Received 21 September 2022, accepted 2 October 2022, date of publication 5 October 2022, date of current version 12 October 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3212393

APPLIED RESEARCH

Deterministic, Fast and Accurate Solution of the Heavy Hitters *q*-Tail Latencies Problem

ANNA FORNAIO, ITALO EPICOCO, MARCO PULIMENO, AND MASSIMO CAFARO, (Senior Member, IEEE)
Department of Engineering for Innovation, University of Salento, 73100 Lecce, Italy
Corresponding authors: Italo Epicoco (italo.epicoco@unisalento.it) and Massimo Cafaro (massimo.cafaro@unisalento.it)

ABSTRACT The heavy hitters q-tail latencies problem has been introduced recently. This problem, framed in the context of data stream monitoring, requires approximating the quantiles of the heavy hitters items of an input stream whose elements are pairs (item, latency). The underlying rationale is that heavy hitters are obviously among the most important items to be monitored, and their associated latency quantiles are of extreme interest in several network monitoring applications. Currently, two randomized (SQUARE and SQUAD) and one deterministic (QUASI) algorithms are available to solve the problem. In this paper, we present a novel deterministic algorithm and empirically show that it outperforms QUASI, the current state of the art deterministic algorithm for the problem, with regard to accuracy and speed.

INDEX TERMS Data stream mining, heavy hitters, quantiles, sketches.

I. INTRODUCTION

One of the key problems related to stream monitoring deals with measuring the latency quantiles. Indeed, latency is strictly related to the health of network connections. As an example, when a web server exhibits high tail latency then its users experience a low quality of service. It is worth to recall here that this may happen even though both the average and median latencies are low. Owing to the streaming setting, determining the exact latencies is not feasible using a limited amount of space. Therefore, we are interested in approximating the latency quantiles. In particular, we deal with the per item quantiles. However, the number of distinct items belonging to the input stream may be huge, requiring again too much space. As a consequence, the heavy hitters q-tail latencies problem has been introduced recently [1], [2]. Solving this problem requires, instead, approximating only the quantiles of the heavy hitters items of an input stream, therefore lowering the space requirement. The heavy hitters, i.e. the items with the highest number of occurrences, are indeed commonly associated to events of interest from a monitoring perspective and are a subset of significant items of the stream.

The associate editor coordinating the review of this manuscript and approving it for publication was Jerry Chun-Wei Lin.

Since the heavy hitters q-tail latencies problem is brand new, to the best of our knowledge, there are only three algorithms published in the literature. Two of them are randomized (SQUARE, Sampled QUAntile REconstruction and SQUAD, Sketching/sampling QUAntiles Duo), the other is deterministic (QUASI, QUAntile Sketches for heavy Items) [1], [2].

In this paper, we present QUHIS, QUantile Heavy Items Sketch, a novel deterministic algorithm. We empirically show that it is much more accurate and fast than QUASI, which is the current state of the art deterministic algorithm for the problem.

In practice, we are concerned with the simultaneous solution of two well known and studied problems, i.e., given an input data stream determining its heavy hitters (also known in the literature as frequent items) and, for each of them, determining its latency distribution. It is worth pointing out here that we aim for an approximate solution, owing to the fact that solving exactly this problem requires a huge amount of space (tracking the heavy hitters alone exactly requires at least $\Omega(n)$ space for a stream of length n). Additional difficulties are tied to the data stream model, in which items can only be accessed in their arrival order (i.e., random access to the data is not allowed). Moreover, in the common case of high-speed data streams each item must be processed quickly, ideally in constant $O(1)$ time, and then immediately discarded. Finally, the stream may not be necessarily finite, most of the streams are continuous and unbounded sequences of items. Therefore, the stream can not even be stored in its entirety. Anyway it is possible, without loosing generality, setting a priori the stream's length: for an infinite stream $\sigma = \sigma_1, \sigma_2, \cdots, \sigma_i, \sigma_{i+1}, \cdots$, setting its length to n is equivalent to consider the initial prefix $\sigma = \sigma_1, \sigma_2, \cdots, \sigma_n$. To recap, any algorithm processing a data stream must do so with limited space, ideally constant time for stream update, and with jus a single pass on the stream.

The data streaming problem we are interested in should not be confused with a *time series*, in which the stream items $\sigma_1, \sigma_2, \cdots, \sigma_i, \sigma_{i+1}, \cdots$ are related to un underlying signal X, i.e., a univariate function over $\mathbb{R}$. In the time series model, an item σ_i represents the i-th value of the signal: $\sigma_i = X[i]$. Instead, we deal with a stream model, called *cash register*, in which an item σ_i represents an *update* of the underlying signal. In this case the signal X consists of the vector of frequencies of the items appearing in the stream: letting $\sigma_i = j$, in this model the arrival of σ_i from the input stream means that the j-th value of the signal must be updated as follows: $X_i[j] = X_{i-1}[j] + 1$. This can be easily generalized to pairs (σ_i, w_i) in which there is a weight $w_i > 0$ associated to σ_i, and the update becomes $X_i[j] = X_{i-1}[j] + w_i$. Finally, when the weight w_i can also be negative, the corresponding streaming model is called *turnstile*.

Regarding heavy hitters, they are defined with regard to their *frequency*: considering a stream of length n consisting of tuples (item, weight) the frequency of a given item is simply the sum of its weights. However, in many applications the items are not associated to a corresponding weight, in which case the weight is implicitly assumed to be unitary.

In this work, the input stream to be processed contains elements which are pairs (item, latency), and the heavy hitters are computed with reference to the number of occurrences of the first component of each pair, the item. Heavy hitter are defined taking into account a user's defined support threshold ϕ, as the items whose frequency exceeds ϕn. In the literature, the problem of determining the heavy hitters has been referred to with different names, such as *market basket analysis* [3], *hot list analysis* [4] and *iceberg query* [5], [6].

Determining frequent items in a stream is a problem important both from a theoretical perspective and for its many practical applications; a few examples follows: (i) *popular products* - the stream may be the page views of products on the web site of an online retailer; frequent items are then the most frequently viewed products; (ii) *popular search queries* - the stream may consist of all of the searches on a search engine; frequent items are then the searches made most often; (iii) *TCP flows* - the stream may consist of the data packets passing through a network switch, each annotated with a source-destination pair of IP addresses, and the frequent items are then the flows that are sending the most traffic. Additional applications include, for instance, analysis of web logs [7], Computational and theoretical Linguistics [8] and the analysis of network traffic [9], [10], [11].

In order to determine the quantiles of the latencies associated to the heavy hitters, we also need to accurately track the input stream latencies. Of course, tracking the latencies for each incoming item would require a huge amount of space, therefore an algorithm must be clever enough to maintain, by using an appropriate data structure, only the latencies that are deemed important, being associated to potential heavy hitters candidate items. Again, solving this problem exactly is quite expensive, computing exact quantiles is impossible without storing all of the data [12]

Quantiles can be used to characterize the distributions of real datasets much better than simpler alternatives, i.e, by using the mean and the variance. Therefore quantiles are important from a practical perspective to both database implementers and users: as an example, they are necessary for query optimization, to split the data in parallel database systems, and are fundamental for statistical data analysis.

Among the possible applications, we recall here Internet-scale network monitoring. For instance, given a client-server application, the overall performance of the service (web site, database etc) is characterized by the latency encountered by the users' requests. Since the distribution of the latency values is usually skewed, tracking some specific quantiles (e.g., the 95th percentile) is commonly done. The Gigascope streaming database [13] relies on quantiles for monitoring network applications and systems. Moreover, as already discussed, quantiles are extensively used in database query optimizers in order to estimate how large intermediate results are; the estimates are then used to determine the overall best execution plan [14].

As discussed, tracking heavy hitters and latencies quantile are important problems *per se*; however, one may wonder why coupling these two problems deserves attention. As stated in [1]:

There are two complementary reasons why focusing on heavy hitters makes sense in the context of tail latency monitoring. First, since each heavy-hitter accounts for a significant fraction of the overall system load, it is important to ensure good quality of service for them. Second, when there are only a few elements associated with a given item, e.g., the item only appears in one or two transactions, it is enough that a single transaction suffers from a longer than usual delay in order for the tail-latency of that item to be very large. Such one-time events can be caused by, e.g., caching initialization, storage warm-up, route discovery overheads, and "bad luck" in terms of temporal overloads on intermediate components and devices. On the other hand, a large tail latency for a heavy hitter points to a repetitive problem, which hopefully is easier to discover and fix, and one that is also very important to resolve.

The rest of this manuscript is organized as follows. Section II introduces preliminary definition and notation. Section III recalls related work. QUASI is introduced in Section IV. We present QUHIS in Section V and analyze the computational and space complexity of QUASI and QUHIS in Section VI. Experimental results are discussed in Section VII. We draw our conclusions in Section VIII.

II. PRELIMINARY DEFINITIONS AND NOTATION

In this Section we briefly recall preliminary definitions and the notation that shall be used throughout the paper. We begin with the heavy hitters.

Definition 1 (Frequency of Items): Given a stream $\sigma = \{s_i\}_{i=1,2,\dots,n}$ *of* n *items drawn from the universe* $[m] = \{1, 2, \dots, m\}$ *the frequency of an item* s *is* $f_\sigma(s) = \sum_{\substack{i\in[n]\\ s_i=s}} 1$.

Next, we define the frequency vector.

Definition 2 (Frequency Vector): A stream $\sigma = \{s_i\}_{i=1,2,\dots,n}$*, whose items are drawn from the universe* $[m]$ *implicitly defines a frequency vector,* $\mathbf{f} = (f_1, f_2, \dots, f_m)$*, where* $f_i = f_\sigma(i)$ *is the frequency of item i.*

We can interpret a stream σ as a sequence of updates to the frequency vector $\mathbf{f}$: initially $\mathbf{f}$ is the null vector, then, for each item i in the stream, the entry in $\mathbf{f}$ corresponding to the frequency of the item i is incremented. We are now ready to define the ϵ-approximate frequency estimation problem.

Definition 3 (ϵ-Approximate Frequency Estimation Problem): Given a stream $\sigma = \{s_i\}_{i=1,2,\dots,n}$ *of n items drawn from the universe* $[m]$*, the frequency vector* $\mathbf{f}$ *defined by* σ*, and a value* $0 < \epsilon < 1$*, the* ϵ*-approximate frequency estimation problem consists in computing a vector* $\hat{\mathbf{f}} = (\hat{f}_1, \hat{f}_2, \dots, \hat{f}_m)$*, so that* $\left|\hat{f}_i - f_i\right| \leq \epsilon \|\mathbf{f}\|_\ell$*, for each* $i \in [m]$*, where* ℓ *can be either 1 or 2.*

Next, we define the ϕ-frequent items.

Definition 4 (ϕ-Frequent Items): Given a stream $\sigma = \{s_i\}_{i=1,2,\dots,n}$ *of n items drawn from the universe* $[m]$ *and a real value* $0 < \phi < 1$*, the* ϕ*-frequent items of* σ *are all those items whose frequency is above* ϕn*, i.e. the elements in the set* $F = \{s \in [m] : f_\sigma(s) > \phi n\}$.

We will often refer to the ϕ-frequent items of a stream simply as frequent items, leaving as implicit the reference to a ϕ value. Frequent items are also commonly referred to as *Heavy Hitters*. The ϵ-approximate frequent items problem related to determining ϕ-frequent items is defined as follows.

Definition 5 (Heavy Hitters): Given a stream $\sigma = \{s_i\}_{i=1,2,\dots,n}$ *of n items drawn from the universe* $[m]$*, a threshold* $0 < \phi < 1$ *and a tolerance* $0 < \epsilon < \phi$*, the* ϵ*-approximate frequent items problem consists in finding the set F, so that*:

1) *F contains all of the items s with frequency* $f_\sigma(s) > \phi n$ *(ϕ-frequent items);*
2) *F does not contain any item s such that* $f_\sigma(s) \leq (\phi - \epsilon)n$.

The definitions and notation related to quantiles are introduced below.

Definition 6 (Rank): Given a set $S \subset \mathbb{R}$ *with n elements, the rank of the element x, denoted by* $R(x)$*, is the number of elements in S less than or equal to x, i.e.*

$$R(x) := \Big|\{z \in S \mid z \leq x\}\Big|. \tag{1}$$

Definition 7 (q-Quantile): Given a set $S \subset \mathbb{R}$ *with n elements and a real number* $0 \leq q \leq 1$*, the inferior q-quantile (respectively superior q-quantile) is the element* x_q *whose rank in S is equal to*

$$x_q \in S \ : \ R(x_q) = \lfloor 1 + q \cdot (n-1) \rfloor \tag{2}$$

(respectively $R(x_q) = \lceil 1 + q \cdot (n-1) \rceil$*).*

The cumulative function $F : \mathbb{R} \rightarrow [0, 1]$ is related to the distribution of an ordered set S with n elements: given an arbitrary value x, $F(x)$ is defined as the ratio between the number of values less than or equal to x and n. In general, for a discrete set not necessarily sorted $\{x_i\}_i$, the cumulative function is defined as $F(x) = \sum_{x_i \leq x} p(x_i) = \sum_{x_i \leq x} \frac{|\{x \in S : x = x_i\}|}{n}$. The element related to the q-quantile $x_q \in S$ is the inverse of the function $F(x)$, i.e. x_q is such that $F(x_q) = q$. By definition, x_0 and x_1 are, respectively, the minimum and maximum element of the set S, and $x_{0.5}$ is the median.

Regarding the accuracy of an algorithm for tracking quantiles, it can be defined in two different ways, as follows.

Definition 8 (Rank Accuracy): Given an item v and a tolerance α*, an estimate of the rank* $\tilde{R}(v)$ *is returned such that*

$$\mid \tilde{R}(v) - R(v) \mid \leq \alpha \cdot n. \tag{3}$$

Definition 9 (Relative Accuracy): Given the item related to the q-quantile $x_q \in S$*, the* α*-accurate q-quantile is defined as the item* $\tilde{x}_q$ *such that*

$$\mid \tilde{x}_q - x_q \mid \leq \alpha \cdot x_q. \tag{4}$$

An algorithm is (q_0, q_1) α*-accurate if it returns* α*-accurate q-quantiles, for* $q_0 \leq q \leq q_1$.

Even though researchers have been focusing for long time their attention on the design of sketches and other data structures that could provide rank accuracy, datasets with heavy tails are such that algorithms providing rank accuracy can actually return values with arbitrary relative errors. In particular, it is well known that rank accuracy is not feasible for tracking high order quantiles of heavy tailed distributions.

We now formally state the heavy hitters q-tail latencies problem addressed in this paper.

Definition 10 (Heavy Hitters q-Tail Latencies Problem): Given a universe set $\mathcal{U}$*, consider a stream of pairs* $\sigma = \{(x_1, l_1), (x_2, l_2), \cdots\} \in (\mathcal{U} \times \mathbb{R})^+$ *in which a pair* (x_i, l_i) *is made of an item's identifier* $x_i \in \mathcal{U}$ *(whose implicit weight is unitary) and a latency* $l_i \in \mathbb{R}$.

Denote by $f_x = |\{(x_i, l_i) \in \sigma : x_i = x\}|$ *the frequency of x. The set of latencies associated to x is denoted by*

$$L_x = \{l_i \in \mathbb{R} : (x, l_i) \in \sigma\}. \tag{5}$$

Let σ *be a stream of length n,* $\phi \in [0, 1]$ *a threshold parameter and let* $0 < \epsilon < \phi$ *and* $0 < \alpha < 1$ *be two additional tolerance parameters. The heavy hitters q-tail latencies problem requires estimating the frequency* $\hat{f}_x$ *for each heavy hitter x so that the conditions stated in Definition 5 hold and, given* $q \in [0, 1]$*, estimating the q-quantile of* L_x *according either to Definition 8 or Definition 9.*

The following example illustrates the heavy hitters q-tail latencies problem. Consider the input stream whose items and relative latencies are listed in Table 1. Assume we are interested in the 0.99-quantile of the ϕ-frequent items in that stream and set $\phi = 0.25$. The length of the stream is $n = 24$, hence the 0.25-frequent items are those whose number of occurrences is greater than $0.25n = 6$.

The items that satisfies that condition are I_1 and I_2. The latencies referred to item I_1 ordered by increasing value are

$$L_{I_1} = \{27, 30, 33, 37, 38, 45, 75, 92, 96\}.$$

The latencies referred to item I_2 ordered by increasing value are

$$L_{I_2} = \{29, 40, 78, 80, 83, 87, 90\}.$$

Therefore, the 0.99-quantile of L_{I_1} is the item $x_{0.99} \in L_{I_1}$ whose rank is $R(x_{0.99}) = \lfloor 1 + 0.99(9-1) \rfloor = 8$, that is, $x_{0.99} = 92$. The 0.99-quantile of L_{I_2} is the item $x_{0.99} \in L_{I_2}$ whose rank is $R(x_{0.99}) = \lfloor 1 + 0.99(7-1) \rfloor = 6$, that is, $x_{0.99} = 87$.

TABLE 1. Data stream example.

$(I_1, 96)$	$(I_2, 90)$	$(I_1, 75)$	$(I_1, 30)$	$(I_1, 33)$	$(I_3, 29)$
$(I_2, 80)$	$(I_2, 78)$	$(I_3, 30)$	$(I_4, 27)$	$(I_1, 38)$	$(I_4, 73)$
$(I_1, 37)$	$(I_1, 45)$	$(I_1, 27)$	$(I_2, 40)$	$(I_4, 73)$	$(I_5, 45)$
$(I_5, 52)$	$(I_4, 32)$	$(I_2, 29)$	$(I_1, 92)$	$(I_2, 87)$	$(I_2, 83)$

III. RELATED WORK

The first algorithm for mining frequent items dates back to 1982, and is due to Misra and Gries [15]. Many years later, the Lossy Counting and Sticky Sampling algorithms by Manku et al. [16], were published in 2002. Interestingly, in 2003, the Misra and Gries algorithm was independently rediscovered and its computational complexity improved by Demaine et al. [9] and Karp et al. [17]. This algorithm is known in the literature as Frequent. Metwally et al. presented a few years later the Space Saving algorithm [18], which significantly improves the accuracy. These algorithms keep track of frequent items through the use of counters, i.e., data structures managing pair (item, estimated frequency).

Another group of algorithms is based on a sketch data structure, usually a bi-dimensional array hosting a counter in each cell. Pairwise independent hash functions are used to map stream's items to corresponding cells in the sketch. Sketch–based algorithms include CountSketch by Charikar et al. [7], Group Test [19] and Count-Min [20] by Cormode and Muthukrishnan, hCount [21] by Jin et al. and CMSS [22] by Cafaro et al. Table 2 recaps the most prominent algorithms for determining frequent items.

Algorithms for Correlated Heavy Hitters (CHHs) have been recently proposed by Lahiri et al. [28] and by Epicoco et al [27] in which a fast and more accurate algorithm for mining CHHs is presented.

All of the previous algorithms give identical importance to each item. However, in many applications is necessary to discount the effect of old data. Indeed, in some situations recent data is more useful and valuable than older data; such cases may be handled using the *sliding window* model [30], [31] or the time–fading model [32]. The key idea in sliding window is the use of a temporal window to capture fresh, recent items. This window periodically slides forward, allowing detection of only those frequent items falling in the window. In the time–fading model recent items are considered more important than older ones by *fading* the frequency count of older items. Among the algorithms for mining time–faded frequent items we recall λ-HCount [25] by Chen and Mei, FSSQ (Filtered Space Saving with Quasi–heap) [33] by Wu et al. and the FDCMSS algorithm [26], [34], [35] by Cafaro et al.

Regarding parallel algorithms, Cafaro et al. [23], [24], [36] provide parallel versions of the Frequent and Space Saving algorithms for message–passing architectures. Shared-memory versions of Lossy Counting and Frequent have been designed by Zhang et al. [37], [38], and parallel versions of Space Saving have been proposed by Dat et al. [39], Roy et al. [40], and Cafaro et al [41]. Accelerator based algorithms include Govindaraju et al. [42], Erra and Frola [43] and Cafaro et al. [41], [44]. Pulimeno et al. [45] present a message-passing based version of the CHHs algorithm [27], [46]; a parallel message-passing based version of [26] is presented in [47].

Distributed mining of heavy hitters include algorithms such as [48], [49], [50], [51], [52]. In the context of unstructured P2P networks, gossip–based algorithms have been proposed for mining frequent items including [29], [53], [54], [55].

The majority of quantile approximation algorithms have been designed to achieve additive (ϵn) approximation, defined as follows. Given a stream $\sigma = x_1, x_2, \cdots, x_n$ (or a dataset) of length n and an error parameter $0 < \epsilon < 1$, the ϵ-approximate ϕ-quantile is any element $x \in \sigma$ with rank $R(x) = |\{y_i \in \sigma : y_i \leq x\}|$ (i.e., the number of elements less than or equal to x) such that $(\phi - \epsilon)n \leq R(x) \leq (\phi + \epsilon)n$. An approximation $\hat{R}(x)$ is additive if $|R(x) - \hat{R}(x)| \leq \epsilon n$.

Among the many algorithms that have been proposed for approximate quantile computation (Table 3 recaps the most prominent algorithms), actually just a few of them are mergeable (i.e., they can be used in a distributed or parallel setting), whilst some are only one-way mergeable (which prevents their use in a distributed or parallel setting). For instance, Greenwald-Khanna [62] and t-digest [63], [64] are one-way mergeable, and therefore can not be used for parallel and distributed processing.

Mergeable quantile approximation algorithms include q-digest [56], M-Sketch [57], KLL [58], DCS [59], REQ [60], DDSketch [61] and UDDSketch with its parallel version PUDDSketch [65].

Quantile Digest or q-digest is a sketch designed to approximate quantiles with a guaranteed error bound on the accuracy, which in turn depends on the space actually used: the bigger the space bound, the smaller the approximation error. q-digest is deterministic and works by grouping data into *variable-sized* buckets, of almost equal height: so, it's similar to an equi-depth histogram except that buckets may overlap.

TABLE 2. Prominent heavy hitters algorithms.

Algorithm	Purpose	Complexity	Space	Features	Type	Model
Misra-Gries [15]	Frequent items	$O(n\ logn)$	$O(1/\epsilon)$	Counter-based	Sequential	Cash register
Frequent [9]	Frequent items	$O(n)$	$O(1/\epsilon)$	Counter-based identical to Misra-Gries but uses a clever data structure	Sequential	Cash register
Karp [17]	Frequent items	$O(n)$	$O(1/\epsilon)$	Counter-based identical to Misra-Gries but uses a clever data structure	Sequential	Cash register
Lossy Counting [16]	Frequent items	$O(n)$	$O(1/\epsilon\ \log \epsilon n)$	Counter-based	Sequential	Cash register
Space Saving [18]	Frequent items	$O(n)$	$O(1/\epsilon)$	Counter-based much more accurate than Misra-Gries	Sequential	Cash register
CountSketch [7]	Frequency of items (can be extended to detect frequent items)	$O(n)$	$O(1/\epsilon^2\ \log 1/\delta)$	Sketch-based	Sequential	Turnstile
CountMin [20]	Frequency of items (can be extended to detect frequent items)	$O(n)$	$O(1/\epsilon\ \log 1/\delta)$	Sketch-based	Sequential	Cash register
Parallel Space Saving [23]	Frequent items	$O(n/p + \log p)$	$O(1/\epsilon)$	Counter-based	Parallel (Message-Passing)	Cash register
Parallel Frequent [24]	Frequent items	$O(n/p + \log p)$	$O(1/\epsilon)$	Counter-based	Parallel (Message-Passing)	Cash register
λ-HCount [25]	Frequent items and frequency of items	$O(\log(-M/\log p))$	$O(\log(-M/\log p)/\epsilon^2)$	Sketch-based	Sequential	Time Fading + Cash register
FDCMSS [26]	Frequent items and frequency of items	$O(n)$	$O(1/\epsilon\ \log 1/\delta)$	Sketch-based (combines CountMin with Space Saving)	Sequential	Time Fading + Cash register
CSSCHH [27]	Correlated heavy Hitters	$O(n)$	$O\left(\frac{1}{\epsilon_2 \phi_1}\right)$ or $O\left(\frac{1}{\epsilon_1 \sqrt{\epsilon_2}}\right)$	Counter-based	Sequential	Cash register
MGCHH [28]	Correlated heavy Hitters	$O(n)$	$O\left(\frac{1}{(\phi_1 - \epsilon_1)\epsilon_2^2}\right)$ or $O\left(\frac{1}{\epsilon_1 \epsilon_2}\right)$	Counter-based	Sequential	Cash register
P2PSS [29]	Frequent Items	Rounds required for convergence $R_{\min} = \left\lvert \frac{\log \delta + 2 \log \left(\frac{2\phi - \epsilon - 2\sqrt{\phi^2 - \epsilon\phi}}{\epsilon p^*} \right)}{\log C} \right\rvert + 1$	$O(1/\epsilon)$	Counter-based	Distributed	Cash Register
CMSS [22]	Frequent items and frequency of items	$O(n)$	$O(1/\epsilon\ \log 1/\delta)$	Sketch-based (combines CountMin with Space Saving)	Sequential	Cash register

The input data belongs to a universe $\mathbb{U}$, represented by the integer values in the range $[0, U-1]$. This model is known in the literature as *fixed-size universe* model. Therefore, the assumption adopted in [56] is that $\mathbb{U}$ is a finite countable universe set of integer values with a known positive range. This is the strength of the approach but also its main drawback, in that it severely limits its wide adoption to other cases (e.g., negative or real values are not allowed). Another limitation is related to the impossibility of deleting inserted values: the algorithm only works *cash-register* model. Finally, q-digest uses $O(\frac{1}{\epsilon} \lg U)$ space.

M-Sketch, also known as Moment Sketch, is a summary intended for efficient mergeability. The summary has a very low memory footprint and low update overhead and makes use of statistical moments to approximate quantiles. The algorithm is deterministic and works in the cash register model. Given an integer k (which refers to the highest power used in the moments), an *order-k* moment sketch is a summary consisting of an array of $2k+3$ floating point values, corresponding to information retrieved from the processed dataset $D \subseteq \mathbb{R}$: x_{min} and x_{max}, the minimum and the maximum values seen so far, n, the count of items processed so far, and the set of sample moments and sample logarithmic moments, respectively $\mu_i = \frac{1}{n}\sum_{x \in D} x^i$ and $\nu_i = \frac{1}{n}\sum_{x \in D} \log^i(x)$ for $1 \leq i \leq k$. Since k is a small constant, the space used is $O(k) = O(1)$.

The moment sketch is aimed at tracking the moments of an empirical continuous interval to approximate quantiles, rather than the underlying distribution. To estimate quantiles, the *method of moments* is used, along with the *principle of maximum entropy*.

The sketch has been evaluated against existing solutions (such as GK and $t-$digest) and it proves to be faster, with a merge operation requiring a smaller memory footprint, whilst achieving the same ϵ error. However, Moment Sketch accuracy is lower than DDSketch [61], owing to a greater relative error.

KLL [58] is a randomized algorithm, hence it provides an additive approximation with probability at least $1-\delta$, with $0 < \delta < 1$. Actually, the authors designed two versions of KLL: one is space optimal with regard to randomized algorithms, requiring $O(\frac{1}{\epsilon} \lg \lg(\frac{1}{\delta}))$ space, but not mergeable, whilst the other is mergeable but not space optimal, requiring $O(\frac{1}{\epsilon} \lg^2 \lg(\frac{1}{\delta}))$ space. The algorithm works in the cash-register model. A recently proposed newer version, called KLL$^{\pm}$, exists and has been proposed to work in the *bounded deletion* model, in which at most a fraction α of the items inserted can be deleted.

KLL makes use of a hierarchy of *compactors* with varying capacities. A compactor is a buffer of size k, into which items ingested from a stream are stored with the same weight w. Each compactor has a *height*, h; in particular, $h = 1$ for the first compactor and $h = H$ for the last compactor in the chain. When a compactor is full, its items are sorted and compacted into a sequence of $k/2$ items, with weight $2w$: either the even or the odd items in the compactor are chosen with equal probability, whereas the others are discarded. The selected items are fed into another compactor (with height $h+1$), and so on with at most $H \leq \lceil \log(n/k) \rceil$ compactors chained together, n being the length of the stream. The total space required is kH. The weight of the items in a compactor with height h is given by $w_h = 2^{h-1}$.

Dyadic Count Sketch (DCS) is a randomized algorithm, and assumes that the data are drawn from an integer domain $[U] = [0, 1, \cdots, U-1]$ (fixed-size universe model). DCS imposes a dyadic structure over the universe U and leverages a Count-Sketch data structure for frequency estimation in each level of the dyadic structure. DCS suffer the same limitation of Q-Digest, namely a bounded integer range $[U] = [0, 1, \cdots, U-1]$. However, it works in the more general *turnstile* model, since it allows deletions. The space used is $O(\frac{1}{\epsilon} \lg^{1.5} u \lg^{1.5}(\frac{\lg u}{\epsilon}))$.

Relative-Error Quantiles Sketch [60] is a randomized algorithm striving to provide a multiplicative rather than additive error approximation; i.e., a relative error approximation. In particular, a multiplicative approximation requires, given an approximation $\hat{R}(x)$, that $|R(x) - \hat{R}(x)| \leq \epsilon R(x)$. It is worth recalling here that achieving multiplicative guarantees is known to be strictly harder than additive ones. Regarding the space bound, REQ uses $O(\log^{1.5}(\epsilon n)/\epsilon)$ space.

Cash register is the underlying model of REQ, so that no deletions of previously inserted items are allowed. The algorithm is based, as in KLL, on a structure of relative-compactor objects. The merge operation is tricky, owing to the need to ensure that relative-error guarantees are satisfied for the merged sketch.

UDDSketch is based on the DDSketch algorithm [61], is a deterministic algorithm and achieves better accuracy by using a different, carefully designed collapsing procedure. Being based on DDSketch, UDDSketch works in the turnstile model, so that deletions of previously inserted items are allowed (this corresponds to items arriving from the input stream with an associated negative weight).

The DDSketch data summary is a collection of buckets. The algorithm handles items $x \in \mathbb{R}_{>0}$ and requires in input two parameters to initialize the sketch: the first one, α, is related to the user's defined accuracy; the second one, m, represents the maximum number of buckets allowed. Using α, the algorithm derives the quantity $\gamma = \frac{1+\alpha}{1-\alpha}$ which is used to define the boundaries of the ith bucket B_i. All of the values x such that $\gamma^{i-1} < x \leq \gamma^i$ fall in the bucket B_i, with $i = \lceil \log_\gamma x \rceil$, which is just a counter variable initially set to zero. We recall here that DDSketch can also handle negative values by using another sketch in which an item $x \in \mathbb{R}_{<0}$ is handled by inserting $-x$.

Inserting a value is done by simply incrementing the counter by one; similarly deleting a value requires decrementing by one the corresponding counter (when a counter reaches the value zero, the corresponding bucket is discarded and thrown away). Initially the summary is empty, and buckets are dynamically added as needed. It is worth noting here that bucket indexes are dynamic as well, depending just on the

TABLE 3. Prominent quantile algorithms.

Algorithm	Space	Type	Model
Q-DIGEST [56]	$O(\frac{1}{\epsilon}\lg U)$	Deterministic	fixed-size universe and cash register
M-SKETCH [57]	$O(1)$	Deterministic	Cash register
KLL [58]	$O(\frac{1}{\epsilon}\lg^2 \lg(\frac{1}{\delta}))$	Randomized	Cash register
DCS [59]	$O(\frac{1}{\epsilon}\lg^{1.5} u \lg^{1.5}(\frac{\lg u}{\epsilon}))$	Randomized	Turnstile
REQ [60]	$O(\log^{1.5}(\epsilon n)/\epsilon)$	Randomized	Turnstile
DDSKETCH [61]	$O\left(\frac{b \log n/\delta}{\log((1+\alpha)/(1-\alpha))}\right)$	Deterministic	Turnstile

input value x to be inserted and on the γ value. In order to avoid that the summary grows without bounds, when the number of buckets in the summary exceeds the maximum number of m buckets, a collapsing procedure is executed. The collapse is done on the first two buckets with counts greater than zero (alternatively, it can be done on the last two buckets). Let the first two buckets be respectively B_y and B_z, with $y < z$. Collapsing works as follows: the count stored by B_y is added to B_z, and B_y is removed from the summary. Algorithm 1 presents the pseudo-code for the insertion of a value x into the summary $\mathcal{S}$.

Owing to its bucket collapsing strategy, DDSketch only provides a α-accurate (q_0, q_1)-sketch for $q_0 > 0$ and $q_1 = 1$, with the actual value of q_0 depending on how many items fall into the collapsed bucket. In particular, Proposition 4 of [61] provides the required condition for a quantile q to be α-accurate.

UDDSketch uses a uniform collapsing procedure that provides far better accuracy with regard to DDSketch, in particular our algorithm provides a α-accurate (q_0, q_1)-sketch for $q_0 = 0$ and $q_1 = 1$, i.e. all of the quantile queries can be answered α-accurately. In practice, we collapse all of the buckets, two by two. Given a pair of indices $(i, i+1)$, with i an odd index and $B_i \neq 0$ or $B_{i+1} \neq 0$, we create and add to the summary a new bucket with index $j = \lceil \frac{i}{2} \rceil$, with counter value equal to the sum of the B_i and B_{i+1} counters. The new bucket replaces the two collapsed buckets. Algorithm 2 reports the pseudocode of the uniform collapse procedure.

The collapsing procedure applied to an α-accurate (0, 1)-quantile sketch produces an α'-accurate (0, 1)-quantile sketch on the same input data with $\alpha' = \frac{2\alpha}{1+\alpha^2}$ (see Lemma 2 of [66]). Moreover, we also provide a theoretical bound on the accuracy achieved by the UDDSketch data summary. Given an input whose data domain is an interval $[x_{min}, x_{max}] \in \mathbb{R}_{>0}$ and an UDDSketch data structure using at most m buckets to process the input, the approximation error committed by UDDSketch using the uniform collapse procedure is bounded by $\hat{\alpha} = \frac{\tilde{\gamma}^2-1}{\tilde{\gamma}^2+1}$, with $\tilde{\gamma} = \sqrt[m-1]{\frac{x_{max}}{x_{min}}}$ (see Theorem 3 of [66]).[1]

[1]Due to a typo, in Theorem 3 of [66] the value of $\tilde{\gamma}$ is erroneously reported as $\sqrt[m]{\frac{x_{max}}{x_{min}}}$.

Algorithm 1 DDSketchUpdate(x, $\mathcal{S}$)

```
Require: $x \in \mathbb{R}_{>0}$: item to be inserted; $\mathcal{S}$: sketch in
         which the item must be inserted
Ensure : Insertion of item $x$ into the sketch $\mathcal{S}$
function DDSketchUpdate(x,S)
    $i \leftarrow \lceil \log_\gamma x \rceil$
    if $B_i \in \mathcal{S}$ then
        $B_i \leftarrow B_i + 1$
    else
        $B_i \leftarrow 1$
        $\mathcal{S} \leftarrow \mathcal{S} \cup B_i$
    end
    if $|\mathcal{S}| > m$ then
        let $B_y$ and $B_z$ be the first two buckets
        $B_z \leftarrow B_y + B_z$
        $\mathcal{S} \leftarrow \mathcal{S} \setminus B_y$
end
```

Algorithm 2 UniformCollapse($\mathcal{S}$)

```
Require: sketch $\mathcal{S} = \{B_i\}_i$
Ensure : resized sketch $\mathcal{S}$
function UniformCollapse(S)
    foreach $\{i : B_i > 0\}$ do
        $j \leftarrow \lceil \frac{i}{2} \rceil$
        $B'_j \leftarrow B'_j + B_i$
    end
    return $\mathcal{S} \leftarrow \{B'_i\}_i$
end
```

IV. THE QUASI ALGORITHM

QUASI is a deterministic algorithm, relying on Space Saving [18] for determining the heavy hitters, and on GK [62] for the quantiles associated to each potential heavy hitter candidate. Intuitively, QUASI uses a Space Saving summary to determine frequent items, allocating a GK sketch for each of the k counters of the summary to track the latency q-quantiles of each potential frequent ITEM.

We now describe the update procedure of QUASI, *Update*(x, l). Given the input stream $\sigma = \{(x_1, l_1), (x_2, l_2), \cdots\} \in (\mathcal{U} \times \mathbb{R})^+$, an incoming (x, l) pair in which x is an item and l its associated latency is processed as follows:

- if the item x is already monitored by one of the counters available in a Space Saving summary S, QUASI increases the corresponding counter and inserts l in the GK sketch associated to the counter;
- if the item x is not monitored by the counters in S:
  - if there is an available counter in S (i.e., not all of the counters are already busy monitoring items), then the item x is stored in one of the available counters and the counter's frequency is set to one; next, a GK sketch is initialized and associated to the Space Saving counter monitoring x, and l is inserted into the sketch;
  - otherwise, the item in the Space Saving summary S whose frequency is the minimum among all of the counters is evicted from the corresponding counter and replaced by x, and its frequency is increased by one; next, the GK sketch associated to the counter is reset and initialized, then l is inserted in the sketch.

In QUASI a query $Query(x, q)$ is performed as follows:

- if x is monitored by one of the Space Saving counters in S, its frequency is estimated as the value stored in the corresponding counter, $\hat{f}(x)$. The GK sketch corresponding to the counter is then used to estimate the q-quantile $\mathcal{L}_{x,q}$;
- otherwise, if x is not monitored in the summary, its frequency is estimated as the minimum value stored in the counters, and no latency quantile is reported in output.

Algorithms 3 and 4 provide the pseudo-code of the QUASI *update* and *query* functions.

Since Space Saving deterministically guarantees that every heavy hitter is monitored (provided that the summary is correctly initialized by using an appropriate number of counters), QUASI always provides a recall equal to 1 for the heavy hitters. However, as we shall see, its main drawback is related to the accuracy for the latencies associated to the heavy hitters. On the contrary, our algorithm QUHIS returns all of the heavy hitters (i.e., its recall is equal to 1) and, simultaneously, provides much more accurate latency quantiles. Moreover, it is also faster.

Algorithm 3 QUASI Update

```
Require: $x$: an item;
         $q$: the required quantile;
         $\mathcal{S}$: the Space Saving summary.
Ensure : insertion of pair $(x, l)$ in $\mathcal{S}$.
function Update($x, l, \mathcal{S}$)
    if $x \in \mathcal{S}$ then
        $\hat{f}(x) \leftarrow \hat{f}(x) + 1$
        // $GK(x)$ is the sketch
            corresponding to $x$
        $GK(x) \leftarrow$ GetGKSketch($x, \mathcal{S}$)
        // Insert $l$ in the $GK(x)$ sketch
        GKUpdate($l, GK(x)$)
    else
        if $|\mathcal{S}| < k$ then
            $\hat{f}(x) \leftarrow 1$
            Initialize a sketch $GK_x$ for the item $x$
        else
            Let $x_{min}$ be the item with minimum
              frequency in $\mathcal{S}$
            $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x_{min}\}$
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$
            $\hat{f}(x) \leftarrow \hat{f}_{x_{min}} + 1$
            Reset the sketch $GK_{x_{min}}$
        end
        // Insert $l$ in the $GK(x)$ sketch
        $GK(x) \leftarrow$ GetGKSketch($x, \mathcal{S}$)
        GKUpdate($l, GK(x)$)
    end
end
```

Algorithm 4 QUASI Query

```
Require: $x$: an item;
         $q$: the required quantile;
         $\mathcal{S}$: the Space Saving summary.
Ensure : estimated frequency of $x$ and $q$-quantile if $x$ is
         frequent.
function Query($x, q, \mathcal{S}$)
    if $x \in S$ then
        // let $GK(x)$ be the sketch
            corresponding to $x$
        $GK(x) \leftarrow$ GetGKSketch($x, \mathcal{S}$)
        $\hat{q} \leftarrow$ GKQuery ($q, GK(x)$)
        return ($\hat{f}(x), \hat{q}$)
    else
        return ($\hat{f}(x_{min}), null$)
    end
end
```

V. THE QUHIS ALGORITHM

We present QUHIS, our deterministic algorithm for the solution of the heavy hitters q-tail latencies problem. Like QUASI, determining the heavy hitters in the input stream is done by using Space Saving [18]. However, for tracking the latency quantiles associated to the heavy hitters we rely instead on our own UDDSketch algorithm [66].

Given the input stream $\sigma = \{(x_1, l_1), (x_2, l_2), \cdots\} \in (\mathcal{U} \times \mathbb{R})^+$, an incoming (x, l) pair in which x is an item and l its associated latency is processed by QUHIS as follows:

- if the item x is already monitored by one of the counters available in a Space Saving summary S, QUHIS increases the corresponding counter and inserts l in the UDDSketch sketch associated to the counter;
- if the item x is not monitored by the counters in S:
  - if there is an available counter in S (i.e., not all of the counters are already busy monitoring items), then the item x is stored in one of the available counters and the counter's frequency is set to one; next, an UDDSketch sketch is initialized and associated

to the Space Saving counter monitoring x, and l is inserted into the sketch;
  - otherwise, the item in the Space Saving summary S whose frequency is the minimum among all of the counters is evicted from the corresponding counter and replaced by x, and its frequency is increased by one; next, the UDDSketch sketch associated to the counter is reset and initialized, then l is inserted in the sketch.

In QUHIS a query $Query(x, q)$ is performed as follows:

- if x is monitored by one of the Space Saving counters in S, its frequency is estimated as the value stored in the corresponding counter, $\hat{f}(x)$. The UDDSketch sketch corresponding to the counter is then used to estimate the q-quantile $\mathcal{L}_{x,q}$;
- otherwise, if x is not monitored in the summary, its frequency is estimated as the minimum value stored in the counters, and no latency quantile is reported in output.

Algorithms 5 and 6 provide the pseudo-code of the QUHIS *update* and *query* functions.

VI. COMPUTATIONAL AND SPACE COMPLEXITY

Regarding the space complexity, the QUASI algorithm solves the heavy hitters q-tail latencies problem using space $O\left(\theta^{-1}\epsilon^{-2} \cdot \left(1 + \log\left(N\epsilon^2\theta\right)\right)\right)$ (see Theorem 2 in [1], where θ is the threshold to determine the heavy hitters, N is the stream length and ϵ is the error tolerance.

The space complexity of QUHIS is given by the following theorem.

Theorem 1: The QUHIS *algorithm solves the heavy hitters q-tail latencies problem using space $O(\epsilon^{-1}\log^{-1}(\frac{1+\alpha}{1-\alpha}))$. This result holds when the algorithm is configured such that* Space Saving *can return all of the heavy hitters with an error bounded by ϵ and* UDDSketch *can return the desired quantiles satisfying the user's defined α-accuracy requirement.*

Proof: Space Saving requires $O(\epsilon^{-1})$ counters in the stream summary data structure. Each counter, besides the identity of an item and its estimated frequency, stores a UDDSketch data structure configured with m buckets. Letting α be the desired accuracy level, the UDDSketch data structure requires a number of buckets equal to $m = O(\log^{-1}(\frac{1+\alpha}{1-\alpha}))$.

Indeed, as stated in Section III, the approximation error committed by UDDSketch using the uniform collapse procedure is bounded by $\hat{\alpha} = \frac{\tilde{\gamma}^2-1}{\tilde{\gamma}^2+1}$, with $\tilde{\gamma} = \sqrt[m-1]{\frac{x_{max}}{x_{min}}}$. Therefore, it holds that $m = 1 + \frac{\log(\frac{x_{max}}{x_{min}})}{\log\tilde{\gamma}}$.

Since $\tilde{\gamma} = \sqrt{\frac{1+\alpha}{1-\alpha}}$, it follows that $m = 1 + \frac{\log(\frac{x_{max}}{x_{min}})}{\log\sqrt{\frac{1+\alpha}{1-\alpha}}}$.

Finally, taking into account that both x_{max} and x_{min} are constants, the theorem follows.

□

We now discuss the computational complexity of both the algorithms. Since Space Saving requires $O(1)$ worst case constant time to insert an incoming item into its stream

Algorithm 5 QUHIS Update

```
Require: $x$: item to be inserted;
         $l$: the latency associated to $x$;
         $\mathcal{S}$: the Space Saving summary.
Ensure : insertion of pair $(x, l)$ in $\mathcal{S}$.

function $Update(x, l, \mathcal{S})$
    if $x \in \mathcal{S}$ then
        $\hat{f}(x) \leftarrow \hat{f}(x) + 1$
        // $UDDS(x)$ is the sketch
           corresponding to $x$
        $UDDS(x) \leftarrow$ GetUDDSketch$(x, \mathcal{S})$
        DDSketchUpdate $(l, UDDS(x))$
    else
        // $k$ is the number of counters
           in $\mathcal{S}$
        if $|\mathcal{S}| < k$ then
            $\hat{f}(x) \leftarrow 1$
            Initialize a sketch $UDDS(x)$ for $x$
        else
            Let $x_{min}$ be the item with minimum
              frequency in $\mathcal{S}$
            $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x_{min}\}$
            $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$
            $\hat{f}(x) \leftarrow \hat{f}_{x_{min}} + 1$
            // $UDDS(x_{min})$ is the sketch
               corresponding to $x_{min}$
            Reset the $UDDS(x_{min})$ sketch
        end
    end
    // let $UDDS(x)$ be the sketch
       corresponding to $x$
    DDSketchUpdate $(l, UDDS(x))$
end
```

Algorithm 6 QUHIS Query

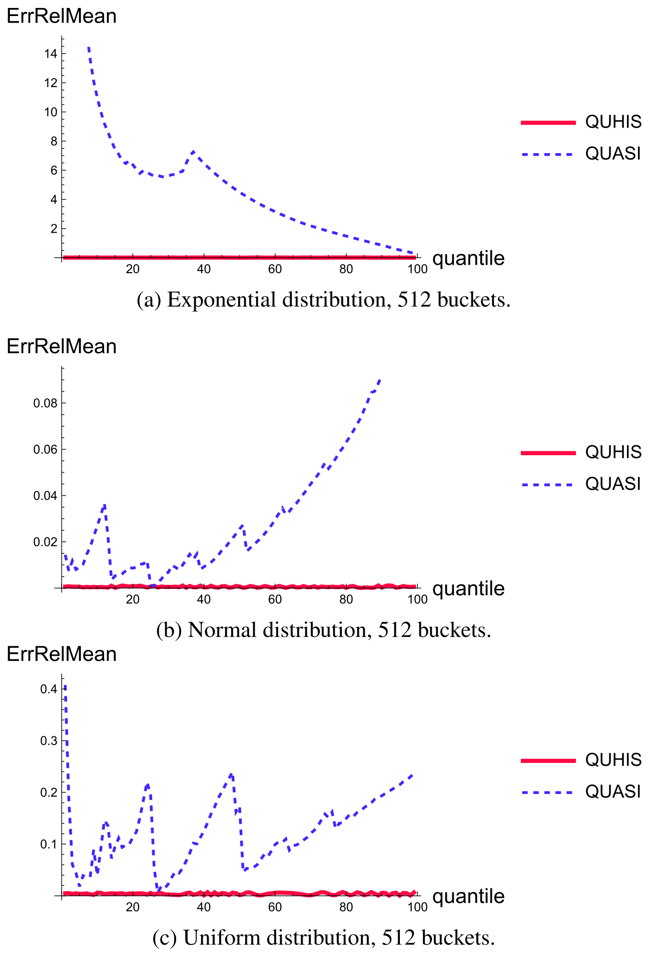
```
Require: $x$: an item;
         $q$: the required quantile;
         $\mathcal{S}$: the Space Saving summary.
Ensure : estimated frequency of $x$ and $q$-quantile if $x$ is
         frequent.

function $Query(x, q, \mathcal{S})$
    if $x \in \mathcal{S}$ then
        // let $UDDS(x)$ be the sketch
           corresponding to $x$
        $\hat{q} \leftarrow$ DDSketchQuery $(q, UDDS(x))$
        return $(\hat{f}(x), \hat{q})$
    else
        return $(\hat{f}(x_{min}), null)$
    end
end
```

summary data structure, and GreenwaldKanna requires $O\left(\log\frac{1}{\varepsilon} + \log\log(\varepsilon n)\right)$, it follows that QUASI requires $O\left(\log\frac{1}{\varepsilon} + \log\log(\varepsilon n)\right)$ to handle an incoming item.

TABLE 4. Parameters' values, with default values highlighted in bold.

Parameters	
n	5×10^6
ρ	1.9
ϵ	10^{-2}
ϕ	2×10^{-2}
α	10^{-3}
m	128, **512**, 1024
$dist$	uniform, **normal**, exponential



(a) Exponential distribution, 512 buckets.

(b) Normal distribution, 512 buckets.

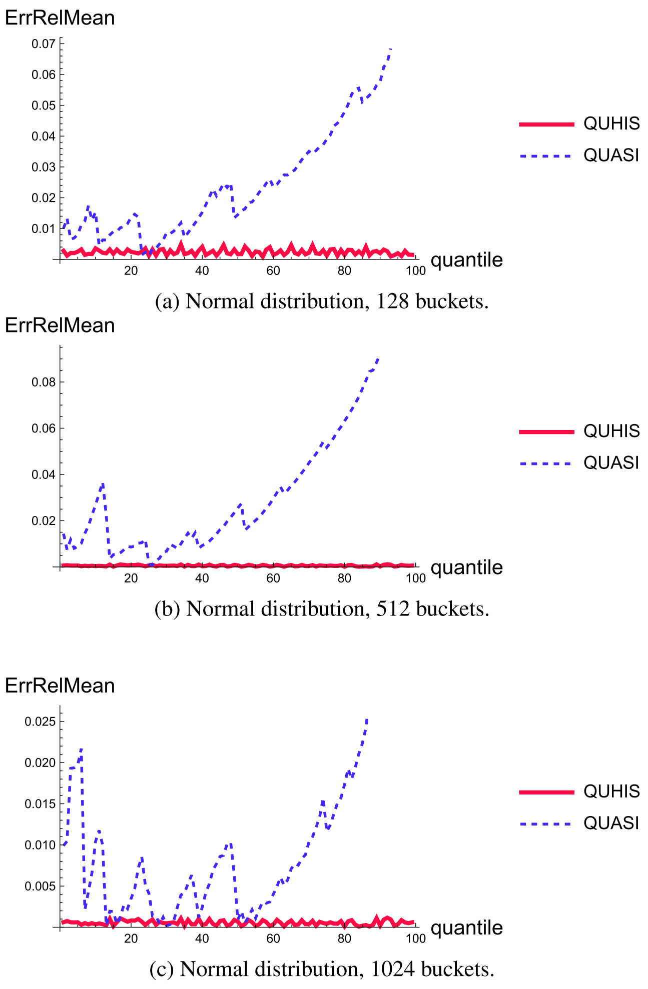
(c) Uniform distribution, 512 buckets.

FIGURE 1. Relative error varying the distributions, 512 buckets.

QUHIS requires $O(1)$ worst case constant time to insert an incoming item into its Space Saving stream summary data structure. The worst case time to update the UDDSketch data structure, under the constraint defined in Theorem 1 (i.e., UDDSketch returns the desired quantiles satisfying the user's defined α-accuracy requirement), is $O(1)$. Indeed, if the number of buckets m is fixed as dictated by α, then no collapse will happen, hence the worst case time to insert an item in the sketch is $O(1)$. Finally, the overall worst case time complexity for the insertion of an incoming item into the QUHIS data structures is $O(1)$.

If we allow m, the number of buckets, to grow

and, fixing the number of buckets m for the UDDSketch data structure as dictated by α, UDDSketch guarantees that no collapses will happen. In this case



(a) Normal distribution, 128 buckets.

(b) Normal distribution, 512 buckets.

(c) Normal distribution, 1024 buckets.

FIGURE 2. Relative error varying the number of buckets, normal distribution.

Therefore, the worst case computational complexity of QUHIS is $O(\log^{-1}(\frac{1+\alpha}{1-\alpha}))$.

VII. EXPERIMENTAL RESULTS

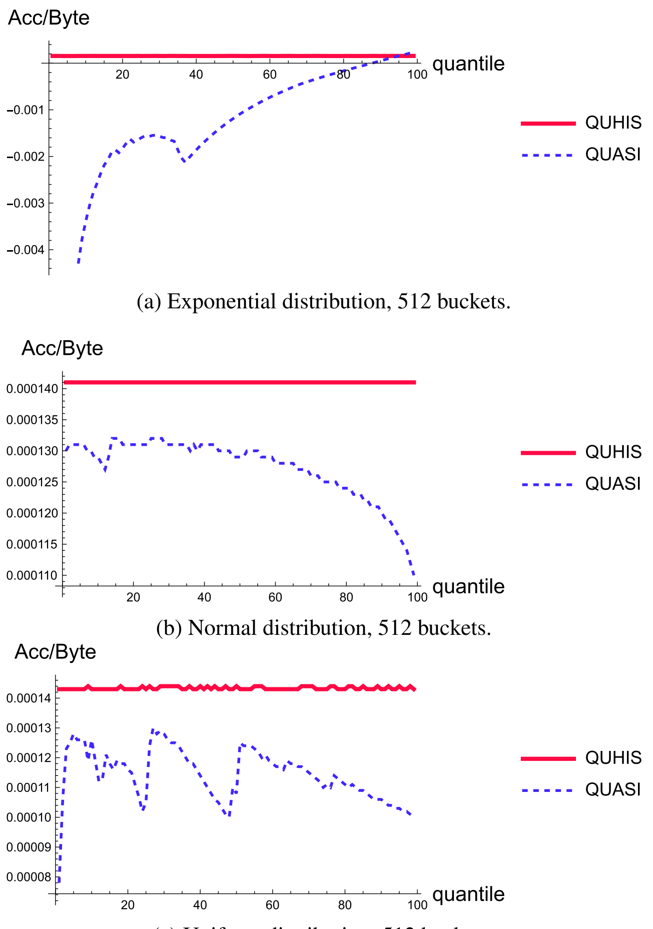
In this Section, we present and discuss experimental results, thoroughly comparing QUASI versus our algorithm QUHIS. In particular, we shall compare and contrast the algorithms with regard to their speed, measured as the number of updates per second, and with regard to their accuracy.

For each item x, let L_x be the set of latencies associated to x in the stream σ (see eq. (5)). For a ϕ-frequent item x and a quantile $0 < q < 1$, let l_q be the q-quantile in L_x and $\hat{l}_q$ the q-quantile estimated by an algorithm.

The *relative error* associated to the computation of the q-quantile is

$$ErrRel_x(q) = \frac{|\tilde{l}_q - l_q|}{l_q}. \tag{6}$$

In general, an estimation is better than another if its relative error is close to zero, since $\tilde{l}_q$ is closer to l_q. Averaging over all of the ϕ-frequent items, we obtain (letting r be the number

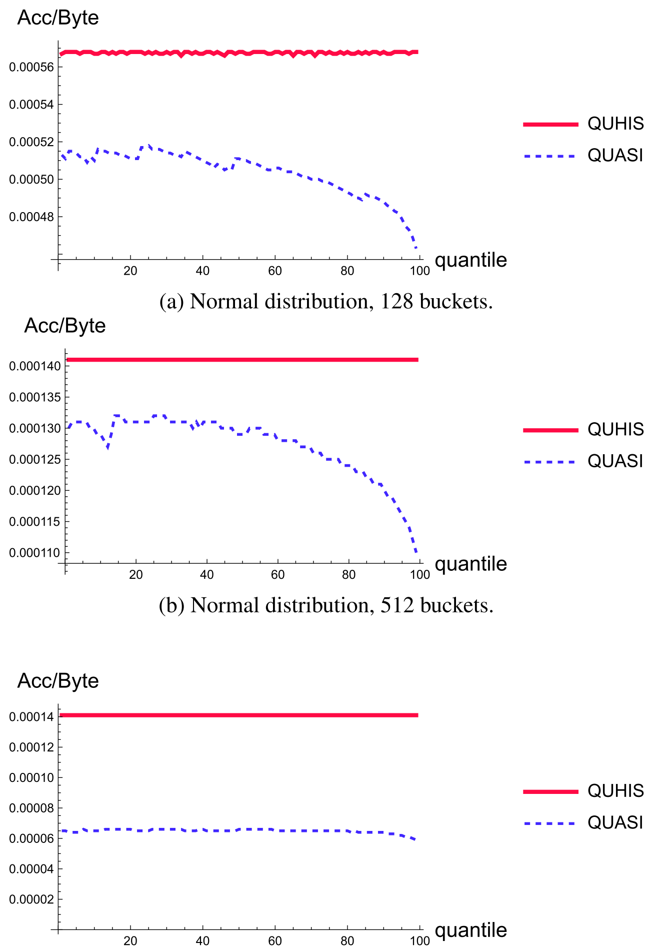


(a) Exponential distribution, 512 buckets.

(b) Normal distribution, 512 buckets.

(c) Uniform distribution, 512 buckets.

FIGURE 3. Accuracy/byte varying the distribution, 512 buckets.



(a) Normal distribution, 128 buckets.

(b) Normal distribution, 512 buckets.

(c) Normal distribution, 1024 buckets.

FIGURE 4. Accuracy/byte varying the number of buckets, normal distribution.

of ϕ-frequent items):

$$ErrRelMean(q) = \frac{1}{r} \sum_{x \; frequent} ErrRel_x(q). \tag{7}$$

Denoting by *size* the sketch dimension (GK for QUASI and UDDSketch for QUHIS), we evaluate the *accuracy per byte* of an algorithm computing a q-quantile as follows:

$$Acc/byte(q) = \frac{1 - ErrRelMean(q)}{size}. \tag{8}$$

The greater the value of $Acc/byte$, the greater is the algorithm's accuracy. To compare the speed of execution, we measure the throughput in *Updates per ms*, i.e. how many pairs (x, l) coming from the stream σ are processed by an algorithm in one millisecond. Letting *time* be the time in milliseconds required to execute n updates, it holds that:

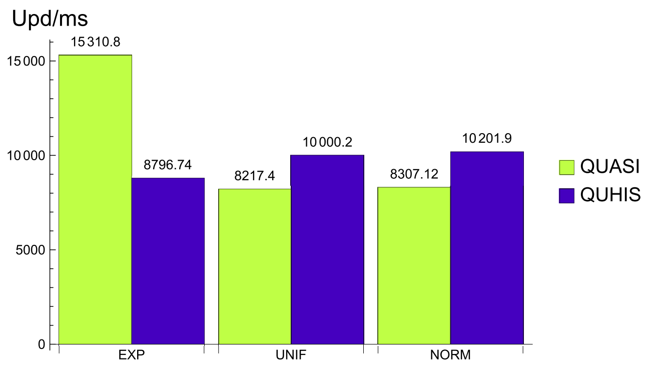
$$Throughput = \frac{n}{time}. \tag{9}$$

The fastest algorithm is therefore the one whose Upd/ms value is greater.

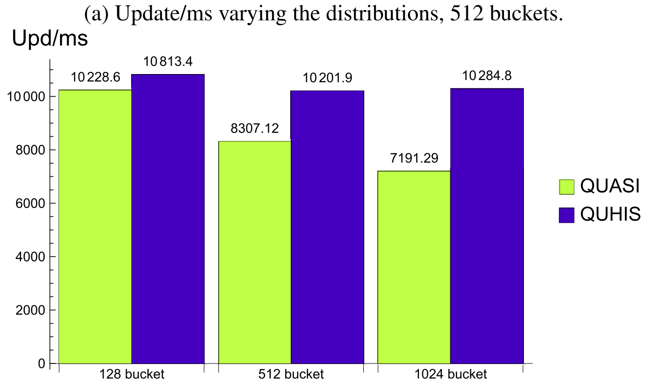
The tests have been carried out on a workstation equipped with 2 Intel Xeon E5-2620 processors with 15 MB cache L3, 6 cores of execution and 64 GB of RAM, using the operating system Linux Ubuntu 20.04.4 LTS and the Intel oneAPI C++ v2022.1.0 compiler.

Three different stream have been synthetically generated according to specific distributions. The items have been generated using a zipfian distribution, since this is the distribution most commonly used in the literature regarding the algorithms for determining the frequent items. It is a discrete distribution with probability mass $\mathbb{P}(x) = \frac{x^{-(\rho+1)}}{\zeta(\rho+1)}$, where ρ is a *skewness* parameter and $\zeta(z)$ is the Riemann Zeta function $\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z}$. The latencies have been generated according to a uniform distribution $\mathcal{U}[1, 10^3]$, a normal distribution $\mathcal{N}(10^4, 10^3)$ and an exponential distribution $Exp(10^4)$. Moreover, the parameters have been set so that both the algorithms under test could have the same maximum number of buckets in their sketches and the same number of counters in their Space Saving summary.

Table 4 reports the parameters used in the experiments: n, the stream's length; ρ, the zipfian distribution skewness; ϵ, the error tolerance used in Space Saving; ϕ, the threshold used to determine the frequent items; α, an accuracy parameter for UDDSketch; m, the maximum number of buckets and *dist*, the distributions used to generate the latencies.



(a) Update/ms varying the distributions, 512 buckets.



(b) Update/ms varying the number of buckets buckets, normal distribution.

FIGURE 5. **Update/ms.**

The experiments have been carried out varying first the distribution of the latencies and setting $m = 512$, then varying m e fixing *dist* as the normal distribution. For each combination, we computed, varying q in $\{0.01, \cdots, 0.99\}$, the mean relative error (see eq. (7)), the *accuracy per byte* (see eq. (8)) an the *Throughput* (see eq. (9)). All of the tests have been repeated ten times and the results averaged.

Figures 1 and 2 depict the mean relative error varying respectively the distribution and the number of buckets in the sketches. As shown, QUHIS strongly outperforms QUASI, exhibiting a mean relative error almost equal to zero. Therefore our algorithm provides very good estimates of q-quantiles. Figures 3 and 4 show the *accuracy per byte* varying respectively the distribution and the number of buckets in the sketches. Again, QUHIS outperforms QUASI, exhibiting a much better accuracy with regard to the space actually used. Regarding the speed of execution, Figure 5 clearly show that QUHIS is faster than QUASI for both the uniform and the normal distributions, and is slightly slower with regard to the exponential distribution. Even though we did not report additional results obtained with other distributions, we note here that, in general, QUHIS is faster than QUASI.

VIII. CONCLUSION

In this paper we tackled the heavy hitters q-tail latencies problem, which has been introduced recently. The problem is related to data stream monitoring and requires approximating the quantiles of the heavy hitters items of an input stream whose elements are pairs (item, latency). In the context of networking, stream monitoring is now recognized as fundamental, owing to a variety of possible applications, spanning several fields. In particular, the problem asks for detecting and reporting the input stream's heavy hitters, and, for each of them, their associated latency quantiles.

The underlying rationale is that heavy hitters are obviously among the most important items to be monitored, and their associated latency quantiles are of extreme interest in several network monitoring applications. To the best of our knowledge, only two randomized (SQUARE and SQUAD) and one deterministic (QUASI) algorithms are available to solve the problem. We introduced QUHIS, a novel deterministic algorithm that solves the heavy hitters q-tail latencies problem and empirically showed that it outperforms QUASI with regard to accuracy and speed.

Possible future developments include the design and analysis of a novel randomized algorithm solving the heavy hitters q-tail latencies problem. This algorithm should be compared versus the SQUARE and SQUAD algorithms in order to assess their performances. Another line of research is related to the design of a corresponding parallel algorithm for the problem at hand, proving that the underlying data structures are therefore mergeable.

REFERENCES

[1] R. Shahout, R. Friedman, and R. Ben Basat, "SQUAD: Combining sketching and sampling is better than either for per-item quantile estimation," 2022, *arXiv:2201.01958*.

[2] R. Shahout, R. Friedman, and R. B. Basat, "SQUAD: Combining sketching and sampling is better than either for per-item quantile estimation," in *Proc. 15th ACM Int. Conf. Syst. Storage*, New York, NY, USA, Jun. 2022, p. 152, doi: 10.1145/3534056.3535009.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1997, pp. 255–264.

[4] P. B. Gibbons and Y. Matias, "Synopsis data structures for massive data sets," in *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization*. Providence, RI, USA: AMS, 1999, pp. 39–70.

[5] K. Beyer and R. Ramakrishnan, "Bottom–up computation of sparse and iceberg cubes," in *Proc. ACM SIGMOD Int. Conf. Manage. Data.*, New York, NY, USA, 1999, pp. 359–370.

[6] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," in *Proc. 24th Int. Conf. Very Large Data Bases*. San Mateo, CA, USA: Morgan-Kaufmann, 1998, pp. 299–310.

[7] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. 29th Int. Colloq. Automata, Lang. Program.* New York, NY, USA: Springer-Verlag, 2002, pp. 693–703.

[8] A. Gelbukhl, Ed., "Computational linguistics and intelligent text processing," in *Proc. 7th Int. Conf.*, in Lecture Notes in Computer Science, vol. 3878. New York, NY, USA: Springer-Verlag, Feb. 2006.

[9] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proc. ESA*, 2002, pp. 348–360.

[10] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. 1st ACM SIGCOMM Workshop Internet Meas.*, 2001, pp. 75–80.

[11] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 23–39, Apr. 2003.

[12] J. I. Munro and M. S. Paterson, "Selection and sorting with limited storage," *Theor. Comput. Sci.*, vol. 12, no. 3, pp. 315–323, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397580900614

[13] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2003, pp. 647–651, doi: 10.1145/872757.872838.

[14] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1998, pp. 426–435, doi: 10.1145/276304.276342.

[15] J. Misra and D. Gries, "Finding repeated elements," *Sci. Comput. Program.*, vol. 2, no. 2, pp. 143–152, 1982.

[16] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. 28th Int. Conf. Very Large Data Bases*, 2002, pp. 346–357. [Online]. Available: http://dl.acm.org/citation.cfm?id=1287369.1287400

[17] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, pp. 51–55, Mar. 2003.

[18] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. Int. Conf. Database Theory*, 2005, pp. 398–412. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-30570-5_27

[19] G. Cormode and S. Muthukrishnan, "What's hot and what's not: Tracking most frequent items dynamically," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 249–278, Mar. 2005.

[20] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[21] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou, "Dynamically maintaining frequent items over a data stream," in *Proc. 12th Int. Conf. Inf. Knowl. Manage.*, New York, NY, USA, 2003, pp. 287–294, doi: 10.1145/956863.956918.

[22] M. Cafaro, I. Epicoco, and M. Pulimeno, "CMSS: Sketching based reliable tracking of large network flows," *Future Gener. Comput. Syst.*, vol. 101, pp. 770–784, Dec. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X1930490X

[23] M. Cafaro, M. Pulimeno, and P. Tempesta, "A parallel space saving algorithm for frequent items and the Hurwitz zeta distribution," *Inf. Sci.*, vol. 329, pp. 1–19, Feb. 2016.

[24] M. Cafaro and P. Tempesta, "Finding frequent items in parallel," *Concurrency Comput., Pract. Exper.*, vol. 23, no. 15, pp. 1774–1788, Oct. 2011.

[25] L. Chen and Q. Mei, "Mining frequent items in data stream using time fading model," *Inf. Sci.*, vol. 257, pp. 54–69, Feb. 2014.

[26] M. Cafaro, M. Pulimeno, I. Epicoco, and G. Aloisio, "Mining frequent items in the time fading model," *Inf. Sci.*, vols. 370–371, pp. 221–238, Nov. 2016.

[27] I. Epicoco, M. Cafaro, and M. Pulimeno, "Fast and accurate mining of correlated heavy hitters," *Data Mining Knowl. Discovery*, vol. 32, no. 1, pp. 162–186, Jan. 2018, doi: 10.1007/s10618-017-0526-x.

[28] B. Lahiri, A. P. Mukherjee, and S. Tirthapura, "Identifying correlated heavy-hitters in a two-dimensional data stream," *Data Mining Knowl. Discovery*, vol. 30, no. 4, pp. 797–818, Jul. 2016, doi: 10.1007/s10618-015-0438-6.

[29] M. Cafaro, I. Epicoco, and M. Pulimeno, "Mining frequent items in unstructured P2P networks," *Future Gener. Comput. Syst.*, vol. 95, pp. 1–16, Jun. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18315838

[30] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows: (extended abstract)," in *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms*. Philadelphia, PA, USA: SIAM, 2002, pp. 635–644.

[31] S. Muthukrishnan, "Data streams: Algorithms and applications," *Found. Trends Theor. Comput. Sci.*, vol. 1, no. 2, pp. 117–236, 2005.

[32] G. Cormode, F. Korn, and S. Tirthapura, "Exponentially decayed aggregates on data streams," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Washington, DC, USA, Apr. 2008, pp. 1379–1381, doi: 10.1109/ICDE.2008.4497562.

[33] S. Wu, H. Lin, L. H. U, Y. Gao, and D. Lu, "Novel structures for counting frequent items in time decayed streams," *World Wide Web*, vol. 20, no. 5, pp. 1111–1133, Sep. 2017, doi: 10.1007/s11280-017-0433-5.

[34] M. Cafaro, I. Epicoco, M. Pulimeno, and G. Aloisio, "On frequency estimation and detection of frequent items in time faded streams," *IEEE Access*, vol. 5, pp. 24078–24093, 2017.

[35] M. Pulimeno, I. Epicoco, and M. Cafaro, "Distributed mining of time-faded heavy hitters," *Inf. Sci.*, vol. 545, pp. 633–662, Feb. 2021. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85092082360&doi=10.1016%2fj.ins.2020.09.048&partnerID=40&md5=4558a50086979a64547a59cc0bd4c7b0

[36] M. Cafaro and M. Pulimeno, "Merging frequent summaries," in *Proc. 17th Italian Conf. Theor. Comput. Sci. (ICTCS)*, vol. 1720, 2016, pp. 280–285.

[37] Y. Zhang, "Parallelizing the weighted lossy counting algorithm in high-speed network monitoring," in *Proc. 2nd Int. Conf. Instrum., Meas., Comput., Commun. Control*, Dec. 2012, pp. 757–761.

[38] Y. Zhang, Y. Sun, J. Zhang, J. Xu, and Y. Wu, "An efficient framework for parallel and continuous frequent item monitoring," *Concurrency Comput., Pract. Exper.*, vol. 26, no. 18, pp. 2856–2879, Dec. 2014.

[39] S. Das, S. Antony, D. Agrawal, and A. E. Abbadi, "Thread cooperation in multicore architectures for frequency counting over multiple data streams," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 217–228, Aug. 2009.

[40] P. Roy, J. Teubner, and G. Alonso, "Efficient frequent item counting in multi-core hardware," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1451–1459.

[41] M. Cafaro, M. Pulimeno, I. Epicoco, and G. Aloisio, "Parallel space saving on multi- and many-core processors," *Concurrency Comput., Pract. Exper.*, vol. 30, no. 7, p. e4160, Apr. 2018, doi: 10.1002/cpe.4160.

[42] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, pp. 611–622.

[43] U. Erra and B. Frola, "Frequent items mining acceleration exploiting fast parallel sorting on the GPU," *Proc. Comput. Sci.*, vol. 9, pp. 86–95, Jun. 2012.

[44] M. Cafaro, I. Epicoco, G. Aloisio, and M. Pulimeno, "CUDA based parallel implementations of space-saving on a GPU," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Genoa, Italy, Jul. 2017, pp. 707–714.

[45] M. Pulimeno, I. Epicoco, M. Cafaro, C. Melle, and G. Aloisio, *Parallel Mining of Correlated Heavy Hitters* (Lecture Notes in Computer Science: Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10964. 2018, pp. 627–641. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85049976155&doi=10.1007%2f978-3-319-95174-4_48&partnerID=40&md5=8c4fd82ce9ca8d56636fbed204b6de15

[46] M. Pulimeno, I. Epicoco, M. Cafaro, C. Melle, and G. Aloisio, *Parallel Mining of Correlated Heavy Hitters on Distributed and Shared-Memory Architectures*, Y. Song et al., Eds. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2019, pp. 5111–5118. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85062632089&doi=10.1109%2fBigData.2018.8622201&partnerID=40&md5=dbcf6e1b0563b34b328d0698da1a2096

[47] M. Cafaro, M. Pulimeno, and I. Epicoco, "Parallel mining of time-faded heavy hitters," *Expert Syst. Appl.*, vol. 96, pp. 115–128, Apr. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417417307777

[48] P. Cao and Z. Wang, "Efficient top-K query calculation in distributed networks," in *Proc. 23rd Annu. ACM Symp. Princ. Distrib. Comput.*, New York, NY, USA, 2004, pp. 206–215, doi: 10.1145/1011767.1011798.

[49] Q. Zhao, M. Ogihara, H. Wang, and J. Xu, "Finding global icebergs over distributed data sets," in *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, New York, NY, USA, 2006, pp. 298–307, doi: 10.1145/1142351.1142394.

[50] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of thresholded counts," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2006, pp. 289–300, doi: 10.1145/1142473.1142507.

[51] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *Proc. 21st Int. Conf. Data Eng. (ICDE)*, Washington, DC, USA, 2005, pp. 767–778, doi: 10.1109/ICDE.2005.68.

[52] S. Venkataraman, D. Song, P. Gibbons, and A. Blum, "New streaming algorithms for fast detection of superspreaders," School Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-04-142, Jan. 2005.

[53] J. Sacha and A. Montresor, "Identifying frequent items in distributed data sets," *Computing*, vol. 95, no. 4, pp. 289–307, Apr. 2013.

[54] E. Çem and Ö. Özkasap, "ProFID: Practical frequent items discovery in peer-to-peer networks," *Future Gener. Comput. Syst.*, vol. 29, no. 6, pp. 1544–1560, Aug. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X12001859

[55] B. Lahiri and S. Tirthapura, "Identifying frequent items in a network using gossip," *J. Parallel Distrib. Comput.*, vol. 70, no. 12, pp. 1241–1253, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731510001358

[56] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: New aggregation techniques for sensor networks," in *Proc. 2nd Int. Conf. Embedded Netw. Sensor Syst.*, 2004, pp. 239–249.

[57] E. Gan, J. Ding, K. S. Tai, V. Sharan, and P. Bailis, "Moment-based quantile sketches for efficient high cardinality aggregation queries," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1647–1660, Jul. 2018.

[58] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *Proc. IEEE 57th Annu. Symp. Found. Comput. Sci. (FOCS)*, Oct. 2016, pp. 71–78.

[59] G. Luo, L. Wang, K. Yi, and G. Cormode, "Quantiles over data streams: Experimental comparisons, new analyses, and further improvements," *VLDB J.*, vol. 25, no. 4, pp. 449–472, Aug. 2016, doi: 10.1007/s00778-016-0424-7.

[60] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Veselý, "Relative error streaming quantiles," in *Proc. 40th ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, New York, NY, USA, Jun. 2021, pp. 96–108, doi: 10.1145/3452021.3458323.

[61] C. Masson, J. E. Rim, and H. K. Lee, "DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2195–2205, Aug. 2019, doi: 10.14778/3352063.3352135.

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

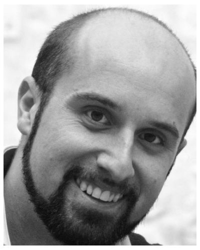
[63] T. Dunning and O. Ertl, "Computing extremely accurate quantiles using t-digests," 2019, *arXiv:1902.04023*.

[64] T. Dunning, "The t-digest: Efficient estimates of distributions," *Softw. Impacts*, vol. 7, Feb. 2021, Art. no. 100049. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2665963820300403

[65] M. Cafaro, C. Melle, I. Epicoco, and M. Pulimeno, "Data stream fusion for accurate quantile tracking and analysis," *Inf. Fusion*, vol. 89, pp. 155–165, Jan. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253522000975

[66] I. Epicoco, C. Melle, M. Cafaro, M. Pulimeno, and G. Morleo, "UDDSketch: Accurate tracking of quantiles in data streams," *IEEE Access*, vol. 8, pp. 147604–147617, 2020.

ANNA FORNAIO received the M.Sc. degree *(magna cum laude)* in mathematics from the University of Salento, Italy. She is currently a Research Grantee at the University of Salento. Her research interests include data mining and cybersecurity.



ITALO EPICOCO received the Ph.D. degree in innovative materials and technologies from the ISUFI, University of Lecce, Italy, in 2003. He is currently an Assistant Professor at the University of Salento, Lecce, and the Director of the Advanced Scientific Computing (ASC) Division at Euro-Mediterranean Center on Climate Change Foundation (CMCC). He is the Director of the Master in Scientific Programming and a Co-Leader of the HPC Laboratory, University of Salento. His main skills concern computer engineering and computer science. His research interests include the design of data mining algorithms on high-end computing architectures, high performance, and distributed computing. He is also working on the optimization of numerical kernels for solving PDEs. The current application field is the earth system models with particular focus on ocean models. His research activity also includes the benchmarking and evaluation of new emerging computational technologies based on GP-GPU, hybrid architectures and FPGA-based computing. He published more than 80 papers in refereed books, journals, and conference proceedings on parallel and grid computing.

MARCO PULIMENO received the Ph.D. degree in mathematics and computer science from the University of Salento, Italy. He is currently a Postdoctoral Researcher at the University of Salento. His research interests include high-performance computing, distributed computing, and, in particular, parallel data mining. He published on the topic of frequent items and quantiles in several refereed journals and conference proceedings.

MASSIMO CAFARO (Senior Member, IEEE) received the Laurea (M.Sc.) degree in computer science from the University of Salerno and the Ph.D. degree in computer science from the University of Bari. He is currently an Associate Professor at the Department of Engineering for Innovation, University of Salento. He is also the Director of the Master in Applied Data Science and the Head of the HPC Laboratory, University of Salento. He is the author of more than 110 refereed articles and holds a patent on distributed database technologies. He focuses his research on high performance and distributed computing on both theoretical and practical aspects, in particular the design and analysis of sequential, parallel, and distributed algorithms. His research interest include parallel and distributed computing, cloud and grid computing, data mining, machine learning, deep learning, big data, security, and cryptography. He is a Senior Member of the IEEE Computer Society, a Senior Member of the ACM, the Vice Chair of Regional Centers, and a Co-ordinator of the Technical Area on Data Intensive Computing for the IEEE Technical Committee on Scalable Computing. He serves as an Associate Editor for IEEE Access and *Future Internet* (MDPI), and as a Moderator for the IEEE TechRxiv preprint repository ("Computing and Processing" category).