Compact Histograms for Hierarchical Identifiers

Frederick Reiss* , Minos Garofalakis† and Joseph M. Hellerstein*
*U.C. Berkeley Department of Electrical Engineering and Computer Science and †Intel Research Berkeley

ABSTRACT

Distributed monitoring applications often involve streams of *unique identifiers* (UIDs) such as IP addresses or RFID tag IDs. An important class of query for such applications involves partitioning the UIDs into groups using a large lookup table; the query then performs aggregation over the groups. We propose using histograms to reduce bandwidth utilization in such settings, using a histogram partitioning function as a compact representation of the lookup table. We investigate methods for constructing histogram partitioning functions for lookup tables over unique identifiers that form a hierarchy of contiguous groups, as is the case with network addresses and several other types of UID. Each bucket in our histograms corresponds to a subtree of the hierarchy. We develop three novel classes of partitioning functions for this domain, which vary in their structure, construction time, and estimation accuracy.

Our approach provides several advantages over previous work. We show that optimal instances of our partitioning functions can be constructed efficiently from large lookup tables. The partitioning functions are also compact, with each partition represented by a single identifier. Finally, our algorithms support minimizing any error metric that can be expressed as a distributive aggregate; and they extend naturally to multiple hierarchical dimensions. In experiments on real-world network monitoring data, we show that our histograms provide significantly higher accuracy per bit than existing techniques.

1. INTRODUCTION

One of the most promising applications for streaming query processing is the monitoring of networks, supply chains, roadways, and other large, geographically distributed entities. A typical distributed monitoring system consists of a large number of small remote *Monitors* that stream intermediate query results to a central *Control Center*. Declarative queries can greatly simplify the task of gathering information with such systems, and stream query processing systems like Borealis [1], HiFi, [7] and Gigascope [6] aggressively target these applications with distributed implementations.

In many monitoring applications, the remote Monitors observe

VLDB '06, September 12-15, 2006, Seoul, Korea.

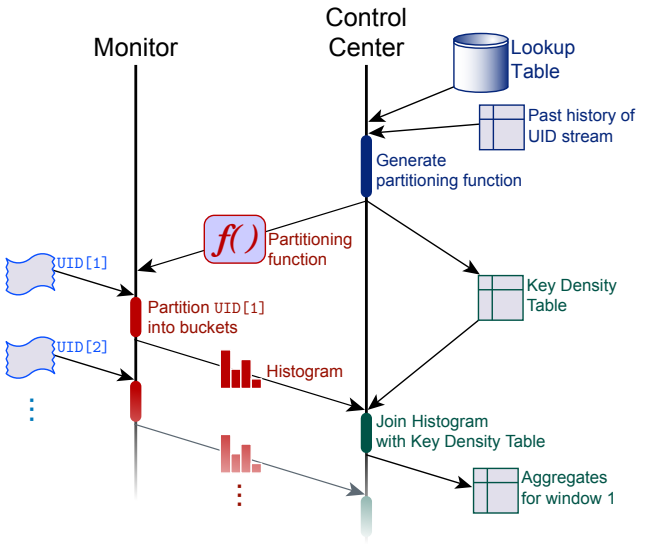

Figure 1: The communication and process sequence for decoding a stream of unique identifiers using a compact partitioning function.

streams of *unique identifiers*, such as network addresses, RFID tag IDs, or UPC symbols. An important class of queries for such applications is the *grouped windowed aggregation query*:

```
select  G.GroupId, AGG(...)
from    UIDStream U [sliding window],
        GroupTable G
where   G.uid = U.uid
group by G.GroupId;
```

where AGG is an aggregate. Such a query might produce a periodic breakdown of network traffic at each Monitor by source subnet; or a listing of frozen chickens in the supply chain by source and expiration date.

The join in this query arises because unique identifiers by themselves do not typically provide enough information to perform interesting data analysis breakdowns via GROUP BY queries. A distributed monitoring system must first "map" each unique identifier to a group that is meaningful to the application domain (e.g., a specific network subnet, a particular frozen-chicken wholesaler, etc.). Most distributed monitoring systems deployed today perform this mapping with lookup tables at the Control Center. A lookup table typically contains an entry for every unique identifier in the system. In large systems, such tables can easily grow to hundreds

of megabytes.

Of course, in order to apply the lookup table to a unique identifier, the system must have both items at the same physical location. This requirement leads to two common approaches: Send the raw streams of unique identifiers to the Control Center, or send the lookup table to the Monitors. Unfortunately, both of these approaches greatly increase the bandwidth, CPU and storage requirements of the system.

We propose an alternative approach: "compress" the lookup table into a smaller partitioning function. Using this function, the Monitors can build compact histograms that allow the Control Center to approximate the aggregation query. Figure 1 illustrates how a monitoring system would work:

1. The Control Center uses its lookup table and the past history of the UID stream to build a partitioning function[1].

2. The Monitor uses the function to partition the UIDs it receives. It keeps aggregation state for each partition and sends the resulting histogram back to the Control Center.

3. The Control Center uses the counters to reconstruct approximate aggregate values for the groups.

In order for such an approach to succeed, the partitioning function needs to have several properties. The Control Center must be able to generate the function from very large lookup tables. The function must be compact and quick to execute on a Monitor's limited resources. And, the function's output must be compact and must contain enough information that the Control Center can recover an accurate approximation of the aggregation query results.

If the lookup table contains a random mapping, then there is little that can be done to decrease its size. In many problem domains, however, the unique identifiers have an inherent structure that we can exploit to compress lookup tables into compact partitioning functions, while still answering aggregate queries with high accuracy.

In this paper, we focus on one such class of problems: The class in which in each table entry corresponds to a *subtree in a hierarchy*. The unique identifiers form the leaves of the hierarchy, and each subtree represents a contiguous range in the UID space. In general, such a hierarchy results whenever organizations assign contiguous blocks of unique identifiers to their sub-organizations. An obvious application is the monitoring of Internet traffic, where network addresses form a hierarchy of nested IP address prefixes [8]. Such hierarchies also appear in other domains, such as ISBN numbers and RFID tag IDs.

1.1 Contributions

In this paper, we introduce the problem of performing GROUP BY queries over distributed streams of unique identifiers. As a solution to this problem, we develop *histogram partitioning functions* that leverage a hierarchy to create a compact representation of a lookup table. Our partitioning functions consist of sets of *bucket nodes* drawn from the hierarchy. The subtrees rooted at these bucket nodes define the partitions.

Within this problem space, we define three classes of partitioning functions: *nonoverlapping*, *overlapping*, and *longest-prefix-match*. For the nonoverlapping and overlapping cases, we develop algorithms that find the optimal partitioning function in $O(n)$ and $O(n \log n)$ time, respectively, where n is the size of the lookup table. For the longest-prefix-match case, we develop an exact algorithm that searches over a limited subset of longest-prefix-match partitionings, requires at least cubic $\Omega(n^3)$ time, and can provide certain approximation guarantees for the final longest-prefix-match histogram. We also propose novel sub-quadratic heuristics that are shown to work well in practice for large data sets.

Unlike most previous work, our algorithms can optimize for any error metric that can be expressed as a distributive aggregate. Also, we extend our algorithms to multiple dimensions and obtain $O(dn^d \log n)$ running times for the extended algorithms, where d is the number of dimensions and n is the size of the lookup table.

Finally, we present an experimental study that compares the histograms arising from our techniques with two leading techniques from the literature. Histograms based on our overlapping and longest-prefix-match partitioning functions provide considerably better accuracy in approximating grouped aggregation queries over a real-life network monitoring data set.

1.2 Relationship to Previous Work

Histograms have a long history in the database literature. Poosala *et al.* [21] give a good overview of one-dimensional histograms, and Bruno *et al.* [3] provide an overview of existing work in multi-dimensional histograms.

Previous work has identified histogram construction problems for queries over hierarchies in data warehousing applications, where histogram buckets can be arbitrary contiguous ranges. Koudas *et al.* first presented the problem and provided an $O(n^6)$ solution [16]. Guha *et al.* developed an algorithm that obtains "near-linear" running time but requires more histogram buckets than the optimal solution [12]. Both papers focus only on *Root-Mean-Squared (RMS)* error metrics. In our work, we consider a different version of the problem in which the histogram buckets consist of nodes in the hierarchy, instead of being arbitrary ranges; and the selection ranges form a partition of the space. This restriction allows us to devise efficient optimal algorithms that extend to multiple dimensions and allow nested histogram buckets. Also, we support a wide variety of error metrics.

The STHoles work of Bruno *et al.* introduced the idea of nested histogram buckets [3]. The "holes" in STHoles histograms create a structure that is similar to our longest-prefix-match histogram buckets. However, we present efficient and optimal algorithms to build our histograms, whereas Bruno *et al.* used only heuristics (based on query feedback) for histogram construction. Our algorithms take advantage of hierarchies of identifiers, whereas the STHoles work assumed no hierarchy.

Bu *et al.* study the problem of describing 1-0 matrices using hierarchical Minimum Description Length summaries with special "holes" to handle outliers [4]. This hierarchical MDL data structure has a similar flavor to the longest-prefix-match partitioning functions we study, but there are several important distinctions. First of all, the MDL summaries construct an exact compressed version of binary data, while our partitioning functions are used to find an approximate answer over integer-valued data. Furthermore, the holes that Bu *et al.* study are strictly located in the leaf nodes of the MDL hierarchy, whereas our hierarchies involve nested holes.

Wavelet-based histograms [17, 18] are another area of related work. The error tree in a wavelet decomposition is analogous to the UID hierarchies we study. Also, recent work has studied building wavelet-based histograms for distributive error metrics [9, 15]. Our overlapping histograms are somewhat reminiscent of wavelet-based histograms, but our concept of a bucket (and its contribution to the histogramming error) is simpler than that of a Haar

[1]Past history of the UID stream is typically available from a data warehouse that is loaded from Monitors' logs on a non-real-time basis.

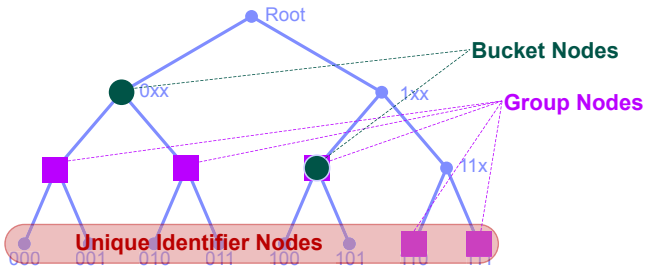

Figure 2: A 3-level binary hierarchy of unique identifiers.

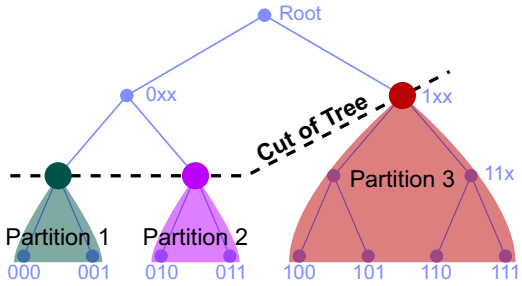

Figure 3: A partitioning function consisting of nonoverlapping subtrees. The roots of the subtrees form a cut of the main tree. In this example, the UID 010 is in Partition 2.

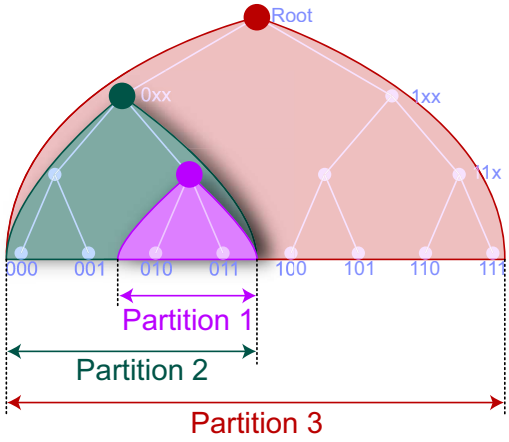

Figure 4: An overlapping partitioning function. Each unique identifier maps to the buckets of all bucket nodes above it in the hierarchy. In this example, the UID 010 is in Partitions 1, 2, and 3.

wavelet coefficient. This results in simpler and more efficient algorithms (in the case of non-RMS error metrics), especially for multi-dimensional data sets [9]. In addition, our histogramming algorithms can work over arbitrary hierarchies rather than assuming the fixed, binary hierarchical construction employed by the Haar wavelet basis.

Our longest-prefix-match class of functions is based on the technique used to map network addresses to destinations in Internet routers [8]. Networking researchers have developed highly efficient hardware and software methods for computing longest-prefix-match functions over IP addresses [20] and general strings [5].

2. PROBLEM DEFINITION

The algorithms in this paper choose optimal partitioning functions over a hierarchy of unique identifiers. In this section, we give a description of the theoretical problem that we solve in the rest of the paper. We start by specifying the classes of partitioning function that our algorithms generate. Then we describe the criteria that we use to rank partitioning functions.

Our partitioning functions operate over streams of unique identifiers (UIDs). These unique identifiers form the leaves of a hierarchy, which we call the *UID hierarchy*. Figure 2 illustrates a simple binary UID hierarchy. Our work handles arbitrary hierarchies, as we show in Section 4.1, but we limit our discussion here to binary hierarchies for ease of exposition.

As Figure 2 shows, certain nodes within the UID hierarchy will have special significance in our discussion:

- *Group nodes* (shown as squares in Figure 2) define the *groups* within the user's GROUP BY query. In particular, each group node resides at the top of a subtree of the hierarchy. The UIDs at the leaves of this subtree are the members of the group. In our problem definition, these subtrees cannot overlap.
- *Bucket nodes* (large circles in Figure 2) define the *partitions* of our partitioning functions. During query execution, each of these partitions defines a *bucket* of a histogram. The semantics of the bucket nodes vary for different classes of partitioning functions, as we discuss in the next section.

In a nutshell, our goal is to approximate many squares using just a few circles; that is, to estimate aggregates at the group nodes by instead computing aggregates for a carefully-chosen (and much smaller) collection of bucket nodes.

2.1 Classes of Partitioning Functions

The goal of our algorithms is to choose optimal histogram partitioning functions. We represent our partitioning functions with sets of bucket nodes within the hierarchy. In this paper, we study three different methods of interpreting a set of bucket nodes: *Nonoverlapping*, *Overlapping*, and *Longest-Prefix-Match*. The sections that follow define the specifics of each of these interpretations.

2.1.1 Nonoverlapping Partitioning Functions

Our simplest class of partitioning functions is for *nonoverlapping* partitionings. A nonoverlapping partitioning function divides the UID hierarchy into disjoint subtrees, as illustrated by Figure 3. We call the hierarchy nodes at the roots of these subtrees the *bucket nodes*. Note that the bucket nodes form a cut of the hierarchy. Each unique identifier maps to the bucket of its ancestor bucket node. For example, in Figure 3, the UID 010 maps to Partition 2.

Nonoverlapping partitioning functions have the advantage that they are easy to compute. In Section 3.2.2, we will present a very efficient algorithm to compute the optimal nonoverlapping partitioning function for a variety of error metrics. Compared with our other types of partitioning functions, nonoverlapping partitioning functions produce somewhat inferior histograms in our experiments. However, the speed with which these functions can be chosen makes them an attractive choice for lookup tables that change frequently.

2.1.2 Overlapping Partitioning Functions

The second class of functions we consider is the *overlapping* partitioning functions. Figure 4 shows an example of this kind of

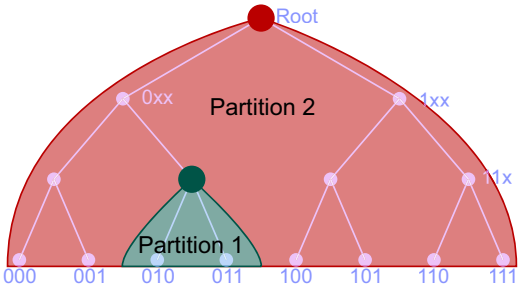

Figure 5: A longest-prefix-match partitioning function over a 3-level hierarchy. The highlighted nodes are called *bucket nodes*. Each leaf node maps to its closest ancestor's bucket. In this example, node 010 is in Partition 1.

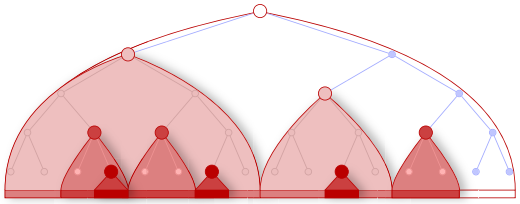

Figure 6: A more complex longest-prefix-match partitioning function, showing some of the ways that partitions can nest.

function. Like a nonoverlapping function, an overlapping partitioning function divides the UID hierarchy into subtrees. However, the subtrees in an overlapping partitioning function may overlap. As before, the root of each subtree is called a bucket node. In this case, "partitioning function" is something of a misnomer, since a unique identifier maps to the "partitions" of *all* the bucket nodes between it and the root. In the example illustrated in the diagram, the UID 010 maps to Partitions 1, 2, and 3.

Overlapping partitioning functions provide a strictly larger solution space than nonoverlapping functions. These additional solutions increase the "big O" running times of our algorithms by a logarithmic factor. This increase in running time is offset by a decrease in error. In our experiments, overlapping partitioning functions produce histograms that more efficiently represent network traffic data, compared with existing techniques.

2.1.3 *Longest-Prefix-Match Partitioning Functions*

Our final class of partitioning functions is called the *longest-prefix-match* partitioning functions. A longest-prefix-match partitioning function uses bucket nodes to define partition subtrees, as with an overlapping partitioning function. However, in the longest-prefix-match case, each UID maps only to the partition of its *closest* ancestor bucket node (selected in the histogram). Figure 5 illustrates a simple longest-prefix-match function. In this example, UID 010 maps to Partition 1. Figure 6 illustrates a more complex longest-prefix-match partitioning function. As the figure shows, partitions can be arbitrarily nested, and a given partition can have multiple "holes".

Longest-prefix-match functions are inspired by the routing tables for inter-domain routers on the Internet. These routing tables map prefixes of the IP address space to destinations, and each address is routed to the destination of the longest prefix that matches it. This routing algorithm not only reflects the inherent structure of Internet addresses, it reinforces this structure by making it efficient for an administrator to group similar hosts under a single prefix.

Longest-prefix-match partitioning has the potential to produce histograms that give very compact and accurate representations of network traffic. However, choosing an optimal longest-prefix-match partitioning function turns out to be a difficult problem. We propose an algorithm that explores a limited subset of longest-prefix-match partitionings and requires at least cubic time (while offering certain approximation guarantees for the resulting histogram), as well as two sub-quadratic heuristics that can scale to large data sets. In our experiments, longest-prefix-match partitioning functions created with these heuristics produce better histograms in practice than optimal partitioning functions from the other classes.

2.2 Measuring Optimality

Having described the classes of partitioning functions that our algorithms produce, we can now present the metric we use to measure the relative "goodness" of different partitioning functions.

2.2.1 *The Groups*

Our target monitoring applications divide unique identifiers into groups and aggregate within each group. In this paper, we focus on groups that consist of non-overlapping subtrees of the UID hierarchy. We call the root of each such subtree a *group node*. Note that, since the subtrees cannot overlap, no group node can be an ancestor of another group node.

2.2.2 *The Query*

If the groups are represented by a table of group nodes, the general form of the aggregation query we target is:

```
select   G.gid, count(*)
from     UIDStream U [sliding window],
         GroupHierarchy G
where    G.uid = U.uid
         -- GroupHierarchy places all UIDs below
         -- a group node in the same group.
group by G.node;
```

This query joins UIDStream, a stream of unique identifiers, with GroupHierarchy, a lookup table that maps every UID below a given group node to a single group ID that is unique to that group node. For ease of exposition, we consider only **count** aggregates here; the extension of our work to other SQL aggregates is straightforward.

2.2.3 *The Query Approximation*

Our algorithms generate partitioning functions for the purposes of approximating a query like the one in Section 2.2.2. The input of this approximation scheme is a window's worth of tuples from the UIDStream stream. We use the partitioning function to partition the UIDs in the window into histogram buckets, and we keep a count for each bucket. Within each bucket, we assume that the counts are uniformly distributed among the groups that map to the bucket. This uniformity assumption leads to an estimated count for each group. For overlapping partitioning functions, only the closest enclosing bucket is used to estimate the count for each group.

2.2.4 *The Error Metric*

The query approximation in the previous section produces an estimated count for each group in the original query (using the conventional uniformity assumptions for histogram buckets [21]). There are many ways to quantify the effectiveness of such an approximate answer, and different metrics are appropriate to different

applications. Our algorithms work for a general class of error metrics that we call *distributive error metrics*.

A distributive error metric is a distributive aggregate [10] $\langle \mathsf{start}, \oplus, \mathsf{finalize} \rangle$, where:

- start is a function on groups that converts the actual and estimated counts for a group into a "partial state record" (PSR);
- $\oplus$ is a function that merges the two PSRs; and,
- finalize is a function that converts a PSR into a numeric error.

In addition to being distributive, the aggregate that defines a distributive error metric must also satisfy the following "monotonicity" properties for any PSRs A, B, and C[2]:

$$\mathsf{finalize}(B) > \mathsf{finalize}(C) \rightarrow \mathsf{finalize}(A \oplus B) \geq \mathsf{finalize}(A \oplus C) \quad (1)$$

$$\mathsf{finalize}(B) = \mathsf{finalize}(C) \rightarrow \mathsf{finalize}(A \oplus B) = \mathsf{finalize}(A \oplus C) \quad (2)$$

As an example, consider the common *average error* metric:

$$\text{Error} = \frac{\sum_{g \in G} |g.\text{actual} - g.\text{approx}|}{|G|} \quad (3)$$

where G is the set of groups in the query result. We can define average error as:

$$\mathsf{start}(g) = \langle |g.\text{actual} - g.\text{approx}|, 1 \rangle \quad (4)$$

$$\langle s_1, c_1 \rangle \oplus \langle s_2, c_2 \rangle = \langle s_1 + s_2, c_1 + c_2 \rangle \quad (5)$$

$$\mathsf{finalize}(\langle s, c \rangle) = \frac{s}{c} \quad (6)$$

Note that this metric uses an intermediate representation of $\langle \text{sum}, \text{count} \rangle$ while summing across buckets. A distributive error metric can use any fixed number of counters in a PSR.

In addition to the average error metric defined above, many other useful measures of approximation error can be expressed as distributive error metrics. Some examples include:

- RMS error:

$$\text{Error} = \sqrt{\frac{\sum_{g \in G} (g.\text{actual} - g.\text{approx})^2}{|G|}} \quad (7)$$

- Average relative error:

$$\text{Error} = \frac{\sum_{g \in G} \frac{|g.\text{actual} - g.\text{approx}|}{\max(g.\text{actual}, b)}}{|G|} \quad (8)$$

  where b is a constant to prevent division by zero (typically chosen as a low-percentile actual value from historical data [9]).

- Maximum relative error:

$$\text{Error} = \max_{G \in G} \left(\frac{|g.\text{actual} - g.\text{approx}|}{\max(g.\text{actual}, b)} \right) \quad (9)$$

We use all four of these error metrics in our experiments.

3. ALGORITHMS

Having defined the histogram construction problems we solve in this paper, we now present dynamic programming algorithms for solving them. Section 3.1 gives a high-level description of our general dynamic programming approach. Then, Section 3.2 gives specific recurrences for choosing partitioning functions.

[2]These properties ensure that the principle of local optimality needed by our dynamic programs holds.

Variable	Description
U	The universe of unique identifiers.
H	The UID hierarchy, a set of nodes $h_1, h_2, \ldots, h_n$. we order nodes such that the children of h_i are h_{2i} and h_{2i+1}.
G	The group nodes; a subset of H.
b	The given budget of histogram buckets.
start	The starting function of the error aggregate (see Section 2.2.4).
$\oplus$	The function that merges error PSRs (Section 2.2.4).
finalize	The function that converts the intermediate error PSRs to a numeric error value (Section 2.2.4).
grperr(i)	The result of applying start and $\oplus$ to the groups below h_i (see Section 3.2).

Table 1: Variable names used in our equations.

3.1 High-Level Description

Our algorithms perform dynamic programming over the UID hierarchy. In our application scenario, the Control Center runs one of these algorithms periodically on data from the recent past history of the UID stream (Section 1). The results of each run parameterize the partitioning function that is then sent to the Monitors.

We expect that the number of groups, $|G|$, will be very large. To keep the running time for each batch tractable, we focus on making our algorithms efficient in terms of $|G|$.

For ease of exposition, we will assume for the time being that the hierarchy is a binary tree; later on, we will relax this assumption. For convenience, we number the nodes of the hierarchy 1 through n, such that the children of the node with index i are nodes $2i$ and $2i+1$. Node 1 is the root.

The general structure of all our algorithms is to traverse the hierarchy bottom-up, building a dynamic programming table E. Each entry in E will hold the smallest error for the subtree rooted at node i, given that B nodes in that subtree are bucket nodes. (In some of our algorithms, there will be additional parameters beyond i and B, increasing the complexity of the dynamic program.) We also annotate each entry E with the set of bucket nodes that produce the chosen solution. In the end, we will look for the solution that produces the least error at the root (for any number of buckets $\leq b$, the specified space budget for the histogram).

3.2 Recurrences

For each type of partitioning function, we will introduce a *recurrence relation* (or "recurrence") that defines the relationship between entries of the table E. In this section, we present the recurrence relations that allow us to find optimal partitioning functions using the algorithm in the previous section. We start by describing the notation we use in our equations.

3.2.1 Notation

Table 1 summarizes the variable names we use to define our recurrences. For ease of exposition, we also use the following shorthand in our equations:

- If A and B are PSRs, we say that $A < B$ if $\mathsf{finalize}(A) < \mathsf{finalize}(B)$.
- For any set of group nodes $G = \{g_1, \cdots, g_k\}$, grperr(G) denotes the result of applying the starting and transition functions of the error aggregate to G:

$$\mathsf{grperr}(G) = \mathsf{start}(g_1) \oplus \mathsf{start}(g_2) \oplus \cdots \oplus \mathsf{start}(g_k) \quad (10)$$

3.2.2 Nonoverlapping Partitioning Functions

Recall from Figure 3 that a nonoverlapping partitioning function consists of a set of nodes that form a cut of the UID hierarchy. Each node in the cut maps the UIDs in its child subtrees to a single histogram bucket.

Let $E[i,B]$ denote the minimum total error possible using B nodes to bucketize the subtree rooted at h_i. Then, we have:

$$E[i,B] = \begin{cases} \mathsf{grperr}(i) & \text{if } B = 1, \\ \min_{1 \le c \le B} \left(E[2i,c] \oplus E[2i+1, B-c] \right) & \text{otherwise} \end{cases} \tag{11}$$

where $\oplus$ represents the appropriate operation for merging errors for the error measure and $\mathsf{grperr}(i)$ denotes the result of applying the start and $\oplus$ components of the error metric to the groups below h_i.

Intuitively, this recurrence consists of a base case ($B = 1$) and a recursive case ($B > 1$). In the base case, the only possible solution is to make node node i a bucket node. For the recursive case, the algorithm considers all possible ways of dividing the current bucket budget B among the left and right subtrees of h_i, and simply selects the one resulting in the smallest error.

We observe that the algorithm does not need to consider making any node below a group node into a bucket node. So the algorithm only needs to compute entries of E for nodes that are either group nodes or their ancestors. The number of such nodes is $O(|G|)$, where G is the set of group nodes. Not counting the computation of grperr, the algorithm does at most $O(b^2)$ work for each node it touches ($O(b)$ work for each of $O(b)$ table entries), where b is the number of buckets. A binary-search optimization is possible for certain error metrics (e.g., maximum relative error), resulting in a smaller per-node cost of $O(b \log b)$.

For RMS error, we can compute all the values of $\mathsf{grperr}(i)$ in $O(|G|)$ amortized time by taking advantage of the fact that the approximate value for a group is simply the average of the actual values within, which can be computed by carrying sums and counts of actual values up the tree. So, our algorithm runs in $O(|G|b^2)$ time overall for RMS error. For other error metrics, it takes $O(|G| \log |U|)$ amortized time to compute the values of grperr, so the algorithm requires $O(|G|(b^2 + \log |U|))$ time.

3.2.3 Overlapping Partitioning Functions

In this section, we extend the recurrence of the previous section to generate overlapping partitioning functions, as illustrated in Figure 5. As the name suggests, overlapping partitioning functions allow configurations of bucket nodes in which one bucket node's subtree overlaps another's. To cover these cases of overlap, we add a third parameter, j to the table E from the previous section to create a table $E[i,B,j]$. Parameter j represents the index of the lowest ancestor of node i that has been selected as a bucket node. We add the j parameter because we need to know about the enclosing partition to decide whether to make node i a bucket node. In particular, if node i is *not* a bucket node, then the groups below node i in the hierarchy will map to node j's partition.

Similarly, we augment grperr with a second argument: $\mathsf{grperr}(i,j)$ computes the error for the groups below node i when node j is the closest enclosing bucket node. The new dynamic programming recurrence can be expressed as:

$$E[i,B,j] = \begin{cases} \mathsf{grperr}(i,j) & \text{if } B = 0, \\ \min_{0 \le c \le B} \left(E[2i,c,i] \oplus E[2i+1, B-c-1, i] \right) & \\ \quad \text{if } B \ge 1 \text{ and } i = j, & (i \text{ is a bucket node}) \\ \min_{0 \le c \le B-1} \left(E[2i,c,j] \oplus E[2i+1, B-c, j] \right) & \\ \quad \text{otherwise} & (i \text{ is not a bucket node}) \end{cases} \tag{12}$$

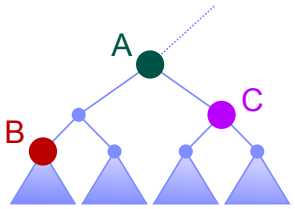

Figure 7: Illustration of the interdependence that makes choosing a longest-prefix-match partitioning function difficult. The benefit of making node B a bucket node depends on whether node A is a bucket node – and also on whether node C is a bucket node.

Intuitively, the recurrence considers all the ways to divide a budget of B buckets among node i and its left and right subtrees, given that the next bucket node up the hierarchy is node j. For the cases in which node i is a bucket node, the recurrence conditions on node i being its children's closest bucket node.

This algorithm computes $O(|G|bh)$ table entries, where h is the height of the tree, and each entry takes (at most) $O(b)$ time to compute. Assuming that the UID hierarchy forms a balanced tree, our algorithm will run in $O(|G|b^2 \log |U|)$ time.

3.2.4 Longest-Prefix-Match Partitioning Functions

Longest-prefix-match partitioning functions are similar to the overlapping partitioning functions that we discussed in the previous section. Both classes of functions consist of a set of bucket nodes that define nested partitions. The key difference is that, in a longest-prefix-match partitioning, these partitions are strictly nested, as opposed to overlapping. This renders the optimal histogram construction problem significantly harder, making it seemingly impossible to make "localized" decisions at nodes of the hierarchy.

An algorithm that finds a longest-prefix-match partitioning function must decide whether each node in the hierarchy is a bucket node. Intuitively, this choice is hard to make because it must be made for every node at once. A given partition can have several (possibly nested) subpartitions that act as "holes", removing chunks of the UID space from the parent partition. Each combination of holes produces a different amount of error both within the holes themselves and also in the parent partition.

For example, consider the example in Figure 7. Assume for the sake of argument that node A is a bucket node. Should node B also be a bucket node? This decision depends on what other nodes below A are also bucket nodes. For example, making node C a bucket node will remove C's subtree from A's partition. This choice could change the error for the groups below B, making B a more or less attractive candidate to also be a bucket node. At the same time, the decision whether to make node C a bucket node depends on whether node B is a bucket node. Indeed, the decision for each node in the subtree could depend on decisions made at every other subtree node.

In the sections that follow, we describe an exact algorithm that explores a limited subset of longest-prefix-match partitionings, by essentially restricting the number of holes in each bucket to a small constant. The resulting algorithm can offer certain approximation guarantees, but requires at least $\Omega(n^3)$ time. Since cubic running times are essentially prhibitive for the scale of data sets we consider, we also develop two sub-quadratic heuristics.

3.2.5 k-Holes Technique

We can reduce the longest-prefix-match problem's search space

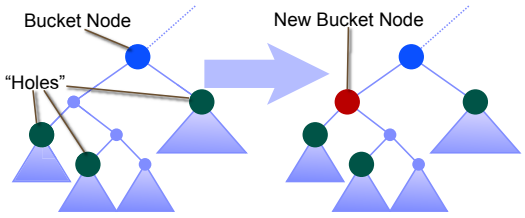

Figure 8: Illustration of the process of splitting a partition with n "holes" into smaller partitions, each of which has at most k holes, where $k < n$. In this example, a partition with 3 holes is converted into two partitions, each with two holes.

by limiting the number of holes per bucket to a constant k. This reduction yields a polynomial-time algorithm for finding longest-prefix-match partitioning functions.

We observe that, if $k \geq 2$, we can convert any longest-prefix-match partition with m holes into the union of several k-hole partitions. Figure 9 illustrates how this conversion process works for an example. In the example, adding a bucket node converts a partition with 3 holes into two partitions, each with 2 holes. Given any set of b bucket nodes, we can apply this process recursively to all the partitions to produce a new set of partitions, each of which has at most k holes. In general, this conversion adds at most $\lfloor \frac{b}{k-1} \rfloor$ additional bucket nodes to the original solution.

Consider what happens if we apply this conversion to the optimal set of b bucket nodes. If the error metric satisfies the "super-additivity" property [19]:

$$\text{Error}(P_1) + \text{Error}(P_2) \leq \text{Error}(P_1 \cup P_2) \tag{13}$$

for any partitions P_1 and P_2, the conversion will not increase the overall error. (Note that several common error metrics, e.g., RMS error, are indeed super-additive [19].) So, if the optimal b-partition solution has error E, there must exist a k-hole solution with at most $b(1 + \lfloor \frac{b}{k-1} \rfloor)$ partitions and an error of at most E.

We now give a polynomial-time dynamic programming algorithm that finds the best longest-prefix-match partitioning function with k holes in each bucket. The dynamic programming table for this algorithm is in the form:

$$E[i, B, j, H]$$

where i is the current hierarchy node, B is the number of partitions at or below node i, j is the closest ancestor bucket node, and $H = \{h_1, \ldots, h_l\}, l \leq k$ are the holes in the node j's partition.

To simplify the notation and avoid repeated computation, we use a second table $F[i, B]$ to tabulate the best error for the subtree rooted at i, *given that node i is a bucket node*.

To handle base cases, we extend grperr with an a third parameter. $\text{grperr}(i, j, H)$ computes the error for the zero-bucket solution to the subtree rooted at i, given that node j is a bucket node with the holes in H.

The recurrence for the k-holes case is similar to that of our overlapping-partitions algorithm, with the addition of the second table F, as illustrated in Figure 9. Intuitively, the first two cases of the recurrence for E are base cases, and the remaining ones are recursive cases. The first base case prunes solutions that consider impossible sets of holes. The second base case computes the error when there are no bucket nodes (and, by extension, no elements of H) below node i.

The first recursive case looks at all the ways that the bucket budget B could be divided among the left and right subtrees of node i,

$$E[i, B, j, H] = \begin{cases} \infty & \text{if } |H| > k \\ & \quad \text{or } |H \cap \text{subtree}(i)| > B \\ & \quad \text{or } \exists h_1, h_2 \in H . h_1 \in \text{subtree}(h_2), \\ \text{grperr}(i, j, H) & \text{if } B = 0, \\ \min \begin{cases} \min_{0 \leq c \leq B} (E[2i, c, j, H] \oplus E[2i+1, B-c, j, H]) \\ \qquad (i \text{ is not a bucket node}) \\ F[i, B] \text{ (only if } i \in H) \\ \qquad (i \text{ is a bucket node}) \end{cases} & \\ & \quad \text{if } B \geq 1 \end{cases}$$

$$F[i, B] = \min_{\substack{H \subseteq \text{subtree}(i) \\ 0 \leq c \leq B-1}} E[2i, c, i, H] + E[2i+1, B-c-1, i, H]$$

Figure 9: The recurrence for our k-holes algorithm.

given that node i is *not* a bucket node. The second recursive case finds the best solution for i's subtree in which node i is a bucket node with $B-1$ bucket nodes below it. Keeping the table F avoids needing to recompute the second recursive case of E for every combination of j and H.

The table E has $O(b|G|^{k+1} \log |U|)$ entries, and each entry takes $O(b)$ time to compute. Table F has $O(b|G|)$ entries, and each entry takes $O(b|G|^k)$ time to compute. The overall running time of the algorithm is $O(b^2|G|^{k+1} \log |G|)$.

Although the above algorithm runs in polynomial time, its running time (for $k \geq 2$) is at least cubic in the number of groups, making it impractical for monitoring applications with thousands of groups. In the sections that follow, we describe two heuristics for finding good longest-prefix-match partitioning functions in sub-quadratic time.

3.2.6 *Greedy Heuristic*

As noted earlier, choosing a longest-prefix-match partitioning function is hard because the choice must be made for every node at once. One way around this problem is to choose each bucket node independently of the effects of other bucket nodes. Intuitively, making a node into a bucket node creates a hole in the partition of the closest bucket node above it in the hierarchy. The best such holes tend to contain groups whose counts are very different from the counts of the rest of the groups in the parent bucket. So, if a node makes a good hole for a partition, it is likely to still be a good hole after the contents of other good holes have been removed from the partition.

Our overlapping partitioning functions are defined such that adding a hole to a partition has no effect on error for groups outside the hole. Consider the example in Figure 7. For an overlapping partitioning function, the error for B's subtree only depends on what is the closest ancestor bucket node; making C a bucket node does not change the contents of A's overlapping partition. In other words, overlapping partitioning functions explicitly codify the independence assumption in the previous paragraph. Assuming that this intuition holds, the overlapping partitioning function algorithm in Section 3.2.3 will find bucket nodes that are also good longest-prefix-match bucket nodes. Thus, our greedy algorithm simply runs the overlapping algorithm and then selects the best b buckets (in terms of bucket approximation error) from the overlapping solution. As our experiments demonstrate, this turns out to be an effective heuristic for longest-prefix-match

$$E[i,B,g,t,d] = \begin{cases} \mathsf{grperr}(i,d) & \text{if } B = 0 \\ & \quad \text{and } g = \text{number of group nodes below } i \\ & \quad \text{and } t = \text{number of tuples below } i \\ \infty & \text{if } B = 0 \\ & \quad \text{and } (t \neq \text{number of tuples below } i \\ & \qquad \text{or } g \neq \text{number of group nodes below } i) \\ \min_{b,g',t'} & \begin{cases} E[2i,b,g',t',d] \\ +E[2i+1,B-b,g-g',t-t',d] \\ \qquad \text{(Node } i \text{ is not a bucket node)} \end{cases} \\ & \begin{cases} E[2i,b,g',t',d] \\ +E[2i+1,B-b-1,g-g',t-t',d] \\ \text{if } d = \frac{t}{g} \\ \qquad \text{(Node } i \text{ is a bucket node)} \end{cases} \\ & \text{if } B \geq 1 \end{cases}$$

Figure 10: The recurrence for our pseudopolynomial algorithm.

partitionings.

3.2.7 *Quantized Heuristic*

Our second heuristic for the longest-prefix-match case is a quantized version of a pseudopolynomial algorithm. In this section, we start by describing a pseudopolynomial dynamic programming algorithm for finding longest-prefix-match partitioning functions. Then, we explain how we quantize the table entries in the algorithm to make it run in polynomial time.

Our pseudopolynomial algorithm uses a dynamic programming table $E[i,B,g,t,d]$ where:

- i is the current node of the UID hierarchy;
- B is the current bucket node budget;
- g is the number of group nodes in the subtree rooted at node i;
- t is the number of tuples whose UIDs are in the subtree rooted at node i; and,
- d, the *bucket density*, is the ratio of tuples to groups in the smallest selected ancestor bucket containing node i.

The algorithm also requires a version of grperr that takes a subtree of groups and a bucket density as arguments. This aggregate uses the density to estimate the count of each group, then compares each of these estimated counts against the group's actual count.

We can compute E by using the recurrence in Figure 10. Intuitively, the density of the enclosing partition determines the benefit of making node i into a bucket node. Our recurrence chooses the best solution for each possible density value. In this way, the recurrence accounts for every possible configuration of bucket nodes in the rest of the hierarchy. The algorithm is polynomial in the total number of tuples in the groups, but this number is itself exponential in the size of the problem.

More precisely, the recurrence will find the optimal partitioning if we let the values of g and t range from 0 to the total number of groups and tuples, respectively; with d taking on every possible value of $\frac{t}{g}$. The number of entries in the table will be $O(|G|^3T^2b)$, where T is the number of tuples in the All the base cases can be computed in $O(|G|^2T)$ amortized time, but the recursive cases each take $O(|G|Tb)$ time. So, the overall running time of this algorithm is $O(|G|^4T^3b^2)$. Note that T is exponential in the size of the problem.

We can approximate the above algorithm by considering only *quantized* values of the counters g, t and d. That is, we round the values of each counter to the closest of a set of k exponentially-distributed values $(1+\Theta)^i$. (Of course, k is logarithmic in the total "mass" of all group nodes.) The quantized algorithm creates $O(k^3b)$ table entries for each node of the hierarchy. For each table entry, the algorithm does $O(k^2b)$ work. The overall running time for the quantized algorithm is $O(k^5|G|b^2)$.

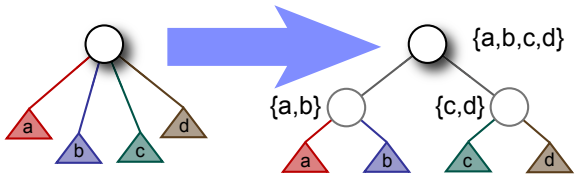

Figure 11: Diagram of the technique to extend our algorithms to arbitrary hierarchies by converting them to binary hierarchies. We label each node of the binary hierarchy with its children from the old hierarchy.

4. REFINEMENTS

Having defined our core algorithms for finding our three classes of partitioning functions, we now present useful refinements to our techniques. The first of these refinements extends our algorithms to hierarchies with arbitrary fanout. The second of these refinements focuses on choosing partitioning functions for approximating *multidimensional* GROUP BY queries. The third makes our algorithms efficient when most groups have a count of zero. Our final refinement greatly reduces the space requirements of our algorithms. All of these techniques apply to all of the algorithms we have presented thus far.

4.1 Extension to Arbitrary Hierarchies

Extending our algorithms to arbitrary hierarchies is straightforward. Conceptually, we can convert any hierarchy to a binary tree, using the technique illustrated in Figure 11. As the diagram shows, we label each node in the binary hierarchy with the set of child nodes from the original hierarchy that are below it. We can then rewrite the dynamic programming formulations in terms of these lists of nodes. For nonoverlapping buckets, the recurrence becomes:

$$E[\{i\},B] = E[\{j_1,\ldots,j_n\},B] \text{ if } j_1,\ldots,j_n \text{ were } i\text{'s children}$$

$$E[\{j_1,\ldots,j_n\},B] = \begin{cases} \mathsf{grperr}(\{j_1,\ldots,j_n\}) \text{ if } B = 1, \\ \min_{1\leq c\leq B}\left(\begin{matrix} E[\{j_1,\ldots,j_{n/2}\},c] \\ \oplus E[\{j_{n/2+1},\ldots,j_n\},B-c] \end{matrix}\right) \\ \qquad \text{otherwise} \end{cases}$$

A similar transformation converts $\mathsf{grperr}(i)$ to $\mathsf{grperr}(\{j_1,\ldots,j_n\})$. The same transformation also applies to the dynamic programming tables for the other algorithms.

The number of interior nodes in the graph is still $O(|G|)$ after the transformation, so the transformation does not increase the order-of-magnitude running time of the nonoverlapping buckets algorithm. For the overlapping and longest-prefix-match algorithms, the longest path from the root to the leaves increases by a multiplicative factor of $O(\log(\text{fanout}))$, increasing "big-O" running times by a factor of $\log^2(\text{fanout})$.

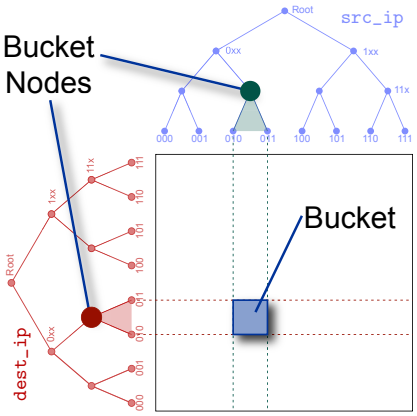

Figure 12: Diagram of a single bucket in a two-dimensional hierarchical histogram. The bucket occupies the rectangular region at the intersection of the ranges of its bucket nodes.

4.2 Extension to Multiple Dimensions

Our histograms extend naturally to multiple dimensions while still computing optimal histograms in polynomial time for a given dimensionality. In d dimensions, we define a bucket as an d-tuple of hierarchy nodes. We assume that there is a separate hierarchy for each of the d dimensions. Each bucket covers the rectangular region of space defined by the ranges of its constituent hierarchy nodes. Figure 12 illustrates a single bucket of a two-dimensional histogram built using this method. We denote the rectangular bucket region for nodes i_1 through i_d as $r(i_1,\ldots,i_d)$.

The extension of the non-overlapping buckets algorithm to d dimensions uses a dynamic programming table with entries in the form $E[(i_1,\ldots,i_d),B]$, where i_1 through i_d are nodes of the d UID hierarchies. Each entry holds the best possible error for $r(i_1,\ldots,i_d)$ using a total of B buckets. We also define a version of grperr that aggregates over the region $r(i_1,\cdots,i_d)$: $\mathsf{grperr}\,(i_1,\ldots,i_d)$

$E[(i_1,\ldots,i_d),B]$ is computed based on the entries for all subregions of $r(i_1,\ldots,i_d)$, in all combinations that add up to B buckets. For a two-dimensional binary hierarchy, the dynamic programming recurrence is shown below. Intuitively, the algorithm considers each way to split the region (i,j) in half along one dimension. For each split dimension, the algorithm considers every possible allocation of the B bucket nodes between the two halves of the region.

$$E[(i_1,i_2),B] = \begin{cases} \mathsf{grperr}(i_1,i_2) & \text{if } B=1, \\ min\begin{cases} min_{1\le c\le B} E[(i_1,2i_2),c] \oplus E[(i_1,2i_2+1),B-c] \\ min_{1\le c\le B} E[(2i_1,i_2),c] \oplus E[(2i_1+1,i_2),B-c] \end{cases} & \\ & \text{otherwise} \end{cases}$$

The extension of the overlapping buckets algorithm to multiple dimensions is similar to the extension of the nonoverlapping algorithm. We make explicit the constraint, implicit in the one-dimensional case, that every bucket region in a given solution be strictly contained inside its parent region, with no partial overlap. For the two-dimensional case, the recurrence is given in Figure 13.

Our algorithms for finding longest-prefix-match buckets can be extended to multiple dimensions by applying the same transformation. We omit the recurrences for these algorithms due to lack of space.

Unlike other classes of optimal multidimensional histograms, the multidimensional extensions of our algorithms run in polynomial time for a given dimensionality. The running time of the extended nonoverlapping algorithm is $O(|G|^d db^2)$ for RMS error, and the running time of the extended overlapping buckets algorithm is $O(db^2|G|^d \log^d |U|)$, where d is the number of dimensions. Similarly, the multidimensional version of our quantized heuristic runs in $O(db^2|G|^d)$ time.

$$E[(i_1,i_2),B,(j_1,j_2)] = \begin{cases} \mathsf{grperr}\,((i_1,i_2),(j_1,j_2)) & \text{if } B=0, \\ \min\begin{cases} \min_{0\le c\le B-1}(E[(2i_1,i_2),c,(i_1,i_2)] \oplus E[(2i_1+1,i_2),B-c-1,(i_1,i_2)]) \\ \min_{0\le c\le B-1}(E[(i_1,2i_2),c,(i_1,i_2)] \oplus E[(i_1,2i_2+1),B-c-1,(i_1,i_2)]) \\ \quad ((i_1,i_2) \text{ is a bucket region}) \\ \min_{0\le c\le B}(E[(2i_1,i_2),c,(j_1,j_2)] \oplus E[(2i_1+1,i_2),B-c,(j_1,j_2)]) \\ \min_{0\le c\le B}(E[(i_1,2i_2),c,(j_1,j_2)] \oplus E[(i_1,2i_2+1),B-c,(j_1,j_2)]) \\ \quad ((i_1,i_2) \text{ is not a bucket region}) \end{cases} & \\ & \text{otherwise} \end{cases}$$

Figure 13: Recurrence for finding overlapping partitioning functions in two dimensions.

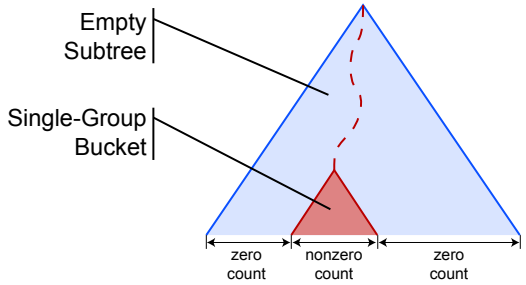

Figure 14: One of the *sparse buckets* that allow our overlapping histograms to represent sparse group counts efficiently. Such a bucket produces zero error and can be represented in $O(\log\log|U|)$ more bits than a conventional bucket.

4.3 Sparse Group Counts

For our target monitoring applications, it is often the case that the counts of most groups are zero. There is generally a very large universe of UIDs, and the number of groups tends to be very large as well. During a given time window, a given Monitor will only observe tuples from a fraction of the groups. With some straightforward optimizations, our algorithms can take advantage of cases when the group counts are sparse. These optimizations make the running time of our algorithms depend only on the height of the hierarchy and the number of nonzero groups.

To improve the performance of the nonoverlapping buckets algorithm in Section 3.2.2, we observe that the error for a subtree whose groups have zero count will always be zero. This observation means that the algorithm can ignore any subtree whose leaf nodes all have a count of zero. Furthermore, the system does not need to store any information about buckets with counts of zero, as these buckets can be easily inferred from the non-empty buckets on the fly.

For overlapping and longest-prefix-match buckets, we introduce a new class of bucket, the *sparse bucket*. A sparse bucket consists of a single-group *sub-bucket* and an empty subtree that contains it, as shown in Figure 14. As a result, the approximation error within a sparse bucket is always zero. Since the empty subtree has zero count and can be encoded as a distance up the tree from the sub-bucket, a sparse bucket takes up only $O(\log\log|U|)$ more space than a single normal bucket.

A sparse bucket dominates any other solution that places bucket nodes in its subtree. As a result, our overlapping buckets algorithm does not need to consider any such solutions when it can create a sparse bucket. Dynamic programming can start at the upper node of each sparse bucket. Since there is one sparse bucket for each nonzero group, the algorithm runs in $O(gb^2 \log |U|)$ time.

For our target monitoring applications, it is important to note that the time required to produce an approximate query answer from one of our histograms is proportional to the number of groups the histogram predicts will have nonzero count. Because of this relationship, the end-to-end running time of the system can be sensitive to how aggressively the histogram marks empty ranges of the UID space as empty. Error metrics that penalize giving a zero-count group a nonzero count will make the approximate group-by query run much more quickly.

4.4 Space Requirements

A naive implementation of our algorithms would require large in-memory tables. However, a simple technique developed by Guha [11] reduces the memory overhead of the algorithms to very manageable sizes. The basic strategy is to compute only the error and number of buckets on the left and right children at the root of the tree. Once entry $E[i, \cdots]$ has been used to compute all the entries for node $\lfloor \frac{i}{2} \rfloor$, it can be garbage-collected.

To reconstruct the entire bucket set, we apply dynamic programming recursively to the children of the root. This multi-pass approach does not change the order-of-magnitude running times of our algorithms, though it can increase the running time by a significant factor in practice. In our actual implementation, we store a set of bucket nodes along with each entry of E in memory. With the bucket nodes encoded in E, we only need one pass to recover the solution.

The number of table entries that must be kept in memory at a given time is also a function of the order in which the algorithm processes the nodes of the UID hierarchy. Our implementation processes nodes in the order of a preorder traversal, keeping the memory footprint to a minimum. To further reduce memory requirements, the nodes themselves could be stored on disk in this order and read into memory as needed.

Applying the above optimizations reduces the memory footprint of our nonoverlapping algorithm to $O(b \log |U|)$ for a balanced hierarchy. Similarly, our overlapping partitions algorithm requires $O(b \log^2 |U|)$ space. Our quantized heuristic requires $O(k^3 b \log^2 |U|)$ space, where k is the number of quanta for each counter.

5. EXPERIMENTAL EVALUATION

To measure the effectiveness of our techniques, we conducted a series of evaluations on real network monitoring data and metadata.

The WHOIS databases store ownership information on publicly accessible subnets of the Internet. Each database serves a different set of addresses, though WHOIS providers often mirror each others' entries. We downloaded publicly-available dumps of the RIPE and APNIC WHOIS databases [22, 2] and merged them, removing duplicate entries. We then used this table of subnets to generate a table of 1.1 million nonoverlapping IP address prefixes that completely cover the IP address space. Each prefix corresponds to a different subnet. The prefixes ranged in length from 3 bits (536 million addresses) to 32 bits (1 address), with the larger address ranges denoting unused portions of the IP address space. Figure 15 shows the distribution of prefix lengths.

We obtained a large trace of "dark address" traffic on a slice of the global Internet. The destinations of packets in this trace are

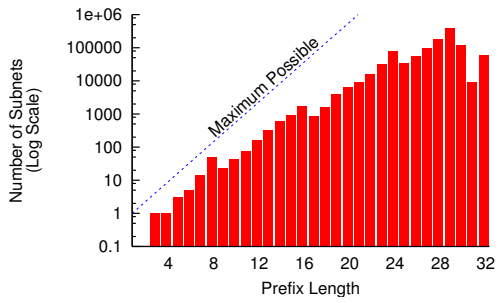

Figure 15: The distribution of IP prefix lengths in our experimental set of subnets. The dotted line indicates the number of possible IP prefixes of a given length (2^{length}). Jumps at 8, 16, and 24 bits are artifacts of an older system of subnets that used only three prefix lengths.

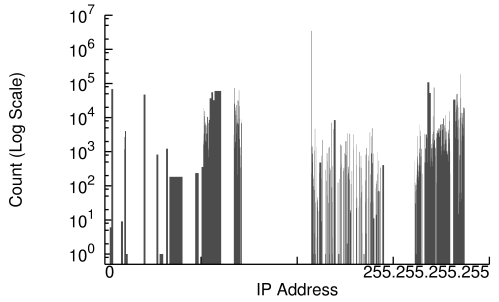

Figure 16: The distribution of network traffic in our trace by source subnet. Due to quantization effects, most ranges appear wider than they actually are. Note the logarithmic scale on the Y axis.

IP addresses that are not assigned to any active subnet. The trace contains 7 million packets from 187866 unique source addresses. Figure 16 gives a breakdown of this traffic according to the subnets in our subnet table.

We chose a query that counts the number of packets in each subnet:

```
select S.id, count(*)
from
    Packet P,
    Subnet S
where
    -- Adjacent table entries with the same subnet
    -- are merged into a single table entry
    P.src_ip ≥ I.id and P.src_ip ≤ I.id
group by S.id
```

We used six kinds of histogram to approximate the results of this query:

- Hierarchical histograms with nonoverlapping buckets
- Hierarchical histograms with overlapping buckets
- Hierarchical histograms with longest-prefix-match buckets, generated with the greedy heuristic
- Hierarchical histograms with longest-prefix-match buckets, generated with the quantized heuristic
- End-biased histograms [13]
- V-Optimal histograms [14]

An end-biased histogram consists of a set of single-group buckets for the $b-1$ groups with the highest counts and a single multi-group bucket containing the count for all remaining groups. We chose to compare against this type of histogram for several reasons. End-biased histograms are widely used in practice. Also, construction of these histograms is tractable for millions of groups, and our data set contained 1.1 million groups. Additionally, end-biased histograms model skewed distributions well, and the traffic in our data set was concentrated in a relatively small number of groups.

A V-Optimal histogram is an optimal histogram where each bucket corresponds to an arbitrary contiguous range of values. For RMS error, the V-Optimal algorithm of Jagadish *et al.* [14] can be adapted to run in $O(|G|^2)$ time, where G is the set of nonzero groups. For an arbitrary distributive error metric, the algorithm takes $O(|G|^3)$ time, making it unsuitable for the sizes of data set we considered. We therefore used RMS error to construct all the V-Optimal histograms in our study.

We studied the four different error metrics discussed in Section 2.2.4:

- Root Mean Square (RMS) error
- Average error
- Average relative error
- Maximum relative error

Note that these errors are computed across vectors of groups in the result of the grouped aggregation query, not across vectors of histogram buckets.

For each error metric, we constructed hierarchical histograms that minimize the error metric. We compared the error of the hierarchical histograms with that of an end-biased histogram using the same number of buckets. We repeated the experiment at histogram sizes ranging from 10 to 20 buckets in increments of 1 and from 20 to 1000 buckets in increments of 10.

5.1 Experimental Results

We divide our experiment results according to the type of error metric used. For each error metric, we give a graph of query result estimation error as a function of the number of histogram buckets. The dynamic range of this error can be as much as two orders of magnitude, so the y axes of our graphs have logarithmic scales.

5.1.1 RMS Error

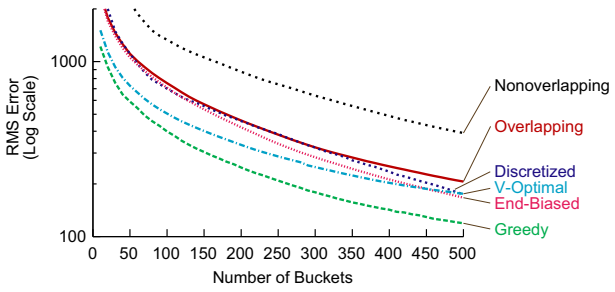

Figure 17: RMS error in estimating the results of our query with the different histogram types.

Our first experiment measured RMS error. The RMS error formula emphasizes larger deviations, making it sensitive to the accuracy of the groups with the highest counts. Longest-prefix-match histograms produced with the greedy heuristic were the clear winner, by virtue of their ability to isolate these "outlier" groups inside nested partitions. Interestingly, the quantized heuristic fared relatively poorly in this experiment, finishing at the middle of the pack. The heuristic's logarithmically-distributed counters were unable to capture sufficiently fine-grained information to produce more accurate results than the greedy heuristic.

5.1.2 Average Error

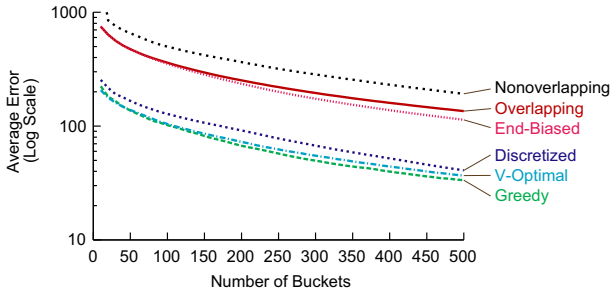

Figure 18: Average error in estimating the results of our query with the different histogram types.

Our second experiment used average error as an error metric. Figure 18 shows the results of this experiment. As with RMS error, the greedy heuristic produced the lowest error, but the V-Optimal histograms and the quantized heuristic produced results that were almost as good. Average error puts less emphasis on groups with very high counts The other types of histogram produced significantly higher error. As before, we believe this performance difference is mainly due to the ability of longest-prefix-match and V-Optimal histograms to isolate outliers by putting them into separate buckets.

5.1.3 Average Relative Error

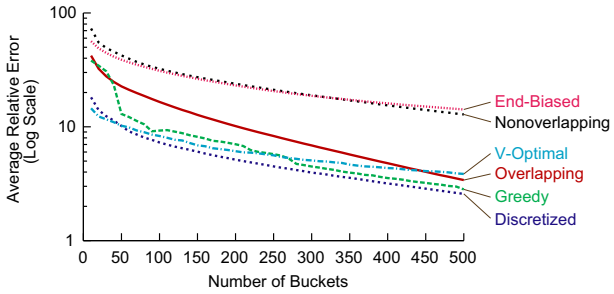

Figure 19: Average relative error in estimating the results of our query with the different histogram types. Longest-prefix-match histograms significantly outperformed the other two histogram types.

Our third experiment compared the three histogram types using average relative error as an error metric. Compared with the previous two metrics, relative error emphasizes errors on the groups with smaller counts. Figure 19 shows the results of this experiment. The quantized heuristic produced the best histograms for this error metric. The heuristic's quantized counters were better at tracking low-count groups than they were at tracking the larger groups that dominated the other experiments. V-Optimal histograms produced low error at smaller bucket counts, but fell behind as the number of buckets increased.

5.1.4 Maximum Relative Error

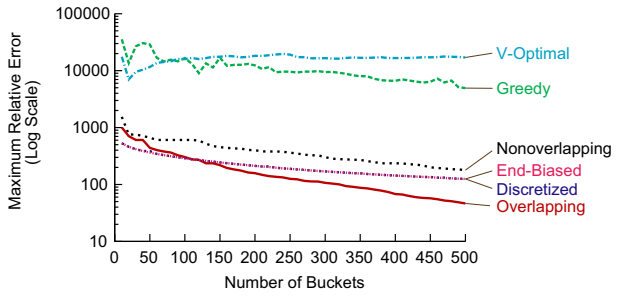

Figure 20: Maximum relative error in estimating the results of our query with the different histogram types.

Our final experiment used maximum relative error. This error metric measures the ability of a histogram to produce low error for every group at once. Results are shown in Figure 20. Histograms with overlapping partitioning functions produced the lowest result error for this error measure. Interestingly, the greedy heuristic was unable to find good longest-prefix-match partitioning functions for the maximum relative error measure. Intuitively, the heuristic assumes that removing a hole from a partition has no effect on the mean count of the partition. Most of the time, this assumption is true; however, when it is false, the resulting histogram can have a large error in estimating the counts of certain groups. Since the maximum relative error metric finds the maximum error over the entire set of groups, a bad choice anywhere in the UID hierarchy will corrupt the entire partitioning function.

6. CONCLUSION

In this paper, we motivate a new class of hierarchical histograms based on our experience with a typical operation in distributed stream monitoring. Our new histograms are quick to compute, and in our experiments on Internet traffic data they provide significantly better accuracy than prior techniques across a broad range of error metrics. In particular, we show that a simple greedy heuristic for constructing longest-prefix-match histograms produces excellent results for most error metrics, while our optimal overlapping histograms excel for minimizing maximum relative error. In addition to our basic techniques, we also provide a set of natural extensions to our basic histograms that accomodate multiple dimensions, arbitrary hierarchies, and sparse data distributions.

Our work raises some interesting open questions for further investigation. On the algorithmic side, the complexity of the optimal longest-prefix-match histogram remains to be resolved. On the more practical side, we are pursuing two thrusts. First, we are currently deploying our algorithms in a live network monitoring environment, which will raise practical challenges in terms of when and how to recalibrate the histograms based on the history of the UID stream. Second, we conjecture that these techniques are useful in a broad range of applications. We have conducted early experiments on several data sets, and preliminary results indicate that our hierarchical histograms provide better accuracy than existing techniques, even when dealing with data that lacks an inherent hierarchy.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, , and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] APNIC. Whois database, Oct. 2005. ftp://ftp.apnic.net/apnic/whois-data/APNIC/apnic.RPSL.db.gz.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a multidimensional workload-aware histogram. *SIGMOD Record*, 30(2):211–222, 2001.

[4] S. Bu, L. V. Lakshmanan, and R. T. Ng. Mdl summarization with holes. In *VLDB*, 2005.

[5] A. Buchsbaum, G. Fowler, B. Krishnamurthy, K. Vo, and J. Wang. Fast prefix matching of bounded strings. In *ALENEX*, 2003.

[6] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: high performance network monitoring with an sql interface. In *SIGMOD*, 2002.

[7] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *CIDR*, 2005.

[8] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, Sept. 1993. ftp://ftp.internic.net/rfc/rfc1519.txt.

[9] M. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *PODS*, 2004.

[10] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[11] S. Guha. Space efficiency in synopsis construction algorithms. In *VLDB*, 2005.

[12] S. Guha, N. Koudas, and D. Srivastava. Fast algorithms for hierarchical range histogram construction. In *PODS*, 2002.

[13] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, 1995.

[14] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[15] P. Karras and N. Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *VLDB*, 2005.

[16] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries (extended abstract). In *PODS*, pages 196–204, 2000.

[17] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, 1998.

[18] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *VLDB*, 2000.

[19] S. Muthukrishnan, V. Poosala, and T. Suel. "On Rectangular Partitionings in Two Dimensions: Algorithms, Complexity, and Applications". In *ICDT*, Jerusalem, Israel, Jan. 1999.

[20] P. Newman, G. Minshall, T. Lyon, L. Huston, and Ipsilon Networks Inc. Ip switching and gigabit routers. *IEEE Communications Magazine*, 1997.

[21] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, 1996.

[22] RIPE. Whois database, Sept. 2005. ftp://ftp.ripe.net/ripe/dbase/ripe.db.gz.