels, and the loop in lines 6–9 creates the prefix corresponding to the current value of j, and also computes $wt(j)$ as $parcount$. We use the function $bit(i,j)$, which returns the ith bit of the integer j when written in binary. The g_e count is added if the binary label has odd weight (i.e. if its least significant bit is 1), and subtracted if even (least significant bit is 0). Lastly, we update the m values for the immediate parents of e (we only update immediate parents: if these are subsequently deleted, then the m values of their parents will get updated in turn). This is carried out in lines 13–14: a prefix is an immediate parent of e if the weight of its binary label is 1.

LEMMA 7. *f_{min} and f_{max} give correct upper and lower bounds on the sub-lattice count of all prefixes.*

PROOF. Fix an arbitrary prefix p and consider how $f_{min}(p)$ and $f_{max}(p)$ vary over the sequence of operations. Initially $f_{min}(p) = f_{max}(p) = f^*(p) = 0$. We proceed inductively over the sequence of insert and compress operations. For an insert operation, suppose e is inserted. If $e \preceq p$, then $f(p)$ increases by 1, either because the existing node t_e has its value of g_e increased, or because a new node t_e is inserted, with its value of g_e initialized to 1. By analogy with previous cases, the value we compute for $f_{max}(p)$ is an upper bound, because we bound the largest possible uncertainty in the sublattice count by our m and Δ values, which are in turn bounded by $b_{current} = \epsilon N$. Because we may delete some entries whose g_e value is small and negative, f_p is not a lower bound, but since we ensure that any deleted value satisfies $|g_e| + \Delta_e \leq b_{current}$, we can lower bound the sublattice count by $f(p) - \Delta_e$, i.e. $f_{min}(p)$ is a lower bound on $f^*(p)$. Lastly, if $e \not\preceq p$, then $f_{min}(p), f_{max}(p)$ and $f^*(p)$ all remain the same. Hence, the insert operation correctly maintains the bounds on the true count.

We now focus on the compress operation. Here we must make critical use of the structure of the lattice and hypercubes to show that our counts remain accurate. Let e be a node that is deleted in a compress operation. if $e \not\preceq p$ then $f_{min}(p)$ and $f_{max}(p)$ do not

change, since the only g values that are altered belong to nodes q such that $e \preceq q$. But $e \not\preceq p \Rightarrow q \not\preceq p$, and so the computation of $f(p)$ is unaffected. We now argue in the case when $e \preceq p$, unless $e = p$, that $f(p)$ is also unchanged and so $f_{min}(p)$ and $f_{max}(p)$ remain the same. The reason for this is that although counts are increased and decreased within the sublattice defined by a and e, the net effect on any sublattice defined by p remains the same.

Consider the effect on $f(p)$ when e is deleted. By the above analysis, $e \prec p$. Therefore, there must exist at least one dimension i of e such that $par(e, i) \preceq p$. Take any e' such that $e \preceq e' \preceq p$ and e' agrees with e on dimension i. Then we argue that $par(e', i) \preceq p$, because of the lattice properties: the least upper bound (or "lub" in lattice terminology) of e' and $par(e, i)$ is $par(e', i)$, and we are guaranteed that the lub exists in the lattice. Thus we can establish a bijection between all $e' \preceq p$ that agree with e on dimension i, and $par(e', i)$. When e is deleted, the effect is to add g_e to e' and subtract g_e from $par(e', i)$, or vice-versa. Hence, for each $e', par(e', i)$ the net effect on $f(p)$ is zero. Summing over all e', we see that there is no overall change in $f(p)$ (unless $e = p$).

Thus, the only way that $f(p)$ can change in a compress operation is when p itself is deleted. In this case, we establish that $|g_p| + \Delta \leq b_{current}$. Hence, our uncertainty in the count of the sublattice of p remains bounded by $b_{current}$, which in turn is bounded by ϵN, giving the required accuracy bounds. □

3.3.1 *Output Procedure.* At any point, we can extract and output HHHs given a user-supplied threshold ϕ. In multiple dimensions, the task of outputting the Hierarchical Heavy Hitters is rather more involved than in the one-dimensional case. This is because certain nodes may have multiple of their ancestors declared to be HHHs, meaning that manipulation of counts has to be handled with more care in order to meet the coverage requirements of the definition of the problem. Recall that the coverage requirement is one sided: we must guarantee that any node that is not output has a sublattice count (adjusted for descendant HHHs) that is at most ϕN. As long as we take a conservative approach, we can guarantee correctness; our goal is to produce an output that is as small as possible but that has this guarantee. Therefore, the better approximation we can give to the adjusted sublattice counts, the fewer unnecessary nodes will be output.

For example, consider the naive algorithm. This outputs any prefix p whose sublattice count exceeds the threshold ϕN. This makes no adjustment for HHH descendants, and consequently outputs potentially many more nodes than are strictly necessary (in particular, if any node is output, then so too are all its ancestors). This clearly satisfies correctness, but since we have much more information available (we can compute accurate bounds on the sublattice count of any given node), we can hope to do much better.

Our approach is based on estimating the adjusted sublattice count itself. We can apply the inclusion/exclusion principle in order to get an accurate answer. As in the one dimensional case, we must proceed bottom up through the lattice progressively computing the HHHs level-by-level. At any point, we will have created a set P of nodes that have been output as HHHs. For any node p currently under consideration, we must compute a count that discounts the counts of any items that we have already output as HHHs in the set P. We define $H_p \subseteq P$ as the set of $h \in P$ such that $\nexists h' \in P : h \prec h' \prec p$. This is the subset of the output HHHs that affects the computation of the adjusted count. We can then compute the adjusted sublattice count of p by taking $f_{max}(p) - \sum_{h \in H_p} f_{min}(p)$. Note that because we are trying to compute an upper bound on this count, we always add upper

bounds and subtract lower bounds. This first approximation of the result is potentially too low, since for nodes that are covered by two or more members of the set H_p, we have subtracted their count twice. Thus we need to compensate by adding their counts back on to the sum, which will result in an overestimate of the sublattice count (i.e., a conservative estimate, as required).

This procedure could be continued, applying further stages of inclusion/exclusion. However, we will see that proceeding further will only increase the upper bound. The bound we produce consists of the sum of g values not covered by the elements in H_p, plus some additional Δ values. Applying further rounds of inclusion/exclusion while maintaining conservative counts keeps the same set of g values, but adds additional Δ values. Hence, the tightest upper bound comes from applying a single round of inclusion/exclusion. This corresponds to computing

$$f_{max}(p) - \sum_{h \in H_p} f_{min}(h) + \sum_{q=glb(h \in H_p, h' \in H_p)} f_{max}(q)(|Dom(q, H_p)| - 1)$$

This makes use of two lattice theoretic notions, the greatest lower bound (glb) of two nodes in the lattice, and the dominating set of a node relative to a set of nodes in the lattice. We write $q = glb(h, h')$ if q is the unique element in the lattice that satisfies $\forall p : (q \preceq p) \wedge (p \preceq h) \wedge (p \preceq h') \Rightarrow p = q$. We define the Dominating set of q relative to H_p as the set $Dom(q, H_p) = \{h \in H_p | q \prec h\}$, the subset of H_p that dominates q under $\prec$. In the second term of the computation, the count of q is subtracted once for each member of its dominating set; however, it only needs to be subtracted off once. The last term then compensates for this over-reduction by adding on sufficiently many copies of the sublattice count.

We can take advantage of the structure of our counts in order to implement this compensation efficiently. Because the sublattice count of a node corresponds to summing the g values of all nodes within the sublattice, then computing the sublattice count of a node p and subtracting the sublattice count of a node p that is contained within it, corresponds to summing the g values of all nodes q such that $q \preceq p$ but $q \not\preceq h$. Hence, computing the adjusted sublattice count for a node p can be done efficiently over our data structures by computing the sum g values for all nodes not dominated by members of P. We then compute the necessary compensations by (i) adding the Δ values for all members h of P that are below p and not dominated by other members of P, which corresponds to subtracting the lower bounds; and (ii) adding Δ values for all glb values of pairs of undominated members of P, corresponding to adding on the upper bounds of the values subtracted twice. Note that some of these glb nodes may not be explicitly materialized in the data structure, and hence we will have to instantiate their Δ values by use of the m values as when we insert a new node. Formally, for node p we compute the conservative upper bound F_p as

$$F_p = \sum_{(e \preceq p) \wedge (\forall h \in P. e \not\preceq h)} g_e + \sum_{h \in P, h \preceq p} \Delta_h + \sum_{(h,h' \in P) \wedge (h \preceq p) \wedge (h' \preceq p) \wedge (q = glb(h,h'))} \Delta_q$$

This routine is implemented in the pseudo-code for `output` in Figure 8. We make use of the function `GeneralizeTo` defined in Section 2.1. For each prefix p we compute two values: f_p is used to give the upper and lower bounds on the sublattice count, as required for accuracy. F_p is used to test whether to output p as one of the HHH nodes. If the upper bound on the count of p using F_p exceeds the threshold, we output p and its sublattice

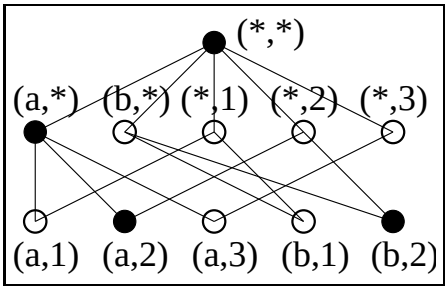
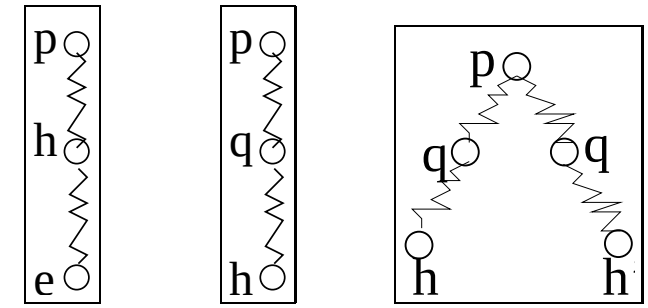


Fig. 9. Example lattice with elements $(a.1), (a.2), (b, 1), (b, 2), (b, 3)$



(a) Line 07 (b) Line 11 (c) Line 15

Fig. 10. Illustration of output routine shown in Fig 6.

counts.

The correctness of the `output` algorithm follows from the above descriptions. The pseudocode computes f_p for every node by passing over the data structure T and summing the g_e values of all $e \preceq p$. For those nodes that are output, it prints f_{min} and f_{max}. As shown in Lemma 7, these give tight bounds on the sublattice count for any prefix. We satisfy the coverage criterion by pursuing the "conservative" approach: the algorithm also computes the quantity F_p defined above which corresponds to an upper bound on the sublattice count after removing the HHHs that have already been output. This we compute similarly to f_p, except that nodes already covered by some $h \in \mathcal{HHH}$ do not contribute; we also have to include the adjustment for potential overcounting of the HHH nodes, by including their Δ values. Lines 04-08 compute the f_p and first term of the F_p values; lines 09-12 compute the second term of the F_p values and lines 13-16 compute the final term of the F_p values. Lastly, lines 17-20 consider the materialized nodes at the current level and determine which to include as HHHs.

Example. Figure 9 illustrates a toy example lattice to demonstrate during the output routine how the discounted frequency F_p is calculated at the root node $p = (*, *)$. Initially, F_p was set to zero and the nodes for $(a, 2), (b, 2)$ and $(a, *)$ were marked as HHHs. Figure 10(a) illustrates the predicate in line 07, which is satisfied only by leaf $e = (b, 1)$; therefore, after lines 04–08, $F_{(*,*)} += g_{(b,1)}$, and that is the only leaf count added to $F_{(*,*)}$. Figure 10(b) illustrates the predicate in line 11, which is satisfied only by leaf $e = (b, 1)$; therefore, after lines 09–12, $F_{(*,*)} += \Delta_{(b,2)}$ (note that $\Delta_{(a,2)}$ is *not* added to $F_{(*,*)}$). Figure 10(c) illustrates the predicate in line 15. After lines 13–16, nothing is added

to $F_{(*,*)}$ because $h = (a, 2)$ and $h' = (b, 2)$ is the only pair of HHHs and there exists a q between one of these HHHs and the root, namely, $q = (a, *)$.

THEOREM 4. *The algorithm given in Figure 8 computes HHHs accurately to ϵN. The space used by the online algorithm for the overlap case with full ancestry is bounded by $O((H/\epsilon)\log(\epsilon N))$.*

PROOF. We demonstrate the space used by the full ancestry is never more than that used by the naive algorithm. First, we show a monotonicity property about counts in the naive algorithm, which means that the full ancestry property is enforced in the naive algorithm. From this property then we can easily show the claimed result.

LEMMA 8. *For any element e stored by the naive algorithm, and for p such that $e \preceq p$, then $g_e + \Delta_e \leq g_p + \Delta_p$.*

PROOF. We show this by induction over the sequence of operations that alter information about e and p. The base case is when the data structure is empty, and the inductive hypothesis is trivially satisfied. For insertions, there are four cases to consider:
Case 1: both e and p are present in the data structure. If the insertion affects some node $q \preceq e$, then g_e and g_p increase by the same amount, and Δ_e and Δ_p are unchanged. Else, if the insertion affects $q \preceq p$ but $q \not\preceq e$, then g_e stays the same while g_p increases. Either way, the hypothesis remains true.
Case 2: e and p are both absent. If the insertion affects both e and p, then both are created with identical values of $g = 1$ and Δ; if it affects only p then the hypothesis does not apply, since it only considers nodes e stored in the data structure.
Case 3: e is present but p is absent. We argue that this cannot occur if the induction hypothesis was always true up to this time: for p to be absent but e present then p must have been subject to a deletion (since it would have been inserted at the same time e was inserted). But for p to have been deleted means that $g_p + \Delta_p < b_{current}$ at the time of deletion, whereas $g_e + \Delta_e \geq b_{current}$ at the same time of deletion. This contradicts the inductive hypothesis, so it cannot have happened.
Case 4: p is present but e is absent. Because p is present in the data structure, we know that $g_p + \Delta_p \geq b_{current}$, else it would have been deleted on the last bucket boundary. The naive algorithm inserts e with $g_e = 1, \Delta_e \leq b_{current} - 1$ and so the inductive hypothesis is satisfied.
Lastly, for compression operations, observe that if e is not deleted, then its counts do not change; if e is deleted, then it is no longer present in the data structure, and so the statement does not apply.

Since every operation maintains the truth of the inductive hypothesis, we conclude that the claim is true. □

Note that an important part of the case analysis showed that if e is present in the data structure for the naive algorithm then, relying on the monotonicity of the $g + \Delta$ values, any ancestor p of e must also be stored. We now conclude the proof by arguing that any node e stored by our algorithm is also stored by the naive algorithm.

Consider some node e that is stored by the full ancestry algorithm. If e is a fringe node — that is, no descendants of e are stored — then by the argument of Lemma 5 it is also stored by the naive algorithm. If e is not a fringe node, then there exists some $q \prec e$ which is a fringe node. By the same argument, this node is also stored by the `LossyCounting`

algorithm, and because we have argued that if the naive algorithm stores any q then it also stores all ancestors of q, then e is also stored by the naive algorithm. Hence, the space required by the full ancestry algorithm is never more than that used by the naive algorithm, and can be much less.

LEMMA 9. *The amortized update cost for the full ancestry algorithm in multiple dimensions is* $O(H \log \epsilon N)$.

PROOF. The proof of this follows from the analysis in Lemma 6 for the one dimensional algorithm. Each insertion takes worst case time $O(H)$ if we have to create every ancestor of the inserted node. If compressions take place after every $O(\frac{1}{\epsilon})$ insertions, and require a linear pass over the data structure then, since the data structure is bounded by $O(\frac{H}{\epsilon} \log \epsilon N)$, the overall cost is dominated by this amortized $O(H \log \epsilon N)$ cost. As before, this amortized cost can be made worst case with a suitable implementation. □

4. EXPERIMENTS

In this section we evaluate both the effectiveness and efficiency of our proposed online strategies, Full Ancestry and Partial Ancestry, for 1-d (trie) and 2-d (lattice) prefix hierarchies. The space usage was quantified using two measures: the size of the output sets generated by the algorithm and the amount of memory used during execution. As a yardstick, we consider the size of the (exact) output from offline computation of HHHs. We compared the performance of these algorithms in various ways: in terms of the number of insertion and deletion operations to the data structure; in terms of the quality of the approximate output compared to the exact algorithm; and using timings based on their implementations in a live data stream management system. For each strategy, we tested several implementation alternatives including (a) the choice of data structure; (b) amortized vs. non-amortized compression; and (c) recursive vs. non-recursive insertion.

For comparison purposes, we also tested the naive algorithm, based on `LossyCount` [Manku and Motwani 2002][2], to find heavy hitters on the set of all multi-dimensional prefixes of all stream items. Whereas this algorithm uses two auxiliary variables (the minimum frequency, f, and the difference between the maximum and minimum frequencies, Δ) and the proposed algorithm uses three ($g, \Delta,$ and m), the overall storage ratio between these data structures depends on the overhead of storing item identifiers, which dominates the total space usage. Hence, we do not normalize by the individual tuple sizes but instead report results in terms of the number of tuples.

We used real IP traffic data in our experiments, containing source and destination IP addresses (among other fields) from network "flow" measurements (FLOW), and packet traces (PACKET). For the experiments on two-dimensional hierarchies, we projected on source-destination IP address pairs; for those on a single dimension we projected only on the source address. To examine the effect of the "bushiness" of the hierarchy, we varied the granularities of the hierarchies: the IP address spaces were viewed on the byte-level ("octets") for some experiments, and on the bit-level for others. The source used for the performance experiments was a live IP packet stream monitored at a network interface inside the AT&T Customer Backbone. On average, the streaming rate at this interface was

[2]While the asymptotics of other heavy hitter algorithms [Misra and Gries 1982; Demaine et al. 2002; Karp et al. 2003; Metwally et al. 2005] are better, `LossyCount` has exhibited the best performance in practice, especially on skewed data sets, as noted by Manku and Motwani [2002] (last para of Sec 4).

about 100,000 packets/sec, or about 400 Mbits/sec. Although the traffic rate fluctuates slightly over the course of a day, such changes are gradual and occur in regular (diurnal) cycles.

We compared the candidate strategies using the User-Defined Aggregate Function (UDAF) facility in Gigascope, a highly optimized system for monitoring very high speed data streams [Cranor et al. 2003]. Gigascope has a two-level query architecture: at the low level, data is taken from the Network Interface Card (NIC) and is placed in a ring buffer; queries at the high level then run over the data from the ring buffer. Gigascope *creates* queries from an SQL-like language (called GSQL) by generating C and C++ code, which is compiled and linked into executable queries. To integrate a UDAF into Gigascope, the UDAF functions (also in C/C++) are added to the Gigascope library and query generation is augmented to properly handle references to UDAFs; for more details, see [Cormode et al. 2004].

4.1 Implementation Details

Translating a complicated algorithm into a fast implementation is rarely trivial, and requires particular care when running it in real-time on a live IP traffic stream. We focus on the details pertaining to implementation of these algorithms in the context of Gigascope UDAFs. Below we describe the data structures and optimizations used for our proposed strategies (for both 1-d as well as 2-d prefixes), as well as for the naive approach of computing heavy hitters on all prefixes of all items, which was used as a basis of comparison. We focus on implementation choices at the high-level query within Gigascope. At the low-level, we employed two different basic processing alternatives: (a) buffering items in an array for blocks transfers; and (b) buffering items as a weighted set using a hash table.

The performance of the algorithms depends partly on the frequency with which the `Compress` routine is run. This can be as often as after every `Insert` operation; after every $1/\epsilon$ tuples; or with some other frequency. Note that the frequency of compressing does not affect the correctness, just the aggressiveness with which we prune the data structure.

We now describe how the individual strategies were implemented:

—**Naive:** This method is based on `LossyCount` and maintains a set of tuples (e, f, Δ) in a hash table on item identifiers e. For each incoming stream item *and all of its prefixes*, a lookup is issued to find an existing tuple for that item. If one exists, f is incremented by 1; otherwise, new tuples are created for all prefixes. For amortized compression, at each step ϵs tuples are visited by traversing the hash table sequentially (note that traversal order does not matter for this algorithm), to determine which nodes can be pruned.

—**Full Ancestry:** This method, henceforth called FullAnces, maintains tuples (e, g, Δ, m) in a hash table on identifier values e. Hence, this data structure effectively implements a trie for one-dimensional hierarchies (resp. lattice for two-dimensional hierarchies). We implemented two variants of this strategy: one that stores pointers to the parent(s) of each node and one that requires hash lookup to retrieve parents based on the node identifier. For each incoming stream item, a lookup is issued to find an existing node for that item. If one exists, g is incremented by 1; otherwise, new lattice nodes are created (with $g = 0$ for nonleaf nodes) up to a closest ancestor(s). Unlike the naive method, which updates every ancestor in every path from e leading up to the root, this method stops updating along a path whenever the closest ancestor is encountered.

This method also maintains a *fringe*, which is the set of nodes without children. The

fringe is used for efficient bottom-up pruning by focusing compression at only the nodes that need to be visited (recall that this strategy only considers nodes having no children for deletion). The fringe must be dynamically maintained during insertion and compression; hence, we employ a hash table for access. During the compress phase, a queue is employed to enforce a proper traversal order, since a node should not be visited until all its children have been visited) by enqueuing parents of visited nodes; compression iterates until the queue is empty. For amortized compression, ϵs nodes are dequeued at each time step.

—**Partial Ancestry:** This method, henceforth called PartialAnces, maintains the nodes as L distinct sets of tuples (e, g, Δ, m), one for each level of the trie/lattice, with a separate hash table on each level. For each stream item a single lookup is issued, resulting in either an increment or creation; prefix nodes are not created. Since intermediate nodes (i.e., nodes with children) are considered for deletion, there is no performance benefit to maintaining the fringe as with FullAnces. Bottom-up traversal therefore proceeds level-by-level. During the compress phase, nodes across each level are visited sequentially by hash value (note that nodes at the same level can be visited in any order). For amortized compression, ϵs nodes are visited at each time step, in level order.

These methods were implemented in C++ and attempts were made to make the three implementations as uniform as possible for a fair comparison. The C++ STL `hash_multiset` container type was used for efficient hash access to the tuples. Insertions for naive and FullAnces strategies, which require all prefixes, were implemented both recursively and iteratively. We considered several other implementation possibilities but eliminated them since the ones described above performed better.

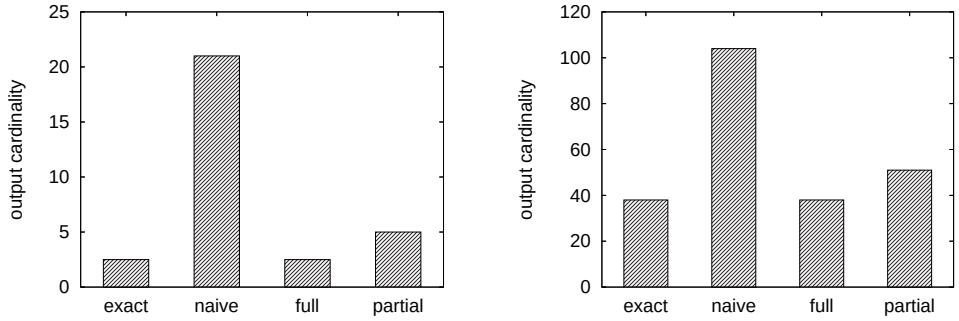
4.2 Experiments on One-dimensional Data

Space Usage.. In the first set of experiments, we compared the output sizes of the online strategies, and included the exact (offline) output size as a frame of reference. Figure 11 summarizes the results using FLOW at time step 100K with (a) $\phi = 0.2$ $(\epsilon = 0.1)$ and (b) $\phi = 0.02$ $(\epsilon = 0.01)$. The naive strategy clearly gives the largest output size and the difference from the output sizes of the proposed strategies grows in cardinality with smaller ϕ and ϵ. Note how close the output size from the Full Ancestry strategy is to the exact output size. The output size of the Partial Ancestry strategy is significantly less than that of the naive one, but also noticeably larger than that of Full Ancestry. Figure 12 presents the results based on PACKET. Here the differences in output sizes are more pronounced.

In the second set of experiments, we report the *a posteriori* space utilization in terms of the number of tuples at each timestep. Figure 13 gives a comparison of the three strategies on PACKET. The left column considered byte-level prefixes (granularity = 8) and the right column bit-level prefixes (granularity = 1). The top row was run with $\epsilon = 0.01$; the bottom row was run with $\epsilon = 0.001$.[3] (The graphs using FLOW were very similar and are omitted for brevity.) The main observation is that PartialAnces used the least space in all cases, especially when ϵ is small and the prefix granularity is small. It is clearly the superior strategy with respect to data structure size.

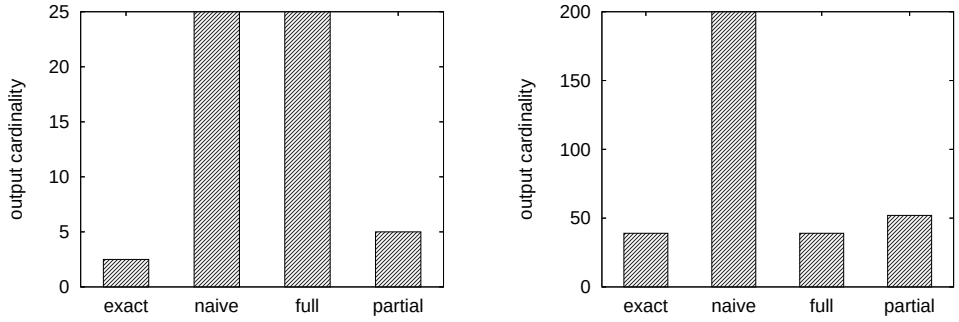
Update Efficiency.. Figure 14 compares the speed of the strategies by measuring the total number of insertion and deletion operations to the summary structure. The data sets

[3]Note that ϕ affects output size but not the space usage during execution.



(a) $\phi = 0.2, \epsilon = 0.1$ (b) $\phi = 0.02, \epsilon = 0.01$

Fig. 11. Comparison of output sizes from the online algorithms on 1-d data, and the exact answer size, using FLOW with bit-level hierarchies.



(a) $\phi = 0.2, \epsilon = 0.1$ (b) $\phi = 0.02, \epsilon = 0.01$

Fig. 12. Comparison of output sizes from the online algorithms on 1-d data, and the exact answer size, using PACKET with bit-level hierarchies.

used were (a) FLOW and (b) PACKET; the operations were totaled after 140K timesteps, with bit-level granularity and $\epsilon = 0.01$. It presents this breakdown as histogram bars where the height gives the sum of all operations. The naive strategy requires slightly more updates than the other strategies because every prefix of every element is inserted. The differences between the proposed strategies are small.

4.3 Experiments on Two-dimensional Data

Space Usage for 2-d Case.. The data structure size of PartialAnces with bit-level granularity was about 7 times more space-efficient than the naive strategy, for different values of ϕ and ϵ ($\phi = 0.2, \epsilon = 0.1$ and $\phi = 0.02, \epsilon = 0.01$). The space differences were even greater using PACKET: PartialAnces gave up to a 40-fold space savings over the naive strategy.

Update Efficiency for 2-d Case.. We have already seen that both proposed strategies yield smaller output sizes and use less space than the naive strategy, with FullAnces having a smaller output than PartialAnces but PartialAnces being more space-efficient with respect

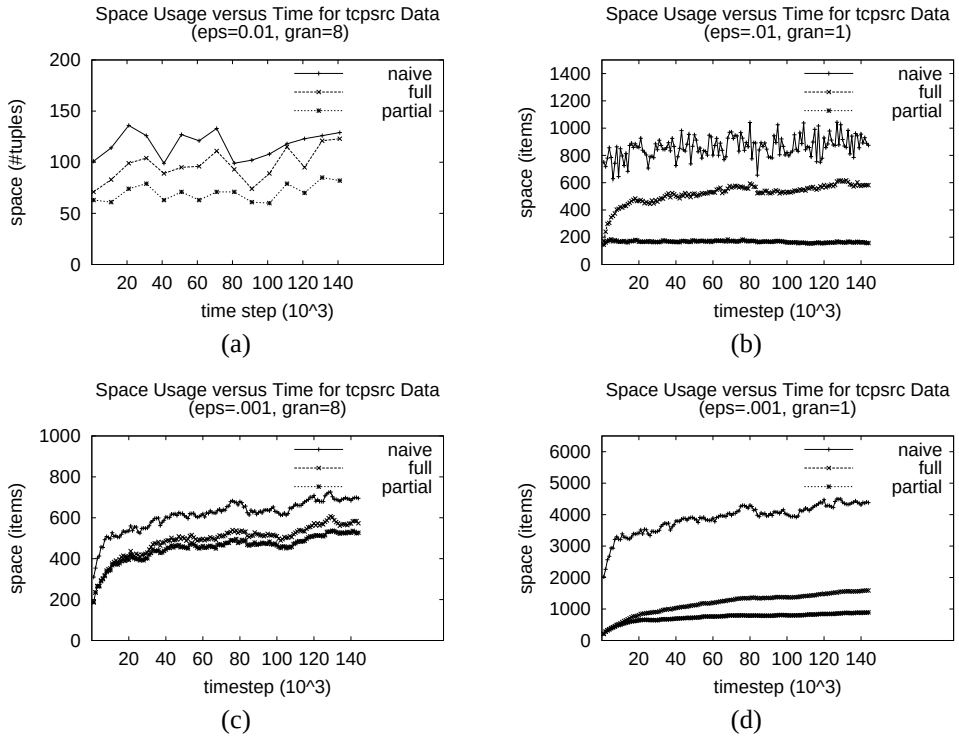
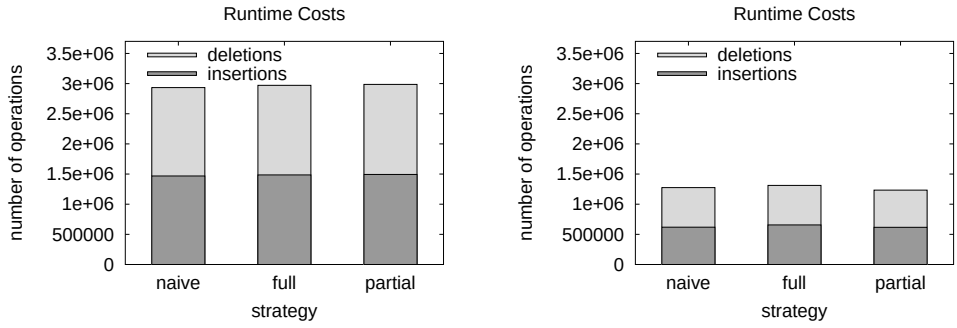


Fig. 13. Comparison of data structure size from the online strategies on 1-d prefixes using PACKET. The left column is at byte-level granularity; the right column is at bit-level granularity. The top row is with $\epsilon = 0.01$; the bottom row is with $\epsilon = 0.001$.

to data structure size. We now consider the performance of the proposed strategies in achieving such benefits.

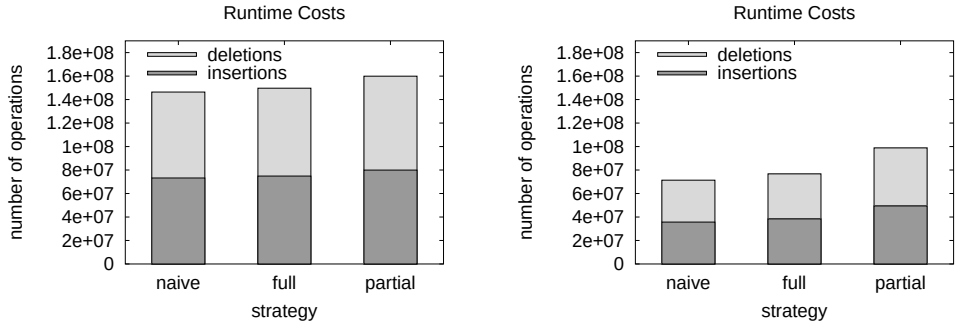
First, to get an implementation-independent comparison, we counted the number of insertion and deletion operations from each strategy over a full pass of the data; see Figure 15. The graphs indicate that, for all strategies, the difference between the number of insertions and deletions is relatively small, which is due to the data structure size remaining fairly constant over time. The graphs also show that the strategies differ in the total number of runtime operations performed, with the naive strategy requiring the least using both FLOW and PACKET data. FullAnces performs slightly more operations than the naive strategy; PartialAnces performs an even greater number. The differences are slightly more apparent with PACKET, though compared to FLOW the number of operations is less for all strategies. It appears that, while PartialAnces was consistently and definitively the best strategy in terms of minimizing space usage, this comes with a performance penalty, due to the extra pruning.

Output Quality.. Next, we study the *quality* of the output. Since we are using approximate algorithms, we cannot hope to find the exact set of HHHs, and so there will be some of these items missing from the output, and some extra. However, the quality and usefulness of this output will vary: for example, we might argue that outputting the parent of an



(a) FLOW (b) PACKET

Fig. 14. Comparing the update efficiency of the 1-d strategies, with $\epsilon = 0.01$ and bit-level prefixes.



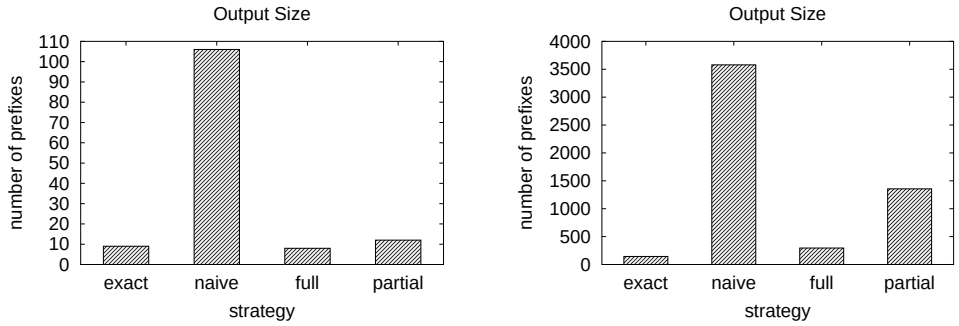
(a) FLOW (b) PACKET

Fig. 15. Comparing the update efficiency of the 2-d strategies, with $\epsilon = 0.01$ and bit-level prefixes.

"exact" HHH instead of the HHH itself is more useful than outputting a distant ancestor. We will introduce a sequence of progressively more refined ways to study the quality of the output.

Firstly, we compared the output sizes of the online strategies, and included the exact (offline) output size as a frame of reference. In practice, this indicates of how valuable the answer is: too large, and there may be too much information to wade through. However, unlike with the 1-d case, a smaller output size is not necessarily better. Therefore, we go on to present precision-recall analysis of the answer sets (with respect to the exact answer) using a variety of scoring functions.

In general, the naive strategy yielded output set cardinalities that were an order of magnitude larger than that of our proposed strategies, for a variety of ϕ-values when $\phi = 20\epsilon$ (a factor of 20 rather than 2 was chosen because of the greater sensitivity to ϕ in 2-d). At the same time, our proposed strategies yielded outputs that were close in size to that of the exact answers computed offline. The output sizes of PartialAnces were larger than those of FullAnces, due to the insertion of intermediate nodes during the output routine (see Fig-



(a) $\phi = 0.2, \epsilon = 0.01$ (b) $\phi = 0.02, \epsilon = 0.001$

Fig. 16. Comparison of output sizes from the online algorithms for the 2-d overlap case, and the exact answer size, using FLOW with bit-level hierarchies.

ure 8), but this difference is eclipsed by that from the naive strategy. The pruning power of the proposed methods appeared to be proportional to the ratio ϕ/ϵ and, though for this data set the order of magnitude difference in output size did not occur for ratios less then 5, there were still benefits at these small ratios. Figure 16 plots the output sizes, at time step 100K, using FLOW with (a) $\phi = 0.2$ ($\epsilon = 0.01$); and (b) $\phi = 0.02$ ($\epsilon = 0.001$), where the hierarchies are induced by considering bit-level prefixes of the IP addresses. Here there is a clear difference between the naive strategy and the others, with PartialAnces yielding slightly larger output sizes than FullAnces, which gave almost the same sizes as the exact query answers. The reduction in output size by the proposed strategies on the PACKET data, using the same parameter values, was even greater: the factors were roughly 35 and 75 for $\phi = 0.2$ and $\phi = 0.02$, respectively. Clearly, the proposed hierarchy-aware strategies are able to filter out a considerable number of prefixes.

To measure the quality of output sets obtained from the various online algorithms with respect to the exact answer, we first use two standard set-based measures of similarity: the Jaccard coefficient $\frac{|A \cap E|}{|A \cup E|}$; and the Dice coefficient $\frac{2|A \cap E|}{|A|+|E|}$, which is the harmonic mean of precision and recall.[4] However, the output objects are multidimensional prefixes at potentially different levels in the lattice, so "flat" set measures are not suitable. For example, how does one compare two prefixes when one is a parent of another? Hence, we used a measure designed for hierarchical domains, the Optimistic Genealogy Measure (OGM) due to Ganesan et al. [2003], and modified it slightly to make more sense for our setting where objects are not all leaf nodes. Thus, we give absolute differences rather than relative ratios and made OGM symmetric as follows:

$$1 - \mathrm{sim}(A, E) = \sum_{a \in A} |\operatorname{depth}(LCA) - \operatorname{depth}(a)| - \sum_{e \in E} |\operatorname{depth}(LCA) - \operatorname{depth}(e)|$$

That is, for each prefix a in the approximate answer A, we find its best match e in the exact answer E, compute the lowest common ancestor $LCA(a, e)$, and retain the difference with respect to a. Symmetrically, we find the best match for each $e \in E$ and compute

[4]We do not consider bag-based measures since the HHH (discounted) frequencies in practice are typically all roughly ϕN.

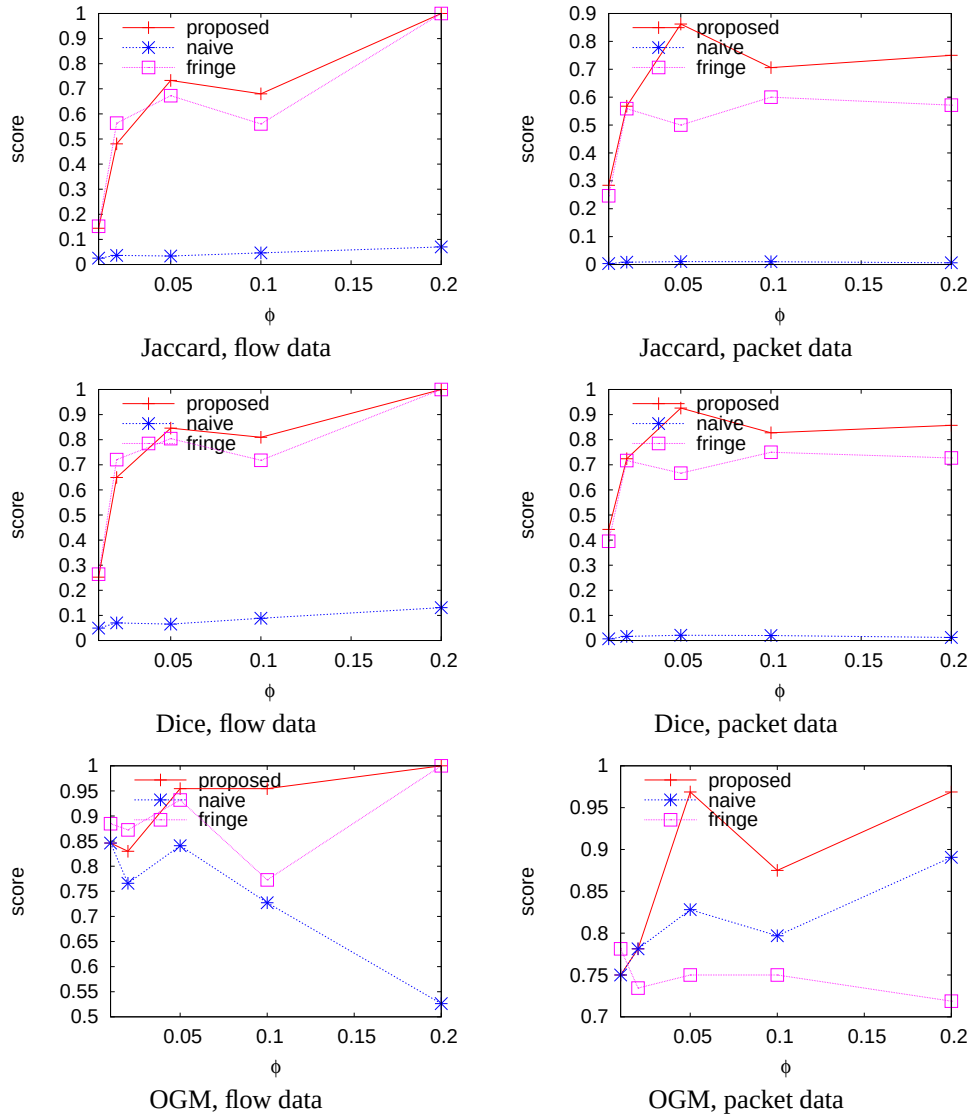


Fig. 17. Comparison of output scores of naive, fringe and proposed online algorithms, with respect to the exact answer, using different similarity measures: flow (left column), and packet (right column) data at bit-level granularity.

$LCA(e, a)$'s. In addition to the naive all-NN output, we show results for the fringe (the subset of all-NN nodes which do not have a HHH descendant), since it is not clear *a priori* which will have a better score under the above measures, which combine both precision and recall.

Figure 17 compares the output of the proposed algorithm with that of all-HH ("naive") and the fringe, using the different score functions at various ϕ-values. Bit-level granularity

Strategy	*CPU utilization*	*user time (μs) per packet*
Lossy Counting	49%	7.5
FullAnces	51%	7.62
PartialAnces	44%	6.6

(a) 1-d prefixes (bit-level)

Strategy	*CPU utilization*	*user time (μs) per packet*
Lossy Counting	48%	6.85
FullAnces	63%	9.18
PartialAnces	63%	10.96

(b) 2-d prefixes (byte-level)

Fig. 18. Average performance for the different methods over live IP packet streams.

prefixes were computed and our proposed algorithm was instantiated with the FullAnces version. The Jaccard and Dice coefficients clearly distinguish that the naive algorithm gives very poor outputs. They also indicate that the proposed algorithm finds better answers than the fringe, but this comparison is better evaluated using the OGM hierarchical measure (rightmost column). Here we observe that, whereas the fringe output was better than naive using flow data, the opposite is true using packet data. So both naive and fringe have bad cases, while the proposed algorithms are consistently good.

4.4 Performance on a Live Data Stream

We measured the performance of the strategies on a live IP traffic stream using Gigascope. We wrote GSQL queries in Gigascope which reported the HHH with the longest prefix length (both prefix label and frequency) at intervals of every minute, for a total duration of 30 minutes. For the 1-d prefix queries, we projected onto `srcIP` over all packets; for the 2-d prefix queries, we selected out only TCP packets projected onto `(srcIP, destIP)`.

We first compared the two low-level alternative implementations described in Section 4.1 and found that, though neither was a bottleneck, hash-based buffering of items as a weighted set was consistently faster than using an array to buffer the multiset, due to the temporal clustering of IP addresses (e.g., during a TCP session). Therefore, the results reported below use the hash-based strategy at the low-level.

At the high-level, we ran the methods over similar live workloads and measured the average CPU utilization and user processing times at 1-minute intervals. Figure 18(a) summarizes the results for HHH on 1-d prefixes at bit-level granularity, with $\epsilon = 0.001$ and $\phi = .05$. The numbers did not vary considerably among the methods, with PartialAnces achieving the fastest processing times, followed by the naive method, followed by FullAnces. For these experiments, the `compress()` operations were amortized rather than doing them all at once at block boundaries; experiments with batch `compress()` gave only slightly slower speeds.

For HHH on 2-d prefixes, none of the algorithms could keep up when using bit-level granularity, so we set the granularity to byte-level in the following experiments. Figure 18(b) summarizes these results. Here we see a difference in the processing speeds of the methods, with the naive method running fastest and the proposed methods doing comparably. There did not appear to be any packet loss for any of the above methods. Emboldened by this, we tried running HHH on 2-d prefixes with a granularity of 2 bits and found that it could just barely keep up with the stream (CPU utilization of all methods were close to 99%). However, the differences in both data structure and output sizes increased dramatically.

We also tested the non-amortized versions of the strategies, where a full compress is performed only at block boundaries, using the same query but the packet loss was so great that we were unable to obtain any measurements. Hence, not only does spreading out the

processing for compression improve performance but, at high CPU utilization, it can make the difference between whether or not the method can keep up with the streaming rate.

Another factor which impacted performance was the error bound. Although larger values of ϵ result in smaller space, more pruning will occur. In fact, for the same query as above, increasing the value of ϵ from 0.001 to 0.01 resulted in packet loss, despite the compress operation being amortized. Indeed, the benefits of amortization decrease at higher values of ϵ due to non-amortized compress processing being more regular and less spiked. There were other optimizations we tried (e.g., removing recursion, reducing hash lookups by storing pointers, etc.) but these had very minor impact so we do not describe them in detail.

4.5 Summary of Experimental Results

With respect to performance, the proposed strategies were competitive with the naive one, requiring only slightly more processing time in general, and slightly less in the case of PartialAnces on 1-d prefixes. As currently implemented and using the CPU speed available in our measurement infrastructure, all of the methods were too slow to compute HHHs on 2-d prefixes at bit-level granularity; for coarser granularities, though, they can keep pace with high speed packet streams.

As shown in Sections 4.2 and 4.3, the space savings of the proposed strategies compared to the naive one were significant, by an order of magnitude, with respect to both data structure and output size. This is due to hierarchy-aware bookkeeping, which results in more accurate frequency estimates and thus smaller data structure sizes, and allows for sophisticated bottom-up calculation of discounted counts and thus smaller output sizes. Our experiments show that the space savings of the proposed strategies, compared to that of the naive one, is more dramatic on 2-d prefixes than on 1-d prefixes; at fine granularity (e.g., bit-level) than at coarse granularity (e.g., byte-level); and with smaller values of ϵ and ϕ.

PartialAnces was the most aggressive at pruning, resulting in slightly slower performance on 2-d prefixes, but always using the least amount of memory; even compared to FullAnces, the data structure size was quite small, as much as a factor of ten at fine prefix granularity and small ϵ. At the same time, PartialAnces yielded slightly larger output sizes than FullAnces. The question of which of these proposed strategies is better depends on several factors:

—In general, a smaller output size means a higher quality query answer; however, it is difficult to quantify the "goodness" of an answer set due to complex combinatorics. Some applications may be very sensitive to false-hits in output sets and thus may benefit more from using FullAnces.

—The space needed during execution of an online strategy may be crucial in some applications, for example, when there are multiple simultaneous groups over which HHHs are computed. When memory usage is of premium importance, PartialAnces may be the best choice.

—Parameter value settings impacted the space usage of the proposed strategies. At larger ratios of ϕ/ϵ when ϵ was small, PartialAnces exhibited the best space usage in both data structure and memory size, but it did not fare as well as FullAnces with respect to output size for ϵ-values close to ϕ.

5. EXTENSIONS

5.1 Merging Summaries

We have so far considered the case where all the data is observed at a single point, and so the summaries for finding HHHs can be computed centrally at that point. A more general scenario arises when the updates are being observed at multiple locations. This can happen in a network, where we observe packets entering or leaving the network at multiple points, and we monitor each of these incoming or outgoing links. It can also happen within a single system, for example in the Gigascope monitoring system, where we periodically want to merge recent observations made by a low level monitoring system with the high level summary of all observations to date. In both cases, we need the ability to merge two summaries over disjoint sets of data to give a summary that is ϵ accurate over the union of the sets of observations. Rather than give specific details of how to achieve this for each of the different algorithms, we outline the main principles in sufficient detail that they can be applied to each of the algorithms in turn.

We focus on merging two summaries, since to merge more, we just have to repeatedly merge each successive summary into one (initially empty) global summary. We assume that each summary is made using the same value of ϵ; if not, our results follow by taking $\epsilon = \max\{\epsilon_1, \epsilon_2\}$. Each node can be considered separately, and we have two cases to consider: when the node is present in both input summaries, or only one. If a node is present in both summaries, then we merge the counts: set f_e to be the sum of the f_e values in each summary, Δ_e to be the sum of the Δ_es, and so on. It is straightforward to show that we now have upper and lower bounds on the count for each node, and further that these differ by at most $\epsilon N = \epsilon N_1 + \epsilon N_2$, on the assumption that the bounds ϵN_1 and ϵN_2 hold for the input summaries. If a node is present in only one of the summaries, then we must be conservative in our setting of the new values. That is, we must insert the item and set $\Delta_e = \Delta_{e,1} + m_{a(e,2)}$ where $\Delta_{e,1}$ is the $\Delta_{e,1}$ is the value of Δ_e from the summary containing e, and $m_{a(e,2)}$ is the value of m we would get if inserting e into the second summary. These are the tightest bounds we can give on e given the available information, but we have guaranteed accuracy from the previous results for inserting items. Lastly, note that if a node is present in neither summary, then we can ignore it without any loss of accuracy.

Although this merging procedure preserves the accuracy of the summaries, its size is not directly bounded. In the worst case, if every merged summary contains a disjoint set of items then the result of merging these summaries can grow without bound. This is because, in the worst case, we are unable to prune any items from the summary in the compress stage, and so the summary can continue to grow without bound. In practice we can argue that such a situation is unlikely to occur, since it would be unusual to see disjoint sets of values in each update. Space bounds can be given based on ideas introduced by Manjhi et al. [2005]: each distributed site runs the algorithm with a smaller ϵ than is needed, and then using the "slack" between this and the desired ϵ in order to prune the result of merging the distributed summaries.

5.2 Dynamic Data with Insertions and Deletions

The algorithms we have discussed so far have input which consists only of arrivals of new items, which may be thought of as insert transactions. One can imagine more general situations where the input stream includes deletions of previously seen items in addition

```
Insert (element e, count c):
01   l = Level(e);
02   for j = l to 0;
03        SketchUpdate(j,e,c);
04        e = par(e);

Output(prefix p,level l):
01   w = SketchQuery(l, p);
02   if w < ⌊φN⌋
03        return 0;
04   else
05        v = 0;
06        foreach child(e) of p do
07             v=v+Output(e, l + 1);
08        if (w − v > ⌊φN⌋)
09             print p;
10             return w;
11        else
12             return 0;
```

Fig. 19. Algorithm using Sketches over one dimension

to insertions. If there are very few deletions relative to the number of insertions, then by simply modifying our online algorithms to subtract from $f_{max}(e)$ to simulate deletions, the results will be reasonably accurate. If there are I insertions and D deletions, then the error in the approximate counts will be in terms of $\epsilon(I + D)$, which will be close to the "desired" error of $\epsilon(I - D)$ for small D. However, if deletions are more frequent, then we will not be able to prove that the counts are adequately approximated, and we will need a different approach.

A *sketch* is a generic term for a small space data structure that allows various properties of a large data set to be approximated with only a bounded amount of space. We will focus on sketches that are randomized data structures that give (probabilistic) guarantees to approximate the counts of items in the presence of insertions and deletions. For this problem, appropriate sketch data structures are those defined in the literature [Gilbert et al. 2002; Charikar et al. 2002; Cormode and Muthukrishnan 2005]. Since all algorithms give similar guarantees, any can be used. Where necessary, we will assume the use of Count-Min Sketches [Cormode and Muthukrishnan 2005], since these have the best bounds. Because they are randomized data structures, sketches also have a parameter δ which bounds the probability of error: with probability at least $1 - \delta$ they are guaranteed to give an answer which has error at most ϵN.

5.2.0.1 *Sketch Algorithm for One Dimensional Data.* . We keep a sketch for each level of the hierarchy. One sketch allows to estimate the counts of individual items (the leaves of the hierarchy). Other sketches allow us to estimate the counts of all leaves that are descendants of individual internal nodes in the hierarchy. Every time a new item arrives or departs, we update the approximate count of the item and of all its ancestor nodes in the sketches.

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 which are descendants of that node. We assume that, given a node in the hierarchy, it is possible to enumerate all children of this node, and to retrieve the index of the parent of the node. The algorithm is given in Figure 19.

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 exceed the threshold of $\lfloor \phi N \rfloor$. We then include any node which exceeds the threshold, after the weight of any HHHs below has been removed. The following analysis follows from the properties of Count-Min sketches:

THEOREM 5. *The space required for this algorithm is that used by h sketches, which is $O(\frac{h}{\epsilon} \log \frac{1}{\phi\delta})$. Updates takes time $O(h \log \frac{1}{\phi\delta})$.*

5.2.0.2 *Multidimensional Sketch Algorithms.* A similar approach to that described above can apply for the multidimensional case, with either split or overlap semantics. We can keep sketches of items—one sketch for each node in the lattice—and starting from *, descend the lattice looking for potential HHHs, then backtrack and adjust the counts as necessary.[5] There is one major disadvantage of this approach, which is that we must maintain a sketch for every node in the lattice, and update this sketch with every item insertion and deletion. Thus the space cost scales with H, the product of the depths of the hierarchies, which may be too costly in some applications. We state the following result:

THEOREM 6. *The space required to identify ϵ approximate Hierarchical Heavy Hitters under insertions and deletions using sketches is $O(\frac{H}{\epsilon} \log \frac{H}{\phi\delta})$.*

The proof case follows immediately, since we just keep H sketches, one for each node in the lattice. In the split case, we keep one sketch for each level in the lattice. Since the counts are divided up so that, over each level, the sum of all adjusted counts is N, then we only need a single sketch over each level to estimate counts accurately up to ϵN. The space bounds follow.

6. RELATED WORK

Multidimensional aggregation has a rich history in database research. We will discuss the most relevant research directions.

There are a number of "flat" methods for summarizing multidimensional data, that are unaware of the hierarchy that defines the attributes. For example, there are histograms [Thaper et al. 2002; Guha et al. 2001] that summarize data using piecewise constant regions. There are also other representations like wavelets [Vitter et al. 1998] or cosine transforms [Lee et al. 1999]; these attempt to capture the skew in the data using hierarchical transforms, but are not synchronized with the hierarchy in the attributes nor do they avoid outputting many hierarchical prefixes that potentially form heavy hitters.

In recent years, there has been a great deal of work on finding the "Heavy Hitters" (HHs) in network data: that is, finding individual addresses (or source-destination pairs) which are responsible for a large fraction of the total network traffic [Manku and Motwani 2002; Cormode and Muthukrishnan 2003; Misra and Gries 1982; Metwally et al. 2005]. Like other flat methods, heavy hitters by themselves do not form an effective hierarchical summarization mechanism. Generalizing HHs to multiple dimensions can be thought of as Iceberg cube [Beyer and Ramakrishnan 1999]: finding points in the data cube which satisfy a clause such as `HAVING COUNT(*) >= n`.

More recently, researchers have looked for hierarchy-aware summarization methods. The Minimum Description Length (MDL) approach to data summarization uses hierarchi-

[5] This is similar to the bottom-up searching approaches in data-cubes.

cally derived regions to cover significant areas [Lakshmanan et al. 2002]. This approach is useful for covering say the heavy hitters at a particular detail using higher level aggregate regions, but it is not applicable for finding hierarchically significant regions, i.e., a region that contains many subregions that are *not* significant by themselves, but the region itself is significant. The case for finding heavy hitters within multiple hierarchies was advanced by Estan et al. [2003] where the authors provide a variety of heuristics for computing the multidimensional HHHs offline.

Subsequent work by Zhang et al. [2004] also considered the topic of HHH detection. However, the authors principally consider the problem of HHH detection without compensating for the count of HHH descendants. The problem therefore simplifies to finding all nodes in the lattice whose count is above the threshold, which in turn can be thought of as maintaining a "fringe" of heavy nodes. The algorithms given by Zhang et al. [2004] give output equivalent to that of what we label the naive algorithm in our context. In one dimension, their worst case bounds are $O(H^2/\epsilon)$, and in two dimensions the space required is $O(AH/\epsilon)$ (where A is the size of the largest anti-chain in the lattice, as before). Our results improve on these worst case bounds significantly, and extend results to arbitrary dimensions.

The material presented in this paper derives from our earlier work [Cormode et al. 2003; 2004], which studied the one-dimensional and the multi-dimensional cases respectively. Here, we extend our prior work with additional algorithms (in particular, PartialAnces for the multi-dimensional case) and give full proofs of important properties of these algorithms as well as analysis of their space and time requirements. We additionally conduct a thorough set of experiments, both offline to evalute the goodness of approximate answers returned with respect to the exact answer, as well as in a real data stream management system (Gigascope), and consider a variety of further extensions.

7. CONCLUSIONS

Finding truly multidimensional hierarchical summarization of data is of great importance in traditional data warehousing environments as well as in emerging data stream applications. We formalized the notion of one-dimensional and multi-dimensional hierarchical heavy hitters (HHHs), and studied them in depth.

For data stream applications, we proposed online algorithms for approximately determining the HHHs to provable accuracy in only one pass using very small space regardless of the number of dimensions. In a detailed experimental study with data from real IP applications, the online algorithms are shown to be remarkably accurate in estimating HHHs.

REFERENCES

AGARWAL, S., AGRAWAL, R., DESHPANDE, P., GUPTA, A., NAUGHTON, J. F., RAMAKRISHNAN, R., AND SARAWAGI, S. 1996. On the computation of multidimensional aggregates. In *Proceedings of the International Conference on Very Large Data Bases*.

BEYER, K. AND RAMAKRISHNAN, R. 1999. Bottom-up computation of sparse and Iceberg CUBE. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. SIGMOD Record (ACM Special Interest Group on Management of Data), vol. 28(2). 359–370.

CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. 2002. Finding frequent items in data streams. In *Procedings of the International Colloquium on Automata, Languages and Programming (ICALP)*. 693–703.

CORMODE, G., KORN, F., MUTHUKRISHNAN, S., JOHNSON, T., SPATSCHECK, O., AND SRIVASTAVA, D. 2004. Holistic UDAFs at streaming speeds. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 35–46.

CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2003. Finding hierarchical heavy hitters in data streams. In *Proceedings of the International Conference on Very Large Data Bases*. 464–475.

CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2004. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 155–166.

CORMODE, G. AND MUTHUKRISHNAN, S. 2003. What's hot and what's not: Tracking most frequent items dynamically. In *Proceedings of ACM Principles of Database Systems*. 296–306.

CORMODE, G. AND MUTHUKRISHNAN, S. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms 55*, 1, 58–75.

CRANOR, C., JOHNSON, T., SPATSCHECK, O., AND SHKAPENYUK, V. 2003. Gigascope: A stream database for network applications. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 647–651.

DEMAINE, E., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2002. Frequency estimation of internet packet streams with limited space. In *Proceedings of the European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 2461. 348–360.

ESTAN, C., SAVAGE, S., AND VARGHESE, G. 2003. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of ACM SIGCOMM*.

GANESAN, P., GARCIA-MOLINA, H., AND WIDOM, J. 2003. Exploiting hierarchical domain structure to compute similarity. *ACM Trans. Inf. Syst. 21*, 1, 64–93.

GILBERT, A. C., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. 2002. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the International Conference on Very Large Data Bases*. 454–465.

GUHA, S., KOUDAS, N., AND SHIM, K. 2001. Data streams and histograms. In *Proceedings of the ACM Symposium on Theory of Computing*. 471–475.

HERSHBERGER, J., SHRIVASTAVA, N., SURI, S., AND TOTH, C. 2005. Space complexity of hierarchical heavy hitters in multi-dimensional data streams. In *Proceedings of ACM Principles of Database Systems*.

KARP, R., PAPADIMITRIOU, C., AND SHENKER, S. 2003. A simple algorithm for finding frequent elements in sets and bags. *ACM Transactions on Database Systems 28*, 51–55.

LAKSHMANAN, L. V. S., NG, R. T., WANG, C. X., ZHOU, X., AND JOHNSON, T. 2002. The generalized MDL approach for summarization. In *Proceedings of the International Conference on Very Large Data Bases*. 766–777.

LEE, J., KIM, D., AND CHUNG, C. 1999. Multidimensional selectivity estimation using compressed histogram information. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 205–214.

MANJHI, A., SHKAPENYUK, V., DHAMDHERE, K., AND OLSTON, C. 2005. Finding (recently) frequent items in distributed data streams. In *IEEE International Conference on Data Engineering*. 767–778.

MANKU, G. AND MOTWANI, R. 2002. Approximate frequency counts over data streams. In *Proceedings of the International Conference on Very Large Data Bases*. 346–357.

METWALLY, A., AGRAWAL, D., AND ABBADI, A. E. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of ICDT*.

MISRA, J. AND GRIES, D. 1982. Finding repeated elements. *Science of Computer Programming 2*, 143–152.

NG, R. T., WAGNER, A. S., AND YIN, Y. 2001. Iceberg-cube computation with PC clusters. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

THAPER, N., INDYK, P., GUHA, S., AND KOUDAS, N. 2002. Dynamic multidimensional histograms. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 359–366.

VITTER, J. S., WANG, M., AND IYER, B. 1998. Data cube approximation and histograms via wavelets. In *Proceedings of the 7th ACM International Conferences on Information and Knowledge Management*. 96–104.

ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., AND LUND, C. 2004. Online identification of hieararchical heavy hitters: Algorithms, evaluation and applications. In *Proceedings of the Internet Measurement Conference (IMC)*.