Quantiles over Data Streams: An Experimental Study

Lu Wang[1] Ge Luo[1] Ke Yi[1,2] Graham Cormode[3]

[1]Department of Computer Science and Engineering, HKUST, Hong Kong, China
[2]School of Software and TNLIST, Tsinghua University, Beijing, China
[3]AT&T Labs – Research, Florham Park, NJ, USA

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 described incrementally, and we must compute the quantiles in an online, streaming fashion. Yet while such algorithms have proved to be tremendously 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 and analyze variations that have not been explicitly studied before, yet which turn out to perform the best. To illustrate this, we provide detailed experimental comparisons demonstrating the tradeoffs between space, time, and accuracy for quantile computation.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

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 normal, 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 measure 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 computation is one of the seven basic operators defined (the others including sum, max, top-k, and count-distinct) [21]. The quantiles also play an important role in network health monitoring for Internet service providers [7] and data collection in wireless sensor networks [17].

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 [3]. In the past 30 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 limited memory to work with. There have been numerous algorithms proposed in this setting, using a variety of different techniques and offering different performance guarantees [9–11, 13, 16, 18–20, 23]. In addition, there have been many studies on variations and extensions of the problem, such as computing quantiles over sliding windows [2], over distributed data [1, 14, 15, 22], continuous monitoring of quantiles [5, 25], biased quantiles [8], computing quantiles using GPUs [12], 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 [20] showed that any algorithm that computes the median 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

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

semantics has been adopted by most prior work, and so do we in this paper, because it 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 at any time for the data seen so far.

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 primarily address the latter issue, and carry out an extensive experimental comparison of various quantile algorithms that have not been compared before, on a full range of performance features and data sets. However, our contribution goes beyond a straightforward implementation study. By carefully examining existing algorithms, we provide new variants in both the cash register and the turnstile model, which turn out to perform the best. We also give theoretical analyses on these new variants to complement the empirical results.

1.1 Classification of algorithms

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

1. 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. In the *cash register* model, only insertions are allowed.

2. 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. Any comparison-based algorithm clearly also works in the fixed universe model, but not vice-versa. Such algorithms can also handle elements that cannot be easily mapped to a fixed universe $[u]$, such as variable-length strings, user-defined types, etc[1].

3. Finally, an algorithm can be *deterministic* or *randomized*. 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 $\phi = \varepsilon, 2\varepsilon, \ldots, 1-\varepsilon$. 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 (often resulting from the use of a union bound), it suffices to set the success probability to a reasonable constant.

[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.

1.2 Existing quantile algorithms and our new findings

1.2.1 The cash register model

In their pioneering paper [20], 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. [18], who also proposed another algorithm that is empirically better, though it has the same worst-case space bound. In 2001, Greenwald and Khanna [13] 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, 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. [18]. All these algorithms are deterministic and comparison-based. Hung and Ting [16] 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. [22] 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 [22] did not include an experimental comparison with the GK algorithm.

Randomized algorithms have also been investigated. Classic results [24] 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 [18] 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. Manku et al. [19] proposed a randomized algorithm 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 larger or smaller than the $\log(\varepsilon n)$ factor of GK, but these two algorithms have not been compared experimentally. Recently, Agarwal et al. [1] gave a more complicated algorithm with a space complexity of $O(\frac{1}{\varepsilon}\log^{1.5}(\frac{1}{\varepsilon}))$

In this paper, we empirically compare the GK algorithm, q-digest, and the randomized algorithm of Manku et al. [19]. We omit results for the algorithms of Munro and Paterson [20] and Manku et al. [18], since they have previously been demonstrated to be outperformed by the GK algorithm. Our experimental study reveals that the randomized algorithm of Manku et al. [19] generally performs the best, but it suffers from the following undesirabilities. 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

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

Algorithm	Space	Update time	Randomization	Model
GKAdaptive	—	$O(\log \text{Space})$	Deterministic	Comparison
GKMixed	$O\left(\frac{1}{\varepsilon}\log(\varepsilon n)\right)$	$O\left(\log\frac{1}{\varepsilon}+\log\log(\varepsilon n)\right)$	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
vDCS *	—	$O(\log u\log(\frac{\log u}{\varepsilon}))$	Randomized	Fixed universe

the algorithm is difficult to analyze, the analysis given in [19] 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 these algorithms were not actually needed, and the algorithm can be significantly simplified without affecting its performance. In addition, we are able to give an improved $O(\frac{1}{\varepsilon}\log^{1.5}(\frac{1}{\varepsilon}))$ bound for this new, simpler algorithm, which we refer to as Random. This matches the best theoretical bound (among those that depend only on ε) for this problem obtained in [1].

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 this 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 [10]. The high dependency on $\frac{1}{\varepsilon}$ and $\log u$ mean that this is not considered practical.

Existing algorithms in the turnstile model generally 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 [4, 9]. Each level keeps a frequency estimation sketch that can be used to estimate the total number of elements in any interval. 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. [11] first proposed the *random subset sum* sketch for this purpose, which results in an overall 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, reducing the overall size to $O(\frac{1}{\varepsilon}\log^2 u\log(\frac{\log u}{\varepsilon}))$ [9]. This remains the best bound in the turnstile model. In this paper, we propose to use the Count-Sketch [4], and give a new analysis showing that it further reduces the space to $O(\frac{1}{\varepsilon}\log^{1.5} u\log^{1.5}(\frac{\log u}{\varepsilon}))$. We also carry out an experimental comparison of these different variants, which shows that the new variant using the Count-Sketch is not only theoretically the best, but also gives superior performance in practice.

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

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 GK algorithm

The GK algorithm [13] 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 $v_i \le v_{i+1}$. The g_i's and Δ_i's are integers satisfying the following conditions:

(1) $\sum_{j\le i} g_j \le r(v_i)+1 \le \sum_{j\le i} g_j + \Delta_i$;

(2) $g_i + \Delta_i \le \lfloor 2\varepsilon n\rfloor$.

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 to ϕn. Thus, to extract the ϕ-quantile, we can find the smallest i such that $\sum_{j\le i} g_j + \Delta_i > 1 + \lceil \phi N\rceil + \max_i(g_i+\Delta_i)/2$, 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 $g_i + g_{i+1} + \Delta_{i+1} \le \lfloor 2\varepsilon n\rfloor$.

In order to keep $|L|$ small, the original paper [13] 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 $O(1)$. An insertion can be done in time $O(\log|L|)$, 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 [13], 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.

The original paper [13] did not specify how to find a removable tuple, as they did not focus on running time. In our implementation, we maintain the tuples in L in a min-heap ordered by $g_i+g_{i+1}+\Delta_{i+1}$. When a new tuple is inserted, we first check if the newly inserted tuple is removable, and remove it immediately if so. Otherwise, we check the top element in the heap, and remove it if it is removable. If the top element in the heap is not removable, then no others are. The heap can be maintained in $O(\log|L|)$ time per element, so the update time is not affected. Note that $|L|$ will remain the same after inserting v if one removable tuple is found, otherwise $|L|$ is increased by 1. We refer to this variant as GKAdaptive.

In this variant, COMPRESS is never called (in fact it is not implemented as such). So it is not clear if the space upper bound of $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ still holds.

2.1.2 *Variant:* GKMixed

Curious about whether the more careful COMPRESS procedure leads to smaller size of L, we implemented another variant that adheres more closely to the original analysis. In this variant, called GKMixed, after inserting an element, we first check if it is removable. If so, we remove it right away. Then, whenever $|L|$ doubles, we call COMPRESS to remove tuples. For this variant, the $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ space bound still holds.

2.2 q-digest

The q-digest [22] designed by Shrivastava et. al. was initially introduced as an algorithm for computing quantiles in a (distributed) sensor environment. It also applies to the streaming model. The algorithm is deterministic and assumes a fixed universe $[u]$.

The q-digest also makes use of the dyadic structure of the universe $[u]$ described previously. This structure naturally corresponds to a complete binary tree with u leaves. Each leaf corresponds to an integer in $[u]$, while each node corresponds to a dyadic interval. We do not distinguish a node and its corresponding dyadic interval. Each node v is associated with a counter c_v, representing c_v elements from the steam in the dyadic interval of v. Initially all counters are 0.

We use $lc(v)$ and $rc(v)$ to denote respectively the left child and right child of v. The following two invariants are maintained for any *internal* node v:

(1) $c_v \le \varepsilon n/\log u$;

(2) $c_v + c_{lc(v)} + c_{rc(v)} > \frac{\varepsilon n}{\log u}$ if $c_{lc(v)}+c_{rc(v)}>0$.

The first condition above ensures the accuracy of quantiles, and the second ensures that there are at most $O(\frac{1}{\varepsilon}\log u)$ nodes with non-zero counters. We say a node v is empty if $c_v=0$, and it is full if $c_v=\varepsilon n/\log u$. Note that a full node may become non-full as n increases. As most tree nodes are empty, we only store the non-empty nodes, and denote by Q the set of non-empty nodes.

To extract the ϕ-quantile, we sort all nodes (dyadic intervals) in Q based on the left endpoints of the intervals, breaking ties by putting smaller intervals first. Then we find the first node v such that the sum of counters of v and all nodes before v are greater than ϕn. Finally we return the right endpoint of v. It can be verified that this is an ε-approximate ϕ-quantile due to condition (1) above.

To merge two q-digests, we first add up the corresponding counters, and then carry out a COMPRESS procedure [22] to make sure that condition (2) above is maintained. To adapt it to the streaming model, for each element in the stream, we simply increment the counter of its corresponding leaf, without enforcing condition (2). Then, we carry out the COMPRESS procedure whenever $|Q|$ doubles. Since COMPRESS takes $O(|Q|\log u)$ time [22], the amortized update cost per element is $O(\log u)$.

We see in our experiments that the $O(\log u)$ update time translates to a high cost in practice. This can be explained by observing that each element begins as a leaf in the structure, but the COMPRESS operation moves it up within the tree structure one step at a time until it comes to rest. Based on this observation we designed a variant called FastQDigest, following the discussion in [8]. The idea is to insert each new update directly in the tree-structure where it would reside following a COMPRESS, without the lengthy search. We start with an empty Q. To insert an element x, we find the lowest ancestor of x in Q, say v (if Q is empty, we choose the root as v). If increasing c_v by 1 does not violate condition (1), we do so. Otherwise, we find the child of v that is also the ancestor of x (or x itself), set its counter to 1, and add it to Q. Finally, we call COMPRESS whenever n doubles. Since the accuracy of the algorithm only depends on condition (1), the correctness is still ensured.

Now, we observe that the set of non-empty nodes Q in this variant always form a connected subtree rooted at the root of the dyadic tree (which is not the case in the original version). This means that to find the lowest ancestor of x in Q, we can do a binary search on the path from x to the root. This can be done in $O(\log\log u)$ time, by storing Q in a hash table. But in the experiments we observe substantial overhead with the hash table, so we used a simpler version where we replaced the hash table with a binary search tree (using `std::map`). We observe that the lowest ancestor of x in Q is exactly the innermost dyadic interval containing x. So we can store all nodes (dyadic intervals) of Q in the binary search tree, ordered by their left endpoints, breaking ties by putting longer intervals first. Now the innermost interval containing x is found by simply locating the predecessor of x, which is efficiently supported by this data structure.

COMPRESS is also more efficient in this variant, in time proportional to $|Q|$. Recall that all nodes in Q form a connected subtree. To compress Q, we try to move as many elements as possible from the counter of each node to its ancestors, without violating condition (1). This can be done by a post-order traversal of Q.

Space and time analysis. After each COMPRESS, it is clear that all nodes with at least one non-empty child must be full, therefore $|Q|$ is bounded by $O(\frac{1}{\varepsilon}\log u)$. Next we show that between two consecutive COMPRESS operations, its size remains $O(\frac{1}{\varepsilon}\log u)$. Suppose n goes from n_0 to $2n_0$ between two consecutive COMPRESS operations. Since we only add a node to Q when its parent's counter is $\varepsilon n/\log u \ge \varepsilon n_0/\log u$, there are at most twice as many of these nodes as those with counters greater than $\varepsilon n_0/\log u$. There are at most $\frac{2n_0}{\varepsilon n_0/\log u} = O(\frac{1}{\varepsilon}\log u)$ nodes of the latter type before

we do the next COMPRESS, so the size of Q remains bounded by $O(\frac{1}{\varepsilon}\log u)$.

It takes time $O(\log |Q|) = O\left(\log \frac{1}{\varepsilon} + \log\log u\right)$ to insert an element into the data structure. Then recall that we call COMPRESS whenever Q doubles, so it is invoked at most $O(\log n)$ times over the entire stream. Its amortized cost of $O(\frac{\log n \log u}{\varepsilon n})$ is negligible for n sufficiently large.

2.3 The randomized algorithm

We now describe a randomized quantile algorithm, which can be seen as a simplified version of the one by Manku et al. [19]. It is also inspired by the current best theoretical algorithm by Agarwal et al. [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.

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.

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, which is $O(\log s)$ per update amortized. So 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.

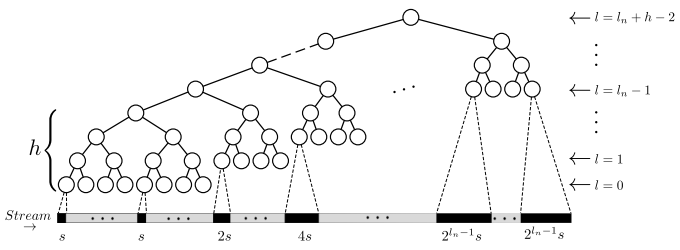


Figure 1: Illustration of Random.

As illustrated in Figure 1, all buffers that ever existed form a binary tree, where any non-leaf buffer is obtained by merging its two children. A leaf buffer is obtained directly by sampling from the stream. There are 2^{h-1} leaf buffers at level 0, each storing s elements from the stream; for $1 \le l < l_n$, 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 $1 \le l \le l_n$, and $2^{l_n+h-l-2}$ non-leaf buffers for $l_n + 1 \le l \le l_n + h - 2$.

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.

2.4 The MRL99 algorithm

As mentioned, the algorithm Random can be seen as a simplified version of the one by Manku et al. [19], which we denote as MRL99. Compared with Random, MRL99 has the following complications. (1) The parameters b, h and s are determined by solving a complicated optimization problem, whereas in our case, they are set easily. (2) In addition to level, each buffer X is also associated with a weight. In our case, the weight is always $2^{l(X)}$ so it is implicit, but this may not be the case in MRL99. (3) A more complex merging procedure is used that may merge buffers of different weights together and may merge more than two buffers at a time. In Random, we only merge 2 at a time and they must have the same weight.

Due to these complex procedures and the delicacy in setting the parameters, MRL99 is very difficult to analyze. As a result, Manku et al. [19] only gave a pessimistic $O\left(\frac{1}{\varepsilon}\log^2\frac{1}{\varepsilon}\right)$ bound, which is not interesting by today's standard since it can also be obtained by simply running $O(\log \frac{1}{\varepsilon})$ instances of the GK algorithm on multiple random samples of the stream. Nevertheless, the practical behavior of the algorithm should be very competitive.

3. TURNSTILE ALGORITHMS

In this section, we describe the quantile algorithms in the turnstile model (see Table 1 for a summary).

Recall that all existing algorithms 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 Random subset sum

Gilbert et al. [11] were the first to consider the quantile problem in the turnstile model, and designed the *random subset sum* sketch as a frequency estimation sketch to be used in the dyadic structure. However, it results in an overall size of $O(\frac{1}{\varepsilon^2}\log^2 u \log(\frac{\log u}{\varepsilon}))$: the dependence on ε is much higher than subsequent methods. Moreover, its update time is proportional to its size. Our implementation experience with this algorithm confirmed these properties, as the results were markedly poorer than with other algorithms. So we exclude it from further experimental evaluation.

3.2 Count-Min sketch

The Count-Min sketch consists of an array C of $w \times d$ counters, all initialized to 0. For each of the d rows of the array, it uses a pairwise independent hash function $h_i : [u] \to [w]$ that maps the elements in the universe to the w counters in this row. To insert/delete an element x in the sketch, we add/subtract 1 from the counter $C[i, h_i(x)]$, for $i = 1, \ldots, d$. To estimate the frequency of x, we return $\min_i C[i, h_i(x)]$.

It has been shown [9] that if $w = O(1/\varepsilon)$ and $d = O(\log \frac{1}{\delta})$, then the estimate has at most εn error with probability at least $1-\delta$. To use this in the dyadic structure, we only allow $\varepsilon n/\log u$ error from each level, so we use an error parameter $\varepsilon' = \varepsilon/\log u$ in each sketch. To find a quantile, we do a binary search with $\log u$ probes, where each probe involves $\log u$ queries to the sketches. We also want all $1/\varepsilon$ quantiles to be correct with constant probability, so a union bound implies that we need to set the failure probability of each sketch to $\delta' = \Theta(\varepsilon/\log^2 u)$. This leads to an overall size of $O(\frac{1}{\varepsilon}\log^2 u \log(\frac{\log u}{\varepsilon}))$. To process an update in the stream, we need to update $\log u$ sketches, one from each level, while updating each sketch requires updating all its d rows. So the total update time is $O(\log u \log(\frac{\log u}{\varepsilon}))$. These results are stated in [9]. We denote this algorithm as DCM (Dyadic Count-Min).

3.3 Count Sketch

The Count Sketch [4] is very similar to the Count-Min sketch. It also consists of an array C of $w \times d$ counters. For each row, in addition to h_i, it uses a second pairwise 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 $g_i(x) \cdot C[i, h_i(x)], i = 1, \ldots, d$.

Using a similar analysis to the Count-Min sketch, when setting $w = O(1/\varepsilon)$ and $d = O(\log \frac{1}{\delta})$, the Count Sketch also returns an estimate with more than εn error with probability at most δ—although the constant factors that emerge from the analysis are larger. On the other hand, the sketch also provides a guarantee based on the second frequency moment of the data [4]. In general, this guarantee is incomparable, but it can be tighter for some data distributions.

However, we observe another property of the Count Sketch that makes it appealing for the quantile problem, that it produces an unbiased estimator. In the dyadic structure, since we add up the estimates from $\log u$ sketches, 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 in fact leads to an asymptotic improvement over using the Count-Min sketch for the quantile problem, although it does not improve over Count-Min for the basic frequency estimation problem.

Analysis. Let $Y_i = g_i(x) \cdot C[i, h_i(x)]$. Each Y_i is clearly unbiased, since $g_i(x)$ maps to -1 or $+1$ with equal probability. Let Y be the median of $Y_i, i = 1 \ldots, d$ (assuming d is odd). 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 a $\sqrt{\log u}$ factor, rather than linearly in the number of levels.

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. Summing over all levels, we have the following guarantee.

THEOREM 1. *There is a randomized algorithm in the turnstile model that computes all ε-approximate quantiles with constant probability, using space $O(\frac{1}{\varepsilon}\log^{1.5} u \log^{1.5}(\frac{\log u}{\varepsilon}))$. Its update time is $O(\log u \log(\frac{\log u}{\varepsilon}))$.*

Note that since $u > 1/\varepsilon$, this is never worse than the guarantee from using the Count-Min sketch (for which symmetric estimators are not known). We denote this algorithm as DCS.

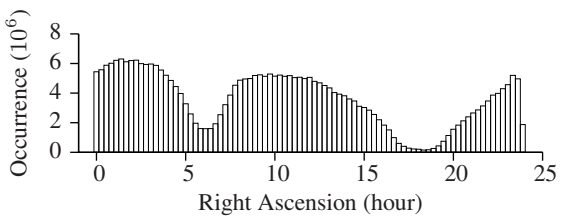


Figure 2: Distribution of MPCAT-OBS

3.4 A variant of the Count Sketch algorithm

Since every row in the Count Sketch is unbiased, it is natural to consider the following variant. Instead of having d rows for the sketch, we only use one row per sketch. But we build d independent copies of the whole structure. To find the rank of an element, we query all d structures and take the median. This still results in an unbiased estimator, though the analysis is less forthcoming. We include this variant, denoted as vDCS, in our experimental comparison. Note that it has the same space and update time as DCS.

4. EXPERIMENTS

4.1 Setup

We implemented all algorithms in C++, compiled with GCC. The executables were tested under Linux 2.6.18 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 2. 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 [13, 18]; 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.

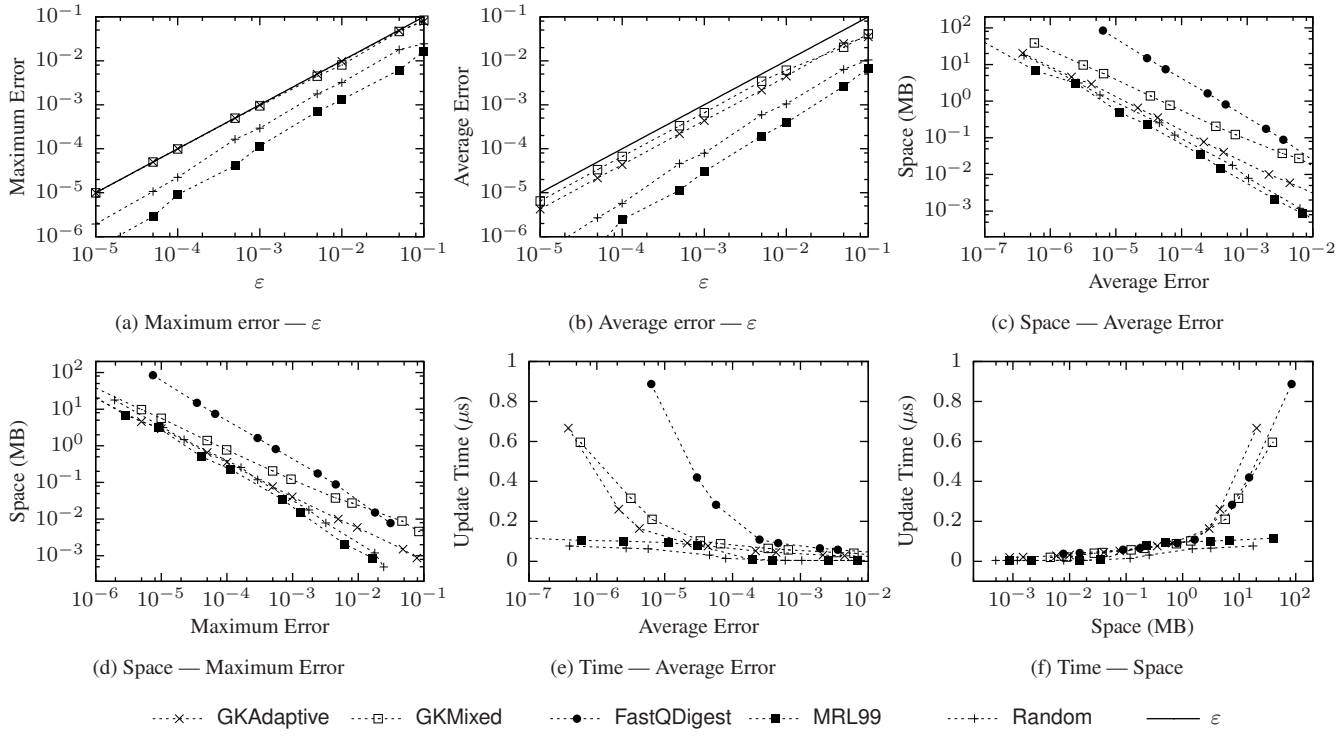
We first extract the ϕ-quantiles for $\phi = \varepsilon, 2\varepsilon, \cdots, (1-\varepsilon)$. For each ϕ-quantile extracted, we compute its true rank from the data, and take its difference from ϕn, divided by n. From all these errors, we take the maximum and average values. The former is exactly 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 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 3a and 3b 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



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

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

Figure 3: Results on MPCAT-OBS

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 3c and 3d 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 comes quite close, especially when max error is considered. GKMixed uses more space than GKAdaptive, despite using the more sophisticated COMPRESS procedure that leads to the $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ space guarantee. Recall that GKAdaptive uses a simple heuristic to remove tuples. 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

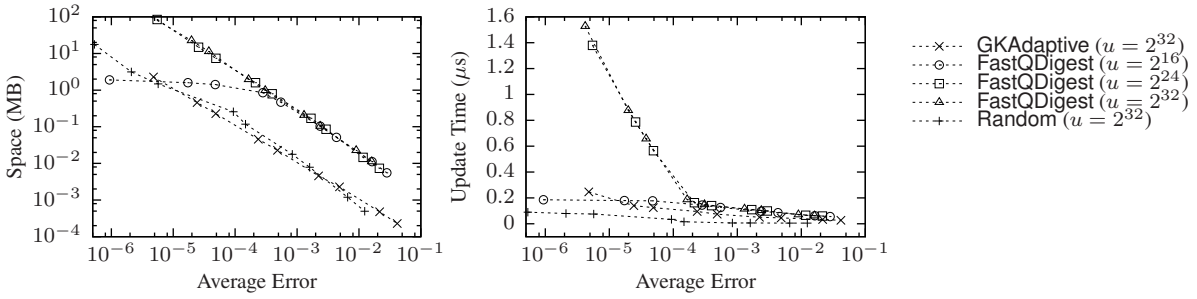
Figure 3e shows the tradeoff between error and the update time per element for each algorithm. Here we use log scale on the x-axis but linear scale on the y-axis, as the update time depends asymptotically on $\log(1/\varepsilon)$. Again, MRL99 and Random are the best two algorithms in terms of running time (for achieving the same error). Between the two, Random is slightly faster, due to the simplicity of the merging procedure. These two algorithms are fast for two reasons. First, they only sample a fraction of the stream to process—although this only kicks in when $n \gg 1/\varepsilon^2$. For small n (equivalently, for small ε), they are very fast primarily due to its simplicity: All they do are just sorting and merging! This phenomenon is more prominent on the space-time tradeoff plotted in Figure 3f, where all the algorithms, except for Random and MRL99, suffer a big speed loss when their space use exceeds 5MB, which is roughly the size of CPU cache. Recall that all these algorithms perform a binary search for each incoming element, so their running times are similar, and are not as cache-friendly as Random or MRL99. As MRL99 and Random are similar in terms of both space and time, we omit the results on MRL99 in the remaining plots to improve readability.

4.2.4 Varying universe size and data skewness

From the analysis, 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 4a and 4b, 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].

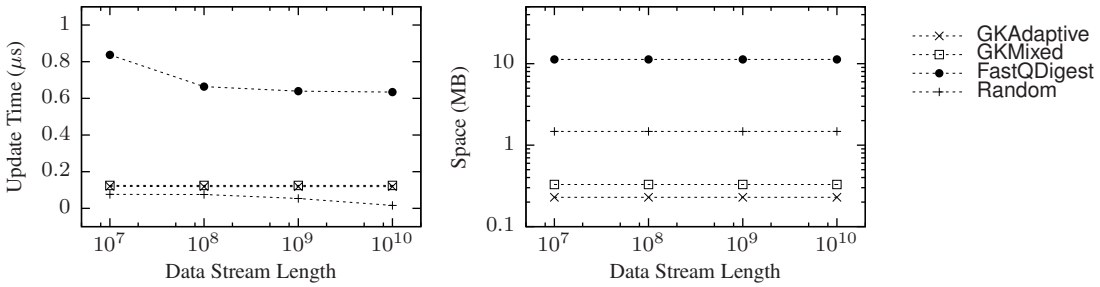
From the figures, we see that q-digest is only competitive when $\log u = 16$ and $\varepsilon < 10^{-5}$. However, when this is the case, storing the frequencies of all the u elements exactly only takes 0.25MB space. We also tested on data sets with different skewness by changing the variance of the 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.

[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

Figure 4: Varying the universe size on the normal distributed data, $\sigma = 0.15$



(a) Time — Stream Length (b) Space — Stream Length

Figure 5: Varying stream length on uniform distributed data, $u = 2^{32}$ and $\varepsilon = 0.0001$

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 5a and 5b. 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. Recall that the algorithm executes COMPRESS $\log n$ times throughout the whole stream.

Looking at Figure 5b, we see that the space used by GKAdaptive and GKMixed 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 *Sorted data*

Finally, we tested how stream order affects the performance of the algorithms. We generated uniformly distributed data with $u = 2^{32}$ and $n = 10^8$, and compared performance on the sorted and randomly ordered streams. Here we show the results for GKAdaptive, Random, the two most competitive algorithms.

In Figure 6a, we see that when the stream order changes from random to sorted, the update time of Random decreases while that of GKAdaptive increases, and the gap widens for smaller error. Recall that for Random, the amortized cost of all the merges per element is $O(1)$, while that of sorting is $O(\log(1/\varepsilon))$. When the stream is already sorted, the cost of the sorting goes down to $O(1)$ as well (the sorting algorithm we use in our implementation can make use of the existing sortedness in the data and reduce its work). On the other hand, when the stream is sorted, GKAdaptive has more trouble finding a removable element. When the stream is random ordered, very often the newly inserted element can be immediately removed. But when the stream is sorted, the heap always has to be checked, leading to a slower running time.

In terms of space, from Figure 6b we see that GKAdaptive suffers from the sorted data because it has more trouble removing elements. However, Random is taking advantage of it, due to its way of sampling. When the data is sorted, the sampling part contributes almost no error at all because the sampled elements are equally spaced.

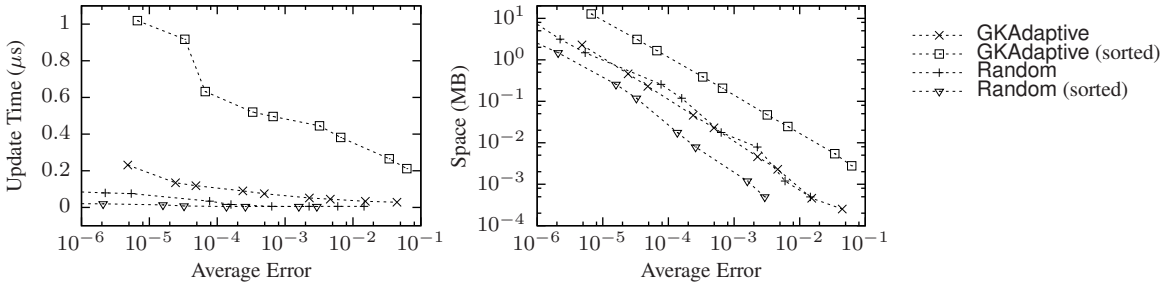
4.2.7 *Conclusions for cash register algorithms*

From our study, we can safely conclude that MRL99 and Random are generally the best performing algorithms with very similar behaviors. Random is slightly faster than MRL99, while the latter uses slightly less spaces. As Random is much easier to implement and has a better bound, it is our recommended method of choice. GKAdaptive is also very competitive, and can be used when a worst-case guarantee on the error is desired. 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 vDCS. We exclude the random subset sum sketch, as its performance is much weaker than these three.

Although we are experimenting with turnstile algorithms, it is not necessary to explicitly include deletions in the data sets: it is



(a) Time — Average Error

(b) Space — Average Error

Figure 6: Random order vs sorted — uniform distributed data, $u = 2^{32}$ and $n = 10^8$

Table 2: 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	11.153	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 3: 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

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 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 2, 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 3. 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 $w = 1/(\varepsilon \log u)$ for DCM and $w = 1/(\varepsilon\sqrt{\log u})$ for DCS and vDCS.

4.3.2 ε vs. actual error

In Figure 7a and 7b, 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.

The actual errors of these three algorithms appear similar, but note that DCM has a larger size while DCS and vDCS have exactly the same size. 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.

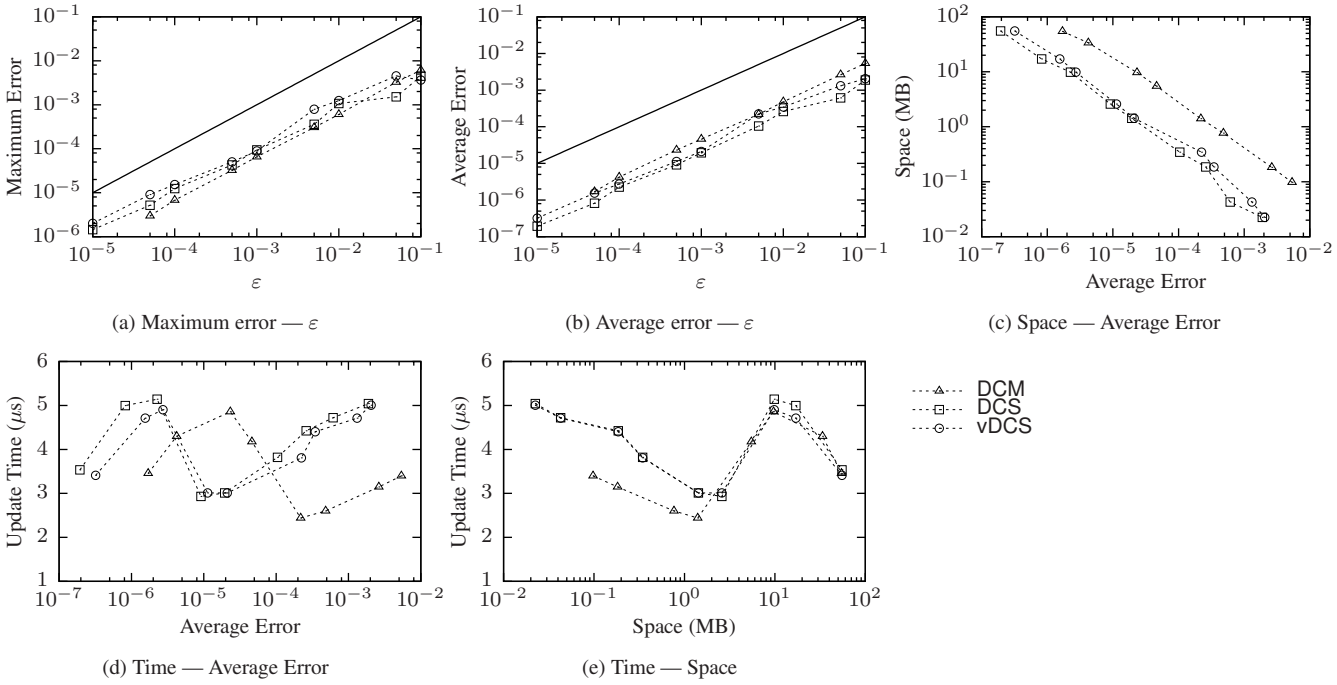
4.3.3 Space

Figure 7c shows the error-space tradeoffs of the algorithms. We see that to achieve the same error, DCS and vDCS require only about $1/10$ of the space required by DCM. The gap appears to be wider for larger error (or smaller space). One explanation is that, when the sketches are smaller, more levels in the dyadic hierarchy will use sketches as opposed to recording the frequencies exactly (recall that we use exact counting when the reduced universe size of a level is smaller than the sketch size). Thus, the effect of positive errors canceling with negative errors becomes more prominent. The difference between DCS and vDCS is small, with DCS being slightly better.

4.3.4 Time

Figure 7d shows the error-time tradeoff of these algorithms. Here, DCS and vDCS have similar curves, which is expected, since they have essentially the same structure and differ only in how quantiles are extracted. But the curve of DCM is quite different.

More revealing is Figure 7e, which shows the space-time tradeoff. Here we see that all three algorithms behave similarly, with DCM being slightly faster. This is expected, since if the space is the same, they will update exactly the same number of counters upon receiving an element. DCM is then slightly faster because it has one less hash function to compute.



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

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

Figure 7: Results on MPCAT-OBS

Looking at the three algorithms together, all of their update times decrease first as space goes up, then increase, and then decrease again. This phenomenon is the result of two competing factors. First, as the sketch size increases, more top levels in the dyadic hierarchy will store the exact frequencies instead of using sketches. Updating such an exact level only takes constant time, while updating a sketch takes $O(d)$ time. So in general the update time should go down as the sketch size goes up. But on the other hand, as the sketch size increases, the size of the whole data structure will eventually exceed the cache size (6MB), at which point we see a sharp increase in the update time due to more cache misses.

It is also instructive to compare Figure 7c and 7d with Figure 3c and 3e. 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 8a 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 8b 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 skewness does not obviously affect space or time (for a given ε), so we only show how the actual errors respond, in Figures 9a and 9b. 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 vDCS. This again is predicted well by the theory: Although in this paper, we analyzed the error of the Count Sketch in terms of n, i.e., the first frequency moment (F_1) of the data, in order to get the theoretical bound, its error actually depends more closely on the F_2 of the data [4]. As the variance decreases, the F_2 decreases, and the Count Sketch gets more accurate. On the other hand, the Count-Min sketch does not depend on the 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 difference between DCS and vDCS is negligible, but DCS has a nice theoretical analysis. All algorithms have almost the same update time.

5. REFERENCES

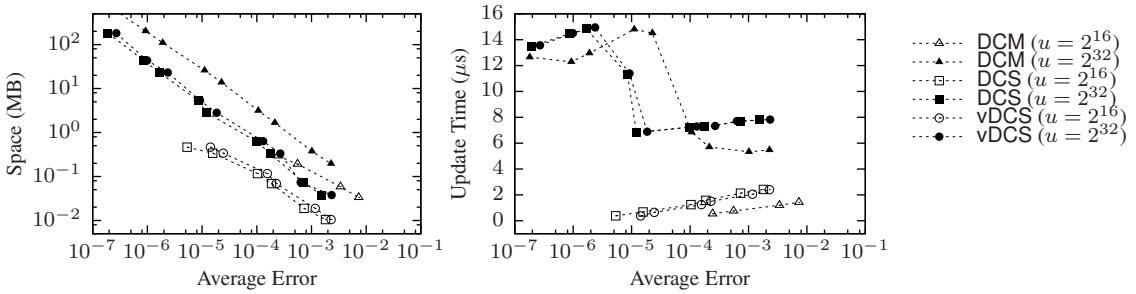
[1] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *ACM PODS*, 2012.

[2] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *ACM PODS*, 2004.

[3] M. Blum, R. W. Floyd, V. Pratt, R. L. Rievest, and R. E. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1973.

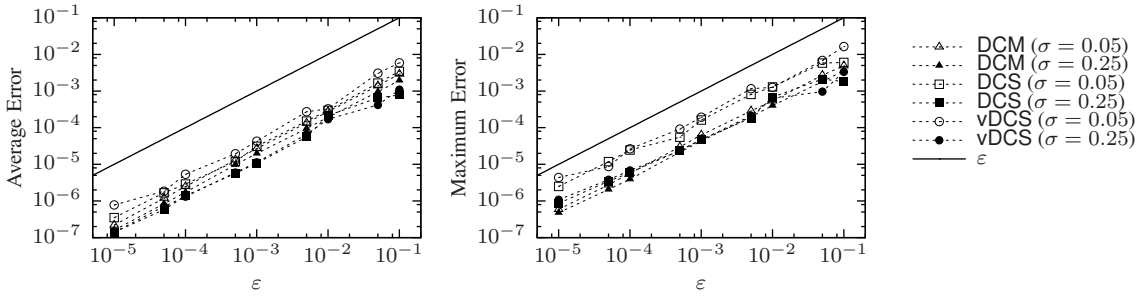
[4] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP*, 2002.

[5] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *ACM SIGMOD*, 2005.



(a) Space — Average Error (b) Time — Average Error

Figure 8: Varying the universe size on the Normal distributed data, $\sigma = 0.15$



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

Figure 9: Varying the deviation on the normal distributed data, $\sigma = 0.05, 0.25$

[6] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *VLDB*, 2008.

[7] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *ACM SIGMOD*, 2004.

[8] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *ACM PODS*, 2006.

[9] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] S. Ganguly and A. Majumder. CR-precis: A deterministic summary structure for update data streams. In *ESCAPE*, 2007.

[11] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, 2002.

[12] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *ACM SIGMOD*, 2005.

[13] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD*, 2001.

[14] M. Greenwald and S. Khanna. Power conserving computation of order-statistics over sensor networks. In *ACM PODS*, 2004.

[15] Z. Huang, L. Wang, K. Yi, and Y. Liu. Sampling based algorithms for quantile computation in sensor networks. In *ACM SIGMOD*, 2011.

[16] R. Y. S. Hung and H.-F. Ting. An $\Omega(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ space lower bound for finding ε-approximate quantiles in a data stream. In *Frontiers in Algorithmics (FAW)*, 2010.

[17] Z. Li, Y. Liu, M. Li, J. Wang, and Z. Cao. Exploiting ubiquitous data collection for mobile users in wireless sensor networks. *IEEE TPDS*, 24(2):312-326, 2013.

[18] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM SIGMOD*, 1998.

[19] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *ACM SIGMOD*, 1999.

[20] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[21] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Dynamic Grids and Worldwide Computing*, 13(4):277–298, 2005.

[22] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *ACM SenSys*, 2004.

[23] S. Suri, C. Toth, and Y. Zhou. Range counting over multidimensional data streams. *Discrete and Computational Geometry*, 2006.

[24] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[25] K. Yi and Q. Zhang. Optimal tracking of distributed heavy hitters and quantiles. In *ACM PODS*, 2009.