# Optimal quantile estimation: beyond the comparison model

Meghal Gupta
*EECS*
*University of California, Berkeley*
Berkeley, US
meghal@berkeley.edu

Mihir Singhal
*EECS*
*University of California, Berkeley*
Berkeley, US
mihirs@berkeley.edu

Hongxun Wu
*EECS*
*University of California, Berkeley*
Berkeley, US
wuhx@berkeley.edu

2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS) | 979-8-3315-1674-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/FOCS61266.2024.00075

***Abstract*—Estimating quantiles is one of the foundational problems of data sketching. Given $n$ elements $x_1, x_2, \ldots, x_n$ from some universe of size $U$ arriving in a data stream, a *quantile sketch* estimates the rank of any element with additive error at most $\varepsilon n$. A low-space algorithm solving this task has applications in database systems, network measurement, load balancing, and many other practical scenarios.**

**Current quantile estimation algorithms described as optimal include the GK sketch (Greenwald and Khanna 2001) using $O(\varepsilon^{-1} \log n)$ words (deterministic) and the KLL sketch (Karnin, Lang, and Liberty 2016) using $O(\varepsilon^{-1} \log \log(1/\delta))$ words (randomized, with failure probability $\delta$). However, both algorithms are only optimal in the comparison-based model, whereas many typical applications involve streams of integers that the sketch can use aside from making comparisons.**

**If we go beyond the comparison-based model, the deterministic q-digest sketch (Shrivastava, Buragohain, Agrawal, and Suri 2004) achieves a space complexity of $O(\varepsilon^{-1} \log U)$ words, which is incomparable to the previously-mentioned sketches. It has long been asked whether there is a quantile sketch using $O(\varepsilon^{-1})$ words of space (which is optimal as long as $n \leq \text{poly}(U)$). In this work, we present a *deterministic* algorithm using $O(\varepsilon^{-1})$ words, resolving this line of work.**

***Index Terms*—streaming algorithm, quantiles, sketching**

## I. Introduction

Estimating basic statistics such as the mean, median, minimum/maximum, and variance of large datasets is a fundamental problem of wide practical interest. Nowadays, the massive amount of data often exceeds the memory capacity of the algorithm. This is captured by *streaming model*: The bounded-memory algorithm makes one pass over the data stream $x_1, x_2, \ldots, x_n$ from a universe $[U] = \{1, \ldots, U\}$ and, in the end, outputs the statistic of interest. The memory state of the algorithm is therefore a *sketch* of the data set that contains the information about the statistic and allows future insertions. Here, memory consumption is conventionally measured in units of words, where 1 word equals $\log n + \log U$ bits.

This work is supported by NSF GRFP Fellowship, Jelani Nelson's NSF award CCF-2427808, Venkatesan Guruswami's Simons Investigator award, Avishay Tal's Sloan Research Fellowship and NSF CAREER Award CCF-2145474.

Most of these simple statistics can be computed exactly with a constant number of words. But the median, or more generally, the $\phi$-quantile, is one exception. In their pioneering paper, Munro and Paterson [1] showed that even an algorithm that makes $p$ passes over the data stream still needs $\Omega(n^{1/p})$ space to find the median. Fortunately, for many practical applications, it suffices to find the $\varepsilon$-approximate $\phi$-quantile: Instead of outputting the element of rank exactly $\phi n$, the algorithm only has to output an element of rank $(\phi \pm \varepsilon)n$. Such algorithms are called *approximate quantile sketches*. They are actually implemented in practice, appearing in systems or libraries such as Spark-SQL [2], the Apache DataSketches project [3], GoogleSQL [4], and the popular machine learning library XGBoost [5].

There are also other queries the sketch could need to answer: For example, online queries asked in the middle of the stream, or rank queries, where the algorithm is asked to estimate the rank of an element up to $\varepsilon n$ error. As finding approximate quantiles is equivalent to answering rank queries. To solve all of them, it suffices to solve the following strongest definition.

**Problem I.1** (Quantile sketch)**.** *The problem of quantile sketching (or specifically, $\varepsilon$-approximate quantile sketching) is to find a data structure $A$ taking as little space as possible in order to solve the following problem: Given a stream of elements $\pi = x_1, x_2, \ldots, x_n \in [U]$, we define the partial stream $\pi_t = x_1, x_2, \ldots, x_t$. For element $x \in [U]$, let $\text{rank}_{\pi_t}(x)$ be be the number of elements in $\pi_t$ that are at most $x$. When a query $x$ arrives at time $t$, then $A$ must output an approximate rank $r$, such that $|r - \text{rank}_{\pi_t}(x)| \leq \varepsilon t$.*

Two notable quantile sketches include the Greenwald and Khanna (GK) sketch [6] using $O(\varepsilon^{-1} \log n)$ words (deterministic) and KLL sketch [7] using $O(\varepsilon^{-1} \log \log(1/\delta))$ words (randomized, with failure probability $\delta$). Both algorithms follow the comparison-based paradigm, where the sketch cannot see anything about the elements themselves and can only make black-box comparisons between elements it has stored. They are known to be optimal in this paradigm ( [8] shows the GK is optimal for deterministic algorithms and [7] shows that KLL is optimal for randomized algorithms).

2575-8454/24/$31.00 ©2024 IEEE
DOI 10.1109/FOCS61266.2024.00075

However, many applications of quantile sketches apply to streams of integers (or elements of some finite universe), rather than just to black-box comparable objects. For example, the elements in the universe could be one of the following: network response times (with a preset timeout), IP addresses, file sizes, or any other data with fixed precision. This may allow for a better quantile sketch than in the comparison-based model. The best previously-known non-comparison-based algorithm is the q-digest sketch introduced in [9], which is a deterministic sketch using $O(\varepsilon^{-1}\log U)$ words. Unfortunately, this isn't really better than the GK sketch, as $n$ is typically much less than $\mathrm{poly}(U)$. On the other hand, the only lower bound we know is the trivial lower bound of $\Omega(\varepsilon^{-1})$ words in the regime where $n \leq \mathrm{poly}(U)$ (which holds for both deterministic and randomized algorithms). Motivated by this gap, Greenwald and Khanna, in their survey [10], asked if the q-digest algorithm is already optimal, and as such, one cannot substantially improve upon comparison-based sketches.

In this work, we resolve this question fully and provide a deterministic quantile sketch that uses the optimal $O(\varepsilon^{-1})$ words. This is the first quantile sketch that goes beyond the comparison-based lower bound (in the natural regime of $n \leq \mathrm{poly}(U)$) and is the first direct improvement on the q-digest sketch in the 20 years since it was proposed. (See Table I for a detailed comparison.)

**Theorem I.2.** *There exists a deterministic streaming algorithm for Problem I.1 using $O(\varepsilon^{-1})$ words (more specifically, $O(\varepsilon^{-1}(\log(\varepsilon n)+\log(\varepsilon U)))$ bits) of space[1].*

Our sketch uses less space than not only the deterministic q-digest and GK sketches but also the randomized KLL sketch, when compared in words. Note that randomized algorithms, like KLL sketch, have failure probabilities and retain their theoretical guarantee only against non-adaptive adversaries. The fact that our algorithm is deterministic provides stronger robustness. As these sketches are already implemented in practice, we hope that our algorithm can help improve the performance of these libraries.

### A. Discussion and further directions

*a) Optimality of our algorithm:* As we discussed earlier, the quantile sketch lower bound of $\Omega(\varepsilon^{-1})$ words only holds in the regime where $n \leq \mathrm{poly}(U)$. However, we conjecture that our algorithm is optimal in general for deterministic algorithms. Specifically, there is a simple example showing any sketch for Problem I.1 requires at least $\varepsilon^{-1}\log(\varepsilon U)$ bits (see Section VII), but we also need to show a lower bound of $\varepsilon^{-1}\log(\varepsilon n)$ bits. We make the following conjecture about deterministic parallel counting, which would imply our lower bound because any algorithm for Problem I.1 can also solve the $k$-parallel counters problem for $k = \Theta(1/\varepsilon)$.

[1]Here, technically, when we write $\log(\varepsilon n)$ and $\log(\varepsilon U)$, it really should be $\max\{\log(\varepsilon n), 1\}, \max\{\log(\varepsilon U), 1\}$ to avoid the uninteresting corner cases.

**Conjecture I.3** (Deterministic parallel counters)**.** *We define the $k$-parallel counters problem as following: There are $k$ counters initiated to 0. Given a stream of increments $i_1, i_2, \ldots, i_n \in [k]$ where $i_t$ means to increment the $i_t$-th counter by 1, the algorithm has to output the final count of each counter up to an additive error of $n/k$.*

*We conjecture that any deterministic algorithm for this problem requires at least $\Omega(k\log(n/k))$ bits of memory.* [2]

This conjecture essentially says that to maintain $k$ counters in parallel, one needs to maintain each counter independently. Aden-Ali, Han, Nelson, and Yu [11] studied this problem for randomized algorithms. We note that our conjecture is resolved in a follow-up paper by Wang [12]. Thus proving the optimality of our algorithm.

*b) Improvements in the randomized setting:* Deterministic algorithms are used at the heart of the randomized ones. Many randomized algorithms (including the algorithm by Felber and Ostrovsky [13], the KLL sketch [7], and the mergeable summary of [14]) follow the paradigm of first sampling a number of elements from the stream and then maintaining them with a careful combination of deterministic sketches.

As long as $n \leq \mathrm{poly}(U)$, our algorithm is optimal even in the randomized setting, but when this condition is not met, it is possible to do better in the randomized setting. If $n$ is known in advance, one can simply sample $\frac{\log 1/\delta}{\varepsilon^2}$ elements and feeds them into our sketch.[3] It uses a memory of $O(\varepsilon^{-1}(\log\log(1/\delta) + \log U) + \log n)$ bits, which strictly improves that of the KLL sketch. We note that, in the most common regime where $\delta > 1/2^{\varepsilon n}$, there is a $\Omega(\varepsilon^{-1}(\log\log(1/\delta) + \log \varepsilon U))$-bit lower bound for streaming quantile sketches.[4] So our algorithm is also very close to optimal in the randomized setting as well.

*c) Finding a simpler algorithm:* Although our basic construction is relatively simple, to obtain the optimal bound, we need to iterate our basic construction recursively. Then it becomes quite intricate. Can the current algorithm be simplified? Or, is there any other algorithm that is at same time simple and optimal?

### B. Related works

*More on quantile sketches.:* Early works on quantiles sketches include [1], [15], [16]. Among them, the MRL sketch [16] and its randomized variant from [14] lead to the aforementioned KLL sketch. Another variant of the problem is

[2]This conjecture is recently sovled by

[3]If $n$ is not known is advance, instead of simple sampling, one can replace the use of GK sketch in KLL with our algorithm. As the compactor hierarchy part of KLL stores only $O(1/\varepsilon)$ elements, it results in the same space complexity as the known $n$ case.

[4]This follows from the $\varepsilon^{-1}\log \varepsilon U$ lower bound in Section VII (which holds for both deterministic and randomized algorithms), and the aforementioned $k \cdot \min(\log(n/k), \log\log(1/\delta))$ lower bound in [11] (setting $k = 1/\varepsilon$).

| Algorithm | Type | Space (words) | Space (bits) |
|---|---|---|---|
| GK sketch [6] | deterministic comparison-based | $O(\varepsilon^{-1}\log(\varepsilon n))$ | $O(\varepsilon^{-1}(\log^2(\varepsilon n)+\log(\varepsilon n)\cdot\log U))$ |
| q-digest [9] | deterministic bounded-universe | $O(\varepsilon^{-1}\log U)$ | $O(\varepsilon^{-1}(\log^2 U+\log(\varepsilon n)\cdot\log U))$ |
| KLL sketch [7] | randomized comparison-based | $O(\varepsilon^{-1}\log\log(1/\delta))$ | $O(\varepsilon^{-1}\log\log(1/\delta)\cdot(\log\log(1/\delta)+\log U)+\log n)$ |
| Our algorithm (Theorem I.2) | deterministic bounded-universe | $O(\varepsilon^{-1})$ | $O(\varepsilon^{-1}(\log(\varepsilon n)+\log(\varepsilon U)))$ |

TABLE I: The word and bit complexity of quantile sketches.

the *biased* quantile sketches (also called *relative error* quantile sketches), meaning that for queries of rank $r$, the algorithm can only have an error of $\varepsilon r$ instead of $\varepsilon n$. That is, we require that the $0.1\%$ quantiles are extremely accurate, while the $50\%$ quantile can allow much more error. This question was raised in [17]; since then, people have proposed deterministic [18], [19] and randomized [20] algorithms for this problem. There are also other variants such as sliding windows [21], weighted streams [22] and relative value error [23]. In practice, there are also the t-digest sketch [24] and the moment-based sketch [25], which do not have strict theoretical guarantees. In particular, [26] shows that there exists a data distribution, such that even i.i.d. samples from that distribution can cause t-digest to have arbitrarily large error.

## II. Preliminaries

### A. Definitions for streams

Define the *rank* of an element $x$ in a stream $\pi$, denoted $\mathrm{rank}_\pi(x)$, to be the total number of elements of $\pi$ that are less than or equal to $x$. We also define a notion of distance between two streams. For two streams $\pi, \pi'$ of equal length, define their *distance* as follows:

$$d(\pi,\pi') = \max_{x\in[1,U]} |\mathrm{rank}_\pi(x) - \mathrm{rank}_{\pi'}(x)|.$$

We observe that this distance satisfies some basic properties, i.e., the triangle inequality, and subadditivity under concatenation of streams:

**Observation II.1** (Triangle inequality)**.** For all streams $\pi, \pi', \pi''$ of the same length,

$$d(\pi,\pi') \le d(\pi,\pi'') + d(\pi'',\pi')$$

**Observation II.2.** For all streams $\pi, \pi'$ of the same length and $\rho, \rho'$ of the same length,

$$d(\pi\circ\rho, \pi'\circ\rho') \le d(\pi,\pi') + d(\rho,\rho'),$$

where $\pi\circ\rho$ denotes concatenation of the streams $\pi$ and $\rho$.

### B. Other notation

Throughout this paper, we use standard asymptotic notation, including big $O$ and little $o$. For clarity, we sometimes omit floor and ceiling signs where they might technically be required.

All logarithms in this paper are considered to be in base 2, and we define the *iterated logarithm* $\log^*(m)$ to be the number of times we need to apply a logarithm to the number $m$ to bring its value below 1.

We also define the function $\llbracket x \rrbracket$, for any $x \in \mathbb{R}^+$, to be the smallest power of 2 that is at least $x$. In particular, we always have $x \le \llbracket x \rrbracket \le 2x$.

## III. Technical overview

In this section, we explain the main idea of our algorithm.

First, we get a few technical details out of the way. We will assume for now that we know $n$ beforehand. For this overview, we will focus on describing a data structure that uses $O(\varepsilon^{-1}\log(\varepsilon^{-1})\log\log U)$ words of memory. After that, we will briefly describe the modifications that we perform to bring the space complexity down to $O(\varepsilon^{-1})$ words.

*a) The eager q-digest sketch:* Before explaining our algorithm, it would be instructive to first reivew the q-digest algorithm because our algorithm is based on it. At a high level, this data structure is a tree where every node represents some subset of the stream elements received so far. The node doesn't store each element exactly, but only an interval that contains all of the elements it represents and a count of how many elements it represents. The version we describe slightly differs from the typical treatment, and we call it eager q-digest. The data structure has the following structure and supports the following operations.

- **Structure:** The eager q-digest is a binary tree of depth $\log U$. The nodes in the bottom level of the tree (which we call the *base level*) correspond left-to-right to each element $1, 2, \ldots, U$ in the universe. Each non-base level node corresponds to a subinterval of $[1, U]$ consisting of the base level nodes below it. Each node $u$ represents a subset of $W[u]$ elements ($W[u]$ is the weight/count of the node) that have been received so far; that is, when an element is inserted, it increments the counter $W[u]$ at some node. The $W[u]$ elements that $u$ represents must all be within the node's interval.
- **Insertion:** We insert elements into the tree top-down as follows: upon receiving an element $x \in [1, U]$, look at the path from the root to $x$ and increment the counter $W[u]$ of the first non-full node $u$. A node is *full* if its weight is

already at capacity, which we set to be $\alpha := \frac{\varepsilon n}{\log U}$. Base level nodes are permitted to exceed capacity.

- **Rank queries:** We are given an element $x \in [U]$ for which we want to return the rank. To do this, answer with the total weight of everything on the path from the root to the base node $x$ and everything to the left of that path in the tree. All the elements inserted in nodes to the left of this path must have been less than $x$ (since their intervals only contain elements less than $x$) and all the elements inserted to the right must be larger. As such, the error in the rank estimate is only the sum of nodes along the path (not including $x$), which is bounded by the depth of the tree times the weight of each node above $x$, at most $\alpha \log U = \varepsilon n$.
- **Quantile queries:** We are given a rank $r \in [n]$ for which we want to return an element between the rank-$(r - \varepsilon n)$ and the rank-$(r + \varepsilon n)$ element of the stream. The ability to do this follows from the ability to answer rank queries, since we can simply perform a binary search.[5]

Let us look at an example of an eager q-digest. Each node has capacity (maximum weight) $\alpha = 5$ for this example.

In this example, triangle represents 5 elements in the interval $[1, 4]$, square represents 5 more elements in the interval $[3, 4]$, and star represents 3 more elements exactly equal to 3. If we insert the number 3 into the example, it would not get inserted into triangle or square because they are full, and so it would be put into star and increment the count by 1. If we want to then find the rank of the number 3 (in the pictured tree exactly, before the insertion), we return the sum of the weights on the circled nodes plus the path to $x$, which is $9 + 5 + 5 + 3 = 22$. This can be off by at most 10 – we know the 9 elements represented by the circled nodes are definitely less than 3, the ones inserted directly to the star are exactly 3, the ones to the right are definitely more than 3. The ones inserted to the triangle and square are the only unknowns.

*b) Analyzing the space complexity of eager q-digest:* The space complexity (in bits) of q-digest (and similarly of eager q-digest) is well known to be $O(\varepsilon^{-1}((\log U)^2 + \log U \log n))$. Let us understand why, so we can see where we might improve upon this. The space complexity is approximately the product of the following two things:

(1) The number of non-empty nodes. This is at most $O(\varepsilon^{-1} \log U)$ since the number of full nodes (which is within a constant factor of the number of non-empty nodes) is $n/\alpha = \varepsilon^{-1} \log U$.
(2) The amount of space necessary per non-empty node. Naively, we would need to store the location of each nonempty node (the interval it corresponds to) and the weight of the node (the number of stream elements it corresponds to). This would take $\log U + \log n$ space.

As such, in total the space complexity is $O(\varepsilon^{-1}(\log U)^2 + \log U \log n)$. In our sketch, we do not reduce (1), the number of nodes. Therefore, we must reduce the storage in (2) substantially. This has two parts: efficiently storing the corresponding interval (location in the binary tree) of each node and efficiently storing the count.

It is actually quite simple to store the interval/location of each node: To see this, notice that the non-empty nodes form a connected tree of their own within the large binary tree. Since the tree is binary, storing the edge from a parent to child in the tree of nonempty nodes takes only $O(1)$ space. This observation is quite straightforward from the way we formulated q-digest, but the usual implementation of q-digest doesn't push to the top eagerly, and so is unable to directly save this $\log U$ term.

*c) The main challenge, avoiding storing counters:* The second challenge is to avoid storing a counter $W[u]$ at each node. One useful observation about the structure of the tree of non-empty nodes is that all internal nodes are full (at capacity) and only its leaves, which we call *exposed nodes*, need counters. Unfortunately, a constant fraction of the non-empty nodes are exposed nodes, so this doesn't actually save on space.

Another idea is to store only an approximate count at each node. Unfortunately, we cannot just store an independent approximate count at each node, or even only a counter that estimates when the count surpasses the threshold $\alpha$; this is impossible to do deterministically without using $\log \alpha$ space (which is too large). Even in the randomized setting, approximately counting each node independently does not improve upon KLL.

The situation is summarized above in Figure 2. At each of the exposed nodes, denoted $v_1, v_2, \ldots, v_\ell$, we want to store some approximate version of counters $W[v_1], W[v_2], \ldots, W[v_\ell]$ that represent how many elements are inserted into that node using significantly less than $\log n$ space, ideally $O(1)$ space.

For simplicity, assume that elements are received in "batches" of size $\widehat{n}$ (to be determined later), which we can use unlimited space to process. Our only constraint is to minimize storage space between batches. Let us assume that before the batch, all the counters $W[v_1], W[v_2], \ldots, W[v_\ell]$ are less than $\alpha/2$ and set $\widehat{n} = \alpha/2$ so the set of exposed nodes won't change within the batch. At the end of the batch, we need to find suitable approximate values $\widehat{W}[v_1], \widehat{W}[v_2] \ldots, \widehat{W}[v_\ell]$ to increment the counters by, based on the true counts $C[v_1], C[v_2], \ldots, C[v_\ell]$ of the stream elements.

Let us quantify how "inaccurate" these approximate counts can be compared to the true counts. The amount of additional error (in rank-space) introduced by answering a rank query for some universe element below a node $v_i$ should be at most $\varepsilon\widehat{n}$ – we can tolerate this much because it only doubles $\varepsilon$ and we could've chosen $\varepsilon$ to be half as big at the start. The value of this rank query, or the total weight of all the nodes to the left of $v_i$ and the path to $v_i$ changes by $\left|\left(\widehat{W}[v_1] + \ldots + \widehat{W}[v_i]\right) -\right.$

[5]This is true in a black-box way; see Section VI for details.

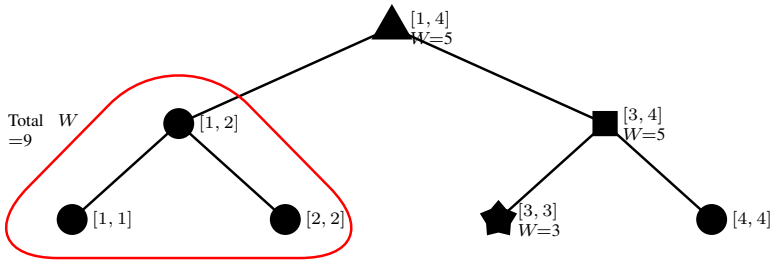


Fig. 1: An example eager q-digest tree.

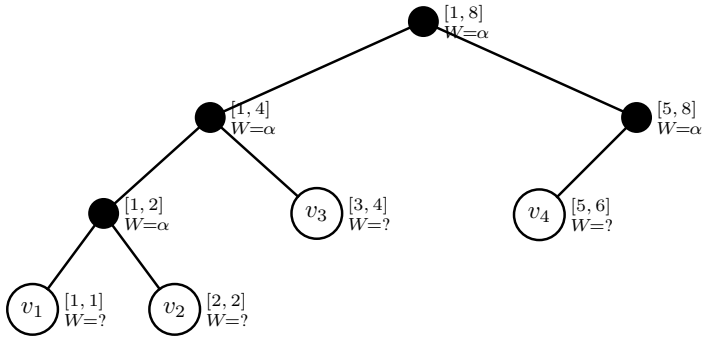


Fig. 2: The tree formed by non-empty nodes in eager q-digest. (The filled nodes are the full nodes.)

$\Big(C[v_1]+\ldots+C[v_i]\Big)\Big|$, and so we need to ensure for all $i$,

$$\left|\Big(\widehat{W}[v_1]+\ldots+\widehat{W}[v_i]\Big)-\Big(C[v_1]+\ldots+C[v_i]\Big)\right|<\varepsilon\widehat{n}. \quad (1)$$

Here is a simple way to make that happen: Take the 0-th element, the $(\varepsilon\widehat{n})$-th element, the $(2\varepsilon\widehat{n})$-th element and so on, and increment the counters $W[v_i]$ corresponding to those elements each by $\varepsilon\widehat{n}$. Then, Equation 1 is satisfied, and also the counters can be stored in $O(\log(\varepsilon^{-1}))$ bits since they are always multiples of $\varepsilon\widehat{n}=\varepsilon\alpha/2$ and so only have $2\varepsilon^{-1}$ possibilities.

*d) The main idea: recursive quantile sketch:* Of course, the glaring issue is how to find (an approximation of) the 0-th element, the $(\varepsilon\widehat{n})$-th element, the $(2\varepsilon\widehat{n})$-th element and so on, or at least which $v_i$ each one corresponds to, *without* storing the entire batch of $\widehat{n}$ elements. In particular, we have reduced to the following problem: We receive $\widehat{n}$ elements in a stream in the universe $\{v_1,\ldots,v_\ell\}$, and we need to return the approximate 0-th element, the $(\varepsilon\widehat{n})$-th element, the $(2\varepsilon\widehat{n})$-th element and so on. These are just quantile queries! In particular, we need a quantile sketch on a universe of size $\ell$ receiving $\widehat{n}$ elements. The new universe size $\ell$ is at most the number of exposed nodes of the eager q-digest, which is at most $\varepsilon^{-1}\log U$, and so we have a big saving – the new quantile sketch is on a logarithmically smaller universe, and so even naively using eager q-digest for the inner sketch will save space.

This solves the problem. The outer quantile sketch requires only $O(\varepsilon^{-1}\log(\varepsilon^{-1})\log U)$ space because it needs $O(\log(\varepsilon^{-1})$ space per node, and the inner sketch requires only $\varepsilon^{-1}\log\log U(\log\log U+\log\widehat{n})$ space because its universe size is $\log U$. Both of these are within $O(\varepsilon^{-1}\log(\varepsilon^{-1})\log\log U)$ words of memory. An illustration of the recursive step is shown in Figure 3, where we build a new eager q-digest whose universe is the exposed nodes of our original eager q-digest. This new eager q-digest will process $\widehat{n}$ elements and ultimately return the 0-th element, the $\varepsilon\widehat{n}$-th element, $2\varepsilon\widehat{n}$-th element, and so on.

*e) Modifications to get the optimal bounds:* We can iterate this construction recursively by building a new eager q-digest on the exposed nodes of the second eager q-digest. This process will continue to reduce the universe size nearly logarithmically each time. The number of layers before reaching a constant sized universe is roughly $\log^* U$, and so to get constant error and constant space, we will need to be careful with how we set the error fraction $\varepsilon_i$ for each recursive layer and argue that the total size of the sketches converges.

We also made an assumption that when we started receiving the batch of $\widehat{n}$ elements, all the exposed nodes had weight at most $\alpha/2$. However, the node could have any weight $j\varepsilon\alpha$. To deal with this, we need the lower level q-digest to deal with nodes getting "overfilled."

Our final algorithm also manages to get rid of $\log(\varepsilon^{-1})$ factors in the space complexity. This takes a number of additional considerations. One is that the nodes cannot even store counts that require $O(\varepsilon^{-1})$ bits, but truly need to just be either empty or full. To deal with this, we will increase

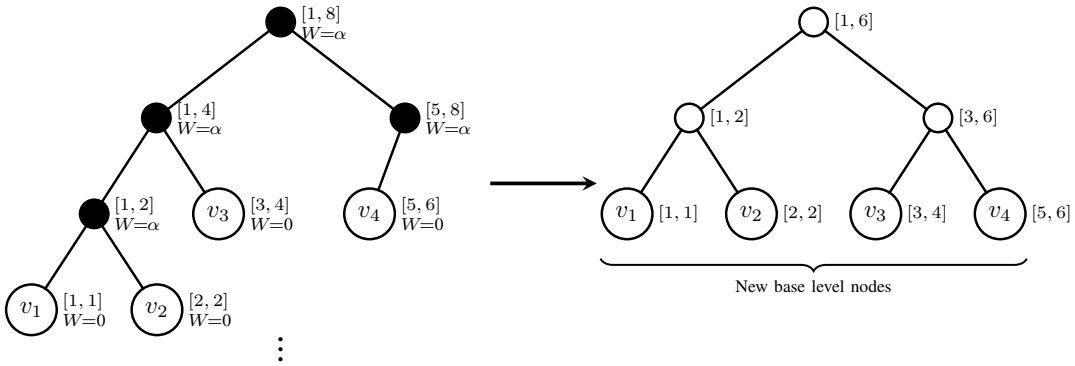


Fig. 3: An inner eager q-digest tree whose universe is the exposed nodes of the original tree. (The filled nodes are the full nodes.)

the batch size to $\widehat{n}\alpha$ but now we will need to deal with nodes getting overfilled again. A second issue is that, as described, at the last layer of recursion, the number of nodes would be $\varepsilon^{-1}\log(\varepsilon^{-1})$, which is slightly too large. To deal with this, we will have to use an optimized eager q-digest, which we discuss in Section IV.

## IV. Warm up: optimized eager q-digest

In this section, we will describe the *optimized eager q-digest* algorithm. This slightly improves the q-digest algorithm of [9]. The space complexity of optimized eager q-digest will be fairly similar to that of q-digest (it achieves $O(\varepsilon^{-1}\log\varepsilon n\log\varepsilon U)$ bits instead of $O(\varepsilon^{-1}(\log U+\log\varepsilon n)\log U)$ bits).

Although it does not contain the main idea of this paper, we need it as a building block of our algorithm. Also, we hope that this section can be a warm-up that familiarizes readers with our notation and the basics about q-digest.

Though we have already talked briefly about the eager q-digest in the technical overview, we will start anew in this section by building the algorithm up from the original q-digest, since we make several more modifications than what we described in that section.

*Tree structure of the original q-digest sketch:* In the original q-digest sketch of [9], there is a underlying complete binary tree $T$ of depth $\log U$. We say that those nodes at depth $\log U$ are at the *base-level* of $T$. These nodes correspond (from left to right) to each element $1,2,\ldots,U$ in the universe.

We label each node in $T$ with a subinterval of $[1,U]$. First, the base-level node corresponding to $i$ is labeled with $[i,i]$. For a node above $u$ the base level, its interval is the union of all its base-level descendants. For every node $u\in T$, it also has a weight $W[u]$ associated to it. Intuitively, one can think of the nodes $u\in T$ with weight $W[u]$ and interval label $[a_u,b_u]$ as a representative of $W[u]$ many elements in the stream that are within $[a_u,b_u]$.

In the original q-digest all nodes $u$ except the base level nodes can have weight at most $W[u]\le\alpha$. This is the *capacity* of the node and is usually set to $\alpha=\frac{\varepsilon n}{\log U}$. When there is an insertion of stream element $x$, the algorithm finds the base-level node $v$ whose interval equals $[x,x]$ and increases $W[x]$ by 1. This is always possible as there is no capacity constraint for base-level nodes.

Since this tree $T$ has as many as $2U-1$ nodes, the q-digest algorithm does not store the tree $T$ nor the labels. It only store the set $S$ of *non-empty nodes*, those nodes $v$ with $W[v]>0$. As there are more and more insertions, the set $S$ grows. Whenever $|S|>\frac{\log U}{\varepsilon}$, the q-digest algorithm performs a *compression*.

One way of performing such compression is to find all nodes $u$ such that $W[u]>0$ and $W[\mathrm{parent}(u)]<\alpha$, and move one unit of weight from $W[u]$ to $W[\mathrm{parent}(u)]$. After there is no such node $u$, let $F\subseteq S$ be the set of *full nodes* $v$ with $W[v]=\alpha$. We know that $|F|\le\frac{n}{\alpha}=\frac{\log U}{\varepsilon}$. Now for every nonempty node $u\in S$, its parent must be a *full node*. So compression gets the number of nonempty node down to $|S|\le 3|F|=O\left(\frac{\log U}{\varepsilon}\right)$. For every $u\in S$, the actual information stored by original q-digest are 1. the position of $u$ in the tree $T$ (which takes $\log U$ bits); 2. weight $W[v]$ (which takes $\log\alpha\approx\log(\varepsilon n)$ bits).

Finally, for all these to make sense, we have to be able to answer rank queries. In order to estimate $\mathrm{rank}(x)$, we simply add up the weights $W[u]$ of all nodes $u$ whose intervals contain at least one element less or equal to $x$. This might overcount the number of actual stream elements which are at most $x$; any node whose interval contains both an element which is at most $x$ and greater than $x$ can contribute to the overcounting. These nodes are all (strict) ancestors of the node in the base level corresponding to $x$, so there are at most $\log U$ of them, and their total weight is thus at most $\alpha\cdot\log U$. Thus the answer to the rank query is off by at most $\alpha\cdot\log U\le\varepsilon n$.

Now, having described the original q-digest algorithm, we will describe the modifications we make to it to get the

optimized eager q-digest.

*Modification 0, Enforcing capacity constraints on base-level nodes:* In our algorithm, we will need every node, including those at the base level, to satisfy the capacity constraint. But an element $x \in [U]$ could potentially have multiplicity $> \alpha$ in the input stream.

To handle this, rather than the trees ending at the base level, we let them continue as infinite paths (i.e., unary trees) descending from each node of the base level. Let $u$ be a base-level node that is labeled with interval $[x, x]$. All nodes on the infinite path below $u$ will also just be labeled $[x, x]$. As a sanity check, since we are not storing the tree $T$ anyway, it makes sense to be infinite.

*Modification 1: Use a forest of $1/\varepsilon$ trees. :* To improve the $\log U$ factor to $\log(\varepsilon U)$, we have to equally divide the universe into $1/\varepsilon$ intervals and maintain a tree for each one. This gives us a forest of $1/\varepsilon$ trees, while allowing us to set $\alpha$ to $\frac{\varepsilon n}{\log(\varepsilon U)}$.[6]

Roughly speaking, this change corresponds to removing the top $\log(1/\varepsilon)$ levels of the q-digest tree while keeping the levels below it. Although only offering a small improvement here, this is actually essential for our final algorithm. It is one of the ingredients that allow us to avoid the extra $O(\varepsilon^{-1}\log(\varepsilon^{-1}))$ term in the number of words used.

*Modification 2: Move weights up eagerly:* Next we describe how nodes are inserted into the eager q-digest. The original q-digest algorithm moves weight up the tree lazily; that is, it does so when the number of nodes stored exceeds its limit. By contrast, the eager q-digest will do so *eagerly*: upon receiving an element of the stream, it will immediately move it up as much as possible.

More formally, when we receive an element $x$ of the stream, we do not increase the weight of the base-level node with interval $[x, x]$ as we would in a normal q-digest. Instead, we immediately move this weight up. That is, we pick the highest non-full node whose interval contains $x$, and we increment its weight by 1.

*Space Complexity: Full nodes and non-full nodes:* We now look at the space complexity of optimized eager q-digest. An ordinary q-digest has to store, for every non-empty node, both its location in $T$ and its weight. However, in an optimized eager q-digest, the non-empty nodes are *upward closed*; that is, every parent of a non-empty node is also non-empty. (In fact, every parent of an non-empty node is actually full, since otherwise the weight would have been pushed up to the parent.) Thus, the non-empty nodes form at most $1/\varepsilon$ trees which include the roots of their components in $T$. Storing the topology of a binary tree of size $k$ only requires space $k$ (it is enough to use 2 bits for each node to record whether it has left/right child). Thus the total space required to describe the locations of the non-empty nodes is only $O(|S| + 1/\varepsilon)$ bits, where $|S|$ is the total number of non-empty nodes.

At this point, for all the full nodes, we are already done. Since we know that their weight is exactly $\alpha$, there is nothing more to store. Since $|S| \le 3|F| \le \frac{3n}{\alpha} = O\left(\frac{\log(\varepsilon U)}{\varepsilon}\right)$, we are able to store all the full nodes with only $O(1/\varepsilon)$ words. However, there are still the non-full nodes in $S$. Since we have to store the weight for each of them, this takes $O(|S|\log\alpha) = O\left(\frac{\log(\varepsilon U)}{\varepsilon}\cdot\log(\varepsilon n)\right)$ space.

This completes the description of eager q-digest. We have saved an $|S|\log U$ term in the space complexity by not having to store the location of each non-empty node, but the $|S|\log\alpha$ term from storing the weights of non-full nodes in $S$ still remains. In the following section, the main idea of our algorithm is to recursively maintain these non-full nodes in $S$ with another recursive layer of our algorithm. When carefully implemented, we are able to ensure that every node in our trees are either full or empty, except at the very last layer of recursion. This removes the extra $|S|\log\alpha$ term.

## V. Our $O(\varepsilon^{-1})$-word algorithm

In this section, we will implement the sketch in Table II, proving Theorem I.2. We assume throughout this section that $\varepsilon U$ is at least a sufficiently large absolute constant, since otherwise we can increase $U$ without affecting our asymptotic space complexity.

To start with, we will also assume that we know an upper bound on $n$ (this upper bound will become $n_0$), and that it is sufficiently large (that is, $n_0 \ge n^*$, where $n^*$ is a function of $U, \varepsilon$). Furthermore, we will initially allow rank queries to have error up to $\varepsilon n_0$. We will maintain these assumptions until Section V-E, where we will then describe how to dispense with these assumptions.

We now outline how this section will proceed. In Sections V-A and V-B, we describe the data structure, and how to handle insertions into the data structure, including how to merge layers of the data structure. In Section V-C, we bound the error introduced into the data structure with each merge. Then, in Section V-D, we describe how to perform rank queries and show a bound of $\varepsilon n_0$ on the error of a query. We next, in Section V-E, describe how to make our data structure work even when $n < n^*$, and also improve our bound on error of a query to $\varepsilon t$ (where $t$ is the size of the stream so far). In Section V-F, we pick the numerical parameters of our data structure such that the claims of the previous section hold. Finally, in Sections V-G and V-H, we analyze the space and time complexity of our algorithm, respectively.

### A. Structure of the sketch

As mentioned before, our sketch will be formed from recursive applications of the eager q-digest. We now define the structure of the recursive layers, which we number $0, 1, \ldots, k$.

[6]This is because the depth of each tree becomes at most $\log(\varepsilon U)$ and the error for answering rank queries is at most the depth multiplied by $\alpha$.

**Our $\varepsilon$-approximate quantile sketch**

Space complexity: $O(\varepsilon^{-1}(\log(\varepsilon n) + \log(\varepsilon U)))$ bits.

Supported operations:

- INSERT($x$): Adds an element $x \in [U]$ to the stream. (Algorithm 1)
- RANK($x$): Returns the rank of $x$ up to $\pm\ \varepsilon t$ error where $t$ is the number of elements in the current stream. (Algorithm 6)

In Section V-H, we will show that each operation takes $O(\log(1/\varepsilon))$ amortized time under mild assumptions.

TABLE II: Our quantile sketch.

*The 0-th layer:* We start with the top layer (layer 0) and introduce our notation. The top layer has the same structure as an ordinary optimized eager q-digest forest. We call this underlying forest $T_0$. It has universe size $U_0 = U$ and error parameter $\varepsilon_0 = \varepsilon/8$. We would like to emphasize that in optimized eager q-digest, $T_0$ is a forest with $1/\varepsilon_0$ infinite trees where most nodes have weight 0. We call these nodes *empty*.

Whether empty or not, each node in this infinite forest is labeled with an interval. The $1/\varepsilon_0$ roots of the trees are labeled with $[1, \varepsilon_0 U], [\varepsilon_0 U + 1, 2\varepsilon_0 U], \ldots, [(1-\varepsilon_0)U + 1, U]$, respectively. Then, if a node is labeled with interval $[a, b]$, its two children are labeled with $[a, (a + b - 1)/2]$ and $[(a + b + 1)/2, b]$ respectively. (Since we assumed that $\varepsilon$ and $U$ are powers of 2, these are all integers.) As a special case, if $a = b$, the node is going to have only one child, labeled $[a, a]$.

In $T_0$, each node $u$ has a weight $W_0[u]$ that cannot exceed capacity $\alpha_0 = \varepsilon_0 n_0 / \lceil\log(\varepsilon_0 U_0) + 1\rceil$, where $n_0$ is an upper bound on $n$. We define the set of *full nodes*, $F_0$, as the set of nodes that have a weight of exactly $\alpha_0$. Recall from optimized eager q-digest that we know $F_0$ is a upward-closed set of nodes and is therefore itself a forest of at most $1/\varepsilon_0$ trees. (See Section IV for details). We will enforce the invariant that every node in the tree $T_0$ is either *full* or *empty*. So nodes in $F_0$ are the only nodes in $T_0$ that we actually use and store. As mentioned before, this allows us to store each node with only a constant number of bits.

Note that if we were to add new full nodes to this structure, the empty children of full nodes in $F_0$, as well as the empty roots of trees, are potentially positions for new nodes. We call these empty nodes the *exposed nodes*. Formally, the *exposed nodes* of $T_0$ is the set of empty nodes that do not have a full parent. For a concrete example, see the forest $T_0$ in Figure 4.

*a) Intuition: Batch processing of insertions:* Let us first jump ahead and sketch the purpose of having layer $i$ ($1 \le i \le k$). Imagine if we insert a new element in the stream. Then, an execution of the eager q-digest algorithm will increase the weight of one exposed node in $V_0$ to $1 \ll \alpha_0$. However, our algorithm cannot do the same, because it would break our invariant of having only full nodes in $T_0$. Instead, we maintain the exposed nodes $V_0$ with our recursive structure (layers $\ge 1$) and insert the new element into layers $\ge 1$. These recursive layers act like a "buffer"; once they accumulate $n_1$ elements, we clear them and compress those elements into new full nodes in $T_0$.

In general, for layer $i$ ($1 \le i < k$), we group $n_{i+1}$ insertions in a *batch* and insert them to layer $\ge i + 1$. After each batch, we compress the elements in layer $\ge i + 1$ into full nodes in layer $i$ and clear layer $\ge i + 1$. Full details of how we handle insertion will be discussed in Section V-B.

*The $i$-th layer ($1 \le i \le k$):* Roughly speaking, the upper part of the layer $i$ structure (which we call $T_i$) resembles an optimized eager $q$-digest forest with whose "universe size" is $U_i$, which is an upper bound on $|V_{i-1}|$ (when we pick the values of the parameters, we will prove this upper bound in Claim V.19). At depth $h_i := \log(\varepsilon_i U_i)$, they have exactly $|V_{i-1}|$ nodes[7]. Each such node $u$ will correspond to an exposed node $v \in V_{i-1}$, in order (from left to right). We call this depth the *base level* of $T_i$. This is the upper part of $T_i$.

For the interval labeling of the upper part, as each base level node $u$ corresponds to an empty node $v \in V_{i-1}$, naturally, $u$ just inherits the interval label of $v$. Strictly above the base level, the interval of each node is the union of the intervals of its base-level descendants.

Now we start to describe the lower part of $T_i$. Unlike the optimized eager $q$-digest, we will also allow $T_i$ to grow beyond the base level. (We give some intuition for this in Remark V.2, which readers may skip on the first read.) For each base-level node $u$ that corresponds to $v \in V_{i-1}$, we copy the empty infinite subtree of $v$ in $T_{i-1}$, and put it as the subtree of $u$ in $T_i$. This also copies the interval labels on nodes in the subtree. For a concrete example, see the forests $T_1, T_2$ in Figure 4.

**Remark V.1.** Because we copied the subtrees from $T_{i-1}$, for any node $u$ below the base level (including the base level itself), there exists a unique node $u' \in T_{i-1}$ corresponding to it. (We will soon see that $u'$ is in fact an empty node.)

A node $u$ in $T_i$ has weight $W_i[u]$ and capacity $\alpha_i = \varepsilon_i n_i / \lceil\log(\varepsilon_i U_i) + 1\rceil$. We again call a node *full* when it reaches its capacity. $F_i$ is defined to be the set of all full nodes in $T_i$. We maintain the similar invariant as layer 0: For all $0 \le i < k$, the forest $T_i$ will contain either *full* or *empty* nodes.[8]

[7]Note that in an optimized eager $q$-digest, the base level contains $U_i$ nodes; we just remove the remaining $U_i - |V_{i-1}|$ nodes and their descendants, and also any inner nodes with no descendants remaining.

[8]Note when $i = k$, since there are no further recursive layers, we do not require the invariant for it. Insertions to $T_k$ are simply handled as in a normal optimized eager q-digest. (See Section V-B for more details.)

Note that this invariant means that, for layers $i < k$, instead of storing the weight map $W_i$, it suffices to only store $F_i$, since the contents of $W_i$ are determined by $F_i$.

Finally, $V_i$, the set of *exposed nodes* of $T_i$, is defined as the set of of empty nodes which do not have a full parent (for $1 \le i < k$)[9]. Note that there may be some exposed nodes above the base level. (This results in a subtlety in the interval labels. See Remark V.3 for details. Readers may skip it on their first read.)

**Remark V.2.** Suppose that we do not allow the tree $T_i$ to grow beyond the base level. Then the total weight of it can be at most $2\alpha_i|V_{i-1}|$. In other words, layer $\ge i$ will not be able to handle more than that many insertions. But it turns out that we will later want to set $n_i \gg 2\alpha_i|V_{i-1}|$, so we have to allow $T_i$ to grow beyond the base level. (More specifically, we want to set $n_i$ so that $\varepsilon_i n_i \ge \alpha_{i-1}$, which is essential for Lemma V.13.)

**Remark V.3.** First, for the upper part of $T_i$, a node labeled with $[a, b]$ may not have children with evenly split interval labels ($[a, (a + b - 1)/2]$ and $[(a + b + 1)/2, b]$). This is clear since the labels of nodes above base level are derived bottom-up by taking the union of intervals at their base-level descendants. It is, though, tempting to think that for the lower part of $T_i$ (below the base level), all nodes labeled will have two children with equally split intervals. This, however, is also not always the case. It is possible that a base-level node $u$ corresponds to an exposed node $v \in V_{i-1}$ that is in the upper part of $T_{i-1}$. Then when we copy the subtree of $v$, those two children will not have equally split interval labels. For example, this happens in Figure 4, at the node labeled $[1, 6]$ in the tree $T_2$. Its two children split into $[1, 4]$ and $[5, 6]$, while an even split is $[1, 3]$ and $[3, 6]$.

**Remark V.4.** In order to avoid interrupting the flow of the paper, we will defer the precise definitions of the parameters $k, n_i, U_i, \varepsilon_i$ until Section V-F. However, so that the reader can have a sense of the scale of each of these parameters, we will give approximate values now that can be used as guidelines. All the parameters except $k$ will be powers of 2, to avoid divisibility issues. We pick the following rough values:

- The number of layers will be $k + 1 \approx \log^*(\varepsilon U)$.
- The approximation parameters $\varepsilon_i$ are all very close to $\varepsilon$, and can be thought of as essentially equal to $\varepsilon$.
- The $U_i$ will satisfy the approximate recursion $\varepsilon U_{i+1} \approx \log(\varepsilon U_i)$, so by the last level we will have $U_k \approx 1/\varepsilon$.
- The batch sizes $n_i$ will shrink very slowly (only by polylogarithmic factors in $\varepsilon U$), so they can all be thought of as roughly $n$, though decreasing.
  - In particular, even the last batch size $n_k$ is almost $n$ in this sense, so one can think of the algorithm as spending most of its time at layer $k$, with a "universe" of size $O(1/\varepsilon)$.
- Similarly, the capacities $\alpha_i$ are also all approximately $\varepsilon n$, though also decreasing in $i$.

[9]For $i = k$, we define $V_k$ instead to be the set of non-full nodes without a full parent.

### B. Handling insertions

In this subsection, we formally explain how we handle insertions.

*Insertions:* Recall that in Section V-A, we only require our invariant to hold for layers $i \ne k$. For layer $k$, it is maintained by a normal eager q-digest. For any insertion $x$, we first insert it into the layer $k$ as we would in a normal optimized eager q-digest. In other words, we find the exposed node in $T_k$ whose interval contains $x$ and increase its weight, $W_k[v]$, by 1. This node always exists due to the following observation.

**Observation V.5.** For all layers $1 \le i \le k$. the intervals of the exposed nodes $V_i$ are always disjoint and cover the entire universe $[1, U]$.

Then for $i = k, k-1, \ldots, 1$, we check if the total number of elements inserted so far, denoted by $t$, is a multiple of $n_i$. If so, we need to compress layers $\ge i$ into full nodes in layer $i-1$. Specifically, we will chose these $n_i$'s so that $n_i$ is always a multiple of $n_{i+1}$ for all $I$ (we prove this in Fact V.20(c)). Therefore if $w_{\text{tot}}$ is a multiple of $n_i$, layers $\ge i+1$ have already been compressed into full nodes of layer $i$. We will only need to compress layer $i$ into full nodes in layer $i-1$ and merge them into $T_{i-1}$. We call this procedure MERGE($i$) and will describe it next. The pseudocode for the insertion procedure as a whole is summarized below in Algorithm 1.

Next, we explain how MERGE($i$) compresses layer $i$ into full nodes in layer $i-1$. We follow a delicate three-step strategy. On a high level, it is carefully designed so that we incur an error (which is defined formally later in Section V-C) of at most $h_i \cdot \alpha_i + \alpha_{i+1}$ from the compression. (Recall that $h_i := \log(\varepsilon_i U_i)$ is the depth of the base level in $T_i$.) This is important to our analysis.

*a) Merge Step 1 - move the weight into $T_{i-1}$:* In the first step, we move all the weight in $T_i$ into empty nodes in $T_{i-1}$. There are two cases:

- For every node $u$ with weight below the base level (including the base level itself) in $T_i$, there is a unique empty node $u'$ in $T_{i-1}$ corresponding to it. (See Remark V.1.) We move all the weights for $u$ into that of $u'$. Formally, we just increase weight $W_{k-1}[u']$ by $W_k[u]$.
- For every node $u$ strictly above the base level of $T_i$, there is no node in $T_{i-1}$ that directly corresponds to it. Instead, we will take an arbitrary descendant $v \in T_i$ of it at the base level. As $v$ corresponds to an (exposed) empty node $v' \in T_{i-1}$, we will move the weight of $u$ there. Formally, we increase weight $W_{k-1}[v']$ by $W_k[u]$.

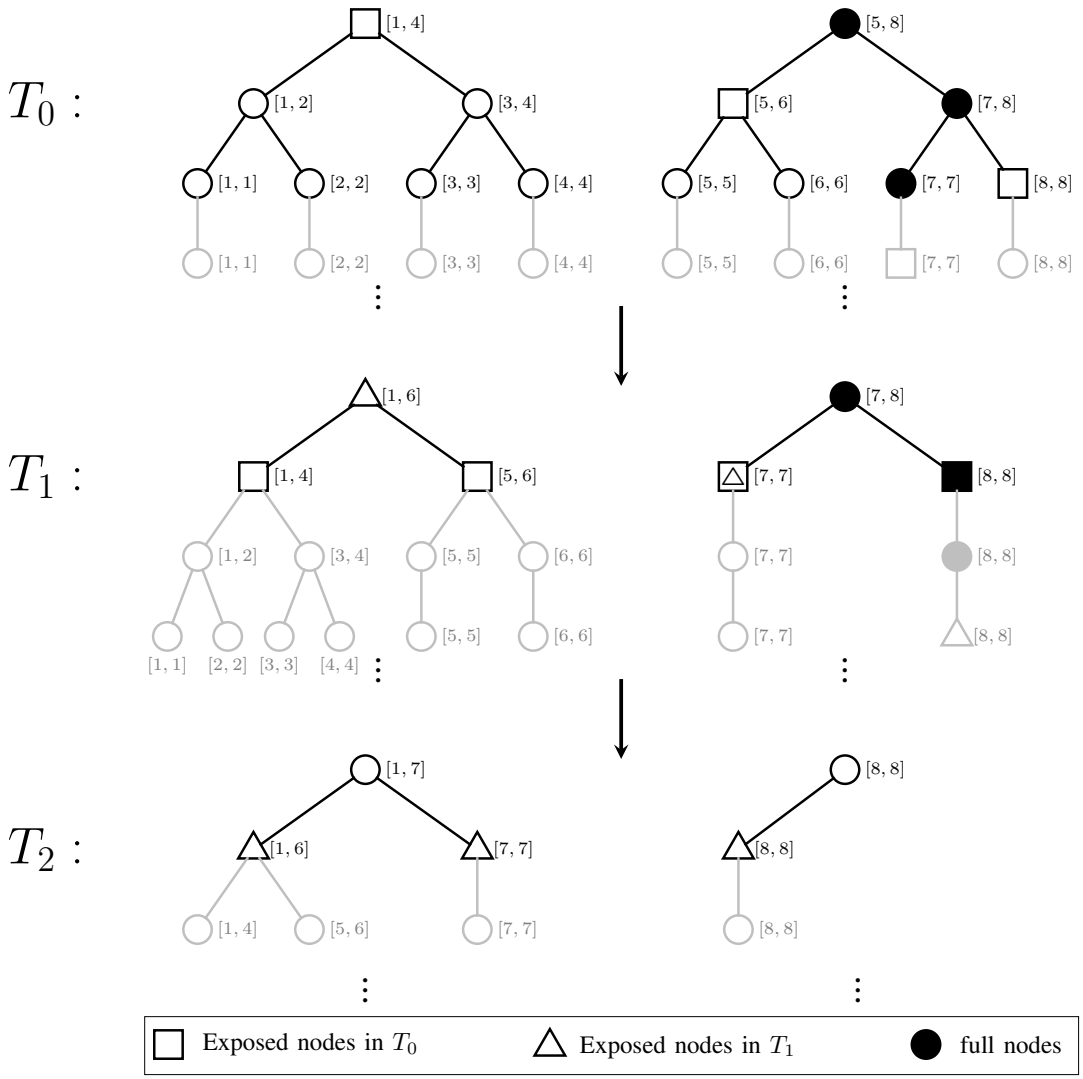


Fig. 4: The structure of different layers. Here $\varepsilon = 0.5$, so there are $1/\varepsilon = 2$ trees in each layer. The nodes below the base-level of each layer is marked as gray. Note that when we construct $T_i$, we take all the exposed nodes in $T_{i-1}$ and use them as the base-level nodes to build $1/\varepsilon$ trees. Then we copy their subtrees in $T_{i-1}$ to be their subtrees in $T_i$.

This is summarized in Algorithm 2. We defer the error analysis of this step to later in this section. Before we proceed, let us state a simple property about this step.

**Observation V.6.** We will choose the parameters so that $\alpha_i \cdot h_i \leq \alpha_{i-1}$ (this will be shown in Fact V.20(d)). (Recall that $h_i \coloneqq \log(\varepsilon_i U_i)$ is the depth of the base level in $T_i$.) Thus, after this step, all nodes in $T_{i-1}$ still have weight at most $\alpha_{i-1}$.

Therefore, this step does not exceed the capacity of nodes in $T_{i-1}$. But it does create a number of non-full nodes: It merges $T_i$ into $T_{i-1}$ while breaking our invariant of having only full or empty node in $T_{i-1}$. So the purpose of Step 2 and 3 is exactly to restore this invariant.

*Merge Step 2 - Compressing into full nodes:* Naturally, given the non-full nodes in $T_{i-1}$, we want to first perform a compression step similar to q-digest: Whenever a node $v \in T_{i-1}$ has a parent that is not full, we move weight from $v$ to $\mathrm{parent}(v)$.

Let $F_{i-1}$ be the set of full nodes after this step. We call the nodes that are neither full nor empty *partial nodes*. All the partial nodes are now either non-full children of full nodes in $F_{i-1}$ or an partially-full root. Importantly, we have the

[10]When $i = k$, this will actually be all $u$ such that $W_k[u]$ is nonzero, rather than just all full nodes.

[12]We are keeping the algorithm description simple by moving weights one unit at a time. In an actual implementation, one should of course move the maximum amount possible at each time.

**Algorithm 1** Inserting an element of the stream

```
1: procedure INSERT(x)
2:     // Insert x into T_k.
3:     v ← the highest non-full node in T_k whose interval contains x
4:     W_k[v] ← W_k[v] + 1                ▷ Recall W_k[v] is the weight of v.
5:     // Compress and merge
6:     t ← t + 1                          ▷ t is the total number of stream elements inserted so far.
7:     for i = k ... 1 do
8:         if n_k | t then
9:             MERGE(i)                   ▷ Compress T_i into full nodes of layer i − 1.
10:            Clear the structure at layer i.
11:            // Note the set V_{i−1} changes after MERGE(i). After we clear layer i, the structure of T_i implicitly changes
               according to the new V_{i−1}.
12:    if t = n_0 then
13:        DOUBLE()                       ▷ This handles unknown n (Section V-E); the reader may ignore it for now.
```

**Algorithm 2** Moving the weights from layer $i$ to empty nodes in layer $i-1$

```
1: procedure MOVE(i)
2:     for all u ∈ F_i[10] do
3:         if u is strictly above the base level of T_i then
4:             Let v ∈ T_i be an arbitrary descendant of u at
               the base level.
5:             Let v' ∈ T_{i−1} be the node corresponding to v
               (by Remark V.1).
6:             W_{i−1}[v'] ← W_{i−1}[v'] + W_i[u]
7:         else
8:             Let u' ∈ T_{i−1} be the node corresponding to u
               (by Remark V.1).
9:             W_{i−1}[u'] ← W_{i−1}[u'] + W_i[u]
```

**Algorithm 3** Compressing weights $W_{i-1}$ into full nodes

```
1: procedure COMPRESS(i − 1)
2:     while there exists v ∈ T_{i−1}, W_{i−1}[v] > 0 and
       W_{i−1}[parent(v)] < α_{i−1} do
3:         W_{i−1}[v] ← W_{i−1}[v] − 1
4:         W_{i−1}[parent(v)] ← W_{i−1}[parent(v)] + 1   ▷ Moving the weights.[12]
```

following observation.

**Observation V.7.** After this step, the interval labels of the partial nodes are all disjoint.

This is because no partial node can be an ancestor of another. These partial nodes are the leftovers that we will round up in Step 3.

*Merge Step 3 - Round up the leftovers:* As the interval labels of these leftover partial nodes are disjoint by Observation V.7, we can sort these nodes by their interval. Then, roughly speaking, we are going to take the (offline) quantile sketch of these nodes as the result for rounding.

More formally, suppose there are $\ell$ partial nodes. After sorting, these nodes are $v_1, v_2, \ldots, v_\ell$. Suppose each partial node $v_j$ is labeled $[a_j, b_j]$. We will have $a_1 \le b_1 < a_2 \le b_2 < \cdots < a_\ell \le b_\ell$. Let $r = \frac{1}{\alpha_{i-1}} \sum_{j=1}^{\ell} W_{i-1}[v_j]$ be the number of full nodes that we are expected to round up to.[13] For every $m \in [r]$, we find the first $q_m \in [\ell]$ such that $\sum_{j=1}^{q_m} W_{i-1}[v_j] \ge m \cdot \alpha_{i-1}$. These $v_{q_1}, v_{q_2} \ldots, v_{q_r}$ are the "quantiles" of these sorted partial nodes.

Then we set the weight of all $v_{q_m}$'s (for all $m \in [r]$) to $\alpha_{i-1}$ and the weight of all other $v_j$'s to zero. Note these $v_{q_m}$'s must be disjoint since by Observation V.6, any node has weight at most $\alpha_{i-1}$. This rounds up the partial nodes into $r$ many full nodes and finishes this step. An implementation of this procedure is given below in Algorithm 4.

**Algorithm 4** Rounding the leftovers

```
1: procedure ROUND(i − 1)
2:     c ← 0                              ▷ c is the cumulative total weight
3:     m ← 1
4:     for all partial node v in left-to-right order do
5:         c ← c + W_{i−1}[v]
6:         if c ≥ m · α_{i−1} then
7:             W_{i−1}[v] ← α_{i−1}
8:             m ← m + 1
9:         else
10:            W_{i−1}[v] ← 0
```

*b) Conclusion:* Finally, our merging operation is implemented by performing these three steps sequentially.

### C. Error analysis for merges

Before we analyze each step of MERGE($i$), let us first define the error metric.

[13]This is always an integer, because $\sum_{j=1}^{\ell} W_{i-1}[v_j]$ is equal to $n_i$ minus the total weight in the full nodes formed in Step 1 and 2, and we always choose $n_i$ to be a multiple of $\alpha_{i-1}$.

**Algorithm 5** Merging layer $i$ into layer $i-1$

```
1: procedure MERGE(i)
2:     MOVE(i)
3:     COMPRESS(i − 1)
4:     ROUND(i − 1)
5:     // During this process; the set of full nodes F_{i−1} that
       we store changes as we move weights around.
```

*Consistency and Discrepancy:* First, we define the notion of *consistency* between our layer $i$ sketch $T_i$ and a stream of elements $\pi$. Intuitively, this describes what layer $i$ should look like upon receiving stream $\pi$ if the merge had not introduced any error.

**Definition V.8** (Consistency)**.** We say that a stream $\pi$ is *consistent* with a subset of nodes $S \subseteq T_i$ if and only if there exists a map $f$ that maps $\{1, 2, \ldots, |\pi|\}$ to $S$ satisfying the following.

1) Each node $u \in S$ is mapped to exactly $W_i[u]$ times.
2) For every $1 \le j \le |\pi|$, the interval label of node $f(j)$ contains $\pi_j$.

Then we define the discrepancy between $T_i$ and the stream $\pi$. This quantifies the amount of additional error we have.

**Definition V.9** (Discrepancy)**.** We define the discrepancy between a stream $\pi$ and a subset of nodes $S \subseteq T_i$ as

$$\operatorname{disc}(\pi, S) \coloneqq \min_{\pi' \text{ consistent with } S} d(\pi, \pi').$$

Here, as defined in Section II, the distance between two streams is

$$d(\pi, \pi') = \max_{x \in [1,U]} |\operatorname{rank}_\pi(x) - \operatorname{rank}_{\pi'}(x)|.$$

*Analysis of Step 1:* Now, we show that Step 1 increases the discrepancy by at most $\varepsilon_i n_i$.

**Lemma V.10** (Step 1)**.** *Let $T_i$ be the layer-$i$ sketch before Algorithm 2 (Step 1). Also, let $S$ be the set of originally empty nodes in $T_{i-1}$ whose weight increases during Algorithm 2.*

*For any stream $\pi$, we have*

$$\operatorname{disc}(\pi, S) \le \operatorname{disc}(\pi, T_i) + \varepsilon_i n_i.$$

*Proof.* Let $\pi^* \coloneqq \arg\min_{\pi^* \text{ consistent with } T_i} d(\pi, \pi^*)$ and $f^*$ be the consistent mapping from $\pi_*$ to $T_i$. We will construct a stream $\pi'$ and a mapping $f'$ such that $\pi'$ is consistent with $S$ with mapping $f'$ and $d(\pi^*, \pi') \le \varepsilon_i n_i$. This finishes the proof because the distance we define satisfies the triangle inequality $d(\pi, \pi') \le d(\pi, \pi^*) + \varepsilon_i n_i$.

For any element $\pi^*_j$ $(1 \le j \le |\pi^*|)$ there are two cases:

1) If $f^*(j) = u$ for a node $u$ below the base level of $T_i$, let $u' \in T_{i-1}$ be the corresponding node (as in Line 9, Algorithm 2). We let $\pi'_j = \pi^*_j$ and set $f'(j) = u'$.
2) If $f^*(j) = u$ is a node $u$ strictly above the base level of $T_i$, let $v \in T_i$ be its descendant at the base level and $v' \in T_{i-1}$ be the corresponding exposed node (as in Line 5, Algorithm 2). We select an arbitrary element $y$ in the interval of $v$ (which is equal to that of $v'$), and let $\pi'_j = y$. Then we set $f'(y) = v'$.

From this construction, it is clear that $\pi'$ is consistent with $S$ under $f'$. To upper bound $d(\pi^*, \pi)$, consider any query $x \in [1, U]$, the difference of the rank of $x$ in $\pi$ and in $\pi'$ is bounded by the number of $j$'s such that $x$ lies strictly between $\pi^*_j$ and $\pi'_j$.

As $\pi^*_j \ne \pi'_j$, this can only happen in Case 2. Moreover, as $\pi^*_j$ was initially in the interval of $u$, and $\pi'_j$ is in the interval of $v$ (wich is contained by that of $u$), we know that $x$ must also be in the interval of $u$. Since there are at most $h_i$ such nodes $u$ strictly above the base level of $T_i$, and each is mapped to $\alpha_i$ times, we have at most $h_i\alpha_i$ many such $j$'s. We will choose the parameters in Section V-F so that $h_i\alpha_i \le \varepsilon_i n_i$ (this will follow from (3)). This proves $d(\pi^*, \pi) \le \varepsilon_i n_i$. □

Then we need to argue that when $S$ is merged with the original nodes in $T_{i-1}$, their discrepancies at most add up. This follows from the following observation, which is a consequence of Observation II.2:

**Observation V.11.** For two disjoint sets of nodes $S, T$ and any two streams $\pi_1$ and $\pi_2$, we have

$$\operatorname{disc}(\pi_1 \circ \pi_2, S \cup T) \le \operatorname{disc}(\pi_1, S) + \operatorname{disc}(\pi_2, T),$$

where $\circ$ means concatenating two streams.

*Analysis of Step 2:* It is not hard to see that Step 2 never increases discrepancy.

**Lemma V.12** (Step 2)**.** *For any stream $\pi$ that is consistent with $T_{i-1}$, after we perform Algorithm 3 on $T_{i-1}$, $\pi$ is still consistent with $T_{i-1}$. This implies that for any stream $\pi$, $\operatorname{disc}(\pi, T_{i-1})$ is always nonincreasing after perform Algorithm 3 on $T_{i-1}$.*

*Proof.* We prove this for each operation we perform. Whenever we move one unit of weight from $v$ to $\operatorname{parent}(v)$, we pick an arbitrary $1 \le j \le |\pi|$ such that $f(j) = v$ and let $f(j) \leftarrow \operatorname{parent}(v)$. Since the interval of $\operatorname{parent}(v)$ contains that of $v$, the consistency map remains valid. □

*Analysis of Step 3:* Finally, we show that the rounding in Step 3 only increases the discrepancy by $\alpha_{i-1} = \varepsilon_i n_i$.

**Lemma V.13** (Step 3)**.** *For any stream $\pi$, whenever we perform Step 3 (Algorithm 4) to $T_{i-1}$ in our algorithm, the discrepancy $\operatorname{disc}(\pi, T_{i-1})$ increases by at most $\alpha_{i-1}$ (which is equal to $\varepsilon_i n_i$).*

*Proof.* First, we only perform Algorithm 4 after Algorithm 3. So, by Observation V.7, all the partial nodes have disjoint intervals before Algorithm 4.

Before the algorithm starts, let $v_1, v_2, \ldots, v_\ell$ be the partial nodes of $T_{i-1}$ in sorted order, and $r = \frac{1}{\alpha_{i-1}} \sum_{j=1}^{\ell} W_{i-1}[v_j]$. Suppose $[a_1, b_1], [a_2, b_2], \ldots, [a_\ell, b_\ell]$ are their disjoint interval labels. Let $\pi^* = \arg\min_{\pi^* \text{ consistent with } T_{i-1}} d(\pi, \pi^*)$ and $f^*$ be corresponding consistency map. For every $m \in [r]$, let $v_{q_m}$ be the first node such that $\sum_{j=1}^{q_m} W_{i-1}[v_j] \geq m \cdot \alpha_{i-1}$. As discussed in Section V-B, these $v_{q_1}, v_{q_2}, \ldots, v_{q_r}$ are all distinct.

After the algorithm, all partial nodes become empty, except that $v_{q_1}, v_{q_2}, \ldots, v_{q_r}$ become full nodes with weight $\alpha_{i-1}$. We let $q_0 = 0$. For all $m \in [r]$, we do the following to construct stream $\pi'$ and its consistency map $f'$ (with $T_{i-1}$ after the algorithm):

- For all nodes $v_s$ with $q_{m-1} < s < q_m$ and all $j \in \{1, 2, \ldots, |\pi^*|\}$ such that $f^*(j) = v_s$, we set $\pi'_j \leftarrow a_{q_m}$ and $f'(j) \leftarrow v_{q_m}$.
- For the node $v_{q_{m-1}}$, we take $\sum_{s=1}^{q_{(m-1)}} W_{i-1}[v_s] - (m-1) \cdot \alpha_{i-1}$ many $j$'s such that $f^*(j) = v_{q_{m-1}}$ and set $\pi'_j \leftarrow a_{q_m}$ and $f'(j) \leftarrow v_{q_m}$.
- For the node $v_{q_m}$, we take $m \cdot \alpha_{i-1} - \sum_{s=1}^{(q_m)-1} W_{i-1}[v_s]$ many $j$'s such that $f^*(j) = v_{q_m}$ and set $\pi'_j \leftarrow \pi^*_j$ and $f'(j) \leftarrow f^*(j) = v_{q_m}$.

Now we prove that $d(\pi^*, \pi') \leq \alpha_{i-1}$, which by our choice of parameters in Section V-F, will be at most $\varepsilon_i n_i$. This will end the proof of this lemma by the triangle inequality $d(\pi, \pi') \leq d(\pi, \pi^*) + d(\pi^*, \pi') \leq d(\pi, \pi^*) + \varepsilon_i n_i$.

For any query $x$, its rank in $\pi^*$ and $\pi'$ differs by at most the number of $j$'s such that $x$ is strictly between $\pi^*_j$ and $\pi'_j$. As $\pi^*_j \neq \pi'_j$, this only happens in the first two cases. Suppose $f'(j) = v_{q_m}$. This implies $\pi'_j = a_{q_m}$. Then $f^*(j)$ must be a node $v_s$ with $q_{m-1} \leq v_s < q_m$, and $\pi^*_j \geq a_{q_{m-1}}$.

This implies $x$ is in the interval $[a_{q_{m-1}}, a_{q_m})$. Thus there is a unique $m$ for each query $x$, and by our construction, there can be at most $\alpha_{i-1}$ many $j$'s that are mapped to $v_{q_m}$ by $f'$. This proves that $d(\pi^*, \pi') \leq \alpha_{i-1}$. □

*Putting everything together:* We have essentially proved the following lemma.

**Lemma V.14.** *Let $\pi$ be the partial stream that arrives at time $[s \cdot n_i + 1, s \cdot n_i]$ for some integer $s$. According to Algorithm 1, after the $(s \cdot n_i)$-th insertion, we will perform* MERGE($k$)*,* MERGE($k-1$)*, …,* MERGE($i$) *in order.*

*Let $T_i$ be the structure at layer $i$ at the exact point that* MERGE($i+1$) *returns and* MERGE($i$) *has not started yet. Then we have*

$$\operatorname{disc}(\pi, T_i) \leq 2\gamma_{i+1} \cdot n_i,$$

*where*

$$\gamma_{i+1} = \varepsilon_{i+1} + \varepsilon_{i+2} + \cdots + \varepsilon_k.$$

*Proof.* We proceed by induction. In the base case where $i = k$, the layer-$k$ structure $T_k$ is always consistent with the partial stream $\pi$ by construction. Suppose that this holds for $i+1$. We split the stream $\pi$ into its batches $\pi = \pi^{(1)} \circ \pi^{(2)} \circ \cdots \circ \pi^{(n_i/n_{i+1})}$ where each $\pi^{(j)}$ has length $n_{i+1}$. For the ease of notation, we define $\pi^{(1\cdots j)} = \pi^{(1)} \circ \pi^{(2)} \circ \cdots \circ \pi^{(j)}$.

By the induction hypothesis, we know that after receiving each $\pi^{(j)}$ but immediately before we perform MERGE($i+1$), we have $\operatorname{disc}(\pi^{(j)}, T_{i+1}) \leq 2\gamma_{i+2} \cdot n_{i+1}$.

Then let us look at the process of MERGE($i+1$) and do another layer of induction. The induction hypothesis is that immediately after receiving $\pi^{(j)}$ and perform MERGE($i+1$), we have $\operatorname{disc}(\pi^{(1\cdots j)}, T_i) \leq 2(\gamma_{i+2} + \varepsilon_{i+1}) \cdot j \cdot n_{i+1}$. When $j = n_i/n_{i+1}$, this is simply $\operatorname{disc}(\pi, T_i) \leq 2(\gamma_{i+2} + \varepsilon_{i+1}) \cdot n_i = 2\gamma_{i+1} \cdot n_i$ and proves the outer induction.

In the base case, $T_i$ is empty, and we have $\operatorname{disc}(\emptyset, T_i) = 0$. Suppose for $j-1$, our induction hypothesis holds.

- It first performs MERGE($i+1$) which, by Lemma V.10, adds a set $S$ of new non-empty nodes to $T_{i-1}$ with $\operatorname{disc}(\pi^{(j)}, S) \leq \operatorname{disc}(\pi^{(j)}, T_{i+1}) + \varepsilon_{i+1} \cdot n_{i+1} \leq (2\gamma_{i+2} + \varepsilon_{i+1}) \cdot n_{i+1}$. Then by Observation V.11, after this step, we have $\operatorname{disc}(\pi^{(1\cdots j)}, T_i) \leq (2\gamma_{i+2} \cdot j + \varepsilon_{i+1} \cdot (2j-1)) \cdot n_{i+1}$.
- Then it performs COMPRESS($i$) which, by Lemma V.12, does not increase the discrepancy.
- Finally, it performs ROUND($i$) which, by Lemma V.13, increases the discrepancy by at most $\varepsilon_{i+1} \cdot n_{i+1}$ and results in $\operatorname{disc}(\pi^{(1\cdots j)}, T_i) \leq 2(\gamma_{i+2} + \varepsilon_{i+1}) \cdot j \cdot n_{i+1}$.

This finishes the inner induction and the proof of this lemma. □

The inner induction in the proof above actually proves the natural corollary below.

**Corollary V.15.** Let $\pi$ be the partial stream that arrives at time $[s \cdot n_i + 1, t]$ for some integer $s$ and $t$ such that $n_{i+1} \mid t$ and $t \leq s \cdot n_i$. After the $t$-th insertion and immediately after MERGE($i+1$) returns. We have

$$\operatorname{disc}(\pi, T_i) \leq 2\gamma_{i+1} \cdot |\pi|$$

where

$$\gamma_{i+1} = \varepsilon_{i+1} + \varepsilon_{i+2} + \cdots + \varepsilon_k.$$

### D. Answering queries

To answer a rank query, we simply add up the weights of all the nodes whose interval contains any element that is at most $x$, as shown in Algorithm 6.

First, we bound the total weight of nodes $v$ which could cause over-counting. To this end, we say that a node is *bad* if its interval contains $x$ and, furthermore, its interval is not the length-1 interval containing only $x$. Then, we show the following.

**Proposition V.16.** *The total weight of all bad nodes, across all layers, is at most $\gamma_0 n_0$.*

**Algorithm 6** Answering a rank query

```
procedure RANK(x)
    r ← 0
    for all i ∈ {0, ..., k} do
        for all vertices v ∈ T_i do
            if the interval of v contains any element less or equal to x then
                r ← r + W_i[v]
    return r
```

*Proof.* Let $w_i$ denote the total weight of all bad nodes in $T_i$ (for $i < k$, this is just $\alpha_i$ times the number of full bad nodes in layer $i$). Moreover, let $c_i$ denote the total *capacity* of all bad nodes in layer $i$, even the empty ones[14] (this is $\alpha_k$ times the total number of bad nodes in layer $k$).

We will prove the following statement for $0 \le i \le k$ by induction:

$$w_0 + \cdots + w_{i-1} + c_i \le \varepsilon_0 n_0 + \cdots + \varepsilon_i n_i. \tag{2}$$

For the base case $i = 0$, there are $h_0$ bad nodes in layer 0 (namely, the strict ancestors of the node in the base level which corresponds to $x$). Therefore we have $c_0 = h_0\alpha_0 \le \varepsilon_0 n_0$.

Now, assume that (2) holds for $i-1$ (where $1 \le i \le k$); we will show that it also holds for $i$. Consider the quantity $c_i$, the total capacity of bad nodes in layer $i$. Above the base level of $T_i$, at most one node in each level is bad (since the intervals in a level are disjoint). Thus, the total contribution from these nodes to $c_i$ is at most $h_i\alpha_i \le \varepsilon_i n_i$.

On the other hand, each bad node in $T_i$ which is at or below the base level corresponds to an *empty* bad node in $T_{i-1}$. Note that the total capacity of empty bad nodes in $T_{i-1}$ is just $c_{i-1} - w_{i-1}$. Moreover, since $\alpha_i \le \alpha_{i-1}$ (by Fact V.20(d)), the capacity of each node at or below the base level of $T_i$ is at most the capacity of the corresponding empty bad node in $T_{i-1}$. Thus, the total capacity of bad nodes in $T_i$ which are at or below the base level is at most $c_{i-1} - w_{i-1}$. Therefore, in total, the total capacity of all bad nodes in $T_i$ is at most

$$c_i \le \varepsilon_i n_i + c_{i-1} - w_{i-1}.$$

Recall also that by the inductive hypothesis, we have

$$w_0 + \cdots + w_{i-2} + c_{i-1} \le \varepsilon_0 n_0 + \cdots + \varepsilon_{i-1} n_{i-1}.$$

Combining these two inequalities, we recover (2).

Having proven (2), it remains to complete the proof of Proposition V.16. Indeed, setting $i = k$ in (2) and using the fact that $c_i \ge w_i$, the total weight of all bad nodes is at most

$$\varepsilon_0 n_0 + \cdots + \varepsilon_i n_i \le (\varepsilon_0 + \cdots + \varepsilon_i) n_0 = \gamma_0 n_0,$$

as desired. □

[14]For layer $i = k$ specifically, $c_i$ includes also the non-full nodes.

**Proposition V.17.** *At any time $t$, suppose that $\pi$ is the stream received so far. Then there exists a decomposition $\pi = \pi_0 \circ \pi_1 \circ \cdots \circ \pi_k$ such that*

$$\sum_{i=0}^{k} \mathrm{disc}(\pi_i, T_i) \le 2\gamma_1 t.$$

*Proof.* Let $\pi_0$ be the first $\lfloor t/n_1 \rfloor \cdot n_1$ elements of $\pi$, $\pi_1$ be the next $\lfloor t/n_2 \rfloor \cdot n_2 - |\pi_1|$ elements, $\pi_2$ be the next $\lfloor t/n_3 \rfloor \cdot n_3 - |\pi_1 \circ \pi_2|$ elements, and so on. In general, $\pi_i$ is the next $\lfloor t/n_{i+1} \rfloor \cdot n_{i+1} - |\pi_1 \circ \pi_2 \circ \cdots \circ \pi_{i-1}|$ elements in $\pi$ after those in $\pi_{i-1}$. Specifically, we let $n_{k+1} = 1$.

By Corollary V.15, we know that $\mathrm{disc}(\pi_i, T_i) \le 2\gamma_{i+1} \cdot |\pi_i|$. Thus,

$$\begin{aligned}\sum_{i=0}^{k} \mathrm{disc}(\pi_i, T_i) &\le 2\gamma_1|\pi_0| + 2\gamma_2|\pi_1| + \cdots + 2\gamma_k|\pi_k| \\ &\le 2\gamma_1(|\pi_0| + |\pi_1| + \cdots + |\pi_k|) \\ &= 2\gamma_1 t,\end{aligned}$$

so we are done. □

These two propositions imply a bound on the error of a rank query:

**Proposition V.18.** *Let $\pi$ be the stream received so far at time $t$. Then, the answer to a rank query, as performed by Algorithm 6, for any element $x$ differs from $\mathrm{rank}_\pi(x)$ by at most $\gamma_0 n_0 + 2\gamma_1 t$.*

*Proof.* Let $\pi = \pi_0 \circ \pi_1 \circ \cdots \circ \pi_k$ be the decomposition from Proposition V.17. Combine Proposition V.17 with the definition of discrepancy (Definition V.9), we know that there exists a sequence of partial streams $\{\pi'_i\}_{i=0}^k$ such that $\pi'_i$ is consistent with $T_i$ and $\sum_{i=0}^k d(\pi_i, \pi'_i) \le 2\gamma_1 t$.

Let $\pi' = \pi'_1 \circ \pi'_2 \circ \cdots \circ \pi'_k$. By the triangle inequality (Observation II.1), we know that $d(\pi, \pi') \le 2\gamma_1 t$. Since we answered the query by counting the total weight of nodes whose intervals include any element which is at most $x$, the quantity obtained is at least $\mathrm{rank}_{\pi'}(x)$, and may overcount at nodes whose interval also contains an element larger than $x$. However, note that any such node must be bad, so the total amount by which the algorithm overcounts is at most $\gamma_0 n_0$ by Proposition V.16. Thus, the output of the algorithm differs from $\mathrm{rank}_{\pi'}(x)$ by at most $\gamma_0 n_0$. Furthermore, by Proposition V.17 (and the definition of distance of streams), we have $|\mathrm{rank}_\pi(x) - \mathrm{rank}_{\pi'}(x)| \le 2\gamma_1 t$, so the conclusion follows. □

Now, since $\gamma_0, \gamma_1 \le \varepsilon/4$ (by Fact V.20(f)), this already means that the error of a rank query is at most $\varepsilon n_0$. However, so far we have still assumed that we know $n$ in advance; moreover, we would actually like the error to be at most $\varepsilon t$, where $t$ is the total number of elements received so far. In Section V-E, we will explain how to rectify this.

### E. Removing assumptions about $n$

In this section, we will describe how to dispense with the assumption that we know $n$, as well as the assumption that $n \geq n^*$. We will also prove that the error of any query is at most $\varepsilon t$.

*a) Unknown $n$:* First, we describe how to maintain the data structure when we don't know $n$ in advance, but still assuming that all queries happen after $t \geq n^*$. At the start of the algorithm, we initialize the data structure with $n_0 = n^*$. Then, whenever $t$, the number of elements so far in the stream, reaches $n_0$, we double $n_0$ (which has the effect of doubling $n_i$ and $\alpha_i$ for all $i$). Note that when $t = n_0$, only layer 0 exists, so we only need to describe how to update layer 0. Every node in layer 0 is now half-full instead of being full; that is, the weight of every node in $F_0$ is now $\alpha_0/2$. Then, we just perform the push-up and rounding, as described in Algorithms 3 and 4, to layer 0. The pseudocode of this procedure is given in Algorithm 7, and it is called in Line 13 of Algorithm 1.

By Lemma V.13, this has the effect of changing the stream represented by layer 0 by a distance of at most $\alpha_0 = \varepsilon_0 n_0 / \lceil\!\lceil \log(\varepsilon_0 U_0) + 1 \rceil\!\rceil \leq \varepsilon t/16$ (since we assumed that $\varepsilon U$ is sufficiently large). Then, at any point in the stream, the total amount the represented stream has been changed by these rounding operations is at most $\varepsilon n_0/16 + \varepsilon n_0/32 + \cdots \leq \varepsilon n_0/8$. Therefore, the bound on distance between $\pi$ and $\pi'$ in Proposition V.17 is increased by $\varepsilon n_0/16$ after adding the doubling step to the algorithm.

Therefore, after this modification to the algorithm, the proof of Proposition V.18 now gives a bound of $\gamma_0 n_0 + \gamma_1 t + \varepsilon n_0/16$. Since $n_0 \leq 2t$ (since we assumed that $t \geq n^*$), and $\gamma_0 \leq \varepsilon/4$ and $\gamma_1 \leq \varepsilon/8$ (by Fact V.20(f)), we have

$$\gamma_0 n_0 + \gamma_1 t + \varepsilon n_0/16 \leq \varepsilon t.$$

In conclusion, for any $t \geq n^*$, the additive error of any rank query after $t$ elements of the stream is at most $\varepsilon t$, as desired. It remains, then, to handle the cases where $t < n^*$.

**Algorithm 7** Doubling size of data structure

```
1: procedure DOUBLE()
2:     for all i ∈ {0, ..., k} do
3:         n_i ← 2n_i
4:         α_i ← 2α_i ▷ The algorithm doesn't actually store
           n_i or α_i; however, this does affect F_0 since the
           nodes in W_0 are now half-full instead of full.
5:     COMPRESS(0)
6:     ROUND(0)
```

*b) Dealing with $1/\varepsilon \leq t < n^*$:* Next, we describe how to modify the algorithm to still be able to answer queries when $1/\varepsilon \leq t < n^*$. Firstly, we still store the original data structure, since we will need to use it after $t$ exceeds $n^*$. However, in addition, we create a new instantiation of the data structure (with the same parameters), where upon receiving an element of the stream, instead of inserting it once, we insert the same element $\varepsilon n^*$ times (by Fact V.20(i), this is an integer). Then, as long as $t \geq 1/\varepsilon$, we will have inserted at least $n^*$ elements into this alternate data structure, so by the previous section, it will be able to answer rank queries with relative error at most $\varepsilon$, as desired. Of course, the effective value of $t$ will have increased by a factor of $\varepsilon n^*$, which will have ramifications for the space complexity. However, we will show in Section V-G that the space complexity is still what we want it to be.

*c) Dealing with $t < 1/\varepsilon$:* Finally, while $t < 1/\varepsilon$, we will just store all the elements of the stream so far explicitly (in addition to keeping the data structures of the previous two sections). We will show in Section V-G that this can actually be done using $O(\varepsilon^{-1} \log(\varepsilon U))$ space. Obviously, if we store all the elements of the stream, rank queries can be answered exactly.

### F. Choosing the parameters

We will now choose values for the parameters of the algorithm ($k$, $n_i$, $U_i$, and $\varepsilon_i$) and verify that they satisfy some necessary properties.

First, note that we may assume that $n$, $U$, and $\varepsilon$ are all powers of 2 (by rounding $n$ and $U$ up and $\varepsilon$ down to the nearest power of 2, costing at most a constant factor). Indeed, we will ensure that $n_i$, $U_i$, $\varepsilon_i$, and $\alpha_i$ are always powers of 2, in order to stave off divisibility issues.

We then pick the following values. Let $k = \log^*(\varepsilon U)$. As described in Section V-A, let $U_0 = U$. Let $n_0$ be an upper bound on $t$, the number of elements so far in the stream. As previously described, we will imagine for now that we know $n$ in advance and that $n_0 = n$. Also, we assume, as we may, that $n_0$ is a power of 2. We then pick $\varepsilon_i$ as follows:

$$\varepsilon_i = \begin{cases} \varepsilon/8, & i = 0, \\ \varepsilon/2^{k-i+4}, & i \geq 1. \end{cases}$$

Also, define

$$\gamma_i = \varepsilon_i + \varepsilon_{i+1} + \cdots + \varepsilon_k.$$

Also, recall from Section V-A that for all $i$, we define the capacities $\alpha_i$ based on $\varepsilon_i$, $n_i$, and $U_i$as follows:

$$\alpha_i = \frac{\varepsilon_i n_i}{\lceil\!\lceil \log(\varepsilon_i U_i) + 1 \rceil\!\rceil}. \tag{3}$$

Now, we define the parameters $n_i$ and $U_i$ for layer $i+1$ recursively (for $i < k$) as follows:

$$U_{i+1} = \left\lceil\!\!\left\lceil \frac{1}{\varepsilon_i} + \frac{n_i}{\alpha_i} \right\rceil\!\!\right\rceil = \left\lceil\!\!\left\lceil \frac{1 + \lceil\!\lceil \log(\varepsilon_i U_i) + 1 \rceil\!\rceil}{\varepsilon_i} \right\rceil\!\!\right\rceil = \frac{2\lceil\!\lceil \log(\varepsilon_i U_i) + 1 \rceil\!\rceil}{\varepsilon_i}, \tag{4}$$

$$n_{i+1} = \frac{\alpha_i}{\varepsilon_{i+1}} = \frac{\varepsilon_i n_i}{\varepsilon_{i+1} \lceil\!\lceil \log(\varepsilon_i U_i) + 1 \rceil\!\rceil}. \tag{5}$$

We let $h_i$ be the depth of the base layer in $T_i$:

$$h_i = \log(\varepsilon_i U_i).$$

We will show soon that indeed $h_i$ is always a positive integer.

Now, so far we have treated $n_0$ as fixed, but this assumption will change later in Section V-E. In anticipation of this, we will briefly discuss here the effects of changing $n_0$. Treating $\varepsilon, U$ as constants, note that the only parameters that are affected by $n_0$ are the $n_i$ and $\alpha_i$, which are all constant multiples of $n_0$. We will need all the $n_i$ and $\alpha_i$ to be integers (or equivalently, at least 1), so to this end, define

$$n^* = \frac{n_0}{\alpha_k}. \tag{6}$$

Then, $n^*$ is fixed (i.e., it depends only on $\varepsilon, U$ and not on $n_0$). Note that $n^*$ is the value of $n_0$ that causes $\alpha_k$ to equal 1 (and we will show in Fact V.20(e) that it will also cause the rest of the $n_i, \alpha_i$ to be integral).

Now we will check some properties of these parameters which we will need. First, we will show the important property of $U_i$: that it is an upper bound on the number of exposed nodes in the previous layer.

**Claim V.19.** For all $0 \le i < k$, we have $U_{i+1} \ge |V_i|$ (recall that $V_i$ is the set of exposed nodes in layer $i$).

*Proof.* The number of full nodes in layer $i$ is at most $n_i/\alpha_i$ (since full nodes have weight $\alpha_i$. If there are no full nodes, then we would have $|V_i| = 1/\varepsilon_i$, since $V_i$ would just be the set of all the roots of trees in $T_i$. Now, imagine building up the set of full nodes by adding them one at a time (from bottom to top). Each time we add a full node, we remove one exposed node, and add back at most two exposed nodes. Thus, the total number of exposed nodes after this process is at most $1/\varepsilon_i + n_i/\alpha_i$, which is indeed at most $U_{i+1}$ by (4). □

Now, we will prove various other properties of the parameters which we will need throughout. We state all these properties now, but we will defer their proof to Appendix A, since they mostly just involve manipulation of the definitions of the parameters.

**Fact V.20.** The parameters satisfy the following properties:

(a) For all $i$, $n_i$, $U_i$, $\varepsilon_i$, and $\alpha_i$ are powers of 2.
(b) For all $i$, $\varepsilon_i U_i \ge 2$ (and thus, $h_i$ is a positive integer).
(c) For all $i < k$, $n_{i+1}$ is a factor of $n_i$.
(d) For all $i < k$, $\alpha_{i+1} = \alpha_i / \lceil h_{i+1} + 1 \rceil$.
(e) If $n_0 \ge n^*$, then $n_i, \alpha_i \ge 1$ for all $i$.
(f) $\gamma_0 \le \varepsilon/4$, and $\gamma_i \le \varepsilon/8$ for all $i > 1$.
(g) $U_k = O(1/\varepsilon)$.
(h) $U_1 + U_2 + \cdots + U_k = O(\varepsilon^{-1} \log(\varepsilon U))$.
(i) $\varepsilon^{-1} \le n^* \le \varepsilon^{-1} (\log(\varepsilon U))^{1+o(1)}$ (where $o(1)$ refers to a term that approaches 0 as $\varepsilon U \to \infty$).
(j) $\alpha_{k-1} = O(n_0/n^*)$.

### G. Space complexity

Now we discuss the space complexity of the algorithm. All space complexities in this section will be in bits, not words. There are two primary things to check: the space taken by the sketch itself, and the space required during a merge step after an insertion.

*a) Space of sketch:* The information stored by the algorithm consists only of the full nodes $F_i$ for layers $0 \le i < k$ and the weights $W_k$ for layer $k$. (Note that we don't need to store $T_i$ since it is determined recursively by $T_{i-1}$ and $F_{i-1}$.)

Each $F_i$ is an upward-closed subset of $T_i$. In each of the $1/\varepsilon_i$ trees that comprise $T_i$, the portion of $F_i$ in that tree (if nonempty) is a connected subgraph including the root. Thus, that portion of $F_i$ is uniquely determined by the topology of the (rooted) tree that it forms (where in a tree topology we ). We can store the topology of an $\ell$-vertex tree using $O(\ell)$ bits (by storing the bracket representation of the tree). The total number of full nodes in $F_i$ is at most $n_i/\alpha_i$ at any time, so this means that the total space to store $F_i$ is $O(1/\varepsilon_i + n_i/\alpha_i)$, which is just $O(U_{i+1})$ by (4). Thus, the total space to store all the $F_i$ is $O(U_1 + \cdots + U_k)$, which is $O(\varepsilon^{-1} \log(\varepsilon U))$ by Fact V.20(h).

Now, it remains to check the space required to store $W_k$. First, the keys of $W_k$ also form an upward-closed subset of $T_i$. This subset consists of full and partial nodes; by the same argument, there are at most $n_k/\alpha_k = O(1/\varepsilon)$ full nodes. Every partial node is either a root (of which there are $O(1/\varepsilon_k) = O(1/\varepsilon)$) or a child of a full node, so there are also at most $O(1/\varepsilon)$ partial nodes. Therefore, as with the $F_i$, the space required to store the set of all nonempty nodes is at most $O(1/\varepsilon_k + 1/\varepsilon) = O(1/\varepsilon)$.

After the set of nonempty nodes has been stored, we just need to store their weights[15] in some order (say pre-order of the trees). The weights are all at most $\alpha_k = n_0/n^*$, and there are $O(1/\varepsilon)$ of them, so the space required to store all the weights is at most $O(\varepsilon^{-1} \log(n_0/n^*))$. Since $n^* \ge 1/\varepsilon$ (by Fact V.20(i)) and $n_0 \le \max(2n, n^*)$ at all times, we have $O(\varepsilon^{-1} \log(n_0/n^*)) \le O(\varepsilon^{-1} \log(\varepsilon n))$.

Putting everything together, the total space complexity of the data structure is at most

$$O(\varepsilon^{-1}(\log(\varepsilon U) + \log(\varepsilon n)),$$

as desired.

*b) Space of sketch while $t$ is small:* Recall that in section Section V-E, we made two modifications to the data structure that lasted while $t < n^*$ and $t < 1/\varepsilon$. We will show now that (asymptotically) they don't require any extra space.

First, while $t < n^*$, we maintained a second data structure identical to the first, except that we repeated each element $\varepsilon n^*$ times. For this data structure, the space analysis that we just performed still holds, except that $n_0$ may now be up to $2\varepsilon n^* t$. The space to store the $F_i$ is unchanged. The space required

[15] Actually, we only need to store the weights of the leaves of the forest formed by the nonempty nodes, since the rest are full. Since it doesn't make a difference to the asymptotic space complexity, we store all the weights for simplicity.

to store the weights is now at most $O(\varepsilon^{-1}\log(n_0/n^*)) \le O(\varepsilon^{-1}\log(\varepsilon t))$, which is still at most $O(\varepsilon^{-1}\log(\varepsilon n))$, as desired.

Finally, for $t < 1/\varepsilon$, we stored all the elements of the stream explicitly. Naively, storing these as an ordered list would take $O(e^{-1}\log U)$ space, but actually, since the set is unordered, we can improve this. Indeed, split the universe $[1, U]$ into $1/\varepsilon$ buckets of size $\varepsilon U$ (based on the $\log \varepsilon^{-1}$ most significant bits). Then, for each bucket, store an ordered list of the $\log(\varepsilon U)$ least significant bits of every stream element in that bucket. Storing such an ordered list of length $\ell$ takes $O(1+\ell\log(\varepsilon U))$ space, so the total space taken is at most $O(1/\varepsilon + t\log(\varepsilon U)) \le O(\varepsilon^{-1}\log(\varepsilon U))$, which is at most a constant multiple of the desired space.

This completes the discussion of the space taken by the sketch itself. Now we will show that the algorithm does not require any extra space (asymptotically) during the merge operation.

*c) Space during merge:* During the merge, the only extra memory we require is that of storing the keys (i.e., vertices) of the map $W_{i-1}$ which weren't already stored in $F_{i-1}$. There are two parts of this: we need to store the new keys of $W_{i-1}$ (that is, the vertices with newly added weight), and we need to store the weights themselves.

Let $S$ denote the set of new keys of $W_{i-1}$. Note that every node in $S$ corresponds to at least one node from $F_i$ which put its weight into that node. Thus, we have $|S| \le |F_i|$. Additionally, $S \cup F_{i-1}$ form an upward-closed set in $T_{i-1}$. Thus, just as we stored $F_{i-1}$, we can also store $S \cup F_{i-1}$ using $|S \cup F_{i-1}| \le |F_i| + |F_{i-1}|$ space. Note that we already used $|F_i| + |F_{i-1}|$ space for the original sketch, so storing $S$ does not require any more space asymptotically.

Now, it remains to store the weights in $W_{i-1}$. Here we must distinguish between the cases $i = k$ and $i < k$. If $i = k$, then we store the weights explicitly. The weights always remain at most $\alpha_{k-1} = O(\alpha_k) = O(n_0/n^*)$ (by Fact V.20(j)), so the total space required to store the weights is $O(|S|\log(n_0/n^*))$. Since $|S| \le F_k = O(1/\varepsilon)$, this is then at most the weight allocated to store $W_k$ originally, so again this does not require extra asymptotic space.

If $i < k$, then we first make one small optimization: as stated in a footnote, in Algorithm 3 (the compression algorithm), we do not need to move the weight up in increments of 1. Indeed, the weights start out as multiples of $\alpha_i$, and the threshold $\alpha_{i-1}$ is also a multiple of $\alpha_i$. Thus, we can move weight in increments of $\alpha_{i-1}$, so that the weights in $W_k$ always remain multiples of $\alpha_i$. Now, since the weights are all multiples of $\alpha_i$, we can store their ratios with $\alpha_i$; we store the ratios in *unary*, so that storing a weight of $\ell\alpha_i$ requires $O(\ell+1)$ bits of space. Then, the total space needed to store the weights is $O(n_i/\alpha_i + |S|)$. Again, $|S| \le F_i$, so we can see that this is again at most the weight allocated to storing $F_i$ originally.

Thus, we have shown that in all cases, the merge step does not require any more space (asymptotically) than storing the sketch already does.

### H. Runtime

In this section, we prove that, for reasonably-sized $n$, our algorithm processes updates and queries in $O(\log(1/\varepsilon))$ amortized time. We will need a few technical assumptions and simplifications to make our algorithm run in $O(\log(1/\varepsilon))$ time. The first is that we relax the space requirement a bit to $O(\varepsilon^{-1}(\log(\varepsilon n) + \log U))$ bits, which still within $O(\varepsilon^{-1})$ words. Secondly, we assume that $n > (\log U)^C/\varepsilon^2$ for some absolute constant $C$ that depends on the computational model. Also, we assume that there are no queries during the first $(\log U)^C/\varepsilon^2$ insertions.

*Insertion into the last layer:* Our procedure for insertion, Algorithm 1, contains two steps. The first step is to insert the new element $x$ into the last-layer sketch $T_k$. The second step is to merge the layer $i$ into $i-1$ (Algorithm 5).

Now, let us focus on the time complexity of the first step (Lines 3 and 4 of Algorithm 1). The reason we relax the space requirement a little is to allow us to store the tree $T_k$ at the last layer explicitly, not in the bracket representation. There are at most $3|F_k| \le 3 \cdot \frac{n_k}{\alpha_k} = O(1/\varepsilon)$ nodes in the last layer. For each node $u \in T_k$, we store its weight $W_k[u]$ (which takes $O(\log(\varepsilon n))$ bits) and the interval $[a_u, b_u]$ (which takes $O(\log U)$ bits).

To efficiently find the highest non-full node containing $x$, we always maintain a sorted list of all exposed nodes (non-full nodes whose parent is full and the non-full roots). By Observation V.5, these nodes have disjoint intervals whose union covers the entire $[U]$. Thus these nodes are simply sorted in the increasing order of these intervals. A binary search in $O(\log 1/\varepsilon)$ time finds the exposed node (which is also the highest non-full node) $u$ whose interval contains $x$. Then, we increase the weight $W_k[u]$ of that node by 1.

In the rare case where the node $u$ becomes full after this, we need to remove it from the list and add its two empty children. Although this takes $O(1/\varepsilon)$ time as we have to modify the entire list and the topology of the tree we store, it only happens once every $\alpha_k = \frac{n_0}{n^*}$ (Equation (6)) insertions. Here $n_0$ is the current estimate of string length, which keeps doubling as explained Section V-E. Since we know that $n > (\log U)^C/\varepsilon^2$ from our assumption, we can run the algorithm starting with $n_0 = (\log U)^C/\varepsilon^2$. As $n^* \le \varepsilon^{-1}(\log(\varepsilon U))^{1+o(1)}$ (Fact V.20(i)), we have $\alpha_k \ge O((\log U)^{C-1}/\varepsilon)$. We can amortize the $O(1/\varepsilon)$ running time to these $\alpha_k$ insertions and get $O(1)$ amortized running time for updating the list.

*Merging layer $i$ into layer $i-1$:* First of all, in each tree $T_i$, the number of all nodes is $|F_i| \le \frac{n_i}{\alpha_i} = O(\log(\varepsilon_i U_i)/\varepsilon_i)$ ($\varepsilon_i = \varepsilon/2^{k-i+4}$). We want to amortize the time cost to $n_i$ insertions. For Algorithm 5, there are three procedures which we will analyze one by one.

- MOVE($i$) (Algorithm 2): At Line 5, we need to find the base-level descendant $v'$ of $v$ for every node $v \in F_i$ above the base level. This can be done by traversing the stored part of tree $T_i$ once, which takes $|F_i|$ time.
  In the rest of this algorithm, since we only maintain the full nodes $F_{i-1}$ in $T_{i-1}$, in this step, all the empty nodes in $T_{i-1}$ whose weights increase are not stored before by our algorithm. We simply store them and their weights as a list using $O((\log U + \log(\varepsilon n)) \cdot |F_i|)$ bits of memory in the depth-first-search order. This takes $O(|F_i|)$ time.
- COMPRESS($i-1$) (Algorithm 3): In the time efficient implementation of COMPRESS($i-1$), instead of moving weights one unit at a time, we process the nodes in the list we stored during Algorithm 2 in top-down order and always move the maximum amount of weight that we can move. Since this process moves weight up at most once from each node, it also only takes $O(|F_i|)$ time.
- ROUND($i-1$) (Algorithm 4): Finally, Algorithm 4 finds the partial nodes in our list while visiting each node at most once. So this takes only $|F_i|$ time as well.

After these three steps, we also have to update the topology of $F_{i-1}$ and add new full nodes to its bracket representation. This takse $|F_{i-1}|$ time. In total, the time complexity is $|F_{i-1}|+|F_i|$. So the amortized time is $(|F_{i-1}|+|F_i|)/n_i = O\left(1/\alpha_i\right) \leq O(1/\alpha_k)$ per layer $i$. As there are $k = \log^* U$ many layers, while $\alpha_k \geq (\log U)^{C-1}/\varepsilon$, the amortized time cost is just $O(1)$.

*Answering rank queries:* For answering rank queries, running exactly Algorithm 6 requires traversing $T_0, T_1, \ldots, T_k$, which takes $O(\sum_{i=0}^{k} |F_i|) = O((\log U)/\varepsilon)$ time. For simplicity, we assume that there are only queries after first $n_0$ elements are inserted. After every $\varepsilon \cdot n_0$ insertions, we run Algorithm 6, compute each $\varepsilon$-approximate quantile and store them. This takes at most $O((\log U)/\varepsilon^2)$ time. Then for every query $x$, we just binary search in $O(\log 1/\varepsilon)$ time, and count the number of stored quantile elements less than that $x$, multiply that by $\varepsilon t$ (where $t$ is the number of current insertions), and output the answer. This has an error of at most $2\varepsilon n$. Since we can amortize the $O((\log U)/\varepsilon^2)$ time cost to $\varepsilon \cdot n_0 \geq (\log U)^C/\varepsilon^2$ elements. This takes $O(\log 1/\varepsilon)$ amortized time per query and $O(1)$ amortized time per insertion.

## VI. PRACTICAL CONSIDERATIONS

*a) Mergeability.:* One popular feature with quantile sketches is being *fully-mergeable*, meaning that any two sketches with the error parameter $\varepsilon$ can be merged into a single sketch without increasing the error parameter $\varepsilon$. A weaker notion of mergeability is the *one-way mergeability*, which, informally speaking, means that it is possible to maintain an accumulated sketch $S$ and keep merging other small sketches into $S$ without increasing the error $\varepsilon$. As pointed out in [10], [14], every quantile sketch is one-way mergeable.

Among these sketches, the GK sketch and the optimal KLL sketch is not fully mergeable, while q-digest is fully mergeable, and KLL sketch has a mode in which it is fully mergeable but loses its optimal space bound. Our sketch is based on the fully mergeable Q-digest sketch, but we do not know whether it is fully mergeable in its current form. We leave it as a future direction to come up with a fully mergeable mode for our algorithm.

However, our algorithm is in a sense partially mergeable. That is, if we have two instances of size at most $n$ each with error parameter $\varepsilon$, we can merge them while incurring an additional discrepancy of at most $O(\varepsilon n/\log(\varepsilon U))$ (as we will soon describe). Though this is not as strong as a fully-mergeable data structure, which incurs additional error of 0, it is still better than the $O(\varepsilon n)$ additional error incurred by merging quantile sketches in a black-box sense (by querying their quantiles to obtain an $O(\varepsilon n)$-approximation to their streams). In practice, this means that one can merge up to $\mathrm{poly}(U)$ of our sketches simultaneously (by performing merges in a binary tree with depth $O(\log(\varepsilon U))$), with only a constant-factor loss in $\varepsilon$.

We now sketch how to perform this partial merge. Suppose we wish to merge the data structures $D$ and $D'$, with current sizes $t > t'$. To begin with, let us first imagine that only layer 0 is occupied (in both structures). Then, we simply add values of the weight map $W_0'$ (of $D'$) into $W_0$ (of $D$). Then, the discrepancy of $W_0$ is now $\varepsilon t + \varepsilon t'$. Now, the only problem is that the invariant that all nodes are either full or empty may not hold anymore, and the full nodes are no longer upward-closed. To fix this, we perform the compression and rounding steps of Algorithms 3 and 4 — by Lemmas V.12 and V.13, this increases the discrepancy by at most $\alpha_0 = O(\varepsilon t/\log(\varepsilon U))$. If there is now a doubling step (Algorithm 7) to be performed (that is, if $t_0 + t_0' \geq n_0$), then we now do it as usual. Note that though the discrepancy has increased, the data structure is otherwise still a valid data structure for the error parameter $\varepsilon$, and we can continue to perform the usual operations (including more merges) on the new data structure, while keeping track of the increased discrepancy.

Now, suppose that there are occupied layers other than layer 0. Then, before merging the two data structures, we simply perform the operation MERGE($i$) early for $i = k, k-1, \ldots, 1$, on both data structures. This proceeds identically to an ordinary MERGE(o)peration, except that during the rounding step, the total weight may not be a multiple of $\alpha_{i-1}$; we simply discard the excess weight down to a multiple of $\alpha_{i-1}$ (and insert arbitrary elements to replace them at the end of the merge). Overall, this has the effect of discarding elements down to the nearest multiple of $\alpha_0$, so it will introduce a discrepancy of at most $\alpha_0 = O(\varepsilon t/\log(\varepsilon U))$. Additionally, the proof of Lemma V.14 still shows that the discrepancy introduced by this merge is at most $\gamma_1 n_1 = O(\varepsilon t/\log(\varepsilon U))$. Thus, overall, this partial merge still adds an additional $O(\varepsilon t/\log(\varepsilon U))$ to the discrepancy, as desired.

*b) Constant factors.:* The parameters that we selected in Section V-F were chosen to make the analysis simple. There is, however, a lot of leeway in choosing the parameters to still satisfy the necessary properties, and our exact choices likely do not attain the best constant factors on space complexity. We use $k+1 = \log^*(\varepsilon U)+1$ layers, but in practice, we expect that around 4 layers is probably enough, and the parameters can then be chosen appropriately.

Additionally, beyond just the setting of our parameters, our analysis has generally been wasteful in terms of constants for ease of presentation and readability. There are several places this can be improved. For example, we can improve the error $\varepsilon$ by a factor of 2 by performing the moving and rounding steps of the merge in different directions; that is, in the moving step, we can move nodes only to their leftmost (least) descendant, and in the rounding step, we round nodes upward only (which is what we already do).

*c) Removing amortization.:* Currently, our runtime analysis is amortized, since a step containing a merge can take a long time compared to a normal insertion step. If one is concerned about worst-case update time, then we can improve performance by executing the time-consuming operations over a longer time period while storing received elements in a buffer, similarly to Claim 3.13 of [22].

*d) Answering select queries with real elements.:* One feature of quantile queries is that they can also answer *select* queries: that is, given a rank $r$, one can query $\mathrm{select}(r)$ to obtain an element $x$ that is between the rank-$(r-\varepsilon t)$ and rank-$(r+\varepsilon t)$ elements of the stream. This is equivalent to being able to answer rank queries, since one can use a binary search of rank queries to answer a select query (and vice versa). One might also desire, though, that the answers to the select queries are actual elements of the stream, rather than arbitrary elements of $[1,U]$. As stated, our algorithm does not provide a way to do this. It turns out, however, that given any quantile sketch algorithm that can answer approximate rank queries, it is possible to augment it (in a black-box manner) so that it can answer select queries with real elements of the stream, with only a constant-factor degradation in the error parameter $\varepsilon$. We will now sketch how to do so.

We initialize a quantile sketch with error parameter $\varepsilon$, and we maintain a list $x_1 < x_2 < \cdots < x_\ell$ which are actual elements of the stream (and by convention we write $x_0 = 0$ and $x_{\ell+1} = U+1$), and rank estimates $r_1, \ldots, r_\ell$ (where again by convention we say $r_0 = 0$) satisfying the following properties at all times $t$:

1) For all $0 \le i \le \ell$, $|\mathrm{rank}_\pi(x_i) - r_i| \le \varepsilon t$.
2) For all $0 \le i \le \ell$, $\mathrm{rank}_\pi(x_{i+1}-1) - r_i \le 2\varepsilon t$.

(Note that the first item is trivially satisfied for $i = 0$.) Now, suppose that we receive an insertion $x$ into the stream. First, we increment $r_i$ for all $i$ such that $x_i \ge r_i$, to maintain property 1 (note that $t$ increases by 1, but this only makes property 1 easier to satisfy).

Now, if $x = x_j$ for some $j$, then property 2 continues to be satisfied since the left-hand side of the inequality remains the same for all $i$. Otherwise, suppose that $x \in (x_j, x_{j+1})$ for some $j$. Then, 2 might become violated for $i = j$, since the left-hand side will have increased by 1. To fix this, we insert a new element $x_{j+1} = x$ (and shift the indices of the existing $x_i, r_i$ of all $i \ge j+1$ up by 1). Then, we execute a rank query on $x$ to get $r$ such that $|\mathrm{rank}_\pi(x) - r| \le \varepsilon t$. Then, we set $r_{j+1} = \max\{r, r_j + 1\}$. Note that property 1 continues to be satisfied by the accuracy of the rank query and because $r_j + 1 \le \mathrm{rank}_\pi(r_j) + \varepsilon t + 1 \le \mathrm{rank}_\pi(r_{j+1}) + \varepsilon t$. It remains to check that property 2 is now satisfied. Indeed, for $i = j+1$, this follows from the fact that $r_{j+1} \ge r_j + 1$ and that the property was previously satisfied for $i = j$. For $i = j$, it follows from the fact that $\mathrm{rank}_\pi(x-1)$ is at most the former value of $\mathrm{rank}_\pi(x_{j+1}-1)$, and that the property was previously satisfied for $i = j$. Thus, we have established that the properties both continue to hold.

Finally, while there is any $j$ such that $r_{j+1} - r_{j-1} \le \varepsilon t$, we delete $x_j$ and $r_j$ (and shift the indices $i > j$ down by 1 to accommodate). This preserves the properties: we only need to check property 2 for $i = j-1$, and indeed, $\mathrm{rank}_\pi(x_j - 1) - r_{j-1} \le (r_j + \varepsilon t) - r_{j-1} \le 2\varepsilon t$ by property 2 and by the assumption that $r_j - r_{j-1} \le \varepsilon t$ (note that the old $r_{j+1}$ has become $r_j$). Thus, this preserves the properties.

Now, we answer a select query as follows: on a query of rank $r$, we pick the minimal $i$ such that $r \le r_i + 2\varepsilon t$, and return $x_i$. As a special case, if $r < 2\varepsilon t$, we return $x_1$ instead of $x_0 = 0$. (Note that by property 2 applied to $i = \ell$, we never return $x_{\ell+1}$.) Then, assuming that $r \ge 2\varepsilon t$, we have by property 1 that $\mathrm{rank}(x_i) \ge r_i - \varepsilon \ge r - 3\varepsilon t$. Also, by property 2, $\mathrm{rank}(x_i - 1) \le r_{i-1} + 2\varepsilon t < r$ (by minimality of $i$), so the rank-$r$ element is at least $x_i$. Thus the error in the select query is at most $O(\varepsilon t)$ as long as $r \ge 2\varepsilon t$. Also, in the special case $r < 2\varepsilon t$, we answer $x_1$, and by property 2, $\mathrm{rank}(x_1 - 1) \le 2\varepsilon t$, so again the error is at most $O(\varepsilon t)$. Thus, the answers to the select queries are always approximately correct.

Finally, it remains to analyze the total space taken. Note that we have $r_{j+1} - r_{j-1} \le \varepsilon t$ for all $j$, so the total number of indices $\ell$ is at most $O(1/\varepsilon)$. Therefore, we only need to store the $O(1/\varepsilon)$ elements $x_1, \ldots, x_\ell$ and $r_1, \ldots, r_\ell$, which takes $O(1/\varepsilon)$ words. Indeed, since the $x_i$ are in increasing order and the increments of the $r_i$ are at most $O(\varepsilon n)$, we can actually store these in $O(\varepsilon^{-1}(\log(\varepsilon U) + \log(\varepsilon n))$ space, so this does not take any additional asymptotic space over our algorithm.

## VII. Lower bounds

The space complexity of our algorithm is $O(\varepsilon^{-1}(\log(\varepsilon U) + \log(\varepsilon n))$. In this section, we'll discuss the optimality of this result. The first term $O(\varepsilon^{-1}\log(\varepsilon U))$ must be incurred by any quantile sketch, even a randomized one that succeeds with reasonable probability, as we will now show. This already implies that when $n \le \mathrm{poly}(U)$, our algorithm is tight[16].

When this is not the case, we conjecture that our algorithm is optimal among deterministic sketches anyway. In particular, Conjecture I.3 implies a space lower bound of $O(\varepsilon^{-1}\log(\varepsilon n))$ for quantiles.

**Theorem VII.1.** *Any randomized streaming algorithm for Problem I.1 that succeeds with probability at least* $0.9$ *(that is, it can answer a rank query chosen by an oblivious adversary with that probability) on a universe of size* $U > C\varepsilon^{-1}$ *for some sufficiently large* $C$ *uses at least* $\Omega(\varepsilon^{-1}\log(\varepsilon U))$ *bits of space.*

*Proof.* It suffices to show that the final state of the algorithm requires $\Omega(\varepsilon^{-1}\log(\varepsilon U))$ bits of space. Let us restrict ourselves to streams that only contain $k = 3\varepsilon^{-1}$ distinct elements, each of which occurs $n/k$ times. Under this model, let the stream be $\pi'_1 < \ldots < \pi'_k$ (each with multiplicity $n/k$). Under this model, the min-entropy of the stream (when the stream is chosen uniformly randomly) is $\log\binom{U}{k}$. We will show that access to the sketch reduces the min-entropy considerably (by at least a constant factor). To do this, we will describe an algorithm for a party to make $\varepsilon^{-1}\log U$ queries to the sketch and with probability at least $0.01$, output at least $0.01$ fraction of the elements $\pi'_1, \pi'_2, \ldots, \pi'_k$ correctly. The min-entropy of this distribution of outputs is much lower: the only possibilities are those that overlap on at least $0.01$-fraction of $\pi'_1 \ldots \pi'_k$, of which there are at most $\binom{k}{0.01k}\binom{U}{0.99k}$. The most likely outcome therefore occurs with probability at least $0.01$ times the log of this quantity, so the min-entropy has decreased by at least

$$\log\binom{U}{k} - \log\left(100\binom{k}{0.01k}\binom{U}{0.99k}\right) \geq \big(\Omega(\varepsilon^{-1}\log(\varepsilon U))\big)$$

by Stirling's approximation when $U > C\varepsilon^{-1}$ for a sufficiently large $C$. Then, by the fact below, the sketch must have contained at least this many bits of information.

**Fact VII.2.** Let $H_{\min}(\cdot)$ denote the min-entropy of random variables. For any two random variables, $\mathbf{x}$ and $\mathbf{y}$ supported on $X$ and $Y$ respectively, we have

$$H_{\min}(\mathbf{x}) - H_{\min}(\mathbf{x} \mid \mathbf{y}) \leq H(\mathbf{y}).$$

In our case, $\mathbf{x}$ is the elements $\pi'_1, \pi'_2, \ldots, \pi'_k$ and $\mathbf{y}$ is the memory state of our algorithm.

[16]Technically speaking, this result alone only implies tightness when $n \leq \mathrm{poly}(\varepsilon U)$. However, if $U > 1/\varepsilon^2$, then $\mathrm{poly}(\varepsilon U)$ and $\mathrm{poly}(U)$ are the same, and when $U < 1/\varepsilon^2$, then $n \leq \mathrm{poly}(U)$ implies that $n \ll \mathrm{poly}(1/\varepsilon)$, and as we discussed in Section I-A, a result of [11] implies that our algorithm is tight when $\varepsilon^{-1} > \log(\varepsilon n)$.

*Proof.*

$$\begin{aligned}
&H_{\min}(\mathbf{x}) - H_{\min}(\mathbf{x} \mid \mathbf{y}) \\
&= H_{\min}(\mathbf{x}) - \sum_{y\in Y} \Pr(\mathbf{y} = y) \min_{x\in X} \log \frac{1}{\Pr(\mathbf{x} = x \mid \mathbf{y} = y)} \\
&= H_{\min}(\mathbf{x}) - \sum_{y\in Y} \Pr(\mathbf{y} = y) \min_{x\in X} \log \frac{\Pr(\mathbf{y} = y)}{\Pr(\mathbf{x} = x, \mathbf{y} = y)} \\
&\leq H_{\min}(\mathbf{x}) - \sum_{y\in Y} \Pr(\mathbf{y} = y) \min_{x\in X} \log \frac{\Pr(\mathbf{y} = y)}{\Pr(\mathbf{x} = x)} \\
&= H_{\min}(\mathbf{x}) - \min_{x\in X} \log \frac{1}{\Pr(\mathbf{x} = x)} + \sum_{y\in Y} \Pr(\mathbf{y} = y) \frac{1}{\Pr(\mathbf{y} = y)} \\
&= H(\mathbf{y})
\end{aligned}$$

□

Now we describe the list of queries to ask the sketch to output least $0.01$ fraction of the elements $\pi'_1 \ldots \pi'_k$ correctly with probability $0.01$. For each rank $i \in [k]$, binary search for the rank $i$'th element in a noise resilient way [27] (resilient to $0.2$ fraction of adversarial error). At the end, this must find the element at rank $i$ exactly, since each element's multiplicity is more than the permissible error. The noisy binary search must succeed whenever the fraction of error is at most $0.2$, which is true on at least $0.01$ fraction of the elements at least $0.01$ fraction of the time. □

**Theorem VII.3.** *Conjecture I.3 implies that any deterministic streaming algorithm for Problem I.1 uses at least* $\Omega(\varepsilon^{-1}\log(\varepsilon n))$ *bits of space.*

*Proof.* We will show the following. Any data structure that can compute a quantile sketch for $0.1\varepsilon^{-1}$ on $n$ elements in the range $[\varepsilon^{-1}]$ can also return counts of each element that are accurate to within $\pm\varepsilon n$. Then, if there is a quantile sketch using $o(\varepsilon^{-1}\log n)$ bits of memory, there is also a deterministic parallel approximate counter using that much space.

Let us try to comp estimate the count of $i \in [\varepsilon^{-1}]$. The true count of $i$ is the difference of the true ranks $r_i - r_{i-1}$, since the rank $r_j$ is the number of elements at most $j$. We query the rank of $i$ in the quantile sketch and get the answer $\widehat{r}_i$ and the rank of $i-1$ and get $\widehat{r}_{i-1}$. Then,

$$\Big|(r_i - r_{i-1}) - (\widehat{r}_i - \widehat{r}_{i-1})\Big| \leq 0.2\varepsilon n,$$

so we have a sufficiently accurate estimate of the count. □


## ACKNOWLEDGMENTS

We would like to thank Jelani Nelson for his excellent mentorship, and specifically, for pointing us to this problem, helpful discussions, and suggestions for the manuscript. We would also like to thank the others who have provided feedback for drafts of the manuscript, including Lijie Chen, Yang Liu, Naren Manoj, and anonymous FOCS reviewers.

## REFERENCES


[1] J. I. Munro and M. S. Paterson, "Selection and sorting with limited storage," *Theoretical computer science*, vol. 12, no. 3, pp. 315–323, 1980. 1, 2

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394. 1

[3] "Quantiles Sketch Overview - Apache DataSketches," https://datasketches.apache.org/docs/Quantiles/QuantilesSketchOverview.html, accessed: 2024-03-26. 1

[4] "Approximate aggregate functions - GoogleSQL," https://cloud.google.com/bigquery/docs/reference/standard-sql/approximate_aggregate_functions, accessed: 2024-03-26. 1

[5] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794. 1

[6] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 58–66, 2001. 1, 3

[7] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2016, pp. 71–78. 1, 2, 3

[8] G. Cormode and P. Veselỳ, "A tight lower bound for comparison-based quantile summaries," in *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2020, pp. 81–93. 1

[9] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 239–249. 2, 3, 6

[10] M. B. Greenwald and S. Khanna, "Quantiles and equi-depth histograms over streams," in *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2016, pp. 45–86. 2, 18

[11] I. Aden-Ali, Y. Han, J. Nelson, and H. Yu, "On the amortized complexity of approximate counting," *arXiv preprint arXiv:2211.03917*, 2022. 2, 20

[12] Y. Wang, "Tight streaming lower bounds for deterministic approximate counting," *arXiv preprint arXiv:2406.12149*, 2024. 2

[13] D. Felber and R. Ostrovsky, "A randomized online quantile summary in $O((1/\varepsilon)\log(1/\varepsilon))$ words," *Theory of Computing*, vol. 13, no. 1, pp. 1–17, 2017. 2

[14] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, pp. 1–28, 2013. 2, 18

[15] K. Alsabti, S. Ranka, and V. Singh, "A one-pass algorithm for accurately estimating quantiles for disk-resident data," in *Very Large Data Bases Conference*, 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:2157195 2

[16] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 426–435, 1998. 2

[17] A. Gupta and F. Zane, "Counting inversions in lists," in *SODA*, vol. 3, 2003, pp. 253–254. 3

[18] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Space- and time-efficient deterministic algorithms for biased quantiles over data streams," in *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2006, pp. 263–272. 3

[19] Q. Zhang and W. Wang, "An efficient algorithm for approximate biased quantile computation in data streams," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, 2007, pp. 1023–1026. 3

[20] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Veselỳ, "Relative error streaming quantiles," in *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2021, pp. 96–108. 3

[21] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2004, pp. 286–296. 3

[22] S. Assadi, N. Joshi, M. Prabhu, and V. Shah, "Generalizing Greenwald-Khanna streaming quantile summaries for weighted inputs," *arXiv preprint arXiv:2303.06288*, 2023. 3, 19

[23] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *arXiv preprint arXiv:1908.10693*, 2019. 3

[24] T. Dunning and O. Ertl, "Computing extremely accurate quantiles using t-digests," *arXiv preprint arXiv:1902.04023*, 2019. 3

[25] E. Gan, J. Ding, K. S. Tai, V. Sharan, and P. Bailis, "Moment-based quantile sketches for efficient high cardinality aggregation queries," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 2018. 3

[26] G. Cormode, A. Mishra, J. Ross, and P. Veselỳ, "Theory meets practice at the median: A worst case comparison of relative error quantile algorithms," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 2722–2731. 3

[27] A. Pelc, "Searching games with errors—fifty years of coping with liars," *Theoretical Computer Science*, vol. 270, no. 1-2, pp. 71–109, 2002. 20


## APPENDIX

Here, we will prove the various parts of Fact V.20, by showing a series of claims. Note that Fact V.20(f) follows directly from the definitions of $\varepsilon_i$ and $\gamma_i$.

**Claim A.1** (Fact V.20(a))**.** For all $i$, $n_i$, $U_i$, $\varepsilon_i$, and $\alpha_i$ are powers of 2.

*Proof.* This follows directly (inductively) from the definitions. □

**Claim A.2** (Fact V.20(b))**.** For all $i$, $\varepsilon_i U_i \geq 2$.

*Proof.* For $i = 0$, this follows from the assumption (made at the start of Section V) that $\varepsilon U$ is sufficiently large. For $i = 1$, we have $U_1 = 2\lceil\log(\varepsilon U/8) + 1\rceil/\varepsilon$ and $\varepsilon_1 = \varepsilon/2^{k+3}$, so $\varepsilon_1 U_1 = \lceil\log(\varepsilon U/8)+1\rceil/2^{\log^*(\varepsilon U)+2}$, which is at least 2 again by the assumption that $\varepsilon U$ is sufficiently large. Finally, for $i \geq 2$, this follows by induction using the recursive definition of $U_i$ and the fact that $\varepsilon_{i-1} < \varepsilon_i$. □

**Claim A.3.** For all $i < k$, we have $\varepsilon_{i+1}U_{i+1} \leq 8\log(\varepsilon_i U_i)$.

*Proof.* Since $\varepsilon_{i+1} \leq 2\varepsilon_i$, we have by the inductive definition of $U_i$, (4), that

$$\varepsilon_{i+1}U_{i+1} \leq 4\lceil\log(\varepsilon_i U_i) + 1\rceil \leq 8\log(\varepsilon_i U_i).$$

(Here we have used the fact that $\log(\varepsilon_i U_i)$ is a positive integer, which follows from Claim A.1 and Claim A.2.) □

Now, define $Q_i = \varepsilon_i U_i/16$. Then we have the following.

**Claim A.4.** For all $i < k$, we have $Q_{i+1} \leq \max\{\log Q_i, 8\}$.

*Proof.* By Claim A.3, we have

$$\begin{aligned} Q_{i+1} = \frac{\varepsilon_{i+1}U_{i+1}}{16} &\leq \frac{\log(\varepsilon_i U_i)}{2} \\ &\leq \frac{\log(16Q_i)}{2} = \frac{4+\log Q_i}{2} \leq \max\{\log Q_i, 8\}, \end{aligned}$$

as desired. □

**Claim A.5** (Fact V.20(g))**.** $U_k = O(1/\varepsilon)$.

*Proof.* We have $k = \log^*(\varepsilon U) \geq \log^*(Q_0)$, so if we iteratively take the logarithm of $Q_0$, we get down below 1 in at most $k$ steps. Thus, by Claim A.4, we have $Q_k \leq 8$, so $U_k = 16Q_k/\varepsilon_k = O(1/\varepsilon)$. □

**Claim A.6** (Fact V.20(h))**.** $U_1 + U_2 + \cdots + U_k = O(\varepsilon^{-1}\log(\varepsilon U))$.

*Proof.* We have $U_1 = 2\lceil\log(\varepsilon_0 U_0 + 1)\rceil/\varepsilon_0 = O(\varepsilon^{-1}\log(\varepsilon U))$. Meanwhile, for $i > 1$, by Claim A.4, we have $Q_i \leq O(\log\log Q_0) = O(\log\log(\varepsilon U))$. Also, $\varepsilon_i \geq 2^{-k+3}\varepsilon = \Omega(2^{-\log^*(\varepsilon U)}\varepsilon)$. Therefore, for $i > 1$, we have $U_i = O(Q_i/\varepsilon_i) \leq O(\varepsilon^{-1}2^{\log^*(\varepsilon U)}\log\log(\varepsilon U))$. Thus, since $k = \log^*(\varepsilon U)$,

$$\begin{aligned} U_2 + \cdots + U_k &\leq O(\varepsilon^{-1}\log^*(\varepsilon U)2^{\log^*(\varepsilon U)}\log\log(\varepsilon U)) \\ &< O(\varepsilon^{-1}\log(\varepsilon U)), \end{aligned}$$

so we are done. □

**Claim A.7** (Fact V.20(c))**.** For all $i < k$, $n_{i+1}$ is a factor of $n_i$.

*Proof.* Since the $n_i$ are powers of 2, it is enough to check that $n_{i+1} \leq n_i$. For $i \geq 1$, this follows directly from the definition of $n_{i+1}$ since $\varepsilon_{i+1} > \varepsilon_i$ (and because of Claim A.2). For $i = 0$, we get $n_0 = n$ and

$$n_1 = \frac{\varepsilon_0 n_0}{\varepsilon_1\lceil\log(\varepsilon_0 U_0) + 1\rceil} = \frac{2^{\log^*(\varepsilon U)}n_0}{\lceil\log(\varepsilon U/8) + 1\rceil},$$

which is at most $n_0$ by the assumption that $\varepsilon U$ is sufficiently large. □

**Claim A.8** (Fact V.20(d))**.** For all $i < k$, $\alpha_{i+1} = \alpha_i/\lceil h_{i+1} + 1\rceil$.

*Proof.* We have, by the inductive definitions (3) and (5), that

$$\alpha_{i+1} = \frac{\varepsilon_{i+1}n_{i+1}}{\lceil h_i + 1\rceil} = \frac{\alpha_i}{\lceil\log(\varepsilon_{i+1}U_{i+1}) + 1\rceil}.$$

□

**Claim A.9** (Fact V.20(e))**.** If $n_0 \geq n^*$, then $\alpha_i, n_i \geq 1$ for all $i$.

*Proof.* Suppose that $n_0 \geq n^*$. Firstly, by definition of $n^*$, (6), we have $\alpha_k \geq 1$. Also, by the definition of $\alpha_i$, (3), we also have $\alpha_k \leq n_k$, so $n_k \geq 1$. By Claims A.7 and A.8, $n_i$ and $\alpha_i$ are decreasing in $i$, so the conclusion follows. □

**Claim A.10** (Fact V.20(i))**.** $\varepsilon^{-1} \leq n^* \leq \varepsilon^{-1}(\log(\varepsilon U))^{1+o(1)}$ (where $o(1)$ refers to a term that approaches 0 as $\varepsilon U \to \infty$).

*Proof.* By successive applications of Claim A.8 and then using the definition of $\alpha_0$, we have

$$\alpha_k = \frac{\alpha_0}{\lceil h_1 + 1\rceil \cdot \ldots \cdot \lceil h_k + 1\rceil} = \frac{n_0\varepsilon_0}{\lceil h_0 + 1\rceil \cdot \ldots \cdot \lceil h_k + 1\rceil}.$$

Thus, we have

$$n^* = \frac{n_0}{\alpha_k} = \frac{\lceil h_0 + 1\rceil \cdot \ldots \cdot \lceil h_k + 1\rceil}{\varepsilon_0}.$$

Since $\varepsilon_0 = \varepsilon/8$, the first inequality of the claim follows immediately. Now, note that we have

$$\lceil h_i + 1\rceil = O(\log(\varepsilon_i U_i)) = O(\max\{\log Q_i, 1\})$$

Now, this means that $\lceil h_0 + 1\rceil = O(\log(\varepsilon U))$, and for $i > 0$, by Claim A.4, we have $\lceil h_i + 1\rceil \leq O(\log\log\varepsilon U)$. Thus, since $k = \log^*(\varepsilon U)$, we have

$$\begin{aligned} n^* &\leq \frac{O(\log(\varepsilon U)) \cdot (O(\log\log\varepsilon U))^{\log^*(\varepsilon U)}}{\varepsilon/8} \\ &= \varepsilon^{-1}(\log(\varepsilon U))^{1+o(1)}, \end{aligned}$$

as desired. □

**Claim A.11** (Fact V.20(j))**.** $\alpha_{k-1} = O(n_0/n^*)$.

*Proof.* By Claim A.4, we have $Q_k = O(1)$, so by Fact V.20(d), we have $\alpha_{k-1} = \alpha_k\lceil\log Q_k + 1\rceil = O(\alpha_k) = O(n_0/n^*)$. □