Relative Error Streaming Quantiles

[Extended Abstract]

Graham Cormode
University of Warwick
Coventry, UK
G.Cormode@warwick.ac.uk

Zohar Karnin
Amazon, USA
zkarnin@gmail.com

Edo Liberty
Pinecone
San Mateo, CA, USA
edo@edoliberty.com

Justin Thaler
Georgetown University
Washington, D.C., USA
justin.thaler@georgetown.edu

Pavel Veselý
Charles University
Prague, Czech Republic
vesely@iuuk.mff.cuni.cz

ABSTRACT

Estimating ranks, quantiles, and distributions over streaming data is a central task in data analysis and monitoring. Given a stream of n items from a data universe equipped with a total order, the task is to compute a sketch (data structure) of size polylogarithmic in n. Given the sketch and a query item y, one should be able to approximate its rank in the stream, i.e., the number of stream elements smaller than or equal to y.

Most works to date focused on additive εn error approximation, culminating in the KLL sketch that achieved optimal asymptotic behavior. This paper investigates *multiplicative* $(1 \pm \varepsilon)$-error approximations to the rank. Practical motivation for multiplicative error stems from demands to understand the tails of distributions, and hence for sketches to be more accurate near extreme values.

The most space-efficient algorithms due to prior work store either $O(\log(\varepsilon^2 n)/\varepsilon^2)$ or $O(\log^3(\varepsilon n)/\varepsilon)$ universe items. We present a randomized sketch storing $O(\log^{1.5}(\varepsilon n)/\varepsilon)$ items, which is within an $O(\sqrt{\log(\varepsilon n)})$ factor of optimal. Our algorithm does not require prior knowledge of the stream length and is fully mergeable, rendering it suitable for parallel and distributed computing environments.

1. INTRODUCTION

Understanding the distribution of data is a fundamental task in data monitoring and analysis. In many settings, we want to understand the cumulative distribution function (CDF) of a large number of observations, for instance, to identify anomalies. In other words, we would like to track the median, percentiles, and more generally quantiles of a massive input in a small space, without storing all the observations. Although memory constraints make an exact computation of such order statistics impossible [22], most applications can be satisfied with approximating the quantiles, which also yields a compact function with a bounded distance from the true CDF.

The problem of streaming quantile approximation captures this task in the context of massive or distributed datasets. Let $\sigma = (x_1, \ldots, x_n)$ be a stream of items, all drawn from a data universe $\mathcal{U}$ equipped with a total order. For any $y \in \mathcal{U}$, let $\mathrm{R}(y; \sigma) = \left|\left\{i \in \{1, \ldots, n\} \mid x_i \leq y\right\}\right|$ be the rank of y in the stream. When σ is clear from context, we write $\mathrm{R}(y)$. The objective is to process the stream in one pass while storing a small number of items, and then use those to approximate $\mathrm{R}(y)$ for any $y \in \mathcal{U}$. A guarantee for an approximation $\hat{\mathrm{R}}(y)$ is *additive* if $|\hat{\mathrm{R}}(y) - \mathrm{R}(y)| \leq \varepsilon n$, and *multiplicative* or *relative* if $|\hat{\mathrm{R}}(y) - \mathrm{R}(y)| \leq \varepsilon\, \mathrm{R}(y)$. Estimating ranks immediately yields approximate quantiles, and vice versa, with a similar error guarantee (recall that a ϕ-quantile for $\phi \in [0, 1]$ is the $\lfloor \phi n \rfloor$'th smallest item in σ). We stress that we do not assume any particular data distribution or that the stream is randomly-ordered.

A long line of work has focused on achieving additive error guarantees [23, 2, 19, 24, 13, 3, 12, 1, 11, 16]. However, additive error is not appropriate for many applications. Indeed, often the primary purpose of computing quantiles is to understand the tails of the data distribution. When $\mathrm{R}(y) \ll n$, a multiplicative guarantee is much more accurate and thus harder to obtain. As pointed out by Cormode et al. [5], a solution to this problem would also yield high accuracy when $n - \mathrm{R}(y) \ll n$, by running the same algorithm with the reversed total ordering (simply negating the comparator).

A quintessential application that demands relative error is monitoring network latencies. In practice, one often tracks response time percentiles 50, 90, 99, 99.9, etc. This is because latencies are heavily long-tailed. For example, Masson et al. [21] report that for web response times, the 98.5th percentile can be as small as 2 seconds while the 99.5th percentile can be as large as 20 seconds. These unusually long response times affect network dynamics [5] and are problematic for users. Furthermore, as argued by Tene in his talk about measuring latency [27], one needs to look at extreme percentiles such as 99.995 to determine the latency such that only about 1% of users experience a larger latency during a web session with several page loads. Hence, highly accurate rank approximations are required for items y whose rank is

This is a minor revision of the paper entitled *Relative Error Streaming Quantiles*, published in PODS '21, ISBN 978-1-4503-8381-3/21/06, June 20–25, 2021, Virtual Event, China. DOI: `https://dl.acm.org/doi/10.1145/3452021.3458323`.

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.
Copyright 2022 ACM 0001-0782/08/0X00 ...$5.00.

very large ($n - \mathrm{R}(y) \ll n$); this is precisely the requirement captured by the multiplicative error guarantee.

Achieving multiplicative guarantees is known to be *strictly* harder than additive ones. There are comparison-based additive error algorithms that store just $\Theta(\varepsilon^{-1})$ items for constant failure probability [16], which is optimal. In particular, to achieve additive error, the number of items stored may be independent of the stream length n. In contrast, any algorithm achieving multiplicative error must store $\Omega(\varepsilon^{-1} \cdot \log(\varepsilon n))$ items (see [5, Theorem 2]).

REMARK 1. *Intuitively, the reason additive-error sketches can achieve space independent of the stream length is because they can take a subsample of the stream of size about $\Theta(\varepsilon^{-2})$ and then sketch the subsample. For any fixed item, the additive error to its rank introduced by sampling is at most εn with high probability. When multiplicative error is required, one cannot subsample the input: for low-ranked items, the multiplicative error introduced by sampling will, with high probability, not be bounded by any constant.*

The best known algorithms achieving multiplicative error guarantees are as follows. Zhang et al. [29] give a randomized algorithm storing $O(\varepsilon^{-2} \cdot \log(\varepsilon^2 n))$ universe items. This is essentially a ε^{-1} factor away from the aforementioned lower bound. There is also an algorithm of Cormode et al. [6] that stores $O(\varepsilon^{-1} \cdot \log(\varepsilon n) \cdot \log |\mathcal{U}|)$ items. However, this algorithm requires prior knowledge of the data universe $\mathcal{U}$ (since it builds a binary tree over $\mathcal{U}$), and is inapplicable when $\mathcal{U}$ is huge or even unbounded (e.g., if the data can take arbitrary real values). Finally, Zhang and Wang [28] designed a deterministic algorithm requiring $O(\varepsilon^{-1} \cdot \log^3(\varepsilon n))$ space. Recent work of Cormode and Veselý [8] proves an $\Omega(\varepsilon^{-1} \cdot \log^2(\varepsilon n))$ lower bound for deterministic comparison-based algorithms, which is within a $\log(\varepsilon n)$ factor of Zhang and Wang's upper bound.

Despite both the practical and theoretical importance of multiplicative error (which is arguably an even more natural goal than additive error), there has been no progress on upper bounds, i.e., no new algorithms, since 2007.

In this work, we give a randomized algorithm that maintains the optimal linear dependence on $1/\varepsilon$ achieved by Zhang and Wang, with a significantly improved dependence on the stream length. Namely, we design a one-pass streaming algorithm that given $\varepsilon > 0$, computes a sketch consisting of $O\left(\varepsilon^{-1} \cdot \log^{1.5}(\varepsilon n)\right)$ universe items, from which one can derive rank or quantile estimates satisfying the relative error guarantee with constant probability (see Theorem 1 for a more precise statement). Ours is the first algorithm to be strictly more space efficient than *any* deterministic comparison-based algorithm (owing to the $\Omega(\varepsilon^{-1} \log^2(\varepsilon n))$ lower bound in [8]) and is within an $O(\sqrt{\log(\varepsilon n)})$ factor of the known lower bound for randomized algorithms achieving multiplicative error. Furthermore, it only accesses items through comparisons, i.e., is comparison-based, rendering it suitable, e.g., for floating-point numbers or strings ordered lexicographically. Finally, our algorithm processes the input stream efficiently, namely, its amortized update time is a logarithm of the space bound, i.e., $O\left(\log(\varepsilon^{-1}) + \log\log(n)\right)$.

Mergeability. The ability to merge sketches of different streams to get an accurate sketch for the concatenation of the streams is highly significant both in theory [1] and in practice [25]. Such mergeable summaries enable trivial, automatic parallelization and distribution of processing massive data sets, by splitting the data up into pieces, summarizing each piece separately, and then merging the results in an arbitrary way. We say that a sketch is *fully mergeable* if building it using any sequence of merge operations (executed on singleton items) leads to the same guarantees as if the entire data set had been processed as a single stream.

The following theorem is the main result of this paper. We stress that our algorithm, which we call ReqSketch, does *not* require any advance knowledge about n, the total size of the input, which indeed may not be available in many applications.

THEOREM 1. *Given parameters $0 < \delta \le 0.5$ and $0 < \varepsilon \le 1$, there is a randomized, comparison-based, one-pass streaming algorithm that, when processing a data stream consisting of n items from a totally-ordered universe $\mathcal{U}$, produces a summary S satisfying the following property. Given S, for any $y \in \mathcal{U}$ one can get an estimate $\hat{\mathrm{R}}(y)$ of $\mathrm{R}(y)$ such that*

$$\Pr\left[|\hat{\mathrm{R}}(y) - \mathrm{R}(y)| \ge \varepsilon\, \mathrm{R}(y)\right] < \delta\,,$$

where the probability is over the internal randomness of the streaming algorithm. If $\varepsilon \le \sqrt{\ln\frac{1}{\delta} / \log_2(\varepsilon n)}$, then the size of S in memory words is

$$O\left(\varepsilon^{-1} \cdot \log^{1.5}(\varepsilon n) \cdot \sqrt{\log\left(\frac{1}{\delta}\right)}\right)\,;$$

otherwise, storing S takes $O\left(\log^2(\varepsilon n)\right)$ memory words. Moreover, the summary produced is fully mergeable.

The space bound for the case $\varepsilon \le \sqrt{\ln\frac{1}{\delta} / \log_2(\varepsilon n)}$ certainly applies for values of ε and n encountered in practice (e.g., for $n \le 2^{64}$ and $\delta \le 1/e$, this latter requirement is implied by $\varepsilon \le 1/8$). A straightforward corollary of Theorem 1 is a space-efficient algorithm whose estimates are simultaneously accurate for *all* $y \in \mathcal{U}$ with high probability, while the space complexity increases by a small factor only. This follows from a standard use of the union bound with an epsilon-net argument (using failure probability $\delta' = \varepsilon\delta / \log(\varepsilon n)$).

There is also an alternative analysis of our algorithm (building on an idea from [16]), that shows a space bound of $O\left(\varepsilon^{-1} \cdot \log^2(\varepsilon n) \cdot \log\log(1/\delta)\right)$; note the exponentially better dependence on $1/\delta$, compared to Theorem 1, which, however, comes at the expense of the exponent of $\log(\varepsilon n)$ increasing from 1.5 to 2. This analysis also implies a deterministic space bound of $O(\varepsilon^{-1} \cdot \log^3(\varepsilon n))$, matching the state-of-the-art result in [28].

A proof-of-concept Python implementation of ReqSketch is available at GitHub [17] and a production-quality implementation is available in the DataSketches library [25].

1.1 Technical Overview

A starting point of the design of our algorithm is the KLL sketch [16] that achieves optimal accuracy-space trade-off for the additive error guarantee. The basic building block of the algorithm is a buffer, called a *compactor*, that receives an input stream of n items and outputs a stream of at most $n/2$ items, meant to "approximate" the input stream. The buffer simply stores items and once it is full, we sort the buffer, output all items stored at either odd or even indexes (with odd vs. even selected via an unbiased coin flip), and

clear the contents of the buffer—this is called the *compaction operation*. Note that a randomly chosen half of items in the buffer is simply discarded, whereas the other half of items in the buffer is "output" by the compaction operation.

The overall KLL sketch is built as a sequence of at most $\log_2(n)$ such compactors, such that the output stream of a compactor is treated as the input stream of the next compactor. We thus think of the compactors as arranged into *levels*, with the first one at level 0. Similar compactors were already used, e.g., in [19, 20, 1, 18], and additional ideas are needed to get the optimal space bound for additive error, of $O(1/\varepsilon)$ items stored across all compactors [16].

The compactor building block is not directly applicable to our setting for the following reasons. A first observation is that to achieve the relative error guarantee, we need to always store the $1/\varepsilon$ smallest items. This is because the relative error guarantee demands that estimated ranks for the $1/\varepsilon$ lowest-ranked items in the data stream are *exact*. If even a single one of these items is deleted from the summary, then these estimates will not be exact. Similarly, among the next $2/\varepsilon$ smallest items, the summary must store essentially every other item to achieve multiplicative error. Among the next $4/\varepsilon$ smallest items in the order, the sketch must store roughly every fourth item, and so on.

The following simple modification of the compactor from the KLL sketch indeed achieves the above. Each buffer of size B "protects" the $B/2$ smallest items stored inside, meaning that these items are not involved in any compaction (i.e., the compaction operation only removes the $B/2$ largest items from the buffer). Unfortunately, it turns out that this simple approach requires space $\Theta(\varepsilon^{-2} \cdot \log(\varepsilon^2 n))$, which merely matches the space bound achieved in [29], and in particular has a suboptimal dependence on $1/\varepsilon$.

The key technical contribution of our work is to enhance this simple approach with a more sophisticated rule for selecting the number of protected items in each compaction. In this abstract, we describe a solution choosing this number in each compaction at random from an appropriate geometric distribution. In the full version [4], to get a cleaner analysis, we derandomize this distribution.

While the resulting algorithm is relatively simple, analyzing the error behavior brings new challenges that do not arise in the additive error setting. Roughly speaking, when analyzing the accuracy of the estimate for $\mathrm{R}(y)$ for any fixed item y, all error can be "attributed" to compaction operations. In the additive error setting, one may suppose that *every* compaction operation contributes to the error and still obtain a tight error analysis [16]. Unfortunately, this is not at all the case for relative error: as already indicated, to obtain our accuracy bounds it is essential to show that the estimate for any low-ranked item y is affected by very few compaction operations. Thus, the first step of our analysis is to carefully bound the number of compactions on each level that affect the error for y, using the definition of the geometric distribution in the compaction operation.

Organization of the abstract. We describe our sketch in Section 2; as mentioned above, the algorithm outlined in this abstract is slightly different to the one in the full version [4]. In Section 3, we sketch an analysis of this algorithm in the streaming setting, assuming a foreknowledge of (a polynomial upper bound on) the stream length. Finally, in Section 4 we briefly outline how to analyze the algorithm under merge operations and without any advance knowledge about the stream length as well as adjustments from [7] to make it more efficient in practice. The details can be found in the full version [4].

1.2 Detailed Comparison to Prior Work

Some prior works on streaming quantiles consider queries to be *ranks* $r \in \{1, \ldots, n\}$, and the algorithm must identify an item $y \in \mathcal{U}$ such that $\mathrm{R}(y)$ is close to r; this is called the *quantile query*. In this work, we focus on the dual problem of *rank queries*, where we consider queries to be universe items $y \in \mathcal{U}$ and the algorithm must yield an accurate estimate for $\mathrm{R}(y)$. Unless specified otherwise, algorithms described in this section directly solve both formulations (this holds for our algorithm as well). Algorithms are randomized unless stated otherwise. For simplicity, randomized algorithms are assumed to have constant failure probability δ. All reported space costs refer to the number of universe items stored. (Apart from storing universe items, the algorithms may store, for example, bounds on ranks of stored items or some counters, but the number of such variables is proportional to the number of items stored or even smaller. Thus, the space bounds are in memory words, which can store any item or an integer with $O(\log(n + |\mathcal{U}|))$ bits.)

Additive error. Manku, Rajagopalan, and Lindsay [19, 20] built on the work of Munro and Paterson [22] and gave a deterministic solution storing at most $O(\varepsilon^{-1} \cdot \log^2(\varepsilon n))$ items, assuming prior knowledge of n. Greenwald and Khanna [13] created an intricate deterministic streaming algorithm that stores $O(\varepsilon^{-1} \cdot \log(\varepsilon n))$ items. This is the best known deterministic algorithm for this problem, with a matching lower bound for comparison-based streaming algorithms [8]. Agarwal et al. [1] provided a mergeable sketch of size $O(\varepsilon^{-1} \cdot \log^{1.5}(1/\varepsilon))$. This paper contains many ideas and observations that were used in later work. Felber and Ostrovsky [11] managed to reduce the space complexity to $O(\varepsilon^{-1} \cdot \log(1/\varepsilon))$ items by combining sampling with the Greenwald-Khanna sketches in non-trivial ways. Finally, Karnin, Lang, and Liberty [16] resolved the problem by providing an $O(1/\varepsilon)$-space solution, which is optimal. For general (non-constant) failure probabilities δ, the space upper bound becomes $O(\varepsilon^{-1} \cdot \log\log(1/\delta))$, and they also prove a matching lower bound for comparison-based randomized algorithms, assuming $\delta \leq 1/n!$ (i.e., δ is exponentially small in n).

Multiplicative error. A large number of works sought to provide more accurate quantile estimates for low or high ranks. Only a handful offer solutions to the relative error quantiles problem considered in this work (sometimes also called the biased quantiles problem). Gupta and Zane [14] gave an algorithm for relative error quantiles that stores $O(\varepsilon^{-3} \cdot \log^2(\varepsilon n))$ items, and used this to approximately count the number of inversions in a list; their algorithm requires prior knowledge of the stream length n. As previously mentioned, Zhang et al. [29] presented an algorithm storing $O(\varepsilon^{-2} \cdot \log(\varepsilon^2 n))$ universe items. Cormode et al. [6] designed a deterministic sketch storing $O(\varepsilon^{-1} \cdot \log(\varepsilon n) \cdot \log|\mathcal{U}|)$ items, which requires prior knowledge of the data universe $\mathcal{U}$. Their algorithm is inspired by the work of Shrivastava et al. [26] in the additive error setting and it is also mergeable (see [1, Section 3]). Zhang and Wang [28] gave a deterministic merge-and-prune algorithm storing $O(\varepsilon^{-1} \cdot \log^3(\varepsilon n))$ items, which can handle arbitrary merges with an upper bound on n, and streaming updates for unknown n. However, it does not tackle the most general case of merging without a prior

bound on n. Cormode and Veselý [8] recently showed a space lower bound of $\Omega(\varepsilon^{-1} \cdot \log^2(\varepsilon n))$ items for any deterministic comparison-based algorithm.

Other related works that do not fully solve the relative error quantiles problem are as follows. Manku, Rajagopalan, and Lindsay [20] designed an algorithm that, for a specified number $\phi \in [0, 1]$, stores $O(\varepsilon^{-1} \cdot \log(1/\delta))$ items and can return an item y with $\mathrm{R}(y)/n \in [(1 - \varepsilon)\phi, (1 + \varepsilon)\phi]$ (their algorithm requires prior knowledge of n). Cormode et al. [5] gave a deterministic algorithm that is meant to achieve error properties "in between" additive and relative error guarantees. That is, their algorithm aims to provide multiplicative guarantees only up to some minimum rank k; for items of rank below k, their solution only provides additive guarantees. Their algorithm does not solve the relative error quantiles problem: [29] observed that for adversarial item ordering, the algorithm of [5] requires linear space to achieve relative error for all ranks.

Dunning and Ertl [10, 9] describe a heuristic algorithm called t-digest that is intended to achieve relative error, but they provide no formal accuracy analysis. Indeed, Cormode et al. [7] show that the error of t-digest may be arbitrarily large on adversarially generated inputs. This latter paper also compares t-digest and ReqSketch (i.e., the algorithm of Theorem 1) on randomly generated inputs and proposes implementation improvements for ReqSketch that make it process an input stream faster than t-digest; see Section 4.

Most recently, Masson, Rim, and Lee [21] introduced a new notion of error for quantile sketches (they also refer to their notion as "relative error", but it is quite distinct from the notion considered in this work). They require that for a query percentile $\phi \in [0, 1]$, if y denotes the item in the data stream satisfying $\mathrm{R}(y) = \phi n$, then the algorithm should return an item $\hat{y} \in \mathcal{U}$ such that $|y - \hat{y}| \le \varepsilon \cdot |y|$. This definition only makes sense for data universes with a notion of magnitude and distance (e.g., numerical data), and the definition is not invariant to natural data transformations, such as incrementing every data item y by a large constant. It is also trivially achieved by maintaining a (mergeable) histogram with buckets $((1 + \varepsilon)^i, (1 + \varepsilon)^{i+1}]$. In contrast, the standard notion of relative error considered in this work does not refer to the data items themselves, only to their ranks, and is arguably of more general applicability.

2. DESCRIPTION OF THE ALGORITHM

2.1 The Relative-Compactor Object

The crux of our algorithm is a building block that we call the relative-compactor. Roughly speaking, this object processes a stream of n items and outputs a stream of at most $n/2$ items (each "up-weighted" by a factor of 2), meant to "approximate" the input stream. It does so by maintaining a buffer of limited capacity.

Our complete sketch, described in Section 2.2 below, is composed of a sequence of relative-compactors, where the input of the $(h + 1)$'th relative-compactor is the output of the h'th. With at most $\log_2(\varepsilon n)$ such relative-compactors, n being the length of the input stream, the output of the last relative-compactor is of size $O(1/\varepsilon)$, and hence can be stored in memory.

Compaction operations. The basic subroutine used by our relative-compactor is a compaction operation. The input to a compaction operation is a list X of $2m$ items $x_1 \le x_2 \le \ldots \le x_{2m}$, and the output is a sequence Z of m items. This output is chosen to be one of the following two sequences, uniformly at random: Either $Z = \{x_{2i-1}\}_{i=1}^m$ or $Z = \{x_{2i}\}_{i=1}^m$. That is, the output sequence Z equals either the even or odd indexed items in the sorted order of X, with both outcomes equally probable.

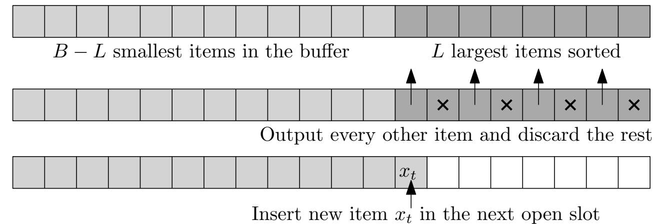

Figure 1: Illustration of the execution of a relative-compactor when inserting a new item x_t into a buffer that is full at time t. See lines 5-13 of Algorithm 1.

Consider an item $y \in \mathcal{U}$ and recall that $\mathrm{R}(y; X) = |\{x \in X \,|\, x \le y\}|$ is the number of items $x \in X$ satisfying $x \le y$ (we remark that both X and $\{x \in X \,|\, x \le y\}$ are multisets of universe items). The following is a trivial observation regarding the error of the rank estimate of y with respect to the input X of a compaction operation when using Z. We view the output Z of a compaction operation (with all items up-weighted by a factor of 2) as an approximation to the input X; for any y, its weighted rank in Z should be close to its rank in X. Observation 2.1 below states that this approximation incurs *zero error* on items that have an even rank in X. Moreover, for items y that have an odd rank in X, the error for $y \in \mathcal{U}$ introduced by the compaction operation is $+1$ or -1 with equal probability.

OBSERVATION 2.1. *A universe item $y \in \mathcal{U}$ is said to be even (odd) w.r.t a compaction operation if* $\mathrm{R}(y; X)$ *is even (odd), where X is the input sequence to the operation. If y is even w.r.t the compaction, then* $\mathrm{R}(y; X) - 2\,\mathrm{R}(y; Z) = 0$. *Otherwise,* $\mathrm{R}(y; X) - 2\,\mathrm{R}(y; Z)$ *is a variable taking a value from* $\{-1, 1\}$ *uniformly at random.*

The observation that items of even rank (and in particular items of rank zero) suffer no error from a compaction operation plays an especially important role in the error analysis of our full sketch.

Full description of the relative-compactor. The complete description of the relative-compactor object is given in Algorithm 1. The high-level idea is as follows. The relative-compactor maintains a buffer of size $B = 2 \cdot k \cdot \lceil \log_2(n/k) \rceil$ where k is an even integer parameter controlling the error and n is the upper bound on the stream length. (In this abstract, we assume that such an upper bound is available; we discuss removing this assumption in Section 4.) The incoming items are stored in the buffer until it is full, and then we perform a compaction operation, as described above.

The input to the compaction operation is not all items in the buffer, but rather the largest L items in the buffer for a parameter $L \le B/2$ such that L is even. These L largest items are then removed from the buffer, and the output of the compaction operation is sent to the output stream of the buffer. This intuitively lets low-ranked items stay in the buffer longer than high-ranked ones. Indeed, the lowest-

	0	1	2	3	4

$B/2$ slots (never compacted) $\lceil \log_2(n/k) \rceil = 5$ sections of k slots

Figure 2: Illustration of a relative-compactor and its sections, together with the indexes of the sections.

Algorithm 1 Relative-Compactor

```
Input: Parameters k ∈ 2ℕ+ and n ∈ ℕ+, and a stream of
    items x_1, x_2, ... of length at most n
 1: Set m = ⌈log_2(n/k)⌉                      ▷ Number of sections
 2: Set B = 2 · k · m                             ▷ Buffer size
 3: Initialize an empty buffer 𝓑 of size B, indexed from 1
 4: for t = 1 ... do
 5:     if 𝓑 is full then                   ▷ Compaction operation
 6:         Randomly pick section I ∈ {0, ..., m − 1} with
    Pr[I = i] = p_i := 2^i/(2^m − 1)
 7:         Set L = i · k and S = B − L + 1
 8:         Pivot 𝓑 s.t. the largest L items occupy 𝓑[S : B]
 9:                         ▷ 𝓑[S : B] are the last L slots of 𝓑
10:         Sort 𝓑[S : B] in non-descending order
11:         Output either even or odd indexed items in the
    range 𝓑[S : B] with equal probability
12:         Mark slots 𝓑[S : B] in the buffer as clear
13:     Store x_t into the next available slot in the buffer 𝓑.
```

ranked half of items in the buffer are *never* removed. We show later that this facilitates the relative error guarantee.

The crucial part in the design of Algorithm 1 is to select the parameter L in a right way, as L controls the number of items compacted each time the buffer is full. If we were to set $L = B/2$ for all compaction operations, then analyzing the worst-case behavior reveals that we need $B \approx 1/\varepsilon^2$, resulting in a sketch with a quadratic dependency on $1/\varepsilon$. To achieve the linear dependency on $1/\varepsilon$, we choose the parameter L via a suitable geometric distribution subject to the constraint that $L \leq B/2$.

In more detail, during each compaction operation, the second half of the buffer (with $B/2$ largest items) is split into $m := \lceil \log_2(n/k) \rceil$ sections, each of size k and indexed $0, \ldots, m-1$ from the left; see Figure 2. The first section involved in the compaction is selected with probability exponentially increasing with its index, namely, section $i \in \{0, \ldots, m-1\}$ is chosen with probability $p_i := 2^i \cdot \gamma$, where $\gamma = 1/(2^m - 1)$ is chosen so that the probabilities p_i sum to 1. Furthermore, the chosen geometric distribution has the following property:

$$p_i = \sum_{j=0}^{i-1} p_j + \gamma\,. \tag{1}$$

That is, the probability of starting the compaction in section i is essentially equal to the probability of also compacting section $i-1$. Since $B = 2 \cdot k \cdot m$ and since at most $m \cdot k$ largest items are involved in the compaction, the smallest $B/2$ items in the buffer are never removed.

Remark 2. *In the original version of the paper, we describe the algorithm with a* derandomized *geometric distribution. Thus, that version of the algorithm uses randomness only to select which items to place in the output stream, not how many items to compact. This leads to a cleaner analysis and isolates the one component of the algorithm for which randomness is essential. In this abstract, we have chosen not to derandomize the geometric distribution for simplicity.*

Algorithm 2 ReqSketch (Relative-Error Quantiles sketch)

```
Input: Parameters k ∈ 2ℕ+ and n ∈ ℕ+, and a stream of
    items x_1, x_2, ... of length at most n
Output: A sketch answering rank (and quantile) queries
 1: Let RelCompactors be a list of relative-compactors
 2: Set H = 0 and initialize relative-compactor with param-
    eters k and n at RelCompactors[0]
 3: for t = 1 ... do
 4:     INSERT(x_t, 0)
 5: function INSERT(x,h)
 6:     if H < h then
 7:         Set H = h and initialize relative-compactor with
    parameters k and n at RelCompactors[h]
 8:     Insert item x into RelCompactors[h]
 9:     for z in output stream of RelCompactors[h] do
10:         INSERT(z, h + 1)              ▷ Items output by the
    compaction (if any)
11: function ESTIMATE-RANK(y)
12:     Set R̂(y) = 0
13:     for h = 0 to H do
14:         for each item y′ ≤ y in RelCompactors[h] do
15:             Increment R̂(y) by 2^h
        return R̂(y)
```

2.2 The Full Sketch

Following prior work [19, 1, 16], the full sketch uses a sequence of relative-compactors. At the very start of the stream, it consists of a single relative-compactor (at level 0) and opens a new one (at level 1) once items are fed to the output stream of the first relative-compactor (i.e., after the first compaction operation, which occurs on the first stream update during which the buffer is full). In general, when the newest relative-compactor is at level h, the first time the buffer at level h performs a compaction operation (feeding items into its output stream for the first time), we open a new relative-compactor at level $h+1$ and feed it these items. Algorithm 2 describes the logic of this sketch.

To answer rank queries, we use the items in the buffers of the relative-compactors as a weighted coreset. That is, the union of these items is a weighted set $\mathcal{C}$ of items, where the weight of items in relative-compactor at level h is 2^h (recall that h starts from 0), and the approximate rank of y, denoted $\hat{\mathrm{R}}(y)$, is the sum of weights of items in $\mathcal{C}$ smaller than or equal to y. Similarly, ReqSketch can answer quantile queries, i.e., for a given rank $r \in \{1, \ldots, n\}$, return an item $y \in \mathcal{U}$ with $\mathrm{R}(y)$ close to r; the algorithm just returns an item y stored in one of the relative-compactors with $\hat{\mathrm{R}}(y)$ closest to the query rank r among all items in the sketch.

The construction of layered exponentially-weighted compactors and the subsequent rank estimation is virtually identical to that explained in prior works [19, 1, 16]. Our essential departure from prior work is in the definition of the compaction operation, not in how compactors are plumbed together to form a complete sketch.

Merge operation. The merge operation takes as input two sketches S' and S'' which have processed two separate streams

σ' and σ'' and outputs a sketch S summarizing the concatenated stream $\sigma = \sigma' \circ \sigma''$ (the order of σ' and σ'' does not matter here). For the simplified sketch presented in this abstract, merging two sketches is straightforward: At each level, concatenate the buffers and if that causes the capacity of the compactor to be reached or exceeded, perform the compaction operation, as in Algorithm 1. There are additional complications when an upper bound n on the combined input size is not available in advance; for instance, the number of sections m (and the section size k) may need to be adjusted during a merge operation. These details are described in the full version [4].

3. ANALYSIS

We provide a sketch of the analysis in the streaming setting, assuming a foreknowledge of (an upper bound on) the stream length n. To analyze the error of the full sketch, we focus on the error in the estimated rank of an arbitrary fixed item $y \in \mathcal{U}$. Let $\mathrm{R}(y)$ be the rank of item y in the input stream, and let $\mathrm{Err}(y) = \hat{\mathrm{R}}(y) - \mathrm{R}(y)$ be the error of the estimated rank for y.

Analysis of the relatively-compactor. We first restrict our attention to any single relative-compactor at level h (Algorithm 1) maintained by our sketching algorithm (Algorithm 2), and we use "time t" to refer to the t'th insertion operation to this particular relative-compactor. We analyze the error introduced by the relative-compactor for item y. Specifically, at time t, let $X^t = (x_1, \ldots, x_t)$ be the prefix of the input stream to the relative-compactor, Z^t be the output stream, and $\mathcal{B}^t$ be the items in the buffer after inserting item x_t. The rank error made by the relative-compactor at time t with respect to item y is defined as

$$\mathrm{Err}_h^t(y) = \mathrm{R}(y; X^t) - 2\,\mathrm{R}(y; Z^t) - \mathrm{R}(y; \mathcal{B}^t). \qquad (2)$$

Conceptually, $\mathrm{Err}_h^t(y)$ tracks the difference between y's rank in the input stream X^t at time t versus its rank as estimated by the combination of the output stream and the remaining items in the buffer at time t (output items are upweighted by a factor of 2 while items remaining in the buffer are not). We denote the overall error of the relative-compactor by $\mathrm{Err}_h(y)$ and the total number of items $x \le y$ inserted to the level-h buffer by $\mathrm{R}_h(y)$. Then $\mathrm{Err}_h(y) = \mathrm{R}_h(y) - 2\,\mathrm{R}_{h+1}(y) - \mathrm{R}(y; \mathcal{B}_h)$, where $\mathcal{B}_h$ is the level-h buffer after Algorithm 2 has processed the input stream (note that $\mathrm{R}_{h+1}(y)$ is the number of items $x \le y$ in the output stream of the level-h relative-compactor). To bound $\mathrm{Err}_h(y)$, we keep track of the error associated with y over time, and define the increment or decrement of it as

$$\Delta_h^t(y) = \mathrm{Err}_h^t(y) - \mathrm{Err}_h^{t-1}(y),$$

where $\mathrm{Err}_h^0(y) = 0$.

Clearly, if the algorithm performs no compaction operation in a time step t, then $\Delta_h^t(y) = 0$. (Recall that a compaction is an execution of lines 6-12 of Algorithm 1.) Let us consider what happens in a step t in which a compaction operation occurs. Observation 2.1 shows that if y is even with respect to the compaction, then y suffers no error, meaning that $\Delta_h^t(y) = 0$. Otherwise, $\Delta_h^t(y)$ is uniform in $\{-1, 1\}$.

It follows that $\mathbb{E}[\mathrm{Err}_h(y)] = 0$, which implies that the estimator $\hat{\mathrm{R}}(y)$ is unbiased. To bound the variance of $\mathrm{Err}_h(y)$, we analyze $\mathrm{Var}[\Delta_h^t(y)]$ for any step t with a compaction operation. We suppose that $\mathrm{R}(y; \mathcal{B}^t) > B/2$ as otherwise, no item $x \le y$ is removed from the buffer and $\Delta_h^t(y) = 0$. Below, we only focus on steps t with a compaction operation such that $\mathrm{R}(y; \mathcal{B}^t) > B/2$. Let $i_t \in \{0, \ldots, m-1\}$ be the largest index of a section of $\mathcal{B}^t$ with an item $x \le y$, i.e., i_t is the smallest integer $i \ge 0$ such that $\mathrm{R}(y; \mathcal{B}^t) \le B/2 + (i+1) \cdot k$. Note that if Algorithm 1 draws $I < i_t$ in line 6 during the compaction in step t, at least k items $x \le y$ are removed from the buffer, which implies that the number of steps t with $I < i_t$ is at most $\mathrm{R}_h(y)/k$. Using this observation, our aim is to bound $\mathrm{Var}[\mathrm{Err}_h(y)]$ in terms of $\mathrm{R}_h(y)/k$.

The probability of $\Delta_h^t(y) \ne 0$ is at most the probability of removing an item $x \le y$, which by the definition of i_t and Algorithm 1 equals

$$\sum_{j=0}^{i_t} p_j = \sum_{j=0}^{i_t - 1} p_j + p_{i_t} = 2 \sum_{j=0}^{i_t - 1} p_j + \gamma\,, \qquad (3)$$

where we use (1); this is where we rely on using the geometric distribution to choose the number of sections involved in the compaction (recall that $\gamma = 1/(2^m - 1)$ is small). Using (3) and that $\Delta_h^t(y) \in \{-1, 0, 1\}$, we get that $\mathrm{Var}[\Delta_h^t(y)] = \mathbb{E}[(\Delta_h^t(y))^2] = \Pr[\Delta_h^t(y) \ne 0] \le 2 \sum_{i=0}^{i_t - 1} p_i + \gamma$.

Since the variables $\Delta_h^t(y)$ are independent,

$$\mathrm{Var}[\mathrm{Err}_h(y)] = \sum_t \mathrm{Var}[\Delta_h^t(y)] = \sum_t \left(2 \sum_{i=0}^{i_t - 1} p_i + \gamma \right).$$

As observed above, the number of steps t with $I < i_t$ in line 6 is at most $\mathrm{R}_h(y)/k$, which implies that $\sum_t \sum_{i=0}^{i_t - 1} p_i \le \mathrm{R}_h(y)/k$ (we omit a formal proof of this observation).

As a compaction is performed at most once in every k steps, we have that $\sum_t \gamma \le \gamma \cdot n/k \le n/(n-k) \le 2$, using the definition of $\gamma = 1/(2^m - 1)$ and $m = \lceil \log_2(n/k) \rceil$; the last step uses $n \ge 2k$ as otherwise the whole input to the relative-compactor would be stored in the buffer which has size at least $2k$ and there would be no compaction operation.

Summarizing, we have obtained the following bound on the variance of the error at level h:

$$\mathrm{Var}[\mathrm{Err}_h(y)] \le \frac{2\,\mathrm{R}_h(y)}{k} + 2 \le \frac{4\,\mathrm{R}_h(y)}{k}\,, \qquad (4)$$

where the second inequality uses $\mathrm{R}_h(y) \ge k$ as otherwise there would be no level-h compaction operation affecting the error for y, i.e., $\mathrm{Err}_h(y) = 0$ if $\mathrm{R}_h(y) < k$

Analysis of the full sketch in the streaming setting. To make the variance bound for a single level in (4) useful, we show that $\mathrm{R}_h(y)$ roughly halves with every level. This is easy to see in expectation, and it holds with high probability up to a certain crucial level $H(y)$. Here, we define $H(y)$ to be the minimal h for which $2^{-h+1}\,\mathrm{R}(y) \le B/2$. For $h = H(y) - 1$ (assuming $H(y) > 0$), we particularly have $2^{2-H(y)}\,\mathrm{R}(y) > B/2$, or equivalently, $2^{H(y)} < 2^3 \cdot \mathrm{R}(y)/B$.

LEMMA 3.1. *With probability at least $1 - \delta$ it holds that $\mathrm{R}_h(y) \le 2^{-h+1}\,\mathrm{R}(y)$ for any $h \le H(y)$.*

We omit the proof by induction, using Chernoff bounds (which also requires choosing k as described below). In what follows, we condition on the bound on $\mathrm{R}_h(y)$ in Lemma 3.1 for any $h \le H(y)$. For $h = H(y)$, we thus have that $\mathrm{R}_{H(y)}(y) \le 2^{-H(y)+1}\,\mathrm{R}(y) \le B/2$ and that $\mathrm{R}_h(y) = 0$ for any $h > H(y)$. Thus, Observation 2.1 implies that $\mathrm{Err}_h(y) = 0$ for any $h \ge H(y)$.

We are now ready to bound the overall error of the sketch for item y, i.e., $\mathrm{Err}(y) = \hat{\mathrm{R}}(y) - \mathrm{R}(y)$ where $\hat{\mathrm{R}}(y)$ is the estimated rank of y. By the observations above, we get

$$\mathrm{Err}(y) = \sum_{h=0}^{H(y)-1} 2^h \, \mathrm{Err}_h(y) \, .$$

Recall that a zero-mean random variable X with variance σ^2 is sub-Gaussian if $\mathbb{E}[\exp(sX)] \leq \exp(-\frac{1}{2} \cdot s^2 \cdot \sigma^2)$ for any $s \in \mathbb{R}$; note that a weighted sum of independent zero-mean sub-Gaussian variables is a zero-mean sub-Gaussian random variable as well. By the analysis of level h, $\mathrm{Err}_h(y)$ is a zero-mean sub-Gaussian variable with $\mathrm{Var}[\mathrm{Err}_h(y)] \leq 4\,\mathrm{R}_h(y)/k$. It follows that $\mathrm{Err}(y)$ is a zero-mean sub-Gaussian random variable with variance

$$\begin{aligned}\sum_{h=0}^{H(y)-1} 2^{2h}\, \mathrm{Var}[\mathrm{Err}_h(y)] &\leq \sum_{h=0}^{H(y)-1} 2^{2h} \cdot \frac{4\,\mathrm{R}_h(y)}{k} \\ &\leq \sum_{h=0}^{H(y)-1} 2^{h+3} \cdot \frac{\mathrm{R}(y)}{k} \\ &\leq 2^{H(y)+3} \cdot \frac{\mathrm{R}(y)}{k} \leq 2^6 \cdot \frac{\mathrm{R}(y)^2}{k \cdot B} \, ,\end{aligned}$$

where the second inequality is due to Lemma 3.1 and the last inequality follows from the definition of $H(y)$.

Given the desired accuracy ε and the desired upper bound δ on failure probability, we choose k so that $\mathrm{Var}[\mathrm{Err}(y)] \leq \varepsilon^2\,\mathrm{R}(y)^2/\ln(1/\delta)$, or equivalently that $k \cdot B = k \cdot 2 \cdot k \cdot \lceil \log_2(n/k) \rceil \geq \Theta\left(\varepsilon^{-2} \cdot \ln(1/\delta)\right)$; this holds for k satisfying $k = \Theta\left(\varepsilon^{-1} \cdot \sqrt{\ln \frac{1}{\delta} / \log_2(n/k)}\right)$. Then using standard (Chernoff) tail bounds for sub-Gaussian variables concludes the calculation of the failure probability.

Finally, for the space bound, by the choice of k above we have that $B = O\left(\varepsilon^{-1} \cdot \sqrt{\log(\varepsilon n) \cdot \log(1/\delta)}\right)$, and it thus remains to observe that the number of relative-compactors ever created by Algorithm 2 is at most $O(\log(\varepsilon n))$. Indeed, each item on level h has weight 2^h, so there are at most $n/2^h$ items inserted to the buffer at that level. For $h = \lceil \log_2(n/B) \rceil$, we get that on this level, there are fewer than B items inserted to the buffer, which is consequently not compacted, so the highest level has index at most $\lceil \log_2(n/B) \rceil$. (In the case $\varepsilon > \sqrt{\ln \frac{1}{\delta} / \log_2(\varepsilon n)}$, we have that $k = O(1)$ and $B = O(\log(\varepsilon n))$.)

4. ANALYSIS EXTENSIONS

We discuss how to extend the analysis presented above to more general settings. We also outline important practical optimizations that are included in the ReqSketch implementation in the Apache DataSketches library [25].

Handling unknown stream lengths. In previous sections, we outlined a proof of Theorem 1 in the streaming setting assuming that (an upper bound on) n is known, where n is the true stream length. The space usage of the algorithm grows polynomially with the logarithm of this upper bound, so if this upper bound is at most n^c for some constant $c \geq 1$, then the space usage of the algorithm will remain as stated in Theorem 1, up to a constant factor.

In the case that such a polynomial upper bound on n is not known, we modify the algorithm slightly, and start with an initial estimate N_0 of n, namely, $N_0 = \Theta(\varepsilon^{-1})$. That is, we begin by running Algorithm 2 with parameters k and N_0. As soon as the stream length hits the current estimate N_i, the algorithm "closes out" the current data structure and continues to store it in "read only" mode, while initializing a new summary based on the estimated stream length of $N_{i+1} = N_i^2$ (i.e., we execute Algorithm 2 with parameters k and N_{i+1}). This process occurs at most $\log_2 \log_2(\varepsilon n)$ many times, before the guess is at least the true stream length n. At the end of the stream, the rank of any item y is estimated by summing the estimates returned by each of the at most $\log_2 \log_2(\varepsilon n)$ summaries stored by the algorithm. A simple extension of the analysis presented above shows that the accuracy-space tradeoff from Theorem 1 holds for this algorithm. Nevertheless, in a practical implementation, we suggest not to close out the current summary and only recompute the parameters of each buffer as described below.

Full mergeability. There are substantial additional technical difficulties to analyze the algorithm under an arbitrary sequence of merge operations, especially with no foreknowledge of the total size of the input. Most notably, when the input size is not known in advance, the parameters of k and B of each relative-compactor must change as more inputs are processed. This makes obtaining a tight bound on the variance of the resulting estimates highly involved. In particular, as a sketch processes more and more inputs, it protects more and more items, which means that items appearing early in the stream may *not* be protected by the sketch, even though they *would have been protected* if they appeared later in the stream (this is because the buffer size increases as the summarized input size grows). As mentioned above, addressing this issue is reasonably simple in the streaming setting. However, that simple approach does not work for a general sequence of merge operations, and the full version [4] contains a much more intricate analysis to give a fully mergeable summary.

Practical adjustments of ReqSketch. The description of our algorithm in Section 2.2 is suitable for a mathematical analysis, however, there are several ways how to improve its efficiency, which are described in [7]. First, we observe that the adjustments of the KLL sketch proposed by Ivkin et al. [15] are applicable to ReqSketch as well, for example, we can allow the buffer to exceed its capacity provided that the overall capacity of the sketch is satisfied; this can be viewed as laziness in performing compaction operations.

The main difference between the KLL sketch and ReqSketch is the use of the geometric distribution to choose the number of compacted items, and this requires the buffer size to depend on the input size n. Instead of setting the buffer size based on an upper bound on n, it is sufficient to count the number C_h of compaction operations at each level h and set the level-h buffer size to $O(\varepsilon^{-1} \cdot \sqrt{\log C_h})$, recomputing the buffer size only when $\log_2 C_h$ increases by a factor of 2; then we also decrease the section size k by a factor of $\sqrt{2}$. With this adjustment, the buffer size will be *non-increasing* in the level h. More importantly, Cormode et al. [7] observed that the aforementioned laziness proposed in [15] should be restricted to level 0 only (which has the largest size). In other words, when we compact level 0, we perform the compaction operation at any other level in which the buffer exceeds its capacity. Somewhat surprisingly, this "partial laziness" significantly decreases the observed update time (averaged over processing the whole stream), compared to the original proposal from [15].

5. CONCLUSIONS

For constant failure probability δ, we show an $O(\varepsilon^{-1} \cdot \log^{1.5}(\varepsilon n))$ space upper bound for relative error quantile approximation over data streams. Our algorithm is provably more space-efficient than any deterministic comparison-based algorithm, and is within an $O(\sqrt{\log(\varepsilon n)})$ factor of the known lower bound for randomized algorithms (even non-streaming algorithms). The sketch output by our algorithm is fully mergeable, with the same accuracy-space trade-off as in the streaming setting, rendering it suitable for a parallel or distributed environment. The main open question is to close the aforementioned $O(\sqrt{\log(\varepsilon n)})$-factor gap.

6. ACKNOWLEDGMENTS

The research is performed in close collaboration with DataSketches [25], the Apache open source project for streaming data analytics. G. Cormode and P. Veselý were supported by European Research Council grant ERC-2014-CoG 647557. P. Veselý was also partially supported by the project 19-27871X of GA ČR and by Charles University project UNCE/SCI/004. J. Thaler was supported by NSF SPX award CCF-1918989 and NSF CAREER award CCF-1845125.

7. REFERENCES

[1] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Transactions on Database Systems*, 38(4):26, 2013.

[2] R. Agrawal and A. Swami. A one-pass space-efficient algorithm for finding quantiles. In *COMAD-95, Pune, India*, 1995.

[3] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS '04*, pages 286–296. ACM, 2004.

[4] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Veselý. Relative error streaming quantiles. *arXiv preprint arXiv:2004.01668*, 2020.

[5] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. In *ICDE '05*, pages 20–31, Washington, DC, USA, 2005. IEEE Computer Society.

[6] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS '06*, pages 263–272. ACM, 2006.

[7] G. Cormode, A. Mishra, J. Ross, and P. Veselý. Theory meets practice at the median: A worst case comparison of relative error quantile algorithms. In *KDD '21*, page 2722–2731, New York, NY, USA, 2021. ACM.

[8] G. Cormode and P. Veselý. A tight lower bound for comparison-based quantile summaries. In *PODS '20*, pages 81–93, New York, NY, USA, 2020. ACM.

[9] T. Dunning. The t-digest: Efficient estimates of distributions. *Software Impacts*, 7:100049, 2021.

[10] T. Dunning and O. Ertl. Computing extremely accurate quantiles using t-digests. *CoRR*, abs/1902.04023, 2019.

[11] D. Felber and R. Ostrovsky. A randomized online quantile summary in O(1/epsilon * log(1/epsilon)) words. In *APPROX/RANDOM '15*, volume 40 of *LIPIcs*, pages 775–785, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] S. Ganguly. A nearly optimal and deterministic summary structure for update data streams. *arXiv preprint cs/0701020*, 2007.

[13] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, volume 30, pages 58–66. ACM, 2001.

[14] A. Gupta and F. X. Zane. Counting inversions in lists. In *SODA '03*, pages 253–254, Philadelphia, PA, USA, 2003. SIAM.

[15] N. Ivkin, E. Liberty, K. Lang, Z. Karnin, and V. Braverman. Streaming quantiles algorithms with small space and update time. *arXiv preprint arXiv:1907.00236*, 2019.

[16] Z. Karnin, K. Lang, and E. Liberty. Optimal quantile approximation in streams. In *FOCS '16*, pages 71–78. IEEE, 2016.

[17] E. Liberty and P. Veselý. relativeErrorSketch.py. In `https://github.com/edoliberty/streaming-quantiles/`, 2021.

[18] G. Luo, L. Wang, K. Yi, and G. Cormode. Quantiles over data streams: Experimental comparisons, new analyses, and further improvements. *The VLDB Journal*, 25(4):449–472, Aug. 2016.

[19] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM SIGMOD Record*, volume 27, pages 426–435. ACM, 1998.

[20] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *ACM SIGMOD Record*, volume 28, pages 251–262. ACM, 1999.

[21] C. Masson, J. E. Rim, and H. K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *PVLDB*, 12(12):2195–2205, 2019.

[22] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[23] I. Pohl. *A minimum storage algorithm for computing the median.* IBM TJ Watson Research Center, 1969.

[24] V. Poosala, V. Ganti, and Y. E. Ioannidis. Approximate query answering using histograms. *IEEE Data Eng. Bull.*, 22(4):5–14, 1999.

[25] L. Rhodes, K. Lang, J. Malkin, A. Saydakov, E. Liberty, and J. Thaler. DataSketches: A library of stochastic streaming algorithms. Open source software: `https://datasketches.apache.org/`, 2013.

[26] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04*, pages 239–249. ACM, 2004.

[27] G. Tene. How NOT to measure latency. `https://www.youtube.com/watch?v=lJ8ydIuPFeU`, 2015.

[28] Q. Zhang and W. Wang. An efficient algorithm for approximate biased quantile computation in data streams. In *CIKM '07*, pages 1023–1026, 2007.

[29] Y. Zhang, X. Lin, J. Xu, F. Korn, and W. Wang. Space-efficient relative error order sketch over data streams. In *ICDE '06*, pages 51–51. IEEE, 2006.