Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-Dimensional Data

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

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

S. Muthukrishnan†
Rutgers University
muthu@cs.rutgers.edu

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

ABSTRACT

Data items archived in data warehouses or those that arrive online as streams typically have attributes which take values from multiple hierarchies (e.g., time and geographic location; source and destination IP addresses). Providing an aggregate view of such data is important to summarize, visualize, and analyze. We develop the aggregate view based on certain hierarchically organized sets of large-valued regions ("heavy hitters"). Such Hierarchical Heavy Hitters (HHHs) were previously introduced as a crucial aggregation technique in one dimension. In order to analyze the wider range of data warehousing applications and realistic IP data streams, we generalize this problem to multiple dimensions.

We identify and study two variants of HHHs for multi-dimensional data, namely the "overlap" and "split" cases, depending on how an aggregate computed for a child node in the multi-dimensional hierarchy is propagated to its parent element(s). For data warehousing applications, we present offline algorithms that take multiple passes over the data and produce the exact HHHs. For data stream applications, we present online algorithms that find approximate HHHs in one pass, with proven accuracy guarantees.

We show experimentally, using real and synthetic data, that our proposed online algorithms yield outputs which are very similar (virtually identical, in many cases) to their offline counterparts. The lattice property of the product of hierarchical dimensions ("diamond") 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.

1. INTRODUCTION

Data warehouses 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.

Emerging applications in which data is *streamed* also typically have multiple hierarchical attributes. The quintessential example of data streams is the IP traffic data. We consider streams of 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 filesharing port, and so on), or in some coarser way: in [5] 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.

Our focus is on aggregating and summarizing such data. A standard approach is to capture the value distribution at the highest 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 the *flat* methods since they focus on one (typically, the highest) 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 one of the nodes were a heavy hitter, so would all its ancestors. For example, if a 32-bit IP address were a heavy hitter, then so too would all its prefixes. A definition was proposed in [3, 5] where heavy hitters can be at potentially any level of detail, but in order to provide the maximum information for a given summary size, the hierarchical heavy hitters (HHH) discounted for descendants that were also HHHs.

*Supported by NSF ITR 0220280 and NSF EIA 02-05116.
†Supported by NSF EIA 0087022, NSF ITR 0220280 and NSF EIA 02-05116.

In practice, data warehousing applications and IP traffic data streams have not one, but 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 dimensions. 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 in [5], 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 two 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.[1] The choice of how to count depends on the semantics desired for the analysis.

Even if we have a suitable definition of multidimensional HHHs, it is typically computationally very difficult to exactly calculate them. Even a straightforward generalization of some of the flat methods for summarization from one dimension to two proves expensive. For histograms in two dimensions with hierarchical attributes, polynomial time algorithms can be obtained by dynamic programming; still the running times are very high, and even methods to approximate them are computationally very expensive [12, 13]. Calculating such flat summaries at every combination of detail will be prohibitive. For very large data sets, such as data warehouses, random access becomes very expensive, and instead we will look for methods which use only one or few linear passes through the data.

We address the challenge of developing and computing HHHs in multiple dimensions, and our contributions are as follows:

1. We generalize HHHs to multiple dimensions and illustrate different variants of multidimensional HHHs (namely, the *overlap* and *split* cases), giving formal definitions of them. Conceptually they depend on how an aggregate computed for a child node in the multi-dimensional hierarchy is propagated to its parent element(s). The lattice structure of the product of the hierarchies on each dimension gives different ways to "contribute" to the parents.

2. We present two sets of algorithms for calculating HHHs in multiple dimensions. For data warehousing applications, we present offline algorithms that take multiple (sorting) passes over the data and produce the exact HHHs. For data stream applications, we present online algorithms that find approximate HHHs in one pass, with accuracy guarantees. They use a very small amount of space and they can be updated fast as the data stream unravels. As in [10, 3], the algorithms keep upper and lower bounds on the counts of items. Here, the items exist at various nodes in the lattice, and we must keep additional information to avoid over- and under-counting in the presence of multiple parents and descendants. The lattice property of the product of hierarchical dimensions (the "diamond") 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*. Going from dimensions 1 to 2 to 3 entails increasing the number of statistics we need to maintain for candidates but, beyond that, surprisingly, no further statistics are needed, irrespective of the number of dimensions.

3. We do extensive experiments with 2- and 3-dimensional 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.

[1]Here we have considered a binary split. As we shall see in Section 3.4, fractional split combinations are also possible.

2. PREVIOUS WORK

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

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 [13, 6] that summarize data using piecewise constant regions. There are also other representations like wavelets [14] or cosine transforms [9]; these attempt to get 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 [10, 7, 4]. 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 [2]: 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 hierarchically derived regions to cover significant areas [8]. 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 notion of Heavy Hitters was generalized to be over a *single* hierarchy in [3] where the authors defined Hierarchical Heavy Hitters (HHHs). The case for finding heavy hitters within multiple hierarchies is advanced in [5] where the authors provide a variety of heuristics for computing the multidimensional HHHs offline. This work is closest in spirit to ours. Our work here studies the HHHs in multiple dimensions in greater depth, identifying two fundamental variations of the approach as well providing the first-known online algorithms that work with small space, and give provable accuracy guarantees.

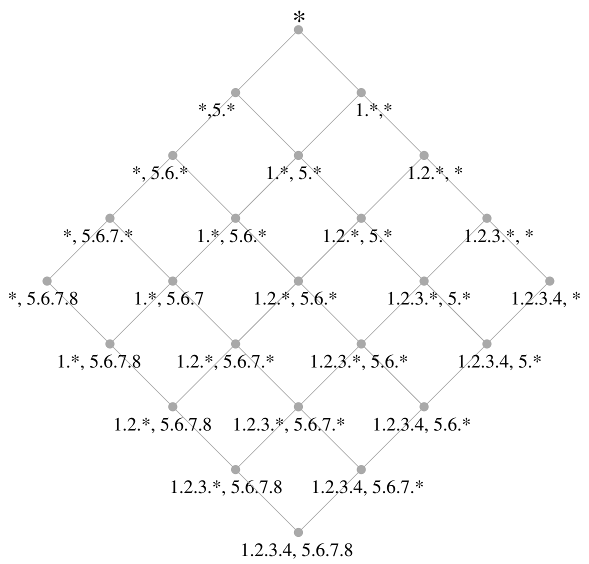


Figure 1: The lattice induced by the element (1.2.3.4, 5.6.7.8)

3. PROBLEM DEFINITIONS

3.1 Notation

Formally, we model the data as N d-dimensional tuples. Each attribute in the tuple is drawn from a hierarchy. Let the (maximum) depth of the i'th dimension be h_i. For concreteness, we shall 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 i'th dimension: $par((1.2.3.4, 5.6.7.*), 2) = (1.2.3.4, 5.6.*)$. An element is *fully general* on some attribute if it cannot be generalized further, and this is denoted *: the pair (*, 5.6.7.*) 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. The operation of generalization 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.

An example lattice induced by the element $(1.2.3.4, 5.6.7.8)$ is shown in Figure 1. Modeling the structure as a lattice is standard on work on computing data cubes and iceberg cubes. It is worth noting that lattices induced by other elements can partially overlap with this lattice. For example, $(1.2.*, 5.6.7.*)$, and all its generalizations, are also part of the lattice induced by the element $(1.2.2.1, 5.6.7.7)$.

In order to facilitate referring to specific points in the lattice, we notationally label each element in the lattice with a vector of length d whose i'th 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 are *comparable* if the label of one is less than or equal to the other on every attribute: items at [3,4] are comparable to ones at [3,2], but [3,4] and [4,3] are not comparable. We define $Level(i)$, the i'th 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. 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 $L + 1$. Lastly, define the *sublattice* of an element e as the set of elements which are related to e under the closure of the parent relation. For instance, (1.2.3.4, 5.6.7.8), (1.2.3.8, 5.6.4.5) and (1.2.3.*, 5.6.8.*) are all in $sublattice(1.2.3.*, 5.6.*)$. We will overload this notation to define the sublattice of a *set* of elements P as $sublattice(P) = \cup_{p \in P} sublattice(p)$.

3.2 Hierarchical Heavy Hitters

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 it is not immediately clear how to compute the count of items at various nodes in the lattice. In previous work considering 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. Then this problem will vary significantly depending on how the count of an element is allocated to its parents. We consider two fundamental variations in the next two sections. These variations 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. The overlap rule says that the full count of the item should be given to each of its parents and, therefore, counted multiple times, in nodes that overlap. The overlap rule is implicit in prior work on network data analysis [5]. However, different summarization semantics may call for different counting schemes. Hence, we also consider the split rule, whereby the count of an item is divided between its parents in some way. This case gives us the useful property that the weight of all items is conserved as they are rolled up, although it may be less intuitive than the overlap rule for many 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 heterogeneous mix of partially and fully specified items, although we do not discuss this case in detail.

In this presentation, we omit all proofs for brevity.

3.3 Overlap Rule

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. 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 a meaningful extension of the 1-d case [3], 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 $f_e = 1$ to denote the count of e), 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. Then, 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. So additional information is needed to avoid overcounting 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 1. **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 ϕ.

- HHH_L contains all heavy hitters $e \in S$ such that $f_e \geq \lfloor \phi N \rfloor$.
- The overlap count of an element p at $Level(l)$ in the lattice where $l < L$ is given by $f'(p) = \sum f_e : e \in S \cap \{sublattice(p) - sublattice(HHH_{l+1})\}$. The set HHH_l is defined as the set
$$HHH_{l+1} \cup \{p : p \in Level(l) \wedge f'(p) \geq \lfloor \phi N \rfloor\}$$
- The Hierarchical Heavy Hitters with the overlap rule for the set S is the set HHH_0. □

PROPOSITION 1. *Let A denote the length of the longest antichain in the lattice. In one dimension, $A = 1$; in two dimensions, $A = 1 + \min(h_1, h_2)$. In higher dimensions, we have $A \leq \prod_{i=1}^{d}(1 + h_i)/\max_i(1 + h_i)$. The size of the set of HHHs under the overlap rule is 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 2. **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: for all $p \in P$, $f_p^* - \epsilon N \leq f_p \leq f_p^*$, where $f_p^* = \sum f_e : e \in S \cap sublattice(p)$; and
2. Coverage: for all items $q \notin P$ in the lattice $\sum f_e : e \in S \cap \{sublattice(q) - sublattice(P)\} < \lfloor \phi N \rfloor$. □

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. We do not know of ways to accurately approximate the discounted count.

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 problem, the exact solution is the smallest satisfying correctness and, hence, a smaller approximate answer size is preferred [3]. In the multi-dimensional problem, one can contrive examples where the approximate output is smaller than the exact one. In practice, we do not expect such situations, and this is borne out by our experiments.

3.4 Split Rule

A simpler alternative to the overlap rule is for the count of an item being rolled up to be split between its parents. This is another meaningful extension of the 1-d case [3], where the count of an item being rolled up is given to its *only* parent. For example, we could give an equal fraction of the count to each parent, or all to one parent, chosen randomly, or by some rule. To encompass these and other variations, we let $par(e, i)$, the i'th parent of node e, to have $s(e, i)$ of the count at e, for $0 \leq s(e, i) \leq 1$ and $\sum_i s(e, i) = 1$.

We shall consider several specific examples of split functions s, including:

- **Even Split.** If e has j parents (e is non-general on j dimensions) then $s(e, i) = 1/j$, for all dimensions i on which e is non-general, and 0 on all other dimensions.
- **Smooth split.** The Even Split Function has a tendency to favor nodes which are closer to the center of the lattice. The Smooth split counteracts this by arranging that all nodes at each level get an equal contribution from each input item. To do this, split weights are assigned to lattice nodes in proportion to the numbers in Pascal's triangle.
- **Random allocate.** $s(e, i)$ is 1 on one non-general dimension, chosen at random *a priori*, and 0 on all others.

Definition 3. **Hierarchical Heavy Hitters with Split 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 ϕ.

- HHH_L contains all $e \in S$ such that $f_e \geq \lfloor \phi N \rfloor$.
- The split count of an item p at $Level(l)$ in the lattice where $l < L$ is given by $f(p) = \sum_{i=1}^{d} s(e, i) * f(e) : p = par(e, i) \wedge f(e) < \lfloor \phi N \rfloor$. The set HHH_l is defined as the set
$$HHH_{l+1} \cup \{p \in Level(l) \wedge f(p) \geq \lfloor \phi N \rfloor\}$$
- The Hierarchical Heavy Hitters with the split rule for the set S is the set HHH_0. □

PROPOSITION 2. *There can be at most $1/\phi$ HHHs from the Split case.* □

Definition 4. **Online HHH Problem: Split Case** The Multi-Dimensional Hierarchical Heavy Hitters problem with the split 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: for all $p \in P$, $f^*_p - \epsilon N \leq f_p \leq f^*_p$, where $f^*_p = \sum_{i=1}^d s(e,i) * f^*_e : p = par(e,i)$; and
2. Coverage: all items q in the lattice not included in the output have $g(q) < \lfloor \phi N \rfloor$, where $g(q)$ is defined as $g(e) = f_e : e \in S$; $g(q) = \sum_{i=1}^d s(e,i) * g(e) : q = par(e,i) \wedge e \notin P$. □

Again, given solutions whose output satisfies both accuracy and coverage, we will prefer those whose output is closer in size to that of the exact solution.

4. OFFLINE ALGORITHMS

We now give offline algorithms which make multiple passes over the input and compute the Hierarchical Heavy Hitters with the overlap rule under Definition 1. We make use of two operators.

- GENERALIZETO 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.*).
- ISAGENERALIZATIONOF takes two items, and determines whether the first is a generalization of the second. For example ISAGENERALIZATIONOF((*, 5.6.7.*),(1.2.3.4, 5.6.7.8)) is true, but the comparison is not true for the pairs ((1.2.3.4, 5.6.7.8),(*, 5.6.7.*)); ((*, 5.6.7.*),(1.2.3.4, 1.2.3.4)); or ((*, 5.6.7.*),(1.2.3.*, 5.*)).

4.1 Offline Algorithm for Overlap Rule

```
Overlap(S, $\phi$):
$L = h_1 + h_2 + \ldots + h_d$;
$hhh \leftarrow \emptyset$;
for ($l = L; l > 0; l--$) {
  forall ($label \in Level(l)$) {
    $list \leftarrow \emptyset$;
    forall ($e \in S$) {
      $p =$ GENERALIZETO($e, label$)
      if $\neg(\exists h \in hhh$ : (ISAGENERALIZATIONOF($h, e$)
        and ISAGENERALIZATIONOF($p, h$))) then {
          if ($p \in list$) {
            $f_p += f_e$; }
          else {
            $list = list \cup \{p\}$;
            $f_p = f_e$; }}}
    forall ($e \in list$) {
      if ($f_e > \lfloor \phi N \rfloor$) {
        $hhh = hhh \cup \{e\}$;
        print ($e, f_e$); }}}}
```

Figure 2: Offline Algorithm for Overlap Case

PROPOSITION 3. *The algorithm given in Figure 2 computes the set of Hierarchical Heavy Hitters defined in Definition 1. It also solves the Hierarchical Heavy Hitters problem defined in Definition 2 with $\epsilon = 0$.* □

Computational Cost. The offline algorithm given in Figure 2 makes use of a set representation. It can be implemented by keeping the set $list$ as a sequence of updates appended one after another, then sorting and aggregating $list$ before searching for HHHs. Each element in the input list corresponds to at most one item in $list$, for any $label$, hence we must sort a list of length at most N. There are $H = \prod_{i=1}^d (h_i + 1)$ distinct labels in the lattice, so the total time complexity of this implementation is the time to sort H lists of length at most N, that is, $O(HN \log N)$. The space needed is proportional to the size of the input, $O(N)$.

```
Split(S, s, $\phi$):
01 $L = h_1 + h_2 \ldots h_d$;
02 $hhh \leftarrow \emptyset$;
03 $list[h_1, h_2, \ldots h_d]$ = S;
04 for ($l = L; l > 0; l--$) {
05   for ($n \in Level(l)$) {
06     for ($e \in list[n]$) {
07       if ($f_e > \lfloor \phi N \rfloor$) {
08         $hhh = hhh \cup \{e\}$;
09         print ($e, f_e$); }
10       else {
11         for ($i = 1; i \leq d; i$++) {
12           if ($par(e,i)$ in domain) {
13             $n_p = label(par(e,i))$;
14             if ($par(e,i) \in list[n_p]$) {
15               $f_{par(e,i)} += s(e,i) * f_e$;
                 /* s(e,i) is the split function */ }
16             else {
17               $list[n_p] = list[n_p] \cup \{par(e,i)\}$;
18               $f_{par(e,i)} = s(e,i) * f_e$; }}}}}}}
```

Figure 3: Offline Algorithm for Split Case

4.2 Offline Algorithm for Split Rule

PROPOSITION 4. *The algorithm given in Figure 3 computes the set of Hierarchical Heavy Hitters defined in Definition 3. It solves the Hierarchical Heavy Hitters problem defined in Definition 4 with $\epsilon = 0$.* □

Computational Cost. The offline split algorithm can be implemented as follows: for each node in the lattice, we keep a list, initially empty. Every time we update a list (lines 15 and 17 in the algorithm of Figure 3), we simply append the item and the additional count to the list. Then, after having processed $Level(l)$, we sort each list in $Level(l-1)$, and aggregate the counts. Observe that any node in $Level(l-1)$ only receives contributions from nodes in $Level(l)$.

In this implementation, each sorted and aggregated list has at most N items in it, and each unsorted list has at most dN items in it. This follows from observing that:

1. each item in the input is represented by at most one item in each list, hence after sorting and aggregating the list, there cannot be more items than were in the input, and
2. merging lists of length N from d children can generate an unaggregated list of length at most dN.

Since there are $H = \prod_{i=1}^d (h_i + 1)$ nodes in the lattice, then the cost of this algorithm is dominated by the time to sort H lists of length at most dN, that is, $O(dHN \log(dN))$. The space requirement depends on how efficient an implementation is required: the implementation we describe above uses space $O((A+d)N)$, where A is the length of the longest anti-chain in the lattice, as before. For two dimensions then, this is $O(\min(h_1, h_2)N)$.

Note that whether we use the Overlap Rule, or any of the Split rules, in the case where we have only a single attribute to consider, then they all behave identically to the rule used in [3].

5. ONLINE ALGORITHMS

We develop *hierarchy-aware* solutions for the multi-dimensional HHH problem, under the insert-only model of data streams, also known as *cash-register* model [11], where data stream elements cannot be deleted. 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 runtime to output ϵ-approximate HHHs above this threshold.

We first introduce the *naive algorithm*, which we will compare against experimentally to show that our results are quantitatively better. At a high level, the naive 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 either method; however it will be very costly in terms of space usage. We evaluate the output on the metric of the size of the output (ie, number of items output), and we would expect this naive algorithm to do badly by this measure. Hence, we propose algorithms which keep one data structure to summarize the whole lattice, and show that they do better in terms of space and output size, both analytically and empirically. For exposition, we discuss the algorithm for the split case first since the overlap case is more involved.

5.1 Online Algorithms: Split

In this section, we consider the *split* case, where the count of an item being rolled up is split between its parents (note that the split counts can be fractions). Our algorithms maintain a summary structure T consisting of a set of tuples that correspond to samples from the input stream; initially, T is empty. Each tuple $t_e \in T$ consists of an element e from the lattice, corresponding to (collections of) elements from the data stream. Associated with each element is a bounded amount of auxiliary information used for determining lower- and upper-bounds on the total frequencies of data stream elements in the sub-lattice rooted at e.

The algorithms we present for insertion into T, compression of T, and output are conservative extensions of Strategy 2 proposed in [3] for the 1-d case. With each element e, we maintain the auxiliary information (f_e, Δ_e, m_e), where:

- f_e is a lower-bound on the (potentially fractional) total split count rolled up (directly or indirectly) into e,
- Δ_e is the difference between an upper-bound on the total split count rolled up into e and the lower-bound f_e, and
- $m_e = \max(f_{d(e)} + \Delta_{d(e)})$, over all descendants $d(e)$ of e that have been rolled up into e.

The input stream is conceptually divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$; we denote the current bucket number as $b_{current} = \lfloor \epsilon N \rfloor$. There are two alternating phases of the algorithm: insertion and compression. Intuitively, when an element e is inserted into the summary structure, its f_e count is updated if it already exists. Otherwise, the element e is (re-)created, and its (f_e, Δ_e, m_e) values suitably estimated, preserving their semantics, from e's closest ancestors in the summary structure T.

During compression, the space is reduced via merging auxiliary values and deleting elements. The algorithm scans through the tuples at the fringe of the summary structure (i.e., those tuples that do not have any descendants in the summary structure), and deletes elements whose upper bound on the total count is no larger than the current bucket number, i.e., $f_e + \Delta_e \le b_{current}$. The auxiliary values f and m of parent elements of e are also suitably updated. In doing so, previously non-fringe elements might become fringe elements and are considered iteratively. At any point, we can extract and output HHHs given a user-supplied threshold ϕ. Figure 4 presents the online algorithms for the arbitrary d-dimensional case.

PROPOSITION 5. *The algorithm given in Figure 4 computes HHHs accurately up to ϵN. The space used by the algorithm is bounded by $O((H/\epsilon)\log(\epsilon N))$.* $\square$

```
Insert(e,f):
  if $t_e$ exists then $f_e += f$;
  else {
    for ($i = 1$; $i \le d$; $i$++) {
      if ($par(e,i)$ in domain) then {
        Insert($par(e,i)$, 0); }}
    create $t_e$ with ($f_e = f$);
    $\Delta_e = m_e = b_{current} - 1$;
    for ($i = 1$; $i \le d$; $i$++) {
      if ($par(e,i)$ in domain) and ($m_{par(e,i)} < m_e$) {
        $\Delta_e = m_e = m_{par(e,i)}$; }}}

Compress:
  for each $t_e$ in fringe do {
    if ($f_e + \Delta_e \le b_{current}$) {
      for ($i = 1$; $i \le d$; $i$++) {
        if ($par(e,i)$ in domain) {
          $f_{par(e,i)} += s(e,i) * f_e$;
          /* s(e,i) is the split function */
          $m_{par(e,i)} = \max(m_{par(e,i)}, f_e + \Delta_e)$;
          if ($par(e,i)$ has no more children) {
            add $par(e,i)$ to fringe; }}}
      delete $t_e$; }}

Output($\phi$):
  let $hhhf_e = f_e$ for all $e$;
  for each $t_e$ in fringe do {
    if ($hhhf_e + \Delta_e \ge \lfloor \phi N \rfloor$) {
      print($e$, $hhhf_e$, $f_e$, $\Delta_e$); }
    else {
      for ($i = 1$; $i \le d$; $i$++) {
        if ($par(e,i)$ in domain) {
          $hhhf_{par(e,i)} += s(e,i) * f_e$;
          if ($par(e,i)$ has no more children) {
            add $par(e,i)$ to fringe; }}}}}
```

Figure 4: Online Algorithm for Split Case

5.2 Online Algorithms: Overlap

In this section, we consider the *overlap* case, where the count of an item being rolled up is given to *each* of its parents. As discussed in Section 3.3, 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, as in the split case above, would result in *overcounting* errors, which is only worsened as the number of hierarchical attributes increases. To effectively address this situation, our algorithms additionally maintain, in the 2-dimensional case, a "compensating" count g_e with each element e in the summary structure. This compensating count, as its name suggests, is used to compensate for the overcounting that results by a straightforward rolling up of counts, exploiting the *diamond* property of count propagation up the lattice structure. A "diamond" is a region of the lattice that corresponds to the inclusion-exclusion principle, to prevent such overcounting. This is illustrated in Figure 5 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 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, compres-

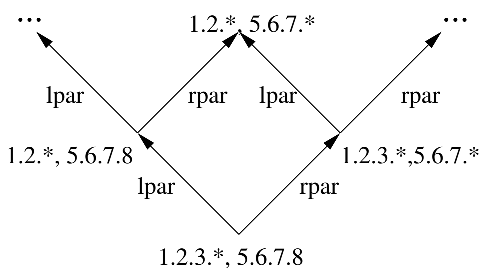


Figure 5: The diamond property: Each item has at most one "common grandparent" in the lattice.

sion of T, and output are non-trivial extensions of Strategy 2 proposed in [3] for the 1-d case, to carefully account for the problem of overcounting. With each element e, in the 2-dimensional case, we maintain the auxiliary information $(f_e, \Delta_e, g_e, m_e)$, where:

- f_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,
- g_e is an upper-bound on the total compensating count, based on counts of rolled up grandchildren of e, and
- $m_e = \max(f_{d(e)} - g_{d(e)} + \Delta_{d(e)})$, over all descendants $d(e)$ of e that have been rolled up into e.

In Figure 6, we present the online algorithms for the 2-dimensional case, to enable better understanding of the subtleties of this approach. For the 2-dimensional case, we use the notation $lpar(e) = par(e, 1), rpar(e) = par(e, 2)$ and $gpar(e) = par(par(e, 1), 2) = par(par(e, 2), 1)$, which evokes the structure of a diamond.

As in the algorithm for the split 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 the split case (see Figure 4).

During compression, the algorithm scans through the tuples at the fringe of the summary structure, and deletes elements whose upper bound on the total count is no larger than the current bucket number. The auxiliary values f and m of parent elements of e are suitably updated. To account for the possibility of overcounting, the g count of the grandparent that is shared by each of e's two parents is also updated with e's count. This strategy guarantees that $f_e - g_e$ is a lower-bound on the total count rolled up into e, after compensating for overcounting. For non-fringe elements in the summary structure, the compensating count g_e is "speculative", and hence should not be taken into account for estimating the upper-bound on the total count; this is then given by $f_e + \Delta_e$. However, for fringe elements of the summary structure, g_e is no longer speculative, and a tighter upper-bound can be obtained as $f_e - g_e + \Delta_e$. It is this tighter upper-bound that is used to determine which fringe elements to compress. As in the split case, during the compression phase, previously non-fringe elements might become fringe elements and are considered iteratively.

At any point, we can extract and output HHHs given a user-supplied threshold ϕ. The output function again needs to be sensitive to the diamond property, and the fact that counts of an HHH element should not be considered for determining the HHH-ness of its parent elements. Observe that, when two elements that share a

```
Insert($e, f$):
  if $t_e$ exists then $f_e += f$;
  else {
    if ($lpar(e)$ in domain) then Insert($lpar(e)$, 0);
    if ($rpar(e)$ in domain) then Insert($rpar(e)$, 0);
    create $t_e$ with ($f_e = f$, $g_e = 0$);
    $\Delta_e = m_e = b_{current} - 1$;
    if ($lpar(e)$ in domain) and ($m_{lpar(e)} < m_e$) {
      $\Delta_e = m_e = m_{lpar(e)}$; }
    if ($rpar(e)$ in domain) and ($m_{rpar(e)} < m_e$) {
      $\Delta_e = m_e = m_{rpar(e)}$; }}

Compress:
  for each $t_e$ in fringe do {
    if ($f_e - g_e + \Delta_e \leq b_{current}$) {
      if ($lpar(e)$ in domain) {
        $f_{lpar(e)} += f_e - g_e$;
        $m_{lpar(e)} = \max(m_{lpar(e)}, f_e - g_e + \Delta_e)$;
        if ($lpar(e)$ has no more children) {
          add $lpar(e)$ to fringe; }}
      if ($rpar(e)$ in domain) {
        $f_{rpar(e)} += f_e - g_e$;
        $m_{rpar(e)} = \max(m_{rpar(e)}, f_e - g_e + \Delta_e)$;
        if ($rpar(e)$ has no more children) {
          add $rpar(e)$ to fringe; }}
      if ($gpar(e)$ in domain) $g_{gpar(e)} += f_e - g_e$;
      delete $t_e$; }}

Output($\phi$):
  let $hhhf_e = f_e, hhhg_e = g_e$ for all $e$;
  let $lstat(e) = rstat(e) = 0$ for all $e$;
  for each $t_e$ in fringe do {
    if (($\neg lstat(e)$ or $\neg rstat(e)$) and
        ($hhhf_e - hhhg_e + \Delta_e \geq \lfloor \phi N \rfloor$)) {
      print($e$, $hhhf_e - hhhg_e$, $f_e - g_e$, $\Delta_e$);
      $lstat(e) = rstat(e) = 1$; }
    else {
      if ($lpar(e)$ in domain) and
          ($\neg lstat(e)$ or $\neg rstat(e)$) {
        $hhhf_{lpar(e)} += \max(0, hhhf_e - hhhg_e)$; }
      else if ($lpar(e)$ in domain) and
          ($lstat(e)$ and $rstat(e)$) {
        $hhhf_{lpar(e)} += \max(0, hhhf_e)$; }
      if ($lpar(e)$ in domain) {
        if ($lpar(e)$ has no more children) {
          add $lpar(e)$ to fringe with
            $lstat(lpar(e)) = lstat(e)$; }}
      if ($rpar(e)$ in domain) and
          ($\neg lstat(e)$ or $\neg rstat(e)$) {
        $hhhf_{rpar(e)} += \max(0, hhhf_e - hhhg_e)$; }
      else if ($rpar(e)$ in domain) and
          ($lstat(e)$ and $rstat(e)$) {
        $hhhf_{rpar(e)} += \max(0, hhhf_e)$; }
      if ($rpar(e)$ in domain) {
        if ($rpar(e)$ has no more children) {
          add $rpar(e)$ to fringe with
            $rstat(rpar(e)) = rstat(e)$; }}
      if ($gpar(e)$ in domain) {
        $hhhg_{gpar(e)} += \max(0, hhhf_e - hhhg_e)$; }}}
```

Figure 6: Online Algorithm for Overlap Case

parent are both HHHs, the compensating count at the parent element should not be used; doing so would result in overcompensation. In general, the compensating count at an element p may be inaccurate if, in the sublattice below it, both a left-recursive (that is, any element e for which $p = lpar(e), p = lpar(lpar(e))$, etc.) and right-recursive child (that is, any element e for which $p = rpar(e), p = rpar(rpar(e))$, etc.) have been determined HHHs. In Figure 6, $lstat(e)$ and $rstat(e)$ are used for this purpose. When such elements are encountered, their compensating counts are ignored to prevent underestimating their discounted counts, and thus satisfy the coverage constraint of correctness.

PROPOSITION 6. *The algorithm given in Figure 6 computes HHHs accurately to ϵN. The space used by the algorithm is bounded by $O((H/\epsilon)\log(\epsilon N))$.* $\square$

Higher Dimensions: The extensions to the d-dimensional case, while not straightforward, naturally build on the diamond property of the 2-dimensional case, and correspond to the generalized inclusion-exclusion principle. Instead of maintaining a single compensating count g_e, in higher dimensions, the algorithm maintains a negative compensating count g_e^- (akin to g_e in 2-dimensions), and a positive compensating count g_e^+ (which has no counterpart in 2-dimensions). When an element e is compressed, some of its ancestors at alternating levels of the lattice structure obtain negative speculative counts, while some others obtain positive speculative counts; these correspond to the negative and positive terms in the inclusion-exclusion formula, respectively. With dimensions $d \geq 3$, we use no more than two counts per element, and can get results similar to ones we have proposed here for $d = 2$. Details will be in the journal version of this paper.

6. EXPERIMENTS

In this section we investigate the goodness of the proposed online algorithms for both the "overlap" and "split" problem variants. They are evaluated according to two metrics: the size of the output generated by the algorithm, and the amount of memory used during execution. As a yardstick, we consider the size of the (exact) output from the offline algorithm.

For comparison purposes, we used the naive algorithm mentioned in Section 5 using `LossyCount` from [10], 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 four (f, Δ, m, and g), the storage ratio between these data structures is much less than two due to the overhead of storing the item identifiers.[2] Even without this, the gaps between these two algorithms will be large enough to justify the extra space per tuple, as we shall see. Hence, all plots are in terms of the number of tuples.

We used both real and synthetic data sets in the experiments. The real data consists of two-dimensional IP addresses (source and destination) from network "flow" measurements (FLOW) and packet traces (PACKET). The IP address space was viewed on the byte-level for some experiments, and on the bit-level for others.

6.1 Overlap Case

We ran the proposed online algorithm for the overlap problem, along with the naive algorithm for comparison, using a variety of parameter values (for ϕ, ϵ, and the depth of the hierarchies used). Figure 7 plots the output sizes for both algorithms as a function of timestep, with (a) $\phi = 0.2, \epsilon = 0.1$, and (b) $\phi = 0.02, \epsilon = 0.01$, where the hierarchies are induced by considering every b-bit prefix of the IP addresses. The difference in sizes is quite significant with the proposed algorithm, yielding a factor of 7 times smaller size in the former, and a factor of 13 in the latter. Compared to the exact answer (as computed by the offline algorithm at timestep 100K), the naive algorithm was 25 and 12 times larger, respectively, whereas, in both cases, the output from the proposed algorithm was slightly less than twice that of the exact. For visualization purposes, we plot the outputs at timestep 100K from the three algorithms in Figure 8, where each prefix in each dimension is mapped to a range in $[0, 2^{32} - 1]$ and drawn as a rectangle in the plane. The x- and y-axes represent the source and destination addresses, respectively. Figures 8(a) and (c) (from the exact algorithm and our online algorithm) indicate that there is a narrower distribution of source than destination addresses; this is consistent with the fact that the flows are outbound. However, the long horizontal rectangles in Figure 8(b) from the naive algorithm are misleading and indicate the opposite. Qualitatively, we see that the outputs of the offline and online algorithms are very similar, and that the online output is only a slightly larger superset of the offline output; in contrast, the output from the naive algorithm is cluttered. We believe that such plots enable network managers to visualize important features in IP traffic.

The differences between the two algorithms with the PACKET data, using the same parameter values, was even greater, as shown in Figure 9. We observed similar results for the other parameter value combinations and data sets we tried but omit them for brevity.

The proposed algorithm not only gave better answers than the naive one, but also did so using less memory. Figure 10 plots the data structure sizes as a function of timestep with the same parameter values used in Figure 7. In both cases, the proposed algorithm on average used roughly half as much memory as the naive one. These differences were still greater using the PACKET data, as shown in Figure 11.

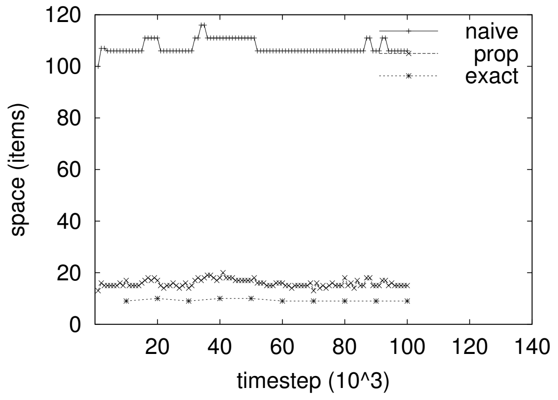
6.2 Split Case

We considered different instances of the split problem based on the hash function used. We implemented the Even Split, the Smooth Split and Random Allocate Functions; see Section 3.4 for their descriptions. Figure 12 presents plots of (a) output size; and (b) data structure size, as a function of timestep, for the Smooth Split Function. Figure 13 presents plots of (a) output size; and (b) data structure size, as a function of timestep, for the Random Allocate Function. All of these experiments were carried out using FLOW with $\phi = 0.02, \epsilon = 0.01$ and byte-level prefixes. The most significant feature of our experiments with split is that, not only was the output size of our methods the same as that of the exact algorithm, but also the items in the output were identical to those from the exact algorithm. Meanwhile, the naive algorithm used a factor of six more space for its data structures, and gave more than six times as much output than our algorithm for the split case using the Smooth Split Function. For the Random Allocate Function, we observed another factor of six difference in the size of the output, and a data structure that was four times as big.

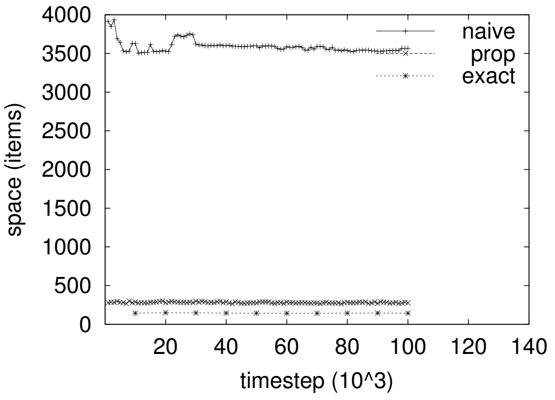
6.3 Higher Dimensions

To understand how the proposed online algorithm performs relative to the naive one in higher dimensions, for the overlap prob-

[2]For example, prefix compression can be employed in the proposed data structure.

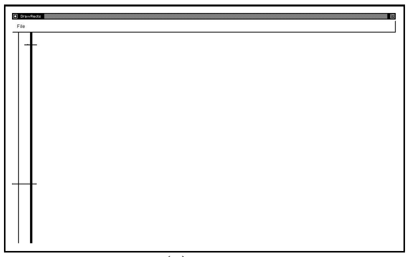


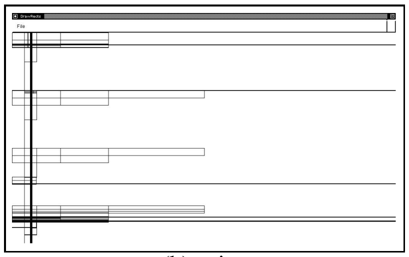
(a) $\phi = 0.2, \epsilon = 0.1$



(b) $\phi = 0.02, \epsilon = 0.01$

Figure 7: Comparison of output sizes from the online algorithms for the overlap case, and the exact answer size, using FLOW with bit-level hierarchies.


(a) exact


(b) naive

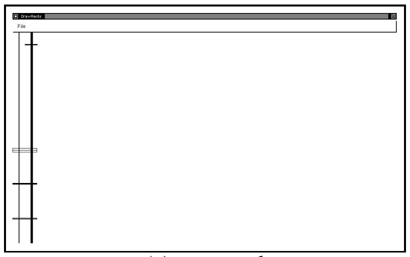

(c) proposed

Figure 8: Visualizations of sample outputs from FLOW on bit-level prefixes, with $\phi = 0.02$ at timestep 100K, from (a) exact (offline), (b) naive (online), and (c) proposed (online) algorithms.

lem, we ran a similar set of experiments on FLOW with `time` (which is hierarchical by hour, minute, second then ms) as a third attribute; the incoming stream elements do not arrive ordered on this attribute. Figure 14 shows (a) output size and (b) data structure size with $\phi = 0.02$, $\epsilon = 0.01$ on byte-level prefixes. The relative output sizes, compared to the exact answer, increased for both of the online algorithms in 3-d, but significantly more for the naive algorithm. Our proposed algorithm gave 1.5 times as many items as the exact algorithm; for FLOW with two attributes, the ratio was 1.25. (In absolute numbers, the difference in sizes went from 8 to 27.) In contrast, the naive algorithm output 6 times as many items as the exact algorithm; the ratio was 2.6 in two dimensions. (In absolute numbers, the difference went from 51 to 275.) In addition, the proportional gap between the sizes of the respective data structures widened even further by more than a factor of two. The superiority of our proposed algorithm compared to the naive one appears to grow with increasing dimension.

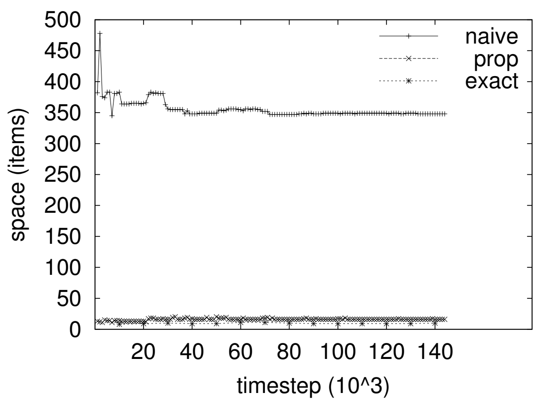
7. EXTENSIONS

Our previous work on finding HHHs in one dimension considered the model where the input consists of deletions as well as insertions of data items [3]. Typically in the applications referred to in Section 1, deletions do not occur, since we can consider the streams or warehouses as representing sequences of insertions only. We now discuss how to generalize our results to allow deletions, also known as the turnstile model [11].

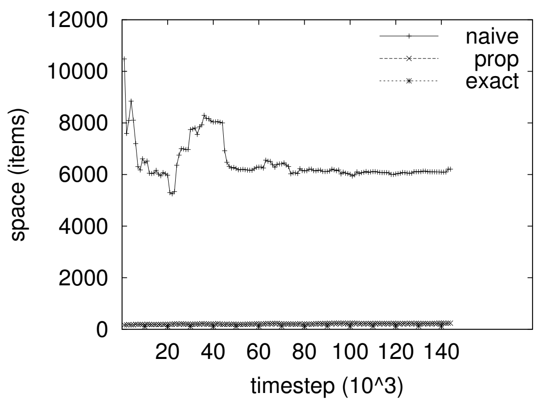
If there are very few deletions relative to the number of insertions, then by simply modifying our online algorithms to subtract from counts to simulate deletions, the results will be reasonably accurate. If there are I insertions and D deletions, then the error in the approximate counts will be in terms 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. In [3], *sketches* are used to handle the case of deletions: using compact, randomized data structures that give (probabilistic) guarantees to approximate the counts of items in the presence of insertions and deletions. A similar approach to that in [3] will apply here: we can keep sketches of items, and starting from *, descend the lattice looking for potential HHHs, then backtrack and adjust the counts as necessary.[3] 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. Since, in our motivating applications of data warehouses and data streams, explicit deletion transactions are not common, we do not discuss this extension further.

Other extensions, such as finding the HHHs in a recent time window can be done by employing our methods in the framework of [1]. It is also straightforward to adapt our methods to take selection predicates when these are given up front; it would be more

[3]This is similar to the bottom-up searching approaches in datacubes.

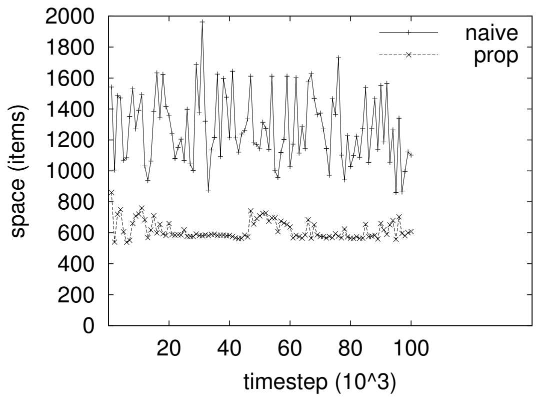


(a) $\phi = 0.2, \epsilon = 0.1$

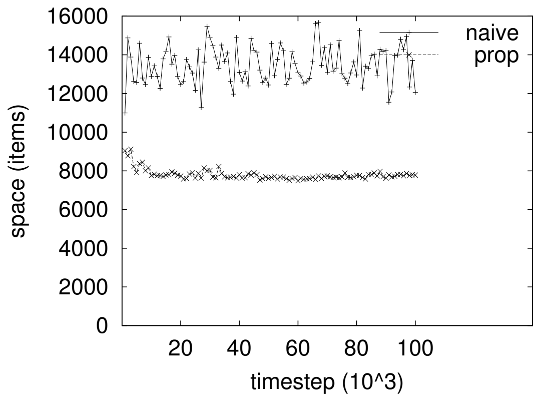


(b) $\phi = 0.02, \epsilon = 0.01$

Figure 9: Comparison of output sizes from the online algorithms for the overlap case and the exact answer size using PACKET with bit-level hierarchies.



(a) $\phi = 0.2, \epsilon = 0.2$



(b) $\phi = 0.02, \epsilon = 0.01$

Figure 10: Comparison of data structure sizes from the online algorithms for the overlap case using FLOW with bit-level hierarchies.

challenging to allow these to be specified after the data has been seen.

8. CONCLUSIONS

Finding truly multidimensional hierarchical summarization of data is of great importance in traditional data warehousing environments as well emerging data stream applications. We formalized the notion of hierarchical heavy hitters (HHHs) in its variations, and studied them in depth. In particular, we proposed online algorithms for approximately determining the HHHs to proven accuracy in only one pass using very small space regardless of the number of dimensions; in detailed experimental study, these algorithms are shown to be remarkably accurate in estimating HHHs.

9. REFERENCES

[1] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of ACM Principles of Database Systems*, 2004.

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

[3] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *International Conference on Very Large Databases*, pages 464–475, 2003.

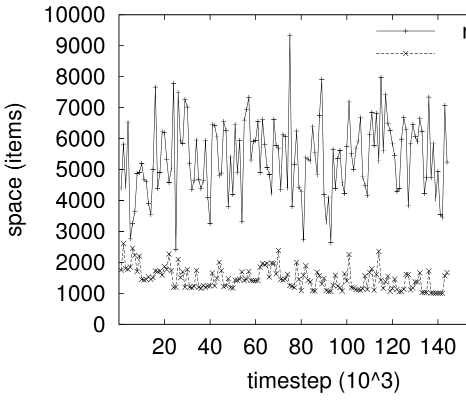
[4] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. In *Proceedings of ACM Principles of Database Systems*, pages 296–306, 2003.

[5] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proceedings of ACM SIGCOMM*, 2003.

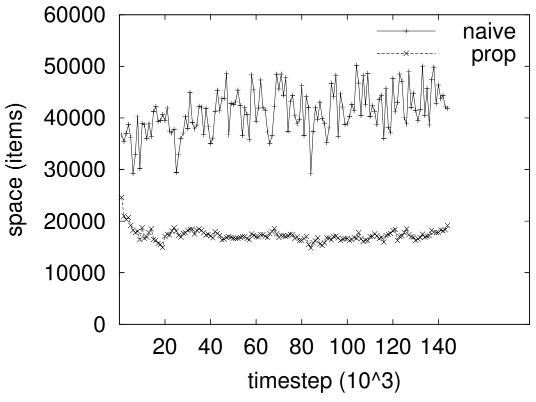
[6] S. Guha, N. Koudas, and K. Shim. Data streams and histograms. In *Proceedings of Symposium on Theory of Computing*, pages 471–475, 2001.

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

[8] L. V. S. Lakshmanan, R. T. Ng, C. X. Wang, X. Zhou, and T. Johnson. The generalized MDL approach for

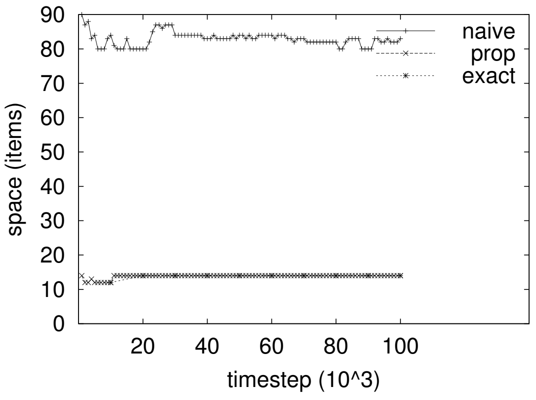


(a) $\phi = 0.2, \epsilon = 0.2$

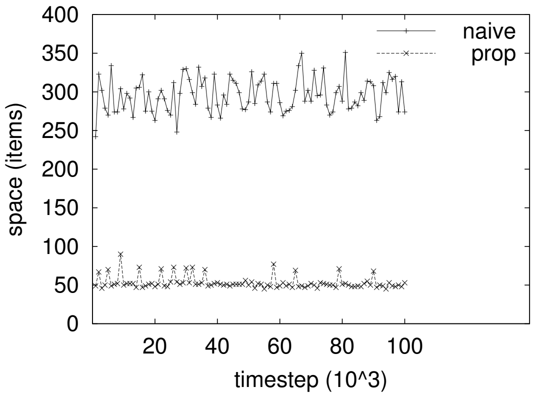


(b) $\phi = 0.02, \epsilon = 0.01$

Figure 11: Comparison of data structure sizes from the online algorithms for the overlap case using PACKET with bit-level hierarchies.



(a) output size



(b) data structure size

Figure 12: Comparison of output and data structure sizes from the online algorithms for the (smooth) split case, using FLOW with byte-level prefixes with $(\phi = 0.02, \epsilon = 0.01)$.

summarization. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 766–777, 2002.

[9] J. Lee, D. Kim, and C. Chung. Multidimensional selectivity estimation using compressed histogram information. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 205–214, 1999.

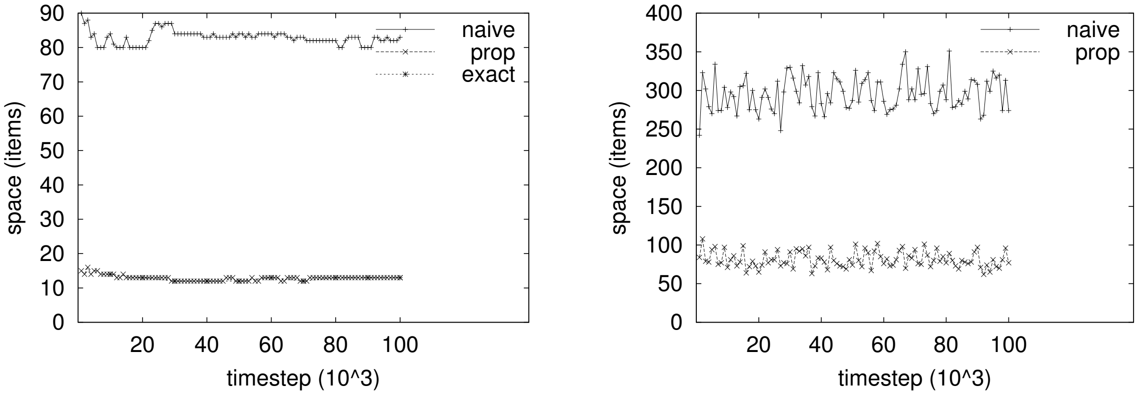
[10] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

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

[12] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *Proceedings of ICDT*, pages 236–256, 1999.

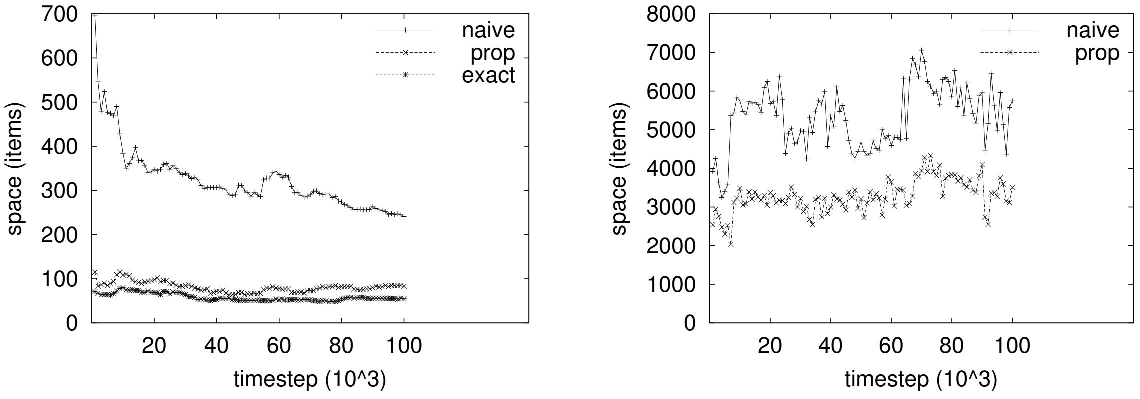
[13] N. Thaper, P. Indyk, S. Guha, and N. Koudas. Dynamic multidimensional histograms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–366, 2002.

[14] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the 7th ACM International Conferences on Information and Knowledge Management*, pages 96–104, 1998.



(a) output size (b) data structure size

Figure 13: Comparison of output and data structure sizes from the online algorithms for the 0/1 split case, using FLOW with byte-level prefixes with $(\phi = 0.02, \epsilon = 0.01)$.



(a) output size (b) data structure size

Figure 14: Comparison of output and data structure sizes from the online algorithms for the overlap problem with three hierarchical dimensions, using FLOW on byte-level prefixes with $(\phi = 0.02, \epsilon = 0.01)$.