A Review of Recent Results in Streaming Quantiles

A Reading Project for MIT 6.854

Abstract— **Karnin, Lang, and Liberty have recently discovered a randomized algorithm for streaming quantile sketches that matches the lower bound, resolving the randomized case of a longstanding open problem. This paper is a beginner's introduction. We build intuition by analyzing several elementary approaches to quantile sketches. Within this framework we explain how the best known deterministic algorithm operates by dropping uncertain items, and how the best previously known randomized algorithm works by maximizing terms that contribute to the Hoeffding concentration inequality. Finally we explain how these techniques are combined and refined to create the new KLL results.**

I. Introduction

Karnin, Lang, and Liberty (KLL) [4] presented a new algorithm at FOCS 2016 that is an optimal solution to the long-standing open problem of efficient online computation of an approximate quantile sketch. Their randomized algorithm uses $O(\varepsilon^{-1}\log\log\delta^{-1})$ memory, and they show this matches the lower bound.

This paper is a technical review and beginner's guide that is designed to help readers understand this exciting result. We introduce the quantile sketch problem by analyzing several important approaches including the previously best known deterministic and randomized algorithms. We culminate in a discussion of the KLL algorithm and the intuition behind its design.

II. The Basic Problem

How can we find the median of a huge stream?

Verifying a median on a stream is trivial in constant space: just count the number of elements above and below the candidate. However, *finding* a median is harder.

A Google search for 'running median' will list dozens of articles with opinions on the best way to compute an exact streaming median. This is all bad advice. Finding an exact median of a one-pass stream is impractical, because it requires $O(n)$ memory.

*This work was done as a class project for MIT 6.854

A. The exact problem is expensive

A simple adversarial stream of odd length n can show that a median-finder must use least $\lceil n/2 \rceil$ memory. Take any algorithm that stores only $m = \lfloor n/2 \rfloor$ items. The adversary begins with $m+1$ distinct values in sorted order, after which some item x_j must not be stored by the algorithm. The adversary completes the stream with $m-1$ items that balance the stream around x_j, adding $j-1$ higher items and $m-j$ lower items. In the end, x_j is the median of the whole stream, and the algorithm cannot output the median since it did not store x_j.

B. The approximate problem is interesting

While exact median-finding is expensive, finding *approximate* stream quantiles can be done economically even in the face of an adversary. In this paper we will analyze several comparison-based methods that can find an item within εn rank of a stream median for any positive ε, while consuming less than n memory:

Sec.	Algorithm	Memory	Based on
III.	Batch	$\sim n/2$	
IV.	MRL [5]	$O(\varepsilon^{-1}\log^2(\varepsilon n))$	Batch
V.	GK [2]	$O(\varepsilon^{-1}\log(\varepsilon n))$	
VI.	Sample (r)[1]	$O(\varepsilon^{-2}\log(\varepsilon^{-1}\delta^{-1}))$	
VII.	ACHPWY [1] (r)	$O(\varepsilon^{-1}\log^{1.5}(\varepsilon^{-1})\log\delta^{-1})$	MRL, Sample
VIII.	KLL [4] (r)	$O(\varepsilon^{-1}\log\log\delta^{-1})$	ACHPWY, GK

Exactly how much memory suffices is an important open problem[2]. The best known deterministic comparison-based algorithm is called GK [2], and it consumes $O(\varepsilon^{-1}\log\varepsilon n)$ space. This does not seem tight because it still grows with n. In recent years, the community has turned to improving randomized approaches. The new KLL result [4] is a randomized algorithm that solves the problem with probability $1-\delta$ using only $O(\varepsilon^{-1}\log\log\delta^{-1})$ memory, which is tight to the randomized lower bound. We will discuss how these algorithms build upon each other.

C. Definition of quantile sketches.

All these algorithms compute a useful generalization to approximate medians called an *ε-all-quantile-sketch*

[1]Randomized algorithms are marked with (r) in the table
[2]For example, it is problem #2 on **http://sublinear.info/2**

(which we shall call an ε-*sketch* for brevity). An ε-sketch is a function $\hat{Q}$ which can be built in one pass over the stream S that can produce values for every $0 \le q \le 1$ such that $\text{rank}(\hat{Q}(q), S) \approx qn$. Setting $q = 1/2$, for example, gives us an estimate of the median $\hat{Q}(1/2)$. We will consider both deterministic sketches and randomized sketches. In the deterministic case, an ε-sketch must be accurate up to ε in the sense of producing values within εn rank of the desired rank for every q:

$$\forall q,\ |\text{rank}(\hat{Q}(q), S) - qn| \le \varepsilon n \qquad (1)$$

In the randomized case, we will require that, except with an error probability less than small δ, every quantile estimate lies within ε accuracy simultaneously:

$$\Pr(\forall q,\ |\text{rank}(\hat{Q}(q), S) - qn| > \varepsilon n) \le \delta \qquad (2)$$

It is important to note the randomized approaches are analyzed with respect to adversarial streams, so the input is allowed to be very nonrandom: no input distribution is assumed. Any shuffling needed for an algorithm to succeed with high probability must be done by the algorithm itself, within its limited memory budget.

For simplicity of analysis, we do assume that the stream has all distinct values. Without ties, the rank of every stream item $s_i \in S$ is uniquely determined as the number of stream items no larger than s_i

$$\text{rank}(s_i, S) \equiv |\{s_j \in S \text{ such that } s_j \le s_i\}| \qquad (3)$$

Duplicates in a stream can be made unique by adding bits. We omit a discussion of this process.

The analysis of some algorithms is easier to understand if we think of an ε-sketch as a solution to rank estimation. A ε-rank-estimator $\hat{R}$ is a function that estimates $\text{rank}(v, S)$ for any v within εn:

$$\forall v,\ |\hat{R}(v) - \text{rank}(v, S)| \le \varepsilon n \qquad (4)$$

For example, some algorithms will represent a sketch as a carefully selected subset $\hat{S} \subset S$ with size $m = |\hat{S}| \ll n$. Then the scaled rank within $\hat{S}$

$$\hat{R}(v, \hat{S}) \equiv \frac{n}{m}\text{rank}(v, \hat{S}) \qquad (5)$$

will be a rank estimator. Rank estimation is equivalent to the ε-all-quantile sketch problem, so when $\hat{R}(v, \hat{S})$ satisfies (4), we also call $\hat{S}$ an ε-sketch.

D. Known lower bounds

An ε-sketch must consume at least $O(\varepsilon^{-1})$ memory: if fewer than $\varepsilon^{-1}/2$ values are stored, there must be some gap exceeding $2\varepsilon n$ consecutive unstored items. Asking for a rank at the midpoint of this gap will fail to find any answer within εn of the requested rank.

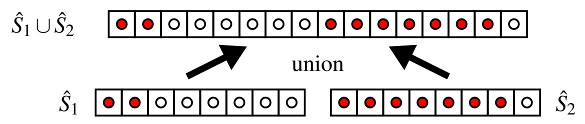


Fig. 1. A simple fact: when the union of two sets is taken, the rank of an element within the union of the sets is the sum of the rank of the element within each of the two sets. This can be seen by red-coloring elements $\le v$, and noting that the rank of v within a set is the count of red elements. This fact means that, even if $\hat{S}_1$ and $\hat{S}_2$ are very dissimilar from each other, as long as they are both ε-sketches that give good rank estimates for underlying streams S_1 and S_2, the union $\hat{S}_1 \cup \hat{S}_2$ will be an ε-sketch that gives a good rank estimate for $S_1 \cup S_2$.

Furthermore, it is known that $O(\varepsilon^{-1})$ memory is not enough: Hung and Ting [3] have shown how to construct an adversarial stream that grows the largest gaps between any algorithm's stored items. Their approach forces a deterministic quantile sketch to store at least $O(\varepsilon^{-1} \log \varepsilon^{-1})$ items. The same adversary can be applied to any randomized algorithm by choosing δ tiny enough to run on all $n!$ permutations of the stream with a high probability of success [4]. The resulting lower bound for randomized algorithms is $O(\varepsilon^{-1} \log \log \delta^{-1})$.

III. A Deterministic Thought Experiment

To warm up our intuition, begin by considering how to improve the trivial algorithm which stores all n items in a single sorted array $\hat{S}$ of length $m = n$. The qth quantile is estimated as $\hat{S}[mq]$. Since we accept ε error, the array should still be a valid ε-sketch if we reduce the size of m by dropping items evenly.

How to identify the right items to drop is tricky when streaming. Imagine dropping $1/2$ of the items by skipping every even-indexed item in the stream. An adversarial stream could place all the smallest items on the skipped indexes so $\hat{S}[1]$ ends up with a value near the median instead a value near the smallest.

A. Compacting a sketch in batches

A more refined method to drop half the items would be to drop all the even-indexed items *after* we have sorted them. Then the maximum gap between the retained items is 1, and the error is minimized. This might not seem helpful since it uses double the storage again, but it turns out to work when applied in small batches.

Theorem 1: Suppose we have two streams S_1 and S_2, each with an ε-sketch $\hat{S}_1 \subset S_1$ and $\hat{S}_2 \subset S_2$. Then $\hat{S}_1 \cup \hat{S}_2$ is a ε-sketch for stream $S_1 \cup S_2$.

Proof: See Figure 1. We show that since $\hat{S}_1$ and $\hat{S}_2$ are good rank estimators for S_1 and S_2, the union of the

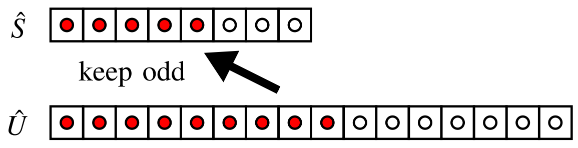


Fig. 2. Compacting a sketch (by sorting and keeping only the odd-numbered items) can be done while only introducing a rank error of 1. Here for some element v, we have colored all elements $\leq v$ red. Sorting the elements of $\hat{U}$ moves all c_u red elements to the left. Keeping the odd-numbered items keeps $c_s = \lceil c_u/2 \rceil$ of the items in $\hat{S}$, which means that $2c_s$ differs from c_u by at most 1.

samples is a good rank estimator for the union of the streams. The true rank of v within S_1 and S_2 may be very different from each other, but they sum to the true rank of v within $S_1 \cup S_2$. Similarly, the ranks within the union of the estimators $\hat{S}_1$ and $\hat{S}_2$ are the sum of the ranks within each estimator. Thus error bounds add, and rank estimates given by the union are within $\varepsilon(|S_1| + |S_2|)$ of $\text{rank}(v, S_1) + \text{rank}(v, S_2) = \text{rank}(v, S_1 \cup S_2)$. ■

The above proof shows that we can always merge subset-based ε-sketches without without making them worse (in terms of ε error).

In particular, we use the fact to make a series of small sketches and save some memory: divide up stream S into a series of buffers of size $|S_i| = k \geq \varepsilon^{-1}$. Then sorting and dropping even-indexed items of each buffer S_i will make an ε-sketch $\hat{S}_i$ of size $\sim k/2$. Taking the union $\hat{S} \equiv \bigcup \hat{S}_i$ results in an overall ε-sketch of size $\sim n/2$.

IV. The MRL Algorithm

In this section we improve the batching idea by applying it recursively. The result is our first interesting algorithm: quantile sketches in $O(\varepsilon^{-1} \log^2 \varepsilon n)$ space.

A. Hierarchical compaction

To apply batching recursively, we use the following idea: instead of only compacting buffers of raw input, we compact buffers that already contain ε-sketches. Again the easiest way to understand the process is to think of the ε-sketches as rank estimators.

Theorem 2: Suppose $\hat{U}$ is an ε-sketch of stream S, where $|\hat{U}| = 2k$ and $|S| = n$ as usual. Then we can create a subset $\hat{S} \subset \hat{U}$, of size $\lceil k \rceil$, which is an $(\varepsilon + 1/2k)$-sketch of S.

Proof: See Figure 2. Sort the elements of $\hat{U}$, and then set $\hat{S}$ to be the subset of elements in odd-indexed sorted order. The picture illustrates that the only errors introduced are due to rounding. To quantify the error, fix any v, and red-color all the elements of $\hat{U}$ which are $\leq v$. Suppose there are c_u red elements in $\hat{U}$. Taking all the odd-indexed elements in sorted order keeps exactly $c_s = \lceil c_u/2 \rceil$ red elements in $\hat{S}$. Thus

$$2c_s = \begin{cases} c_u & \text{if } c_u \text{ is even} \\ c_u + 1 & \text{if } c_u \text{ is odd} \end{cases} \tag{6}$$

$\hat{U}$ is assumed to be an ε-sketch, so $(n/2k)c_u$ is within $n\varepsilon$ of the true rank of v. This means that relation (6) implies $(n/k)c_s$ is within $n\varepsilon + (n/2k)$ of the true rank of v. Dividing by n, we find that the compacted array $\hat{S}$ is an $(\varepsilon + 1/2k)$-sketch of the original stream. ■

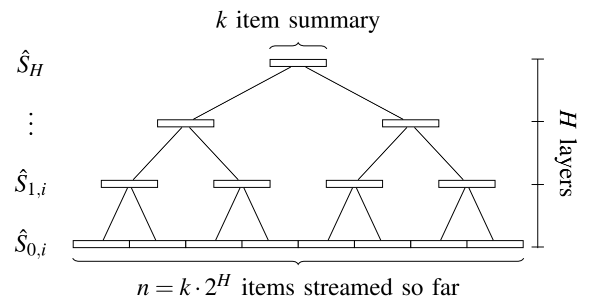


Fig. 3. By cascading equal-weight merges, we create a series of sketches that summarize larger ranges of input. The entire process can be thought of as a binary tree of merges.

This compaction process halves the size of any sketch while introducing only incremental errors. By combining Figure 1 and Figure 2, we can see that a merge and compaction can reduce any two k-sized sketches into a single k-sized sketch with only incrementally more error. Figure 3 shows a generalization of this procedure.

To process $n = k \cdot 2^H$ elements, we only need at most two k-sized buffers at each level $0 \leq h \leq H$. When a new bottom-level buffer of k input elements is read, it is stored in the buffer at level $h = 0$ unless there is already a buffer stored at that level. If it is the second buffer at that level, then the new and old buffer are joined and sorted together. The sorted pair is then compacted back down to a buffer of size k by keeping every odd element. The result is stored in level $h+1$. If there is already a buffer at that level, the process is repeated. We recurse upwards as much as needed. After all $k \cdot 2^H$ elements are processed, the result is a single buffer $\hat{S}_H$ containing a sample of size k.

Theorem 2 tells us that each level of recursion gives rise to an additional $1/2k$ error, so we can withstand at most $2k\varepsilon$ recursions while staying within ε error. Since the depth of recursion is $H = \log_2(n/k)$, we can carefully choose k to ensure that the whole error is bounded $\log_2(n/k)/2k \leq \varepsilon$. Choosing $k = O(\varepsilon^{-1} \log_2(\varepsilon n))$ satisfies the needed constraint.

How much space does this process use? As we have discussed, it uses at most two buffers for each level of Figure 3, since each compaction frees a buffer at that level. The number of buffers needed is thus $O(H) = O(\log_2(n/k)) = O(\log_2(\varepsilon n))$. So the total space we need is $O(kH) = O(\varepsilon^{-1}\log^2(\varepsilon n))$.

In the family tree of quantile sketches, this was the first clever algorithm. Manku, Rajagopalan, and Lindsay describe it as the balanced merging algorithm in [5], and they attribute the original idea to Munro and Patterson [6]. We refer to this algorithm as MRL.

As we have described it, MRL must consume a multiple of 2^H stream size $n = k \cdot 2^H$, but [5] describes several improvements including a generalization for any n. This modification achieves the same space bound.

V. GK SUMMARIES

When choosing elements to drop from the sketch, it is possible to do better than MRL by tracking detailed information about each stored item. This is the strategy used by the GK algorithm designed by Greenwald and Khanna [2], achieving space $O(\varepsilon^{-1}\log \varepsilon n)$. GK is the best deterministic algorithm known, but it is also idiosyncratic. Here we give some intuition about how it works, and how it different from other approaches.

Before we begin, we must warn the reader that the full algorithm and analysis of GK is complex. Fortunately, the basic idea of GK is easy enough to understand, and knowing all details of GK is not critical for understanding the other algorithms in this paper. While KLL does incorporate GK, it only uses it as an opaque subroutine.

A. Intuition: find and exploit easy situations

GK can be motivated by imagining an easy stream: suppose that the first element of the stream s_1 happens to be the stream median. By counting later items higher and lower than s_1, we can verify s_1 is the exact median in $O(1)$ space. This same idea can be extended to all quantiles. We can maintain higher- and lower-counts for each of the first $k = \varepsilon^{-1}$ elements. These counts will enable us to check if these first elements are already a good ε-sketch using only $O(\varepsilon^{-1})$ memory.

GK maintains information for every stored item that is equivalent to these counts. And by applying a form of rank verification, it can (in the best case) use as little as $O(\varepsilon^{-1})$ memory in the fortunate case where the early items in the stream already form an ε-sketch.

However, GK is also different from MRL in that it consumes a varying amount of memory depending on the ordering of the input stream. In the worst case, GK will detect that the early elements are not a good ε-sketch and will retain more elements to create an ε-sketch that includes later items, accumulating up to another $O(\varepsilon^{-1})$ elements each time the stream doubles in size.

B. Details: what GK tracks for each item

How does GK know when to keep or drop an element? GK is based on the observation that by maintaining counts, we can always know the lower (r_{min}) and upper (r_{max}) bound on the possible rank of any stored item. These numbers allow us to ensure that the stored set is an ε-sketch of the stream so far by maintaining this invariant for all adjacent stored items:

$$r_{max}(\hat{s}_i) - r_{min}(\hat{s}_{i-1}) < \lfloor 2\varepsilon n \rfloor \qquad (7)$$

Here the stored $\hat{s}_i$ are indexed in sorted order, and n is the size of the stream so far. With (7), GK will never drop an item that would leave adjacent elements that are uncertain enough (or far enough apart) to have a gap of ranks greater than $2\varepsilon n$.

Because it leads to both cleaner analysis and faster implementations, GK represents r_{min} and r_{max} using attributes Δ_i and g_i for each element, where:

$$\Delta_i \equiv r_{max}(\hat{s}_i) - r_{min}(\hat{s}_i) \qquad (8)$$

$$g_i \equiv r_{min}(\hat{s}_i) - r_{min}(\hat{s}_{i-1}) \qquad (9)$$

With this encoding, adjacent g_i must be updated to keep track of gaps when an element is dropped. Elements initially have $g_i = 1$, but when an element $\hat{s}_{i-1}$ is dropped, the right-adjacent g_i must be updated to add g_{i-1}.

Note that the invariant (7) can now be written:

$$g_i + \Delta_i = r_{max}(\hat{s}_i) - r_{min}(\hat{s}_{i-1}) < \lfloor 2\varepsilon n \rfloor \qquad (10)$$

C. Insight: uncertainty and capacity

While (10) dictates when *not* to drop an element, GK has additional rules to decide when to finally drop an element. The rules are based on two key insights.

The first insight is that every incoming stream element should be assigned $\Delta_i = \lfloor 2\varepsilon n \rfloor - 1$ (with a $\Delta_i = 0$ exception for new min and max elements), and that Δ_i never changes. The large initial Δ_i means that we are maximally uncertain about the true rank of each incoming element since the existing set may have gaps of up to $\lfloor 2\varepsilon n \rfloor - 1$. But afterwards, uncertainty never increases, because all future element ranks relative to $\hat{s}_i$ are known and tracked by sorting them into the set.

The second key insight is that maintaining the invariant (10) is expensive if Δ_i nearly equal to $2\varepsilon n$. Then g_i can not be increased much larger than 1, which means we cannot drop many (or any) items adjacent to the left of $\hat{s}_i$. Motivated by this consideration, the *capacity* of an element is defined as $\lfloor 2\varepsilon n \rfloor - \Delta_i$. High capacity elements are the oldest elements whose ranks are known with high precision and which permit a maximal number of adjacent dropped elements.

D. Rough idea of algorithm and analysis

Capacity gives GK a score to use when deciding which elements to drop. To maximize overall capacity, the algorithm strongly favors dropping elements with low capacity first. The way GK arranges to do this is somewhat intricate: it organizes stored elements in a tree with levels according to capacity, and it chooses items to drop by identifying subtrees of low capacity that can be dropped at once while maintaining the invariant.

Full details can be found Sections 2 and 3 of the original GK paper [2]. The key to the analysis is a way to organize all stream elements into $O(\log \varepsilon n)$ capacity bands where it can be shown that at most $O(\varepsilon^{-1})$ elements are retained in each band. Summing the bands yields a total space bound of $O(\varepsilon^{-1}\log(\varepsilon n))$.

This concludes our rough sketch of GK. The current theoretical lower bound does not grow with n, so it is still not known whether GK is optimal, or if there is a more efficient deterministic approach.

VI. A Randomized Thought Experiment

Is there a simpler way to estimate quantiles? If we allow randomness, the simplest approach is to use a perfectly random sample $|\hat{S}| = m$ to estimate ranks:

$$\hat{R}(v,\hat{S}) \equiv \frac{n}{m}\operatorname{rank}(v,\hat{S}) \approx \operatorname{rank}(v,S) \tag{11}$$

To ease analysis, we draw $\hat{S}$ uniformly with replacement [7] so each sampled item is independent.

We can quantify the accuracy of $\hat{R}(v,\hat{S})$ by appealing to Hoeffding's inequality, which provides an exponential limit on the probability of deviations of a sample mean from the true mean of any set of bounded random variables $V_i \in [V_{min}, V_{max}]$ with $V_{max} - V_{min} \leq \gamma_i$.

(Hoeffding) $\Pr(\sum V_i - \sum \mathrm{E}(V_i) > \alpha) \leq e^{-2\alpha^2/\Sigma\gamma_i^2}$ (12)

In the unit case where all m variables have range $\gamma_i = 1$, the inequality can be written as:

$$\Pr(\sum V_i - \sum \mathrm{E}(V_i) > \varepsilon m) \leq e^{-2m\varepsilon^2} \tag{13}$$

How does a bound on means of sums have anything to do with a median or any other quantile? To see how, we convert rank estimation to a sum of random variables, and prove a bound in the form of (4).

Theorem 3: When $\hat{S}$ is a sample of m independent elements of S, the rank estimate $\hat{R}(v,S)$ satisfies the following bound for any single v:

$$\Pr(|\hat{R}(v,\hat{S}) - \operatorname{rank}(v,S)| \geq \varepsilon n) \leq 2e^{-2m\varepsilon^2} \tag{14}$$

Proof: Define independent indicator variables V_i whose sum is the estimated rank of v:

$$\hat{R}(v,\hat{S}) \equiv \frac{n}{m}\operatorname{rank}(v,\hat{S}) = \frac{n}{m}\sum_{s\in\hat{S}} V_s \tag{15}$$

where we define, for each $s \in \hat{S}$:

$$V_s = \begin{cases} 1 & \text{if } s \leq v \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

Because $\hat{S}$ is uniformly sampled from S, the expected value of V_s is equal to the portion of stream elements $\leq v$, i.e., $\mathrm{E}(V_s) = \operatorname{rank}(v,S)/n$, so

$$\operatorname{rank}(v,S) = n\,\mathrm{E}(V_s) \tag{17}$$

$$= \frac{n}{m}\sum_{s\in\hat{S}} \mathrm{E}(V_s) \tag{18}$$

It is now clear that the error in estimated rank is a difference between a sample mean (15) and true mean (18), so it can be bounded by Hoeffding's (13):

$$\Pr(\hat{R}(v,\hat{S}) - \operatorname{rank}(v,S) \geq \varepsilon n) \tag{19}$$

$$= \Pr(\frac{n}{m}\sum V_s - \frac{n}{m}\sum \mathrm{E}(V_s) \geq \varepsilon n) \tag{20}$$

$$= \Pr(\sum V_s - \sum \mathrm{E}(V_s) \geq \varepsilon m) \tag{21}$$

$$\leq e^{-2m\varepsilon^2} \tag{22}$$

A similar computation limits the probability that $\hat{R}(v,\hat{S})$ is an underestimate below $\operatorname{rank}(v,S) - \varepsilon n$. Therefore the whole probability of error has double the bound. Doubling (22) yields (14), proving the assertion. ■

Theorem 3 describes the probability of error for one particular v. To limit error so that with probability $1-\delta$ all values of v satisfy $|\hat{R}(v,\hat{S}) - \operatorname{rank}(v,S)| < \varepsilon n$ simultaneously, we must choose m that achieves:

$$2e^{-2m(\varepsilon/2)^2} \leq \varepsilon\delta/2 \tag{23}$$

This allows us to guarantee that for $2\varepsilon^{-1}$ values selected to tile the range of ranks in S, each individual falls within $\varepsilon/2$ of its true quantile with probability $1-\varepsilon\delta/2$, so all of them simultaneously fall within those bounds with probability $1-\delta$, by union bound. This is enough to provide accurate answers within ε for every possible query simultaneously. Plugging in these values, the total size needed for simultaneously satisfying all quantiles within ε with probability $1-\delta$ comes to:

$$m = \frac{2\log(\delta\varepsilon/4)}{\varepsilon^2} = O(\varepsilon^{-2}\log(\delta^{-1}\varepsilon^{-1})) \qquad (24)$$

Note that the space used exceeds ε^{-2}, which is wasteful: when querying our sample sketch, we could discard all but ε^{-1} samples, so this summary is wasting a factor of more than ε^{-1} space.

A. Using sampling as a subroutine

Although it is wasteful in ε, the random sampling approach has the remarkable property that the needed sample size does not grow with the stream size n. That means that sampling can be used as a tool to remove the n dependence from other quantile algorithms.

For example, suppose we have a quantile sketch algorithm $\mathscr{A}$ whose space requirement depends on n. We can choose a target sample size $m = O(\varepsilon^{-2}\log\delta^{-1}\varepsilon^{-1})$ large enough to make an $(\varepsilon/2)$-sketch with probability $1-\delta/2$. But we do not need to store m items. Instead, we can configure $\mathscr{A}$ to also create an $(\varepsilon/2)$-sketch with probability $1-\delta/2$ when fed a stream of size m. As long as n is known in advance, with constant space, we can sub-sample the stream down to length m and feed these samples into $\mathscr{A}$. With probability $1-\delta/2$, the output of $\mathscr{A}$ will be an $(\varepsilon/2)$-sketch of the random sample which itself is an $(\varepsilon/2)$ sketch of the original stream with probability $1-\delta/2$. By summing errors and applying union bound on probabilities, the sketch by $\mathscr{A}$ will be an ε-sketch with probability $1-\delta$. The storage consumed will be the space complexity of $\mathscr{A}$ with n replaced by m, removing any dependence on n.

This simplistic approach only works if n is known in advance. In the more common and useful case when n is not known in advance, sampling is still a key tool for eliminating dependence on n, but the rate of sampling must drop as the stream grows, so the algorithm which consumes the sampled stream must be structured to be expect varying input sample rates. We will see such an approach when discussing KLL.

B. Hacking Hoeffding's inequality

Although simple random sampling requires too much storage, the analysis of the random sampling approach reveals a strategy for achieving good efficiency.

1) The random choices are arranged so that the final sample is an unbiased estimator of the true result. The mean expected behavior is exactly correct.

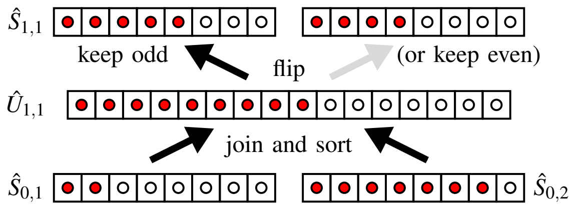


Fig. 4. An individual join-sort-flip-compact operation. The c items $\leq v$ are highlighted in red. Choosing odd items will result in $\lceil c/2 \rceil$ elements in the merge, and choosing even items instead will result in $\lfloor c/2 \rfloor$. The ACHPWY algorithm chooses between these alternatives on each merge using a random coin-flip.

2) Randomness is arranged such that overstepping error bounds requires an unlucky accumulation of many small accidents.

One opportunity for improvement is to apply strategy (2) more effectively. Simple sampling accumulates only m random events where each contributes $\sim \pm 1/m$ randomness, so to achieve small tail bounds, we need to increase the memory m proportionally to the number of desired independent events.

We might improve the efficiency of the algorithm if we could create a procedure that uses a larger number of tiny independent events, for example 2^m events where each contributes $\sim \pm 2^{-m}$ randomness. Then a small increase in memory would create a much tighter concentration towards the mean.

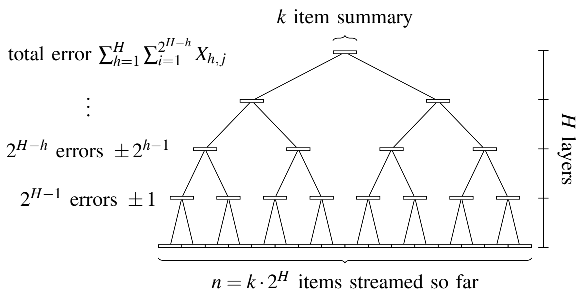
This is the idea that drives the recent clever algorithms by KLL and ACHPWY. These methods use a memory-efficient sampling scheme that utilize $O(2^m)$ tiny independent events instead of $O(m)$ larger events. The story is a bit more complicated because, in these schemes, not all the events can be arranged to be equally tiny. Nevertheless, the idea still works well enough to eliminate a factor of ε^{-1} space.

VII. Mergeable Summaries

The ACHPWY mergeable summaries approach designed by Agarwal, et al [1] is very similar to the MRL balanced merging algorithm we described in Section IV, but the compaction step is modified to use one bit of randomness. In symbols, it computes a hierarchy of merged buffers $\hat{S}_{h,j}$ using coin-flips $F_{h,j}$:

$$\begin{aligned} &\hat{U}_{h,j} \equiv \text{sort}(\hat{S}_{h-1,2j} \cup \hat{S}_{h-1,2j+1}) \\ &F_{h,j} \in \{-1,1\} \text{ randomly} \\ &\hat{S}_{h,j} \equiv \begin{cases} \text{odd-half}(\hat{U}_{h,j}) & \text{if } F_{h,j} \text{ is 1} \\ \text{even-half}(\hat{U}_{h,j}) & \text{if } F_{h,j} \text{ is -1} \end{cases} \end{aligned} \qquad (25)$$

This process is illustrated in Figure 4. When a $2k$-sized



k item summary
total error $\sum_{h=1}^{H}\sum_{i=1}^{2^{H-h}} X_{h,j}$
2^{H-h} errors $\pm 2^{h-1}$
2^{H-1} errors ± 1
H layers
$n = k \cdot 2^H$ items streamed so far

Fig. 5. The entire merge process can be seen as producing a series of coarser-grained estimates of counts of elements $\leq v$ for any given v. After the hth level of merging, the counts are off by $2^{H-h} \times \pm 2^{h-1}$.

sorted join $\hat{U}_{k,j}$ is compacted to size k buffer $\hat{S}_{k,j}$ by discarding every other item, the decision to keep the odd or even-indexed items is done with a random coin-flip $F_{k,j}$. The result is a randomized process within which the rank of each item in the summarized buffer converges to the mean, correct rank.

After all the sorting and compacting, $\hat{S}_H$ contains a very smoothed out random sample whose elements are not independent of each other. Instead, as we will see, the jth element in sorted order is much more likely to be close to rank $j \cdot 2^H$ in the stream than it would be in a sample of independent elements.

A. Error bounds for ACHPWY

To see why the convergence is effective, we use the same technique as in MRL (Section IV): choose a threshold element value v at any rank of interest, and imagine red-coloring all elements which satisfy $\leq v$, so the number of red elements in the stream is the rank of v.

As with MRL, the errors in counts occur only when rounding $c/2$, but unlike with MRL, we now round up and down with equal probability. Since a fair coin-flip is used to choose between the $\lceil c/2 \rceil$ and $\lfloor c/2 \rfloor$ case, the mean number of red elements in the result is exactly $c/2$, i.e., doubling the sample count produces an unbiased estimator of the original count before compaction.

As illustrated in Figure 5, errors can be accounted for one level at a time. When the full algorithm processes a stream of $k \cdot 2^H$ elements at height $h = 0$, the 2^H buffers of size k are all joined, sorted, and compacted pairwise to become 2^{H-1} buffers $\hat{Q}1,i$ at level $h = 1$. Each compaction will over- or under-estimate its number of red elements by a random variable $X_{1,i} \in +1, 0, -1$, so the final total error in the estimate of red elements summing over all the results of first-level merges will be $\sum_{i=1}^{2^{H-1}} X_{1,i}$.

Higher levels of the hierarchy are similar; we just need to remember that at each higher level h there are half the number of merged buffers (2^{H-h}), and each merged buffer has weight 2^h instead of 2; so compaction will introduce estimation errors of $\pm 2^{h-1}$. That is, each compaction will over- or under-estimate its number of red elements by a random variable $X_{h,i} \in \{+2^{h-1}, 0, -2^{h-1}\}$, and the total error in the estimated count of red elements introduced by h-level merges will be $\sum_{i=1}^{2^{H-h}} X_{h,i}$.

Theorem 4: When $\hat{S}_H$ is selected from a stream of size $k \cdot 2^H$ using ACHPWY (25), the probability of error in estimated rank $\hat{R}(v, \hat{S}_H)$ exceeding εn is bounded by

$$\Pr(|\hat{R}(v, \hat{S}_H) - \text{rank}(v, S)| \geq \varepsilon n) \leq 2e^{-2k^2\varepsilon^2} \tag{26}$$

Proof: The errors in estimated counts are cumulative at each level, so the total in error when using the final buffer $\hat{S}_H$ to estimate the number of red elements is a sum of random $X_{h,i} \in [-2^{h-1}, 2^{h-1}]$ at all layers:

$$\hat{R}(v, \hat{S}_H) - \text{rank}(v, S) = \sum_{h=1}^{H} \sum_{i=1}^{2^{H-h}} X_{h,i} \tag{27}$$

Recall, as before, the notation $\hat{R}(v, \hat{S}_H)$ denotes the weighted count

$$\hat{R}(v, \hat{S}_H) \equiv 2^H \cdot |\{s \in \hat{S}_H \text{ such that } s \leq v\}| \tag{28}$$

Each merge provides an unbiased estimate of the count of red items in its input, so $\hat{R}(v, \hat{S}_H)$ is an unbiased estimate of rank(v, S), and we can use Hoeffding's (12) to quantify the probability of error. The main complexity is that the random variables $X_{h,i}$ are weighted so $\gamma_{h,i} = 2^h$. Nevertheless, an evaluation of geometric sums reveals $\sum_{h=1}^{H}\sum_{i=1}^{2^{H-h}} 2^{2h} = 2^{H+1}(2^H - 1) \geq 2^{2H}$, so the exponential terms in (32) cancel:

$$\Pr(|\hat{R}(v, \hat{S}_H) - \text{rank}(v, S)| \geq \varepsilon n) \tag{29}$$

$$= \Pr(|\sum_{h=1}^{H} \sum_{i=1}^{2^{H-h}} X_{h,i} - \text{E}(\sum_{h=1}^{H} \sum_{i=1}^{2^{H-h}} X_{h,i})| \geq \varepsilon n) \tag{30}$$

$$\leq 2\exp(-2(\varepsilon n)^2 / \sum_{h=1}^{H} \sum_{i=1}^{2^{H-h}} \gamma_{h,i}^2) \tag{31}$$

$$= 2\exp(-2\varepsilon^2 k^2 2^{2H} / \sum_{h=1}^{H} \sum_{i=1}^{2^{H-h}} 2^{2h}) \tag{32}$$

$$\leq 2e^{-2k^2\varepsilon^2} \tag{33}$$

thus establishing the bound. ■

Comparing (33) to (14) shows that this new procedure does succeed in reducing error more quickly for each unit of additional memory. We used kH units of memory, and our probability of error is $2e^{-2k^2\varepsilon^2}$ rather than $2e^{-2m\varepsilon^2}$: the exponent grows quadratically.

With a suitable choice of $k_1 = \varepsilon^{-1}\sqrt{(1/2)\log 2\delta_1^{-1}}$, we can achieve error of ε with probability $1-\delta_1$ for a single rank estimation for one v. To achieve this estimate for all quantiles simultaneously requires choosing a δ_1 that is a factor of ε smaller, so for all quantiles we need

$$k = O(\varepsilon^{-1}\sqrt{(1/2)\log 2\varepsilon^{-1}\delta^{-1}}) \qquad (34)$$

Notice that the error behavior does not depend on H, so to process any $n = k \cdot 2^H$ elements we can just select an appropriate buffer size k, and continue the process with arbitrarily large $H = O(\log \varepsilon n)$ while maintaining the same error bounds. The total memory needed is:

$$m = kH = O(\varepsilon^{-1}\log^{1/2}(\varepsilon^{-1}\delta^{-1})\log \varepsilon n) \qquad (35)$$

B. Further improvements

Agarwal, et al describe how to generalize this merging process to stream lengths n which are not a multiple of 2^H, and also how to eliminate the $\log \varepsilon n$ factor in the size by attaching their merging process to a random sampler. Their sampling procedure reduces the sampling rate as n grows, but it also increases the weights of the items outputted from the sampler so that sampled buffers enter directly into an increasing level h of the merging hierarchy instead of level 0 as n grows. This allows them to keep a constant-height active merging tree even as the top level H grows. We will revisit this technique in Section VIII-B in our discussion of KLL, where we discuss a growing sampler in detail.

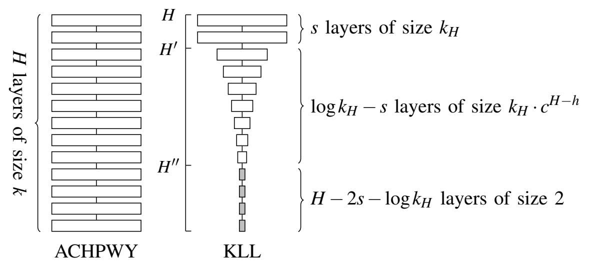
Attaching the sampler to ACHPWY increases the error, and in the end the algorithm achieves a memory bound of $O(\varepsilon^{-1}\log^{3/2}(\varepsilon^{-1})\log \delta^{-1})$.

VIII. KLL Sketches

The KLL algorithm introduces four main insights that operate in conjuction with the ACHPWY mergeable summaries and GK sketch previously discussed. Each insight lowers the space complexity of the algorithm while preserving ε accuracy. As will be shown, KLL does this by optimizing the sizes of the stack of buffers.

A. From ACHPWY to KLL layers

The first idea is to use different buffer sizes at each layer as illustrated in Figure 6. The total memory consumed by ACHPWY is $O(kH)$ because every one of the H layers requires $O(k)$ memory. KLL reduces this to $O(k_H)$ (where k_H is the maximum buffer size) by taking exponentially-decreasing buffer sizes (see Figure 6). Since the sum of a geometric series is proportional to the size of the largest item, this idea can reduce the total size to $O(k_H)$ instead of $O(kH)$. However, we do need to ensure that errors remain small enough; and we also need to consider what to do with the tiny fractional size buffers given by the geometric series at the bottom.



H layers of size k
ACHPWY
KLL
H
H'
H''
s layers of size k_H
$\log k_H - s$ layers of size $k_H \cdot c^{H-h}$
$H - 2s - \log k_H$ layers of size 2

Fig. 6. Both the mergeable summaries algorithm designed by ACHPWY and KLL execute a tree of merges over a stream using a stack of buffers with one buffer per layer of the hierarchy. While ACHPWY uses uniform k-size buffers at every level, KLL uses varying buffer sizes, with only a few top $s = O(\log\log\delta)$ layers of size k_H; then $O(\log k_H)$ layers with exponentially decreasing buffer sizes. Remaining layers have buffer size 2 and can be simulated using a single constant-size sampler. While the data structure on the left consumes $O(kH)$ memory, the data structure on the right consumes only $O(k_H)$ memory.

More specifically, suppose the total height of hierarchy is H. The buffer size k_h at height h is set to

$$k_h = k_H \cdot c^{H-h} \qquad (36)$$

where c is a hierarchy shape perimeter that lies in $(.5,1)$. Note that KLL choose $c = 2/3$, but that any $c \in (.5,1)$ results in the same asymptotic error performance. To deal with the fact that buffer sizes must be integers, for low levels of the hierarchy, we must round up and define the capacity of the smallest buffers to be at least 2. We will seperately analyze this bottom subset of size 2 buffers later.

Similar to the analysis of ACHPWY, we need to bound the height of hierarchy first. Note that each level up in the buffer hierarchy downsamples the input stream by a factor of 2. Thus at level h, the sampling rate, or weight, is given by $w_h = 2^{h-1}$. This can be interpreted as how many items in the original input stream are represented by a single item in level h. By definition, the top layer buffer is not empty, and every element in it must be the result of compacting elements from the level below. Therefore, we can bound the number of elements seen so far by $n \geq k_{H-1}w_{H-1} = k_{H-1}2^{H-2}$. Solving for H:

$$H \leq \log(n/k_{H-1}) + 2 \leq \log(n/ck_H) + 2 \qquad (37)$$

Next, we need to analyze the total error produced by the sketch, which can be determined by considering the error generated at each level. As in ACHPWY, each level only introduces error during compaction, which is related to its sampling rate (i.e. weight). For some level

h with weight $w_h = 2^{h-1}$, the number of compactions m_h is constrained by

$$\begin{aligned} & k_h w_h m_h \le n \\ \Rightarrow\quad & m_h \le n/k_h w_h \le (2/c)^{H-h-1} \end{aligned} \tag{38}$$

Abbreviating the pattern of analysis we have seen in previous sections, denote the function $\hat{R}(v,h)$ as the weighted rank of any element v in the union of all level-h buffers in the compaction tree. We define the error due to level h to be $\mathrm{err}(v,h) = \hat{R}(v,h) - \hat{R}(v,h-1)$. The total error introduced after H levels of compaction is

$$\sum_{h=1}^{H} \mathrm{err}(v,h) = \sum_{h=1}^{H}\sum_{i=1}^{m_h} w_h X_{i,h} \tag{39}$$

where $X_{i,h} \in \{-1,1\}$ is the random coin flip determining if the odd or the even elements are chosen for compaction. For reasons we shall see in Section VIII-C, we parameterize error analysis by counting error for the bottom H' layers (for any $H' \le H$). Then we can apply Hoeffding (12) as we did for the analysis of ACHPWY:

$$Pr(\sum_{h=1}^{H'} \mathrm{err}(v,h) \ge \varepsilon n) \tag{40}$$

$$= Pr(\sum_{h=1}^{H'}\sum_{i=1}^{m_h} w_h X_{i,h} \ge \varepsilon n) \tag{41}$$

$$\le 2\exp(-(\varepsilon^2 n^2)/(2\sum_{h=1}^{H'}\sum_{i=1}^{m_h} w_h^2)) \tag{42}$$

The geometric series sums evaluate as follows:

$$\sum_{h=1}^{H'}\sum_{i=1}^{m_h} w_h^2 = \sum_{h=1}^{H'} m_h w_h^2 \tag{43}$$

$$\le \sum_{h=1}^{H'} (c/2)^{H'-h-1} 2^{2h-1} \tag{44}$$

$$\le ((2/c)^{H'-1}(2c)^{H'})/4(2c-1) \tag{45}$$

$$\le (c2^{2H'})/8(2c-1) \tag{46}$$

Plugging (46) into (42) and using the identities $2^{2H'} = 2^{2H}/2^{2(H-H')}$ and $H = \log(n/ck_H) + 2$, we get

$$Pr(\sum_{h=1}^{H'} \mathrm{err}(v,h) \ge \varepsilon n) \le 2e^{-c^2(2c-1)\varepsilon^2 k_H^2 2^{2(H-H')}} \tag{47}$$

Setting $H' = H$ and requiring failure probability of at most δ (i.e. $2e^{-c^2(2c-1)\varepsilon^2 k_H^2 2^{2(H-H')}} \le \delta$), KLL sets $k_H = c^2(2c-1)\varepsilon^{-1}\sqrt{\log 2\delta^{-1}}$. The space complexity of this version of KLL is $O(\sum_{h=1}^{H} k_h)$. Recalling that all buffers must be of at least size 2, we know that

$$\sum_{h=1}^{H} k_h \le \sum_{h=1}^{H} (k_H c^{H-h} + 2) \tag{48}$$

$$\le k_H/(1-c) + 2H \tag{49}$$

$$= O(k_H + \log n/k_H) \tag{50}$$

Union bounding over the failure probabilities of all quantiles, this version of KLL uses $O(\varepsilon^{-1}\sqrt{\log \varepsilon^{-1}\delta^{-1}} + \log(n\varepsilon/\log(\varepsilon^{-1}\delta^{-1})))$. Note that this space bound is not ideal, as it grows with the size of the stream. The next improvement removes this dependence.

B. Replacing the bottom layers with a sampler

We have previously seen that a dependence on n can be removed by introducing a sampler. KLL does this in an elegant way that can be visualized in the bottom layers of Figure 6: specifically, they notice that any sequence of H'' buffers (where $H'' \le H' \le H$) all of size 2 perform a uniform sampling. To see this, note that for every $2^{H''}$ elements, the element that ends up in the top buffer is the winner of a tournament of $2^{H''} - 1$ random binary choices. In other words, this simply is a uniform random choice that can be done efficiently in $O(1)$ space instead of $O(H'')$ space. It follows that the total space used by buffers of size greater than 2 is $\sum_{h=H''+1}^{H} k_H c^{H-h} \le k_H/(1-c) = O(k_H)$.

To select the right k_H, we once again use Hoeffding to bound the probability of error, this time bounding the total error by the sum of sampler and buffer errors. Let $\hat{R}(v,h)$ be the height-h estimate of the rank of element v after applying the sampler at height h. Denote the error at height h as $\mathrm{err}(v,h) = \hat{R}(v,h) - \hat{R}(v,h-1)$; the total error up toe height H'' is:

$$\sum_{h=1}^{H''} \mathrm{err}(v,h) = \sum_{h=1}^{H''}\sum_{i=1}^{m_h} 2^h X_{i,h} \tag{51}$$

In this case, m_h is the number of times a sample operation can occur at height h. Because a sampler at height h takes elements of weight 2^{h-1} and the overall weight of all items must be n, we know that

$$m_h \le n/2^{h-1} \tag{52}$$

Substituting in to solve for the total error due to the sampler, we get

$$\sum_{h=1}^{H''}\sum_{i=1}^{m_h} 2^{2h} \le \sum_{h=1}^{H''} n2^{h+1} \le 4n2^{H''} \tag{53}$$

We can now use this expansion to apply the Hoeffding bound again. Using identities $2^{2H''} = 2^{2H}/2^{2(H-H'')}$ and

$H = \log(n/ck_H) + 2$, we get

$$Pr(\sum_{h=1}^{H''} \text{err}(v,h,i) \geq \varepsilon n) \leq 2e^{-c\varepsilon^2 k_H 2^{H-H''}/32} \quad (54)$$

The total error due to a sampler and a set of buffers is the sum of the two Hoeffding bounds. In other words, it must hold that

$$\begin{aligned} Pr(\sum_{h=1}^{H'} \text{err}(v,h) \geq 2\varepsilon n) \\ \leq 2e^{-c^2(2c-1)\varepsilon^2 k_H^2 2^{2(H-H')}} + 2e^{-c\varepsilon^2 k_H 2^{H-H''}/32} \end{aligned} \quad (55)$$

Observe that the bottom $H'' = H - \log k_H / \log(1/c)$ buffers must all be of size 2. Setting $H' = H$ and union bounding over all quantiles, we find that the optimal selection of buffer size is $k_H = O(\varepsilon^{-1}\sqrt{\log(\varepsilon^{-1}\delta^{-1})})$. Since the total capacity of buffers of size greater than 2 is bounded by $\sum_{h=H''+1}^{H} k_H c^{H-h} \leq k_H/(1-c) = O(k_H)$, the total space requirement of this algorithm is also $O(k_H) = O(\varepsilon^{-1}\sqrt{\log(\varepsilon^{-1}\delta^{-1})})$.

To understand why this space does not depend on n even though H can grow unbounded, consider the progress of the algorithm as n grows. Obviously, there are a finite number of buffers, meaning that at some point they will all fill up. Eventually, the top-most buffer at level H will overflow, forcing it to compact. This results in the creation of a new buffer at level $H+1$ with weight 2^H.

With a new top layer we must redefine $H \leftarrow H+1$ and continue the algorithm. Applying the sizing formula $k_h = k_H \cdot c^{H-h}$, we can see that the creation of a new top level buffer has caused every buffer in the hierarchy to have less capacity because each buffer becomes c times smaller. Besides causing a cascade of compaction operations to prevent overflowing, the bottom buffer above the sampler will also shrink below capacity of 2. This simply results in that bottom buffer being absorbed by the sampler, which now samples at half the rate. Thus, the longer the stream, larger the height H of the buffer stack, yet, due to clever use of the sampler, the overall algorithm uses a non-increasing amount of memory.

While this is already an elegant result by itself (smoothly integrating a sampler, and improving on the previous state of the art for a randomized algorithm), KLL employs two further improvements to achieve the lower bound on randomized streaming algorithms.

C. Optimizing the top layers

In the previous section, we used Hoeffding to bound the probability that the error introduced by buffers and a sampler is more than εn. A key observation made by KLL is that the majority of the contribution to error is due to the top set of buffers. In other words, the lower portion buffers (i.e. buffers of smaller size and weight) are more accurate than they need to be (and thus use more space than necessary) to ensure that the top buffers do not break ε accuracy with constant probability. In addition, for the top layers, Hoffeding bound is an inadequate measure — since we are only looking at a small portion of layers, their total error can be far from the mean. To fix this issue, KLL separately considers the error due to the top s layers and imposes a deterministic bound. Details are given as follows.

Instead of making the top s layers exponentially decrease in size (as was the case before), the top s buffers are assigned size k_H (as the MRL algorithm does). Due to this, a deterministic bound similar to MRL can be applied to bound the maximum amount of error. As before, each layer of weight w_h has m_h compactions, resulting in maximum error of $\sum_s^H m_h w_h = \sum_s^H n/k_h = sn/k_H$. Obviously, we would like this error to be no more than εn, which gives us the relation $s \leq k_H \varepsilon$.

Now consider the other layers (with $H' = H - s$ and $H'' = H - 2s - \log k_H$). Using Hoeffding, we see that $k_H 2^s \geq \sqrt{\log(2\varepsilon^{-1}\delta^{-1})}/(\varepsilon\sqrt{c^2(2c-1)})$. We would like to choose values of s and k_H such that both the top s and bottom H' layers are within their respective error bounds. Assigning $s = O(\log\log(\delta^{-1}))$ and $k_H = O(\varepsilon^{-1}\log\log(\delta^{-1}))$ satisfies both conditions.

The space requirement of the top s layers is $O(sk_H) = O(\varepsilon^{-1}\log^2\log(\delta^{-1}))$. The space requirement of the bottom H' layers is still $O(k_H) = O(\varepsilon^{-1}\log\log(\delta^{-1}))$. The space complexity is therefore dominated by the topmost layers, meaning that this version of KLL (which includes both a sampler at the bottom and a fixed size buffer region at the top) uses $O(\varepsilon^{-1}\log^2\log(\delta^{-1}))$ space. Not surprisingly, this result further improves upon the state of the art.

The result can be made tighter still.

D. Adding the GK sketch

The overall space complexity of KLL as described above is $O(sk_H)$. Knowing this, is it possible to further reduce space usage with smart choice of k_H? KLL answer this question in the affirmative, claiming that it is possible to set $k_H = \varepsilon^{-1}$. Intuitively, this makes sense because a deterministic bound for GK is better than that of MRL. Thus substituting GK into the top s layers should provide reduced space usage. A more careful analysis follows.

Let us ignore the top s layers for now. From before, we know that the bottom H' layers are bounded by $k_H 2^s \geq \sqrt{\log(2\varepsilon^{-1}\delta^{-1})}/(\varepsilon\sqrt{c^2(2c-1)})$, which can be rewritten as $s \geq O(\log(\varepsilon^{-1}k_H^{-1}\log(\delta^{-1})))$. Setting $k_H = \varepsilon^{-1}$ and $s = O(\log\log(\delta^{-1}))$ satisfies the condition. Note that this choice of parameters results in undesirable error bounds for the top s layers.

To fix this problem, we use a set of 2 GK sketches positioned near the top of the hierarchy of buffers. Specifically, they will be placed at heights h_1 and h_2, which are both the largest height values that are also multiples of s. It follows that h_1 and h_2 are placed at heights of at least $H-2S$ and $H-s$, respectively. Each GK sketch receives as input the output of the buffer one layer below it. The main reason for having 2 GK sketches is that each time s additional buffers are formed, the lower of the two is discarded to make room for the next GK sketch. In this way, there is always at least one non-empty GK sketch at all times.

Recall that GK sketches grow proportionally to the size of the stream, an undesirable characteristic for a randomized algorithm. That being said, this is not of concern in this scenario because we can bound the total number of elements inputted into a GK sketch before it will be discarded. Recall that we previously used the identity $n \geq K_{H-1}2^{H-2}$ to bound the total number of items seen so far. Knowing that the height of a GK sketch is at least $h_1 = H-2s$, we can bound the total number of elements fed into the sketch by $n_1 = k_H 2^{2s}$. It follows that the memory required by the GK sketch is at most $O(\varepsilon^{-1}\log(\varepsilon n_1)) = \varepsilon^{-1}(\log(\varepsilon k_H 2^{2s}))$. Using optimal values of k_H and s determined previously, total space usage of each GK sketch is no more than $O(\varepsilon^{-1}\log\log(\delta^{-1}))$. The space complexity of the rest of the sketch is considerably smaller ($O(k_H) = \varepsilon^{-1}$), so the overall space requirement is simply the cost of maintaining the GK sketch.

One disadvantage of incorporating the GK sketch is that, unlike ACHPWY and KLL-using-MRL, it is not known to be mergeable. In detail, a GK sketch for two streams containing the same exact values but streaming in a different order may produce two different (but correct) summaries. Because of this, there is no known way to combine two sketches GK_1 and GK_2 other than to replay all the values stored in GK_2 into GK_1. The resulting GK sketch will have increased space usage and may exceed the established size bounds for GK. Therefore in practice, for very large streams which are to be counted in a distributed system using merging, it may be prudent to run a fully mergeable version of KLL based on MRL, sacrificing single-machine memory footprint for improved merging efficiency.

IX. Conclusion

We have given an overview of the state-of-the-art approaches for streaming quantile sketches. Starting with a simple buffered thought experiment, we developed the deterministic balanced merging MRL approach. We reviewed the principles underlying the GK sketch. We analyzed random sampling using Hoeffding's inequality. We then used the same technique to analyze the ACHPWY randomized improvement upon the MRL algorithm. Then we showed how KLL improves upon ACHPWY, smoothly combining it with sampling to create a more economical algorithm. Finally we showed how the space bound on KLL can be tightened by analyzing the top layers as MRL, and tightened further by adding the GK algorithm in the top layers. With this final approach, KLL solves the streaming quantile sketch problem in the smallest possible memory, achieving the randomized lower bound of $O(\varepsilon^{-1}\log\log\delta)$.

References

[1] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. "Mergeable summaries". In: *ACM Transactions on Database Systems (TODS)* 38.4 (2013), p. 26.

[2] M. Greenwald and S. Khanna. "Space-efficient online computation of quantile summaries". In: *ACM SIGMOD Record*. Vol. 30. 2. ACM. 2001, pp. 58–66.

[3] R. Y. Hung and H. F. Ting. "An $\Omega(\frac{1}{\varepsilon}\log(\frac{1}{\varepsilon}))$ space lower bound for finding epsilon-approximate quantiles in a data stream". In: *International Workshop on Frontiers in Algorithmics*. Springer. 2010, pp. 89–100.

[4] 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, in press.

[5] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. "Random sampling techniques for space efficient online computation of order statistics of large datasets". In: *ACM SIGMOD Record*. Vol. 28. 2. ACM. 1999, pp. 251–262.

[6] J. I. Munro and M. S. Paterson. "Selection and sorting with limited storage". In: *Theoretical computer science* 12.3 (1980), pp. 315–323.

[7] B.-H. Park, G. Ostrouchov, N. F. Samatova, and A. Geist. "Reservoir-Based Random Sampling with Replacement from Data Stream." In: *SDM*. SIAM. 2004, pp. 492–496.