Space- and Time-Efficient Deterministic Algorithms for Biased Quantiles over Data Streams

Graham Cormode
Bell Laboratories
cormode@lucent.com

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

Skew is prevalent in data streams, and should be taken into account by algorithms that analyze the data. The problem of finding "biased quantiles"— that is, approximate quantiles which must be more accurate for more extreme values — is a framework for summarizing such skewed data on data streams. We present the first deterministic algorithms for answering biased quantiles queries accurately with small—sublinear in the input size— space and time bounds in one pass. The space bound is near-optimal, and the amortized update cost is close to constant, making it practical for handling high speed network data streams. We not only demonstrate theoretical properties of the algorithm, but also show it uses less space than existing methods in many practical settings, and is fast to maintain.

Keywords

Data Stream Algorithms, Biased Quantiles

General Terms

Algorithms, Performance

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; F.2 [**Theory**]: Analysis of Algorithms

1. INTRODUCTION

Many queries over large data sets require non-uniform responses. Consider published lists of wealth distributions: one typically sees details of the median income, the 75th percentile, 90th, 95th and 99th percentiles, and a list of the 500 top earners. While the detail around the center of the distribution is quite sparse, at one end of the distribution we see increasingly fine gradations in the accuracy of the response, ultimately down to the level of individuals. Similar variations in the level of accuracy required are seen in analyzing massive data streams: for example, in monitoring performance in packet networks, the distribution of round trip times is used to analyze the quality of service. Again, it is important to know broad information about the center of the distribution (median, quartiles), and increasingly precise information about the tail of the distribution, since this is most indicative of unusual or anomalous behavior in the network. A particular challenge is how to maintain such distributional information in a data streaming scenario, where massive amounts of data are seen at very high speeds: the requirement is to summarize this huge volume of data into small space to be able to accurately capture the key features of the distribution, within allowable approximation bounds.

This problem of capturing the distribution was formalized as the *biased quantiles* problem [3]. Much of the prior work studied the problem of summarizing the distribution with *uniform accuracy*: this means reporting the median with the same accuracy as the 99th percentile [7, 4]. For the applications we have outlined, such guarantees are insufficient since, for the more extreme values, the uncertainty can be too large: with a 1% accuracy, all values above the 99th percentile are indistinguishable. The *(fully) biased accuracy guarantee* demands that the accuracy scales with the place in the distribution that is queried: for example, the accuracy with which the 90th percentile is given should be five times more accurate than the median, and the 99th percentile should be ten times more accurate than the 90th percentile.

There are many variations of this problem that we address in this paper. An essentially equivalent problem is to report the *rank* of a given item: the rank is the item's position in the sorted input. Quantile queries can be used to answer rank queries, and vice-versa. The *(partially) biased* accuracy guarantee gives a cut-off for the tightest accuracy that is required: this can be leveraged to reduce further the resources needed by the summary. The *targeted* quantiles problem is when the quantiles that are required are given in advance. Focusing on answering just these queries allows the resources to be reduced further still. Throughout, our focus is on what are the fundamental costs required to support these queries, in terms of

(a) what is the *space* required to summarize the whole distribution and allow these queries to be answered with the requisite accuracy?

(b) what is the *time per update* required to maintain this summary as new data arrives in the stream?

In order to handle the large data streams prevalent in networks and other monitoring scenarios, it is vital to keep these as small as possible, depending either sublinearly or not at all on the size of the data stream.

Our Contributions. In this paper, our contributions are as follows:
• We present the first deterministic algorithms for biased quantiles, rank queries and targeted quantile queries that have guaranteed space bounds that are strictly sublinear in the size of the data stream, N, and the universe from which the items are drawn, U. In particular, we present a deterministic algorithm that takes only space $O(\frac{\log U}{\epsilon}\log(\epsilon N))$ to provide an ϵ approximation on the bi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS'06, June 26–28, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-318-2/06/0003 ...$5.00.

ased quantiles of different kinds. In contrast, previously known deterministic algorithms used $\Omega(N)$ space and previously known randomized algorithms need $\Omega(\frac{1}{\epsilon^2}\log(\epsilon N))$ worst case space which is greater than ours for even moderate ϵ. The space requirements of our algorithms are close to optimal: they are within $O(\log U)$ factors of the lower bounds. Our algorithms are also fast, with amortized time per update that is close to constant, and independent of the size of the data stream.

• We present an experimental study showing that not only do we have very strong guarantees on accuracy, time cost and space required, but that our algorithms are also competitive in comparison with existing methods, with consistently higher throughput and less space than prior algorithms for these problems in most practical settings.

1.1 Problem Definitions

The input consists of a stream of items in the range $\{1 \dots U\} = [U]$, and the input stream can be thought of as defining a multiset of items from $[U]$. At any point, the number of points observed thus far is denoted by N. The *rank* of an item x from the domain is the number of items from the input which are less than x. We denote this by $\mathrm{rank}(x)$. A basic problem is to find $\mathrm{rank}(x)$ given $x \in [U]$. Trivially, $\mathrm{rank}(x)$ can be computed by storing the whole input in sorted order. However, our focus is on situations when the size of the input, N, is so large that we cannot afford this much space to store and sort the input, and so we must use smaller–sublinear—space (consequently, we need to allow approximate answers to rank and quantile queries). Several different styles of approximation guarantee are possible:

DEFINITION 1. *(a) A* uniform rank query *is, given x to return an approximation $\hat{r}(x)$ of* $\mathrm{rank}(x)$ *such that (for an accuracy parameter ϵ, supplied in advance)*

$$\mathrm{rank}(x) - \epsilon N \leq \hat{r}(x) \leq \mathrm{rank}(x) + \epsilon N$$

(b) A fully biased rank query *is, given x to return an approximation $\hat{r}(x)$ of* $\mathrm{rank}(x)$ *such that*

$$(1-\epsilon)\,\mathrm{rank}(x) \leq \hat{r}(x) \leq (1+\epsilon)\,\mathrm{rank}(x)$$

(c) Let $t(x, N) = \max(\epsilon\,\mathrm{rank}(x), \epsilon_{\min} N)$ for parameters ϵ and $\epsilon_{\min}$. A (partially) biased rank query *is, given x to return an approximation $\hat{r}(x)$ of* $\mathrm{rank}(x)$ *such that*

$$\mathrm{rank}(x) - t(x,N) \leq \hat{r}(x) \leq \mathrm{rank}(x) + t(x,N)$$

Note that in all cases, the value being approximated is the same and what differs is the nature of the approximation guarantee required. The use of bias in accuracy allows us to give sharper results for the tails of the distribution, which typically are skewed, and are of more interest in data stream applications.[1] For $\epsilon = \epsilon_{\min} = 0$, the notions converge, but for non-zero approximation guarantees, the fully biased rank query has stricter requirements than the uniform rank query. The biased rank query is a compromise between the fully biased and uniform versions. In general, we require *relative* error in response to our queries, which is given by the $\epsilon\,\mathrm{rank}(x)$ component. However, giving such guarantees can lead to high cost, and there is a certain minimum accuracy beyond which it is not important to give finer accuracy. This is the $\epsilon_{\min} N$ component of the guarantee (a special case is then when $\epsilon_{\min} \leq 1/N$, i.e. when the $\epsilon\,\mathrm{rank}(x)$ term always dominates, in which case we have the fully biased case). From these primitives, we can define related quantile queries.

[1] We focus on the case when the finer accuracy is needed on items with low rank, referred to as the low-biased case in [3]. Results for the high-biased case follow by reversing the ordering relation, but for simplicity of presentation we focus only on the low-biased case.

DEFINITION 2 (UNIFORM QUANTILES PROBLEM). *A* uniform quantile query *is, given ϕ, to return x so that*

$$\mathrm{rank}(x) - \epsilon N \leq \phi N \leq \mathrm{rank}(x+1) + \epsilon N$$

For example, finding the median corresponds to querying for the $\phi = \frac{1}{2}$ quantile. Observe that, given a solution to answering uniform rank queries, we can answer uniform quantile queries by (binary) searching for x that satisfies the above inequalities. These two problems have been extensively studied on streams of values, and solutions are known with space guarantees in terms of ϵ: the space required is bounded by $O(\frac{1}{\epsilon}\log \epsilon N)$ [4], and $O(\frac{\log U}{\epsilon})$ [9].

DEFINITION 3 (BIASED QUANTILES PROBLEM). *Let $t(x, N) = \max(\epsilon\,\mathrm{rank}(x), \epsilon_{\min} N)$. A* biased quantile query *is, given ϕ, to return x so that*

$$\mathrm{rank}(x) - t(x,N) \leq \phi N \leq \mathrm{rank}(x+1) + t(x+1,N)$$

This problem was introduced explicitly in [3] (but inherent in prior work such as [5]), where algorithms with good space usage in practice were demonstrated. As with the uniform case, given a data structure for the biased rank query problem, we can answer biased quantile queries by searching for an item whose rank satisfies the query. Lastly, the *targeted quantiles* problem is defined as follows:

DEFINITION 4 (TARGETED QUANTILES PROBLEM [3]). *The parameter is a set of tuples $T = \{(\phi_j, \epsilon_j)\}$. Following a stream of input values, the goal is to return a set of $|T|$ values v_j such that*

$$\mathrm{rank}(v_j) - \epsilon_j N \leq \phi_j N \leq \mathrm{rank}(v_j+1) + \epsilon_j N$$

The targeted quantiles problem was defined in [3]; it can also be thought of as a generalization of the "extreme values" quantile finding problem in [7].

2. RELATED WORK

In recent years there has been significant interest in the area of data streams, where the space available for processing is considerably smaller than the input, which is presented in a "one-pass" fashion [1, 8]. For the problem of tracking quantiles in data streams, the most relevant work is the "GK algorithm" due to Greenwald and Khanna [4]. It is a deterministic algorithm which allows uniform quantile queries to be answered with error at most ϵN, using space $O(\frac{\log \epsilon N}{\epsilon})$. This improved a series of previous results of deterministic and randomized algorithms (see, eg. [7]). Running time for this algorithm is not analyzed in the paper, but since it maintains a list of items in sorted order, and inserts a new item into this list for every update, the algorithm can be implemented with amortized time cost $O(\log(\frac{1}{\epsilon}) + \log\log \epsilon N)$. Another algorithm for the problem was given in [9], whose space cost is $O(\frac{\log U}{\epsilon})$, where U is the number of distinct values possible; this is not directly comparable with the GK algorithm but also uses small space in practice. The amortized time cost is $O(\log \frac{1}{\epsilon} + \log\log U)$.

The problem of biased quantiles was formally introduced in [3]. An algorithm for biased quantiles based on GK was given, and shown to be effective on real data. However, for carefully crafted input data, the space used by the algorithm can grow linearly with the input size [10]. In contrast, we show a new algorithm here whose space cost grows at most logarithmically with the size of the input, and which often uses less space in practice than the previous algorithm. Although not called biased quantiles as such, Gupta and Zane [5] studied this problem in the context of counting inversions in streams. They gave a randomized algorithm whose space cost is $O(\frac{1}{\epsilon^3}\log^2 \epsilon N)$. For small values of ϵ, this cost rapidly becomes too high in practice. Manku *et al.* [7] gave randomized algorithms for a single targeted quantile in space $O(\frac{\phi}{\epsilon}\log\frac{1}{\delta})$.

Most recently Zhang *et al* [10] gave a randomized algorithm for the biased case based on sampling at different rates, with space cost $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log(\epsilon^2 N))$. Here, δ is the probability that each query fails to meet its error bounds. A modified version of the algorithm has average space cost which is $\Omega(\frac{1}{\epsilon}\log^2 \epsilon N)$, but worst case cost as before. The running time per update of the first algorithm is $O(\log(\epsilon N)\log(\epsilon\log(\frac{1}{\delta})))$, and the second is $\Omega(\frac{1}{\epsilon^2})$.

3. OUR ALGORITHMS

In this section we show that the biased rank and biased quantile problems can be solved with space strictly sublinear (in fact, logarithmic) in N and U. The algorithm uses a similar approach to that of [9], in that it places a binary tree structure over the domain and maintains counts associated with certain nodes in the tree. But maintaining this structure, the invariants that apply, and proof of correctness require significantly new approaches and insights.

Throughout, we make use of standard dictionary data structures [2] that support the following operations: (a) insert an item; (b) delete an item; (c) test for the presence of an item; and (d) list all items. We will measure the number of per item operations. In our experiments we will use a hash table to implement this dictionary data structure in constant (expected) time per item.

We impose a binary tree of height $\log U$ over the domain $[1\ldots U]$ in the obvious manner. For any node v in the tree, let $\mathrm{lf}(v)$ denote the leftmost leaf item in the subtree of v. Let $\mathrm{par}(v)$ denote the (unique) parent node of v, let $\mathrm{left}(v)$ denote the left child of v, and $\mathrm{right}(v)$ the right child of v.[2] Lastly, define $\mathrm{anc}(v)$ as the set of nodes that are ancestors of v in the tree.

A *bq-summary* is a subset of nodes of this tree, with associated counts, corresponding to a count of items appearing in the range covered by the node. We represent the bq-summary as a set of nodes *bq*, and for each node $v \in bq$ we also store a count c_v for that node. This count represents items from the input that were drawn from the leaves of the subtree of the node v. If node v is not stored in the bq-summary, we use $c_v = 0$ when c_v is queried.

3.1 Accuracy Guarantees

We first describe our algorithm for the fully biased version of the problem. We will prove space bounds for this version, then give the results for biased rank queries when $\epsilon_{\min}$ is used. Firstly, we define two functions over the tree, which we denote L (for Left-count) and A (for Ancestor-count).

DEFINITION 5. *We define two functions: $L(v)$, a function over tree nodes, and $A(x)$, a function over universe items:*

$$L(v) = \sum_{\mathrm{lf}(w)<\mathrm{lf}(v)} c_w \quad \text{and} \quad A(x) = \sum_{w\in\mathrm{anc}(x)} c_w$$

By maintaining certain properties on the counts, we will ensure that the uncertainty in our query answers is bounded, and at the same time the space required is also bounded. We can now define a set of formal correctness criteria for the bq-summary to give guarantees for finding biased quantiles. We use a parameter $\alpha < 1$, that we will set later based on the analysis. To guarantee queries can be answered correctly, we maintain two invariants at all times:

$$\forall x \in [U]: \quad L(x) - A(x) \le \mathrm{rank}(x) \le L(x) \qquad (1)$$
$$\forall v \in bq: \quad v \ne \mathrm{lf}(v) \Rightarrow c_v \le \alpha L(v) \qquad (2)$$

Given such a data structure, we can answer biased rank queries. Given a particular value x, its rank is at least the sum of all counts of nodes that are strictly to the left of the leaf in the tree corresponding to x. Its rank is at most this quantity plus the nodes that are ancestors of the leaf. We bound the first of these quantities by $L(x) - A(x)$, and the second by $L(x)$. Therefore, the uncertainty in the answer to the query is bounded by the sum of counts of all ancestors the queried value ($A(x)$).

[2] All of these functions can be computed in constant time under a reasonable machine model and appropriate representations of nodes v.

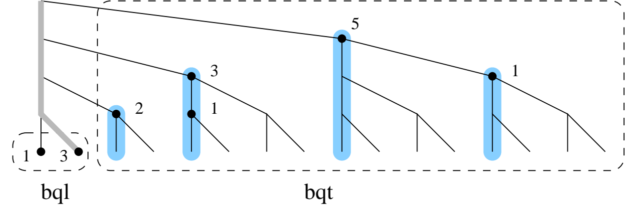


Figure 1: One dimensional data structure run on the input $\{1,2,2,2,3,4,5,6,6,6,10,12,14,14,15,16\}$ with $\alpha = \frac{1}{2}$.

LEMMA 1. *Any bq-summary that obeys* (1) *and* (2) *with* $\alpha\log(U) \le \epsilon \le \frac{1}{2}$ *allows fully biased rank queries on x to be answered with $\hat{r}(x)$ so that $|\hat{r}(x) - \mathrm{rank}(x)| \le \epsilon\,\mathrm{rank}(x)$.*

PROOF. We compute and output $\hat{r} = L(x) - \frac{1}{2}A(x)$. Applying invariant (1), we know that

$$-\tfrac{1}{2}A(x) \le (\hat{r}(x) - \mathrm{rank}(x)) \le \tfrac{1}{2}A(x).$$

So the only uncertainty comes from those nodes that are ancestors of the x in the tree. For each $w \in \mathrm{anc}(x)$, $\mathrm{lf}(w) \le \mathrm{lf}(x)$ and so $L(w) \le L(x)$. Thus, using (2) to obtain a bound on $A(x)$ in terms of $\mathrm{rank}(x)$ and $A(x)$, and rearranging to eliminate $A(x)$:

$$\begin{aligned} |\hat{r}(x) - \mathrm{rank}(x)| &\le \tfrac{1}{2}A(x) \le \tfrac{1}{2}\textstyle\sum_{w\in\mathrm{anc}(x)} \alpha L(w) \\ &\le \tfrac{1}{2}\textstyle\sum_{w\in\mathrm{anc}(x)} \alpha L(x) \\ &\le \log(U)\tfrac{\alpha}{2}(\mathrm{rank}(x) + A(x)) \\ &\le \log(U)\tfrac{\alpha}{2(1-\alpha\log(U))}\,\mathrm{rank}(x) \\ &\le \alpha\log(U)\,\mathrm{rank}(x) \le \epsilon\,\mathrm{rank}(x) \end{aligned}$$

The last step makes use of the fact that $\alpha \le \frac{\epsilon}{\log U}$ and $\epsilon \le \frac{1}{2}$. □

LEMMA 2. *Any bq-summary that obeys* (1) *and* (2) *with* $\alpha\log(U) \le \epsilon \le \frac{1}{2}$ *allows fully biased quantile queries on ϕ to be answered with x so that* $(1-\epsilon)\,\mathrm{rank}(x) \le \phi N \le (1+\epsilon)\,\mathrm{rank}(x+1)$.

PROOF. We perform a binary search over $[U]$ for the greatest x such that $\hat{r}(x) \le \phi N$. Applying the above Lemma, we have that $(1-\epsilon)\,\mathrm{rank}(x) \le \hat{r}(x) \le \phi N$. Since $\hat{r}(x+1) > \phi N$, we also have (using the same Lemma again) $\phi N < \hat{r}(x+1) \le (1+\epsilon)\,\mathrm{rank}(x+1)$. Combining these gives the required result. □

3.2 Space Bounds

We have shown that if the bq-summary satisfies conditions (1) and (2) then it can answer rank and quantile queries accurately. We now describe how we will maintain our data structure to give these guarantees, while ensuring that the space used is tightly bounded. Our approach is common to previous data streaming algorithms for a variety of problems: we process new updates x by running a procedure INSERT(x); periodically (after a set number of arrivals) we will run a COMPRESS routine which compacts the data structure and removes redundant information, to ensure that the space bound always holds; when a query to x is posed, we run RANKQUERY(x) to return the approximate rank of x within the stated bounds. To make these routines efficient, we will ensure that our data structure has some additional properties.

Data Structure. Our data structure will be split into two pieces, which we refer to as *bq-leaves* and *bq-tree*. The bq-leaves, denoted bql, are a set of leaf nodes from the original tree, and associated counts, while the bq-tree, bqt, is a set of nodes (internal or leaf) from the original tree. These partition bq: $bq = bqt \cup bql$ and $bqt \cap bql = \emptyset$. We will maintain the additional following properties:

$$(v \in bqt) \wedge (\max_{u \in bql} u < \mathrm{lf}(\mathrm{par}(v))) \Rightarrow c_{\mathrm{par}(v)} \geq \alpha L(\mathrm{par}(v)) \tag{3}$$

$$\frac{1}{\alpha}\log(\alpha N) \geq |bql| \geq \min(N, \frac{1}{\alpha}) \tag{4}$$

$$\max_{u \in bql} u < \min_{v \in bqt} \mathrm{lf}(v) \tag{5}$$

$$\sum_{v \in bq} c_v = N \tag{6}$$

Condition (3) ensures that most internal nodes in bqt are "full": condition (2) limits the count of any node to at most $\alpha L(v)$, while this condition demands the count be at least this much, and both are satisfied at equality[3]. This condition ensures that the space needed to store bqt is compact, while condition (4) bounds the size of bql; these two conditions may lapse as the data structure is updated, so periodically we will restore them. Condition (5) says that all the leaves stored in bql occur to the left of all the tree nodes stored in bqt. These conditions will also enable us to update the data structure rapidly. Lastly, condition (6) is a check on the correctness of the manipulation of counts (in fact, it is a consequence of (1)).

Example. Figure 1 gives an example configuration of a bq-summary that obeys the required conditions with $\alpha = \frac{1}{2}$. The tree placed over the universe $[1 \ldots 16]$ is shown, with nodes in bq are marked with a black dot. Their count, c_v is also shown. These are divided into two parts: bql, which consists of leaves only, and bqt, which can consist of any nodes. The dividing line between bql and bqt is marked in grey: we insist that for every node in bqt, its parent is also in bqt unless the parent node falls on this dividing line (this follows from condition (3) above). The Figure is arranged so that all nodes with the same $\mathrm{lf}()$ value fall in the same vertical line; these nodes also have the same L values. The bq-summary represents the input $\{1, 2, 2, 2, 3, 4, 5, 6, 6, 6, 10, 12, 14, 14, 15, 16\}$ with $\alpha = \frac{1}{2}$. We can use it to find the rank of the item 6: $L(6) = 1+3+2+3+1 = 10$ and $A(6) = 1 + 3 = 4$, so we know that $6 \leq \mathrm{rank}(6) \leq 10$, and we estimate $\hat{r}(6) = 8$. (In fact, $\mathrm{rank}(6) = 7$.)

THEOREM 1. *Any bq-summary which satisfies* (3), (4) *and* (5) *with* $\alpha \leq \frac{1}{2\log U}$ *uses space* $O(\frac{\log(\alpha N)}{\alpha})$.

PROOF. We will consider the set of nodes V such that $v \in bqt$ and $c_v \geq \alpha L(v)$. By bounding the number of such nodes, we can bound the total number of nodes in the summary. Let W be the set of nodes such that $w \in bqt$ and $c_w < \alpha L(w)$. By condition (3), each $w \in W$ must have a parent satisfying the property $c_{\mathrm{par}(w)} \geq \alpha L(\mathrm{par}(w))$, or else it satisfies $\mathrm{lf}(\mathrm{par}(w)) < \max_{u \in bql} u < \mathrm{lf}(w)$. There can be at most $\log U$ nodes w that satisfy this latter property, and so $|W| \leq 2|V| + \log U$. We divide the nodes $v \in V$ into a sequence of *equivalence classes* based on the L value computed for v. That is, group together all nodes $u, v \in V$ for which $L(u) = L(v)$. Observe that all nodes in the same group must also share the same $\mathrm{lf}()$ value: suppose u and v are in the same group but $\mathrm{lf}(u) < \mathrm{lf}(v)$. Then $\mathrm{lf}(u) < \mathrm{lf}(v)$ and, by Definition 5, $L(u) < L(v)$, since $L(v)$ is bigger by at least c_u. These are illustrated in Figure 1: all nodes in bqt in the same equivalence class are shown in the same shaded vertical strips. Suppose there are a total of q equivalence classes. We will sort these q classes by the L value shared by all nodes in the class. Let E_i denote the set of nodes in the ith equivalence class, and let L_i denote the L value for that class, so $L_1 < L_2 \ldots < L_q$. Since E_i partitions V, $|V| = \sum_{i=1}^{q} |E_i|$. Observe that $|E_i| \leq \log U$, that is, there can be at most $\log U$ nodes in bqt in each equivalence class, at most one node from each level of the binary tree structure; this is because of the preceding observation, that all nodes in the class i must share the same $\mathrm{lf}(w)$ value, lf_i — there can be at most $\log U$ nodes sharing the same $\mathrm{lf}()$ value. Each bqt node v in class i must satisfy $c_v \geq \alpha L_i$, by applying condition (3). Thus for any equivalence class E_i, we have $\sum_{v \in E_i} c_v \geq \alpha |E_i| L_i$. Also, using condition (3)

$$L_{i+1} = \sum_{\mathrm{lf}(w) < \mathrm{lf}_{i+1}} c_w = \sum_{\mathrm{lf}(w) < \mathrm{lf}_i} c_w + \sum_{v \in E_i} c_v = L_i + \sum_{v \in E_i} c_v$$
$$\geq L_i + \alpha |E_i| L_i = (1 + \alpha |E_i|) L_i$$

Expanding the above expression, we have for any $j < i$

$$L_{i+1} \geq (1 + \alpha|E_i|)(1 + \alpha|E_{i-1}|) \ldots (1 + \alpha|E_j|) L_j.$$

If $|bql| < \frac{1}{\alpha}$, then by condition (4) $|bql| = N$, and so $|bqt| = 0$. So if $|bqt| > 0$, we also have $L_1 \geq 1/\alpha$, by combining (4) and (5): since all of bql precedes bqt, and there are at least $1/\alpha$ leaves in bql, we have that the L value of every node in bqt must be at least $1/\alpha$. Consider the "artificial" item, $U + 1$. Using (1), we have $\mathrm{rank}(U + 1) = N \leq L(U + 1)$. Place a 'fake' equivalence class $q + 1$ to the right of all items. We can write $L_{q+1} = L(U + 1) \geq \mathrm{rank}(U+1) = N$, since all items must be to the left of this notional extra equivalence class. Substituting into the above expression:

$$N \geq L_{E+1} \geq L_1(1 + \alpha|E_1|)(1 + \alpha|E_2|) \ldots (1 + \alpha|E_q|)$$

$$\ln \alpha N \geq \sum_{i=1}^{q} \ln(1 + \alpha|E_i|) \geq \sum_{i=1}^{q} \frac{3\alpha}{4}|E_i|$$

Here we use the fact that $\ln(1 + x) \geq \frac{3}{4}x$ for $x \leq \frac{1}{2}$, and also that $\alpha|E_i| \leq 1/2$, since we set $\alpha \leq \frac{1}{2\log U}$ and $|E_i| \leq \log U$.

So $|bq| = |bql| + |bqt| = |bql| + |V| + |W|$

$$\leq \frac{\log \alpha N}{\alpha} + \log U + 3|V| = O(\frac{\log \alpha N}{\alpha}) + 3\sum_{i=1}^{q} |E_i|$$

$$\leq O(\frac{\log \alpha N}{\alpha}) + \frac{4}{\alpha}(\log \alpha N) = O(\frac{1}{\alpha}\log(\alpha N))$$

□

Setting $\alpha = \epsilon / \log U$, this is within an $O(\log U)$ factor of the $\Omega(\frac{1}{\epsilon}\log(\epsilon N))$ lower bound for this problem proved in [3].

3.3 Insert Procedure

We next describe how to maintain the data structure in the presence of arrivals of updates. The INSERT (x) procedure takes a new item x and includes it in bq. We first determine whether to include x in bql or bqt: if $x \leq \max_{u \in bql}$ or $|bqt| = 0$, then we insert x into bql: if x is already present in bql then we increment c_x; else we insert x into bql and set $c_x = 1$. If we don't put x in bql, we will insert x into bqt: we will find the closest ancestor of x in bqt, v, and update the count of v, if this does not violate condition (2) — if it would, then we insert the descendant of v that is an ancestor of x into bqt and set its count to 1. This routine is illustrated in pseudo-code in Figure 2.

[3] Note that this means some counts will be fractional. We later show that restricting to integer counts does not affect the asymptotic space bounds.

```
INSERT(x)
Input: new item x

1: N := N + 1;
2: if (x < maxleaf) or (|bqt| = 0) then
3:   if x ∈ bql then
4:     c_x := c_x + 1;
5:   else
6:     bql := bql ∪ {x};  c_x = 1;
7: else
8:   w := binary search right(lca(maxleaf, x)) to x
     for least w ∉ bqt;
9:   if (par(w) = lca(maxleaf, x)) then
10:    bqt := bqt ∪ {w};  c_w := 1;  L(w) := |bql|;
11:  else
12:    if (c_par(w) + 1 ≤ αL(par(w))) then
13:      c_par(w) := c_par(w) + 1;
14:    else
15:      bqt := bqt ∪ {w};  c_w := 1;  L(w) := L(par(w));
```

Figure 2: INSERT **algorithm for maintaining bq-summary.**

LEMMA 3. INSERT*(x) maintains conditions* (1)*,* (2)*,* (5)*, and* (6)*.*

PROOF. When we insert x into bql, this is seen easily: for all $y \le x$, $\mathrm{rank}(y)$, $L(y)$ and $A(y)$ are unchanged; for all $y > x$, $A(y)$ is unchanged, but $L(y)$ and $\mathrm{rank}(y)$ both increase by 1. Hence, if (1) was true before the insertion, it remains true afterward. For (2), $L(v)$ either stays the same or increases for all $v \in bq$, while c_v stays the same. The only exception is c_x, which increases, but since $x = \mathrm{lf}(x)$, this condition does not apply. We only insert x into bql if $x \le \max_{u \in bql} u$ or $|bqt| = 0$, so (5) is maintained. We either add one to an existing c_v, or create a $c_v = 1$, so both $\sum_{v \in bq} c_v$ and N increase by 1, ensuring condition (6).

For inserting x into bqt, similarly $L(y)$ is either unchanged or increases by 1 for all y. Let v be the node in bqt that is affected by the insertion (i.e. either v was already in bqt and c_v was incremented, or else v was inserted into bqt). We ensure that (2) is not violated by the changes to (v, c_v), and no other c_vs are touched, so it must remain true. For $y < \mathrm{lf}(v)$, then $L(y)$, $\mathrm{rank}(y)$ and $A(y)$ are unchanged; for $y \ge x$ then $\mathrm{rank}(y)$ and $L(y)$ both increase by 1, and $A(y)$ either stays the same or increases by 1; lastly, if $y \ge \mathrm{lf}(v)$ and $y < x$ then $v \in \mathrm{anc}(y)$ so $\mathrm{rank}(y)$ stays the same, $L(y)$ increases by 1, but $A(y)$ also increases by 1. Hence, in all cases (1) is preserved. By design of the INSERT routine, (5) is preserved, since we ensure that $\max_{u \in bql} u < \mathrm{lf}(v)$. As in the leaf case, either an existing c_v is incremented or a new $c_v = 1$ is created, so condition (6) is preserved. □

LEMMA 4. INSERT*(x) can be carried out in time* $O(\log \log U)$*.*

PROOF. Inserting x into bql takes constant time: we just have to decide whether to insert into bql, and then update bql with the information about x. Inserting x into bqt can be accomplished initially using time $\log U$, by linearly searching along the path from the root to x for a place to insert x; however, using the additional properties we insist of our data structure this can be reduced to a binary search in less time.

Let $z = \max_{u \in bql} u$, the largest leaf stored in bql. We consider the case when $x > z$. Write $\mathrm{lca}(z, x)$ for the least common ancestor of z and x. Observe that $\mathrm{lf}(\mathrm{lca}(z, x)) \le z$, so by condition (5) $\mathrm{lca}(z, x)$ cannot be in bqt, and nor can any of its ancestors. Note that $\mathrm{lca}(z, -)$ can be computed efficiently in time $O(\log \log U)$ with some preprocessing: there are $\log U$ possible answers to this lca query, corresponding to the nodes on the path from z to the root. By finding this set and the $\mathrm{lf}()$ value of each node in the set, we can find $\mathrm{lca}(z, x)$ by binary searching into this set of leaves, at a cost of $O(\log \log U)$. Since $x > z$ then z must be in the left subtree of $\mathrm{lca}(z, x)$ and x must be in the right subtree, by definition of lca. Consequently, $\mathrm{lf}(\mathrm{right}(\mathrm{lca}(z, x))) > z$, and so by repeatedly applying (3), if any descendant w of $\mathrm{right}(\mathrm{lca}(z, x))$ is in bqt, then every node between w and $\mathrm{right}(\mathrm{lca}(z, x))$ must be in bqt. Thus, to find the closest ancestor of x in bqt, we can perform a binary search on the path between $\mathrm{right}(\mathrm{lca}(z, x))$ to find the (unique) node w such that $w \notin bqt$ but $\mathrm{par}(w)$ in bqt, and try to increment the count of $c_{\mathrm{par}(w)}$ or if this would violate condition (2), we insert w into bqt and set $c_w = 1$.[4] Two boundary conditions are easily handled: if $x \in bqt$, then we increment c_x; and if $\mathrm{right}(\mathrm{lca}(z, x)) \notin bqt$, then we insert it into bqt and set its count to 1. This binary search is over a path of length at most $\log U$, and so can be completed in time $O(\log \log U)$. □

```
COMPRESSTREE(v, dbt, L)
Input: Start node v, current debt dbt, L(v) value L

1: L(v) := L;  c' := c_v;
2: if (lf(v) = v) then
3:   c_v := c' − dbt;
4: else
5:   c_v := min(αL, weight(v) − dbt);
6:   if (c_v ≥ αL) then
7:     dbt := dbt + c_v − c';
8:     wl := weight(left(v));
9:     COMPRESSTREE(left(v), min(dbt, wl), L);
10:    COMPRESSTREE(right(r), max(dbt−wl, 0), L+wl+c');
11:  else
12:    remove all descendants of v from bqt
13:  if (c_v = 0) then
14:    remove v from bqt
```

Figure 3: COMPRESSTREE **Routine**

3.4 Compress Procedure

The COMPRESS procedure takes the data structure, and ensures that conditions (1)–(6) hold, ensuring that the data structure remains accurate for answering queries, but additionally the space used is tightly bounded. The procedure has several steps, which we outline and then explain in detail: first, we reduce the size of bql to its smallest permitted size, and insert the leaves that are removed from bql into bqt. We then recompute L values for all nodes in bqt to reflect the insertions that have happened, and then we compress each subtree within bqt by reallocating the weights.

Resizing bql is straightforward: we aim to ensure that $|bql| = \min(N, \frac{1}{\alpha})$. If this is already satisfied, then we need to take no action. If $|bql| > \frac{1}{\alpha}$, then we find the $\frac{1}{\alpha}$-largest leaf u (in universe order), and remove all leaves v from bql for which $v > u$. Let z denote the previous largest leaf in bql; now u fulfills this role. Note that condition (3) may now be violated, since $u < z$, and so some nodes in bqt may now have parents that should be present with non-zero count. To fix this up, we insert all nodes needed to ensure that every node in bqt also has its parent present unless this parent is an ancestor of u. These nodes are introduced to bqt with c_v set to

[4] Note that we do not compute $L(v)$ exactly here, since it would be too expensive. Instead, for each node we store an old value of $L(v)$. Since $L(v)$ only increases with time, there is no accuracy problem with using an outdated $L(v)$ value. We may choose to insert a child of v when we could have increased the count of v, but this does not affect our (worst case) space bounds. Periodically, when we run a COMPRESS operation we will recalculate the $L(v)$ values.

0. In a subsequent step, we will ensure that these "dummy" nodes are allocated non-zero count and are treated identically to all other nodes in bqt. All these dummy nodes are nodes on the path from z to the root, so there are at most $\log U$ such nodes needed. We then take all leaves $v > u$ that were in bql, and run INSERT(v) on each of them, to put them into bqt.

We now define an operation COMPRESSTREE(v, dbt, L), which takes a node $v \in bqt$, and manipulates the counts stored in the subtree defined by v so as to restore condition (3). This is illustrated in Figure 3. It makes use of the weight of a node v, which is defined by $\mathrm{weight}(v) = c_v + \sum_{w:v\in\mathrm{anc}(w)} c_w$. We can precompute $\mathrm{weight}(v)$ for all $v \in bqt$ with a recursive algorithm that takes time $O(|bqt|)$. The COMPRESSTREE procedure runs recursively over the tree, and restores condition (3), by ensuring that every node v in bqt that has children has $c_v = \alpha L(v)$. This is done top down. For any node, it computes the difference between c_v and $\alpha L(v)$: this is the slack that can be filled up by "borrowing" counts from nodes below. The amount of count that is borrowed is denoted by dbt. This is propagated down to the children, with preference to the left child[5]. When the total amount of borrowed weight equals the weight of the descendants of a node, we can remove all of these descendants from bqt, to "repay the debt". This is where we gain in COMPRESS, since we can reduce $|bqt|$. A side-effect of COMPRESSTREE is to compute $L(v)$ for each node in bqt as it operates. Throughout, we take care to ensure that $\sum_v c_v = N$ at all times, i.e. no counts are lost or added and condition (6). The details of this procedure are given in pseudo-code in Figure 3.

For each node v on the path from u to the root, if $v \in bqt$, we run COMPRESSTREE on the right child of v (if such a node is materialized), setting the initial value of dbt to zero, and the initial value of L to the L value of v, which can be derived easily from information already computed.

LEMMA 5. *Conditions (1) – (6) are true after running* COMPRESS *over the bq-summary data structure.*

PROOF. We first argue that the accuracy bounds (conditions (1) and (2)) remain true after a COMPRESS operation. (2) is straightforward: for all nodes in bqt that are not leaves of the tree, we explicitly ensure that $c_v \le \alpha L(v)$. For (1), both the operations on the leaves, and inserting some leaves into bqt, preserve (1), following from Lemma 3. So we just have to argue that COMPRESSTREE operations also preserve this condition. We first argue that for any $x \in [U]$, $L(x)$ only increases when COMPRESSTREE is carried out. This is because when we remove all descendants of a node v, all the counts associated with these deleted nodes are added on to v or one of its ancestors. Since for any u in the tree, every $v \in \mathrm{anc}(u)$ satisfies $\mathrm{lf}(v) \le \mathrm{lf}(u)$, so no $L(x)$ can decrease when the count of u is moved to v. Thus, if $\mathrm{rank}(x) \le L(x)$ was true before the COMPRESSTREE operation, it must remain true after. We now argue that if $L(x)$ increases, then $A(x)$ increases by the same amount. Let x be some node such that after a COMPRESSTREE operation, $L(x)$ has increased by some value d. This increase must be due to some nodes w satisfying $x < \mathrm{lf}(w)$ whose count was allocated to some node v satisfying $\mathrm{lf}(v) < x$, else $L(x)$ would not increase. Since count is only propagated to ancestors, we must have $v \in \mathrm{anc}(w)$. Since $\mathrm{anc}(w) \subseteq \mathrm{anc}(\mathrm{lf}(w))$, we have $v \in \mathrm{anc}(\mathrm{lf}(v))$ and $v \in \mathrm{anc}(\mathrm{lf}(w))$, and $\mathrm{lf}(v) < x < \mathrm{lf}(w)$. So $v \in \mathrm{anc}(x)$. Thus $A(x)$ also increases by d. Hence $L(x) - A(x)$ does not change, and so (1) is preserved.

For the space bounds, observe that condition (4) is true since we force $|bql| > \frac{1}{\alpha}$, and (5) is also true, since we only add nodes v to bqt that have $\mathrm{lf}(v) > \max_{u\in bql} u$. Lastly, (3) and (6) are enforced by COMPRESSTREE, and if we delete any node v from bqt then we ensure that the entire subtree rooted at v is deleted while its count is added on to that of an ancestor. □

LEMMA 6. COMPRESS *can be carried out in time* $O(|bqt| + |bql| \log\log U)$.

PROOF. Scanning bql to find the $\frac{1}{\alpha}$-largest leaf takes time linear in $|bql|$, using standard algorithms [2]. Adding dummy nodes to bqt takes time $O(\log U)$. Inserting each of the larger leaves into bqt takes time $O(\log\log U)$ per leaf, from Lemma 4. Computing $\mathrm{weight}(v)$ for all $v \in bqt$ takes time $O(|bqt|)$ as observed above. Calling COMPRESSTREE(v) takes time linear in the number of nodes that are descendants of v, as can easily be proved by induction. This routine is called once for each $v \in bqt$, a total of $|bqt|$ nodes, and takes constant time per node visited. Thus, the total time to run a COMPRESS is given by $O((|bql| \log\log U) + \log U + |bqt|) = O(|bqt| + |bql| \log\log U)$. □

Query Procedure. We have already given the algorithm to return the approximate rank of an item x: we find $L(x)$ and $A(x)$, and return $\hat{r}(x) = L(x) - \frac{1}{2}A(x)$. To compute $L(x)$ requires a linear scan of the data structure. If many queries are posed in a batch, we can reduce this cost, by computing the L values for all nodes in bq. Computing these values requires sorting the nodes in bql, but can be computed in linear time for bqt (indeed, this is done in the COMPRESSTREE algorithm). Queries can then be answered quickly: given a query x, we determine whether $x \le \min_{u\in bqt} \mathrm{lf}(u)$. If it is, then we find $v = \max_{u\in bql, u<x}$ and output $L(v)$. If x falls in bqt, we find w such that w is where we would insert x if we were performing an insertion, and output $L(w)$. Thus, the time cost is $O(\log\log U)$ in the bqt case, and $O(\log\log U + \log(1/\epsilon))$ in the bql case (a binary search into a sorted list of leaves).

THEOREM 2. *We can maintain a data structure that allows us to answer biased quantile queries and biased rank queries using space* $O(\frac{\log U}{\epsilon}\log(\epsilon N))$. *The amortized cost is* $O(\log\log U)$ *per update.*

PROOF. In order to ensure that all bounds hold, we must specify how often to run the COMPRESS procedure. Too frequent, and the amortized cost is too high; too rare, and the space bounds may be exceeded. We first run COMPRESS after $N = 4/\alpha$ insertions have been seen. Then we run COMPRESS whenever the number of updates since the last COMPRESS operation, n, exceeds $\frac{\log \epsilon N'}{\alpha}$, for the current value of N'. Note after the previous COMPRESS the space used was bounded by $O(\frac{\log \epsilon N}{\alpha})$, and in the worst case, after this many updates, the space used has grown by $O(\frac{\log(\epsilon N')}{\alpha})$, since every INSERT can add at most one new tuple to bq. Thus the total size of the data structure is $O(\frac{\log(\epsilon N')}{\alpha})$, since $N < N'$. The running time of the COMPRESSTREE is linear in the worst case size of the data structure, which in turn is bounded by the number of updates, so the cost can be amortized against the number of updates. We also incur a $O(\log\log U)$ cost for each member of bql that is moved into bqt, but observe that this conversion happens at most once for each leaf in bql, and can be charged back to an INSERT operation. So, combining COMPRESS and INSERT, the amortized time cost of each update is dominated by $O(\log\log U)$. The worst case space cost is just prior to a compress, which we have argued is $O(\frac{\log(\epsilon N)}{\alpha})$, giving the stated bounds. □

[5] This preference does not affect correctness or space bounds; we experimented with preferring both left and right, and found little difference in practice between the two versions.

4. APPLICATIONS AND EXTENSIONS

4.1 Simplified Algorithm

We now describe a simplified version of the above algorithm which has some slightly weaker bounds but is much simpler to implement. Instead of splitting bq into bqt and bql, it treats all of bq the same way. We do not maintain conditions (3), (4) or (5), but instead use a new condition for proving space bounds:

$$\forall v \in bq : \mathrm{par}(v) \in bq \text{ and } c_{\mathrm{par}(v)} \geq \lfloor \alpha L(\mathrm{par}(v)) \rfloor \quad (7)$$

This allows insertions of x to be performed quickly, by binary searching along the path from x to the root of the tree for a node to place the new count. However, we must allow some nodes in bq to have $c_v = 0$, which increases the space cost. In addition, we force all c_v counts to integral, by replacing the occurrences of $\alpha L(v)$ in Condition (2), INSERT and COMPRESSTREE with $\lfloor \alpha L(v) \rfloor$.

THEOREM 3. *We can maintain a data structure that allows us to answer biased quantile queries and biased rank queries using space $O(\frac{\log U}{\epsilon}(\log \epsilon N + \log U))$. The amortized cost is $O(\log U)$ per update.*

PROOF SKETCH. We outline the key differences between the simplified algorithm and the preceding algorithm. The main observation that links the two is that the leaves beneath dummy nodes that have $c_v = 0$ correspond to bql from the previous algorithm. After the first $\frac{\log U}{\epsilon}$ leaves, we can start populating internal nodes with integer counts, since $L(v) > 1/\alpha$ and so $\lfloor \alpha L(v) \rfloor > 1$. Further, when $L(v) > 2/\alpha$, we have $\lfloor \alpha L(v) \rfloor \geq \frac{\alpha}{2} L(v)$, thus we know that, applying the same equivalence class argument as in Theorem 1, each L_i is at least a $(1 + \alpha/2)$ factor more than its preceding value, and one can show the same $O(\frac{\log \epsilon N}{\alpha})$ space bound. Thus, the total space is given by adding on the $O(\frac{1}{\alpha})$ leaves, each of which has $O(\log U)$ ancestors, giving the stated space bound.

This "full ancestry" property (every node in the data structure has all its ancestors also present in the data structure) makes maintenance of the data structure conceptually simpler. INSERT(x) operations just have to binary search the path from x to the root to determine where to insert x; the only complication is that we may have to insert x as a leaf and create some missing ancestors with count zero, which gives a worst case $O(\log U)$ insertion cost. COMPRESS is straightforward: we just have to run the COMPRESSTREE operation on the root node periodically. Following the same argument as Theorem 2, the amortized cost of updates can be bounded by the worst case $O(\log U)$ cost of INSERT. □

4.2 Partially Biased Algorithm

In the partially biased case (Definition 1 (b)), we are allowed to give slightly weaker accuracy guarantees, so we should be able to take advantage of this to reduce the space needed. In order to do this, we can modify our previous algorithms slightly and give a new analysis that shows reduced space costs. We adapt the conditions (2) and (3) by replacing $\alpha L(v)$ with $\max(\frac{\epsilon_{\min} N}{\log U}, \alpha L(v))$. This gives a potentially larger slack to the data structure for some nodes that have small L values.

We begin by running the unmodified biased quantiles algorithm, with space $O(\frac{\log \epsilon N}{\alpha})$, because until $N = \frac{\log U}{\epsilon_{\min}}$, there is little benefit from applying the tighter pruning condition. However, when $N = \frac{\log U}{\epsilon_{\min}}$, every internal node has slack at least 1, and so we can convert the data structure into one with the "full ancestry" property, by inserting all the leaves from bql into the main tree structure, and running COMPRESS TREE. We can perform INSERT(x) by binary searching into the path between x and the root, finding a childless node with capacity, or creating its child and inserting there.

THEOREM 4. *We can answer partially biased rank queries with error $\max(\epsilon_{\min} N, \epsilon\, \mathrm{rank}(v))$ using space $O(\frac{\log U}{\epsilon} \log \frac{\epsilon}{\epsilon_{\min}})$.*

PROOF. To begin with, we use the algorithm of Theorem 1, which uses space $O(\frac{\log \alpha N}{\alpha})$, while $N \leq \frac{\log U}{\epsilon_{\min}} = \frac{\epsilon}{\epsilon_{\min}\alpha}$. Thus, this space for this initial phase is bounded by $O(\frac{\log U}{\epsilon} \log \frac{\epsilon}{\epsilon_{\min}})$.

For the analysis of the case when $N > \frac{\log U}{\epsilon_{\min}}$, we (notionally) split the data structure into two parts: a left hand part to which the $\frac{\epsilon_{\min}}{\log U} N$ bound applies, and a right hand part to which the $\alpha L(v)$ bound applies. For this right hand part, we have $\epsilon L(v) \geq \epsilon_{\min} N$. We can now apply the same approach as in Theorem 1, of dividing the nodes into equivalence classes based on their L values. As before, we have $L_{i+1} \geq (1 + \alpha |E_i|) L_i$. Starting from the first equivalence class where $\epsilon L(v) \geq \epsilon_{\min} N$, we can show that the total number of internal nodes in this right hand side is $O(\frac{\log U}{\epsilon} \log(\epsilon/\epsilon_{\min}))$, by modifying the proof of Theorem 1.

For the left hand part, the sum of counts of all nodes is at least $\epsilon_{\min} N/\epsilon$, by the condition on $L(v)$. There is an equivalence class i such that $L_i < \epsilon_{\min} N/\epsilon$ but $L_{i+1} \geq \epsilon_{\min} N/\epsilon$, which marks the division between the left and right parts. The size of this equivalence class is $|E_i| < \log U$. The total number of internal nodes retained in the left hand part each have count at least $\epsilon_{\min}/\log U$, and the sum of their counts is at most $\epsilon_{\min} N/\epsilon$. So there can be at most $(1 + 1/\epsilon) \log U = O(\frac{\log U}{\epsilon})$ nodes retained, accounting for the extra $\log U$ nodes in E_i. Combining these two parts, the space required is dominated by the right hand part, $O(\frac{\log U}{\epsilon} \log \frac{\epsilon}{\epsilon_{\min}})$. So, irrespective of the value of N, the space is bounded by this quantity. □

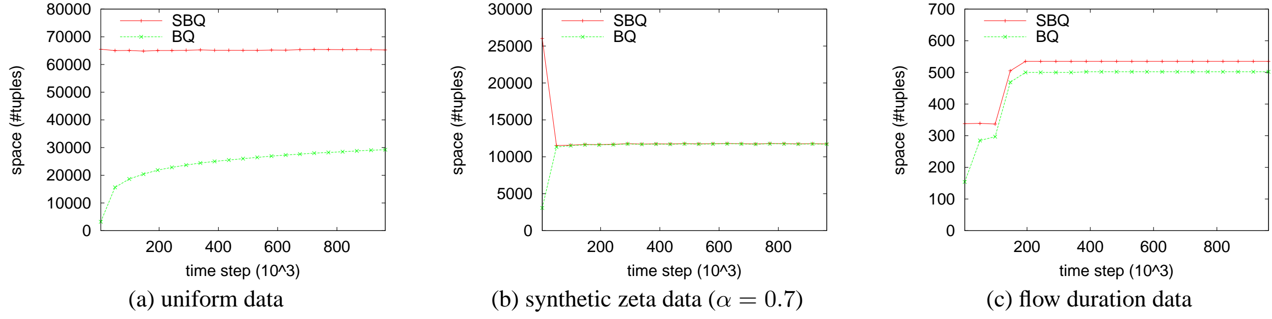
Observe that in the case that $\epsilon_{\min} \leq 1/N$, this cost reduces to the bound for the fully biased case, as one would hope. We note that in [3], a lower bound of $O(\frac{1}{\epsilon} \log(\epsilon \min(N, 1/\epsilon_{\min})))$ on the space required was shown; hence this data structure is within a factor of $O(\log U)$ of being optimal. Further, this is the first-known *deterministic* algorithm with proven space and accuracy guarantees.

4.3 Uniform Rank and Quantile Queries

Our data structure can be modified slightly in order to give uniform quantile error guarantees (Definition 2). That is, the error is at most $\epsilon_{\min} N$ for *any* rank query v, instead of $\epsilon\, \mathrm{rank}(v)$. To do this, we run the same algorithms, but replace all references to $\alpha L(v)$ with $\epsilon_{\min} N/\log U$. This is similar to the structures given in [9, 6], but gives a slightly improved amortized time bound ($O(\log \log U)$ instead of $O(\log \log U + \log 1/\epsilon)$), which follows from [9].

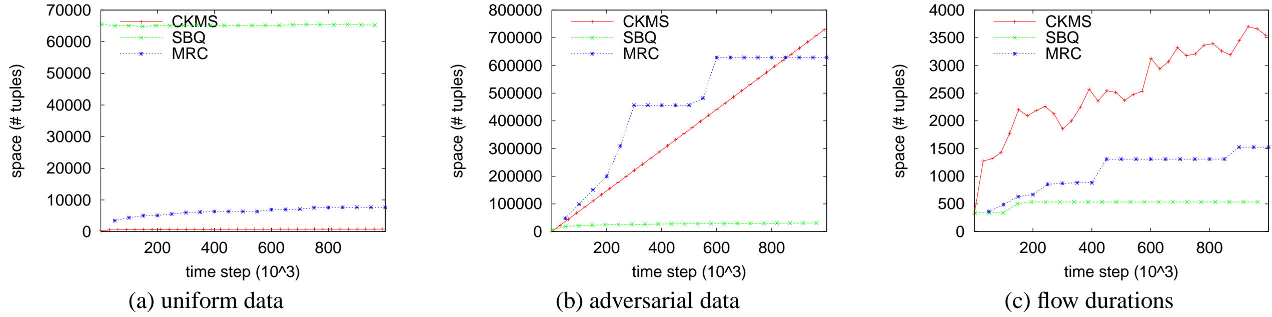
THEOREM 5. *Tracking quantiles in 1D with uniform error guarantees can be carried out using space $O(\frac{\log U}{\epsilon_{\min}})$. Each update takes amortized time $O(\log \log U)$, and queries take time $O(\frac{\log U}{\epsilon_{\min}})$.*

PROOF. We no longer need to make a distinction between bql and bqt; instead, we just have to buffer the first $\frac{2 \log U}{\epsilon_{\min}}$ updates. This ensures that $N > 2\frac{\log U}{\epsilon_{\min}}$, and so $\lfloor \epsilon_{\min} N/\log U \rfloor > 2$. COMPRESSTREE ensures that each node stored in our data structure such that some of its child nodes are stored, has count at least $\lfloor \epsilon_{\min} N/\log U \rfloor$, and because we maintain a complete subtree, the number of non-internal nodes is at most twice the number of internal nodes. The space bounds follows by considering the number of internal nodes: since the sum of all counts must be N, we have that the number of such nodes is at most $N/\lfloor \epsilon_{\min} N/\log U \rfloor = O(\frac{\log U}{\epsilon_{\min}})$. Insertions of x take time worst case $O(\log \log U)$ to binary search for where to insert on the path from x to the root. The amortized cost of running COMPRESSTREE after every $O(\frac{\log U}{\epsilon_{\min}})$ insertions is $O(1)$ per insertion. Queries can be answered with a linear scan over the data structure, i.e. in time $O(\frac{\log U}{\epsilon_{\min}})$. □



(a) uniform data (b) synthetic zeta data ($\alpha = 0.7$) (c) flow duration data

Figure 4: Comparison of algorithm from Section 3 with simplified version for fully biased quantiles ($\epsilon = 0.01$).



(a) uniform data (b) adversarial data (c) flow durations

Figure 5: Space usage of the algorithms for fully biased quantiles ($\epsilon = 0.01$).

4.4 Targeted Quantiles

Recall that the targeted quantiles problem (Definition 4) is specified by a set T of pairs $\{\phi_j, \epsilon_j\}$, and requires that we return a set of items v_j whose rank is $\phi_j N \pm \epsilon_j N$. Any method which answers quantile queries with uniform guarantees can be used to solve the targeted quantiles problem by setting the accuracy parameter $\epsilon_{\min} = \min_j\{\epsilon_j\}$. This gives a space requirement of $\Omega(\max_j\{\epsilon_j^{-1}\})$. However, we know that we should be able to do better, since we do not need this guarantee over all the whole domain, just for ϕ_j. For a single pair $T = \{\phi_1, \epsilon_1\}$, Manku et al. obtained a sampling based randomized algorithm using $O(\frac{\phi_1}{\epsilon_1} \log \frac{1}{\delta})$ samples [7]. Thus, for larger sets T, $O(\sum_j \frac{\phi_j}{\epsilon_j} \log(|T|/\delta))$ space is required. By applying our results for biased and fully biased quantiles, we obtain improved deterministic bounds for the targeted quantiles problem.

THEOREM 6. *A single targeted quantile query can be answered using space* $O(\frac{\phi_1}{\epsilon_1} \log(\frac{1}{\phi_1}) \log U)$

PROOF. For simplicity, assume $\phi_1 \leq \frac{1}{2}$ (if not, then we can reverse the ordering and replace ϕ_1 with $1 - \phi_1$ to get the tighter bound). We run the biased quantiles algorithm with parameters ϵ and $\epsilon_{\min}$ chosen as appropriate functions of ϵ_1 and ϕ_1, as follows: The smallest error we need to guarantee is $\epsilon_{\min} = \epsilon_1$. When the rank is $\phi_1 N$, we need to give error $\epsilon_1 N$ This is a relative error of $\epsilon = \epsilon_1/\phi_1$. Substituting these values into the bounds for the biased algorithm gives $O(\frac{1}{\epsilon} \log \frac{\epsilon}{\epsilon_{\min}} \log U) = O(\frac{\phi_1}{\epsilon_1} \log(\frac{1}{\phi_1}) \log U)$. □

THEOREM 7. *A set of targeted quantile queries can be answered using space* $O(\frac{\phi_k}{\epsilon_k} \log(\frac{\epsilon_k}{\phi_k} \min(N, \max_i\{\epsilon_i^{-1}\})) \log U)$ *where* $k = \text{argmax}_j \frac{\phi_j}{\epsilon_j}$.

PROOF. As above, consider the relative error implied by each (ϕ_j, ϵ_j) pair. The smallest relative error is achieved by $\min_j \frac{\epsilon_j}{\phi_j}$, i.e. (ϕ_k, ϵ_k). Hence if we can guarantee relative error $\epsilon = \frac{\epsilon_k}{\phi_k}$ then we can satisfy the accuracy requirements of all targeted quantile requests (and potentially give tighter than required answers). We apply the fully biased quantiles algorithm above, but since we never require accuracy tighter than $\epsilon_{\min} = \min_j\{\epsilon_j\}$, we can get slightly tighter bounds for this case, giving the stated bounds. □

4.5 Distributed Streams

Given two bq-summaries, one can easily merge the two summaries to create a bq-summary of the union of the inputs. We just have to take the union of the two bqls as the new bql, and the union of the bqts as the new bqt: if v is present in both summaries then we set its count c_v to be the sum of the counts, else if it is only present in one summary, then we keep the previous value as its new count. It is straightforward to show that merging preserves the conditions on the counts, since L, A, and rank are linear functions. Following the merge, one can run COMPRESS to restore the space bounds. This means that we can compute biased rank queries and biased quantiles over distributed streams, by computing the summaries locally and then merging the summaries at a central site.

5. EXPERIMENTS

In this section, we discuss our experimental results. In the first subsection, we compare our bq-summary with fully biased error guarantees against the deterministic algorithm for biased quantiles [3] as well as the randomized algorithm [10]. For fair time comparison, we obtained the implementations used by the authors of [3] and [10] in their prior experimental evaluations. In the second subsection, we compare the partially biased version of our algorithm

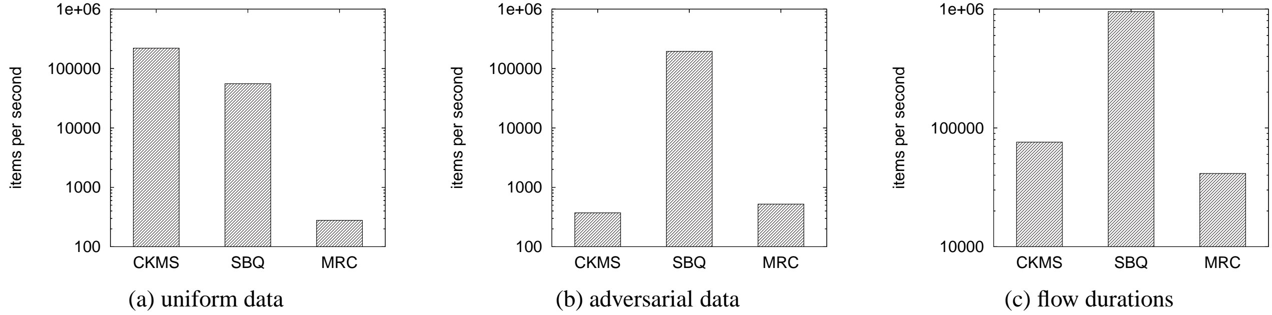


Figure 6: Runtime of the algorithms for fully biased quantiles ($\epsilon = 0.01$).

against the partially biased version of the one in [3], as well as the straightforward application of existing uniform quantile algorithms [4, 6] using the minimum allowable error bound ϵ_{min}.

We implemented the simplified version of the proposed algorithm described in Section 4.1[6]. Recall that this version satisfies invariants (1) and (2), and thus gives the desired accuracy guarantees, but in theory may require $O(\frac{1}{\epsilon}\log^2 U)$ additional space compared to the one presented in Section 3. As we shall see, its space usage is much less in practice. We also used our implementation of the uniform quantiles problem described in Section 4.3.

Experiments were run on a Pentium 4 i686 machine running Linux with 2.8 GHz CPU speed, 2 GB of main memory, and 512 KB cache size. We used both synthetic and real data in our experiments. The synthetic data includes uniform random data from a universe of size 2^{32}; skewed data generated using the zeta (discrete Pareto) distribution with parameter α, where the probability of the ith most frequent item is proportional to $i^{-\alpha}$; and adversarial data for the existing algorithm [3] as described in [10]: a sequence of batches of items whose values are between the current maximum and second-maximum tuples in the quantile summary. The real data sets include flow-level IP traffic measurements obtained using Cisco NetFlow at an ISP router carrying a heavy load of traffic, and projected out the fields *#pkts*, *#octets*, *duration*, *srcIP* and *destIP*. We report space usage in terms of the number of tuples kept by the respective data structures, as a function of the number of stream items that have arrived. We measured the time cost of our algorithms, and computed average throughput in items per second.

5.1 Fully Biased Quantiles

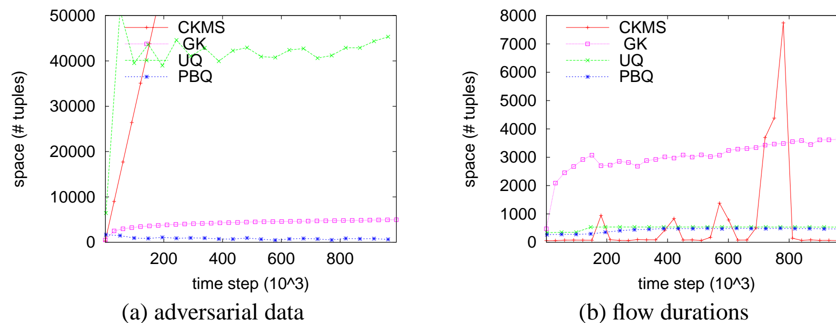
Our first set of experiments set out to study the space cost incurred by using the simplified version of our algorithm ("SBQ") described in Section 4.1, compared to the one in Section 3 ("BQ"); we estimated the space cost of BQ by discounting the number of nodes stored by SBQ with zero counts (this gives an upper bound on the cost of BQ). We observed only small differences in the space usage of BQ and SBQ on all of our data sets except on uniform data (Figure 4 (a)). On uniform data, the difference was about 35K after a million input items, which is considerably less than the theoretical worst case cost, $\frac{1}{\epsilon}\log^2 U \approx$ 102K. On skewed data, the space usage decreases with skew–from $|bq|$ =30K for zeta with $\alpha = 0$ (uniform) to $|bq|$ =12K with $\alpha = 0.7$ (shown in Figure 4(b)) to 6K with $\alpha = 0.9$. Figure 4(c) shows that the space usage of SBQ on real data: as expected, the cost is low, and the difference between SBQ and BQ small, since in practice data exhibits significant skew.

To compare against prior algorithms, we tested the space and time costs for biased quantiles with $\epsilon = 0.01$. We compared our methods with the deterministic algorithm ("CKMS") from [3], and the randomized algorithm ("MRC") from [10], even though this algorithm does not give deterministic error guarantees; the confidence level $(1 - \delta)$ was set to be as generous as possible, at 90%. Figure 5 graphs the space usage for these algorithms on three different data sets, and Figure 6 plots histograms of the throughput (stream items per second) for the respective algorithms in log scale. Whereas CKMS used very little space on (randomly-ordered) uniform data (see Figure 5(a)), SBQ required significantly more; in fact, as was observed in [10], CKMS used less space than MRC on uniform data. SBQ used the most space, since uniform data appears to be one of the hardest cases for SBQ — it resulted in the largest observed space usage out of all the data sets we tried — and our experiments below demonstrate much better space and time efficiency on real data. However, it was still over two orders of magnitude faster than MRC (Figure 6 (a)). Figure 5(b) shows the results on the "adversarial" data set described in [10], where SBQ used the smallest space and processing time compared to both CKMS and MRC (running time again better by two orders of magnitude). Finally, Figures 5(c) and 6(c) compare the performances of these algorithms on (real) flow durations. After a million items, SBQ used three times less space than MRC and seven times less than CKMS; its processing throughput was 12 times better than CKMS, and 27 times better than MRC, showing significant wins on both time and space cost on real data sets.

5.2 Partially Biased Quantiles

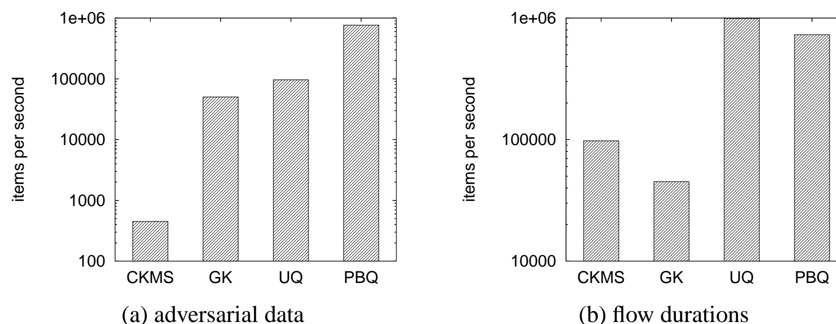
We compared the partially biased variant of CKMS (see [3]) with our partially biased algorithm, "PBQ", using parameters $\epsilon = 0.1$ and $\epsilon_{min} = 0.001$. We also ran the GK algorithm ("GK") as well as our implementation of a uniform quantile algorithm, "UQ", (described in Section 4.3), both at the conservative error bound $\epsilon_{min} = 0.001$ (thus guaranteeing to meet the accuracy requirements). Figures 7(a) and 8(a) present the space usage and throughput using the adversarial data. Again, CKMS exhibits linear space usage on this data, whereas PBQ, which is provably sublinear, uses significantly less space in practice. Interestingly, CKMS does much worse than the GK algorithm from which it is adapted, since CKMS prunes its summary structure more aggressively, which turns out to be detrimental in the long term. UQ also required a lot more space than PBQ, although unlike CKMS its space requirement is independent of N, and so it levels off. Figures 7(b) and 8(b) present results using flow duration data. Note the bursty space usage of CKMS in Figure 7(b). The space increase is due to values at low ranks requiring very fine accuracy; the error constraint relaxes as these ranks falls below the span of the partial bias, and with it the

[6]We have recently implemented the main algorithm, and experimental results will be reported in the full version of this paper.



(a) adversarial data (b) flow durations

Figure 7: Space usage from the algorithms for partially biased quantiles on (a) adversarial data and (b) flow durations ($\epsilon = 0.1, \epsilon_{min} = 0.001$).



(a) adversarial data (b) flow durations

Figure 8: Runtime comparison from the algorithms for partially biased quantiles on (a) adversarial data and (b) flow durations ($\epsilon = 0.1, \epsilon_{min} = 0.001$).

space usage. GK requires less space than CKMS, but still over six times more than that of PBQ. On this data, it appears that the additional pruning power of PBQ over its uniform counterpart gives only a slight decrease in space cost (and a slight increase in time cost). The overall runtime performance of PBQ/UQ is well over an order of magnitude better than that of CKMS and GK.

6. CONCLUDING REMARKS

We have given the first space-efficient deterministic algorithms for a variety of problems including biased quantiles, biased rank queries, and targeted quantiles. They are fast to process each update in high volume data streams, and have strong space guarantees that are close to optimal. Experimentally, they often outperform existing methods in both time and space requirements.

Our algorithms given here use a "universe-aware" approach to tracking the distributional information. They require knowledge of the universe, U, from which the items are drawn, and incorporate $\log U$ explicitly into the algorithm. This constraint is reasonable for many data streaming scenarios—indeed, we saw that it is no handicap in our experimental study—but for some settings, when U is very large, or unbounded (e.g., arbitrary real values), it can become problematic. For uniform guarantees, the GK algorithm has no explicit dependency on U. Prior work on biased quantiles gave algorithms derived from GK that did not require knowledge of U [3]; however, as shown in [10], these algorithms exhibit worst case behavior that is linear in U. Thus, it remains open to find solutions to the biased quantiles problems we study which are "universe-agnostic": they do not require specific knowledge of U, and $\log U$ does not enter into their asymptotic bounds.

7. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of ACM Principles of Database Systems*, pages 1–16, 2002.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. In *IEEE International Conference on Data Engineering*, 2005.

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

[5] A. Gupta and F. Zane. Counting inversions in lists. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 253–254, 2003.

[6] J. Hershberger, N. Shrivastava, S. Suri, and C. Toth. Adaptive spatial partitioning for multidimensional data streams. In *ISAAC*, 2004.

[7] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, volume 28(2) of *SIGMOD Record*, pages 251–262, 1999.

[8] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[9] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *ACM SenSys*, 2004.

[10] Y. Zhang, X. Lin, J. Xi, F. Korn, and W. Wang. Space-efficient relative error order sketch over data streams. In *IEEE International Conference on Data Engineering*, 2006.