VLDBJ manuscript No.
(will be inserted by the editor)

Quantiles over Data Streams: Experimental Comparisons, New Analyses, and Further Improvements

Ge Luo · Lu Wang · Ke Yi · Graham Cormode

Received: date / Accepted: date

Abstract A fundamental problem in data management and analysis is to generate descriptions of the distribution of data. It is most common to give such descriptions in terms of the cumulative distribution, which is characterized by the quantiles of the data. The design and engineering of efficient methods to find these quantiles has attracted much study, especially in the case where the data is given incrementally, and we must compute the quantiles in an online, streaming fashion. While such algorithms have proved to be extremely useful in practice, there has been limited formal comparison of the competing methods, and no comprehensive study of their performance. In this paper, we remedy this deficit by providing a taxonomy of different methods, and describe efficient implementations. In doing so, we propose new variants that have not been studied before, yet which outperform existing methods. To illustrate this, we provide detailed experimental comparisons demonstrating the tradeoffs between space, time, and accuracy for quantile computation.

Keywords Data stream algorithms · quantiles.

1 Introduction

Given a large amount of data, a first and foundational problem is to describe the data distribution. If the data follows a known distribution family, such as Gaussian, it can be described succinctly by the parameters of the distribution. This is rarely the case in practice, which thus calls for nonparametric methods. Quantiles are the mostly commonly used nonparametric representation for data distributions. They correspond to the cumulative distribution function (cdf), which in turn yields the probability distribution function (pdf). Thus, quantile computation is arguably one of the most fundamental problems in data analysis. For example, rankings are often expressed in terms of percentiles, such as for giving results of standardized testing, or measuring children's physical development. Distributions are commonly compared via quantiles, in the form of quantile-quantile plots, which leads to the *Kolmogorov-Smirnov divergence*, one of the most commonly used distance measures between distributions.

Computing the quantiles has significant practical importance: Standard statistical packages, such as R and Excel, include functions to compute the median and other quantiles. In the Sawzall language that is the basis for all of Google's log data analysis, quantile is one of the seven basic operators defined (the others include sum, max, top-k, and count-distinct) [24]. The quantiles also play an important role in network health monitoring for Internet service providers [8] and data collection in wireless sensor networks [26].

The problem is also intellectually interesting enough to have attracted a lot of prior study, from both the algorithms and the database community, sometimes investigated under the name of "the selection problem" or "order statistics". Algorithmic interest dates back to at least 1973, when the celebrated *linear-time selection* algorithm was invented [4]. In the past 35 years, this problem has received particular attention in the streaming model, i.e., the data elements arrive one by one in a streaming fashion, and the algorithm only has lim-

G. Luo · L. Wang · K. Yi
Department of Compute Science and Engineering, HKUST, HongKong
E-mail: (luoge, luwang, yike)@cse.ust.hk

G. Cormode
Department of Computer Science, University of Warwick
E-mail: G.Cormode@warwick.ac.uk

ited memory to work with. There have been numerous algorithms proposed in this setting, using a variety of different techniques and offering different performance guarantees [23, 15, 21, 22, 13, 7, 12, 18, 27]. In addition, there have been many studies on variations and extensions of the problem, such as computing quantiles over sliding windows [3], over distributed data [26, 1, 16, 17], continuous monitoring of quantiles [9, 30], biased quantiles [10], computing quantiles using GPUs [14], etc.

The median has long been recognized as a more stable statistic of data distribution than, say, the average, in the sense that it is very robust to outliers. The quantiles are a natural generalization of the median. Let S be a (multi)set of n elements drawn from a totally ordered universe. Recall that the ϕ-quantile of S, for some $0 < \phi < 1$, is the element whose rank is $\lfloor \phi n \rfloor$ in S, where the rank of an element x is the number of elements in S smaller than x.

The quantiles can be easily found by sorting if sufficient space is available. The problem becomes significantly more challenging in the streaming model, which is the focus of this work. It dates back to 1980, when Munro and Paterson [23] showed that any algorithm that computes the median exactly with p passes over the data has to use $\Omega(n^{1/p})$ space. Thus, approximation is necessary for any streaming quantile algorithm using sublinear space. Recall that a streaming algorithm is one that makes one pass over the data and perform the desired computation. Often, the algorithm is not given the knowledge of n, the length of the stream, so that the algorithm has to be ready to stop and provide the results at any time. This corresponds to the practical setting where the stream is conceptually an infinite sequence of elements, and the algorithm should always be ready to provide the results for the data seen so far. In line with most prior work, we also adopt this requirement.

Subsequently, the problem of computing approximate quantiles over streaming data has been widely studied in the past three decades (which will be reviewed shortly). The commonly used notion of approximation for this problem is the following: For an error parameter $0 < \varepsilon < 1$, the *ε-approximate ϕ-quantile* is any element with rank between $(\phi - \varepsilon)n$ and $(\phi + \varepsilon)n$. Since quantiles are used for approximating the data distribution anyway, and the input data is often noisy in itself, allowing some errors in the computed quantiles is often tolerable.

However, despite the importance of the problem and the many efforts devoted, a complete and clear picture of the problem still appears elusive, both theoretically and empirically. We lack matching upper and lower bounds for the problem, which constitutes a top open problem in data stream algorithms (see `http://sublinear.info/2`). Moreover, existing empirical studies are both incomplete and outdated. In this work, we set out to address this issue, and carry out an extensive experimental comparison of various quantile algorithms that have not been compared before. In doing so, we also propose new variants that have not been studied before, yet which turn out to perform the best.

1.1 Classification of algorithms

Depending on different models, algorithms for computing quantiles of data streams can be classified along the following axes:

1. Whether elements can only be added or can be both inserted and deleted.

   In the *cash register* model, elements arrive one by one in the stream and they are never removed. In the *turnstile model*, the stream consists of a sequence of updates where each update either inserts an element or deletes one, but a deletion cannot delete an element that does not exist. When there are duplicates, this means that the multiplicity of any element cannot go negative.

2. What operations are allowed on the elements.

   In the *comparison model*, the algorithm can only access the elements through comparisons. Implicitly, this means that the algorithm must store a set of elements that it has observed from the stream (together with some extra information), and only return from this set as quantiles in the end, namely it cannot "create" or "compute" elements to return. In the *fixed universe* model, the elements are integers in the universe $[u] = \{0, \ldots, u-1\}$. Here, the algorithm is allowed to perform bit manipulation tricks, and return elements that may have never appeared in the stream as quantiles provided they satisfy the approximation guarantees. Clearly, the comparison model is more restrictive, so any comparison-based algorithm also works in the fixed universe model, but not vice-versa. However, the benefit of comparison-based algorithms is that they can handle elements that cannot be easily mapped to a fixed universe $[u]$, such as variable-length strings or user-defined types[1].

3. Whether the algorithm is deterministic or randomized.

[1] Note that floating-point numbers in standard representations (e.g. IEEE 754) can be mapped to integers in a fixed universe in an order-preserving fashion.

We are not aware of any Las Vegas quantile algorithms, so we will only consider Monte Carlo randomization, where an algorithm may return an incorrect quantile (i.e., exceeding the stated ε error) with a small probability. We usually consider the probability that the algorithm returns *all* quantiles correctly, but this will be the case as long as it is correct on the $1/\varepsilon - 1$ quantiles for $\phi = \varepsilon, 2\varepsilon, \ldots, 1-\varepsilon$. The quantiles in between any two of these quantiles will thus have error at most 2ε, and scaling ε down by a factor of 2 will restore the ε-approximation guarantee for all quantiles. To simplify the bounds, most theoretical analyses make this probability a constant. This probability can always be boosted using standard techniques; in practice, due to the looseness of the analysis, it suffices to set the success probability to a reasonable constant.

1.2 Existing quantile algorithms and our new findings

1.2.1 The cash register model

In their pioneering paper [23], Munro and Paterson also gave a p-pass algorithm for computing exact quantiles. Although not analyzed explicitly, the first pass of the algorithm yields a streaming algorithm for computing ε-approximate quantiles using $O(\frac{1}{\varepsilon}\log^2(\varepsilon n))$ space. This fact was made more explicit by Manku et al. [21], who also proposed another algorithm that is empirically better, though it has the same worst-case space bound. In 2001, Greenwald and Khanna [15] designed a quite ingenious algorithm (referred to as the GK algorithm below) and showed that it uses $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ space in the worst case. But interestingly, their experimental study implements a simplified algorithm (referred to as GKAdaptive below), for which it is not clear if the $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ space bound still holds. Nevertheless, they showed that this algorithm empirically outperforms that of Manku et al. [21]. All these algorithms are deterministic and comparison-based. Hung and Ting [18] showed an $\Omega(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ space lower bound for such algorithms. In this category, the GK algorithm is generally considered to be the best, both theoretically and empirically (in its respective versions).

In 2004, Shrivastava et al. [26] designed a deterministic, fixed-universe algorithm, called *q-digest*, that uses $O(\frac{1}{\varepsilon}\log u)$ space. This algorithm was designed for quantile computation in sensor networks, and is a *mergeable summary* [1], a model that is more general than streaming. But no better fixed-universe algorithm is known in the streaming model. Note that the $\log u$ and $\log(\varepsilon n)$ terms are not comparable in theory, and [26] did not include an experimental comparison with the GK algorithm.

Randomized algorithms have also been investigated. Classic results [28] show that a random sample of size $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$ preserves all quantiles within ε error with at least a constant probability. This fact was reproved in [21] and exploited for computing quantiles by feeding a random sample to a deterministic algorithm. But this algorithm requires the *a priori* knowledge of n, so it is not a true streaming algorithm. Later, Manku et al. [22] proposed a randomized algorithm (henceforth referred to as MRL99) that does not need the knowledge of n, and showed that its space requirement is $O(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon})$. Note that the $\log^2\frac{1}{\varepsilon}$ factor could be either larger or smaller than the $\log(\varepsilon n)$ factor of GK, and these two algorithms have not been compared experimentally. Subsequently, Agarwal et al. [1] gave a more complicated algorithm with a space complexity of $O(\frac{1}{\varepsilon}\log^{1.5}(\frac{1}{\varepsilon}))$ without implementation. Very recently, Felber and Ostrowsky [11] provided a randomized algorithm (also without implementation) for this problem that achieves $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ space cost. However, the hidden constant in the big-Oh is very substantially large that it makes this algorithm only of theoretical interests. Our prototype implementation of this algorithm confirmed that it is not competitive in practice with others, so we do not consider it further in this empirical study.

In this paper, we empirically compare GKAdaptive, q-digest, and MRL99. We omit results for the algorithms of Munro and Paterson [23] and the earlier algorithm of Manku et al. [21], since they have previously been demonstrated to be outperformed by the GK algorithm. We have also implemented GKTheory, and found out that it does not perform as well as GKAdaptive, despite the $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ space guarantee of the former.

Our experimental study reveals that MRL99 generally performs the best, but it suffers from the following undesirable properties. First, it uses some fairly complex rules for maintaining its samples and sets its parameters delicately by solving an optimization problem, which increases implementation difficulty. Second, as the algorithm is difficult to analyze, the analysis given in [22] is quite pessimistic, resulting in an $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon}))$ bound. In practice, this mean that for an error target ε, we often allocate more space than necessary. Through our experimental study, we observed that many of the details of MRL99 were not actually needed, and the algorithm can be significantly simplified without affecting its performance. In addition, we give a new analysis on this simpler algorithm (referred to as Random), leading to an improved $O(\frac{1}{\varepsilon}\log^{1.5}(\frac{1}{\varepsilon}))$ bound.

Table 1: All algorithms evaluated in this paper. Those marked with * are new variants.

Algorithm	Space	Update time	Randomization	Model
GKAdaptive	—	$O(\log \text{Space})$	Deterministic	Comparison
GKTheory	$O\left(\frac{1}{\varepsilon}\log(\varepsilon n)\right)$	$O\left(\log\frac{1}{\varepsilon}+\log\log(\varepsilon n)\right)$	Deterministic	Comparison
FastQDigest	$O\left(\frac{1}{\varepsilon}\log u\right)$	$O\left(\log\frac{1}{\varepsilon}+\log\log u\right)$	Deterministic	Fixed universe
MRL99	$O\left(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon}\right)$	$O\left(\log\frac{1}{\varepsilon}\right)$	Randomized	Comparison
Random *	$O\left(\frac{1}{\varepsilon}\log^{1.5}\frac{1}{\varepsilon}\right)$	$O\left(\log\frac{1}{\varepsilon}\right)$	Randomized	Comparison
Random subset sum	$O(\frac{1}{\varepsilon^2}\log^2 u\log(\frac{\log u}{\varepsilon}))$	$O(\frac{1}{\varepsilon^2}\log^2 u\log(\frac{\log u}{\varepsilon}))$	Randomized	Fixed universe
DCM	$O(\frac{1}{\varepsilon}\log^2 u\log(\frac{\log u}{\varepsilon}))$	$O(\log u\log(\frac{\log u}{\varepsilon}))$	Randomized	Fixed universe
DCS *	$O(\frac{1}{\varepsilon}\log^{1.5} u\log^{1.5}(\frac{\log u}{\varepsilon}))$	$O(\log u\log(\frac{\log u}{\varepsilon}))$	Randomized	Fixed universe

On the other hand, GKAdaptive remains the most competitive deterministic algorithm. However, the original paper [15] focused only on space usage, and did not elaborate on the running time of the algorithm. In this paper, we have identified two different ways to implement the algorithm, and investigated their practical performance.

1.2.2 The turnstile model

The turnstile model presents additional challenges, due to the deletions of elements. Attempts to adapt the above algorithms to this model can often be thwarted by finding particularly adversarial patterns of insertions and subsequent deletions. In fact, it can be argued that no comparison-based algorithm is possible using sublinear space under the turnstile model: Imagine that we first insert n elements and then delete all but one. Before the deletions, the algorithm has no information about which element will survive, and because the comparison-based model does not allow the creation or computation of elements to return, it has to retain all n elements. Therefore, all turnstile algorithms work only for a fixed universe, and are mostly randomized algorithms. Deterministic algorithms for the fixed universe model have been provided: Ganguly and Majumder describe an algorithm which uses $O(\frac{1}{\varepsilon^2}\log^5 u\log(\frac{\log u}{\varepsilon}))$ space [12]. The high dependency on $\frac{1}{\varepsilon}$ and $\log u$ is not considered practical.

Existing algorithms in the turnstile model all make use of a *dyadic structure* imposed over the universe of possible elements. More precisely, we build $\log u$ levels, decomposing the universe $[u]$ as follows. In level 0, every integer in $[u]$ is by itself; in level i, the universe is partitioned into intervals of size 2^i; the top level thus consists of only one interval $[0, u-1]$. Every interval in every level in this hierarchy is called a *dyadic* interval. The algorithms make use of randomized *sketch* data structures which process a stream of updates in the turnstile model, and allow the frequency of any element to be estimated [5, 7]. Each level keeps a frequency estimation sketch that can be used to estimate the total number of elements in any interval on that level. To find the rank of a given element x, we decompose the interval $[0, x-1]$ into the disjoint union of at most $\log u$ dyadic intervals, one from each level. From the frequency estimation sketch, we estimate the number of elements in each dyadic interval, and then add them up. Then for any given ϕ, we can find an approximate ϕ-quantile by doing a binary search on $[u]$ to find the largest element whose rank is below ϕn.

Different frequency estimation sketches have been proposed to instantiate this outline. Gilbert et al. [13] first proposed the *random subset sum* sketch for this purpose, which results in a size of $O(\frac{1}{\varepsilon^2}\log^2 u\log(\frac{\log u}{\varepsilon}))$. Later, Cormode and Muthukrishnan applied the Count-Min sketch in the dyadic structure (the resulting algorithm is referred to as DCM, for "Dyadic Count-Min"), reducing the overall size to $O(\frac{1}{\varepsilon}\log^2 u\log(\frac{\log u}{\varepsilon}))$ [7]. This remains the best bound in the turnstile model. In this paper, we propose to use the Count-Sketch [5] (the algorithm is thus referred to as DCS, for "Dyadic Count-Sketch"), and give a new analysis showing that it further reduces the space to $O(\frac{1}{\varepsilon}\log^{1.5} u\log^{1.5}(\frac{\log u}{\varepsilon}))$, which is the best bound for this problem under the turnstile model. We also carry out an experimental comparison of these different variants, which shows that DCS is not only theoretically the best, but also gives superior performance in practice.

Finally, to further improve the accuracy of DCS, we design a fast post-processing step to eliminate the discrepancies in the frequency estimates across different levels, using the *ordinary least squares* method. Experimental results show that this post-processing step can further reduce the error of DCS by 60–80%.

Table 1 summarizes all the algorithms that we evaluate in this paper, in both the cash register and the turnstile model.

1.3 Relation to conference publication

This paper extends our earlier work [29], in which we identified GKAdaptive to be the best deterministic algorithm, Random to be the best randomized algorithm, while DCS the best algorithm in the turnstile model. In this paper, we develop several new ideas that lead to further improvements to these algorithms. In particular, we give a new implementation of GKAdaptive, called GKArray, which uses buffering techniques to significantly improve the running time (Section 2.1.2). For DCS, we give a novel "post-processing" step to make better use of the estimates generated, to give substantially improved accuracy for this class of algorithms (Section 3.2).

To keep this paper focused and avoid redundancy, we will only describe and analyze the above three most competitive algorithms in their respective categories, together with the new improvements introduced in this paper. The details of the other algorithms can be found in the original conference version of this paper [29].

2 Cash Register Algorithms

In this section, we describe the cash register algorithms. Recall that in this model, there are only insertions in the stream. We use n to denote the current number of elements in the stream. We use $r(x)$ to denote the rank of x in all the elements received so far.

2.1 The GK algorithm

The GK algorithm [15] is a deterministic, comparison-based quantile algorithm. It maintains a list of tuples $L = \langle (v_i, g_i, \Delta_i) \rangle$, where the v_i's are elements from the stream such that $v_i \leq v_{i+1}$. The g_i's and Δ_i's are integers satisfying the following conditions:

(1) $\sum_{j \leq i} g_j \leq r(v_i) + 1 \leq \sum_{j \leq i} g_j + \Delta_i$;
(2) $g_i + \Delta_i \leq \lfloor 2\varepsilon n \rfloor$.

Note that condition (1) gives both a lower and an upper bound on the possible ranks of v_i. Also, $g_i + \Delta_i - 1$ is the maximum possible number of elements between v_{i-1} and v_i, so (2) ensures that for any $0 < \phi < 1$, there must be an element in the list whose rank is within εn of ϕn. Thus, to extract the ϕ-quantile, we can find the smallest i such that $\sum_{j \leq i} g_j + \Delta_i > 1 + \lceil \phi N \rceil + \max_i (g_i + \Delta_i)/2$, and then report v_{i-1}. It can be verified that this v_{i-1} will be a valid ε-approximate ϕ-quantile.

The list is initialized as $L = \langle (\infty, 1, 0) \rangle$. To insert a new element v, we find its successor in L, i.e., the smallest v_i such that $v_i > v$, and insert the tuple $(v, 1, \lfloor 2\varepsilon n \rfloor)$ right before v_i. We may also remove tuples: To remove (v_i, g_i, Δ_i), we set $g_{i+1} \leftarrow g_i + g_{i+1}$ and remove the tuple from L. Note that this may violate condition (2) for the next tuple, so we call a tuple *removable* if $g_i + g_{i+1} + \Delta_{i+1} \leq \lfloor 2\varepsilon n \rfloor$.

In order to keep $|L|$ small, the original paper [15] gave a fairly complex COMPRESS procedure to carefully select tuples to remove while maintaining (1). It is performed once every $\frac{1}{2\varepsilon}$ incoming elements. It has been shown that after the COMPRESS procedure, $|L|$ is at most $\frac{11}{2\varepsilon} \log(2\varepsilon n)$. COMPRESS can be done in time $O(|L|)$, so if it is performed only when $|L|$ doubles, its amortized cost is $O(1)$ per update. An insertion can be done in time $O(\log |L|)$ if we maintain a binary search tree on top of L, therefore the amortized per-element update time is $O\left(\log \frac{1}{\varepsilon} + \log\log(\varepsilon n)\right)$.

2.1.1 Variant: GKAdaptive

The algorithm described was structured to permit theoretical analysis of the space cost; in the paper [15], the authors instead implemented the following variant:

1. To insert v, insert to L a tuple $(v, 1, g_i + \Delta_i - 1)$ instead of $(v, 1, \lfloor 2\varepsilon N \rfloor)$.
2. Following an insertion, try to find a removable tuple in L. If there is one, remove it; otherwise $|L|$ increases by 1.

Note that COMPRESS is never called in this variant, so the $O(\frac{1}{\varepsilon} \log(\varepsilon n))$ bound may not hold.

The original paper [15] did not specify how to find a removable tuple, as they did not focus on running time. There can be two ways to implement this efficiently. The first is to maintain a min-heap on the tuples in L ordered by $g_i + g_{i+1} + \Delta_{i+1}$. When a new tuple is inserted, we first check if the tuple itself is removable, and remove it immediately if so. Otherwise, we check the top tuple in the heap, and remove it if it is removable. If the top tuple in the heap is not removable, then no others are. When this happens, $|L|$ increases by 1. The heap can be maintained in $O(\log |L|)$ time per element, so the asymptotic update time is not affected. We refer to this variant as GKAdaptive.

2.1.2 Variant: GKArray

A quite different way to implement the algorithm above is to do defer some actions and operate in a "batch mode". We store all tuples in L in an array instead of a list. We do not insert tuples into the array so we will not need the binary search tree. We remove the use of the heap as well. Instead, the algorithm maintains a buffer to store the incoming elements from the stream, and merges the buffered elements into L whenever the buffer is full. Specifically, the algorithm proceeds as follows.

1. Buffer the arriving elements into an array A. When A is full, sort the elements in A.
2. Merge A into L. More precisely, we scan A and L in parallel. In doing so, for each $v \in A$, we can find the smallest $v_i \in L$ such that $v_i > v$, and thus can compute the (v, g, Δ) tuple for v. During the merge, we also check each tuple in A or L to see if it is removable, and if so remove it (i.e., don't output it to the new L).
3. After the merging, flush the buffer and go back to step 1).

Since the size of the buffer is $\Theta(|L|)$, the cost of step 1) is $\Theta(|L| \log |L|)$, which is $O(\log |L|)$ per element amortized. The cost of step 2) is just $O(1)$ per element amortized. Thus, this variant has the same asymptotic update time (though amortized) as GKAdaptive. However, since sorting and merging are both much more cache-efficient than searching in a binary search tree and heap operations, this variant could be much faster in practice than GKAdaptive. We refer to this variant as GKArray.

2.2 The randomized algorithm: Random

We now describe a randomized quantile algorithm, which can be seen as a simplified version of the one by Manku et al. [22]. It is also inspired by the algorithm by Agarwal et al. that provides the mergable property [1]. We denote this algorithm as Random. It will correctly report all quantiles with constant probability.

Setting $h = \log \frac{1}{\varepsilon}$, $b = h + 1$ and $s = \frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon}}$, Random maintains b buffers of size s each. Each buffer X is associated with a level $l(X)$.

Two buffers at the same level l can be merged into one buffer at level $l+1$. To do so, in the sorted sequence of elements from both buffers, we randomly choose half of them: either those at odd positions, or those at even positions, each with probability $1/2$. The merged 2 buffers are then marked as empty.

Initially all buffers are marked as empty. We set the active level $l = \max\{0, \lceil \log \frac{n}{s2^{h-1}} \rceil\}$. If there is an empty buffer X, we read the next $2^l s$ elements from the stream. For every 2^l elements, we randomly pick one and add it to X. Thus X contains s sampled elements, becoming full, unless the stream is terminated. X is associated with level l. Whenever all buffers becomes full, we find the lowest level that contains at least 2 buffers, and merge 2 of them together.

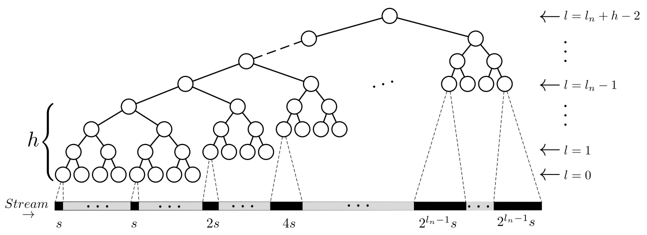


Fig. 1: Illustration of Random.

In the end, the rank of an element v is estimated as $\hat{r}(v) = \sum_X 2^{l(X)} |\{i < v | i \in X\}|$, where X ranges over all nonempty buffers. A ϕ-quantile is reported as the element whose estimated rank is closest to ϕn, which can be found using a binary search.

Figure 1 illustrates the algorithm. New elements of the stream arrive at the right of the figure. The algorithm can be understood in terms of a binary tree imposed over the stream. Each node in the tree corresponds to a buffer, and internal nodes are formed from the merger of their two children. Initially, leaf buffers are filled from s elements directly from the stream, but as the stream goes on, sampling is applied to fill the leaf buffers. There are 2^{h-1} leaf buffers at level 0, each storing s elements from the stream; for $1 \leq l < l_n$, there are 2^{h-2} leaf buffers at level l, each storing s elements sampled from $2^l s$ elements in the stream. There are 2^{h-2} non-leaf buffers at level l for any $1 \leq l \leq l_n$, and $2^{l_n+h-l-2}$ non-leaf buffers for $l_n+1 \leq l \leq l_n+h-2$.

Space and time analysis. Two buffers can be merged in $O(s)$ time, and the total number of merges is $O(n/s)$ throughout the entire data stream, which is amortized $O(1)$ for each update. Each buffer is sorted when it just becomes full, which can be done in $O(s \log s)$ time, so $O(\log s)$ per update amortized. Hence the amortized update time is $O(\log s) = O\left(\log \frac{1}{\varepsilon}\right)$.

The space bound is simply $bs = O\left(\frac{1}{\varepsilon} \log^{1.5} \frac{1}{\varepsilon}\right)$.

Error analysis. We show that with constant probability, this algorithm finds all quantiles correctly.

Since our analysis will focus on the asymptotics, we assume that n/s is a power of 2, which means that when the stream terminates, l has been just increased by 1 and becomes $l_n = \log(\frac{n}{2^h s}) + 2$. In order to simplify the proof, at this point we merge all the buffers into one, whose level is $l_n + h - 2 = \log(n/s)$. Note that this operation can only increase the error.

If the estimated ranks of all the $1/\varepsilon - 1$ elements that rank at $\varepsilon n, 2\varepsilon n, \ldots, (1-\varepsilon)n$ are correct (i.e., with at most additive εn error), then all the quantiles can be answered correctly. By the union bound, it suffices to ensure that each rank is correct with probability at least $1 - \varepsilon$.

When the algorithm estimates the rank of any element, the error comes from two sources: random sampling and random merging. Clearly, the expected error of each type is zero, so the estimator is unbiased. Now we analyze the probability that the error is larger than εn. For the random sampling part, consider any sampled element at level l, which has been chosen from 2^l elements, so the error is between -2^l and 2^l. By Hoeffding's inequality, the probability of the absolute value of their sum exceeding εn is at most

$$\exp\left(-\frac{2(\varepsilon n)^2}{\sum_{\text{leaf buffer } X} 4^{l(X)} s}\right) = \exp\left(-\Theta\left(\varepsilon^2 2^h s\right)\right) < \varepsilon/2,$$

since the summation over X is dominated by the contribution from the highest level, where $l(X) = \log n/s$.

Next consider the error from the random merging. Merging 2 buffers at level l may contribute an error between -2^l and 2^l. Again by Hoeffding's inequality, the probability that the total error exceeds εn is bounded in terms of the sum of the squares of the absolute errors (also dominated by the contribution of the highest level), as

$$\exp\left(-\frac{2(\varepsilon n)^2}{\sum_{\text{non-leaf buffer } X} 4^{l(X)}}\right) = \exp\left(-\Theta\left(\varepsilon^2 s^2\right)\right) < \varepsilon/2.$$

Finally, when n/s is not a power of 2, then there will be more than one buffer left even if we perform all possible merges. However, as the weights of these buffers are geometrically smaller, this does not change the error asymptotically.

3 Turnstile Algorithms

Recall that all algorithms in the turnstile model build upon the dyadic structure over the universe $[u]$ as described in Section 1, and use a frequency estimation sketch for each level. Known turnstile quantile algorithms only differ in the sketches they choose to use. Over a stream of updates with both insertions and deletions of elements, a frequency estimation sketch should be able to return an estimate of the frequency of any given element x. Note that when used in level i in the dyadic structure (the bottom level is level 0), an "element" is actually a dyadic interval of length 2^i, and the frequency estimation sketch summarizes a reduced universe $[u/2^i]$. Thus, for an integer x in the stream, we take its first $\log(u) - i$ bits to map it to level i. Finally, it is obvious that if the reduced universe size $u/2^i$ is smaller than the sketch size, we should maintain the frequencies exactly, rather than using a sketch.

In the turnstile model, we use n to denote the number of elements currently remaining, which is at most the stream length.

3.1 DCS: Dyadic Count-Sketch

We propose to use the Count-Sketch [5] for frequency estimation in the dyadic structure. The Count-Sketch consists of an array C of $w \times d$ counters. For each row i, it uses a pairwise independent hash function $h_i : [u] \to [w]$ that maps the elements in the (reduced) universe to the w counters in this row, as well as a 4-wise independent hash function $g_i : [u] \to \{-1, +1\}$ that maps each element to -1 or $+1$ with equal probability. To insert/delete an element x in the sketch, for each row i, we add/subtract $g_i(x)$ to $C[i, h_i(x)]$. To estimate the frequency of x, we return the median of $g_i(x) \cdot C[i, h_i(x)], i = 1, \ldots, d$ (assuming d is odd).

By setting $w = O(1/\varepsilon)$ and $d = O(\log \frac{1}{\delta})$, the Count-Sketch returns an estimate with more than εn error with probability at most δ, which is the same as the Count-Min sketch. However, we observe another property of the Count Sketch that makes it appealing for the quantile problem, that it produces an unbiased estimator. Since we add up the estimates from $\log u$ sketches in the dyadic structure, it is likely that some of the positive and negative errors will cancel each other out, leading to a more accurate final result. Below we give a new analysis showing that this intuition in fact leads to an asymptotic improvement over using the Count-Min sketch for the quantile problem (DCM).

Analysis. Below we prove that DCS can return all ε-approximate quantiles with constant probability using space $O(\frac{1}{\varepsilon} \log^{1.5} u \log^{1.5}(\frac{\log u}{\varepsilon}))$.

Consider the estimators $Y_i = g_i(x) \cdot C[i, h_i(x)], i = 1, \ldots, d$. Each Y_i is clearly unbiased, since $g_i(x)$ maps to -1 or $+1$ with equal probability. Let Y be the median of the Y_i's. The median of independent unbiased estimators is not necessarily unbiased, but if each estimator also has a symmetric pdf, then this *is* the case. This result seems to be folklore. In our case, each Y_i has a symmetric pdf, so Y is still unbiased.

Using the same argument as for the Count-Min sketch, we have

$$\Pr[|Y_i - \mathrm{E}[Y_i]| > \varepsilon n] < 1/4.$$

Since Y is the median of the Y_i's, by a Chernoff bound, we have

$$\Pr[|Y - \mathrm{E}[Y]| > \varepsilon n] < \exp(-O(d)).$$

Now consider adding up $\log u$ such estimators; the sum must still be unbiased. By the union bound, the probability that every estimate has at most εn error is at least $1 - \exp(-O(d)) \cdot \log u$. Conditioned upon this event happening, we can use Hoeffding's inequality to bound the probability that the sum of $\log u$ such (independent) estimators deviate from its mean by more than t as

$$2 \exp\left(-\frac{2t^2}{(2\varepsilon n)^2 \log u}\right).$$

We see that if we set $t = \Theta\left(\varepsilon n\sqrt{\log u}\right)$, this probability will be a constant. This means that, summing over the $\log u$ levels, the error only grows proportionally to $\sqrt{\log u}$.

To make this bound rigorous, we must ensure that all quantiles are correct with constant probability. So each such sum should fail with probability no more than $\varepsilon/\log u$. Thus, we set $t = \Theta\left(\varepsilon n\sqrt{\log u \log(\frac{\log u}{\varepsilon})}\right)$. In addition, we need to choose $d = \Theta(\log(\frac{\log u}{\varepsilon}))$ to ensure that the prerequisite condition holds with probability at least $1 - \varepsilon/\log u$. Finally, to get εn error in the end, we use a parameter $\varepsilon' = \varepsilon \Big/ \sqrt{\log u \log(\frac{\log u}{\varepsilon})}$ in the sketches (i.e., $w = 1/\varepsilon'$). Thus, the total space of DCS is $w \cdot d \cdot \log u = O(\frac{1}{\varepsilon} \log^{1.5} u \log^{1.5}(\frac{\log u}{\varepsilon}))$, and its update time is $d \cdot \log u = O(\log u \log(\frac{\log u}{\varepsilon}))$, as claimed in Table 1.

3.2 Post processing

All the quantile algorithms make use of the dyadic structure and use an independent frequency estimation sketch for each level. However, the true frequencies across different levels are *not* independent. Consider the toy example in Figure 2, which shows a dyadic structure on a tiny universe $\{0, 1, 2, 3\}$. For each node v in this binary tree, DCS returns an unbiased estimator Y_v for the number of elements in the corresponding interval. Writing x_v to denote the true frequency at node v, we have the prior knowledge that $x_1 = x_2 + x_3$, $x_2 = x_4 + x_5$ and $x_3 = x_6 + x_7$. However, it is very unlikely that $Y_1 = Y_2 + Y_3$ or $Y_2 = Y_4 + Y_5$. The question is thus, can we use this prior knowledge to improve the accuracy of the Y_i's (which in turn leads to better accuracy for quantile approximation). The answer is yes, at least on this toy example. Let us assume that all the Y_i's are independent (actually, pairwise independence suffices), and have the same variance σ^2. Consider Y_2. If we set $Y_2' = Y_1/2 + (Y_2 - Y_3)/3 + (Y_4 + Y_5 - Y_6 - Y_7)/6$, it can be checked that $\mathrm{E}[Y_2'] = \mathrm{E}[Y_2]$ and $\mathrm{Var}(Y_2') = \frac{7}{12}\sigma^2$, namely, Y_2' is still an unbiased estimator of x_2 but with a smaller variance than the original estimator. The reason we can achieve this improvement is that the other estimators include information about x_2, which can be used in conjunction with Y_2. For instance, $Y_4 + Y_5$ and $Y_1 - Y_6 - Y_7$ are unbiased estimators of x_2 (from independent estimators), and these can be combined to reduce the variance.

More questions naturally follow: Is this the best estimator of x_2? Is there a principled approach? Can we compute the improved estimators efficiently? These are the questions we address in this section.

3.2.1 Ordinary least squares

We can formalize the problem as follows. Let $x = (x_i)$ be a vector of hidden values. We are given a vector of observations $y = (y_i)$, where each y_i is an unbiased estimator of some linear combination of the x_i's. This can be succinctly expressed in a matrix form:

$$y = Ax + \delta,$$

where $\delta = (\delta_i)$ is a vector of pairwise independent random variables with mean 0, and $\mathrm{Var}(\delta_i) = \sigma_i^2$. In our case, the x_i's are the true frequencies at the leaves of the binary tree T corresponding to the dyadic structure, the y_i's are the estimators returned by the Count-Sketch at every node of T, and A is a 0-1 matrix that encodes which x_i's are covered by each y_i.

The problem is thus to obtain the best estimates for the x_i's. Once the x_i's are known, the values at internal nodes of T can be computed easily. This is exactly the *ordinary least squares (OLS)* problem, and the *best linear unbiased estimator* for x is the vector x^* such that $\sum_i (y_i - A_i x^*)^2/\sigma_i^2$ is minimized. According to the Gauss-Markov theorem [25], the variance of any linear combination of the x_i^*'s is also minimized, so it exactly serves our purpose, since the rank of any element is the linear combination of some x_i's.

However, one technicality in our case is that some of the σ_i's are 0: for some nodes high in the tree, we record the exact frequencies when this is more space efficient than using a sketch. This means that for some k, we have $\sigma_i = 0$ for $i = 1, \ldots, k$. Thus the modified problem formulation is as follows.

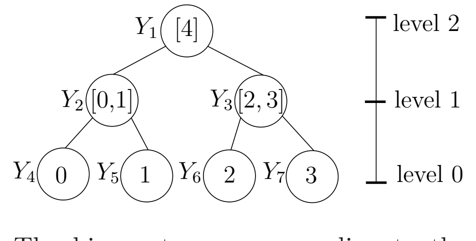


Fig. 2: The binary tree corresponding to the dyadic structure on the universe $\{0, 1, 2, 3\}$.

Definition 1 Given a vector $x = (x_i)$ of unknowns, a vector $y = (y_i)$ of observations, an $m \times n$ matrix A, and an integer k, such that

$$\begin{cases} y_i = A_i x, & i \le k \\ y_i = A_i x + \delta_i, & i > k, \end{cases} \tag{1}$$

where A_i is the i-th row of A. The δ_i's are pairwise independent random variables with mean 0 and variance $\sigma_i^2 > 0$. A vector $x^* = (x_i^*)$ is the *best linear unbiased estimator (BLUE)* for x if $y_i = A_i x^*$ for all $i \le k$ and $\sum_{i>k}(y_i - A_i x^*)^2/\sigma_i^2$ is minimized.

The general method for solving this problem is to first eliminate the y_i's for $i \le k$ by Gauss elimination, then use the method of Lagrange multipliers [19]. However, this would take $O(u^3)$ time as A in our case is a $(2u-1) \times u$ matrix, where u is the universe size! In the rest of this section, we will exploit the special properties of our setting and develop much more efficient algorithms.

3.2.2 Truncating the tree

Our first observation is that we should not work on the entire dyadic structure. Because we can tolerate an error of εn, any interval that has less than εn weight can be safely discarded. More precisely, we extract a truncated binary tree $\hat{T}$ from the DCS as follows. Starting from root we traverse the dyadic structure top-down. For each node (interval), we estimate its frequency from the Count Sketch. If it is larger than εn, we recursively visit its children; otherwise we skip this node as well as its subtree. In Appendix A.1, we show that the size of the truncated tree $\hat{T}$ is only $O(\frac{1}{\varepsilon}\log u)$ in expectation. In our implementation, in order to have better accuracy, we set the truncating threshold to $\eta\varepsilon n$ for some small constant η, and experimentally tune the parameter η to achieve a desired tradeoff between accuracy and cost.

Even after truncating the tree, the cubic running time using the standard OLS method is still too expensive. By exploiting the special properties of the tree structure, Hay et al. [20] designed an algorithm to compute the BLUE in linear time. However, their algorithm can only work on a perfectly balanced tree. In our case, $\hat{T}$ can be very unbalanced, which is especially the case on skewed distributions. Furthermore, their algorithm cannot handle the case where some σ_i's are 0. Below we present our algorithm that resolves these issues.

3.2.3 The algorithm

First, we decompose the tree $\hat{T}$ into subtrees such that the only node with an exact frequency is the root. It is clear that an exact node "shields" the influence of its subtree from other parts of tree, so each subtree can be handled separately.

Let T_r be such a tree with root r. We use $w \prec v$ to denote that w is a leaf in below node v. For a node v, let lpath(v) be the set of nodes on the path from v to the leftmost leaf below v, anc(v) the set of all ancestors of v (including v), and parent(v) the parent node of v. For each node $v \in T_r$, we are given y_v, which is an unbiased estimator of $x_v = \sum_{w \prec v} x_w$, with variance σ_v^2, and we wish to compute the BLUE x_v^* for each v.

An important technique to handle an unbalanced tree is to introduce a *weight* λ_v for each node v. Intuitively, the weight of a node measures how important it is for computing the BLUE for the whole system. When the tree is perfectly balanced, the weights are the same for all the nodes on the same level. But on an unbalanced tree, the weights will depend on both σ_v and the structure of the tree. Consider the example in Figure 3, where the number inside each node is the estimated frequency y_v. Here we assume $\sigma_i^2 = 2$ for all i except that $\sigma_1^2 = 0$. Node 4 and 5 are on the same level and have the same variance. However, node 4 has no children, while node 5 has two children whose sum is another estimate of the true frequency at node 5. Thus, the importance of the estimate at node 5 itself should be discounted relative to that at node 4. This intuition is captured quantitatively by the following equations:

$$\begin{cases} \lambda_v = \sum_{w \prec v} \lambda_w, \text{for all internal nodes } v; \\ \pi_{\text{left child of } v} = \pi_{\text{right child of } v}, \text{for all internal nodes } v, \end{cases} \tag{2}$$

where $\pi_v = \sum_{w \in \text{lpath}(v)} \lambda_w/\sigma_w^2$.

Note that if T_r has τ leaves, hence $\tau - 1$ internal nodes, then there are $2\tau - 1$ weights and (2) has $2\tau - 2$ equations in total. Thus (2) does not uniquely determine the weights but they only differ by scaling. This will not be a problem since the weights will only measure the relative importance of the nodes. For convenience, we add the constraint $\lambda_r = 1$ to make the weights uniquely defined. This way, the weights can be computed by solving the system of linear equations (2), and those for the example in Figure 3 are given in Table 2.

We can use a bottom-up traversal of T_r to efficiently solve (2) in linear time. We start from any node that is just above the leaf level. Let v be such a node with two children leaves w_1 and w_2. We have two linear equations at v involving 3 unknowns $\lambda_v, \lambda_{w_1}, \lambda_{w_2}$. We solve them and obtain two relationships $\lambda_{w_1} = \alpha_{w_1}\lambda_v$, and $\lambda_{w_2} = \alpha_{w_2}\lambda_v$ for some $\alpha_{w_1}, \alpha_{w_2}$. We also have

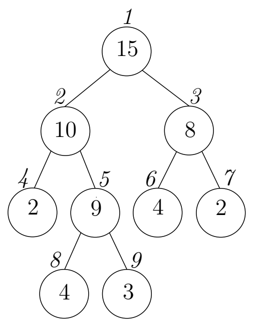


Fig. 3: The binary tree.

node	λ	π	Z	F	x^*
1	1	/	419/62	0	15
2	15/31	12/31	243/62	4.47	8.94
3	16/31	12/31	88/31	3.03	6.06
4	9/31	9/62	54/31	/	1.16
5	6/31	9/62	135/62	8.36	7.77
6	8/31	4/31	48/31	/	4.04
7	8/31	4/31	40/31	/	2.03
8	3/31	3/62	69/62	/	4.38
9	3/31	3/62	33/31	/	3.38

Table 2: Computing the BLUE. $\varDelta = 59/62$.

$\pi_v = \lambda_{w_1}/\sigma^2_{w_1} + \lambda_v/\sigma^2_v = \beta_v\lambda_v$ for some β_v. After this, we mark v as "done" and move on to any other node whose children are both done. In general, when we reach a node v with children u_1 and u_2, we will have inductively obtained $\pi_{u_1} = \beta_{u_1}\lambda_{u_1}, \pi_{u_2} = \beta_{u_2}\lambda_{u_2}$ for some β_{u_1}, β_{u_2}. We can then solve the 2 equations at v involving 3 unknowns $\lambda_v, \lambda_{u_1}, \lambda_{u_2}$ and obtain relationships $\lambda_{u_1} = \alpha_{u_1}\lambda_v, \lambda_{u_2} = \alpha_{w_2}\lambda_v$, as well as $\pi_v = \pi_{u_1} + \lambda_v/\sigma^2_v = \beta_v\lambda_v$, for some $\alpha_{u_1}, \alpha_{u_2}, \beta_v$. After this traversal, we will have discovered the relationship between any λ_v and $\lambda_{\text{parent}(v)}$, and plugging in $\lambda_r = 1$ will yield all the λ_v's and π_v's in linear time.

In order to compute the x^*_i's efficiently, we need some more auxiliary variables. For each leaf $w \in T_r$, let $Z_w = \lambda_w \sum_{z\in\text{anc}(w)\setminus r} y_z/\sigma^2_z$, and for any internal node v, $Z_v = \sum_{w\prec v} \lambda_w Z_w$. For any node v except the root r, let $F_v = \sum_{w\in\text{anc}(v)\setminus\{r\}} x^*_w/\sigma^2_w$. Finally, set $\varDelta = (Z_r - y_r\pi_{r\text{'s left child}})/\lambda_r$. The values of these auxiliary variables for the example in Figure 3 are given in Table 2.

Using these auxiliary variables, we are able to obtain the following equations with respect to the x^*_i's (proof given in Appendix A.2):

$$\begin{cases} x^*_r = y_r, \\ x^*_v = (Z_v - \lambda_v F_{\text{parent}(v)} - \lambda_v \varDelta)/\pi_v, \text{for all nodes } v \neq r. \end{cases} \tag{3}$$

Note that (3) has exactly $2\tau-1$ unknowns and $2\tau-1$ equations. We can then efficiently solve (2) and (3) in three traversals of T_r, as follows:

1. *Top-down traversal to compute a temporary Z':* Initialize $Z'_r = 0$; for any other v, set $Z'_v = Z'_{\text{parent}(v)} + y_v/\sigma^2_v$ recursively. Note that after this traversal, for any leaf w, we have computed $Z_w = \lambda_w Z'_w$.
2. *Bottom-up traversal to compute Z*: With the values of Z_w's at the leaves, we can easily compute all the Z_v's a bottom-up traversal.
3. *Top-Down Traversal to compute F, x^*:* We first compute $\varDelta = (Z_r - y_r\pi_s)/\lambda_r$, and initialize $F_r = 0$, $x^*_r = y_r$. During this top-down traversal, we compute $F_v = F_{\text{parent}(v)} + x^*_v/\sigma^2_v$ for every v. Meanwhile, each x^*_v can be computed from (2).

It is easy to see that the algorithm takes time linear in the size of the truncated tree $\hat{T}$, which is $O(\frac{1}{\varepsilon}\log u)$ in expectation.

3.2.4 Discussion

There are a few issues to discuss with respect to our post-processing algorithm. First, the OLS framework requires pairwise independence among the estimators y_v. This is not completely true in our setting. Two nodes on different levels are clearly independent, as they are returned from independent Count-Sketches. However, two nodes on the same level are not. Note that although the Count-Sketch uses a pairwise independent hash function to distribute elements into counters, the counters themselves are *not* pairwise independent. Nevertheless, we have analyzed the covariance of any two counters, and shown that it is much smaller than the variance (details in Appendix A.3). Therefore, we argue that the OLS framework is still suitable to apply.

The second issue is that our algorithm needs the variance σ^2_v for each v. Conveniently, the Count-Sketch itself actually provides a good estimator for this variance, which is simply the sum of all the counters squared in a row [2]. However, when we use multiple rows and return the median estimator, the variance does not follow so easily. Nevertheless, our algorithm is not affected if all the σ^2_v's are reduced by the same factor, so we use the variance of one row of the sketch as a good empirical approximation.

4 Experiments

4.1 Setup

We implemented all algorithms in C++, compiled with GCC. The executables were tested under Linux 2.6.18

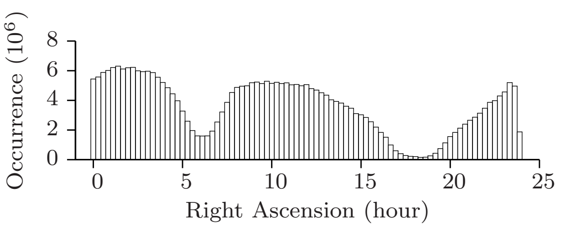


Fig. 4: Distribution of MPCAT-OBS

on a machine with a 3GHz CPU, 6MB CPU cache and 16GB memory.

4.1.1 Data sets

We used 2 real data sets and 12 synthetic data sets in the experiments. The first real data set is the MPCAT-OBS data set, which is an observation archive available from the Minor Planet Center[2]. We used the optical observation records from 1802 to 2012. The records are ordered by the timestamp, and we feed the right ascensions[3] as a stream to the algorithms. The stream values appear to arrive randomly overall, but consist of chunks of ordered data of various lengths. This is because an observatory usually traces a planet continuously in a session, and then moves on to other planets. The right ascension is not uniformly distributed, as shown in Figure 4. This data set contains 87,688,123 records, and the right ascensions are integers ranging from 0 to 8,639,999. The second real data set is the terrain data for the Neuse River Basin[4], which contains LIDAR points measuring the elevation of the terrain. This data set contains about 100 million points.

In order to study how different data characteristics affect the algorithms' performance, we also generated 12 synthetic data sets with different sizes (10^7 to 10^{10}), universe sizes (2^{16} to 2^{32}), distributions (uniform and normal with different variances), and order (random and sorted). Further details are given in context. Note that we know that certain factors do not affect certain algorithms, due to their definition. For example, the universe size and distribution should not affect any comparison-based algorithms; the stream order should not affect (the space and accuracy of) the turnstile algorithms; and the stream length should not affect q-digest and the turnstile algorithms.

[2] `http://www.minorplanetcenter.net/iau/ecs/mpcat-obs/mpcat-obs.html`

[3] Right ascension is an astronomical term used to locate a point (a minor planet in this case) in the equatorial coordinate system.

[4] `http://www.ncfloodmaps.com`

4.1.2 Measures

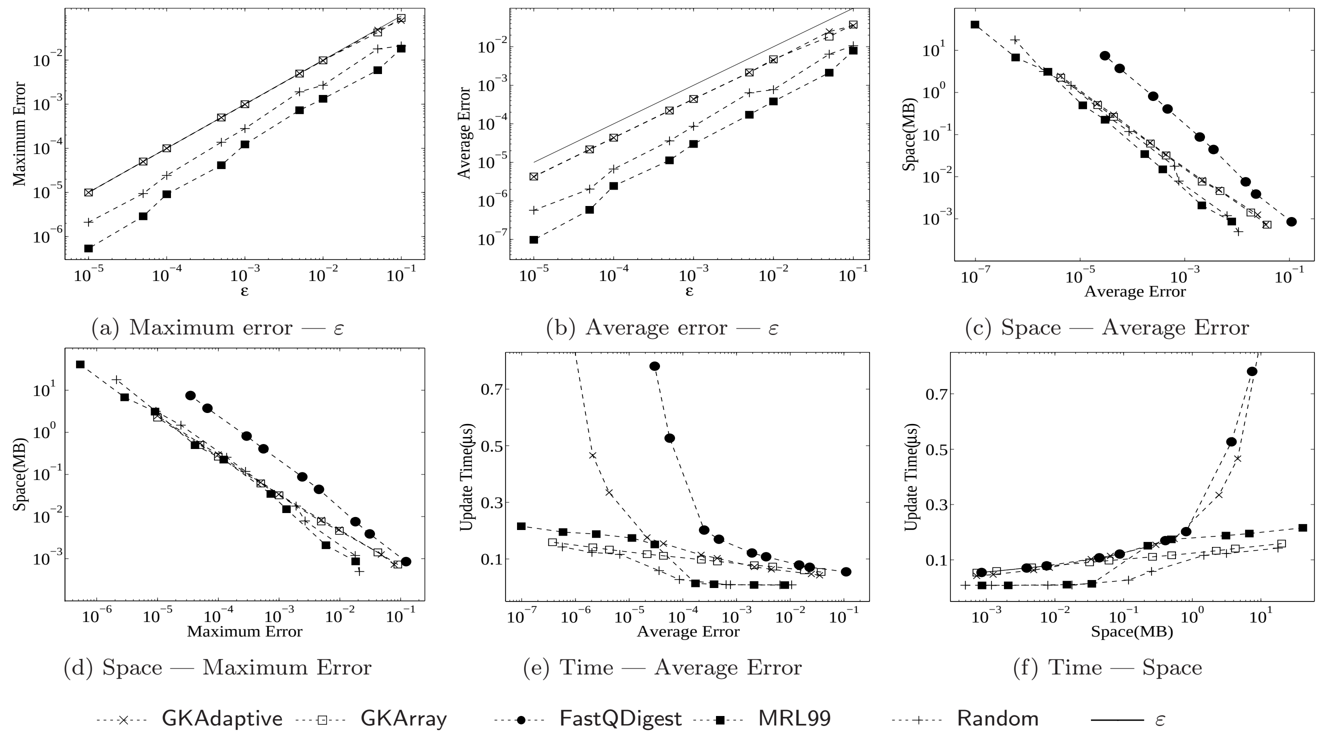
We measure the algorithms along the following dimensions:

Space is one of the most important measures for streaming algorithms. We report space usage in bytes, where every element from the stream, counter, or pointer consumes 4 bytes. When an algorithm uses auxiliary data structures such as a binary tree or a hash table, the space needed by these internally is carefully accounted for. For algorithms whose space usage changes over time, we measured the maximum space usage.

Update time is as important as space, if not more so, as it translates to the throughput of the streaming algorithm. Prior empirical studies have overlooked this issue [15, 21]; more recent works on other streaming problems have included time as a main consideration [6]. In our experiments, we measured the average wall-clock processing time per element in the stream. In some cases, it is important to bound the worst-case time per element, and some algorithms periodically use a slower pruning procedure (e.g. a Compress or merge step). We note that standard de-amortization techniques, such as use of buffering, can be adopted to avoid blocking operations.

Accuracy is the third factor we measure: we want to understand the accuracy-space and accuracy-time tradeoffs. There are some technical subtleties in measuring the error. The error parameter ε used by the algorithms controls the accuracy, but it is not suited for use as the measure of empirical accuracy for two reasons. First, the error analysis usually considers worst-case input and may be loose: the actual error could be substantially better; and second, the deterministic algorithms provide an ε-error guarantee while the randomized ones give such a guarantee only probabilistically, so it is not a fair comparison. Therefore, in our experiments, we measure the observed errors, and used the following two error metrics.

We first extract the ϕ-quantiles for $\phi = \varepsilon, 2\varepsilon, \cdots, (1-\varepsilon)$. For each ϕ-quantile extracted, we compute its true rank from the data, and take its difference from ϕn, divided by n. From all these errors, we take the maximum and average values. The former is exactly the *Kolmogorov-Smirnov divergence* between the true CDF and that of the extracted quantiles, while the latter is determined by the *total variation distance* of the two CDFs, both of which are standard statistical distances between distributions. There is some ambiguity over the rank of elements which appear multiple times in the data. We favor the algorithms, so that the rank of such items is taken as an interval. We compute the error as



(a) Maximum error — ε (b) Average error — ε (c) Space — Average Error

(d) Space — Maximum Error (e) Time — Average Error (f) Time — Space

Fig. 5: Results on MPCAT-OBS

the difference from ϕn to the closer interval endpoint, or 0 if ϕn is contained within the interval.

Thus, in total we make 5 measurements (space, time, ε, actual maximum error, actual average error) for each algorithm in each experiment. For randomized algorithms, we repeat the algorithm 100 times and take the average. For space reasons, we present a selection of most representative results in this paper; the full comparison across all 9 algorithms and 5 measurements over 14 real and synthetic data sets can be explored (anonymously) through an interactive interface at `http://quantiles.github.com`. Below, all results are on the MPCAT-OBS data set unless specified otherwise.

4.2 Results on cash register algorithms

4.2.1 ε vs. actual error

Figures 5a and 5b show how the actual errors of the algorithms deviate from the given ε parameter. All the deterministic algorithms indeed never exceed the ε guarantee, and they usually obtain average error between $\frac{1}{4}\varepsilon$ and $\frac{2}{3}\varepsilon$. The maximum errors of Random and MRL99 are much smaller than ε, and the average errors are even smaller, revealing that their bounds are loose. We subsequently use the observed errors (max and average) as the primary error metric.

4.2.2 Space

Figure 5c and 5d show the error-space tradeoff of the algorithms using the max error and the average error, respectively. We see that MRL99 and Random are the best two algorithms with very similar performance. Between the two, MRL99 looks slightly better. This shows that the detailed choices of MRL99 offer a minor advantage, but not much. GKAdaptive and GKArray come quite close, especially when max error is considered. FastQDigest uses the largest space among all algorithms. Note that $\log u = 24$ in this case; we study other universe sizes subsequently.

4.2.3 Time

Figure 5e shows the tradeoff between error and the update time per element for each algorithm. Here we use log scale on the x-axis but linear scale on the y-axis. We see that for larger errors, all algorithms perform similarly, but GKAdaptive and FastQDigest degrades rapidly for smaller errors. This phenomenon can be better explained by the space-time tradeoff plotted in Figure 5f, in which we see that GKAdaptive and FastQDigest suffer a big speed loss when their space use exceeds 5MB, which is roughly the size of CPU cache. This is because they perform a binary search for each incoming element, which is not cache-friendly. On the other hand, MRL99, Random, and GKArray still perform well, as they only use sorting and merging as their basic operations. Among these three, the two randomized algorithms are better than GKArray on larger errors, since this is where sampling kicks in. On smaller errors, the three algorithms have similar performance, with Random being slightly better than the other two.

4.2.4 Varying universe size and data skewness

From the $O(\frac{1}{\varepsilon} \log u)$ bound, q-digest should work better with a smaller universe size. We tested the algorithms on synthetic data sets following a normal distribution, but with different universe sizes. The length of the stream is fixed at $n = 10^8$, and elements arrive in a random order. In Figures 6a and 6b, we plot the error-space and error-time tradeoffs of FastQDigest on data sets with different $\log u$. We also plot the curves of GKAdaptive and Random, the best deterministic and randomized comparison-based algorithms, which are unaffected by the universe size[5].

From the figures, we see that q-digest is only competitive when $\log u = 16$ and $\varepsilon < 10^{-5}$. However, when this is the case, storing the frequencies of all the u elements exactly only takes 0.25MB space. We also tested on data sets with different skewness by changing the variance of the normal distribution, but did not observe significant changes in the performance of q-digest. Therefore, we do not find any streaming situation where q-digest is the method of choice. Nevertheless, the algorithm remains of importance, since it is the only deterministic mergeable summary for quantiles [1], needed when summaries are merged in an arbitrary fashion.

4.2.5 Varying stream length

We tested the algorithm on streams whose length increases from 10^7 to 10^{10}, and plot how the time and space changes in Figures 7a and 7b. We used uniformly distributed data, with the universe size fixed at $u = 2^{32}$ and $\varepsilon = 10^{-4}$. Elements arrive in a random order. We observe that there is little direct effect on update time or space usage as stream length grows, implying that these algorithms can scale to increasingly large data sets. Indeed, the per-element update time for Random actually *decreases*, due to random sampling playing a more major role as n goes up. The update time of the q-digest also goes down, since the cost of COMPRESS is amortized over more elements, as the algorithm only executes COMPRESS $\log n$ times throughout the whole stream.

Looking at Figure 7b, we see that the space used by GKAdaptive and GKArray is essentially flat; we conjecture that they have a space bound independent of n on randomly ordered data. The space used by Random is constant, because the buffers are pre-allocated according to ε.

4.2.6 Conclusions for cash register algorithms

From our study, we can safely conclude that GKArray and Random are generally the best deterministic and randomized algorithm, respectively. Random is slightly better than GKArray in terms of both space and time, but the latter offers a worst-case guarantee on the error. However, note that we still lack a guarantee on its size as it uses a heuristic to remove tuples. On the other hand, Random uses a fixed amount of space that depends only on ε, and should be used when there is a hard limit on space.

4.3 Results on turnstile algorithms

In this section, we compare the empirical performances of DCM, DCS, and DCS with post processing, which we denote as Post. We exclude the random subset sum sketch, as its performance is much worse than these three.

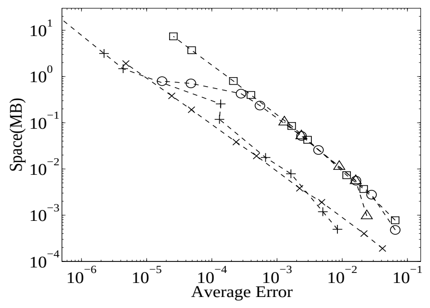
Although we are experimenting with turnstile algorithms, it is not necessary to explicitly include deletions in the data sets: it is clear that the algorithms proceed in exactly the same way as on insertion-only data sets. Deleting a previously inserted element completely removes its impact on the data structure, so it has no effect on the accuracy, either. What matters is only those elements that remain.

4.3.1 Parameter tuning

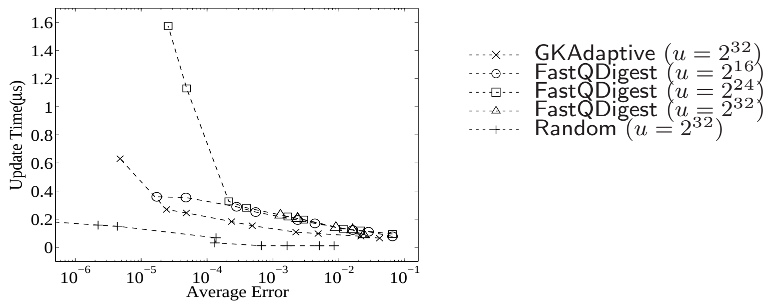
Recall that all the three algorithms use a sketch that is a $w \times d$ array, for each level in the dyadic hierarchy. Theoretically speaking, w determines the error while d determines the confidence of obtaining an estimate within the error bound. In Section 3 we have given their relationships with the commonly used notion of an (ε, δ)-error guarantee. Intuitively, both w and d are meant to reduce the observed errors. So the question is, given a certain total sketch size, what is the best allocation to w and d?

To this end, we first conduct a series of experiments trying out different combinations of w and d. Specifically, for a fixed sketch size, we vary d, which in

[5] It is possible for the error to be affected due to more duplicates in smaller universes, but we found this effect negligible in practice.

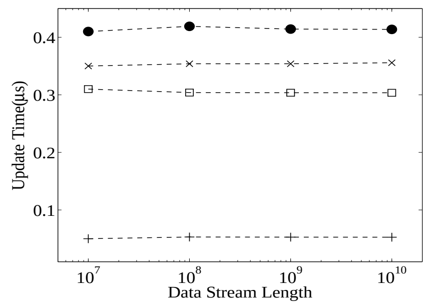


(a) Size — Average Error

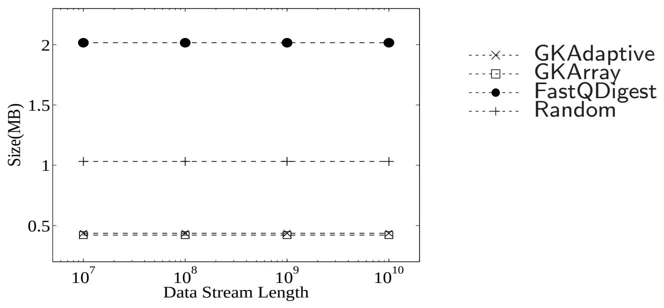


(b) Time — Average Error

Fig. 6: Varying the universe size on normally distributed data, $\sigma = 0.15$

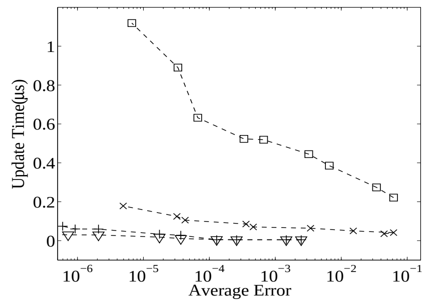


(a) Time — Stream Length

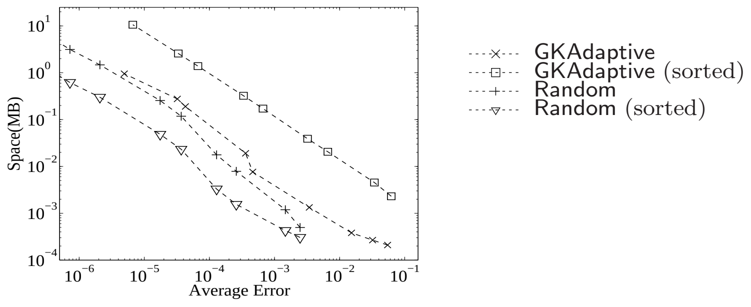


(b) Space — Stream Length

Fig. 7: Varying stream length on uniform distributed data, $u = 2^{32}$ and $\varepsilon = 0.0001$



(a) Time — Average Error



(b) Space — Average Error

Fig. 8: Random order vs sorted — uniform distributed data, $u = 2^{32}$ and $n = 10^8$

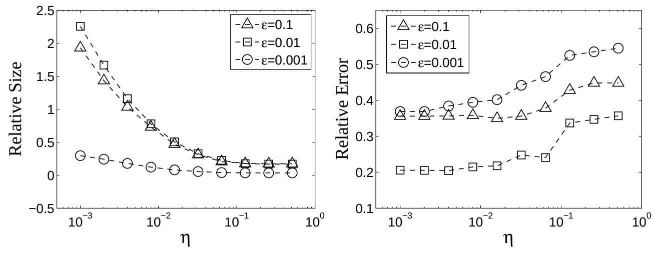
turn determines w, and record the maximum and average errors of the computed quantiles. Here we used a uniformly distributed data set with $n = 10^7$ elements drawn from a universe of size $u = 2^{32}$. In Table 3, we show the average errors ($\times 10^{-4}$) of DCS using a series of sketch sizes, and find out that $d = 7$ appears to be a good choice. Similarly, we did the same for the maximum error in Table 4. We observe that for the maximum error, we generally require a slightly larger d (which makes sense), but still 7 appears to be a good choice. We performed the same study for DCM and vDCS and found that $d = 7$ is the best choice there also. So we set $d = 7$ for all the subsequent experiments. We set $w = \frac{1}{\varepsilon} \log u$ for DCM and $w = \sqrt{\log u}/\varepsilon$ for DCS.

Table 3: Tuning d for average error.

	sketch size (KB)						
d	64	128	256	512	1024	2048	4096
3	10.24	4.307	1.924	**0.826**	**0.425**	0.279	0.134
5	9.558	4.447	2.084	0.933	0.558	0.304	**0.132**
7	**8.947**	**4.198**	**1.851**	1.108	0.621	**0.261**	0.146
9	11.15	5.043	2.287	1.37	0.603	0.373	0.142
11	11.14	5.753	3.055	1.418	0.652	0.363	0.173
13	21.93	5.121	2.642	1.557	0.707	0.355	0.167

Table 4: Tuning d for maximum error.

	sketch size (KB)						
d	64	128	256	512	1024	2048	4096
3	53.67	22.92	9.27	7.71	3.58	2.56	0.931
5	**50.04**	25.11	11.13	8.07	3.383	2.498	0.931
7	65.26	**22.28**	**8.71**	**5.49**	2.923	**1.693**	2.419
9	75.41	27.39	8.87	9.543	**2.63**	2.389	**0.542**
11	61.03	33.32	13.5	8.769	3.067	2.261	0.824
13	139.3	29.25	17.34	7.503	2.843	1.824	0.869



(a) Relative Size - η (b) Relative Error - η

Fig. 9: tradeoff between relative size and error

In Post, there is an additional parameter η which determines the tradeoff between the size of the truncated tree $\hat{T}$ and the accuracy improvement. The tree size in turn determines the running time for the post processing algorithm. We conducted a series of experiments by varying η on different values of $\varepsilon = 0.1, 0.01, 0.001$ on our real data set. Figure 9 reports the size of $\hat{T}$ relative to that of the DCS sketch, as well as the reduced error relative to that of the original DCS sketch before the post processing. From the results we find that $\eta = 0.1$ is a sweet spot; further reducing η increases $|\hat{T}|$ without too much gain in terms of error reduction. We can see that our post-processing algorithm is quite effective, reducing the error to 20–40% of the original DCS sketch. It works better for larger ε, which also makes sense since DCS with a small ε is already quite accurate.

4.3.2 ε vs. actual error

In Figure 10a and 10b, we plot the actual maximum and average errors on the real data for different ε. This shows that the asymptotic analysis is rather loose: The actual maximum error is typically only $\varepsilon/10$, while the average error is even smaller, and Post is quite effective at further reducing the error of DCS.

The actual errors of these three algorithms appear similar, but note that DCM has a larger size than DCS. Looking more closely at the curves, we see that DCM tends to be better in terms of the maximum error, but not as good in terms of average error. This might be due to the fact that the Count-Min sketch gives out biased estimators, while the Count Sketch is unbiased. Subsequently we will use average error as the error metric unless specified otherwise.

4.3.3 Space

Figure 10c shows the error-space tradeoffs of the algorithms. We see that to achieve the same error, DCS require only about 1/10 of the space required by DCM. While using the same amount of space, Post can further reduce the error by 60–80%.

4.3.4 Time

Figure 10d shows the error-time tradeoff. Note that since post processing is only applied at the end of the stream and it is quite efficient, it has negligible impact on the amortized update time of DCS, so its curve is just that of DCS shifted to a smaller error. In the space-time tradeoff Figure 10e, Post is thus identical to DCS, which is also very similar to DCM.

It is also instructive to compare Figure 10c and 10d with Figure 5c and 5e. This shows that the turnstile model in indeed more difficult to deal with than the cash register model. To achieve the same accuracy, the best turnstile algorithm has to spend significantly more space and time (roughly by an order of magnitude) than the best algorithm in the cash register model.

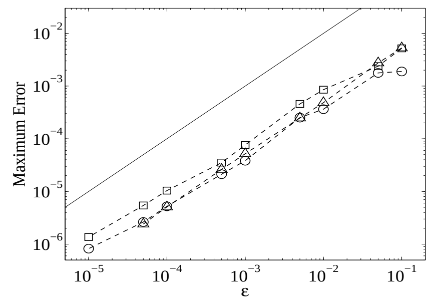
4.3.5 Varying universe size

The universe size u plays an important role in the turnstile algorithms, as it determines the height of the dyadic hierarchy. We tested the algorithms with data sets generated according to a normal distribution with $\sigma = 0.15$, but on different universe sizes. Figure 11a shows two series of trade-offs between error and space: one is on $u = 2^{16}$, and the other is on $u = 2^{32}$. Clearly, we see that a smaller universe indeed makes the algorithms more accurate, or equivalently speaking, makes the data structures smaller. The $u = 2^{16}$ curves halt at a small error value, since at this point the algorithms have sufficient space to store all frequencies exactly.

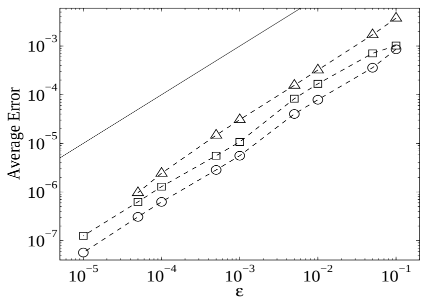
Similarly, Figure 11b shows two series of trade-offs between error and update time for different universe sizes. Again, a small universe makes the algorithms much faster.

4.3.6 Varying data skewness

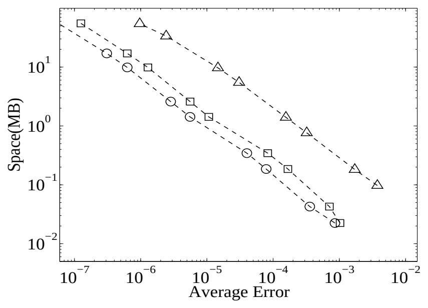
Finally, we tested the algorithms on data sets with different levels of skewness. We used data generated by a normal distribution with $\sigma = 0.05$ and 0.25. Data



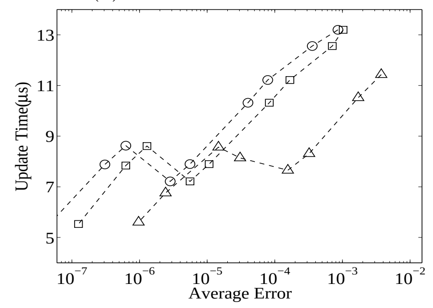
(a) Maximum error — ε



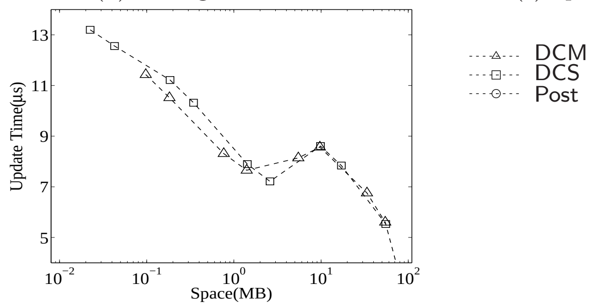
(b) Average error — ε



(c) Space — Average Error

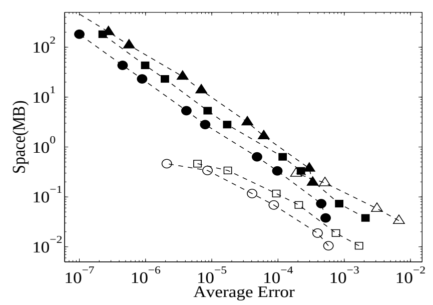


(d) Time — Average Error

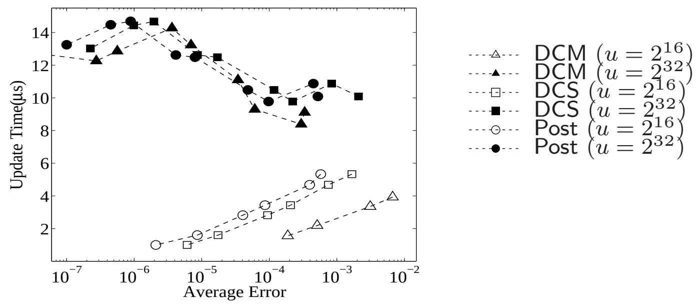


(e) Time — Space

Fig. 10: Results on MPCAT-OBS



(a) Space — Average Error



(b) Time — Average Error

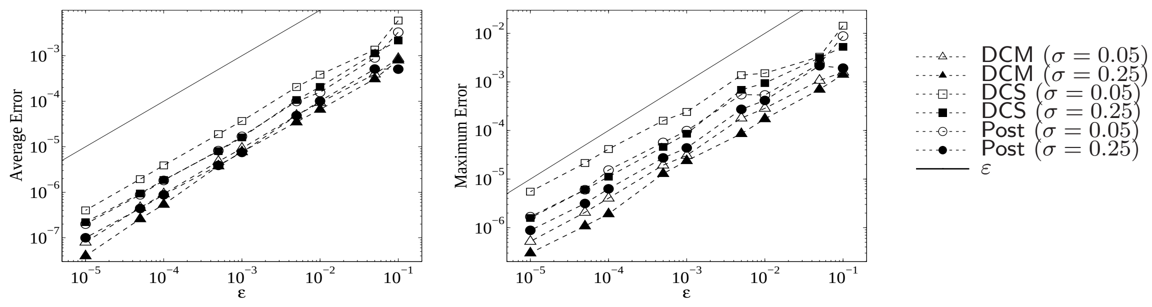
Fig. 11: Varying the universe size on the Normal distributed data, $\sigma = 0.15$

skewness does not obviously affect space or time (for a given ε), so we only show how the actual errors respond, in Figures 12a and 12b. From the figures, we see that as the data gets less skewed, the accuracy improves for all three algorithms. The improvement for DCM is very small, but it is more prominent for DCS and consequently for Post. This again is predicted well by the theory: Although in this paper, we analyzed the error of the Count Sketch in terms of n in order to get the theoretical bound, its error actually depends more closely on the second frequency moment of the data, F_2 [5]. As the variance decreases, F_2 decreases, and the Count Sketch gets more accurate. On the other hand, the Count-Min sketch does not depend directly on F_2.

4.3.7 Conclusions for turnstile algorithms

From the experiments, it should be clear that DCS is the preferred turnstile algorithm for computing quantiles. DCM uses a much larger amount of space than DCS. The running time of the two algorithms are similar. Finally, the post-processing algorithm is always beneficial to DCS, incurring no more space and time (during streaming) while being quite effective at further reducing the error.

Acknowledgements The work of GC is supported in part by European Research Council grant ERC-2014-CoG 647557, the Yahoo Faculty Research and Engagement Program and a Royal Society Wolfson Research Merit Award.



(a) ϵ — Average Error (b) ϵ — Maximum Error

Fig. 12: Varying the deviation on the normal distributed data, $\sigma = 0.05, 0.25$

References

1. Agarwal PK, Cormode G, Huang Z, Phillips JM, Wei Z, Yi K (2013) Mergeable summaries. ACM Transactions on Database Systems 38
2. Alon N, Matias Y, Szegedy M (1999) The space complexity of approximating the frequency moments. Journal of Computer and System Sciences 58(1):137–147, DOI DOI:10.1006/jcss.1997.1545, URL `http://www.sciencedirect.com/science/article/B6WJ0-45JCBTJ-D/2/2a71f12f1f0112bc83447b9d48eba529`
3. Arasu A, Manku G (2004) Approximate counts and quantiles over sliding windows. In: Proc. ACM Symposium on Principles of Database Systems
4. Blum M, Floyd RW, Pratt V, Rievest RL, Tarjan RE (1973) Time bounds for selection. Journal of Computer and System Sciences 7:448–461
5. Charikar M, Chen K, Farach-Colton M (2002) Finding frequent items in data streams. In: Proc. International Colloquium on Automata, Languages, and Programming
6. Cormode G, Hadjieleftheriou M (2008) Finding frequent items in data streams. In: Proc. International Conference on Very Large Data Bases
7. Cormode G, Muthukrishnan S (2005) An improved data stream summary: The count-min sketch and its applications. Journal of Algorithms 55(1):58–75
8. Cormode G, Korn F, Muthukrishnan S, Johnson T, Spatscheck O, Srivastava D (2004) Holistic UDAFs at streaming speeds. In: Proc. ACM SIGMOD International Conference on Management of Data, pp 35–46
9. Cormode G, Garofalakis M, Muthukrishnan S, Rastogi R (2005) Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In: Proc. ACM SIGMOD International Conference on Management of Data
10. Cormode G, Korn F, Muthukrishnan S, Srivastava D (2006) Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In: Proc. ACM Symposium on Principles of Database Systems
11. Felber D, Ostrovsky R (2015) A randomized online quantile summary in O($1/\varepsilon$ log $1/\varepsilon$) words. CoRR abs/1503.01156, URL `http://arxiv.org/abs/1503.01156`
12. Ganguly S, Majumder A (2007) CR-precis: A deterministic summary structure for update data streams. In: ESCAPE
13. Gilbert AC, Kotidis Y, Muthukrishnan S, Strauss MJ (2002) How to summarize the universe: Dynamic maintenance of quantiles. In: Proc. International Conference on Very Large Data Bases
14. Govindaraju NK, Raghuvanshi N, Manocha D (2005) Fast and approximate stream mining of quantiles and frequencies using graphics processors. In: Proc. ACM SIGMOD International Conference on Management of Data
15. Greenwald M, Khanna S (2001) Space-efficient online computation of quantile summaries. In: Proc. ACM SIGMOD International Conference on Management of Data
16. Greenwald M, Khanna S (2004) Power conserving computation of order-statistics over sensor networks. In: Proc. ACM Symposium on Principles of Database Systems
17. Huang Z, Wang L, Yi K, Liu Y (2011) Sampling based algorithms for quantile computation in sensor networks. In: Proc. ACM SIGMOD International Conference on Management of Data
18. Hung RYS, Ting HF (2010) An $\Omega(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon})$ space lower bound for finding ε-approximate quantiles in a data stream. In: FAW
19. Lagrange JL (1853) Mécanique analytique, vol 1. Mallet-Bachelier
20. Li C, Hay M, Rastogi V, Miklau G, McGregor A (2009) Optimizing histogram queries under differential privacy. CoRR abs/0912.4742, URL `http:`

//arxiv.org/abs/0912.4742

21. Manku GS, Rajagopalan S, Lindsay BG (1998) Approximate medians and other quantiles in one pass and with limited memory. In: Proc. ACM SIGMOD International Conference on Management of Data
22. Manku GS, Rajagopalan S, Lindsay BG (1999) Random sampling techniques for space efficient online computation of order statistics of large datasets. In: Proc. ACM SIGMOD International Conference on Management of Data
23. Munro JI, Paterson MS (1980) Selection and sorting with limited storage. Theoretical Computer Science 12:315–323
24. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: Parallel analysis with sawzall. Dynamic Grids and Worldwide Computing 13(4):277–298
25. Rao CR (2009) Linear statistical inference and its applications, vol 22. John Wiley & Sons
26. Shrivastava N, Buragohain C, Agrawal D, Suri S (2004) Medians and beyond: New aggregation techniques for sensor networks. In: Proc. ACM SenSys
27. Suri S, Toth C, Zhou Y (2006) Range counting over multidimensional data streams. Discrete and Computational Geometry
28. Vapnik VN, Chervonenkis AY (1971) On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability and its Applications 16:264–280
29. Wang L, Luo G, Yi K, Cormode G (2013) Quantiles over data streams: An experimental study. In: Proc. ACM SIGMOD International Conference on Management of Data
30. Yi K, Zhang Q (2013) Optimal tracking of distributed heavy hitters and quantiles. Algorithmica 65(1):206–223

A Additional lemmas and proofs

A.1 Size of the truncated tree $\hat{T}$

Lemma 1 *The truncated tree $\hat{T}$ has size $O(\frac{1}{\varepsilon}\log u)$ in expectation.*

Proof Recall that only nodes whose estimated frequency is above $\eta\varepsilon n$ are added to the truncated tree $\hat{T}$, where η is a constant. We classify these nodes into *heavy* nodes and *non-heavy* nodes. A node is heavy if its true frequency is above $\frac{1}{2}\eta\varepsilon n$, and non-heavy otherwise.

We first observe that, on any level of the dyadic structure, there are at most $\frac{n}{\frac{1}{2}\eta\varepsilon n} = O(1/\varepsilon)$ heavy nodes, treating η as a (small) constant. So even if they are all added to $\hat{T}$, there are only $O(\frac{1}{\varepsilon}\log u)$ of them in total.

For a non-heavy node v, the Count-Sketch may overestimate its frequency to above $\eta\varepsilon n$ with a constant probability, say 1/4. Note that there can be $\Theta(u)$ non-heavy nodes being overestimated in expectation, but we will argue below that only $O(\frac{1}{\varepsilon}\log u)$ will be added during the top-down construction of $\hat{T}$.

Note that for a non-heavy node to be added, its parent must be a heavy node or another non-heavy node that has been overestimated. Thus, all the non-heavy nodes in $\hat{T}$ make up a number of subtrees, where the root of each subtree must be a child of a heavy node. Let $\mathbf{t}$ be any such non-heavy subtree, and we will bound $\mathrm{E}[|\mathbf{t}|]$. Let r be the root of $\mathbf{t}$. For any node v below r, let I_v be an indicator variable where $I_v = 1$ if $v \in \mathbf{t}$ and $I_v = 0$ otherwise. Let $d(v)$ be the depth of v in $\mathbf{t}$, and we define $d(r) = 0$. For v to be added to $\mathbf{t}$, all of its $d(v)$ ancestors must have been added, so $\mathrm{E}[I_v] \le (1/4)^{d(v)+1}$ since each level uses an independent Count-Sketch. We then have

$$\begin{aligned} E[|\mathbf{t}|] &= \sum_{v \text{ below } r} E[I_v] \\ &= \sum_{v \text{ below } r} (1/4)^{d(v)+1} \\ &\le \sum_{d=0}^{\log u} 2^d (1/4)^{d+1} \le 1. \end{aligned}$$

Finally, we observe that at most two such $\mathbf{t}$'s can be attached to a heavy node, and there are only $O(\frac{1}{\varepsilon}\log u)$ heavy nodes, so we conclude that there are $O(\frac{1}{\varepsilon}\log u)$ non-heavy nodes in $\hat{T}$ in expectation.

A.2 Constraints on the x_i^*'s

Lemma 2 *Let x^* be the BLUE of (1). Let λ_v be any solution to (2). For any leaf w, let $Z_w = \lambda_w \sum_{z \in anc(w)\setminus r} y_z/\sigma_z^2$, and for any internal node v, $Z_v = \sum_{w \prec v} \lambda_w Z_w$. For any node v except the root r, let $F_v = \sum_{w \in anc(v)\setminus\{r\}} x_w^*/\sigma_w^2$. Let $\Delta = (Z_r - y_r \pi_{r\text{'s left child}})/\lambda_r$. Then we have*

$$\begin{cases} x_r^* = y_r, \\ x_v^* = (Z_v - \lambda_v F_{parent(v)} - \lambda_v \Delta)/\pi_v, \text{for all nodes } v \ne r. \end{cases}$$

Proof We will follow the method of Lagrange multiplier [19] to find the BLUE of (1). Since only the subtree root is known exactly, we introduce a single Lagrange multiplier η. We set $\sigma_r^2 = 1/\eta$ instead of 0, and will later take the limit as η goes to ∞. Denote by $\text{diag}(1/\sigma_v)$ the diagonal matrix with $1/\sigma_v$ at entry (v, v). We further define $Z = \text{diag}(1/\sigma_v)y$ and $U = \text{diag}(1/\sigma_v)A$. Then the Lagrange function can be rewritten as $(Z - Ux^*)^T(Z - Ux^*)$. By differentiation, we can derive (i) $y_r = x^*_r$; and (ii)

$$U^TUx^* = U^TZ. \tag{4}$$

This is sufficient to define a solution; we can solve for x^* by premultiplying by $(U^TU)^{-1}$, at the cost of a computing a matrix inverse. In the following, to derive the equations stated in the lemma from (4), which leads to a much more efficient algorithm to compute x^*.

Let $\text{anc}(u, v) = \text{anc}(u) \cap \text{anc}(v)$. Let u be any node of T_r. We also use $[\tau]$ denote the τ leaves of T_r. Then by simple calculation, we can see that $(U^TU)_{u,w} = \sum_{v\in\text{anc}(u,w)} \sigma_v^{-2}$, and $(U^TZ)_u = \sum_{v\in\text{anc}(u)} y_v/\sigma_v^2$.

First, we take the weighted sum of corresponding rows on the LHS of (4) to obtain $\sum_{u\prec v} \lambda_u(U^TU)_u x^* =$

$$\begin{aligned}
&\sum_{u\prec v}\sum_{z\in[\tau]}\left(\sum_{w\in\text{anc}(u,z)\setminus\text{anc}(v)}\frac{\lambda_u x^*_z}{\sigma_w^2}+\sum_{w\in\text{anc}(u,z)\cap\text{anc}(v)}\frac{\lambda_u x^*_z}{\sigma_w^2}\right)\\
&=\sum_{u\prec v}\sum_{z\prec v}\sum_{w\in\text{anc}(u,z)\setminus\text{anc}(v)}\frac{\lambda_u x^*_z}{\sigma_w^2}+\sum_{w\in\text{anc}(v)}\sum_{u\prec v}\sum_{z\prec w}\frac{\lambda_u x^*_z}{\sigma_w^2}\\
&=\sum_{u\prec v}\sum_{w\in\text{anc}(u)\setminus\text{anc}(v)}\frac{\lambda_u}{\sigma_w^2}\sum_{z\prec w}x^*_z+\sum_{w\in\text{anc}(v)}\sum_{u\prec v}\frac{x^*_w}{\sigma_w^2}\lambda_u\\
&=\sum_{u\prec v}\sum_{w\in\text{anc}(u)\setminus\text{anc}(v)}\lambda_u x^*_w/\sigma_w^2+\lambda_v\sum_{w\in\text{anc}(v)}x^*_w/\sigma_w^2.
\end{aligned} \tag{5}$$

Note that in the last line of (5), the second component can be written as

$$\lambda_v\sum_{w\in\text{anc}(v)}\frac{x^*_w}{\sigma_w^2}=\lambda_v(F_{\text{parent}(v)}+\frac{x^*_v}{\sigma_v^2}+\frac{x^*_r}{\sigma_r^2}).$$

We can also derive that the first component is

$$\sum_{u\prec v}\sum_{w\in\text{anc}(u)\setminus\text{anc}(v)}\frac{\lambda_u x^*_w}{\sigma_w^2}=(\pi_v-\frac{\lambda_v}{\sigma_v^2})x^*_v.$$

To see this, let us assume that this holds for any descendant of v. Then we can derive $\sum_{u\prec v}\sum_{w\in\text{anc}(u)\setminus\text{anc}(v)}\frac{\lambda_u x^*_w}{\sigma_w^2}$

$$\begin{aligned}
&=\sum_{\{s\text{ is a child of }v\}}\sum_{u\prec s}\left(\frac{\lambda_u x^*_s}{\sigma_s^2}+\sum_{w\in\text{anc}(u)\setminus\text{anc}(s)}\frac{\lambda_u x^*_w}{\sigma_w^2}\right)\\
&=\sum_{\{s\text{ is a child of }v\}}\left(\frac{\lambda_s x^*_s}{\sigma_s^2}+\sum_{u\prec s}\sum_{w\in\text{anc}(u)\setminus\text{anc}(s)}\frac{\lambda_u x^*_w}{\sigma_w^2}\right)\\
&=\sum_{\{s\text{ is a child of }v\}}(\frac{\lambda_s x^*_s}{\sigma_s^2}+(\pi_s-\frac{\lambda_s}{\sigma_s^2})x^*_s)\\
&=\sum_{\{s\text{ is a child of }v\}}\pi_s x^*_s=\pi_s x^*_v=(\pi_v-\frac{\lambda_v}{\sigma_v^2})x^*_v.
\end{aligned}$$

Combining the above two results, we have

$$\sum_{u\prec v}\lambda_u(U^TU)_u x^*=\pi_v x^*_v+\lambda_v(\frac{x^*_r}{\sigma_r^2}+F_{\text{parent}(v)}). \tag{6}$$

Secondly, we take the weighted sum of corresponding rows on the RHS of (4) to obtain

$$\begin{aligned}
&\sum_{u\prec v}\lambda_u(U^TZ)_u=\sum_{u\prec v}\lambda_u\sum_{w\in\text{anc}(u)}\frac{y_w}{\sigma_w^2}\\
&=\sum_{u\prec v}\lambda_u(Z_u+\frac{y_r}{\sigma_r^2})=Z_v+\frac{\lambda_v y_r}{\sigma_r^2}.
\end{aligned} \tag{7}$$

Finally, by combing (6) and (7), we derive that $\forall v$,

$$\pi_v x^*_v=Z_v-\lambda_v F_{\text{parent}(v)}-\lambda_v(x^*_r-y_r)\eta. \tag{8}$$

Substituting v by r in (8), we can derive

$$x^*_r=\frac{(\frac{Z_r}{\eta}+y_r\lambda_r)}{(\frac{\pi_r}{\eta}+\lambda_r)}.$$

We already have $x^*_r = y_r$. Then we can conclude that either $\eta = +\infty$ or $y_r\pi_r = Z_r$. As we know, y_r is given and irrelevant to $\pi_r Z_r$ which implies $\eta = +\infty$. To handle this infinity, we first express

$$\Delta(\eta)=(x^*_r-y_r)\eta=(Z_r-y_r\pi_s)(\pi_r/\eta+\lambda_r),$$

where s is a child of root r. Now we take limit of $\Delta(\eta)$ and derive that

$$\Delta=\lim_{\eta\to+\infty}\Delta(\eta)=(Z_r-y_r\pi_s)/\lambda_r.$$

Finally by taking limit on (8) for any $v \neq r$, we derive $x^*_v = (Z_v - \lambda_v F_{\text{parent}(v)} - \lambda_v\Delta)/\pi_v$, and the lemma is proved.

A.3 Covariance analysis

Lemma 3 *Suppose we build a Count-Sketch with $d = 1$ row and w columns on a vector x. For any two elements $u \neq v$, let y_u and y_v be the estimators for x_u and x_v. Then $\text{Cov}(y_u, y_v) = x_u x_v/w$.*

Proof Recall that in the Count-Sketch, for any element v, the estimator is computed as $y_v = g(v)\sum_z g(z)x_z I_v(z)$, where $I_v(z) = 1$ if $h(v) = h(z)$ and 0 otherwise. So we have

$$\begin{aligned}
&\text{Cov}(y_u,y_v)=\text{E}(y_u y_v)-\text{E}(y_u)E(y_v)\\
&=\text{E}\left[\left(g(u)\sum_{h(i)=u}g(i)x_i\right)\left(g(v)\sum_{h(j)=v}g(j)x_j\right)\right]-x_u x_v\\
&=\sum_{i,j}\text{E}[g(u)g(v)g(i)g(j)]x_i x_j\text{E}[I_u(i)I_v(j)]-x_u x_v
\end{aligned}$$

Let $f(i, j) = \text{E}[g(u)g(v)g(i)g(j)]x_i x_j \text{E}[I_u(i)I_v(j)]$. We find that $f(u, v) = x_u x_v$ and $f(v, u) = x_u x_v/w$ (note, the last term in $f(i, j)$ is *not* symmetric in i and j). If $\{i, j\} \neq \{u, v\}$, then $f(i, j) = 0$, since $g(\cdot)$ is 4-wise independent hash function. Therefore, we derive that $\text{Cov}(y_u, y_v) = x_u x_v/w$.

On the other hand, prior analysis on the Count-Sketch shows that $\text{Var}(y_v) = \frac{1}{w}\sum_i x_i^2$. Thus, the covariance is usually order-of-magnitude smaller than the variance.