q-MAX: A Unified Scheme for Improving Network Measurement Throughput

Ran Ben Basat
Harvard University

Gil Einziger
Ben Gurion University

Junzhi Gong
Harvard University

Jalil Moraney
Technion

Danny Raz
Technion

ABSTRACT

Network measurement is an essential building block for a variety of network applications such as traffic engineering, quality of service, load-balancing and intrusion detection. Maintaining a per-flow state is often impractical due to the large number of flows, and thus modern systems use complex data structures that are updated with each incoming packet. Therefore, designing measurement applications that operate at line speed is a significant challenge in this domain.

In this work, we address this challenge by providing a unified mechanism that improves the update time of a variety of network algorithms. We do so by identifying, studying, and optimizing a common algorithmic pattern that we call q-MAX. The goal is to maintain the largest q values in a stream of packets. We formally analyze the problem and introduce interval and sliding window algorithms that have a worst-case constant update time. We show that our algorithms perform up to ×20 faster than library algorithms, and using these new algorithms for several popular measurement applications yields a throughput improvement of up to ×12 on real network traces. Finally, we implemented the scheme within Open vSwitch, a state of the art virtual switch. We show that q-MAX based monitoring runs in line speed while current monitoring techniques are significantly slower.

CCS CONCEPTS

• **Networks → Network algorithms**.

ACM Reference Format:
Ran Ben Basat, Gil Einziger, Junzhi Gong, Jalil Moraney, and Danny Raz. 2019. q-MAX: A Unified Scheme for Improving Network Measurement Throughput. In *Internet Measurement Conference (IMC '19), October 21–23, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3355369.3355569

1 INTRODUCTION

Measurement tasks such as finding the heavy hitter flows [29, 49], identifying the most frequent subnets [12, 17], approximating the traffic's entropy [10, 43], detecting loss [39, 57], traffic anomalies [50, 68] and micro-bursts [9] are at the core of network applications such as routing, traffic engineering, load balancing, and intrusion detection [20, 23, 36, 42, 44, 51, 52, 63]. Measurements are performed at a single device [15, 40, 59, 64] or on multiple *Network Measurement Points (NMP)* [34, 47, 48]. They can also be categorized as datapath algorithms [15, 64], or ones that require the controller involvement in answering queries [47, 59]. The measurement data is collected by a network controller that creates a *Network-wide* view of the traffic. Such a holistic view is necessary for identifying various network anomalies such as *Super Spreaders* and port scanners [50]. The measurement period may refer to a fixed interval (e.g., a day) or a sliding window, (e.g., the last ten minutes) [14, 41, 58]. The literature offers a variety of measurement algorithms. These vary in their optimization goals, measurement metrics, implementation, accuracy guarantees, and measurement period (e.g., interval or sliding window, single-device, or network-wide).

Our work identifies a (previously overlooked) design pattern that naturally arises in a variety of measurement algorithms. Specifically, algorithms often maintain a fixed sized reservoir of the q largest (or smallest) values according to some metric. The functionality of this reservoir varies; e.g., in some cases, it is used to maintain a list of the heavy hitters [60], while [11] uses it to estimate the number of distinct keys. In general, the stored values are used to estimate specific properties of the traffic [11, 18, 38]. The best current implementations of maintaining the q largest values utilize standard data structures such as Heaps, SkipLists, and Balanced Search Trees. However, in all these methods the worst-case update time is logarithmic $O(\log q)$. Such a limitation is significant since several relevant applications require large values of q (e.g., values of $q = 10^6$ or $q = 10^7$ are reasonable for the network-wide sample of [18], and for Priority Sampling [37]). It turns out that one can improve the performance of many network applications by optimizing the algorithm for maintaining the largest q values. We emphasize that q-MAX is *not* a sampling, heavy hitters, or sketching algorithm; it is designed for the simple sub-task for finding the large values in a numbers stream, a building block which is used in all the above applications and many others.

Contribution: Our work is based on the observation that many algorithms maintain a reservoir of the q maximal values and list them upon demand and that this interface is slightly weaker than that of standard data structures such as heaps and SkipLists. We formalize the q-MAX problem and present algorithms for it that use the optimal $O(q)$ space, and operate in *constant* update time. Such algorithms asymptotically improve the update time of standard solutions. This implies a broad range impact on numerous algorithms in many domains. Examples include: (1) Network-wide heavy hitters [18]; (2) Priority Sampling [37]; (3) Priority Based

IMC '19, October 21–23, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6948-0/19/10…$15.00
https://doi.org/10.1145/3355369.3355569

Aggregation [38]; (4) finding the number of distinct flows [11]; (5) Bottom-k sketches [24]; (6) the Universal Monitoring sketch [60], and (7) the LRFU cache algorithm [55]. We also show lower bounds that indicate that our q-MAX algorithm is near optimal.

We further study an extension of the q-MAX problem to W-sized sliding windows. Disappointingly, such algorithms are known to require $\Omega(W)$ space, which is prohibitive for large windows. Instead, we consider similar definitions such as exponential decay (which enables us to implement the LRFU policy), and *slack windows* [14] whose size varies between W and $W(1-\tau)$ for some small parameter $\tau > 0$. We prove a lower bound for slack-windows and show space-optimal algorithms that update in constant time, and allow for efficient queries.

We asymptotically improve the update time of the state-of-the-art for numerous tasks. A notable exception is the Count Distinct problem, where existing techniques achieve constant update time using different methods. However, we asymptotically improve the query time for the Count Distinct problem [14] on slack windows. Additionally, our extension to network-wide heavy hitters yields the first routing oblivious network-wide algorithm for sliding windows.

We evaluate the throughput of the q-MAX algorithm compared to solutions based on standard open source implementations of SkipList and Heap. Our results indicate that the q-MAX algorithm is up to ×23 faster than these algorithms. We further study the impact of replacing a standard data structure algorithm with our q-MAX algorithm in Priority Sampling [37], Priority Based Aggregation [38], and in network-wide heavy hitters [18]. We evaluate the throughput on real packet traces and show a speedup of up to ×4 compared to a Heap-based implementation and up to ×12 compared to a SkipList based implementation. We also show that the throughput of our slack algorithm is attractive for a variety of configurations.

The actual expected speedup of software-based measurements was studied through an integration of the q-MAX algorithms within the Open vSwitch (OVS) stack [1]. Open vSwitch is a prevalent production quality virtual switch designed to enable massive network automation through programmatic extension. Our evaluation indicates that when q increases to values over 10^6, the Heap and SkipList implementations reduce the switch throughput while the q-MAX implementations keep up with the OVS well until $q = 10^7$.

We further show that the same scheme can also be used to enhance the performance of caching algorithms. Specifically, we show that an Exponential-Decay version of q-MAX enables constant time LRFU caches. Combining this result with the theoretical analysis, the experimental evaluation and the practical implementation of our algorithms in the network monitoring domain provide a solid indication for the impact and importance of q-MAX as an essential building block in delivering high performance streaming algorithms.

2 MOTIVATION AND RELATED WORK

We first outline five measurement methods that utilize the q-MAX pattern and outline the improvement. We first observe that the q-MAX pattern is different than the pattern of some heavy hitter algorithms that maintain the largest flows [13, 16, 35, 62, 67] as the size of flows changes whereas q-MAX finds the largest numbers in a stream of (fixed) numbers.

2.1 Priority Sampling and Priority Based Aggregation

Sampling is extensively used in network telemetry. It is used both as a mean reduce the bandwidth to the control [39, 69] and to accelerate software implementations of measurement algorithms [12]. Sampling can be done either for the number of packets or, more generally, according to some weight (e.g., the payload size). Weighted sampling provides more accurate visibility of the underlying byte-traffic. This is desired by applications such as traffic engineering and load balancing that attempt not to exceed the links' bandwidth limitations.

Priority Sampling [37] is an optimal weighted sampling technique. That is, Priority Sampling has smaller or equal variance compared to any other sampling method. Given a weighted stream $\langle x_1, w_1\rangle, \ldots$ of distinct keys (i.e., $i \neq j \implies x_i \neq x_j$) and a weight for each key. The goal is to produce a sample of k keys such that keys are sampled with a probability proportional to their weight. To do so, Priority Sampling assigns each key with the value: $\frac{w_i}{r}$, where r is uniformly selected in $[0, 1]$. The Priority sample contains the k keys with the maximal values. Priority Based Aggregation (PBA) [38], generalizes Priority Sampling so that keys can appear multiple times, and the goal is to sample each key with a probability proportional to its **total** weight. That is, flow x is sampled proportionally to its byte volume $w_x \triangleq \sum_{\langle x_i, w_i\rangle | x_i = x} w_i$. Here, w_i is the byte size of the i'th packet. This is a natural extension for network measurement that is often focused on per-flow aggregation [7, 49]. PBA also maintains a fixed sized reservoir that stores the items with maximal values. It differs from Priority Sampling manifests in the way values are calculated. Thus, our q-MAX algorithms improve the update time to a constant and extend these methods to slack windows.

2.2 Bottom-k Sketch

Bottom-k Sketches [24] summarize weighted streams and enable diverse statistical properties to be calculated on any **subset** of the stream Examples include averages, percentiles, variance and higher frequency moments. Bottom-k sketches can be merged at a central SDN controller, and thus achieve network-wide visibility of the traffic. The properties derived from Bottom-k Sketches are often used as an input for networking applications. For example, the tail latency of flows is a useful quality of service metric [54].

2.3 Count Distinct

Estimating the number of distinct items in a subset of the packets is often used by network applications. For example, identifying a source IP that contacts many distinct ports is used to identify port-scanners [50].

The work of [11] suggests several algorithms for estimating the number of distinct items in a stream. The fastest of these algorithms applies a hash function to each key and maintains a fixed size reservoir containing the smallest observed hash values. The number of distinct keys is statistically inferred from the maximal value in the reservoir. Intuitively, if the hashed values are numbers within $[0, 1]$, then a value smaller than 0.001 is encountered once per $1,000$ distinct values. Increasing the reservoir size reduces the variance of the method and reduces outliers. For example, when using a reservoir size of 2 the estimation is unaffected if a very small hash

value is encountered once very early during the measurement. The original implementation uses a Heap that works in logarithmic time, and our q-MAX solution allows for constant time updates. In sliding windows, our slack window q-MAX asymptotically improves query time compared to the best algorithm in this model [14].

2.4 Universal Monitoring

Universal Monitoring (UnivMon) [60] is a space-efficient technique for monitoring multiple useful metrics in a single unified sketch. The user then provides the desired metric at query time. UnivMon maintains multiple Count Sketch [22] instances for various substreams, and each sketch is also has a min-Heap to track its heavy hitters. The heavy hitters of the different substreams are then used to calculate a variety of measurement functions (e.g., the number of distinct flows, the distribution's entropy, and its frequency moments). We can reduce a logarithmic factor from the per-packet update time by replacing the heap with a q-MAX algorithm.

2.5 The DBM Monitoring method

The Dynamic Bucket Merge (DBM) algorithm [65] is designed to monitor bandwidth at granularities which are defined *at query time*. To that end, it dynamically partitions the measurement period into m buckets (where m controls the allocated memory, and thus, the error). When the number of buckets exceed m they merge the two buckets whose merging will result in the smallest error. To that end, they use a heap of all consecutive bucket pairs and update it for each arrival/merge. By replacing the heap with q-MAX, we can speed up this lookup and support faster updates.

2.6 Network-wide Heavy Hitters

The works of [18, 47, 49, 56] consider network-wide settings where multiple *Network Measurement Points (NMPs)* perform local measurements and report to a centralized controller that merges their reports to form a global view of the traffic. The work of [18] offers a passive measurement solution without assumptions on the routing or topology of the network. In their algorithm, multiple NMPs each sees some of the packets in the stream (but not all), and the same packet may traverse multiple NMPs. The goal is to detect the largest heavy hitter flows in the network without double-counting the same packet. Their algorithm assigns a hashed identifier to each packet, and each NMP stores the k packets with the minimal hash value. The controller merges reports from all NMPs to obtain the k globally minimal packets (according to hash value). Such a sample is a uniform sample from the entire network that can be used to identify the heavy hitter flows without double counting. The original implementation uses a heap to maintain the sample which results in logarithmic update complexity, which we improve to a constant with our q-MAX algorithm.

2.7 The LRFU Cache Policy

The LRFU cache policy [55] is among the most famous cache policies. It offers a spectrum of strategies that combine recency and frequency and achieves performance in many domains. In LRFU, the cache maintains an exponentially decaying score for each stored item, and the item with the smallest score is evicted. Typical LRFU implementation leverage heaps or priority queues that operate in logarithmic complexity. Unfortunately, the logarithmic update complexity makes LRFU impractical in many domains [61]. Our work achieves constant update time by employing our Exponential Decay q-MAX algorithm.

2.8 Other Related Work

In principle, while our work touches multiple applications in diverse fields. It is far from inclusive, and only touch a small portion of the algorithms in each area. For example, many works do not use the q-MAX interface, and thus cannot be improved by our algorithm. Examples include passive measurement works that manage a flow cache that monitors a subset of the flows [13, 27, 30, 33, 53, 62]. Other works, target P4 programmable switches whose programming model makes identifying the minimal item challenging [64]. Thus, such algorithms avoid patterns that require reading a non-constant portion of the memory [9, 39, 56, 64, 69], and to the best of our knowledge none of them use the q-MAX pattern. Further study is required to determine if the q-MAX approach itself can be modified for such switches. Finally,most works that use shared counters [22, 28, 31] do not follow the q-MAX interface. In principle, such algorithms often boil down to counter arrays that are maintained for multiple items. The exception is when such algorithms maintain a list of heavy hitters [60]. In such cases, a heap or a skiplist are often used to maintain the list, and q-MAX can replace these data structure to increase the speed. Thus, within the bigger perspective, we observe that the$q$-MAX is not very common, but there are important applications that follow it.

3 QUANTIFYING THE POTENTIAL SPEEDUP

To estimate the possible speedup obtainable by replacing existing data structures with an optimized q-MAX implementation, we used a profiler to measure the amount of time spent on the data structure update for several algorithms on a 150M-sized trace using Heap and SkipList. The experiment result shows that, for $q = 10^4$, Priority Sampling spends 50-58% of the time in updating the data structure. Similarly, in Network-wide heavy hitters and Priority Based Aggregation, it takes 22-28% and 18-19% accordingly. The bottleneck becomes worse for larger q, with up to 96% of the time spent on updating the data structures for $q = 10^7$. Further, the recent work of [59] found the heap update to be a significant bottleneck in sketching algorithms and the Universal Monitoring sketch [60] in particular. We conclude that optimizing the q-MAX data structure may significantly accelerate many algorithms for which the current method is a bottleneck.

4 THE q-MAX PROBLEM

In this section we study the q-MAX problem on intervals and sliding windows. We first define the streaming model and the q-MAX problem in Section 4.1, and then study intervals in Section 4.2, and sliding windows in Section 4.3.

4.1 Model and Definitions

A stream $\mathfrak{S}$ is a list of items of the form (id, val), where at each step a new item is appended to $\mathfrak{S}$. $id \in \mathcal{U}$ is an identifier taken from a domain $\mathcal{U}$, and val is a value taken from a fully ordered domain (e.g., real or natural numbers). For sliding windows, we denote the

window size by W and by $\mathfrak{S}^W$ a list of the last W items in $\mathfrak{S}$. Given a parameter $\tau \in [0,1]$, a W,τ-slack window is a W'-sized sliding window whose size varies between $W(1-\tau)$ and W. That is, it is $\mathfrak{S}^{W'}$ for some W' s.t. $W(1-\tau) \leq W' \leq W$. Similarly, a W, 0-slack window is simply a (W-sized) sliding window. We assume that comparison requires $O(1)$ time and that each item requires a single space unit.

The q-MAX problem is about processing $\mathfrak{S}$ and upon query listing the q maximal items in the entire stream. The (W,q)-max problem requires listing the q maximal items over a W sized sliding window ($\mathfrak{S}^W$). Finally, the (W,τ,q)-max problem is about listing the q maximal items over a W,τ-slack window. Here, $q \in \mathbb{N}^+$ is the number of maximal items to list. A q-MAX algorithm supports two methods: *update* and *query*. The update method reports a new item and returns the replaced one (which is not among the q largest items). This item can either be one of the stored items or the current one. The query method lists the q maximal items according to value. This minimal interface is sufficient to implement the previously mentioned algorithms and can be realized with a constant update complexity. Current implementations use standard data structures that can only be implemented in logarithmic complexity.

4.2 q-MAX on Intervals

4.2.1 q-MAX algorithms (upper bounds). Our interval algorithm requires $\lceil q(1+\gamma)\rceil$ space and operates in $O(1)$ worst-case time for any constant γ. Here, γ is a space-time tradeoff parameter; the larger γ is, the more speedup we get. Intuitively, our algorithm is based upon the observation that finding a percentile can be done in linear time [21]. Instead of maintaining an ordered list of q items, we maintain a larger list of $\lceil q(1+\gamma)\rceil$ items. Then, we periodically find a percentile where q items are above it, retain the q largest items, and remove the rest. This operates in $O(1)$ *amortized* complexity.

Intuitively, we improve the *worst case* update complexity to a constant by performing the percentile calculation in small fixed sized steps per each update. Algorithm 1 provides pseudo-code for this process, and we explain it below.

We maintain an array of $\lceil q(1+\gamma)\rceil$ items (A) for some constant $\gamma > 0$. A is split into two regions, pointed to by S_1 and S_2. The computation takes place in *iterations*. At the beginning of each iteration, the subarray pointed to by S_1 contains $q(1+\gamma/2)$ items that include the largest q. S_2 points to a $q\gamma/2$ sized subarray of items that are guaranteed *not* to be among the top-q and thus are logically deleted. We interchangeably refer to S_1 and S_2 as sets that contain the elements in the subarrays they point to. The algorithm also maintains a quantity Ψ, which is initialized to $-\infty$, that denotes a lower bound on the q'th largest item that is currently in A. As items arrive, those that are smaller than Ψ are surely not among the largest q and are therefore discarded. The processing of every item larger than Ψ is called a *step*. Each iteration lasts exactly $q\gamma/2$ *steps*, throughout which the arriving elements are inserted into the array part of S_2. The algorithm uses a global variable called *steps* that tracks the progress of the iteration and denotes the number of items inserted to S_2. During the first $q\gamma/4$ *steps* of the iteration, and while inserting the (larger than Ψ) arriving elements to S_2, we find the percentile for which there are q items larger or equal to in S_1. This is done by breaking the execution of the Select algorithm [21], which requires $O(q)$ operations for finding a percentile in an $O(q)$-sized array, into $q\gamma/4$. Each such sequence of operations is called a SelectStep(). Since the entire Select procedure requires $O(q)$ time, we make $O(\gamma^{-1})$ CPU operations per step, which is constant for any fixed γ. At each step, after the insertion to S_2, we perform a single SelectStep(); therefore, after these $q\gamma/4$ *steps* we are done computing the percentile, and set it as the new value for Ψ.

Next follows $q\gamma/4$ *steps* during each we perform a PivotStep() in addition to inserting an item to S_2. Here, we break a *pivoting* operation, which brings the largest q items in S_1 (using the threshold Ψ we computed) to the middle of A. As in the percentile computation, pivoting an $O(q)$-sized array takes $O(q)$ and each PivotStep() executes $O(1)$ operations for fixed γ. This concludes the $q\gamma/2$ *steps* of the iteration. At this point, we have the largest q items of S_1 in the middle and thus we are guaranteed that the remaining $q\gamma/2$ items of S_1 are not among the maximal in the stream. At this point, we change the positions of S_1 and S_2, and a new iteration begins.

Finally, our algorithm serves queries in a straightforward manner; it computes the percentile of A that corresponds to the q'th largest item and then makes another pass on A to report the maximal items. Figure 1 illustrates this algorithm. The main result for q-MAX is:

Algorithm 1 q-MAX

```
    Initialization:
    A ← q(1+γ)-sized array, initialized to −∞
    Ψ ← −∞; s1 ← 0; s2 ← q
1:  function ADD(id, val)
2:      if val > Ψ then                          ▷ If larger than the admission filter
3:          A[s2 + steps] ← (id, val)
4:          DeamortizedStep()                    ▷ run O(γ^−1) operations
5:  function DeamortizedStep()
6:      steps ← steps + 1
7:      if steps ≤ qγ/4 then
8:          SelectStep(s1, q(1 + γ/2))
9:          return
10:     Ψ ← q-Percentile()                       ▷ The return value of the Select
11:     if steps ≤ qγ/2 then
12:         PivotStep(s1, Ψ, q(1 + γ/2))
13:         return
14:     s1 ← qγ/2 − s1
15:     s2 ← q − s2
16:     steps = 0
17: function Query()
18:     return q-Largest(A)
```

Theorem 1. *For any $\gamma > 0$, there exists a q-MAX algorithm with $\lceil q(1+\gamma)\rceil$ space and $O(\gamma^{-1})$ update complexity.*

We now analyze the expected number of updates, for a fixed size stream and fixed input distribution, in which the items are larger than the threshold Ψ. Since smaller items are immediately discarded, such updates are faster than those that change A. This is also apparent from our evaluation (see Section 6 and specifically Figure 10) that shows that q-MAX becomes faster as the trace prolongs. Thus, in the following, we assume that each item in $\mathfrak{S}$ is sampled independently from a fixed distribution $\mathcal{D}$. Without loss of generality, we assume that the items are i.i.d. distributed real numbers.

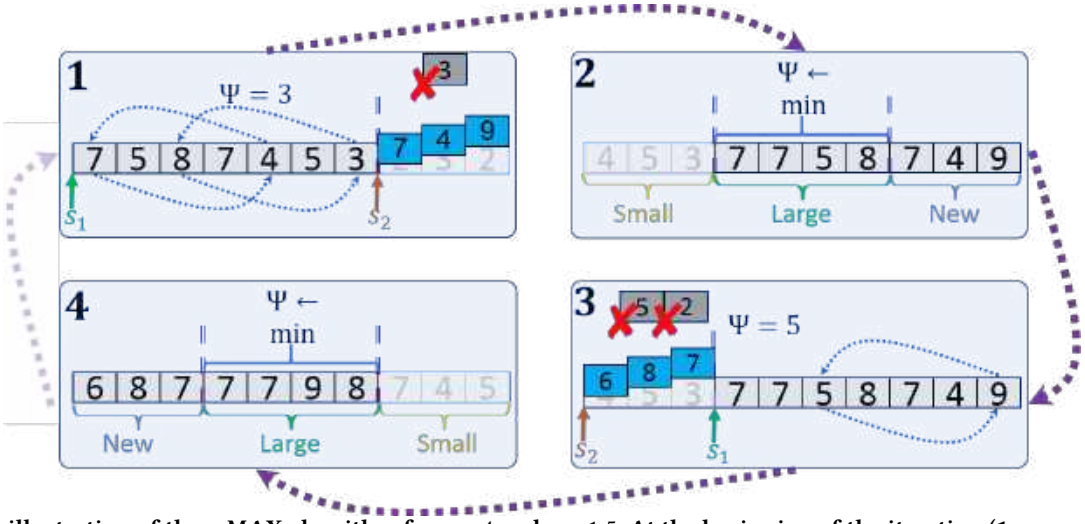


Figure 1: An illustration of the q-MAX algorithm for $q = 4$ and $\gamma = 1.5$. At the beginning of the iteration (1, upper left corner), the threshold is $\Psi = 3$, and only items larger than Ψ are admitted. Throughout the next $q\gamma/2$ insertions (*steps*), admitted items are inserted next to S_2. During the first $q\gamma/4$ insertions, we perform a Select operation that finds the percentile (Ψ) in the $q(1+\gamma/2)$-sized subarray pointed to by S_1 such that there are q items of value at least Ψ. On the remaining $q\gamma/4$ insertions we *pivot* the array such that the q largest among S_1 are placed in the middle. After these *steps* (Subfig 2), the leftmost $q\gamma/2$ items (Small) are effectively deleted, and we are guaranteed that the largest q are among the middle (Large) and rightmost (New) values. The procedure now repeats (Subfig 3) where S_2 points to the rightmost $q\gamma/2$ locations and S_1 to the rest. Finally, after additional $q\gamma/2$ larger than Ψ we delete the rightmost ("Small" on Subfig 4) items and a new iteration begins.

Theorem 2. *Let $\mathcal{D}$ be some distribution on the real numbers and assume that $|\mathfrak{S}| \geq q$. The expected number of updates made by Algorithm 1 is bounded by $O(q\log(|\mathfrak{S}|/q))$.*

Proof. We first claim that the i'th item (x_i) is added with a probability of at most $\min\{1, 2q/i\}$ for any $\gamma \leq 2$. Recall that at each iteration we compute $\mathfrak{m}$ – the q'th largest item in S_2. This means that at any point during the run, $\mathfrak{m}$ is guaranteed to be one of the $q + \lfloor q\gamma/2 \rfloor \leq 2q$ largest items in $\mathfrak{S}$. Denote by E_i the event that x_i is strictly larger than $\mathfrak{m}$. As items are drawn independently from $\mathcal{D}$, it follows that $\Pr[E_i] \leq 2q/i$ (and clearly, $\Pr[E_i] \leq 1$). Thus, the expected number of updates to Algorithm 1 is bounded by

$$\sum_{i\leq|\mathfrak{S}|}\Pr[E_i] \leq \sum_{i\leq|\mathfrak{S}|}\min\{1, 2q/i\} \leq \sum_{i=1}^{2q}1 + \sum_{2q}^{|\mathfrak{S}|}2q/i$$
$$= 2q(1+\ln(|\mathfrak{S}|/q) + O(1)) = O(q\log(|\mathfrak{S}|/q)),$$

where the middle equality follows as $H_z \triangleq \sum_{i=1}^{z}1/i = \ln(z) + \Theta(1)$ is a bound on the i'th harmonic number. □

4.2.2 q-MAX lower bounds. Next, we study lower bounds for the q-MAX problem. We do so by showing a reduction between q-MAX and integer sorting. Namely, we prove that a q-MAX solution that uses $q+\Psi$ space and updates in $O(\phi)$ time implies an algorithm that sorts an n-sized integer array in $O(n\Psi)$ space and $O(n\Psi\phi)$ time. This immediately imply that any sorting-based q-MAX that uses $q+O(1)$ space has $\Omega(\log q)$ update time using known lower bounds. The current state of the art for deterministic sorting algorithms runs in $O(n\log\log n)$ [8, 45], and for randomized sorting, an $O(n\sqrt{\log\log n})$ time algorithm is known [46]. This means that any q-MAX solution that updates in $O(1)$ time and has $(q + o(\log\log q))$-space implies an improvement for deterministic integer sorting, and any randomized algorithm with $(q + o(\sqrt{\log\log q}))$-space results in improving the state of the art for randomized sorting algorithms. Further, a deterministic $q + O(1)$ space solution with $o(\log\log q)$ update time or a randomized algorithm with $q + O(1)$ space and $o(\sqrt{\log\log q})$ update time also means faster integer sorting. Thus, we conclude that Algorithm 1 (that uses $q(1+\gamma)$ space and updates in worst case $O(1)$ time for constant γ) is near optimal, unless there exist better integer sorting algorithms.

Theorem 3. *The existence of a deterministic (randomized) q-MAX algorithm with $q + \Psi$ space that updates in $O(\phi)$ time implies a deterministic (randomized) integer sorting algorithm that sorts an n-sized integer array in $O(n\Psi\phi)$ time.*

Proof. Let $A = \langle a_1, \ldots, a_n\rangle$ denote the n-sized integer array we wish to sort. Consider the sequence $\Lambda \triangleq \langle a_1, a_1, \ldots, a_1, a_2, a_2 \ldots, a_2, \ldots, a_n, a_n, \ldots, a_n\rangle$ in which the i'th element is $\Lambda_i = a_{\lfloor i/\Psi \rfloor}$. Intuitively, we use this sequence to infer the value of the next smallest item in the array given the element discarded from the q-MAX solution. Notice that Λ is of size $|\Lambda| = n\Psi$. For $q = n\Psi$, We initialize a q-MAX solution M and insert to it the elements of Λ one at a time. Next, let $\mathfrak{M} \triangleq 1 + \max\{a_i \in A\}$ be an integer that is strictly larger than any item in A. For $i = 1, \ldots, n$ and $j = 1, \ldots, \Psi$, we insert $\mathfrak{M}$ into M. At the end of each outer-loop iteration (every Ψ insertions), we look at the value discarded by M and use it to infer the next smaller integer as explained above. After this operation is completed we successfully recovered the sorted order of the array A. Since we fed M with $2q = 2n\Psi$ elements, the overall time required is $O(n\Psi\phi)$. For convenience, we provide the pseudo code of this reduction in Algorithm 2. □

Algorithm 2 Integer sorting an n-sized array A given a q-MAX solution

```
1: function Sort(Array A)
2:     M ← (nΨ)-Max.init()
3:     for i = 1, ..., n do
4:         for j = 1, ..., Ψ do
5:             M.add(A_i)
6:     𝔐 ≜ 1 + max {a_i ∈ A}                ▷ A bound on items in A
7:     for i = 1, ..., n do
8:         for j = 1, ..., Ψ do
9:             M.add(𝔐)
10:        Report the last replaced item as the next smallest element in A
```

4.3 q-MAX on Sliding Windows

In this section, we study the extension of q-MAX to sliding windows. We start by explaining a known lower bound of $\Omega(W)$ items for q-MAX on a sliding window in Section 4.3.1. Then, we show a lower bound of $\Omega\left(\min\left\{W, q\cdot\tau^{-1}\right\}\right)$ items for τ, W slack windows.

4.3.1 Infeasibility of q-MAX on sliding windows. Unfortunately, it is known that even for $q = 1$ (finding the maximum over a sliding window) and a multiplicative approximation, any such algorithm requires $\Omega(W)$ space [32]. Thus, any sliding window q-MAX algorithm would store $\Omega(W)$ items at the worst case which is prohibitively expensive for large windows. This motivates our focus on W, τ-slack windows that do allow efficient q-MAX algorithms. Recall that in this model, the goal is to return the q maximal items with respect to some window whose size varies between $W(1-\tau)$ and W.

4.3.2 Lower bound for q-MAX on slack windows. We now provide a lower bound of $\Omega\left(\min\left\{W, q\cdot\tau^{-1}\right\}\right)$ items for τ, W windows. Such a lower bound is interesting when $q\cdot\tau^{-1} = o(W)$. For constant τ the bound is $O(q)$ items which is optimal.

Theorem 4. *Any algorithm that solves (W, τ, q)-max must store $\Omega\left(\min\left\{W, q\cdot\tau^{-1}\right\}\right)$ items.*

Proof. Clearly, one could store the entire W items window and solve the problem. Thus, we hereafter assume that $q\cdot\tau^{-1} = o(W)$ and show an $\Omega(q\cdot\tau^{-1})$ lower bound. Let $Q = \{x_0, \ldots, x_z\}$, for $z = \tau^{-1}/2\cdot q$, be a set of $(z+1)$ distinct values such that $x_0 \succ x_1 \succ \ldots \succ x_z$. Consider the sequence:

$$x_z^{2W\tau-q} x_1 x_2 \cdots x_q x_z^{2W\tau-q} x_{q+1} x_{q+2} \cdots x_{2q} \cdots x_z^{2W\tau-q} \cdots x_{z-1} x_z,$$

which contains $\tau^{-1}/2$ parts, each starts with $(2W\tau - q)$ occurrences of x_z followed by the next q items of Q.

We claim that the algorithm must keep in memory all items in $Q\setminus\{x_{z+1}\}$. Assume by contradiction that there exists $i \in \{0, \ldots, z-1\}$ such that the algorithm does not store the item x_i. Next, consider the case where this initial sequence is followed by $\lfloor i/q\rfloor\cdot(2W\tau)$ occurrences of x_{z+1}. This leads to a contradiction as x_i is now among the q-largest in *any* window of size between W and $W + W\tau$. Since the algorithm does not have x_i in memory, it cannot list the largest q items and therefore fails. Thus, the algorithm must store $|Q| - 1$ items, which requires $\Omega\left(\min\left\{W, q\cdot\tau^{-1}\right\}\right)$ items. □

We note that the above proof assumes that the domain of the input keys is of size at least $z+1$. If the size of the domain is $D = O(\tau^{-1}q)$, one could maintain the set of items that are candidates for the top-q in some future window. To make use of the slack for reducing the space, one may use $O(\tau^{-1})$ bits per item and encode just the approximate timestamp (within a $W\tau$-additive error) in which the item has last appeared. This approach is a variant of the List of Possible Maxima algorithm presented in [41] and requires $O(D\log\tau^{-1})$ memory bits. However, networking applications often have $D \gg \tau^{-1}q$ (e.g., D can be all 2^{64} (srcip,dstip) pairs, or all possible 5-tuples), which makes this approach infeasible.

4.3.3 q-MAX algorithms for slack windows (Upper bound). First, we design a simple algorithm that solves the problem in optimal space and $O(1)$ update time, when D is arbitrarily large. We partition the stream into consecutive $W\tau$-sized blocks and maintain a q-MAX instance for each block. To reduce the space, we keep the instances in a τ^{-1}-sized cyclic buffer and only store q-MAX instances that are contained in the current window. Whenever a block ends, we reset the q-MAX of the oldest block, and every item updates just the q-MAX of its block. Therefore, we get a $O(q\cdot\tau^{-1})$ space algorithm that updates in constant time. For queries, we propose the Partial(t_1, t_2) procedure that merges all blocks between t_1 and t_2. Here, a merge refers to querying the underlying q-MAX instances and adding their top items into a q-MAX (denoted R) that is dedicated to the results. Finally, by querying R we get the largest q items in the interval (t_1, t_2). For finding the top q items over the entire window we perform Partial$(0, n-1)$, where $n = \tau^{-1}$ is the number of blocks. We summarize the asymptotics of the method, whose pseudo code appears in Algorithm 3, in the following theorem.

Algorithm 3 Basic-(q, W, τ)-max

```
    Initialization:
    n ← τ^{-1}                                  ▷ Number of blocks
    s ← W/n                                     ▷ Block Size
    ∀J = 0, ... n − 1 : B[J] ← q-max()
    i ← 0
1:  function Add(id, val)
2:      B[⌊i/s⌋].ADD()
3:      i ← (i + 1) mod W
4:      if (i mod s) = 0 then                   ▷ End of a block
5:          B[i/s].Reset()                      ▷ Reset the oldest q-MAX
6:  function Query()
7:      return Partial(0, n − 1)
8:  function Partial(t_1, t_2)                  ▷ q-MAX over blocks t_1, ..., t_2
9:      R ← q-max()                             ▷ Result q-MAX
10:     for J = t_1, ..., t_2 do
11:         Merge(R, B[J])                      ▷ Merges into R
12:     return R.Query()
13: procedure Merge(R, R_2)
14:     for (id, val) ∈ R_2.Query() do
15:         R.Add(id, val)
```

Theorem 5. *Algorithm 3 computes q-MAX over τ-slack windows; uses $O(q\cdot\tau^{-1})$ space; makes updates in $O(1)$ time; and answers queries in $O(q\cdot\tau^{-1})$ time.*

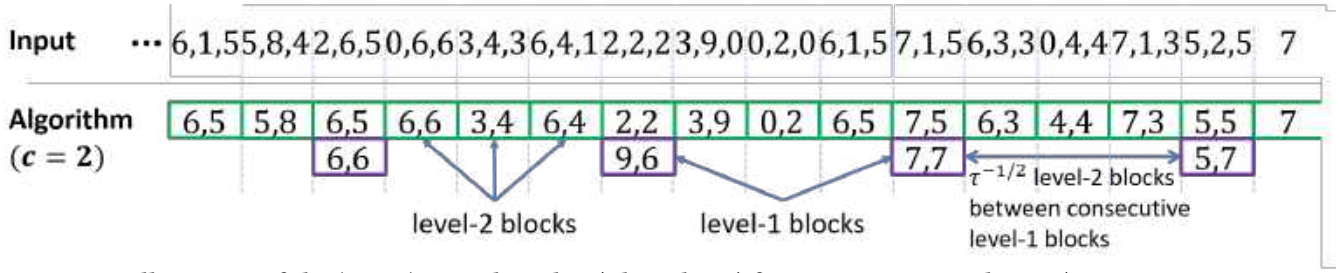


Figure 2: An illustration of the (q, W, τ)-max algorithm (Algorithm 4) for $q = 2$, $W = 48$ and $\tau = 1/16$. Here, $c = 2$ instances of Algorithm 3 are used. The first, illustrated in green, stores a 2-max for each 3-sized block and the second, in purple, stores a 2-max for each 12-sized block. The most recent block contains only 7, and thus $W' = 46$ is the size of the slack window. To obtain the q-maximal items over that window, using only the green algorithm one would need to merge all sixteen green blocks. Using the two algorithms, we can do so by merging just six blocks: the four (level-1) blocks and the two oldest green blocks (level-3) block.

While this algorithm has optimal space and update time, its query time is $O(q(t_2 - t_2)) = O(q \cdot \tau^{-1})$ which is too slow when τ is small. We now develop methods that reduce the query time dependence on $O(\tau^{-1})$ without asymptotically increasing the space or update time.

To that end, we use a parameter $c \in \mathbb{N}^+$ as a performance tradeoff parameter. When c is large, queries are faster, but our $O(c)$ update time is slower. This is achieved by maintaining c separate instances of Algorithm 3 with varying values of the slack parameter. Namely, for the ℓ'th instance we use slack of $\tau^{\ell/c}$. This means that ℓ'th instance blocks, hereafter referred to as ℓ-blocks or level-ℓ blocks, are $n_\ell = \tau^{-(\ell/c)}$ in number and each reflects W/n_ℓ input items. Whenever an item arrives, it updates the q-MAX instances of all c levels. The tricky part is the query, where we decompose the window into non-overlapping blocks of different levels. Specifically, the window can be expressed as the union of at most $\tau^{-1/c}$ blocks of each of the c levels. For answering a query, we first merge all level-1 blocks using the Query procedure of Algorithm 3. Intuitively, this covers all but at most a $\tau^{-(c-\ell)/c}$ fraction of the items, which can be covered using higher level blocks. For the remaining levels, we use the Partial query to merge the additionally required blocks. Our solution, whose pseudo code is given in Algorithm 4, is illustrated in Figure 2.

We now state the asymptotic behavior of Algorithm 4. Observe that for any *constant* c the result is a space optimal algorithm with constant update time. In contrast, setting $c = \lceil \log \tau^{-1} \rceil$ gives an algorithm with $O(\log \tau^{-1})$ update time and $O(q \log \tau^{-1})$ query time.

Theorem 6. *For any constant $c \in \mathbb{N}^+$, Algorithm 4 solves q-MAX over τ-slack windows; uses $O(q \cdot \tau^{-1})$ space; updates in $O(c)$ time; and answers queries in $O(q \cdot c \cdot \tau^{-1/c})$ time.*

Algorithm 4 provides two options for the user – either have a constant update time and a query time overhead that is polynomial in τ^{-1}, or (by setting $c = \log \tau^{-1}$) have logarithmic update time and $O(q \log \tau^{-1})$ query time. However, it is desirable to get the faster query time without giving up on the constant time updates. Intuitively, the above algorithm is wasteful in the sense that many small items update all c levels, thereby implying an $\Omega(c)$ update time. Instead, we can update just a single q-MAX instance for most packets. To that end, we employ a single q-MAX that resets every $W\tau$ items. Before its reset, we query for the largest q items of the block and feed them into all the c underlying instances of Algorithm 3. That is, we only touch the c levels once every $W\tau$ items, adding the largest q in this interval. The overall work in every such consecutive sequence of $W\tau$ items is then $O(W\tau + qc)$, which means an (amortized) update time of $O(1 + q \cdot \tau^{-1} c/W)$, that can be deamortized without affecting the asymptotic query time. Finally, recall that $q \cdot \tau^{-1} = o(W)$, which implies an asymptotic time improvement. Further, if $W = \Omega(q \cdot \tau^{-1} \log \tau^{-1})$ we set $c = \log \tau^{-1}$ to conclude:

Theorem 7. *If $W = \Omega(q \cdot \tau^{-1} \log \tau^{-1})$, there exists a q-MAX algorithm for τ-slack windows that uses $O(q \cdot \tau^{-1})$ space, updates in constant time, and serves queries in $O(q \log \tau^{-1})$ time.*

4.3.4 Exact window network-wide heavy hitters. We now show how our slack window q-MAX can be used for finding network-wide routing-oblivious heavy hitters [18] over an *exact window*. That is, while computing the maximum over an exact window requires $\Omega(W)$ space, finding the heavy hitters does not. Intuitively, for heavy hitters, we are allowed an additive error in the frequency estimations, which can be partially traded for a slack in the window size. For example, consider finding all flows with frequencies larger than 100 within the last 1000 items. If we compute a q-MAX over a slack window whose size varies between 990 and 1000, and estimate frequencies within an additive error of 10, we can return all items whose size estimate is at least 80 and have no false negatives.

First, we define a "sliding window" in distributed settings. Defining the window size in time makes more sense than defining it in packets. Thus, we assume that packets are associated with *timestamps* and define the window in time units. For example, consider a window size of 24 hours; if $\tau = \frac{1}{24}$, we get a slack window that varies between 23 and 24 hours.

We use our slack solutions (Algorithm 3 or Algorithm 4) for $\tau = \varepsilon/2$ and utilize the estimation mechanism of [18] with a guarantee of $(\varepsilon/2, \delta)$ on all measurement points. With probability $1 - \delta$, we get an error of at most $W\varepsilon/2$. The result is a sliding window algorithm with an accuracy guarantee of (ε, δ), where the additional error comes from monitoring a slack window which differs from the original window by at most $W\tau = W\varepsilon/2$ items. The following theorem formalizes our result for sliding window based network-wide heavy hitters.

Algorithm 4 (q, W, τ)-max

Initialization: $\forall \ell = 1, \ldots c$:
  $n_\ell \leftarrow \tau^{-(\ell/c)}$ ▷ Number of ℓ-blocks
  $\mathfrak{n}_\ell \leftarrow \tau^{-(c-\ell)/c}$ ▷ #c-blocks in an ℓ-block
  $C[\ell] \leftarrow$ Basic-$(q, W, \tau^{\ell/c})$-max ▷ Level-ℓ instance
  $\mathfrak{s}_\ell \leftarrow W/n_\ell$ ▷ ℓ-Block Size
$i \leftarrow 0$
1: **function** ADD(id, val)
2: **for** $\ell = 1, \ldots, c$ **do**
3: $C[\ell]$.ADD()
4: $i \leftarrow (i+1) \mod W$ ▷ Used for answering queries
5: **function** QUERY()
6: $R \leftarrow q$-max() ▷ Result Instance
 ▷ First, add the largest items among the 1-blocks
7: **for** $(id, val) \in C[1]$.QUERY() **do**
8: R.ADD(id, val)
9: $U \leftarrow \mathfrak{n}_1 - (\lfloor i/\mathfrak{s}_1 \rfloor \mod \mathfrak{n}_1)$ ▷ #uncovered c-blocks
10: $\ell \leftarrow 1$
11: **while** $U > 0$ **do** ▷ While not all blocks were covered
12: $\ell \leftarrow \ell + 1$
13: $m_\ell \leftarrow \lfloor U/\mathfrak{n}_\ell \rfloor$ ▷ Number of ℓ-blocks to merge
14: $U \leftarrow (U \mod \mathfrak{n}_\ell)$ ▷ Remaining c-blocks to cover
15: $\mathfrak{I}_\ell \leftarrow \lfloor i/\mathfrak{s}_\ell \rfloor$ ▷ Current ℓ-block index
16: $first_\ell \leftarrow (\mathfrak{I}_\ell + 1) \mod n_\ell$
17: $last_\ell \leftarrow (\mathfrak{I}_\ell + m_\ell + 1) \mod n_\ell$
18: **for** $(id, val) \in C[\ell]$.PARTIAL($first_\ell$, $last_\ell$) **do**
19: R.ADD(id, val)
20: **return** R.QUERY()

THEOREM 8. *If $W = \Omega(q\varepsilon^{-1}\log\varepsilon^{-1})$, there exists an algorithm for W-sized window network-wide heavy hitters, that updates in constant time, serves queries in $O\left(\varepsilon^{-2}\log\delta^{-1}\log\varepsilon^{-1}\right)$ time, and uses $O\left(\varepsilon^{-3}\log\delta^{-1}\right)$ space.*

5 EXPONENTIAL-DECAY q-MAX

As recent data is often more important than old one, different aging mechanisms associate current elements with higher weights than old ones [25]. Sliding windows is one model which gives equal weights to the last W elements and zero weight to older ones. In this section, we explore the exponential decay aging model [26] for q-MAX. Specifically, for an *aging parameter* $c \in (0, 1]$, the *weight* of an item (id_i, val_i) that arrive at time i is $weight_i \triangleq val \cdot c^{t-i}$ where t is the current time. That is, whenever a new element arrives the weight of all previous items is decreased by a factor of c. This generalizes the standard q-MAX problem which is the special case of $c = 1$ (which means all items retain their weight throughout the measurement). The goal of Exponential-Decay q-MAX is to report the q elements with the largest weight at the time of the query.

We propose to solve Exponential-Decay q-MAX by a reduction to the standard q-MAX algorithm. Specifically, consider the stream $\mathfrak{S}_1 = (id_0, val_0), (id_1, val_1), \ldots, (id_i, val_i)$, which is given as input to the Exponential-Decay q-MAX. Instead of aging elements as time goes, one can feed the modified stream $\mathfrak{S}_2 = (id_0, val_0 \cdot c^{-0}), (id_1, val_1 \cdot c^{-1}), \ldots, (id_i, val_i \cdot c^{-i})$, into a standard q-MAX. Indeed, item i will have a larger weight than item $j > i$ at time t if and only if $weight_i > weight_j \iff val_i \cdot c^{t-i} > val_j \cdot c^{t-j} \iff val_i \cdot c^{-i} > val_j \cdot c^{-j}$. However, this simplistic reduction may be numerically unstable; specifically, computing c^{-i} may not be accurately representable if one uses standard floating-point operations (e.g., consider $c = 0.9$ and $i = 100M$). Instead, we consider applying the logarithm function on the item weights and when processing item (id_i, val_i) adding (id_i, val_i') to a standard q-MAX solution where $val_i' = \log(val_i \cdot c^{-i}) = \log(val_i) - i\log c$. By the monotonicity of the logarithm we have that $weight_i > weight_j \iff val_i \cdot c^{-i} > val_j \cdot c^{-j} \iff val_i' > val_j'$.

5.1 Constant Time LRFU Caches

We now show that our Exponential-Decay q-MAX enables constant time LRFU caches. Caching is a fundamental technique in computer science, where a small portion of some data is kept in memory with faster access. Realistic access patterns are not entirely random and thus significant performance gains are often obtained even with small caches.

The seminal work of [55], presents a spectrum of cache policies that mix the two most fundamental heuristics in caching. *Recency* - that speculates that recently accessed items are more likely to be accessed in the future, and *Frequency* that speculates that frequently accessed items are more likely to be accessed in the future.

An LRFU cache maintains a fixed number of entries, and each entry is associated with a score. When needed, the cache replaces an item with the minimal score with a new item. The score of item x, at time t is: $\sum_{i|id_i=x} c^{t-i}$, where $id_i = x$ means that the i'th request was for x. The parameter $c \in (0, 1)$ balances between Recency and Frequency. Adapting q-MAX for LRFU is not straightforward as items' score is an aggregation over multiple requests.

Our main result here is a constant (per-element) time caching algorithm with a cache of size $q(1+\gamma)$ that guarantees to store the q heaviest LRFU elements (just like a q-sized LRFU cache). The hit ratio of our LRFU algorithm is very similar to the hit ratios of the original q-sized LRFU cache.

We start with a simpler amortized constant time algorithm. Intuitively, the solution is similar to the Exponential-Decay q-MAX from the previous section except for two changes: (i) all weights in LRFU are 1; this means that computing the weight of element $-i\log c$ is faster and can be done in a single floating-point multiplication by storing the value of $\log c$; and (ii) there can now be multiple requests *for the same item*. Therefore, we need to maintain an aggregate between items and their overall weight. To do so, we denote the log-weights by w_1, w_2 the merged log-weight is $\log(e^{w_1} + e^{w_2})$ which may not be numerically feasible to compute. Instead, assuming without loss of generality that $w_1 > w_2$, we set the new weight to $w_1 + \log(1 + e^{w_2 - w_1})$. To conclude, we add items to q-MAX similarly to the above until the $q(1+\gamma)$-sized array is filled. We then merge all elements that have multiple entries using the above procedure; compute the q'th largest element; pivot the array; and start a new iteration with $q \cdot \gamma$ free slots (from the elements that had the lowest log-weight). Since this maintenance operation takes $O(q)$ time, and it is done once every $(q \cdot \gamma/2)$-elements iteration, the amortized complexity is $O(\gamma^{-1})$ which is constant for fixed γ.

Next, we design a deamortization procedure to achieve *worst case* constant update time. The difficulty is that the Select algorithm that runs the percentile algorithm may fail if we update the array

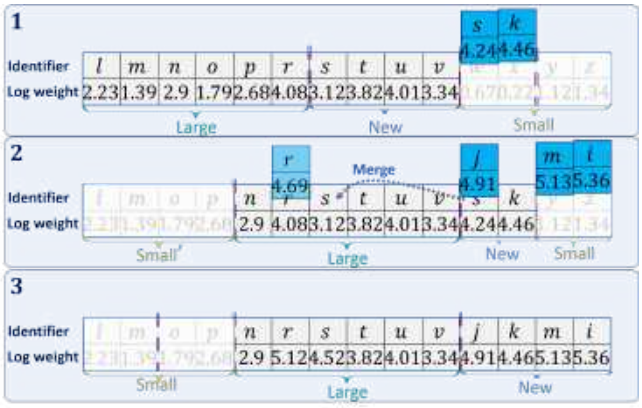


Figure 3: Illustration of our worst-case constant time LRFU algorithm for $c = 0.8$. In the first part, we compute the q'th largest element in the union of Large and New and pivot this array part. We do it in a deamortized way while processing new requests (s and k). Notice that some requests may be for elements that are in the cache (s in this example). After the first part is done, we have computed $q\gamma/2$ elements (Small') that are not among the q with the highest score, but may have duplicates (e.g., s). During the next part, we merge the duplicated counters (s) while processing new requests (r, j, m, and i). If an element with a counter is requested (r in the example) we increase its counter. Finally, in Part 3, we are again left with a $q(1+\gamma/2)$-sized array with no duplicates (Large+New) and $q\gamma/2$ elements that are guaranteed not to be the largest; then a new iteration begins.

during its operation. On the other hand, if we wait until the array is filled then we will not find the small elements in time to start a new iteration. To circumvent this issue, we now break each iteration into three intervals, *Large*, *Small*, and *New*. At the beginning of each iteration, we guarantee that the elements in Small are not among the heaviest q, similarly to Algorithm 1. During the first $q \cdot \gamma/4$ requests, we insert all elements into the array *even if they exist in the cache*. This enables us to keep the Large and New constant, as we pivot them as in the standard q-MAX. Unlike Algorithm 1, at the end of this part we may have duplicates which we merge during the next $q \cdot \gamma/4$ requests. Each merge requires $O(1)$ time and releases a counter from New that can be reused in the current iteration. At this time, we can safely update the Large counters. Therefore, after these $q \cdot \gamma/2$ requests we have a new Small array of elements that we can evict from the cache, and a new iteration begins. This algorithm is illustrated in Figure 3. Our LRFU algorithm operates in worst case $O(\gamma^{-1})$ time which is constant for any fixed $\gamma > 0$. Further, it differs from standard cache algorithms as the number of stored elements varies between q, and $q \cdot (1+\gamma)$. However, for small γ the difference is negligible.

As LRFU is heuristic in nature, we cannot guarantee to achieve a certain hit ratio. Instead, we consider the following property of LRFU; let C_t be the LRFU cache at time t, and for $x \in C_t$ let t_x be the time at which it was admitted into the cache. LRFU associates x with the weight $\sum_{i \in [t_x, t] | id_i = x} c^i$ and guarantees that the $q-1$ elements with the smallest weight will not be evicted. Our q-MAX

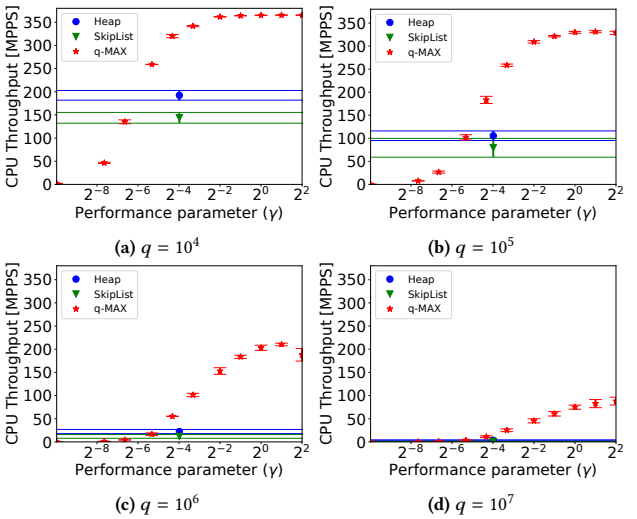


Figure 4: CPU Throughput of q-MAX as function of γ on a randomly generated stream of 150M numbers.

algorithm provides a similar guarantee: the top q elements (by weight) in the cache will not be evicted at any time.

6 EVALUATION

The evaluation of q-MAX is performed on a 16 CPU server running 64-bit Ubuntu-16.04.1 with 128GB RAM, 32KB L1 cache, 256KB L2 cache, and 25.6MB L3 cache. Our implementation is in C++ and is available as open source [3]. We also evaluate q-MAX for flows in a DPDK enabled OVS platform. To do so, we modify the datapath code of OVS, to record the source IP address, packet ID, and packet size of selected packets packet. We build one shared memory block for each PMD thread of OVS and copy the recorded information into the corresponding shared memory blocks. We also implement a user-space program that can read the packet information from shared memory blocks. Then we implement q-MAX and other algorithms to process those packet information.

We used four traces in the evaluation: (1) CAIDA'16: CAIDA Internet Traces from "Equinix-Chicago" in 2016 [5], (2) CAIDA'18: CAIDA Internet Traces from "Equinix-NewYork" in 2018 [6], (3) UNIV1: data center trace [19], and (4) P1-ARC: "P1.lis" Cache access trace [61], for evaluating caches. For evaluating our algorithms on the traces, we used the decimal representation of the IP source address of TCP and UDP packets as the key and total length field in the IP header as key. The evaluation considers the first 5 minutes of CAIDA'16 and CAIDA'18 traces and all of the UNIV1 trace. We evaluate a similar error range ($\approx 0.3\% - 3\%$) to previous works [7, 12, 49]. We ran each data point ten times, and we report the mean and 99% confidence intervals according to Student's t-test.

6.1 The Effect of γ on CPU Throughput

Intuitively, as we increase γ the q-MAX algorithm requires more space, but the maintenance operations are shorter. Figure 4 depicts CPU throughput in *Millions of Packets Per Second (MPPS)* as function of γ for various reservoir sizes (q) over a randomly generated streams of numbers. Indeed, γ has a significant impact on CPU

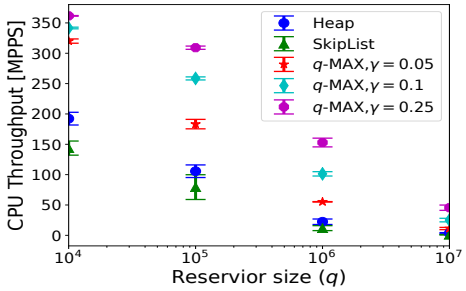
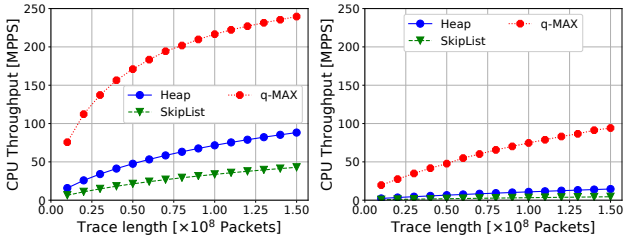


Figure 5: CPU Throughput for q-MAX, Heap and SkipList as function of q on a randomly generated stream of 150M numbers.



(a) $q = 10^5$ **(b)** $q = 10^6$

Figure 6: CPU Throughput of q-MAX ($\gamma = 0.1$) vs Heap and SkipList as function of the trace length with varying q.

γ	2.5%	5%	10%	25%	50%	100%	200%
Min Speedup vs. Heap	×0.73	×1.66	×1.77	×1.88	×1.89	×1.89	×1.89
Max Speedup vs. Heap	×1.34	×3.16	×7.11	×12.88	×17.16	×21.22	×23.39
Min Speedup vs. SkipList	×1.28	×2.22	×2.37	×2.51	×2.53	×2.53	×2.54
Max Speedup vs. SkipList	×4.01	×11.71	×26.28	×47.63	×63.45	×78.46	×86.48

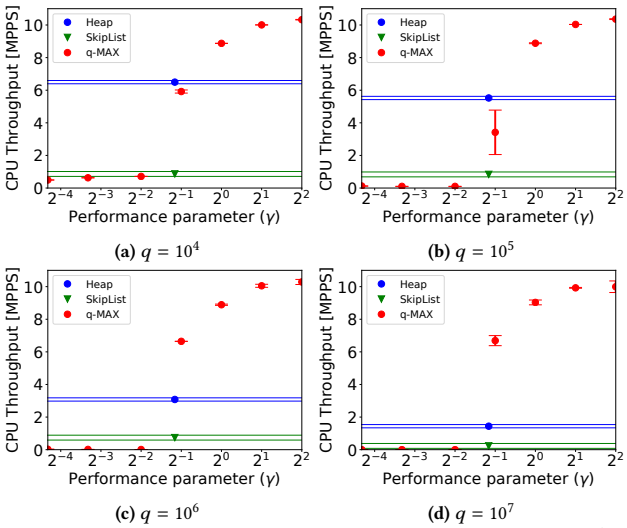
Table 1: Minimal and maximal speedups of q-MAX compared to Heap and SkipList for each value of γ.

throughput. For $q \leq 10^4$ the performance is dominated by the reservoir size q. However, as q get larger the solution becomes less cache resident which makes it slower for large γ values.

In order to be able to compare the results to the alternatives, we illustrate the update speed of the Heap and Skiplist algorithms (that have no γ parameter). As illustrated, q-MAX can be faster than the Heap and Skiplist alternatives. Notice that the break even point is around $\gamma = 0.025 = 2^{-5.3}$ for various q's. That is, we already achieve speedup when increasing the memory by as little as 2.5%, while 5% extra memory often doubles the throughout! Interestingly, q-MAX exposes an interesting trade-off, we can marginally increase the memory and benefit from asymptotic, and empirical speedup. Table 1 shows the range of improvement of q-MAX for a given γ.

6.2 CPU Throughput of q-MAX algorithm vs. Current Algorithms

Figure 5 compares the q-MAX algorithm to optimized versions of a Heap and a SkipList based algorithms. The Heap implementation is based on the standard C++ algorithm library (using C++11, g++ version 5.4.0, and STD library version 6.0.4). The SkipList implementation, as a linked-list of elements with pointers for the various levels, is based on [66] and [4]. As can be observed, for all values of q with $\gamma \geq 0.025$ the q-MAX is at least as fast as the other solutions. In some settings with only extra 5% memory, it achieves speedups



(a) $q = 10^4$ **(b)** $q = 10^5$ **(c)** $q = 10^6$ **(d)** $q = 10^7$

Figure 7: CPU Throughput of Exponential Decay q-MAX (for $c = 0.75$) as function of γ on a randomly generated stream of 150M numbers.

of up to ×3 and ×11 compared to Heap and SkipList accordingly. For $\gamma = 0.025$, q-MAX's performs more percentile calculations but is still comparable to other solutions.

Figure 6 shows the CPU throughput of q-MAX, Heap and SkipList throughout the trace. All implementations accelerate over the trace as new items are less likely to be included within the top q. However, q-MAX is considerably faster than the alternatives. Also notice, that increasing the reservoir size (q) makes all algorithms considerably slower due to cache locality issues.

6.3 Exponential Decay q-MAX

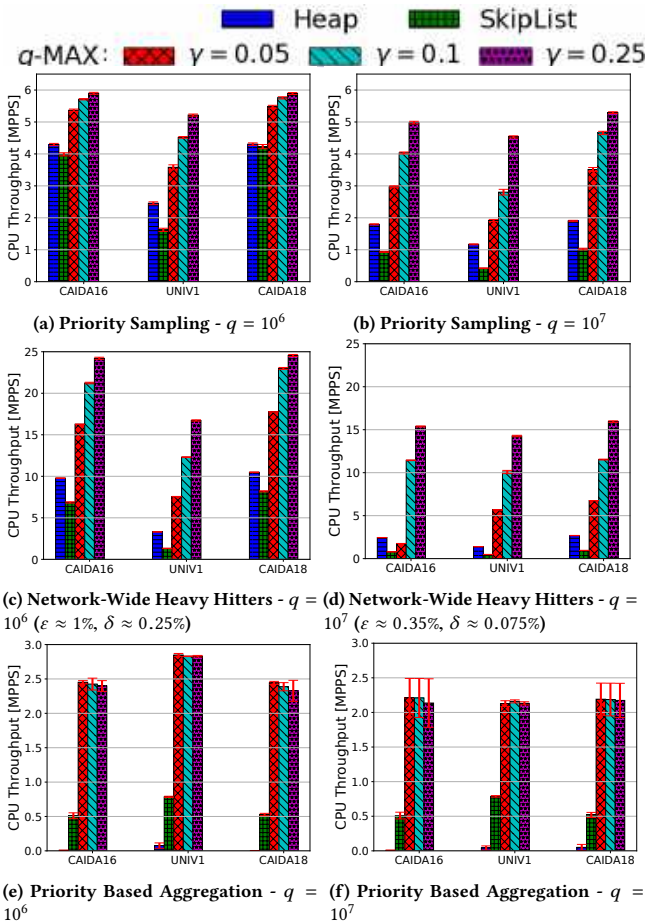
Figure 7 shows the throughput of our Exponential Decay q-MAX. As expected, increasing γ improves the throughput. In this case, the break even point happens than for larger γ than in (plain) q-MAX (see Figure 5). This is explained as the need to age counters, diminishes some of the returns from better maintenance of the reservoir.

6.4 Applications' CPU Throughput

Next, we implement Priority Sampling and Priority Based Aggregation using Heap, SkipList, and our q-MAX. In network-wide heavy hitters, we used the released open source. We emphasize that we use the exact same implementation for all alternatives and only replace the Heap/Skiplist with our q-MAX.

Subfigures 8a and 8b depict the throughput of Priority Sampling for three traces. Note that q-MAX increases space by $\approx 5\%$ to improve the throughput by up to ×1.84 when compared with Heap-based implementation and up to ×3.89 when compared with SkipList-based implementation.

Subfigures 8c and 8d depict the throughput of network-wide heavy hitters [18]. As a benchmark, we used the open-source code released by the paper's authors [2]. Note that as the measurement in this application is *routing oblivious*, the result of the computation only depends on the traffic *distribution* and not the network topology or routing. Notice that q-MAX, with $\gamma = 5\%$, improves



(a) **Priority Sampling - $q = 10^6$** (b) **Priority Sampling - $q = 10^7$**

(c) **Network-Wide Heavy Hitters - $q = 10^6$ ($\varepsilon \approx 1\%$, $\delta \approx 0.25\%$)** (d) **Network-Wide Heavy Hitters - $q = 10^7$ ($\varepsilon \approx 0.35\%$, $\delta \approx 0.075\%$)**

(e) **Priority Based Aggregation - $q = 10^6$** (f) **Priority Based Aggregation - $q = 10^7$**

Figure 8: CPU throughput of various applications when implemented using q-MAX, Heap and SkipList.

throughput by up to ×4 compared to the original (Heap) implementation, and by up to ×11.7 compared to a SkipList based implementation. Subfigures 8e and 8f depict the throughput of Priority Based Aggregation [38] in three real traces, when varying between Heap, SkipList and q-MAX implementations. Note that q-MAX implementations are faster and achieve a speedup of up to ×5.76 and ×875 compared to SkipList and Heap accordingly with $\gamma = 0.05$. Here, Heap is quite slow as the standard C++ library does not support value updates or sifts, which makes the update operation run in $O(q)$ time.

Figure 9 depicts the throughput of LRFU algorithm [55]. As can be observed, our q-MAX approach is up to ×4.13 faster than standard LRFU. For small caches ($q = 10^4$) we require a slightly large γ to outperform the benchmark. However, in large caches ($q = 10^5, 10^6$) we achieve over ×3.93 speedup even with $\gamma = 0.05$ which renders our algorithm practical. Also, here the Heap requires $O(q)$ time per update operation. Table 2 shows the hit ratio of q-MAX based LRFU algorithm compared to the original LRFU algorithm with q and $q(1+\gamma)$ entries. The hit ratio of q-MAX based LRFU cache is better than the original LRFU cache and slightly lower than a LRFU cache of size $q(1+\gamma)$. For a large enough cache ($q \geq 10^5$) all algorithms achieve a hit rate of 94.7%.

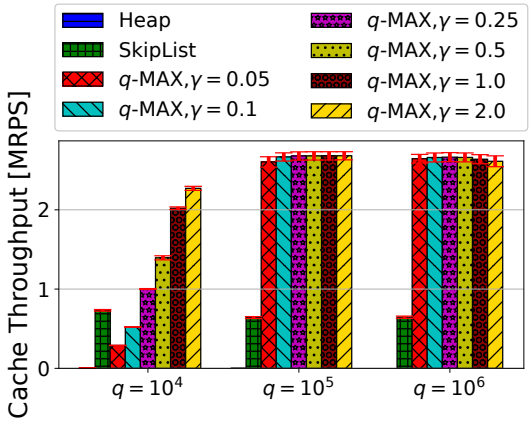


Figure 9: Throughput (Million Requests Per Second) of an LRFU cache ($c = 0.75$) which is implemented using q-MAX, Heap and SkipList on the P1 trace [61].

γ value	Algorithm	Hit Ratio
–	q-sized LRFU	51.6%
10%	q-MAX based LRFU	53.1%
	$q(1+\gamma)$-sized LRFU	54.6%
50%	q-MAX based LRFU	58.9%
	$q(1+\gamma)$-sized LRFU	64.4%
100%	q-MAX based LRFU	65.4%
	$q(1+\gamma)$-sized LRFU	73.3%

Table 2: The hit ratio of q-MAX-based LRFU compared to the original LRFU algorithm with $q = 10^4$, and $q(1+\gamma)$ entries on P1-ARC trace (for $c = 0.75$). Our algorithm operates in $O(1)$ time while LRFU in $O(\log q)$.

6.5 Sliding Windows

Figure 10 presents CPU throughput of the q-MAX and the sliding q-MAX algorithms, varying reservoir sizes q in multiple points throughout a 150M long trace of random numbers. As expected, for the q-MAX algorithm we see a gradual increase in the throughput as the trace progresses. This is due to the fact that the minimum bound of the q-maximal values gradually increases, causing more values to be ignored. In contrast, the throughput of the sliding q-MAX algorithm is constant throughout the trace, as the algorithm refers to a window of at most W numbers.

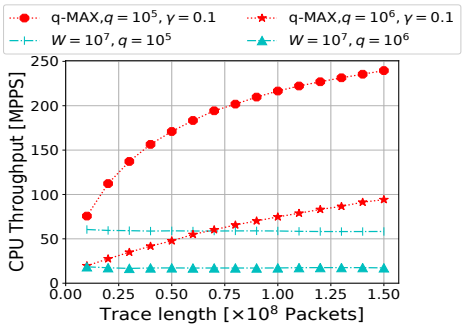


Figure 10: CPU Throughput of q-MAX vs. sliding windows q-MAX ($\gamma = 0.1$, $\tau = 1$) as function of the trace length with varying values of q.

In Figure 11 we evaluate the impact of the slack parameter (τ) on the throughput of the sliding q-MAX algorithm, for various γ and

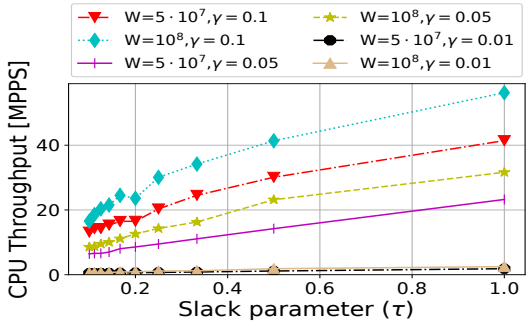
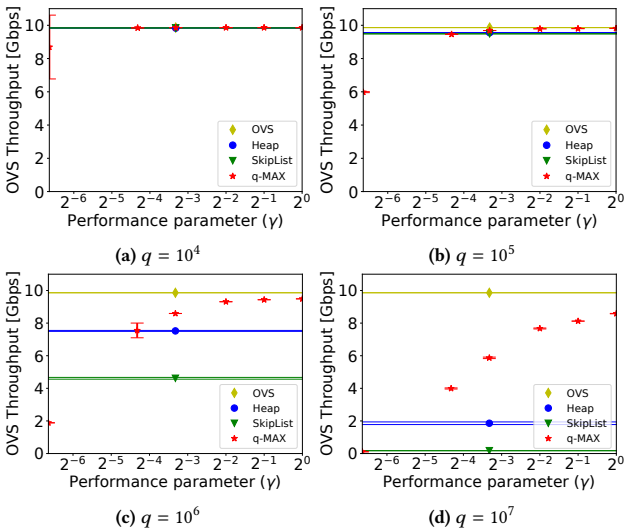


Figure 11: CPU throughput for sliding window q-MAX as function of τ for various values of W and γ on a random stream of 150M numbers and $q = 10^6$.

W values. Notice that: (i) increasing γ improves the throughput. (ii) large τ means higher throughput due to lower memory consumption. (iii) Larger W means higher throughput as fewer items are added to the sliding q-MAX.

6.6 Open vSwitch Integration

Next, we evaluate q-MAX and other algorithms on the Open vSwitch (OVS) platform. This evaluation shows that our solution is readily deployable in existing systems and that our solution could improve the measurement throughput by replacing the current data structures with, the faster,q-MAX. First, we run our evaluation with 10Gbps traffic, composed of minimal sized packets to stress-test our system. Figure 12 shows the OVS mean throughput (and the standard deviation as error bars) for Heap, SkipList, and q-MAX compared to the vanilla OVS that does not run additional algorithms. As can be observed, for $q = 10^4$ the Heap, and q-MAX do not restrict the OVS throughput, while the SkipList reduces the throughput. When increasing q further, the Heap gradually reduces the OVS throughput, while the q-MAX algorithms keep up with the OVS well until 10^7. Figure 13 provides similar results, and shows that q-MAX keeps up with the OVS even for small values of γ.



(a) $q = 10^4$ (b) $q = 10^5$ (c) $q = 10^6$ (d) $q = 10^7$

Figure 12: Throughput of q-MAX as function of γ on a randomly generated stream of 150M numbers.

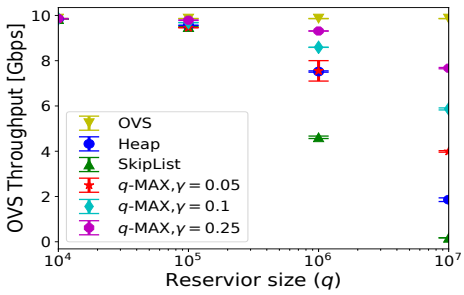
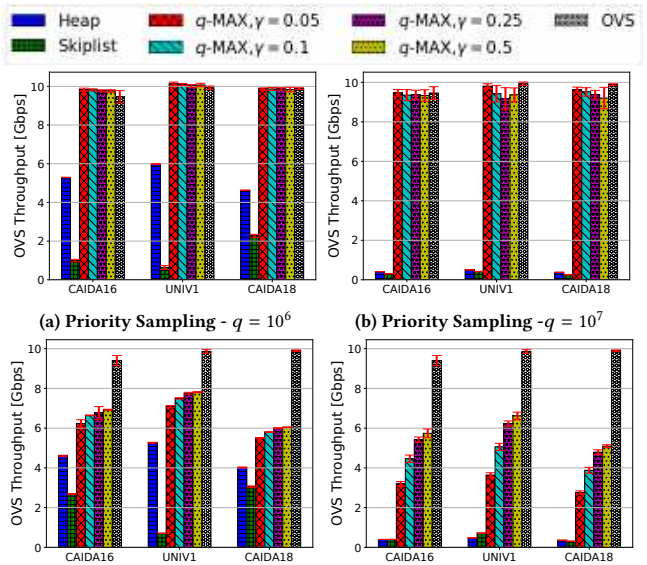


Figure 13: Throughput for q-MAX, Heap and SkipList as function of q on a randomly generated stream of 150M numbers.

Next, we run our evaluation with 10Gbps traffic, composed of real traffic traces, to evaluate the impact of adding q-MAX and other algorithms upon OVS in real measurement applications. Subfigure 14a and 14b show the achieved throughput for OVS without measurement, and OVS that performs Priority Sampling with different implementations. As can be observed, q-MAX implementations are faster and result in up to ×2.5 better throughput, the benefit is larger the more we increase q. Specifically, the overheads of performing Priority Sampling in OVS is only 6.1% with q-MAX compared to 60.1% with the best alternative.

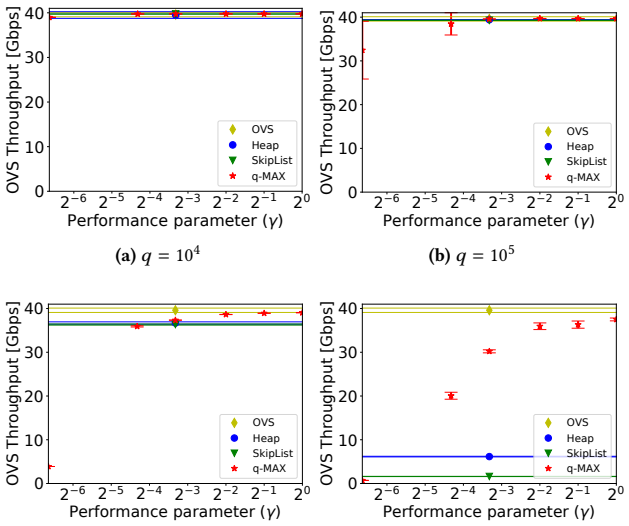


(a) Priority Sampling - $q = 10^6$ (b) Priority Sampling -$q = 10^7$

(c) Network-Wide Heavy Hitters - $q = 10^6$ ($\varepsilon \approx 1\%$, $\delta \approx 0.25\%$) (d) Network-Wide Heavy Hitters - $q = 10^7$ ($\varepsilon \approx 0.35\%$, $\delta \approx 0.075\%$)

Figure 14: The throughput of Priority Sampling and update throughput of Network-Wide Heavy Hitters using q-MAX, Heap and SkipList on 10G link.

Subfigures 14c and 14d show the throughput impact of running network-wide heavy hitters [18] on OVS. Note that (i) q-MAX implementations attain higher OVS throughput, (ii) q-MAX is especially needed for $q = 10^7$, where we achieve a throughput improvement of up to ×2.41 compared to the best alternative. $q = 10^7$ corresponds to dedicating about 100MB of space to measurement which is reasonable given today's architecture. Specifically, the overheads of performing network-wide heavy hitters in OVS is at most 5.0% with q-MAX compared to 41.6% with the best alternative.



(a) $q = 10^4$ (b) $q = 10^5$

(c) $q = 10^6$ (d) $q = 10^7$

Figure 15: Throughput of q-MAX as function of γ on a randomly generated stream of 150M numbers on a 40G link.

6.6.1 Experiments with 40Gbps Open vSwitch. In this appendix, we discuss the applicability of our results for 40G with real-sized packets. That is, unlike the 10G experiments, we are not using packets of minimal size but rather actual sized packets taken as the average packet size in the Univ1 trace. As depicted in figures 15 and 16, all algorithms can meet line rate for $q = 10^4$ and $q = 10^5$. However, for $q = 10^6$, Heap imposes a throughput degradation of nearly 15% while Skiplist reduces it by 41%. In contrast, q-MAX has a minimal impact of 2.9% with $\gamma = 0.25$. For $q = 10^7$, Heap and Skiplist fail completely and do not even get to 10Gbps. q-MAX, with double the space ($\gamma = 1$), achieves 36Gbps, less than 8% lower than OVS with no measurement. Finally, evaluate the performance of measurement applications in 40Gbps OVS deployment. As shown in Figure 17, q-MAX enables line-rate measurement for $q = 10^6$, and is the only solution able to achieve acceptable throughput for $q = 10^7$.

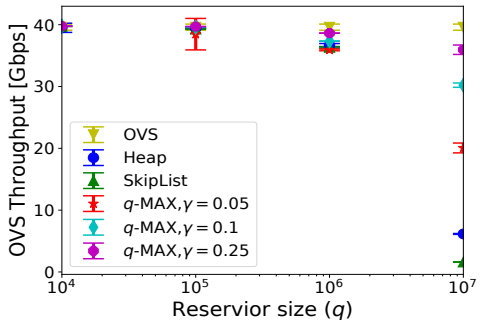
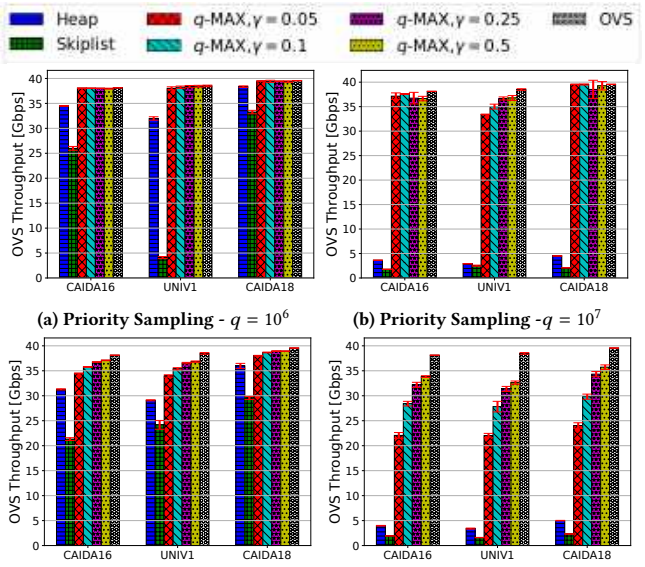


Figure 16: Throughput for q-MAX, Heap and SkipList as function of size (q) on a randomly generated stream of 150M numbers on a 40G link.



(a) Priority Sampling - $q = 10^6$ (b) Priority Sampling -$q = 10^7$

(c) Network-Wide Heavy Hitters - $q = 10^6$ ($\varepsilon \approx 1\%$, $\delta \approx 0.25\%$) (d) Network-Wide Heavy Hitters - $q = 10^7$ ($\varepsilon \approx 0.35\%$, $\delta \approx 0.075\%$)

Figure 17: The throughput of Priority Sampling and update throughput of Network-Wide Heavy Hitters using q-MAX, Heap and SkipList on 40G link.

7 DISCUSSION

Our work introduces constant time algorithms to the fundamental problem of maintaining the q largest numbers in a stream. Maintaining the q largest numbers occurs in numerous networking applications that use standard logarithmic data structures such as heaps, and skip lists. Thus, our work reduces the update complexity of measurement applications that relay on this pattern including (1) Network-wide heavy hitters [18]; (2) Priority Sampling [37]; (3) Priority-Based Aggregation [38]; (4) Bottom-k sketches [24]; and (5) the universal monitoring sketch [60]. Our extensive evaluation shows that q-MAX is faster than heaps and skiplists, and that using q-MAX offers a speedup for diverse network application, and facilitate line speed processing in Open vSwitch. Beyond network algorithms, our work also reduces the complexity of the well known LRFU algorithm to a constant [55].

We hope that application designers will consider our q-MAX solutions when applicable, and we released an open-source library to facilitate future use of our algorithms. As future work, we suggest exploring how a q-MAX like solution could be implemented on programmable switches. Implementing a heap on such switches is difficult and perhaps a q-MAX algorithm would enable other measurement algorithms.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Srikanth Sundaresan, for their thorough comments and feedback that helped improve the paper. This work was supported by the Zuckerman Foundation, the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel Cyber Directorate, the Cyber Security Research Center and the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

REFERENCES

[1] https://www.openvswitch.org/.
[2] Network-wide routing-oblivious heavy hitters - available: https://github.com/jalilm/tdhh.
[3] q-max: A unified scheme for improving network measurement throughput - available: https://github.com/jalilm/q-max.
[4] Redis - in-memory data structure store.
[5] The CAIDA UCSD Anonymized Internet Traces 2016 - January. 21st.
[6] The CAIDA UCSD Anonymized Internet Traces 2018 - equinix-nyc 2018-03-15, Direction A. https://www.caida.org/data/monitors/passive-equinix-nyc.xml.
[7] Anderson, D., Bevan, P., Lang, K., Liberty, E., Rhodes, L., and Thaler, J. A high-performance algorithm for identifying frequent items in data streams. In *ACM IMC* (2017).
[8] Andersson, A., Hagerup, T., Nilsson, S., and Raman, R. Sorting in linear time? *J. Computer and System Sciences* (1998).
[9] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. Snap: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM* (2016).
[10] Assaf, E., Ben-Basat, R., Einziger, G., and Friedman, R. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM* (2018).
[11] B Yossef, Z., Jayram, T., Kumar, R., Sivakumar, D., and Trevisan, L. Counting distinct elements in a data stream. In *RANDOM* (2002).
[12] Basat, R., Einziger, G., Friedman, R., Luizelli, M., and Waisbard, E. Constant time updates in hierarchical heavy hitters. In *ACM SIGCOMM* (2017).
[13] Basat, R. B., Chen, X., Einziger, G., Friedman, R., and Kassner, Y. Randomized admission policy for efficient top-k, frequency, and volume estimation. *IEEE/ACM Trans. Netw.* (2019).
[14] Basat, R. B., Einziger, G., and Friedman, R. Give Me Some Slack: Efficient Network Measurements. In *MFCS* (2018).
[15] Ben-Basat, R., Chen, X., Einziger, G., and Rottenstreich, O. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE ICNP* (2018).
[16] Ben-Basat, R., Einziger, G., Friedman, R., and Kassner, Y. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM* (2016).
[17] Ben-Basat, R., Einziger, G., Keslassy, I., Orda, A., Vargaftik, S., and Waisbard, E. Memento: making sliding windows efficient for heavy hitters. In *ACM CoNEXT* (2018).
[18] Ben Basat, R., Einziger, G., Moraney, J., and Raz, D. Network-wide routing oblivious heavy hitters. In *ACM/IEEE ANCS* (2018).
[19] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *ACM IMC* (2010).
[20] Benson, T., Anand, A., Akella, A., and Zhang, M. Microte: Fine grained traffic engineering for data centers. In *ACM CoNEXT* (2011).
[21] Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., and Tarjan, R. E. Time bounds for selection. *J. of Computer and System Sciences* (1973).
[22] Charikar, M., Chen, K., and Farach-Colton, M. Finding frequent items in data streams. In *ICALP* (2002).
[23] Chiesa, M., Rétvári, G., and Schapira, M. Lying your way to better traffic engineering. In *ACM CoNEXT* (2016).
[24] Cohen, E., and Kaplan, H. Summarizing data using bottom-k sketches. In *ACM PODC* (2007).
[25] Cohen, E., and Strauss, M. Maintaining time-decaying stream aggregates. In *ACM SIGMOD* (2003), pp. 223–233.
[26] Cormode, G., Korn, F., and Tirthapura, S. Exponentially decayed aggregates on data streams. In *IEEE ICDE* (April 2008).
[27] Cormode, G., and Muthukrishnan, S. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *In Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data* (2004).
[28] Cormode, G., and Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms 55* (2004).
[29] Cormode, G., and Muthukrishnan, S. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst.* (2005).
[30] Cormode, G., and Muthukrishnan, S. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst.* (2005).
[31] Cormode, G., and Muthukrishnan, S. What's hot and what's not: Tracking most frequent items dynamically. *ACM Trans. Database Syst. 30*, 1 (Mar. 2005).
[32] Datar, M., Gionis, A., Indyk, P., and Motwani, R. Maintaining stream statistics over sliding windows. *SIAM J. Comput.* (2002).
[33] Demaine, E. D., López-Ortiz, A., and Munro, J. I. Frequency estimation of internet packet streams with limited space. In *Proc. of the 10th Annual European Symposium on Algorithms* (2002), ESA, Springer-Verlag.
[34] Demianiuk, V., Gorinsky, S., Nikolenko, S. I., and Kogan, K. Robust Distributed Monitoring of Traffic Flows. In *IEEE ICNP* (2019).
[35] Dimitropoulos, X., Hurley, P., and Kind, A. Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. *SIGCOMM Comput. Commun. Rev. 38*, 1 (Jan. 2008).
[36] Dittmann, G., and Herkersdorf, A. Network processor load balancing for high-speed links. In *SPECTS* (2002).
[37] Duffield, N., Lund, C., and Thorup, M. Priority sampling for estimation of arbitrary subset sums. *J. ACM* (2007).
[38] Duffield, N., Xu, Y., Xia, L., Ahmed, N. K., and Yu, M. Stream aggregation through order sampling. In *ACM CIKM* (2017).
[39] Duffield, N. G., and Grossglauser, M. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions Networking* (2001).
[40] Estan, C., and Varghese, G. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev. 32*, 4 (Aug. 2002).
[41] Fusy, E., and Giroire, F. Estimating the number of active flows in a data stream over a sliding window. In *ANALCO* (2007).
[42] Garcia-Teodoro, P., DÃ¡az-Verdejo, J. E., MaciÃ¡-FernÃ¡ndez, G., and VÃ¡zquez, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security* (2009).
[43] Gu, Y., McCallum, A., and Towsley, D. Detecting anomalies in network traffic using maximum entropy estimation. In *ACM IMC* (2005).
[44] Gupta, A., Harrison, R., Pawar, A., Birkner, R., Canini, M., Feamster, N., Rexford, J., and Willinger, W. Sonata: Query-driven network telemetry. *ACM SIGCOMM* (2018).
[45] Han, Y. Deterministic sorting in o(nlog log n) time and linear space. In *ACM STOC* (2002).
[46] Han, Y., and Thorup, M. Integer sorting in o (n(log log n)) expected time and linear space. In *IEEE FOCS* (2002).
[47] Harrison, R., Cai, Q., Gupta, A., and Rexford, J. Network-wide heavy hitter detection with commodity switches. In *ACM SOSR* (2018).
[48] Harrison, R., Cai, Q., Gupta, A., and Rexford, J. Network-wide heavy hitter detection with commodity switches. In *SOSR* (2018).
[49] Huang, Q., Jin, X., Lee, P. P. C., Li, R., Tang, L., Chen, Y.-C., and Zhang, G. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM* (2017).
[50] Jung, J., Paxson, V., Berger, A. W., and Balakrishnan, H. Fast portscan detection using sequential hypothesis testing. In *IEEE S&P* (2004).
[51] Kabbani, A., Alizadeh, M., Yasuda, M., Pan, R., and Prabhakar, B. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *HOTI* (2010).
[52] Katta, N., Ghag, A., Hira, M., Keslassy, I., Bergman, A., Kim, C., and Rexford, J. Clove: Congestion-aware load-balancing at the virtual edge. In *ACM CoNEXT* (2017).
[53] Kranakis, E., Morin, P., and Tang, Y. Bounds for frequency estimation of packet streams. In *In SIROCCO* (2003).
[54] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C. B., Shi, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J. C., Roskind, J., Kulik, J., Westin, P. G., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., and Chang, W.-T. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM* (2017).
[55] Lee, D., Choi, J., Kim, J.-H., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. Lrfu: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. on Comp.* (2001).
[56] Li, Y., Miao, R., Kim, C., and Yu, M. Flowradar: A better netflow for data centers. In *USENIX NSDI* (2016).
[57] Li, Y., Miao, R., Kim, C., and Yu, M. Lossradar: Fast detection of lost packets in data center networks. In *ACM CoNEXT* (2016).
[58] Liu, Y., Chen, W., and Guan, Y. Near-optimal approximate membership query over time-decaying windows. In *IEEE INFOCOM* (2013).
[59] Liu, Z., Ben-Basat, R., Einziger, G., Kassner, Y., Braverman, V., Friedman, R., and Sekar, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM* (2019).
[60] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM* (2016).
[61] Megiddo, N., and Modha, D. S. Arc: A self-tuning, low overhead replacement cache. In *USENIX FAST* (2003).
[62] Metwally, A., Agrawal, D., and Abbadi, A. E. Efficient computation of frequent and top-k elements in data streams. In *IN ICDT* (2005).
[63] Mukherjee, B., Heberlein, L., and Levitt, K. Network intrusion detection. *Network, IEEE 8*, 3 (May 1994).
[64] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane. In *ACM SOSR* (2017).
[65] Uyeda, F., Foschini, L., Baker, F., Suri, S., and Varghese, G. Efficiently measuring bandwidth at all time scales. In *NSDI* (2011).
[66] Wang, D. Skiplist - ustcdane on github. https://github.com/ustcdane/skiplist.
[67] Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., and Uhlig, S. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM* (2018).

[68] Yu, M., Jose, L., and Miao, R. Software defined traffic measurement with opensketch. In *USENIX NSDI* (2013).
[69] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., and Zheng, H. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015).