Quantile sketch algorithms

— MATH PROJECT —

Author:

Levente Birszki

Applied Mathematics MSc

Supervisors:

Dr. Gábor Rétvári — Senior Research Fellow

Dr. Balázs Vass — Assistant Lecturer

Eötvös Loránd University
Faculty of Science
Budapest, 2024.

1 Introduction

Within the scope of the project, I dealt with the quantile sketch algorithms in database applications. It is a well-researched area of mathematics that has many practical applications such as big data [1], distributed systems [2] and the area that led me here, network traffic monitoring. Within the latter, quantile sketches are used, for example, catching heavy flows [3], attack detection [4] and even in traffic control [5].

Our main goal was to create a self-improving quantile sketch that becomes faster as it processes more data. The purpose of this report is to present the problem and the results achieved so far, along with the asymptotic bounds. I also present some improvements to a fundamental sketching algorithm, whose efficiency I demonstrate through practical measurements.

2 The quantile problem

A *sketch* $S(X)$ of some data set X with respect to some function f is a compression of X that allows us to compute, or approximately compute $f(X)$ given access only to $S(X)$. A *streaming* algorithm is processing data streams in which the input is presented as a sequence of items and can be examined only in one pass. Streaming algorithms often produce approximate answers based on a sketch of the data stream.

Given a stream of items $y_1, y_2, \dots, y_n$ in some arbitrary order, and let $x_1 \leq x_2, \leq \dots \leq x_n$ the sorted sequence. If its necessary, we can assume, that all the elements are distinct, since instead of y_i we can take (y_i, i) with lexicographical ordering.

Definition 2.1. Given an x element from the input stream. $r(x)$, the rank of x is the number of elements smaller or equal than x in the sorted input.

Definition 2.2. The q-quantile for $q \in [0,1]$ is defined to be the element in position $\lceil qn \rceil$ in the sorted sequence of the input. In other words, the element whose rank is $\lceil qn \rceil$. Denote this element with x_q.

There are different versions of this problem. Sometimes an element x is given, and we need to compute $r(x)$, and sometimes the opposite; given a rank r (or a quantile q) and the task is to return the item from the stream, with rank r (or $\lceil qn \rceil$). But usually if we can answer one question, we can also answer the other.

2.1 Theoretical results

Munro and Paterson [6] showed that $\Omega(n^{1/p})$ space is required to determine the quantile q with p passes. Furthermore, Blum, Floyd, Pratt, Rivest and Tarjan [7] showed, that we need at least $1.5n$ comparisons to compute an exact median of a data set of size n. This paper also shows, that $5.43n$ comparisons is always sufficient for any quantile.

Later Dor and Zwick showed, that the lower bound for the median is $(2+2^{-40})n$, [8], and the upper bound for an arbitrary quantile $2.9423n$ [9].

Typically we only have opportunity to a one-pass algorithm, and with limited space, therefore our main goal is to *approximate* the quantiles.

Definition 2.3. An element $\tilde{x}_q$ is an ε-approximate q quantile if $\lceil (q-\varepsilon)n \rceil \leq r(\tilde{x}_q) \leq \lceil (q+\varepsilon)n \rceil$. In other words $|r(x_q) - r(\tilde{x}_q)| \leq \varepsilon n$. This also known as rank error.

Remark. *There are other possible ways to define the error of an approximation, e.g. relative error, which is defined in the paper in which DDSketch was introduced [10]: $\tilde{x}_q$ is an α-accurate q-quantile if $|\tilde{x}_q - x_q| \leq \alpha x_q$, for a given x_q. Since most algorithms use the rank-error, I also use that in the following.*

In 1974 Yao showed, that computing an approximate median requires $\Omega(n)$ comparisons for any deterministic algorithm. In 2016 Hung and Ting [11] proved, that any comparison-based algorithm for finding ε-approximate quantiles needs $\Omega(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ space.

3 Major milestones

Definition 3.1. In the *single quantile approximation problem*, given an $x_1,\ldots,x_n$ input stream in arbitrary order, q, ε and δ. Construct a streaming algorithm, which computes an ε-approximate q-quantile with probability at least $1-\delta$.

Definition 3.2. In the *all quantiles approximation problem*, given an $x_1,\ldots,x_n$ input stream in arbitrary order, ε and δ. Construct a streaming algorithm, which computes an ε-approximate q-quantile with probability at least $1-\delta$ for all q simultaneously.

Definition 3.3. A sketching algorithm is (fully) mergeable, if given two sketches S_1 and S_2 created from inputs X_1 and X_2, a sketch S of $X := X_1 \sqcup X_2$ can be created with no degradation in quality of error or failure probability, and satisfying the same efficiency constraints as S_1, S_2.

Publication	Algorithm	Space Complexity	Properties
1988	MRL [12]	$O(\frac{1}{\varepsilon}\log^2 \varepsilon n)$	non-mergeable, all quantiles deterministic, comparison-based
1988	MRL [12]	$O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon}+\frac{1}{\varepsilon}\log^2\log\frac{1}{\delta})$	non-mergeable, all quantiles randomized, comparison-based
2001	GK [13]	$O\left(\frac{1}{\varepsilon}\log(\varepsilon n)\right)$	non-mergeable, all quantiles deterministic, comparison-based
2004	q-digest [14]	$O(\frac{1}{\varepsilon}\log u)$	mergeable, all quantiles deterministic, fixed universe (of size u)
2016	KLL [15]	$O(\frac{1}{\varepsilon}\log^2\log\frac{1}{\delta})$	mergeable, singe quantile randomized, comparison-based
2016	KLL [15]	$O(\frac{1}{\varepsilon}\log^2\log\frac{1}{\delta\varepsilon})$	mergeable, all quantiles randomized, comparison-based
2016	KLL [15]	$O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta})$	non-mergeable, singe quantile randomized, comparison-based
2016	KLL [15]	$O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta\varepsilon}))$	non-mergeable, all quantiles randomized, comparison-based
2017	FO [16]	$O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$	non-mergeable, all quantiles randomized, comparison-based
2019	SweepKLL [17]	$O(\frac{1}{\varepsilon}\log\log\frac{1}{\delta\varepsilon}))$	non-mergeable, all quantiles randomized, comparison-based runtime is $O(\log\frac{1}{\varepsilon})$ instead of $O(\frac{1}{\varepsilon})$

Its worth to mention two other sketches; QPipe [18] which is an accelerated version of SweepKLL, and can be fully implemented in the data plane of a programmable switch, and Moment Sketch [19], which has no rank error guarantees, but its widely used in practice.

4 MP-sketch

In 1998 Manku, Rajagopalan and Lindsay gave a solution to all quantile approximation problem [12], based on the work of Munro and Peterson [20]. They gave a uniform framework for three sketching algorithms, including the original MP-sketch. In the followings I'll sum up the framework, based on their paper.

Remark. *Originally MRL was used for databases. If we want to use it for data streams, we need some clever sampling, e.g. reservoir sampling [21].*

In this framework, we have b buffers, each can store k elements. for each buffer X, we associate a positive integer $w(X)$, whits denotes its weight. Intuitively, the weight of a buffer is the number of elements represented by each element in the buffer. There are three operations on a buffer, *New*, *Collapse* and *Quantile*.

4.1 $New(X)$ operation

It takes an empty buffer X as an input. The operation simply populates the input buffer with the next k elements from the input stream, and set $w(X) = 1$. If the buffer cannot be filled completely, because there are less than k remaining elements in the input stream, an equal number of $-\infty$ and ∞ elements are added to make up the deficit.

4.2 $Collapse(X_1, X_2, \ldots, X_c)$ operation

It takes $c \geq 2$ full input buffers, $X_1, X_2, \ldots, X_c$ and outputs a buffer Y, which is physically use the same space as X_1. The weight of the output buffer is the sum of the weights of the input buffers, so $w(Y) = \sum_{i=1}^{c} w(X_i)$.

Consider making $w(X_i)$ copies of each element in X_i and sorting all the input buffers together, taking into account the multiple copies. The elements of Y are k equally spaced elements in this (sorted) sequence.

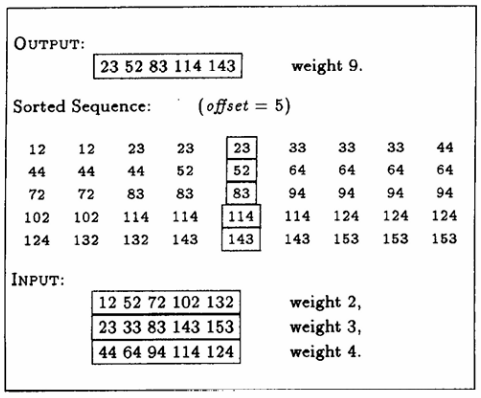


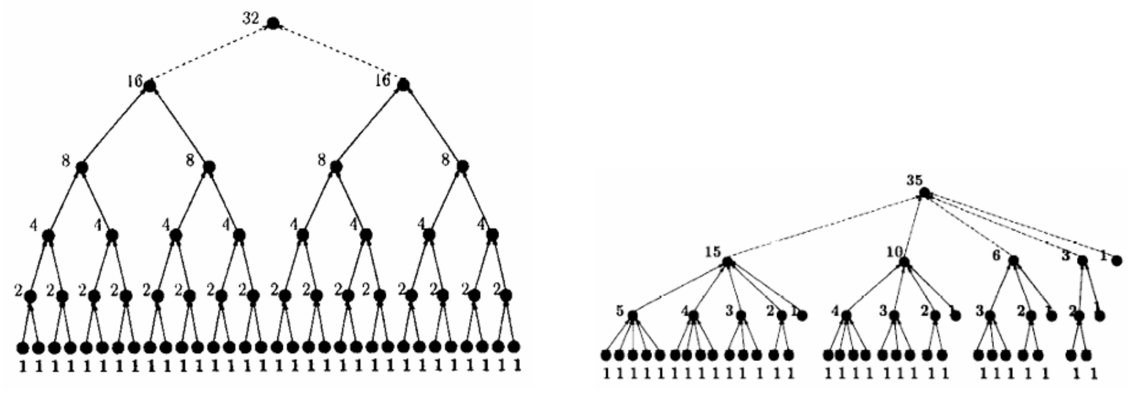
Figure 1: Collapse illustrated.

4.3 $Quantile(q)$ operation

This operation is invoked only after the end of the input stream, when all the elements are processed by the data structure, and there is only one full buffer X, as the result of a *Collapse* operation. It returns the $q \cdot k$ element of buffer X.

4.4 Algorithms

An algorithm for computing approximate quantiles consists of a series of invocations of *New* and *Collapse*, and then we can use *Quantile* as many times, as needed. The key difference between algorithms from this family is the collapse policy. *New* populates empty buffers, and *Collapse* reclaims some of them by collapsing a chosen set of full buffers. In figure 2 we can see two different collapsing policies.



(a) MP-sketch for $b = 6$ buffers.   (b) MRL-sketch for $b = 5$ buffers.

Figure 2: Different collapsing policies.

5 GK-sketch

This data structure was created by Greenwald and Khanna in 2001 [13]. It is widely used due to its simplicity and efficiency, and it serves as the basis for many applications in the field of quantile sketches. For example, the asymptotically optimal version of the KLL-sketch and the special case where the desired quantiles are predetermined [22].

We build a summary data structure $S := S(s)$ consisting of s elements, where we store tuples $t_i = (v_i, g_i, \Delta_i)$. These are composed of the following:

- v_i: one of the elements seen so far.
- $g_i = r_{\min}(v_i) - r_{\min}(v_{i-1})$.
- $\Delta_i = r_{\max}(v_i) - r_{\min}(v_i)$.

Using these, the following values can be calculated:

- $r_{\min}(v_i) = \sum\limits_{j \leq i} g_j$, a lower bound for $r(v_i)$.
- $r_{\max}(v_i) = \sum\limits_{j \leq i} g_j + \Delta_i$, an upper bound for $r(v_i)$.
- Therefore, $r_{\max}(v_i) - r_{\min}(v_{i-1}) - 1 = g_i + \Delta_i - 1$ is an upper bound for the number of observations that may have fallen between v_{i-1} and v_i.
- $r_{\min}(v_0) = r_{\max}(v_0) = 1$ and $r_{\min}(v_{s-1}) = r_{\max}(v_{s-1}) = n$, thus $\sum\limits_{i=0}^{s-1} g_i = n$.

Proposition 5.1. *Given a quantile summary S, a q-quantile can always be identified within an error of* $\max\limits_i (g_i + \Delta_i)/2$.

Corollary 5.1. *If at any time n, the summary $S(n)$ satisfies the property that* $\max_i(g_i + \Delta_i) \leq 2\varepsilon n$*, then we can answer any q-quantile query with ε rank error.*

This data structure has four essential operations.

5.1 *Quantile*(q)

$r := \lceil qn \rceil$. Find i such that both $r - r_{\min}(v_i) \leq \varepsilon n$ and $r_{\max}(v_i) - r \leq \varepsilon n$, and return v_i. This is possible based on Proposition 5.1 and Corollary 5.1. The tuples are stored in an array, and this index can be found via binary search in $O(\log s)$ time.

5.2 *Insert*(v)

Find the smallest i such that $v_{i-1} \leq v \leq v_i$, and insert the tuple $(v, 1, \lfloor 2\varepsilon n \rfloor)$ between t_{i-1} and t_i. Increment s. As a special case, if v is the new minimum or maximum observation seen, insert $(v, 1, 0)$. This operation maintains the correct relationships between g_i, Δ_i, $r_{\min}(v_i)$, and $r_{\max}(v_i)$ by modifying the g_i value of the element after the newly inserted one. Runtime is $O(\log s)$. If $g_i + \Delta_i = \lfloor 2\varepsilon n \rfloor$, it invokes a *Compress*() operation.

5.3 *Delete*(v)

Replace (v_i, g_i, Δ_i) and $(v_{i+1}, g_{i+1}, \Delta_{i+1})$ with the new tuple $(v_{i+1}, g_i + g_{i+1}, \Delta_{i+1})$, and decrement s. This operation maintains the correct relationships between g_i, Δ_i, $r_{\min}(v_i)$, and $r_{\max}(v_i)$ by modifying the g_i value of the element after the recently deleted one. Runtime is $O(\log s)$.

5.4 Compress()

The high-level concept is to traverse the tuples of $S(n)$ from right to left. When a mergeable pair (t_i, t_{i+1}) is found, merge t_i into t_{i+1}, as well as other tuples that are descendants of t_i in a special tree representation.

6 KLL-sketch

This sketch was developed by Khanna, Lang, and Liberty in 2016 [15]. It is similar to the *MRL*-sketch but also incorporates the *GK*-sketch. Instead of buffers, it uses compactors, and in the tree, each level has exactly one compactor. When a compactor fills up, it retains only every second element from its sorted array, sends the retained elements to the next level, and then clears itself.

6.1 Basic Idea

A compactor can store k items, all with the same weight w. It can also compact its k elements into $k/2$ elements of weight $2w$.

- Total items in the sequence: n.
- Number of compactors storing these items: n/k.
- Let H denote the maximum number of compactors chained together. Each compactor halves the number of elements: $H \leq \lceil \log \frac{n}{k} \rceil$ (height of the tree).
- Let h denote the height of a compactor. The bottom level has a height $h = 1$, so $w_h = 2^{h-1}$.
- Let m_h denote the number of compact operations it performs at height h: $m_h = \frac{n}{kw_h}$.

- Summing up the errors:

$$\sum_{h=1}^{H} m_h \cdot w_h = \sum_{h=1}^{H} \frac{n}{k} = \frac{Hn}{k} = \frac{n \log \frac{n}{k}}{k}$$

- Since there are H compactors, the space usage is $kH \leq k \log \frac{n}{k}$.

Setting $k := O(\frac{1}{\varepsilon} \log \varepsilon n)$, the error guarantee becomes $n\varepsilon$, and the space usage becomes $\frac{1}{\varepsilon} \log^2 \varepsilon n$.

6.2 Improvements

6.2.1 Randomized Compression and Input Sampling

The contribution of Agarwal et al. [23] is to have each compactor randomly delete either the odd or even items with equal probability. This reduces the error guarantee to $O(\log^{3/2} \frac{1}{\varepsilon})$.

For $n \geq \mathrm{poly}(\frac{1}{\varepsilon})$, we can sample items from the stream before feeding the sketch. For the single quantile problem, this yields a space usage of $\frac{1}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right) \sqrt{\log \frac{1}{\delta}}$, and for the all quantiles problem, $\frac{1}{\varepsilon} \log\left(\frac{1}{\varepsilon}\right) \sqrt{\log \frac{1}{\varepsilon\delta}}$.

6.2.2 Different Sized Compactors

Let $k_h \approx k_H \cdot \left(\frac{2}{3}\right)^{H-h}$. This approach does not affect the error bounds but improves space complexity.

- Lower levels have smaller compactors.
- A compactor chain with capacity 2 and randomization is essentially sampling. Denote the number of such compactors by H''.
- The total capacity of all compactors with capacity greater than 2 is:

$$\sum_{h=H''+1}^{H} k \left(\frac{2}{3}\right)^{H-h} \leq 3k$$

- Space complexity is therefore $O(k)$.

Set $k := O\left(\frac{1}{\varepsilon}\sqrt{\log \frac{1}{\delta}}\right)$. This gives a space complexity of $O(k)$ for the single quantile problem, and $O\left(\frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon}}\right)$ for the all quantiles problem.

6.2.3 Using GK-sketch

Handle the top $\log\log \frac{1}{\delta}$ levels of compactors differently. Use fixed k_h values instead of diminishing values. Thus, $k_h = k$ when $h > H - O\left(\log\log \frac{1}{\delta}\right)$. Analyzing these top levels differently yields a space complexity of $O\left(\frac{1}{\varepsilon} \log^2 \log \frac{1}{\delta}\right)$. Up to this point, the sketch remains fully mergeable.

If we replace the top $\log\log \frac{1}{\delta}$ compactors with the GK-sketch, the space complexity reduces to $O\left(\frac{1}{\varepsilon} \log\log \frac{1}{\delta}\right)$. This final step, however, prevents mergeability.

7 MP-sketch Improvements

Our primary objective was to design a sketching algorithm that enhances its performance by leveraging its own predictions. For instance, a sketching algorithm for quantile estimation could approximate the CDF of the input stream. We hypothesize that having prior knowledge about the distribution of the input would enable us to compute its quantiles more efficiently. This specific goal was not achieved, but some of the tools we examined could be useful in various applications. In the following, we present these tools.

In MP-sketch, if we invoke *Collapse* on some buffers with $w(X_i) = 1$, each buffer must first be sorted individually. Once this step is completed, all buffers on which *Collapse* was applied will be sorted. The final step involves performing a modified version of merge sort. Based on the measures, sorting the buffers is the slowest part of the algorithm, as shown in Figure 3. Thus, the first idea is to speed this up.

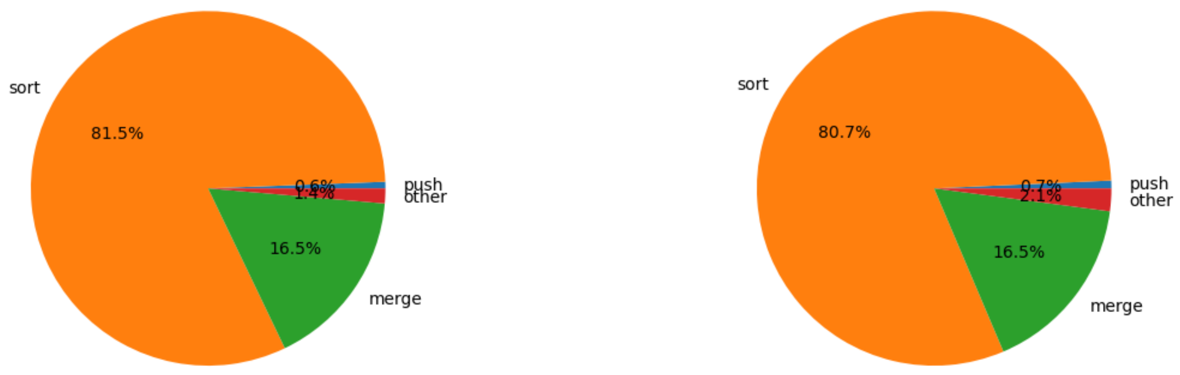


Figure 3: Operation proportions to the runtime in the original sketch. $n = 10^5$ left, $n = 10^9$ right.

7.1 Sorting with prediction

This section summarizes the results of Bai and Coester [24] and presents the operation of their algorithm. Their sorting algorithm uses a positional predictor $\hat{p}$, and a scapegoat tree with a finger.

7.1.1 Scapegoat Tree

A Scapegoat tree is a self-balancing binary search tree, invented by Arne Andersson [25]. It realizes the *Search*, *Insert*, and *Delete* operations in $O(\log n)$ amortized time. In our case, the *Delete* operation is not needed.

The *Search* operation is is not modified from a standard binary search tree, and has a worst-case time of $O(\log n)$.

The insertion operation is complicated. There is a "loosely α-height-balanced" invariant property of the scapegoat tree, which means

$$\text{height}(\text{tree}) \leq \lfloor \log_{1/\alpha} \text{size}(\text{tree}) \rfloor + 1.$$

Unlike most other self-balancing search trees, scapegoat trees are entirely flexible as to their balancing. They support any α such that $0.5 < \alpha < 1$. A high α value results in fewer balances, making insertion quicker but lookups and deletions slower, and vice versa for a low α.

When inserting a new node, we need to record its height. If it violates the height-balanced property, then a re-balance is required. First we need to find a scapegoat node. This is a node with the following properties.

$$\text{size}(\text{left}) \leq \alpha \cdot \text{size}(\text{node})$$
$$\text{size}(\text{right}) \leq \alpha \cdot \text{size}(\text{node})$$

We start the search for the scapegoat node at the newly inserted node and move upwards along the parent pointers. In fact, at each step, it is sufficient to calculate the size of the sibling subtree since we already know the size of the current node's subtree; it is initially 1, and as we move upwards, we keep track of it. The size of the parent is the sum of the size of its two children plus 1, so knowing the size of the sibling node, this can also be easily computed.

Once the scapegoat is found, the subtree rooted at the scapegoat is completely rebuilt to be perfectly balanced. This can be done in $O(n)$ time by traversing the nodes of the subtree to find their values in sorted order and recursively choosing the median as the root of the subtree. It can be seen that the worst-case time for the $Insertion$ operation is $O(n)$, but since this operation rarely needs to be performed, the amortized number of steps is $O(\log n)$.

In addition to the scapegoat tree, we also need a finger. When we talk about search trees, a finger is a pointer to one of the nodes of the tree, providing direct $O(1)$ access to that node. This could point to the smallest, or the largest node, but in our setting it will point to the last inserted node. The idea behind this is that when we insert a value into the tree that is not too far from the previous value, it is faster to find its place from here than from the root.

7.1.2 Sorting

Let $A = a_1, \dots, a_n$ be an array of n items, equipped with a strict linear order $<$. Let $p : [n] \to [n]$ be the permutation that maps each index i to the position of a_i in the sorted list.

Definition 7.1 (Positional predictor)**.** In sorting with positional predictions, the algorithm receives for each item a_i a prediction $\hat{p}(i)$ of its position $p(i)$ in the sorted list. We allow $\hat{p}$ to be any function $[n] \to [n]$, which need not be a permutation (i.e., it is possible that $\hat{p}(i) = \hat{p}(j)$ for some $i \neq j$).

The error of a positional prediction can be naturally quantified by the displacement of each element's prediction; that is, the absolute difference of the predicted ranking and the true ranking. We define the displacement error of item a_i as

$$\eta_i^{\Delta} := |\hat{p}(i) - p(i)|.$$

The sorting method is as follows. We first bucket sort (in time $O(n)$) the items in A based on their predicted positions, such that we may assume for all $i < j$ that $\hat{p}(i) \leq \hat{p}(j)$. Following the rearranged order, items in A are sequentially inserted into an initially empty finger tree T. After all insertions, we obtain the exactly sorted array by an inorder traversal of T.

Theorem 7.1. *Algorithm 1 sorts an array within $O(\sum_{i=1}^{n} \log(\eta_i^{\Delta} + 2))$ running time and comparisons.*

Algorithm 1 Sorting with prediction

```
Input: $A = a_1, \dots, a_n$, prediction $\hat{p}$
  BucketSort($A, \hat{p}$)
  $T \leftarrow$ an empty scapegoat tree with finger.
  $N \leftarrow n$
  for $i = 1, \dots, n$ do
      Insert $a_i$ into $T$.
  end for
  return nodes in $T$ in sorted order (via inorder traversal)
```

7.1.3 Application

In some network applications, it is necessary to determine the rank of elements, that is, how many elements are not greater than a given one. If this is the case, it can be considered as a freely available positional predictor and can be applied using the method described above.

If these predictions are not provided, we can use the sketch itself to generate such predictions. To estimate $r(x)$, we perform a binary search in the buffer at the highest level to find the first element that is not smaller than x. As a predictor, the earlier estimate for ε-relative error will be correct as long as the distribution of elements does not change over time. If the elements stored in the buffer do not represent the distribution, we still get an $O(k \log k)$-time algorithm asymptotically, similar to traditional sorting.

If the keys are integers, we can do even better using a clever data structure and maintain the buffer in sorted order during every insertion. This eliminates the need for sorting, though each insertion will no longer be a constant-time operation. We aim to achieve better amortized time for filling the buffer than $O(k \log k)$.

Figure 4, displayed on a logarithmic scale, shows the number of comparisons required to sort an array using `std::sort`, as well as with the best-case and worst-case predictors, for different buffer sizes. We can observe that if the predictor is sufficiently good, only $O(k)$ comparisons are needed instead of $O(k \log k)$.

7.2 x-fast trie

This structure was proposed by Dan Willard in 1982, [26] along with the y-fast trie, which can be used alternatively in our algorithm for slightly better space complexity. It uses a $\log M$ deep bitwise trie for storing integers, where M denotes the maximum value among the stored values. The leaves are stored in a doubly-linked list.

However, we do not store it as a traditional tree with parent and child pointers. Each level is stored in a hash table, and for the nodes, we only store 2 pointers, which we will call min and max. We will only need these if a node doesn't have a left or right child in the tree representation. If a node has no left child, the min points to the smallest leaf of its subtree. Correspondingly the max points to the largest leaf in its subtree. The hash function needs to be a dynamic perfect hashing or a cuckoo-hash.

7.2.1 *Successor*(x)

If x is in the lowest level (among the leaves) then we return it. If not, then make a binary search for the longest prefix of x in the levels of the tree. If this element doesn't have a right child, then

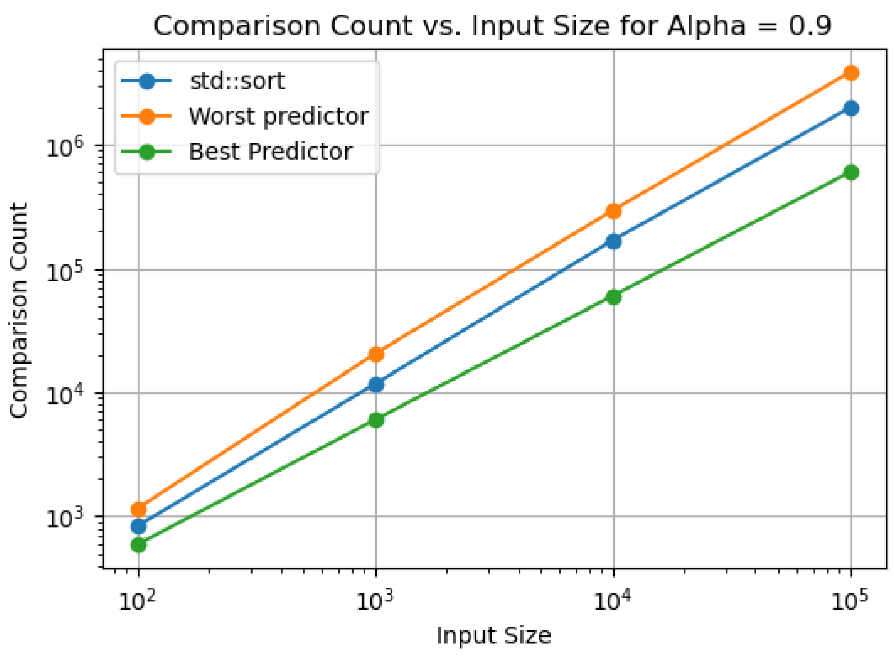


Figure 4: The number of comparisons needed to sort a buffer.

it has a max pointer to a leaf. The right sibling of this leaf node will be the successor of x. This can be done in $O(\log\log M)$.

Similarly, for the $Predecessor(x)$, we find x in the lowest level, or we find a node with binary search with a min pointer to a leaf. The left sibling of that leaf will be the predecessor of x.

7.2.2 $Insert(x)$

Let y be the longest prefix. We find the predecessor and successor of x, and then insert x between them as a new leaf. Then, moving downwards from y, we traverse the levels and insert the necessary prefixes into the hash tables. This takes $O(\log M)$ amortized time.

7.2.3 Application

By storing the buffers as x-fast tries, newly arriving elements can be easily inserted, and traversing all the elements happens in linear time by traversing the list of leaves. In addition to this, it is necessary to implement the merge operation required for collapse, that is, merging two x-fast tries. This can be done by merging the two leaf lists, removing every second element, and building a new tree in $O(k\log M)$ time. However, this is slower than merging two arrays.

In practice, it is sufficient to replace arrays with x-fast tries only at the lowest level; at higher levels, we can keep the traditional array representation. This way, we lose nothing in terms of merge time. Moreover, if the buffer at the highest level is stored as an x-fast trie, we can also handle rank queries in $O(\log\log M)$ time using the successor operation. To achieve this, the tree needs to be constructed from a sorted array (thus also providing a better implementation for the merge operation).

It can be done by building the levels from bottom to top. We start by creating the list of leaves from the array. Then, by going through this list, we will construct the level above it while maintaining the min and max pointers, and hashing the level. If we have a level in the tree, then we can construct the level above in the same way. This takes $O(k+\log M)$ time.

References

1. Chen, T. & Guestrin, C. *XGBoost: A Scalable Tree Boosting System* in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (ACM, Aug. 2016). `http://dx.doi.org/10.1145/2939672.2939785`.
2. DeWitt, D. J., Naughton, J. F. & Schneider, D. A. *Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting* in (IEEE Computer Society Press, Miami, Florida, USA, 1991), 280–291. ISBN: 0818622954.
3. Cormode, G. & Hadjieleftheriou, M. Methods for finding frequent items in data streams. *The VLDB Journal* **19,** 3–20. ISSN: 0949-877X. `https://doi.org/10.1007/s00778-009-0172-z` (Feb. 2010).
4. Kompella, R. R., Singh, S. & Varghese, G. *On Scalable Attack Detection in the Network* in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement* (Association for Computing Machinery, Taormina, Sicily, Italy, 2004), 187–200. ISBN: 1581138210. `https://doi.org/10.1145/1028788.1028812`.
5. Vass, B., Sarkadi, C. & Rétvári, G. *Programmable Packet Scheduling With SP-PIFO: Theory, Algorithms and Evaluation* in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2022), 1–6.
6. Chan, T. M., Munro, J. I. & Raman, V. Selection and Sorting in the "Restore" Model. *ACM Trans. Algorithms* **14.** ISSN: 1549-6325. `https://doi.org/10.1145/3168005` (Apr. 2018).
7. Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L. & Tarjan, R. E. Time Bounds for Selection. *J. Comput. Syst. Sci.* **7,** 448–461. `https://api.semanticscholar.org/CorpusID:3162077` (1973).
8. Dor, D. & Zwick, U. Finding the αn-th largest element. *Combinatorica* **16,** 41–58. ISSN: 1439-6912. `https://doi.org/10.1007/BF01300126` (Mar. 1996).
9. Dor, D. & Zwick, U. *Median Selection Requires Comparisons* in *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (IEEE Computer Society, USA, 1996), 125.
10. Masson, C., Rim, J. E. & Lee, H. K. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *CoRR* **abs/1908.10693.** arXiv: `1908.10693`. `http://arxiv.org/abs/1908.10693` (2019).
11. Hung, R. Y. S. & Ting, H. F. *An Omega(1/e Log 1/e) Space Lower Bound for Finding e-Approximate Quantiles in a Data Stream* in *Proceedings of the 4th International Conference on Frontiers in Algorithmics* (Springer-Verlag, Wuhan, China, 2010), 89–100. ISBN: 3642145523.
12. Manku, G. S., Rajagopalan, S. & Lindsay, B. G. Approximate Medians and Other Quantiles in One Pass and with Limited Memory. *SIGMOD Rec.* **27,** 426–435. ISSN: 0163-5808. `https://doi.org/10.1145/276305.276342` (June 1998).
13. Greenwald, M. & Khanna, S. Space-Efficient Online Computation of Quantile Summaries. **30,** 58–66. ISSN: 0163-5808. `https://doi.org/10.1145/376284.375670` (May 2001).
14. Shrivastava, N., Buragohain, C., Agrawal, D. & Suri, S. Medians and Beyond: New Aggregation Techniques for Sensor Networks. *CoRR* **cs.DC/0408039.** `http://arxiv.org/abs/cs.DC/0408039` (2004).

15. Karnin, Z., Lang, K. & Liberty, E. *Optimal Quantile Approximation in Streams* 2016. arXiv: `1603.05346 [cs.DS]`.

16. Felber, D. & Ostrovsky, R. *A randomized online quantile summary in $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ words* 2015. arXiv: `1503.01156 [cs.DS]`.

17. Ivkin, N., Liberty, E., Lang, K., Karnin, Z. & Braverman, V. *Streaming Quantiles Algorithms with Small Space and Update Time* 2019. arXiv: `1907.00236 [cs.DS]`.

18. Ivkin, N., Yu, Z., Braverman, V. & Jin, X. *QPipe: Quantiles Sketch Fully in the Data Plane* in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Association for Computing Machinery, Orlando, Florida, 2019), 285–291. ISBN: 9781450369985. `https://doi.org/10.1145/3359989.3365433`.

19. Gan, E., Ding, J., Tai, K. S., Sharan, V. & Bailis, P. Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proc. VLDB Endow.* **11,** 1647–1660. ISSN: 2150-8097. `https://doi.org/10.14778/3236187.3236212` (July 2018).

20. Munro, J. & Paterson, M. Selection and sorting with limited storage. *Theoretical Computer Science* **12,** 315–323. ISSN: 0304-3975. `https://www.sciencedirect.com/science/article/pii/0304397580900614` (1980).

21. Li, K.-H. Reservoir-Sampling Algorithms of Time Complexity O(n(1 + Log(N/n))). *ACM Trans. Math. Softw.* **20,** 481–493. ISSN: 0098-3500. `https://doi.org/10.1145/198429.198435` (Dec. 1994).

22. Cormode, G., Korn, F., Muthukrishnan, S. & Srivastava, D. *Effective computation of biased quantiles over data streams* in *21st International Conference on Data Engineering (ICDE'05)* (2005), 20–31.

23. Agarwal, P. K. *et al. Mergeable summaries* in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Association for Computing Machinery, Scottsdale, Arizona, USA, 2012), 23–34. ISBN: 9781450312486. `https://doi.org/10.1145/2213556.2213562`.

24. Bai, X. & Coester, C. *Sorting with predictions* in *Proceedings of the 37th International Conference on Neural Information Processing Systems* (Curran Associates Inc., New Orleans, LA, USA, 2024).

25. Andersson, A. *Improving Partial Rebuilding by Using Simple Balance Criteria* in *Proceedings of the Workshop on Algorithms and Data Structures* (Springer-Verlag, Berlin, Heidelberg, 1989), 393–402. ISBN: 3540515429.

26. Willard, D. E. Log-logarithmic worst-case range queries are possible in space Θ(N). *Information Processing Letters* **17,** 81–84. ISSN: 0020-0190. `https://www.sciencedirect.com/science/article/pii/0020019083900753` (1983).